Package: bnlearn (via r-universe)

October 28, 2026

Type Package

Title Bayesian Network Structure Learning, Parameter Learning and Inference

Version 5.2.1

Date 2026-07-17

Depends R (>= 4.4.0), methods

Suggests parallel, graph, Rgraphviz, igraph, lattice, gRbase, gRain (>= 1.3-3), Rmpfr, gmp, glmnet

Maintainer Marco Scutari <scutari@bnlearn.com>

Description Bayesian network structure learning, parameter learning and inference. This package implements constraint-based (PC, GS, IAMB, Inter-IAMB, MMPC, Hiton-PC, HPC), pairwise (ARACNE and Chow-Liu), score-based (Hill-Climbing, Tabu Search, DirectLiNGAM) and hybrid (MMHC, RSMAX2, H2PC) structure learning algorithms for discrete, Gaussian, conditional Gaussian and zero-inflated 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, interventions, counterfactuals, cross-validation, bootstrap and model averaging. Development snapshots with the latest bugfixes are available from <https://www.bnlearn.com/>.

URL https://www.bnlearn.com/

SystemRequirements USE_C17

License GPL (>= 2)

LazyData yes

NeedsCompilation yes

Author Marco Scutari [aut, cre], Tomi Silander [ctb]

Repository https://cran.r-universe.dev

Date/Publication 2026-07-17 17:34:28 UTC

RemoteUrl https://github.com/cran/bnlearn

RemoteRef HEAD

RemoteSha 7d44622a52c29c2d666a8661d048e2b5709a7944

Contents

bnlearn-package . . . 3
alarm . . . 5
alpha.star . . . 7
arc operations . . . 8
arc.strength . . . 10
asia . . . 13
BF . . . 15
bn class . . . 16
bn.boot . . . 18
bn.cv . . . 19
bn.fit . . . 22
bn.fit class . . . 28
bn.fit plots . . . 30
bn.fit utilities . . . 31
bn.kcv class . . . 34
bn.strength class . . . 35
causal discovery algorithms . . . 35
causal inference . . . 37
ci.test . . . 38
clgaussian.test . . . 40
compare . . . 41
configs . . . 43
constraint-based algorithms . . . 44
coronary . . . 46
cpdag . . . 47
cpquery . . . 48
data preprocessing . . . 51
dsep . . . 53
foreign files utilities . . . 54
gaussian.test . . . 55
gRain integration . . . 56
graph enumeration . . . 57
graph generation utilities . . . 58
graph integration . . . 60
graph utilities . . . 61
graphviz.chart . . . 63
graphviz.plot . . . 64
hailfinder . . . 66

hybrid algorithms . . . 69
igraph integration . . . 70
independence-tests . . . 71
information theoretic quantities . . . 73
insurance . . . 74
learning.test . . . 76
lizards . . . 77
lm integration . . . 78
local discovery algorithms . . . 79
marks . . . 80
misc utilities . . . 81
model string utilities . . . 84
multivariate normal distribution . . . 85
naive.bayes . . . 86
network-classifiers . . . 89
network-scores . . . 89
node operations . . . 92
node ordering utilities . . . 93
pcalg integration . . . 94
plot.bn . . . 95
plot.bn.strength . . . 96
predict and impute . . . 97
rbn . . . 99
scm class . . . 100
score . . . 102
score-based algorithms . . . 105
single-node local discovery . . . 107
strength.plot . . . 108
structural.em . . . 110
structure-learning . . . 112
test counter . . . 115
utilities for whitelists and blacklists . . . 115
whitelists-blacklists . . . 117

Index **118**

`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 (via approximate inference algorithms).

Details

bnlearn implements key algorithms covering all stages of Bayesian network modelling: data preprocessing, structure learning combining data and expert/prior knowledge, parameter learning, and inference (including causal inference via do-calculus). **bnlearn** aims to be a one-stop shop for Bayesian networks in R. It provides tools for discrete, Gaussian and conditional linear Gaussian, and zero-inflated count Bayesian networks on real-world data. It also supportsd incomplete data with missing values. Furthermore, the modular nature of **bnlearn** makes it easy to use it for simulation studies.

Implemented structure learning algorithms include:

- *Constraint-based algorithms*, which use conditional independence tests to learn conditional independence constraints from data. The constraints, in turn, are used to learn the structure of the Bayesian network under the assumption that conditional independence implies graphical separation (so, two variables that are independent cannot be connected by an arc).
- *Score-based algorithms*, which are general-purpose optimisation algorithms that rank network structures with respect to a goodness-of-fit score.
- *Hybrid algorithms* combine aspects of both constraint-based and score-based algorithms. They use conditional independence tests (to reduce the search space) and network scores (to find the optimal network in the reduced space) at the same time.

For more details about structure learning algorithms, see structure learning. Available conditional independence tests are described in independence tests and available network scores are described in network scores. Specialized algorithms for learning the structure of Bayesian network classifiers are described in network classifiers. All algorithms support the use of whitelists and blacklists to include or exclude arcs from the networks (see whitelists and blacklists). Many have parallel implementation built on the **parallel** package. Bayesian network scores also support graphical priors.

Parameter learning approaches include both frequentist and Bayesian estimators. Approximate inference is implemented particle filters approaches, such as likelihood weighting, and covers conditional probability queries, prediction and imputation. Exact inference is implemented for discrete and Gaussian networks.

Additional facilities include support for bootstrap and cross-validation; advanced plotting capabilities implemented on top of **Rgraphviz** and **lattice**; model averaging; random graphs and random samples generation; import/export functions to integrate **bnlearn** with software such as Hugin and GeNIe; an associated Bayesian network repository of golden-standard networks at `https://www.bnlearn.com/bnrepository/`.

Use `citation("bnlearn")` to find out how to cite **bnlearn** in publications and other materials; and visit `https://www.bnlearn.com/` for more examples and code from publications using **bnlearn**.

Author(s)

Marco Scutari
Istituto Dalle Molle di Studi sull'Intelligenza Artificiale (IDSIA)

Maintainer: Marco Scutari <scutari@bnlearn.com>

References

reference books:

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.

from the author:

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.

Scutari M (20107). "Bayesian Network Constraint-Based Structure Learning Algorithms: Parallel and Optimized Implementations in the bnlearn R Package." *Journal of Statistical Software*, **77**(2):1–20.

Examples

```
## the workflow of Bayesian network modelling in bnlearn:
# choose the data set to work on...
data(learning.test)
# ... choose an algorithm and learn the structure of the network from the data...
net = hc(learning.test)
# ... plot it...
## Not run: graphviz.plot(net)
# ... learn the parameters of the network...
bn = bn.fit(net, learning.test)
# ... explore the network with a classic barchart...
## Not run: graphviz.chart(bn)
# ... and perform inference to answer any question that interests you!
cpquery(bn, event = (A == "a"), evidence = (C == "a"))
```

---

alarm *ALARM monitoring system (synthetic) data set*

Description

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

Usage

```
data(alarm)
```

Format

The alarm data set contains the following 37 variables:

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

- ACO2 (*arterial CO2*): a three-level factor with levels LOW, NORMAL and HIGH.
- VALV (*pulmonary alveoli ventilation*): a four-level factor with levels ZERO, LOW, NORMAL and HIGH.
- VLNG (*lung ventilation*): a four-level factor with levels ZERO, LOW, NORMAL and HIGH.
- VTUB (*ventilation tube*): a four-level factor with levels ZERO, LOW, NORMAL and HIGH.
- VMCH (*ventilation machine*): a four-level factor with levels ZERO, LOW, NORMAL and HIGH.

Note

The complete BN can be downloaded from https://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." *Proceedings of the 2nd European Conference on Artificial Intelligence in Medicine*, 247–256.

Examples

```
# load the data.
data(alarm)
# create and plot the network structure.
modelstring = paste0("[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]")
dag = model2network(modelstring)
## Not run: graphviz.plot(dag)
```

---

alpha.star *Estimate the optimal imaginary sample size for BDe(u)*

Description

Estimate the optimal imaginary sample size for the BDe score, given a network structure and a data set, assuming a uniform prior.

Usage

```
alpha.star(x, data, 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.
debug	a boolean value. If TRUE, a lot of debugging output is printed. Otherwise the function is completely silent.

Value

alpha.star() returns a positive number, the estimated optimal imaginary sample size value.

Author(s)

Marco Scutari

References

Steck H (2008). "Learning the Bayesian Network Structure: Dirichlet Prior versus Data." *Proceedings of the 24th Conference on Uncertainty in Artificial Intelligence*, 511–518.

Examples

```
data(learning.test)
dag = hc(learning.test, score = "bic")

for (i in 1:3) {

  a = alpha.star(dag, learning.test)
  dag = hc(learning.test, score = "bde", iss = a)

}#FOR
```

---

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

Description

Drop, add or set the direction of a directed or undirected arc (that is, an edge).

Usage

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

# edge (that is, undirected arc) operations
set.edge(x, from, to, check.cycles = TRUE, check.illegal = 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.
check.cycles	a boolean value. If TRUE, the graph is tested for acyclicity; otherwise the graph is returned anyway.
check.illegal	a boolean value. If TRUE, arcs that break the parametric assumptions of x, such as those from continuous to discrete nodes in conditional Gaussian networks, cause an error.
debug	a boolean value. If TRUE, a lot of debugging output is printed; otherwise the function is completely silent.

Details

The set.arc() function operates in the following way:

- If there is no arc between from and to, the arc from → 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 an updated copy of x invisibly.

Author(s)

Marco Scutari

Examples

```
dag = cpdag(model2network("[A][C][F][B|A][D|A:C][E|B:F]"))
dag

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

---

arc.strength *Measure arc strength*

Description

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

Usage

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

# average arc strengths.
## S3 method for class 'bn.strength'
mean(x, ..., weights = NULL)

# averaged network structure.
```

```
averaged.network(strength, threshold)
# strength threshold for inclusion in the averaged network structure.
inclusion.threshold(strength)
```

Arguments

x	an object of class bn.strength (for mean()) or of class bn (for all other functions).
networks	a list, containing either objects of class bn or arc sets (matrices or data frames with two columns, optionally labelled "from" and "to"); or an object of class bn.kcv or bn.kcv.list from bn.cv().
data	a data frame containing the data the Bayesian network was learned from (for arc.strength()) or that will be used to compute the arc strengths (for boot.strength() and bf.strength()).
cluster	an optional cluster object from package **parallel**.
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.
criterion, score	a character string. For arc.strength(), the label of a score function or an independence test; see network scores for details. For bf.strength(), the label of the score used to compute the Bayes factors; see BF 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 arc strengths (in mean()) or network structures (in custom.strength()) to compute strength coefficients. If NULL, weights are assumed to be uniform.
cpdag	a boolean value. If TRUE, the (CPDAG of) the equivalence class is used instead of the network structure itself. It should make it easier to identify score-equivalent arcs.
shuffle	a boolean value. If TRUE, the columns in the data are permuted in each bootstrap sample to enforce the fact that the ordering of the variables in the data should be an invariant.
algorithm	a character string, the structure learning algorithm to be applied to the bootstrap replicates. See structure learning and the documentation of each algorithm for details.
algorithm.args	a list of extra arguments to be passed to the learning algorithm.
...	in arc.strength(), the additional tuning parameters for the network score (if criterion is the label of a score function, see score for details), the conditional independence test (currently the only one is B, the number of permutations). In mean(), additional objects of class bn.strength to average.
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 used to drop the arc from the network. The only two possible additional arguments are `alpha`, which sets the significance threshold that is used in `strength.plot()`; and `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 in which the arc is not present and the score of the network in which the arc is 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 arguments depending on choen score, see `score` for details. The significance threshold is set to `0`.

`boot.strength()` estimates the strength of each arc as its empirical frequency over a set of networks learned from bootstrap samples. It computes the probability of each arc (modulo its direction) and the probabilities of each arc's directions conditional on the arc being present in the graph (in either direction). The significance threshold is computed automatically from the strength estimates.

`bf.strength()` estimates the strength of each arc using Bayes factors to overcome the fact that Bayesian posterior scores are not normalised, and uses the latter to estimate the probabilities of all possible states of an arc given the rest of the network. The significance threshold is set to `1`.

`custom.strength()` takes a list of networks and estimates arc strength in the same way as `boot.strength()`.

Model averaging is supported for objects of class `bn.strength` returned by `boot.strength`, `custom.strength` and `bf.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()`, `custom.strength()`, `bf.strength()` and `mean()` 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." *Proceedings of the 15th Annual Conference on Uncertainty in Artificial Intelligence*, 196–201.

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

Scutari M, Nagarajan R (2013). "On Identifying Significant Edges in Graphical Models of Molecular Networks." *Artificial Intelligence in Medicine*, **57**(3):207–217.

See Also

strength.plot, score, ci.test.

Examples

```
data(learning.test)
dag = hc(learning.test)
arc.strength(dag, 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))

bf.strength(dag, learning.test, score = "bds", prior = "marginal")

## End(Not run)
```

---

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

Description

Synthetic data set on lung diseases and Asia travel from Lauritzen and Spiegelhalter (1988).

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 seem to behave better than their parametric counterparts.

The complete BN can be downloaded from https://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*, **50**(2):157–224.

Examples

```
# load the data.
data(asia)
# create and plot the network structure.
dag = model2network("[A][S][T|A][L|S][B|S][D|B:E][E|T:L][X|E]")
## Not run: graphviz.plot(dag)
```

BF *Bayes factor between two network structures*

Description

Compute the Bayes factor between the structures of two Bayesian networks.

Usage

```
BF(num, den, data, score, ..., log = TRUE)
```

Arguments

num, den	two objects of class bn, corresponding to the numerator and the denominator models in the Bayes factor.
data	a data frame containing the data to be used to compute the Bayes factor.
score	a character string, the label of a posterior network score, a BIC scores or "custom-score" for the custom score. If none is specified, the default score is the *Bayesian Dirichlet equivalent* score ("bde") for discrete networks and the *Bayesian Gaussian score* ("bge") for Gaussian networks. Other kinds of Bayesian networks are currently supported by using their BIC score to approximate their marginal log-likelihoods.
...	extra tuning arguments for the posterior scores. See score for details.
log	a boolean value. If TRUE, the Bayes factor is given as log(BF).

Value

A single numeric value, the Bayes factor of the two network structures num and den.

Note

The Bayes factor for two network structures, by definition, is the ratio of the respective marginal likelihoods. If we assume the uniform prior over all possible DAGs, it is equivalent to the ratio of the corresponding posterior probabilities. However, note that it is possible to specify different priors using the "..." arguments of BF(). In that case, the value returned by the function will not be the standard Bayes factor.

Author(s)

Marco Scutari

See Also

score, compare, bf.strength.

Examples

```
data(learning.test)

dag1 = model2network("[A][B][F][C|B][E|B][D|A:B:C]")
dag2 = model2network("[A][C][B|A][D|A][E|D][F|A:C:E]")
BF(dag1, dag2, learning.test, score = "bds", iss = 1)
```

---

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 is never changed afterwards.
  - whitelist: a copy of the whitelist argument (a two-column matrix, whose columns are labelled from and to) as transformed by sanitisation functions.
  - blacklist: a copy of the blacklist argument (a two-column matrix, whose columns are labelled from and to) as transformed by sanitisation functions.
  - 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 algorithms; the label of the network score used in the "Maximize" phase of hybrid algorithms; "none" for randomly generated graphs. For hybrid algorithms, test always has the same value as maxscore (see below).
  - 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.
    - k: a positive numeric value, the penalty coefficient used by the "aic", "aic-g", "aic-cg", "bic", "bic-g", "bic-cg", "pnal", "pnal-g" and "pnal-cg" 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.
    * prior: the graphical prior used in combination with a Bayesian score such as "bde" or "bge".
    * beta: the parameters of the graphical prior.
- 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 labelled 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 optimisations have been used in the learning algorithm (a boolean value).
- illegal: arcs that are illegal according to the parametric assumptions used to learn the network structure (a two-column matrix, whose columns are labelled from and to).
- 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).
- max.sx: the maximum allowed size of the conditioning sets in the conditional independence tests used in constraint-based algorithms.

An object of class bn may also have auxiliary class labels, depending on how it has been created. In particular:

- bn.naive, identifying naive Bayes classifiers from naive.bayes();
- bn.tan, identifying tree-augmented naive Bayes (TAN) classifiers from tree.bayes();
- bn.twin, identifying twin networks from twin();
- bn.ctf, identifying counterfactual networks from twin().

Author(s)

Marco Scutari

`bn.boot` *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), algorithm,
  algorithm.args = list(), statistic.args = list(), cluster,
  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.
`algorithm`	a character string, the learning algorithm to be applied to the bootstrap replicates. See `structure learning` and the documentation of each algorithm for details.
`algorithm.args`	a list of extra arguments to be passed to the learning algorithm.
`statistic.args`	a list of extra arguments to be passed to the function specified by `statistic`.
`cluster`	an optional cluster object from package **parallel**.
`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 `statistic.args` are extracted from the list and passed to `statistic` as subsequent 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." *Proceedings of the 15th Annual Conference on Uncertainty in Artificial Intelligence*, 196–201.

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, ..., algorithm.args = list(),
  loss.args = list(), fit, fit.args = list(), method = "k-fold",
  cluster, 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)

loss(x)
```

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 *Classification Error* for Bayesian network classifiers; otherwise, the *Log-Likelihood Loss* for both discrete and continuous data sets. See below for additional details.

algorithm.args	a list of extra arguments to be passed to the learning algorithm.
loss.args	a list of extra arguments to be passed to the loss function specified by loss.
fit	a character string, the label of the method used to fit the parameters of the network. See bn.fit for details.
fit.args	additional arguments for the parameter estimation procedure, see again bn.fit for details.
method	a character string, either "k-fold", "custom-folds" or "hold-out". See below for details.
cluster	an optional cluster object from package **parallel**.
debug	a boolean value. If TRUE, a lot of debugging output is printed. Otherwise the function is completely silent.
x	an object of class bn.kcv or bn.kcv.list returned by bn.cv().
...	additional objects of class bn.kcv or bn.kcv.list to plot alongside the first.
main, xlab, ylab	the title of the plot, an array of labels for the boxplot, the label for the y axis.
connect	a logical value. If TRUE, the median points in the boxplots will be connected by a segmented line.

Value

bn.cv() returns 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.

loss() returns a numeric vector with a length equal to runs.

Cross-Validation Strategies

The following cross-validation methods 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. 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.
- *custom-folds*: the data are manually partitioned by the user into subsets, which are then used as in k-fold cross-validation. Subsets are not constrained to have the same size, and every observation must be assigned to one subset.
- *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 - nrow(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 cross-validation is used with multiple runs, the overall loss is the average of the loss estimates from the different runs.

To clarify, cross-validation methods accept the following optional arguments:

- k: a positive integer number, the number of groups into which the data will be split (in k-fold cross-validation) or the number of times the data will be split in training and test samples (in hold-out cross-validation).

- `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 k-fold or hold-out cross-validation will be run.
- `folds`: a list in which each element corresponds to one fold and contains the indices for the observations that are included in that fold; or a list with an element for each run, in which each element is itself a list of the folds to be used for that run.

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. Lower values are better.
- *Classification Error* (`"pred"`): the *prediction error* for a single discrete node. Lower values are better.
- *Exact Classification Error* (`"pred-exact"`): closed-form exact posterior predictions are available for Bayesian network classifiers. Lower values are better.
- *Predictive Correlation* (`"cor"`): the *correlation* between the observed and the predicted values for a single continuous node. Higher values are better.
- *Mean Squared Error* (`"mse"`): the *mean squared error* between the observed and the predicted values for a single continuous node. Lower values are better.
- *F1 score* (`"f1"`): the *F1* score between observed and predicted values for both binary and multiclass target variables.
- *AUROC* (`"auroc"`): the *area under the ROC curve* for both binary and multiclass target variables. The multiclass AUROC score is computed one-vs-rest by averaging the AUROC for each level of the target variable.

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

- `predict`: a character string, the label of the method used to predict the observations in the test set. The default is `"parents"`. Other possible values are the same as in `predict()`.
- `predict.args`: a list containing the optional arguments for the prediction method. See the documentation for `predict()` for more details.
- `target`: a character string, the label of the target node for prediction in all loss functions but `"logl"`.

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 respective 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(predict = "bayes-lw", target = "F"))

folds = list(1:2000, 2001:3000, 3001:5000)
bn.cv(learning.test, 'hc', loss = "logl", method = "custom-folds",
  folds = folds)

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

## Not run:
# comparing algorithms with multiple runs of cross-validation.
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)

# use custom folds.
folds = split(sample(nrow(gaussian.subset)), seq(5))
bn.cv(gaussian.subset, "hc", method = "custom-folds", folds = folds)

# multiple runs, with custom folds.
folds = replicate(5, split(sample(nrow(gaussian.subset)), seq(5)),
          simplify = FALSE)
bn.cv(gaussian.subset, "hc", method = "custom-folds", folds = folds)

## End(Not run)
```

---

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

Description

Fit, assign or replace the parameters of a Bayesian network conditional on its structure.

Usage

```
bn.fit(x, data, cluster, method, ..., keep.fitted = TRUE,
  debug = FALSE)
custom.fit(x, dist, ordinal, debug = FALSE)
bn.net(x)
```

Arguments

x	an object of class bn (for bn.fit() and custom.fit()) or an object of class bn or bn.fit (for bn.net).
data	a data frame containing the variables in the model.
cluster	an optional cluster object from package **parallel**.
dist	a named list, with element for each node of x. See below.
method	a character string, see below for details.
...	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).
keep.fitted	a boolean value. If TRUE, the object returned by bn.fit will contain fitted values and residuals for all Gaussian and conditional Gaussian nodes, and the configurations of the discrete parents for conditional Gaussian nodes.
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.

bn.fit() accepts data with missing values encoded as NA. If the parameter estimation method was not specifically designed to deal with incomplete data, bn.fit() uses locally complete observations to fit the parameters of each local distribution.

Available methods for *discrete Bayesian networks* are:

- "mle": the maximum likelihood estimator for conditional probabilities.
- "bayes": the classic Bayesian posterior estimator with a uniform prior matching that in the Bayesian Dirichlet equivalent ("bde") score.
- "hdir": the hierarchical Dirichlet posterior estimator for related data sets from Azzimonti, Corani and Zaffalon (2019).
- "hard-em": the Expectation-Maximimation implementation of the estimators above.

Available methods for *Gaussian Bayesian networks* are:

- "mle-g": the maximum likelihood estimator for least squares regression models.
- "hard-em-g": the Expectation-Maximisation implementation of the estimators above.

Available methods for *conditional Gaussian Bayesian networks* are:

- "mle-cg": a combination of the maximum likelihood estimators "mle" and "mle-g".
- "hard-em-cg": the Expectation-Maximisation implementation of the estimators above.

Available methods for *zero-inflated Bayesian networks* are:

- "mle-zihp": the maximum likelihood estimator for zero-inflated hyper-Poisson regression models, computed by expectation-maximization with the generalised linear model components fitted by iteratively reweighted least squares.
- "mle-zinb": the maximum likelihood estimator for zero-inflated negative binomial regression models, computed by expectation-maximization with the generalised linear model components fitted by iteratively reweighted least squares.

Additional arguments for the bn.fit() function:

- iss: a numeric value, the imaginary sample size used by the "bayes" method to estimate the conditional probability tables associated with discrete nodes (see score for details).
- replace.unidentifiable: a boolean value. If TRUE and method is a maximum likelihood estimator, unidentifiable parameters are replaced by zeroes (in the case of regression coefficients and standard errors in Gaussian and conditional Gaussian nodes) or by uniform conditional probabilities (in discrete nodes).

  If FALSE (the default), the conditional probabilities in the local distributions of discrete nodes have a maximum likelihood estimate of NaN for all parent configurations that are not observed in data. Similarly, regression coefficients are set to NA if the linear regressions correspoinding to the local distributions of continuous nodes are singular. Such missing values propagate to the results of functions such as predict().
- alpha0: a positive number, the amount of information pooling between the related data sets in the "hdir" estimator.
- group: a character string, the label of the node with the grouping of the observations into the related data sets in the "hdir" estimator.
- impute and impute.args: a character string, the label of the imputation method (and its arguments) used by "hard-em", "hard-em-g" and "hard-em-cg" to complete the data in the *expectation* step. The default method is the same as for impute().
- fit and fit.args: a character string, the label of the parameter estimation method used by "hard-em", "hard-em-g" and "hard-em-cg" to estimate the parameters in the *maximisation* step. The default method is the same as for bn.fit().
- loglik.threshold: a non-negative numeric value, the minimum improvement threshold to continue iterating in "hard-em", "hard-em-g" and "hard-em-cg". The threshold is defined as the relative likelihood improvement divided by the sample size of data, and defaults to 1e-3. Setting it to zero means that iterations stop only when the improvement is negative, which can happen due to stochastic noise if the imputation of missing data uses approximate inference.
- params.threshold: a non-negative numeric value, the minimum maximum relative change in the parameter values to continue iterating in "hard-em", "hard-em-g" and "hard-em-cg". The threshold is defined as the maximum difference between parameter values, divided by the parameter value in the latest iteration. The default value is 1e-3.

- max.iter: a positive integer value, the maximum number of iterations in "hard-em", "hard-em-g" and "hard-em-cg". The default value is 5.
- start: a bn.fit object, the fitted network used to initialise the "hard-em", "hard-em-g" and "hard-em-cg" estimators. The default is to use the bn.fit object obtained from x with the default parameter estimator for the data, which will use locally complete data to fit the local distributions.
- newdata: a data frame, a separate set of data used to assess the convergence of the "hard-em", "hard-em-g" and "hard-em-cg" estimators. The data in data are used by default for this purpose.
- em.max.iter: a positive integer value, the maximum number of expectation-maximization iterations in the "mle-zihp" and "mle-zinb" estimators. The default value is 100.
- em.tol: a non-negative numeric value, the relative convergence tolerance (on both the log-likelihood and the parameters) of the expectation-maximization iteration in the "mle-zihp" and "mle-zinb" estimators. The default value is 1e-8.
- m.step: a character string, either "full" (the default) to re-fit each generalised linear model component to convergence in every maximisation step, or "one-step" to take a single iteratively reweighted least squares step per iteration (a cheaper generalized-EM update) in the "mle-zihp" and "mle-zinb" estimators.

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 conditional Gaussian 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 pensim 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.cgnode), 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 of the discrete parents of the conditional Gaussian node, stored as a factor.
- For a zero-inflated hyper-Poisson node (class bn.fit.zihpnode), the new parameters can be defined by a list with elements named inflation, intensity, dispersion and optionally fitted and resid.
- For a zero-inflated negative binomial node (class bn.fit.zinbnode), the new parameters can be defined by a list with elements named inflation, intensity, dispersion and optionally fitted and resid.

custom.fit() takes a set of user-specified distributions and their parameters and uses them to build a bn.fit object. Its purpose is to specify a Bayesian network (complete with parameters, not just the structure) using expert knowledge rather than 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()` and `custom.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 (that is, there are no undirected arcs). See `set.arc` and `cextend` for two ways of manually setting the direction of one or more arcs.

In the case of maximum likelihood estimators, `bn.fit()` may produce `NA` parameter estimates. For instance, for discrete and conditional Gaussian nodes, this happens when there are (discrete) parent configurations that are not observed in `data`. To avoid this, either set `replace.unidentifiable` to `TRUE` or, in the case of discrete networks, use `method = "bayes"`.

Author(s)

Marco Scutari

References

Azzimonti L, Corani G, Zaffalon M (2019). "Hierarchical Estimation of Parameters in Bayesian Networks." *Computational Statistics & Data Analysis*, **137**:67–91.

See Also

`bn.fit utilities`, `bn.fit plots`.

Examples

```
data(learning.test)

# learn the network structure.
cpdag = pc.stable(learning.test)
# set the direction of the only undirected arc, A - B.
dag = set.arc(cpdag, "A", "B")
# estimate the parameters of the Bayesian network.
fitted = bn.fit(dag, learning.test)
# replace the parameters of 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(dag, bn.net(fitted))

# learn the network structure.
dag = hc(gaussian.test)
# estimate the parameters of the Bayesian network.
fitted = bn.fit(dag, gaussian.test)
# replace the parameters of the node F.
```

```
fitted$F = list(coef = c(1, 2, 3, 4, 5), sd = 3)
# set the original parameters again.
fitted$F = lm(F ~ A + D + E + G, data = gaussian.test)

# discrete Bayesian network from expert knowledge.
dag = 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(dag, dist = list(A = cptA, B = cptB, C = cptC))
# for ordinal nodes, it is nearly the same.
cfit = custom.fit(dag, 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(dag, 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(dag, dist = list(A = cptA, B = distB, C = distC))

# zero-inflated hyper-Poisson nodes from expert knowledge.
dag = model2network("[A][B|A][C|A:B]")
ldistA = list(inflation = c("(Intercept)" = 0),
              intensity = c("(Intercept)" = 0),
              dispersion = 1)
ldistB = list(inflation = c("(Intercept)" = 0, A = 1),
              intensity = c("(Intercept)" = 0, A = 1),
              dispersion = 0.5)
ldistC = list(inflation = c("(Intercept)" = 0, A = 0.5, B = 0.5),
              intensity = c("(Intercept)" = 0, A = 0.4, B = 0.6),
              dispersion = 0.25)
zifit = custom.fit(dag, list(A = ldistA, B = ldistB, C = ldistC))

# zero-inflated negative binomial nodes from expert knowledge.
ldistA = list(inflation = c("(Intercept)" = 0),
              prsucc = c("(Intercept)" = 0),
              failures = 1)
ldistB = list(inflation = c("(Intercept)" = 0, A = 1),
              prsucc = c("(Intercept)" = 0, A = 1),
              failures = 0.5)
ldistC = list(inflation = c("(Intercept)" = 0, A = 0.5, B = 0.5),
              prsucc = c("(Intercept)" = 0, A = 0.4, B = 0.6),
```

```
              failures = 0.25)
zifit = custom.fit(dag, list(A = ldistA, B = ldistB, C = ldistC))
```

`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 network is discrete (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 (ordered factors) have class `bn.fit.onode` and contain the same elements as `bn.fit.dnode` nodes. Networks with 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 (that is, 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 (that is, the standard error).

Hybrid (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 (the standard error) for each configuration of the discrete parents.

Zero-Inflated (non-integer count) networks also have class bn.fit.zihp (hyper-Poisson) or bn.fit.zinb (negative binomial). Zero-inflated hyper-Poisson nodes have class bn.fit.zihpnode and contain 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.
- inflation: a numeric vector, the coefficients of the zero-inflation (logistic) regression model associated with the parents.
- intensity: a numeric vector, the coefficients of the (log-linear) regression model for the Poisson intensity parameter associated with the parents.
- dispersion: a numeric value, the dispersion parameter that augments the Poisson (pseudo)counts into a hyper-Poisson distribution.

Zero-inflated negative binomial nodes have class bn.fit.zinbnode and contain 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.
- inflation: a numeric vector, the coefficients of the zero-inflation (logistic) regression model associated with the parents.
- prsucc: a numeric vector, the coefficients of the (logistic) regression model for the probability of success parameter associated with the parents.
- failures: a numeric value, the number of failures.

In addition, Bayesian network classifiers store the label of the training node in an attribute named training.

An object of class bn.fit may also have auxiliary class labels, depending on how it has been created. In particular:

- bn.naive, identifying naive Bayes classifiers from naive.bayes();
- bn.tan, identifying tree-augmented naive Bayes (TAN) classifiers from tree.bayes();
- bn.twin, identifying twin networks from twin();
- bn.ctf, identifying counterfactual networks from twin().

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, ...)
bn.fit.histogram(fitted, density = TRUE, xlab = "Residuals",
  ylab = ifelse(density, "Density", ""), main, ...)
bn.fit.xyplot(fitted, xlab = "Fitted values", ylab = "Residuals", main, ...)
## for discrete (multinomial and ordinal) Bayesian networks.
bn.fit.barchart(fitted, xlab = "Probabilities", ylab = "Levels", main, ...)
bn.fit.dotplot(fitted, xlab = "Probabilities", ylab = "Levels", main, ...)
```

Arguments

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

Details

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

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

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

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

Value

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

Author(s)

Marco Scutari

See Also

bn.fit, bn.fit class.

---

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

Description

Assign, extract or compute various quantities of interest from objects 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'
logLik(object, data, nodes, by.sample = FALSE, na.rm = FALSE, debug = FALSE, ...)
## S3 method for class 'bn.fit'
AIC(object, data, ..., k = 1)
## S3 method for class 'bn.fit'
BIC(object, data, ...)

## non-method functions for "bn.fit"
identifiable(x, by.node = FALSE)
singular(x, by.node = FALSE)

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

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

## 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, for.parents, ...)
## S3 method for class 'bn.fit.cgnode'
residuals(object,  ...)
## S3 method for class 'bn.fit.cgnode'
sigma(object, for.parents, ...)
```

Arguments

object	an object of class bn.fit, bn.fit.dnode, bn.fit.gnode, bn.fit.cgnode or bn.fit.onode.
x	an object of class bn.fit.
nodes	a vector of character strings, the labels of the nodes whose log-likelihood components are to be computed.
data	a data frame containing the variables in the model.
...	additional arguments, currently ignored.
k	a numeric value, the penalty coefficient 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 log-likelihood of each observation in the sample. If FALSE, logLik() returns a single value, the log-likelihood of the whole sample.
by.node	a boolean value. If TRUE, identifiable() and singular() return a vector containing one value for each local distribution. If FALSE, identifiable() and singular() return a single value for the whole model.
na.rm	a boolean value, whether missing values should be used in computing the log-likelihood. See below for details. The default value is FALSE, and it only has an effect if by.sample = FALSE.
debug	a boolean value. If TRUE, a lot of debugging output is printed. Otherwise the function is completely silent.
for.parents	a named list in which each element contains a set of values for the discrete parents of the nodes. coef() and sigma() will only return the parameters associated with those parent configurations. (Only relevant for conditional Gaussian nodes.)

Details

coef() (and its alias coefficients()) extracts model coefficients (that is, conditional probabilities for discrete nodes; linear regression coefficients for Gaussian and conditional Gaussian nodes; regression coefficients of the count and zero-inflation components for zero-inflated nodes).

residuals() (and its alias resid()) extracts model residuals, and fitted() (and its alias fitted.values()) extracts fitted values from nodes other than discrete ones. 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.

logLik() returns the log-likelihood for the observations in data. If na.rm is set to TRUE, the log-likelihood will be NA if the data contain missing values. If na.rm is set to FALSE, missing values will be dropped, and the log-likelihood will be computed using only locally-complete observations (effectively returning the node-average log-likelihood times the sample size). Note that the log-likelihood may be NA even if na.rm = TRUE if the network contains NA parameters or is singular.

The for.parents argument in the methods for coef() and sigma() can be used to have both functions return the parameters associated with a specific configuration of the discrete parents of a node. If for.parents is not specified, all relevant parameters are returned.

Value

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.cgnode, bn.fit.onode, bn.fit.zihpnode, bn.fit.zinbnode.).

Author(s)

Marco Scutari

See Also

bn.fit, bn.fit-class.

Examples

```
data(gaussian.test)
dag = hc(gaussian.test)
fitted = bn.fit(dag, gaussian.test)
coefficients(fitted)
coefficients(fitted$C)
str(residuals(fitted))

data(learning.test)
dag2 = hc(learning.test)
fitted2 = bn.fit(dag2, learning.test)
coefficients(fitted2$E)
coefficients(fitted2$E, for.parents = list(F = "a", B = "b"))
```

`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.
- `learning`: the `learning` element of the `bn` object that was used for parameter learning from the training set (either learned from the training set as well or specified by the user).
- `loss`: the value of the loss function (for `method = "hold-out"` or `loss = "logl"`), or `NA` (otherwise).

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

- `predicted`: a factor or a numeric vector, the predicted values for the target node for the observations in the test set.
- `observed`: a factor or a numeric vector, the observed values for the target node for the observations 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 across all `k` iterations of the cross-validation, which is printed as the "expected loss" or averaged to compute the "average loss over the runs".
- `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 and 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:

- nodes: a vector of character strings, the labels of the nodes of the network(s) the strength was computed from.
- method: a character string, the method used to compute the strength coefficients. It can be equal to "test", "score" or "bootstrap".
- threshold: a numeric value, the threshold for determining whether a strength coefficient is significant.

An optional column called direction may also be present, giving the probability of the arc's irection 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

causal discovery algorithms *Causal discovery algorithms*

Description

Learn the structure of a (causal) Bayesian network with DirectLiNGAM.

Usage

```
direct.lingam(x, cluster, whitelist, blacklist, mi, maximize = "alasso",
  maximize.args = list(), debug = FALSE)
```

Arguments

x	a data frame containing the variables in the model.
cluster	an optional cluster object from package **parallel**.
whitelist	a data frame with two columns (optionally labelled "from" and "to"), containing a set of arcs to be included in the graph.
blacklist	a data frame with two columns (optionally labelled "from" and "to"), containing a set of arcs not to be included in the graph.
mi	a character string, the label of the mutual information measure used to identify the topological ordering of the nodes. It can take values "pwling" (the default) or "gkernel".
maximize	a character value, the label of the method used to identify the parents of each node given the causal ordering. The only possible value is "alasso" (adaptive LASSO).
maximize.args	a list of arguments to be passed to the method specified by maximize. See below for details.
debug	a boolean value. If TRUE, a lot of debugging output is printed. Otherwise, the function is completely silent.

Details

The "alasso" method has the following optional arguments:

- gamma: the coefficient of the weights used in the adaptive LASSO. The default value is 1.
- lambda.min.ratio and pmax: identical to the arguments of the same names in glmnet(), with the same default values and interpretation.
- k: the coefficient of the penalised likelihood score used to choose the optimal shrinkage in both ridge and LASSO models in the adaptive LASSO. It defaults to the BIC penalty.

Value

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

Note

See structure learning for a complete list of structure learning algorithms with the respective references.

Author(s)

Marco Scutari

See Also

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

`causal inference` *Perform causal inference*

Description

Perform interventions and counterfactuals, using Bayesian networks as causal models.

Usage

```
# network transformations used in causal inference.
mutilated(x, evidence)
twin(x)

# causal inference types.
intervention(x, evidence)
counterfactual(x, evidence, merging = TRUE)
```

Arguments

`x`	an object of class `bn`, `bn.fit` or `scm`.
`evidence`	a list, in the same format as the evidence for likelihood weighting (see `cpquery`).
`merging`	a boolean value. If `TRUE`, the twin network is reduced by node merging after introducing the counterfactual.

Details

`mutilated()` constructs the mutilated network arising from an ideal intervention setting the nodes involved to the values specified by `evidence`.

`twin()` constructs the twin network required to compute counterfactuals, separating noise (exogenous) from functional (factual) nodes and adding counterfactual nodes.

`intervention()` is an alias of `mutilated()`.

`counterfactual()` constructs the twin network and introduces a counterfactual (that is, an intervention on the counterfactual nodes) before applying node merging (optional). The intervention in `evidence` is specified using the labels of the nodes in `x`, and is then applied to the corresponding counterfactual nodes.

Both `twin()` and `counterfactual()` label nodes as follows:

- A counterfactual node has the label of the corresponding factual node with a period appended (as in `paste0(label, ".")`).
- A noise node has the label of the corresponding factual node with a `"u"` prepended (as in `paste0("u", label)`).

Evidence in `intervention()` can reference factual and counterfactual nodes. Evidence in `counterfactual()` can only reference counterfactual nodes; factual nodes will be replaced by the corresponding counterfactual nodes with a warning. Referencing exogenous nodes in either function will produce an error.

Value

A bn, bn.fit or scm object, depending on the class of x.

Objects returned by twin() will additionally have class bn.twin (if x is a bn or bn.fit object) or scm.twin (if x is an scm object).

Objects returned by counterfactual() will additionally have classes bn.twin and bn.ctf (if x is a bn or bn.fit object) or scm.twin (if x is an scm object).

Author(s)

Marco Scutari

References

Balke A, Pearl J (2022). "Probabilistic Evaluation of Counterfactual Queries." *Probabilistic and Causal Inference: The Works of Judea Pearl*, 237–254. The original is in the Proceedings of AAAI, 1994.

Graham D, Lee CM, Perov Y (2019). "Copy, Paste, Infer: A Robust Analysis of Twin Networks for Counterfactual Inference." *NeurIPS19 CausalML workshop*.

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

Shpitser I, PearlJ (2007). "What counterfactuals can be tested." *Proceedings of the 23rd Conference on Uncertainty in Artificial Intelligence (UAI)*, 352–359.

Examples

```
# example from Graham, Lee and Perov.
dag = model2network("[C][A|C][B|C][D|A:B]")
intervention(dag, evidence = list(A = 0))
twin(dag)
counterfactual(dag, evidence = list(A. = 0), merging = FALSE)
counterfactual(dag, evidence = list(A. = 0), merging = TRUE)
```

---

ci.test *Independence and conditional independence tests*

Description

Perform an independence or a conditional independence test.

Usage

```
ci.test(x, y, z, data, test, ..., 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 *mutual information* for categorical variables, the Jonckheere-Terpstra test for ordered factors and the *linear correlation* for continuous variables. See independence tests for details.
...	optional arguments to be passed to the test specified by test. See below for details.
debug	a boolean value. If TRUE, a lot of debugging output is printed. Otherwise, the function is completely silent.

Details

Additional arguments of the ci.test() function:

- B: a positive integer, the number of permutations used to compute the p-value of permutation tests. The default value is 5000 for nonparametric permutation tests and 100 for semiparametric permutation tests.
- fun: the function that computes the conditional independence test in the "custom-test" test. fun must have arguments x, y, z, data and args, in this order; in other words, it must have signature function(x, y, z, data, args). x and y will be the labels of the nodes to test for independence (one character string each); z will be the labels of the nodes in the conditioning set (a vector of character strings, possibily NULL for empty sets); data will contain the complete data set, with all the variables (a data frame); and args will be a list containing any additional arguments to the test.
- args: a list containing the optional arguments to fun, for tuning "custom-test" test functions.

Value

An object of class htest containing the following components:

statistic	the value of 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. Semiparametric tests have both.
p.value	the p-value for the test.

method	a character string indicating the type of test performed, and whether Monte Carlo simulation or continuity correction was used.
data.name	a character string giving the name(s) of the data.
null.value	the value of the test statistic under the null hypothesis, always 0.
alternative	a character string describing the alternative hypothesis.

Author(s)

Marco Scutari

See Also

independence tests, 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 *Synthetic (mixed) data set to test learning algorithms*

Description

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

Usage

```
data(clgaussian.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 available from https://www.bnlearn.com/documentation/networks/.

Examples

```
# load the data.
data(clgaussian.test)
# create and plot the network structure.
dag = model2network("[A][B][C][H][D|A:H][F|B:C][E|B:D][G|A:D:E:F]")
## Not run: graphviz.plot(dag)
```

---

compare *Compare two or more different Bayesian networks*

Description

Compare two different Bayesian networks; compute their Structural Hamming Distance (SHD) or the Hamming distance between their skeletons. Or graphically compare them by plotting them side by side,

Usage

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

sid(learned, true, debug = FALSE)
shd(learned, true, wlbl = FALSE, cpdag = TRUE, debug = FALSE)
hamming(learned, true, debug = FALSE)

graphviz.compare(x, ..., groups, layout = "dot", shape = "rectangle",
  fontsize = 12, main = NULL, sub = NULL, diff = "from-first",
  diff.args = list())
```

Arguments

target, learned	an object of class bn.
current, true	another object of class bn.
...	extra arguments from the generic method (for all.equal(), currently ignored); or a set of one or more objects of class bn (for graphviz.compare).
wlbl	a boolean value. If TRUE, arcs whose directions have been fixed by a whitelist or a blacklist are preserved when constructing the CPDAGs of learned and true.
cpdag	a boolean value. If TRUE, the (CPDAG of) the equivalence class is used instead of the network structures themselves, which is more appropriate if either was learned from data.
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.
x	an object of class bn.

groups	a list of character vectors, representing groups of node labels of nodes that should be plotted close to each other.
layout	a character string, the layout argument that will be passed to **Rgraphviz**. Possible values are "dot", "neato", "twopi", "circo" and "fdp". See **Rgraphviz** documentation for details.
shape	a character string, the shape of the nodes. Can be "circle", "ellipse" or "rectangle".
fontsize	a positive number, the font size for the node labels.
main	a vector of character strings, one for each network. They are plotted at the top of the corresponding figure(s).
sub	a vector of character strings, the subtitles that are plotted at the bottom of the corresponding figure(s).
diff	a character string, the label of the method used to compare and format the figure(s) created by graphviz.compare(). The default value is "from-first", see below for details.
diff.args	a list of optional arguments to control the formatting of the figure(s) created by graphviz.compare(). See below for details.

Details

graphviz.compare() can visualise differences between graphs in various ways depending on the value of the diff and diff.args arguments:

- "none": differences are not highlighted.
- "from-first": the first bn object, x, is taken as the reference network. All the other networks passed via the ... argument are compared to that first network and their true positive, false positive and false negative arcs are highlighted. Colours, line types and line widths for each category of arcs can be specified as the elements of a list via the diff.args argument, with names tp.col, tp.lty, tp.lwd, fp.col, fp.lty, fp.lwd, fn.col, fn.lty, fn.lwd. In addition, it is possible not to plot the reference network at all by setting show.first to FALSE.

Regardless of the visualisation, the nodes are arranged in the same positions across all networks to make comparison easier.

shd() computes the Structural Hamming Distance (SHD). 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.

sid() computes the Structural Interventional Distance (SID) between two DAGs, between a CPDAG and a DAG, between a DAG and a CPDAG, or between two CPDAGs.

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.

`sid()` returns a vector of non-negative integer numbers, with additional attributes describing the distribution over all the evaluated graphs.

`graphviz.compare()` plots one or more figures and invisibly returns a list containing the `graph` objects generated from the networks that were passed as arguments (in the same order). They can be further modified using the **graph** and **Rgraphviz** packages. Unlike the other functions, `graphviz.compare()` accepts networks with different node sets as long as they overlap.

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.

Peters J, Buhlmann P (2015). "Structural Intervention Distance for Evaluating Causal Graphs." *Neural Computation*, **27**(3):771–779.

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)
sid(e2, e1, debug = TRUE)
unlist(compare(e1,e2))
compare(target = e1, current = e2, arcs = TRUE)
## Not run: graphviz.compare(e1, e2, diff = "none")
```

---

configs *Construct configurations of discrete variables*

Description

Create configurations of discrete variables, which are 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 configurations are included as levels in the return value. Otherwise, only configurations that are actually observed are considered.

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 PC, Grow-Shrink (GS), Incremental Association (IAMB), Fast Incremental Association (Fast-IAMB), Interleaved Incremental Association (Inter-IAMB), Incremental Association with FDR (IAMB-FDR), Max-Min Parents and Children (MMPC), Semi-Interleaved HITON-PC or Hybrid Parents and Children (HPC) constraint-based algorithms.

Usage

```
pc.stable(x, cluster, whitelist, blacklist, test = NULL,
  alpha = 0.05, ..., max.sx = NULL, debug = FALSE, undirected = FALSE)
gs(x, cluster, whitelist, blacklist, test = NULL,
  alpha = 0.05, ..., max.sx = NULL, debug = FALSE, undirected = FALSE)
iamb(x, cluster, whitelist, blacklist, test = NULL,
  alpha = 0.05, ..., max.sx = NULL, debug = FALSE, undirected = FALSE)
fast.iamb(x, cluster, whitelist, blacklist, test = NULL,
  alpha = 0.05, ..., max.sx = NULL, debug = FALSE, undirected = FALSE)
inter.iamb(x, cluster, whitelist, blacklist, test = NULL,
  alpha = 0.05, ..., max.sx = NULL, debug = FALSE, undirected = FALSE)
iamb.fdr(x, cluster, whitelist, blacklist, test = NULL,
  alpha = 0.05, ..., max.sx = NULL, debug = FALSE, undirected = FALSE)
mmpc(x, cluster, whitelist, blacklist, test = NULL,
  alpha = 0.05, ..., max.sx = NULL, debug = FALSE, undirected = TRUE)
si.hiton.pc(x, cluster, whitelist, blacklist, test = NULL,
  alpha = 0.05, ..., max.sx = NULL, debug = FALSE, undirected = TRUE)
hpc(x, cluster, whitelist, blacklist, test = NULL,
  alpha = 0.05, ..., max.sx = NULL, debug = FALSE, undirected = TRUE)
```

Arguments

x	a data frame containing the variables in the model.
cluster	an optional cluster object from package **parallel**.
whitelist	a data frame with two columns (optionally labelled "from" and "to"), containing a set of arcs to be included in the graph.
blacklist	a data frame with two columns (optionally labelled "from" and "to"), containing a set of arcs not to be included in the graph.
test	a character string, the label of the conditional independence test to be used in the algorithm. If none is specified, the default test statistic is the *mutual information* for categorical variables, the Jonckheere-Terpstra test for ordered factors and the *linear correlation* for continuous variables. See independence tests for details.
alpha	a numeric value, the target nominal type I error rate.
...	optional arguments to be passed to the test specified by test. See ci.test for details.
max.sx	a positive integer, the maximum allowed size of the conditioning sets used in conditional independence tests. The default is that there is no size limit.
debug	a boolean value. If TRUE, a lot of debugging output is printed. Otherwise, the function is completely silent.
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.

Note

Note that even when undirected is set to FALSE, there is no guarantee that all arcs in the returned network will be directed; some arc directions are impossible to learn just from data due to score equivalence. cextend() provides a consistent extension of partially directed networks into directed acyclic graphs, which can then be used (for instance) for parameter learning.

See structure learning for a complete list of structure learning algorithms with the respective references. All algorithms accept incomplete data and handle it by computing individual conditional independence tests on locally complete observations.

Author(s)

Marco Scutari

See Also

independence tests, local discovery algorithms, score-based algorithms, hybrid algorithms, causal discovery algorithms, cextend.

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 in 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, check.cycles = FALSE) = 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")))
## Not run: graphviz.plot(ug, shape = "ellipse")
```

---

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, wlbl = FALSE, debug = FALSE)
cextend(x, strict = TRUE, debug = FALSE)
moral(x, debug = FALSE)

cextend.all(x, debug = FALSE)

colliders(x, arcs = FALSE, debug = FALSE)
shielded.colliders(x, arcs = FALSE, debug = FALSE)
unshielded.colliders(x, arcs = FALSE, debug = FALSE)
vstructs(x, arcs = FALSE, debug = FALSE)
```

Arguments

x	an object of class bn or bn.fit (with the exception of cextend(), which only accepts objects 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.
wlbl	a boolean value. If TRUE, arcs whose directions have been fixed by a whitelist or a blacklist are preserved when constructing the CPDAG.
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

Note that arcs whose directions are dictated by the parametric assumptions of the network are preserved as directed arcs in cpdag(). This means that, in a conditional Gaussian network, arcs from discrete nodes to continuous nodes will be treated as whitelisted in their only valid direction.

cextend.all() returns all possible consistent extensions of a CPDAG, whereas cextend() returns only one. cextend.all() also accepts a PDAG or an MPDAG (a partially directed graph carrying background knowledge, in which some directed arcs are not compelled), in which case it enumerates the consistent extensions that agree with the given arc directions; it stops with an error if no such extension exists.

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`.

`cextend.all()` returns an object of class `bn` or a list of objects of class `bn`.

`vstructs()`, `colliders()`, `shielded.colliders()` and `unshielded.colliders()` return a matrix with either 2 or 3 columns, according to the value of the `arcs` argument.

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.

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.

Andersson SA, Madigan D, Perlman MD (1997). "A Characterization of Markov Equivalence Classes for Acyclic Digraphs." *The Annals of Statistics*, **25**(2):505–541.

Wienobst M, Luttermann M, Bannach M, Liskiewicz M (2023). "Efficient Enumeration of Markov Equivalent DAGs." *Proceedings of the Thirty-Seventh AAAI Conference on Artificial Intelligence*, 12313–12320.

Examples

```
data(learning.test)
dag = hc(learning.test)
cpdag(dag)
vstructs(dag)
```

---

cpquery *Perform conditional probability queries*

Description

Perform conditional probability queries (CPQs).

Usage

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

Arguments

fitted	an object of class bn.fit.
event, evidence	see below.
nodes	a vector of character strings, the labels of the nodes whose conditional distribution we are interested in.
cluster	an optional cluster object from package **parallel**.
method	a character string, the method used to perform the conditional probability query. Currently, only *logic sampling* ("ls", the default) and *likelihood weighting* ("lw") are implemented.
...	additional tuning parameters.
debug	a boolean value. If TRUE, 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 samples conditional on the evidence using the method specified in the method argument.

Both cpquery() and cpdist() are based on Monte Carlo particle filters, so they may return slightly different values on different runs due to simulation noise.

Value

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

cpdist() returns a data frame containing the samples 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 likelihood weighting, the weights are also attached as an attribute named weights.

Logic Sampling

Logic sampling is an *approximate inference* algorithm.

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 samples to generate from fitted. The default value is 5000 * log10(nparams(fitted)) for discrete and conditional Gaussian networks and 500 * nparams(fitted) for Gaussian networks.

- batch: a positive integer number, the number of random samples that are generated at one time. Defaults to 10^4. If the n is very large (say, 10^12), R would run out of memory if it tried to generate them all at once. Instead, random samples are generated in batches of size batch, and each batch is discarded before generating the next.
- query.nodes: a vector of character strings, the labels of the nodes involved in event and evidence. Simple queries do not require generating samples 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 samples 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. Furthermore, the probabilities returned by cpquery() are approximate estimates, and they will not sum up to 1 even when the corresponding underlying values do if they are computed in separate calls to cpquery().

Likelihood Weighting

Likelihood weighting is an *approximate inference* algorithm based on Monte Carlo sampling.

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 samples returned by cpdist() are generated from the mutilated network and need to be appropriately weighted 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. Furthermore, the probabilities returned by cpquery() are approximate estimates, and they will not sum up to 1 even when the corresponding underlying values do if they are computed in separate calls to cpquery().

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"))
# programmatically build a conditional probability query...
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

cmd = paste("cpquery(fitted, ", str2, ", ", str, ")", sep = "")
eval(parse(text = cmd))
# ... but note that predict works better in this particular case.
attr(predict(fitted, "C", learning.test[obs, -3], prob = TRUE), "prob")
# do the same with likelihood weighting.
cpquery(fitted, event = eval(parse(text = str2)),
  evidence = as.list(learning.test[2, -3]), method = "lw")
attr(predict(fitted, "C", learning.test[obs, -3],
               method = "bayes-lw", prob = TRUE), "prob")
# 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))
```

`data preprocessing` *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

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

Arguments

`data`	a data frame containing numeric columns (for `dedup()`) or a combination of numeric or factor columns (for `discretize()`).
`threshold`	a numeric value between zero and one, the absolute correlation used as a threshold in screening highly correlated pairs.
`method`	a character string, the label of the method used to preprocess the data. For `discretize()`, the possible values are `"interval"` for *interval discretisation*, `"quantile"` for *quantile discretisation* (the default) or `"hartemink"` for *Hartemink's pairwise mutual information* method. For `dedup()`, the only possible value is `"cor"` for screening based on *linear correlation*.
`breaks`	an integer number, the number of levels the variables will be discretised into; or 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 discretised 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 as its first argument and returns a second data frame of discrete variables, transformed using one of three methods: `"interval"`, `"quantile"` or `"hartemink"`. Discrete variables are left unchanged.

The `"hartemink"` method has two additional tuning parameters:

- `idisc`: the method used for the initial marginal discretisation of the variables, either `"interval"` or `"quantile"`.
- `ibreaks`: the number of levels the variables are initially discretised into, in the same format as in the `breaks` argument.

It is sometimes the case that the `"quantile"` method cannot discretise one or more variables in the data without generating zero-length intervals because the quantiles are not unique. If `method = "quantile"`, `discretize()` will produce an error. If `method = "quantile"` and `idisc = "quantile"`, `discretize()` will try to lower the number of breaks set by the `ibreaks` argument until quantiles are distinct. If this is not possible without making `ibreaks` smaller than `breaks`, `discretize()` will return an error.

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

Both `discretize()` and `dedup()` accept data with missing values.

Value

`discretize()` returns a data frame with the same structure (number of columns, column names, etc.) as `data`, containing the discretised variables. The data frame has an attribute `"cutpoints"` containing the cutpoints used internally by `discretize()`, which are made available so that discretising another data set yields the same set of factors.

`dedup()` returns a data frame with a subset of the columns of `data`.

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 = 10)
plot(hc(d))
d2 = dedup(gaussian.test)
```

---

dsep *Test d-separation*

Description

Check whether two nodes are d-separated.

Usage

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

Arguments

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

Value

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

Author(s)

Marco Scutari

References

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

Examples

```
bn = model2network("[A][C|A][B|C]")
dsep(bn, "A", "B", "C")
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 `bn.fit`.
`graph`	an object of class `bn` or `bn.fit`.
`debug`	a boolean value. If `TRUE`, a lot of debugging output is printed. Otherwise, the function is completely silent.

Value

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

`write.bif()`, `write.dsc()`, `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 operate on a "best effort" basis, as the parsing of these formats was implemented by reverse-engineering the file formats from publicly available examples.

Author(s)

Marco Scutari

References

Bayesian Network Repository, https://www.bnlearn.com/bnrepository/.

---

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

Description

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

Usage

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

Format

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

Note

The R script to generate data from this network is available from https://www.bnlearn.com/documentation/networks/.

Examples

```
# load the data.
data(gaussian.test)
# create and plot the network structure.
dag = model2network("[A][B][E][G][C|A:B][D|B][F|A:D:E:G]")
## Not run: graphviz.plot(dag)
```

---

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, including.evidence = FALSE, ...)
## S3 method for class 'bn.fit'
as.grain(x)
## S3 method for class 'grain'
as.bn(x, ...)
```

Arguments

x	an object of class grain(code) (for as.bn.fit) or bn.fit() (for as.grain).
including.evidence	a boolean value. If FALSE, the grain object is converted without considering any evidence that has been set into it. If TRUE, any hard evidence is carried over into the bn.fit object as a zero-one probability distribution.
...	extra arguments from the generic method (currently ignored).

Value

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

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 m 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 enumeration *Count graphs with specific characteristics*

Description

Count directed acyclic graphs of various sizes and/or with specific characteristics.

Usage

```
count.graphs(type = "all.dags", ..., debug = FALSE)
```

Arguments

type	a character string, the label describing the types of graphs to be counted (see below).
...	additional 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

The types of graphs, and the associated additional parameters, are:

- "all-dags": all directed acyclic graphs with nodes nodes.
- "dags-given-ordering": all directed acyclic graphs with a specific topological ordering and nodes nodes.
- "dags-with-k-roots": all directed acyclic graphs with k root nodes and nodes nodes.
- "dags-with-r-arcs": all directed acyclic graphs with r arcs and nodes nodes.
- "dags-in-equivalence-class": all directed acyclic arcs in the equivalence class described by eqclass.

Value

count.graphs() returns an object of class bigz from the **gmp** package, a vector with the graph counts.

Author(s)

Marco Scutari

References

Harary F, Palmer EM (1973). "Graphical Enumeration." Academic Press.

Rodionov VI (1992). "On the Number of Labeled Acyclic Digraphs." *Discrete Mathematics*, **105**:319–321.

Liskovets VA (1976). "On the Number of Maximal Vertices of a Random Acyclic Digraph." *Theory of Probability and its Applications*, **20**(2):401–409.

Wienobst M, Luttermann M, Bannach M, Liskiewicz (2023). "Efficient Enumeration of Markov Equivalent DAGs." *Proceedings of the 37th AAAI Conference on Artificial Intelligence*, 12313–12320.

Examples

```
## Not run:
count.graphs("dags.with.r.arcs", nodes = 3:6, r = 2)

## End(Not run)
```

---

graph generation utilities

Generate empty, complete or random graphs

Description

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

Usage

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

Arguments

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

Graph generation algorithms available in `random.graph()` are:

- *full ordering* based generation (`"ordered"`): generates graphs whose node ordering is given by the order of the labels in the `nodes` argument. 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.

Additional arguments for the `random.graph()` function are:

- `prob`: the probability of each arc to be present in a graph generated by the `"ordered"` algorithm. The default value is `2 / (length(nodes) - 1)`, which results in a sparse graph (the number of arcs should be of the same order as the number of nodes).
- `burn.in`: the number of iterations for the `"ic-dag"` and `"melancon"` algorithms to converge to a stationary (and uniform) probability distribution. The default value is `6 * length(nodes)^2`.
- `every`: return only one graph `every` number of steps instead of all the graphs generated with `"ic-dag"` and `"melancon"`. Since both algorithms are Markov Chain Monte Carlo methods, higher values of `every` yield more diverse set of networks. The default value is `1`, that is, to return all generated networks.
- `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`.

`empty.graph()` generates `num` identical empty graphs, while `complete.graph()` generates `num` identical complete directed acyclic graphs.

Value

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

Author(s)

Marco Scutari

References

Ide JS, Cozman FG (2002). "Random Generation of Bayesian Networks." *Proceedings of the 16th Brazilian Symposium on Artificial Intelligence*, 366–375.

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, ..., check.cycles = TRUE)
## S3 method for class 'graphAM'
as.bn(x, ..., check.cycles = TRUE)
## 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.
...	extra arguments from the generic method (currently ignored).
check.cycles	a boolean value. If FALSE, the returned network will not be checked for cycles.

Value

An object of the relevant class.

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, directed = FALSE, debug = FALSE)
directed(x)
valid.dag(x, debug = FALSE)
valid.cpdag(x, debug = FALSE)
valid.ug(x, debug = FALSE)
# check whether there is a path between two nodes.
path.exists(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)
# perturb a network by random arc additions, removals and reversals.
perturb(x, nops, ops = c("set", "drop", "reverse"), maxp = Inf, debug = FALSE)
```

Arguments

x	an object of class bn. skeleton(), acyclic(), directed() and path.exists() also accept objects of class bn.fit.
from	a character string, the label of a node.
to	a character string, the label of a node (different from from).
direct	a boolean value. If FALSE, ignore any arc between from and to when looking for a path.

underlying.graph
: a boolean value. If TRUE, the underlying undirected graph is used instead of the (directed) one from the x argument.

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.

directed
: a boolean value. If TRUE, only completely directed cycles are considered. Otherwise, undirected arcs will also be considered and treated as arcs present in both directions.

nops
: a positive integer value, the number of arc operations applied to the network to perturb it.

ops
: a vector of character strings, one or more of "set", "drop" or "reverse". It defines which arc operations are applied to the network to perturb it.

maxp
: the maximum number of parents allowed for a node after each arc operation. The default value is Inf.

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(), subgraph() and perturb() return an object of class bn.

valid.dag(), valid.cpdag() and valid.ug() return a boolean value.

Author(s)

Marco Scutari

References

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

Andersson SA, Madigan D, Perlman MD (1997). "A Characterization of Markov Equivalence Classes for Acyclic Digraphs." The Annals of Statistics, **25**(2):505–541.

Examples

```
data(learning.test)
cpdag = pc.stable(learning.test)

acyclic(cpdag)
directed(cpdag)
dag = pdag2dag(cpdag, ordering = LETTERS[1:6])
dag
directed(dag)
skeleton(dag)
perturb(dag, nops = 3, debug = TRUE)
```

graphviz.chart *Plotting networks with parameter summaries*

Description

Plot a Bayesian network as a graph whose nodes are barplots representing the distributions of the corresponding variables. Requires the **Rgraphviz** package and, for discrete networks, also the **gRain** package.

Usage

```
graphviz.chart(x, type = "barchart", layout = "dot", draw.labels = TRUE,
  grid = FALSE, scale = c(0.75, 1.1), col = "black", bg = "transparent",
  text.col = "black", bar.col = "black", strip.bg = bg, main = NULL,
  sub = NULL)
```

Arguments

x	an object of class bn.fit.
type	a character string, the type of graph used to plot the probability distributions in the nodes. Possible values are "barchart", "dotplot" and "barprob" (a barchart with parameter values printed over the bars).
layout	a character string, the layout argument that will be passed to **Rgraphviz**. Possible values are "dot", "neato", "twopi", "circo" and "fdp". See **Rgraphviz** documentation for details.
draw.labels	a boolean value, whether to print the labels of the parameters of each variable.
grid	a boolean value, whether to draw to a reference grid for the probability distributions. If grid is TRUE, a vertical grid is drawn at probabilities c(0, 0.25, 0.50, 0.75) for discrete nodes, and at the quartiles of the regression coefficients range for continuous nodes. If grid is a numeric vector, a vertical grid is drawn at the specified values. If grid is a named list, each element is a set of grid points for the corresponding node.
scale	a vector of two positive numbers, used by **Rgraphviz** to determine the size and the aspect ratio of the nodes.
col, bg, text.col, bar.col, strip.bg	the colours of the node border, of the barchart background, of the text, of the bars and of the strip background.
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.

Value

graphviz.chart() invisibly returns NULL.

Author(s)

Marco Scutari

Examples

```
## Not run:
modelstring = 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 = "")
dag = model2network(modelstring)
fitted = bn.fit(dag, alarm)

# Netica style.
graphviz.chart(fitted, grid = TRUE, bg = "beige", bar.col = "black")
# Hugin style.
graphviz.chart(fitted, type = "barprob", grid = TRUE, bar.col = "green",
  strip.bg = "lightyellow")
# GeNIe style.
graphviz.chart(fitted, col = "darkblue", bg = "azure", bar.col = "darkblue")
# personal favourites.
graphviz.chart(fitted, type = "barprob", grid = TRUE, bar.col = "darkgreen",
  strip.bg = "lightskyblue")
graphviz.chart(fitted, type = "barprob", grid = TRUE, bar.col = "gold",
  strip.bg = "lightskyblue")
# dot-plot version.
graphviz.chart(fitted, type = "dotplot")

## End(Not run)
```

---

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, groups, layout = "dot",
  shape = "rectangle", fontsize = 12, main = NULL, sub = NULL,
  render = TRUE)
```

Arguments

x	an object of class bn or bn.fit.
highlight	a list, see below.
groups	a list of character vectors, representing groups of node labels of nodes that should be plotted close to each other.
layout	a character string, the layout argument that will be passed to **Rgraphviz**. Possible values are "dot", "neato", "twopi", "circo" and "fdp". See **Rgraphviz** documentation for details.
shape	a character string, the shape of the nodes. Can be "circle", "ellipse" or "rectangle".
fontsize	a positive number, the font size for the node labels.
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.
render	a logical value. If TRUE, graphviz.plot() actually draws the figure in addition to returning the corresponding graph object. If FALSE, no figure is produced.

Details

The highlight argument 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 labelled from and to).

and optionally one or more of the following graphical 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 transparent.
- 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**.

Note that all these parameters take a single value, which is then applied to all highlighted nodes and arcs.

Value

graphviz.plot() returns invisibly the graph object produced from the network passed as the x argument. It can be further modified using the **graph** and **Rgraphviz** packages.

Author(s)

Marco Scutari

See Also

plot.bn.

---

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

Description

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

Usage

```
data(hailfinder)
```

Format

The hailfinder data set contains the following 56 variables:

- N07muVerMo (*10.7mu vertical motion*): a four-level factor with levels StrongUp, WeakUp, Neutral and Down.
- SubjVertMo (*subjective judgment of vertical motion*): a four-level factor with levels StrongUp, WeakUp, Neutral and Down.
- QGVertMotion (*quasigeostrophic vertical motion*): a four-level factor with levels StrongUp, WeakUp, Neutral and Down.
- CombVerMo (*combined vertical motion*): a four-level factor with levels StrongUp, WeakUp, Neutral and Down.
- AreaMesoALS (*area of meso-alpha*): a four-level factor with levels StrongUp, WeakUp, Neutral and Down.
- SatContMoist (*satellite contribution to moisture*): a four-level factor with levels VeryWet, Wet, Neutral and Dry.
- RaoContMoist (*reading at the forecast center for moisture*): a four-level factor with levels VeryWet, Wet, Neutral and Dry.
- CombMoisture (*combined moisture*): a four-level factor with levels VeryWet, Wet, Neutral and Dry.
- AreaMoDryAir (*area of moisture and adry air*): a four-level factor with levels VeryWet, Wet, Neutral and Dry.
- VISCloudCov (*visible cloud cover*): a three-level factor with levels Cloudy, PC and Clear.
- IRCloudCover (*infrared cloud cover*): a three-level factor with levels Cloudy, PC and Clear.
- CombClouds (*combined cloud cover*): a three-level factor with levels Cloudy, PC and Clear.

- CldShadeOth (*cloud shading, other*): a three-level factor with levels Cloudy, PC and Clear.
- AMInstabMt (*AM instability in the mountains*): a three-level factor with levels None, Weak and Strong.
- InsInMt (*instability in the mountains*): a three-level factor with levels None, Weak and Strong.
- WndHodograph (*wind hodograph*): a four-level factor with levels DCVZFavor, StrongWest, Westerly and Other.
- OutflowFrMt (*outflow from mountains*): a three-level factor with levels None, Weak and Strong.
- MorningBound (*morning boundaries*): a three-level factor with levels None, Weak and Strong.
- Boundaries (*boundaries*): a three-level factor with levels None, Weak and Strong.
- CldShadeConv (*cloud shading, convection*): a three-level factor with levels None, Some and Marked.
- CompPlFcst (*composite plains forecast*): a three-level factor with levels IncCapDecIns, LittleChange and DecCapIncIns.
- CapChange (*capping change*): a three-level factor with levels Decreasing, LittleChange and Increasing.
- LoLevMoistAd (*low-level moisture advection*): a four-level factor with levels StrongPos, WeakPos, Neutral and Negative.
- InsChange (*instability change*): three-level factor with levels Decreasing, LittleChange and Increasing.
- MountainFcst (*mountains (region 1) forecast*): a three-level factor with levels XNIL, SIG and SVR.
- Date (*date*): a six-level factor with levels May15_Jun14, Jun15_Jul1, Jul2_Jul15, Jul16_Aug10, Aug11_Aug20 and Aug20_Sep15.
- Scenario (*scenario*): an eleven-level factor with levels A, B, C, D, E, F, G, H, I, J and K.
- ScenRelAMCIN (*scenario relevant to AM convective inhibition*): a two-level factor with levels AB and CThruK.
- MorningCIN (*morning convective inhibition*): a four-level factor with levels None, PartInhibit, Stifling and TotalInhibit.
- AMCINInScen (*AM convective inhibition in scenario*): a three-level factor with levels LessThanAve, Average and MoreThanAve.
- CapInScen (*capping withing scenario*): a three-level factor with levels LessThanAve, Average and MoreThanAve.
- ScenRelAMIns (*scenario relevant to AM instability*): a six-level factor with levels ABI, CDEJ, F, G, H and K.
- LIfr12ZDENSd (*LI from 12Z DEN sounding*): a four-level factor with levels LIGt0, N1GtLIGt_4, N5GtLIGt_8 and LILt_8.
- AMDewptCalPl (*AM dewpoint calculations, plains*): a three-level factor with levels Instability, Neutral and Stability.
- AMInsWliScen (*AM instability within scenario*): a three-level factor with levels LessUnstable, Average and MoreUnstable.
- InsSclInScen (*instability scaling within scenario*): a three-level factor with levels LessUnstable, Average and MoreUnstable.

- `ScenRel34` (*scenario relevant to regions 2/3/4*): a five-level factor with levels `ACEFK`, `B`, `D`, `GJ` and `HI`.
- `LatestCIN` (*latest convective inhibition*): a four-level factor with levels `None`, `PartInhibit`, `Stifling` and `TotalInhibit`.
- `LLIW` (*LLIW severe weather index*): a four-level factor with levels `Unfavorable`, `Weak`, `Moderate` and `Strong`.
- `CurPropConv` (*current propensity to convection*): a four-level factor with levels `None`, `Slight`, `Moderate` and `Strong`.
- `ScnRelPlFcst` (*scenario relevant to plains forecast*): an eleven-level factor with levels `A`, `B`, `C`, `D`, `E`, `F`, `G`, `H`, `I`, `J` and `K`.
- `PlainsFcst` (*plains forecast*): a three-level factor with levels `XNIL`, `SIG` and `SVR`.
- `N34StarFcst` (*regions 2/3/4 forecast*): a three-level factor with levels `XNIL`, `SIG` and `SVR`.
- `R5Fcst` (*region 5 forecast*): a three-level factor with levels `XNIL`, `SIG` and `SVR`.
- `Dewpoints` (*dewpoints*): a seven-level factor with levels `LowEverywhere`, `LowAtStation`, `LowSHighN`, `LowNHighS`, `LowMtsHighPl`, `HighEverywher`, `Other`.
- `LowLLapse` (*low-level lapse rate*): a four-level factor with levels `CloseToDryAd`, `Steep`, `ModerateOrLe` and `Stable`.
- `MeanRH` (*mean relative humidity*): a three-level factor with levels `VeryMoist`, `Average` and `Dry`.
- `MidLLapse` (*mid-level lapse rate*): a three-level factor with levels `CloseToDryAd`, `Steep` and `ModerateOrLe`.
- `MvmtFeatures` (*movement of features*): a four-level factor with levels `StrongFront`, `MarkedUpper`, `OtherRapid` and `NoMajor`.
- `RHRatio` (*realtive humidity ratio*): a three-level factor with levels `MoistMDryL`, `DryMMoistL` and `other`.
- `SfcWndShfDis` (*surface wind shifts and discontinuities*): a seven-level factor with levels `DenvCyclone`, `E_W_N`, `E_W_S`, `MovigFtorOt`, `DryLine`, `None` and `Other`.
- `SynForcng` (*synoptic forcing*): a five-level factor with levels `SigNegative`, `NegToPos`, `SigPositive`, `PosToNeg` and `LittleChange`.
- `TempDis` (*temperature discontinuities*): a four-level factor with levels `QStationary`, `Moving`, `None`, `Other`.
- `WindAloft` (*wind aloft*): a four-level factor with levels `LV`, `SWQuad`, `NWQuad`, `AllElse`.
- `WindFieldMt` (*wind fields, mountains*): a two-level factor with levels `Westerly` and `LVorOther`.
- `WindFieldPln` (*wind fields, plains*): a six-level factor with levels `LV`, `DenvCyclone`, `LongAnticyc`, `E_NE`, `SEquad` and `WidespdDnsl`.

Note

The complete BN can be downloaded from https://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.

Examples

```
# load the data.
data(hailfinder)
# create and plot the network structure.
modelstring = paste0("[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]",
  "[InsSclInScen|InsChange:AMInsWliScen][R5Fcst|MountainFcst:N34StarFcst]",
  "[PlainsFcst|CapInScen:InsSclInScen:CurPropConv:ScnRelPlFcst]")
dag = model2network(modelstring)
## Not run: graphviz.plot(dag, shape = "ellipse")
```

`hybrid algorithms` *Hybrid structure learning algorithms*

Description

Learn the structure of a Bayesian network with Max-Min Hill Climbing (MMHC), Hybrid HPC (H2PC), and the more general 2-phase Restricted Maximisation (RSMAX2) hybrid algorithms.

Usage

```
rsmax2(x, whitelist, blacklist, restrict = "si.hiton.pc",
  maximize = "hc", restrict.args = list(), maximize.args = list(), debug = FALSE)
mmhc(x, whitelist, blacklist, restrict.args = list(),
  maximize.args = list(), debug = FALSE)
h2pc(x, whitelist, blacklist, restrict.args = list(),
  maximize.args = list(), debug = FALSE)
```

Arguments

`x`	a data frame containing the variables in the model.
`whitelist`	a data frame with two columns (optionally labelled "from" and "to"), containing a set of arcs to be included in the graph.

`blacklist`	a data frame with two columns (optionally labelled "from" and "to"), containing a set of arcs not to be included in the graph.
`restrict`	a character string, the constraint-based or local search algorithm to be used in the "restrict" phase. See `structure learning` and the documentation of each algorithm for details.
`maximize`	a character string, the score-based algorithm to be used in the "maximize" phase. Possible values are `"hc"` and `"tabu"`. See `structure learning` for details.
`restrict.args`	a list of arguments to be passed to the algorithm specified by `restrict`, such as `test` or `alpha`.
`maximize.args`	a list of arguments to be passed to the algorithm specified by `maximize`, such as `restart` for hill-climbing or `tabu` for tabu search.
`debug`	a boolean value. If `TRUE`, a lot of debugging output is printed. Otherwise, the function is completely silent.

Value

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

Note

`mmhc()` is simply `rsmax2()` with `restrict` set to `"mmpc"` and `maximize` set to `"hc"`. Similarly, h2pc is simply `rsmax2()` with `restrict` set to `"hpc"`and `maximize` set to `"hc"`.

See `structure learning` for a complete list of structure learning algorithms with the respective references.

Author(s)

Marco Scutari

See Also

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

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

Description

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

Usage

```
## S3 method for class 'igraph'
as.bn(x, ..., check.cycles = TRUE)
## S3 method for class 'bn'
as.igraph(x, ...)
## S3 method for class 'bn.fit'
as.igraph(x, ...)
```

Arguments

x	an object of class bn, bn.fit, or igraph.
...	extra arguments from the generic method (currently ignored).
check.cycles	a boolean value. If FALSE, the returned network will not be checked for cycles.

Value

An object of the relevant class.

Author(s)

Marco Scutari

Examples

```
## Not run:
a = bn.fit(hc(learning.test), learning.test)
b = as.igraph(a)
plot(b, edge.arrow.mode = 2L * !igraph::which_mutual(b))
c = as.bn(b)
## End(Not run)
```

---

independence-tests *Conditional independence tests*

Description

Overview of the conditional independence tests implemented in **bnlearn**, with the respective reference publications.

Details

Unless otherwise noted, the reference publication for conditional independence tests is:

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

Additionally, 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.

and for semiparametric discrete tests:

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

Available conditional independence tests (and the respective labels) for *discrete Bayesian networks* (categorical variables) are:

- *Mutual information*: an information-theoretic distance measure. It's proportional to the log-likelihood ratio (they differ by a factor of $2n$) and is related to the deviance of the tested models. The asymptotic χ^2 test (`"mi"` and `"mi-adf"`), the Monte Carlo permutation test (`"mc-mi"`), the sequential Monte Carlo permutation test (`"smc-mi"`), and the semiparametric test (`"sp-mi"`) are implemented. Compared to `"mi"`, `"mi-adf"` adjusts the degrees of freedom for structural zeroes and automatically favours independence if there are fewer than 5 observations per parameter.
- *Shrinkage estimator* for the *mutual information* (`"mi-sh"`): an improved asymptotic χ^2 test based on the James-Stein estimator for the mutual information.

  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.
- *Pearson's* X^2: the classical Pearson's X^2 test for contingency tables. The asymptotic χ^2 test (`"x2"` and `"x2-adf"`, 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.

Available conditional independence tests (and the respective labels) for *discrete Bayesian networks* (ordered factors) are:

- *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.

Available conditional independence tests (and the respective labels) for *Gaussian Bayesian networks* (normal variables) are:

- *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.

  Hotelling H (1953). "New Light on the Correlation Coefficient and its Transforms." *Journal of the Royal Statistical Society: Series B*, **15**(2):193–225.
- *Fisher's Z*: a transformation of the linear correlation with asymptotic normal distribution. 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 factor of $2n$). The asymptotic χ^2 test (`"mi-g"`), the Monte Carlo permutation test (`"mc-mi-g"`) and the sequential Monte Carlo permutation test (`"smc-mi-g"`) are implemented.

- *Shrinkage estimator* for the *mutual information* (`"mi-g-sh"`): an improved asymptotic χ^2 test based on the James-Stein estimator for the mutual information.
  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.

No conditional independence tests are available for *zero-inflated Bayesian networks*.

Available conditional independence tests (and the respective labels) for *conditional Gaussian Bayesian networks* (mixed discrete and normal variables) are:

- *Mutual information*: an information-theoretic distance measure. Again, it is proportional to the log-likelihood ratio (they differ by a factor of $2n$). Only the asymptotic χ^2 test (`"mi-cg"`) is implemented.

`information theoretic quantities`

Compute the distance between two fitted Bayesian networks

Description

Compute Shannon's entropy of a fitted Bayesian network and the Kullback-Leibler divergence between two fitted Bayesian networks.

Usage

```
H(P)
KL(P, Q)
```

Arguments

`P`, `Q`	objects of class `bn.fit`.

Value

`H()` and `KL()` return a single numeric value.

Note

Note that in the case of Gaussian and conditional Gaussian netwoks, the divergence can be negative. Regardless of the type of network, if at least one of the two networks is singular, the divergence can be infinite.

If any of the parameters of the two networks are `NA`s, the divergence will also be `NA`.

Zero-inflated networks are not supported at the moment.

Author(s)

Marco Scutari

Examples

```
## Not run:
# discrete networks
dag = model2network("[A][C][F][B|A][D|A:C][E|B:F]")
fitted1 = bn.fit(dag, learning.test, method = "mle")
fitted2 = bn.fit(dag, learning.test, method = "bayes", iss = 20)

H(fitted1)
H(fitted2)

KL(fitted1, fitted1)
KL(fitted2, fitted2)
KL(fitted1, fitted2)

## End(Not run)

# continuous, singular networks.
dag = model2network("[A][B][E][G][C|A:B][D|B][F|A:D:E:G]")
singular = fitted1 = bn.fit(dag, gaussian.test)
singular$A = list(coef = coef(fitted1[["A"]]) + runif(1), sd = 0)

H(singular)
H(fitted1)

KL(singular, fitted1)
KL(fitted1, singular)
```

`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 https://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.

Examples

```
# load the data.
data(insurance)
# create and plot the network structure.
modelstring = paste0("[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]")
dag = model2network(modelstring)
## Not run: graphviz.plot(dag, shape = "ellipse")
```

---

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

Description

This is 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 available from https://www.bnlearn.com/documentation/networks/.

Examples

```
# load the data.
data(learning.test)
# create and plot the network structure.
dag = model2network("[A][C][F][B|A][D|A:C][E|B:F]")
## Not run: graphviz.plot(dag)
```

---

lizards *Lizards' perching behaviour data set*

Description

Real-world data set on 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.
data(lizards)
# create and plot the network structure.
dag = model2network("[Species][Diameter|Species][Height|Species]")
## Not run: graphviz.plot(dag, shape = "ellipse")

# 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)
```

---

lm integration *Produce lm objects from Bayesian networks*

Description

Take a bn object or bn.fit object encoding a Gaussian network and refit all the local distributions using lm(). This allows using all the functions provided by R for lm objects (summary(), anova(), etc.) to investigate the network.

Usage

```
## S3 method for class 'bn'
as.lm(x, data, ...)
## S3 method for class 'bn.fit'
as.lm(x, data, ...)
## S3 method for class 'bn.fit.gnode'
as.lm(x, data, ...)
```

Arguments

x	an object of class bn, bn.fit or bn.fit.gnode.
data	a data frame containing the variables in the model.
...	additional arguments, currently ignored.

Value

If x is an object of class bn or bn.fit, as.lm() returns a list of lm objects, one for each node in x. If x is an object of class bn or bn.fit.gnode, as.lm() returns a single lm object.

Author(s)

Marco

Examples

```
dag = hc(gaussian.test)
fitted = bn.fit(dag, gaussian.test)
as.lm(dag, gaussian.test)
as.lm(fitted, gaussian.test)
as.lm(fitted$F, gaussian.test)
```

---

local discovery algorithms

Local discovery structure learning algorithms

Description

ARACNE and Chow-Liu learn simple network structures from data using pairwise mutual information coefficients.

Usage

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

Arguments

`x`	a data frame containing the variables in the model.
`whitelist`	a data frame with two columns (optionally labelled "from" and "to"), containing a set of arcs to be included in the graph.
`blacklist`	a data frame with two columns (optionally labelled "from" and "to"), containing a set of arcs not to be included in the graph.
`mi`	a character string, the estimator used for the pairwise (that is, unconditional) mutual information coefficients in the ARACNE and Chow-Liu algorithms. Possible values are `"mi"` (discrete mutual information) and `"mi-g"` (Gaussian mutual information).
`debug`	a boolean value. If `TRUE`, a lot of debugging output is printed. Otherwise, the function is completely silent.

Value

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

Author(s)

Marco Scutari

See Also

constraint-based algorithms, score-based algorithms, hybrid algorithms, causal discovery 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, check.cycles = FALSE) = 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")))
## Not run: graphviz.plot(ug)
```

misc utilities *Miscellaneous utilities*

Description

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

Usage

```
## nodes
mb(x, node)
nbr(x, node, max.dist = 1)
parents(x, node)
parents(x, node, debug = FALSE) <- value
children(x, node)
children(x, node, debug = FALSE) <- value
spouses(x, node)
ancestors(x, node)
descendants(x, node)
in.degree(x, node)
out.degree(x, node)
root.nodes(x)
leaf.nodes(x)
isolated.nodes(x)
nnodes(x)

## arcs
arcs(x)
arcs(x, check.cycles = TRUE, check.illegal = TRUE, 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, check.cycles = TRUE, check.illegal = TRUE, debug = FALSE) <- value

## arc list
alst(x)
alst(x, check.cycles = TRUE, check.illegal = TRUE, debug = FALSE) <- value

## graphs
```

```
nparams(x, data, estimator = NULL, debug = FALSE)
ntests(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)
```

Arguments

x, object	an object of class bn or bn.fit. The replacement form of parents(), children(), arcs(), amat and alst() requires an object of class bn.
node, Nodes	a character string, the label of a node.
max.dist	a positive integer, the maximum number of arcs (distance) from the node at the center of the neighbourhood.
value	either a vector of character strings (for parents() and children()), an adjacency matrix (for amat()), an arc list (for alst()) or a data frame with two columns (optionally labelled "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.
estimator	a character string, the parameter estimator that will be used to fit the network; see bn.fit for the available estimators. It determines the number of parameters of the continuous nodes. If NULL, it defaults to the estimator implied by the type of the data.
check.cycles	a boolean value. If FALSE, the returned network will not be checked for cycles.
check.illegal	a boolean value. If TRUE, arcs that break the parametric assumptions of x, such as those from continuous to discrete nodes in conditional Gaussian networks, cause an error.
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 node's parents plus one (the intercept), as per Neapolitan (2003).

- For conditional linear Gaussian networks, the number of parameters of discrete and Gaussian nodes is as above. The number of parameters of conditional Gaussian nodes is equal to 1 plus the number of continuous parents (who get one regression coefficient each, plus the intercept) times the number of configurations of the discrete parents (each configuration has an associated regression model).
- For zero-inflated networks, the number of parameters of each node is equal to twice the number of parents plus three. They correspond to the set of regression coefficients of the zero-inflation component, that of the counts component, and the shape parameter (the dispersion in the hyper-Poisson, the number of failures in the negative binomial).

Both Gaussian and zero-inflated networks are learned from numeric data, so the number of parameters of the continuous nodes is ambiguous. The `estimator` argument selects which parametric assumption is used to count them; by default it is the estimator implied by the type of the data (which amounts to the Gaussian parametrisation for numeric data).

bnlearn implements arc lists (equivalently adjacency lists) as a list with one element per node. Elements are named after the nodes in `x`, and contain a character vector holding the labels of the children of that node. Undirected arcs are treated as if they were bi-directed, so each node is listed as the other's child.

Value

`mb()`, `nbr()`, `nodes()`, `parents()`, `children()`, `spouses()`, `ancestors()`, `descendants()`, `root.nodes()`, `leaf.nodes()` and `isolated.nodes()` return a vector of character strings.

`arcs()`, `directed.arcs()`, `undirected.arcs()`, `incoming.arcs()`, `outgoing.arcs()`, `incident.arcs()`, `compelled.arcs()`, `reversible.arcs()`, 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.

`alst()` returns a list of character vectors.

`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." *Proceedings of the Eleventh Annual Conference on Uncertainty in Artificial Intelligence*, 87–98.

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

Examples

```
data(learning.test)
cpdag = pc.stable(learning.test)

##  the Markov blanket of A.
mb(cpdag, "A")
```

```
## the neighbourhood of F.
nbr(cpdag, "F")
## the arcs in the graph.
arcs(cpdag)
## the nodes of the graph.
nodes(cpdag)
## the adjacency matrix for the nodes of the graph.
amat(cpdag)
## the adjacency list of a graph.
alst(cpdag)
## the parents of D.
parents(cpdag, "D")
## the children of A.
children(cpdag, "A")
## the root nodes of the graph.
root.nodes(cpdag)
## the leaf nodes of the graph.
leaf.nodes(cpdag)
## number of parameters of the Bayesian network.
dag = set.arc(cpdag, "A", "B")
nparams(dag, 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 `bn`. `modelstring()` (but not its replacement form) also accepts objects of class `bn.fit`.
`string`	a character string describing the Bayesian network.
`ordering`	the labels of all the nodes in the graph; their order is the node ordering used in the construction of the `bn` object. If `NULL`, the nodes are sorted alphabetically.

value	a character string, the same as the string.
debug	a boolean value. If TRUE, a lot of debugging output is printed. Otherwise, the function is completely silent.
...	extra arguments from the generic method (currently ignored).

Details

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

The format of the model strings is as follows. The local structure of each node is enclosed in square brackets ("[]"); the first string is the that node's label. The parents of the node (if any) are listed after a ("|") and separated by colons (":"). All nodes (including isolated and root nodes) must be listed.

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)
dag = hc(learning.test)
dag
modelstring(dag)
dag2 = model2network(modelstring(dag))
dag2
all.equal(dag, dag2)
```

---

multivariate normal distribution

Gaussian Bayesian networks and multivariate normals

Description

Convert a Gaussian Bayesian network into the multivariate normal distribution that is its global distribution, and vice versa.

Usage

```
gbn2mvnorm(fitted)
mvnorm2gbn(dag, mu, sigma)
```

Arguments

fitted	an object of class bn.fit.
dag	an object of class bn, the structure of the network that will be returned.
mu	a numeric vector, the expectation of the multivariate normal.
sigma	a square numeric matrix, the covariance matrix of the multivariate normal.

Value

gbn2mvnorm() returns a list with elements "mu" (the vector of expectations) and "sigma" (the covariance matrix).

mvnorm2gbn() returns an object of class bn.fit.

Author(s)

Marco Scutari

References

Pourahmadi M (2011). "Covariance Estimation: The GLM and Regularization Perspectives." *Statistical Science*, 26(3), 369–387.

See Also

bn.fit.

Examples

```
data(gaussian.test)
dag = model2network("[A][B][E][G][C|A:B][D|B][F|A:D:E:G]")
bn = bn.fit(dag, gaussian.test)
mvn = gbn2mvnorm(bn)
bn2 = mvnorm2gbn(dag, mu = mvn$mu, sigma = mvn$sigma)
all.equal(bn, bn2)
```

---

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

Arguments

training	a character string, the label of the training variable.
explanatory	a vector of character strings, the labels of the explanatory variables.
object	an object of class bn.fit and bn.naive.
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 normalised 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 labelled "from" and "to"), containing a set of arcs to be included in the graph.
blacklist	a data frame with two columns (optionally labelled "from" and "to"), containing a set of arcs not to be included in the graph.
mi	a character string, the estimator used for the mutual information coefficients for the Chow-Liu algorithm in TAN. Possible values are "mi" (discrete mutual information) and "mi-g" (Gaussian mutual information).
root	a character string, the label of the explanatory variable to 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 prob.
debug	a boolean value. If TRUE, a lot of debugging output is printed. Otherwise, the function is completely silent.

Details

The naive.bayes() function 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`.

See `network classifiers` for a complete list of network classifiers with the respective references.

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*, that is, sampling at random from the tied values. Therefore, setting the random seed is required to get reproducible results.

`tan.tree()` supports whitelisting and blacklisting arcs but not their directions. Moreover, it is not possible to whitelist or blacklist arcs incident on `training`.

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)

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"])
```

network-classifiers *Bayesian network Classifiers*

Description

Structure learning algorithms for Bayesian network classifiers.

Details

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 that assumes all classifiers are independent and uses the target variable's posterior probability for classification.
- *Tree-Augmented Naive Bayes* (`tree.bayes`): improves over Naive Bayes by using the Chow-Liu algorithm to approximate the dependence structure of the classifiers.

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

network-scores *Network scores*

Description

Overview of the network scores implemented in **bnlearn**, with the respective reference publications.

Details

Note that all instances of AIC, BIC and eBIC are rescaled by `-2` to make them comparable with the corresponding marginal log-likelihoods. The same is true for the penalised node average log-likelihoods.

Available scores (and the respective labels) for *discrete Bayesian networks* (categorical variables) are:

- The multinomial *log-likelihood* (`"loglik"`) score, which is equivalent to the *entropy measure* used in Weka.
- The *Akaike Information Criterion* (AIC) score (`"aic"`).
- The *Bayesian Information Criterion* (BIC) score (`"bic"`), which is equivalent to the *Minimum Description Length* (MDL) and is also known as *Schwarz Information Criterion*.

  Chickering DM (1995). "A Transformational Characterization of Equivalent Bayesian Network Structures." *Proceedings of the Eleventh Annual Conference on Uncertainty in Artificial Intelligence*, 87–98.

- The *extended Bayesian Information Criterion* (`"ebic"`), which adds a second penalty to BIC to penalise dense networks.

  Foygel R, Drton M (2010). "Extended Bayesian Information Criteria for Gaussian Graphical Models." NIPS 23, 604–612.

- The *predictive log-likelihood* (`"pred-loglik"`) computed on a separate test set.

  Chickering DM, Heckerman D (2000). "A Comparison of Scientific and Engineering Criteria for Bayesian Model Selection." *Statistics and Computing*, **10**:55–62.

  Scutari M, Vitolo C, Tucker A (2019). "Learning Bayesian Networks from Big Data with Greedy Search: Computational Complexity and Efficient Implementation." *Statistics and Computing*, **25**(9):1095–1108.

- The logarithm of the *Bayesian Dirichlet equivalent (uniform)* score (`"bde"`) (also denoted BDeu), a score-equivalent Dirichlet posterior density.

  Heckerman D, Geiger D, Chickering DM (1995). "Learning Bayesian Networks: The Combination of Knowledge and Statistical Data." *Machine Learning*, **20**(3):197–243.

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

- The logarithm of the *Bayesian Dirichlet sparse* score (`"bds"`) (BDs), a sparsity-inducing Dirichlet posterior density (not score equivalent).

  Scutari M (2016). "An Empirical-Bayes Score for Discrete Bayesian Networks." *Journal of Machine Learning Research*, **52**:438–448.

- The logarithm of the *Bayesian Dirichlet* score with *Jeffrey's prior* (not score equivalent).

  Suzuki J (2016). "A Theoretical Analysis of the BDeu Scores in Bayesian Network Structure Learning." *Behaviormetrika*, **44**(1):97–116.

- The logarithm of the modified *Bayesian Dirichlet equivalent* score (`"mbde"`) for mixtures of experimental and observational data (not score equivalent).

  Cooper GF, Yoo C (1999). "Causal Discovery from a Mixture of Experimental and Observational Data." *Proceedings of the Fifteenth Annual Conference on Uncertainty in Artificial Intelligence*, 116–125.

- The logarithm of the *K2* score (`"k2"`), a Dirichlet posterior density (not score equivalent).

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

- The logarithm of the *factorized normalized maximum likelihood* score (`"fnml"`, not score equivalent).

  Silander T, Roos T, Kontkanen P, Myllymaki P (2008). "Factorized Normalized Maximum Likelihood Criterion for Learning Bayesian Network Structures." *Proceedings of the 4th European Workshop on Probabilistic Graphical Models*, 257–272.

- The logarithm of the *quotient normalized maximum likelihood* (`"qnml"`).

  Silander T, Leppa-Abo J, Jaasaari, Roos T (2018). "Quotient Normalized Maximum Likelihood Criterion for Learning Bayesian Network Structures." *Proceedings of Machine Learning Research*, **84**:948–957.

- The node-average (log-)likelihood (`"nal"`) and the penalised node-average (log-)likelihood (`"pnal"`).

  Bodewes T, Scutari M (2021). "Learning Bayesian Networks from Incomplete Data with the Node-Averaged Likelihood." *International Journal of Approximate Reasoning*, **138**:145–160.

Available scores (and the respective labels) for *Gaussian Bayesian networks* (normal variables) are:

- The multivariate Gaussian *log-likelihood* (`"loglik-g"`) score.
- The corresponding *Akaike Information Criterion* (AIC) score (`"aic-g"`).
- The corresponding *Bayesian Information Criterion* (BIC) score (`"bic-g"`).
  Geiger D, Heckerman D (1994). "Learning Gaussian Networks." *Proceedings of the Tenth Annual Conference on Uncertainty in Artificial Intelligence*, 235–243.
- The *extended Bayesian Information Criterion* (`"ebic-g"`), which adds a second penalty to BIC to penalise dense networks. Like BIC, eBIC is rescaled by `-2`.
  Foygel R, Drton M (2010). "Extended Bayesian Information Criteria for Gaussian Graphical Models." NIPS 23, 604–612.
- The *predictive log-likelihood* (`"pred-loglik-g"`) computed on a separate test set. The reference paper is the same as that for `"pred-loglik"`. It is currently implemented to be score-equivalent like `"pred-loglik"`, but this may change.
- A score-equivalent *Gaussian posterior density* (`"bge"`).
  Kuipers J, Moffa G, Heckerman D (2014). "Addendum on the Scoring of Gaussian Directed Acyclic Graphical Models." *The Annals of Statistics*, **42**(4):1689–1691.
- The node-average (log-)likelihood (`"nal-g"`) and the penalised node-average (log-)likelihood (`"pnal-g"`).
  Bodewes T, Scutari M (2021). "Learning Bayesian Networks from Incomplete Data with the Node-Averaged Likelihood." *International Journal of Approximate Reasoning*, **138**:145–160.

Available scores (and the respective labels) for *conditional Gaussian Bayesian networks* (mixed categorical and normal variables) are:

- The conditional linear Gaussian *log-likelihood* (`"loglik-cg"`) score.
- The corresponding *Akaike Information Criterion* (AIC) score (`"aic-cg"`).
- The corresponding *Bayesian Information Criterion* (BIC) score (`"bic-cg"`).
- The *extended Bayesian Information Criterion* (`"ebic-cg"`), which adds a second penalty to BIC to penalise dense networks.
  Foygel R, Drton M (2010). "Extended Bayesian Information Criteria for Gaussian Graphical Models." NIPS 23, 604–612.
- The *predictive log-likelihood* (`"pred-loglik-cg"`) computed on a separate test set. The reference paper is the same as that for `"pred-loglik"`.
- The node-average (log-)likelihood (`"nal-cg"`) and the penalised node-average (log-)likelihood (`"pnal-cg"`).
  Bodewes T, Scutari M (2021). "Learning Bayesian Networks from Incomplete Data with the Node-Averaged Likelihood." *International Journal of Approximate Reasoning*, **138**:145–160.

Available scores (and the respective labels) for *zero-inflated Bayesian networks* (non-integer count variables with zero inflation) are:

- The zero-inflated *hyper-Poisson log-likelihood* (`"loglik-zihp"`) and *negative binomial log-likelihood* (`"loglik-zinb"`) scores.
- The corresponding *Akaike Information Criterion* (AIC) scores (`"aic-zihp"` and `"aic-zinb"`).

- The corresponding *Bayesian Information Criterion* (BIC) scores (`"bic-zihp"` and `"bic-zinb"`).
  Choi J, Ni Y (2023). "Model-Based Causal Discovery for Zero-Inflated Count Data." *Journal of Machine Learning Research*, **24**(200):1–32.
- The corresponding node-average (log-)likelihood (`"nal-zihp"` and `"nal-zinb"`) and the penalised node-average (log-)likelihood (`"pnal-zihp"` and `"pnal-zinb"`) scores.
  Bodewes T, Scutari M (2021). "Learning Bayesian Networks from Incomplete Data with the Node-Averaged Likelihood." *International Journal of Approximate Reasoning*, **138**:145–160.
  The local distributions underlying these scores are estimated by expectation-maximization with the generalised linear model components fitted by iteratively reweighted least squares (as in `bn.fit`); the estimator is controlled by the `em.max.iter`, `em.tol` and `m.step` arguments documented (with their defaults) in `score`.

Other scores (and the respective labels):

- A *custom decomposable* (`custom`) score interface that takes an R function as an argument. It can be used to trial experimental score functions without having to code them in C and hook them up to the internals of **bnlearn**.

`node operations` *Manipulate nodes in a graph*

Description

Add, remove and rename nodes in a graph.

Usage

```
# add and remove nodes.
add.node(x, node)
remove.node(x, node)

# re-label nodes.
rename.nodes(x, names)
## S4 replacement method for signature 'bn'
nodes(object) <- value
## S4 replacement method for signature 'bn.fit'
nodes(object) <- value
```

Arguments

`x`	an object of class `bn` for `add.node()` and `remove.node()`; an object of class `bn` or `bn.fit` for `rename.nodes()`.
`object`	an object of class `bn` or `bn.fit`.
`node`	a character string, the label of a node.
`value`, `names`	a vector of character strings, the new set of labels that will be used to rename the nodes.

Details

add.node() adds a new (isolated) node to an existing bn object.

remove.node() removes a node from a bn object.

rename.nodes() replaces the node labels with new ones, relabelling the whole node set. The assignment method for nodes() is an alias of rename.nodes().

Value

add.node(), remove.node() and rename.nodes() return an updated bn object.

Author(s)

Marco Scutari

Examples

```
dag = random.graph(LETTERS[1:5])
add.node(dag, "Z")
remove.node(dag, "A")

nodes(dag)
nodes(dag) = LETTERS[6:10]
nodes(dag)
```

---

node ordering utilities

Partial node orderings

Description

Find the partial node ordering implied by a network.

Usage

```
node.ordering(x, debug = FALSE)
```

Arguments

x	an object of class bn or bn.fit.
debug	a boolean value. If TRUE, a lot of debugging output is printed. Otherwise, the function is completely silent.

Value

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

Note

`node.ordering()` supports only completely directed Bayesian networks.

Author(s)

Marco Scutari

Examples

```
dag = random.graph(LETTERS[1:10])
ord = node.ordering(dag)
ord
```

---

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

Description

Convert `pcAlgo` objects to `bn` objects.

Usage

```
## S3 method for class 'pcAlgo'
as.bn(x, ..., check.cycles = TRUE)
```

Arguments

`x`	an object of class `pcAlgo`.
`...`	extra arguments from the generic method (currently ignored).
`check.cycles`	a boolean value. If `FALSE`, the returned network will not be checked for cycles.

Value

An object of class `bn`.

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 graphical parameters to be passed through to plotting functions.

Note

The following arguments 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)
cpdag = pc.stable(learning.test)

plot(cpdag)

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

---

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 to be passed through to plotting functions.

Note

The xlim and ylim arguments 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)
```

---

predict and impute *Predict or impute missing data from a Bayesian network*

Description

Impute missing values in a data set or predict a variable from a Bayesian network.

Usage

```
## S3 method for class 'bn.fit'
predict(object, node, data, cluster, method = "parents", ...,
  prob = FALSE, debug = FALSE)

impute(object, data, cluster, method, ..., strict = TRUE, debug = FALSE)
```

Arguments

object	an object of class bn.fit for impute; or an object of class bn or bn.fit for predict.
data	a data frame containing the data to be imputed. Complete observations will be ignored.
node	a character string, the label of a node.
cluster	an optional cluster object from package **parallel**.
method	a character string, the method used to impute the missing values or predict new ones. The default value is "parents" for predict() and "bayes-lw" for impute().
...	additional arguments for the imputation method. See below.
prob	a boolean value, used only by predict(). If TRUE and the node being predicted is discrete (including a discrete node in a conditional Gaussian network), the probabilities used for prediction are attached to the predicted values as an attribute called prob, a matrix with one row per level of the target variable and one column per observation. This includes the observations predicted from incomplete predictors with the "bayes-lw" and "exact" methods, which are handled by imputing the target variable.
strict	a boolean value. If TRUE, impute() will produce an error if the data were not imputed successfully, that is, if they still contain missing values. If FALSE, it will return the partially imputed data with a warning.

debug — a boolean value. If TRUE, a lot of debugging output is printed. Otherwise, the function is completely silent.

Details

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. The variables used to compute the predicted values can be specified with the from optional argument; the default is to use all relevant variables from the data. Note that the predicted values will differ in each call to predict() since this method is based on a stochastic simulation.
- "exact": the predicted values are computed using exact inference. They are maximum a posteriori estimates obtained using junction trees and belief propagation for discrete networks, or posterior expectations computed using closed-form expressions for the multivariate normal distribution for Gaussian networks. Other networks are not supported. The variables that are used to compute the predicted values can be specified with the from optional argument; the default is to use those in the Markov blanket of node.

impute() is based on predict(), and can impute missing values with the same methods ("parents", "bayes-lw" and "exact"). The method "bayes-lw" can take an additional argument n, the number of random samples averaged for each observation. As in predict(), imputed values will differ in each call to impute() when method is set to "bayes-lw".

If object contains NA parameter estimates (because parameters were learned from data with unobserved discrete parent configurations), predict will predict NAs when those parent configurations appear in data. See bn.fit for details on how to make sure bn.fit objects contain no NA parameter estimates.

Value

predict() returns a numeric vector (for Gaussian, conditional Gaussian and zero-inflated nodes), a factor (for categorical nodes) or an ordered factor (for ordinal nodes). If prob = TRUE and the predicted node is discrete, the probabilities used for prediction are attached to the predicted values as an attribute called prob.

impute() returns a data frame with the same structure as data.

Note

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

Classifiers have a separate predict() method, see naive.bayes.

Author(s)

Marco Scutari

Examples

```
# missing data imputation.
with.missing.data = gaussian.test
with.missing.data[sample(nrow(with.missing.data), 500), "F"] = NA
fitted = bn.fit(model2network("[A][B][E][G][C|A:B][D|B][F|A:D:E:G]"),
           gaussian.test)
imputed = impute(fitted, with.missing.data)

# predicting a variable in the test set.
training = bn.fit(model2network("[A][B][E][G][C|A:B][D|B][F|A:D:E:G]"),
           gaussian.test[1:2000, ])
test = gaussian.test[2001:nrow(gaussian.test), ]
predicted = predict(training, node = "F", data = test)

# obtain the conditional probabilities for the values of a single variable
# given a subset of the rest, they are computed to determine the predicted
# values.
fitted = bn.fit(model2network("[A][C][F][B|A][D|A:C][E|B:F]"), learning.test)
evidence = data.frame(A = factor("a", levels = levels(learning.test$A)),
                      F = factor("b", levels = levels(learning.test$F)))
predicted = predict(fitted, "C", evidence,
              method = "bayes-lw", prob = TRUE)
attr(predicted, "prob")
```

---

rbn *Simulate random samples from a given Bayesian network*

Description

Simulate random samples from a given Bayesian network.

Usage

```
rbn(x, n = 1, ..., debug = FALSE)
```

Arguments

x	an object of class bn.fit.
n	a positive integer giving the number of observations to generate.
...	additional arguments for the parameter estimation procedure, see again bn.fit for details.
debug	a boolean value. If TRUE, a lot of debugging output is printed. Otherwise, the function is completely silent.

Details

rbn() implements forward/logic sampling: values for the root nodes are sampled from their (unconditional) distribution, then those of their children conditional on the respective parent sets. This is done iteratively until values have been sampled for all nodes.

If x contains NA parameter estimates (because of unobserved discrete parent configurations in the data the parameters were learned from), rbn() will produce samples that contain NAs when those parent configurations appear in the simulated samples. See bn.fit for details on how to make sure bn.fit objects contain no NA parameter estimates.

Value

A data frame with the same structure as the data originally used to fit the parameters of the Bayesian network.

Author(s)

Marco Scutari

References

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

See Also

cpdist.

Examples

```
data(learning.test)
dag = hc(learning.test)
fitted = bn.fit(dag, learning.test)
rbn(fitted, 5)
```

---

scm class *The scm class structure*

Description

The structure of an object of S3 class scm.

Usage

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

Arguments

x	an object of class bn.
...	additional arguments, currently ignored.

Details

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

- roles: a list with three named elements:
  - factual: a vector of character strings, the labels of the factual nodes in the structural causal model.
  - exogenous: a vector of character strings, the labels of the exogenous nodes in the structural causal model.
  - counterfactual: a vector of character strings, the labels of the counterfactual nodes in the structural causal model. Unlike previous elements, it can be empty.
- nodes: a list. Each element is named after a node and contains the following elements:
  - counterfactual: a character string.
    - For a factual node, it contains the label of the corresponding counterfactual node (if any).
    - For an exogenous node, it contains the label of the counterfactual node linked to it (if any).
    - For a counterfactual node, it is set to character(0).
  - exogenous: a character string.
    - For a factual or counterfactual node, it contains the label of the corresponding exogenous node (if any).
    - For an exogenous node, it is set to character(0).
  - factual: a character string.
    - For a counterfactual node, it contains the label of the corresponding factual node.
    - For an exogenous node, it contains the label of the factual node linked to it (if any).
    - For a factual node, it is set to character(0).
  - parents: the parents of the node (a vector of character strings). Note that exogenous nodes are not listed among the parents of factual and counterfactual nodes.
  - children: the children of the node (a vector of character strings). Note that factual and counterfactual nodes are not listed among the children of exogenous nodes.
- arcs: the arcs of the structural causal model (a two-column matrix, whose columns are labelled from and to), including the arcs between exogenous and factual/counterfactual nodes.

Author(s)

Marco Scutari

score *Score of the Bayesian network*

Description

Compute the score of the Bayesian network.

Usage

```
## S4 method for signature 'bn'
score(x, data, type = NULL, ..., by.node = FALSE, debug = FALSE)
## S4 method for signature 'bn.naive'
score(x, data, type = NULL, ..., by.node = FALSE, debug = FALSE)
## S4 method for signature 'bn.tan'
score(x, data, type = NULL, ..., by.node = FALSE, 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 that will be used to compute the score.
type	a character string, the label of a network score. If none is specified, the default score is the *Bayesian Information Criterion* for both discrete and continuous data sets. See network scores for details.
by.node	a boolean value. If TRUE and the score is decomposable, the function returns the score terms corresponding to each node. Otherwise, it returns their sum (the overall score of x).
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 coefficient to be used; the default k = 1 gives the expression used to compute the AIC in the context of scoring Bayesian networks.

Details

Additional arguments of the score() function:

- iss: the imaginary sample size used by the Bayesian Dirichlet scores ("bde", "mbde", "bds"). It is also known as "equivalent sample size". The default value is equal to 1.
- iss.mu: the imaginary sample size for the normal component of the normal-Wishart prior in the Bayesian Gaussian score ("bge"). The default value is 1.
- iss.w: the imaginary sample size for the Wishart component of the normal-Wishart prior in the Bayesian Gaussian score ("bge"). The default value is ncol(data) + 2.
- nu: the mean vector of the normal component of the normal-Wishart prior in the Bayesian Gaussian score ("bge"). The default value is equal to colMeans(data).
- 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 contain a numeric vector with the indexes of the observations whose values have been manipulated for that node.
- k: the penalty coefficient to be used by the AIC, BIC and penalised node-average log-likelihood scores. The default value is 1 for AIC, log(nrow(data)) / 2 for BIC and 1 / nnnodes(x) * nrow(data) ^ -0.25 for the node-average log-likelihood scores.
- gamma: the additional penalty in the extended BIC scores. The default value is 0.5.
- prior: the prior distribution to be used with the various Bayesian Dirichlet scores ("bde", "mbde", "bds", "bdj" 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.
  - "marginal": an independent marginal uniform for each arc.
  - "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/ncol(data).
  - If prior is "marginal", beta is the probability of inclusion of an arc. Each direction has a probability of inclusion of beta / 2 and the probability that the arc is not included is therefore 1 - beta. The default value is 0.5, so that arc inclusion and arc exclusion have the same probability.
  - 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.
- em.max.iter: the maximum number of expectation-maximization iterations used to fit the local distributions of the zero-inflated count scores ("loglik-zihp", "loglik-zinb", "aic-zihp", "aic-zinb", "bic-zihp", "bic-zinb", "nal-zihp", "nal-zinb", "pnal-zihp" and "pnal-zinb"). The default value is 100.
- em.tol: the relative convergence tolerance (on both the log-likelihood and the parameters) of the expectation-maximization iteration used by the zero-inflated count scores. The default value is 1e-8.
- m.step: the type of maximisation step in that expectation-maximization, either "full" (the default) to re-fit each generalised linear model component to convergence at every iteration or "one-step" to take a single iteratively reweighted least squares step per iteration (a cheaper generalized-EM update).

- newdata: the test set whose predictive likelihood will be computed by "pred-loglik", "pred-loglik-g" or "pred-loglik-cg". It should be a data frame with the same variables as data.
- fun: the function that computes the score component for a single node in the custom score. fun must have arguments node, parents, data and args, in this order. In other words, it must have signature function(node, parents, data, args). node will contain the label of the node to be scored (a character string); parents will contain the labels of its parents (a character vector); data will contain the complete data set, with all the variables (a data frame); and args will be a list containing any additional arguments to the score.
- args: a list containing the optional arguments to fun, for tuning custom score functions.

Value

For score() with by.node = TRUE, a vector of numeric values, the individual node contributions to the score of the Bayesian network. Otherwise, a single numeric value, the score of the Bayesian network.

Note

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

When using the Castelo & Siebes prior in structure learning, the prior probabilities associated with an arc are bound away from zero and one by shrinking them towards the uniform distribution as per Hausser and Strimmer (2009) with a lambda equal to 3 * sqrt(.Machine$double.eps). 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 arcs from being included in the network, and a whitelist can be used to force the inclusion of particular arcs. 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

See Also

network scores, arc.strength, alpha.star.

Examples

```
data(learning.test)
dag = hc(learning.test)
score(dag, learning.test, type = "bde")

## let's see score equivalence in action!
dag2 = set.arc(dag, "B", "A")
score(dag2, learning.test, type = "bde")

## K2 score on the other hand is not score equivalent.
```

```
score(dag, learning.test, type = "k2")
score(dag2, learning.test, type = "k2")

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

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

## equivalent to AIC(dag, learning.test)
score(dag, learning.test, type = "aic")

## custom score, computing BIC manually.
my.bic = function(node, parents, data, args) {

  n = nrow(data)

  if (length(parents) == 0) {

    counts = table(data[, node])
    nparams = dim(counts) - 1
    sum(counts * log(counts / n)) - nparams * log(n) / 2

  }#THEN
  else {

    counts = table(data[, node], configs(data[, parents, drop = FALSE]))
    nparams = ncol(counts) * (nrow(counts) - 1)
    sum(counts * log(prop.table(counts, 2))) - nparams * log(n) / 2

  }#ELSE

}#MY.BIC
score(dag, learning.test, type = "custom-score", fun = my.bic, by.node = TRUE)
score(dag, learning.test, type = "bic", by.node = TRUE)
```

`score-based algorithms`

Score-based structure learning algorithms

Description

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

Usage

```
hc(x, start = NULL, whitelist, blacklist, score = NULL, ...,
```

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

Arguments

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

Value

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

Note

See structure learning for a complete list of structure learning algorithms with the respective references.

Author(s)

Marco Scutari

See Also

network scores, constraint-based algorithms, hybrid algorithms, local discovery algorithms, causal discovery algorithms, alpha.star.

`single-node local discovery`

Discover the structure around a single node

Description

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

Usage

```
learn.mb(x, node, method, whitelist, blacklist, start = NULL,
  test = NULL, alpha = 0.05, ..., max.sx = NULL, debug = FALSE)
learn.nbr(x, node, method, whitelist, blacklist,
  test = NULL, alpha = 0.05, ..., max.sx = 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 listed in structure learning.
`whitelist`	a vector of character strings, the labels of the whitelisted nodes.
`blacklist`	a vector of character strings, the labels of the blacklisted nodes.
`start`	a vector of character strings, the labels of the nodes to be included in the Markov blanket before the learning process (in `learn.mb`). Note that the nodes in `start` can be removed from the Markov blanket by the learning algorithm, unlike the nodes included due to whitelisting.
`test`	a character string, the label of the conditional independence test to be used in the algorithm. If none is specified, the default test statistic is the *mutual information* for categorical variables, the Jonckheere-Terpstra test for ordered factors and the *linear correlation* for continuous variables. See `independence tests` for details.
`alpha`	a numeric value, the target nominal type I error rate.
`...`	optional arguments to be passed to the test specified by `test`. See `ci.test` for details.
`max.sx`	a positive integer, the maximum allowed size of the conditioning sets used in conditional independence tests. The default is that there is no limit on size.
`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()`).

Note

All algorithms used by `learn.mb()` and `learn.nbr()` accept incomplete data, which they handle by computing individual conditional independence tests on locally complete observations.

Author(s)

Marco Scutari

See Also

constraint-based algorithms.

Examples

```
learn.mb(learning.test, node = "D", method = "iamb")
learn.mb(learning.test, node = "D", method = "iamb", blacklist = c("A", "F"))

learn.nbr(gaussian.test, node = "F", method = "si.hiton.pc", whitelist = "D")
```

---

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, groups,
  layout = "dot", shape = "rectangle", fontsize = 12, main = NULL, sub = NULL,
  render = TRUE, 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` argument.
`threshold`	a numeric value. See below.
`cutpoints`	an array of numeric values. See below.
`highlight`	a list, see `graphviz.plot` for details.

groups	a list of character vectors, representing groups of node labels of nodes that should be plotted close to each other.
layout	a character string, the layout argument that will be passed to **Rgraphviz**. Possible values are "dot", "neato", "twopi", "circo" and "fdp". See **Rgraphviz** documentation for details.
shape	a character string, the shape of the nodes. Can be "circle", "ellipse" or "rectangle".
fontsize	a positive number, the font size for the node labels.
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.
render	a logical value. If TRUE, strength.plot() actually draws the figure in addition to returning the corresponding graph object. If FALSE, no figure is produced.

Details

The threshold argument 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) less than or equal to the threshold is considered significant.
- 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) less than the threshold is considered significant.
- If arc strengths have been computed using bootstrap or using Bayes factors, any strength coefficient (which can be interpreted as a probability for inclusion) greater than or equal to the threshold is considered significant.

The default value is the value of the strength attribute of the bn.strength object passed via the strength argument.

Non-significant arcs are plotted as dashed lines.

The cutpoints argument is an array of numeric values used to divide the range of 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

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

for bootstrap estimates or by the quantiles

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

of the significant differences if network scores are used.

Value

graphviz.plot() returns invisibly the graph object produced by **Rgraphviz**. It can be further modified using the commands present in the **graph** and **Rgraphviz** packages, and it contains the arc strengths in the edge weight attribute.

Author(s)

Marco Scutari

Examples

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

## End(Not run)
```

---

structural.em *Structure learning from missing data*

Description

Learn the structure of a Bayesian network from a data set containing missing values using Structural EM.

Usage

```
structural.em(x, maximize = "hc", maximize.args = list(), fit,
    fit.args = list(), impute, impute.args = list(), return.all = FALSE,
    start = NULL, max.iter = 5, debug = FALSE)
```

Arguments

x	a data frame containing the variables in the model.
maximize	a character string, the score-based algorithm to be used in the "maximisation" step. See structure learning for details.
maximize.args	a list of arguments to be passed to the algorithm specified by maximize, such as restart for hill-climbing or tabu for tabu search.
fit	a character string, the parameter learning method to be used in the "maximisation" step. See bn.fit for details.

fit.args	a list of arguments to be passed to the parameter learning method specified by fit.
impute	a character string, the imputation method to be used in the "expectation" step. See impute for details.
impute.args	a list of arguments to be passed to the imputation method specified by impute.
return.all	a boolean value. See below for details.
start	a bn or bn.fit object, the network used to perform the first imputation and as a starting point for the score-based algorithm specified by maximize.
max.iter	an integer, the maximum number of iterations.
debug	a boolean value. If TRUE, a lot of debugging output is printed. Otherwise, the function is completely silent.

Value

If return.all is FALSE, structural.em() returns an object of class bn. (See bn-class for details.)

If return.all is TRUE, structural.em() returns a list with three elements named dag (an object of class bn), imputed (a data frame containing the imputed data from the last iteration) and fitted (an object of class bn.fit, again from the last iteration; see bn.fit-class for details).

Note

If at least one of the variables in the data x does not contain any observed value, the start network must be specified, and it must be a bn.fit object. Otherwise, structural.em() is unable to complete the first *maximisation* step because it cannot fit the corresponding local distribution(s).

Note that if impute is set to "bayes-lw", each call to structural.em may produce a different model since the imputation is based on a stochastic simulation.

Author(s)

Marco Scutari

References

Friedman N (1997). "Learning Belief Networks in the Presence of Missing Values and Hidden Variables." *Proceedings of the 14th International Conference on Machine Learning*, 125–133.

See Also

score-based algorithms, bn.fit, impute.

Examples

```
data(learning.test)

# learn with incomplete data.
incomplete.data = learning.test
incomplete.data[1:100, 1] = NA
```

```
incomplete.data[101:200, 2] = NA
incomplete.data[1:200, 5] = NA
structural.em(incomplete.data)

## Not run:
# learn with a latent variable.
incomplete.data = learning.test
incomplete.data[seq(nrow(incomplete.data)), 1] = NA
start = bn.fit(empty.graph(names(learning.test)), learning.test)
wl = data.frame(from = c("A", "A"), to = c("B", "D"))
structural.em(incomplete.data, start = start,
  maximize.args = list(whitelist = wl))

## End(Not run)
```

`structure-learning` *Structure learning algorithms*

Description

Overview of the structure learning algorithms implemented in **bnlearn**, with the respective reference publications.

Available Constraint-Based Learning Algorithms

- *PC* (`pc.stable`), a modern implementation of the first practical constraint-based structure learning algorithm.

  Colombo D, Maathuis MH (2014). "Order-Independent Constraint-Based Causal Structure Learning." *Journal of Machine Learning Research*, **15**:3921–3962.

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

  Margaritis D (2003). *Learning Bayesian Network Model Structure from Data*. Ph.D. thesis, School of Computer Science, Carnegie-Mellon University, Pittsburgh, PA.

- *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).

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

- *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.

  Yaramakala S, Margaritis D (2005). "Speculative Markov Blanket Discovery for Optimal Feature Selection." *Proceedings of the Fifth IEEE International Conference on Data Mining*, 809–812.

- *Incremental Association with FDR* (`iamb.fdr`): a variant of IAMB which adjusts the tests' significance threshold with FDR.

  Pena JM (2008). "Learning Gaussian Graphical Models of Gene Networks with False Discovery Rate Control." *Proceedings of the Sixth European Conference on Evolutionary Computation, Machine Learning and Data Mining in Bioinformatics*, 165–176.

  Gasse M, Aussem A, Elghazel H (2014). "A Hybrid Algorithm for Bayesian Network Structure Learning with Application to Multi-Label Learning." *Expert Systems with Applications*, **41**(15):6755–6772.

bnlearn includes two implementations of each algorithm: a vanilla implementation, and a parallel one that requires a running cluster set up with the `makeCluster` function from the **parallel** package.

Available Score-based Learning Algorithms

- *Hill-Climbing* (`hc`): a *hill climbing* greedy search that explores the space of the directed acyclic graphs by single-arc addition, removal and reversals; with random restarts to avoid local optima. The optimised 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.

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

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).

  Tsamardinos I, Brown LE, Aliferis CF (2006). "The Max-Min Hill-Climbing Bayesian Network Structure Learning Algorithm." *Machine Learning*, **65**(1):31–78.
- *Restricted Maximisation* (`rsmax2`): a general implementation of the Sparse Candidate algorithms, which can use any combination of constraint-based and score-based algorithms.

  Friedman N, Nachman I, Pe'er D (1999). "Learning Bayesian Network Structure from Massive Datasets: the Sparse Candidate Algorithm." *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence (UAI)*, 206–215.
- *Hybrid HPC* (`h2pc`): a hybrid algorithm combining HPC and hill-climbing.

  Gasse M, Aussem A, Elghazel H (2014). "A Hybrid Algorithm for Bayesian Network Structure Learning with Application to Multi-Label Learning." *Expert Systems with Applications*, **41**(15):6755–6772.

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. Therefore, by default, all 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 maximisation of the minimum association measure observed with any subset of the nodes selected in the previous iterations.

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

- *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.

  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.

- *Hybrid Parents and Children* (`hpc`): an algorithm building on `iamb.fdr` to learn the parents and children of each node like `mmpc` and `si.hiton.pc`. The reference publication is the same as that for Hybrid HPC.

Pairwise Mutual Information Algorithms

These algorithms learn approximate network structures using only pairwise mutual information.

- *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.

  Chow CK, Liu CN (1968). "Approximating Discrete Probability Distributions with Dependence Trees." IEEE Transactions on Information Theory, IT-14 **3**:462–467.

- *ARACNE* (`aracne`): an improved version of the Chow-Liu algorithm that is able to learn polytrees.

  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.

All these algorithms have two implementations (vanilla and parallel), like other constraint-based algorithms.

Modern Causal Discovery Algorithms

These algorithms are based on assumptions that ensure the complete identifiability of arc direction, which is standard for causal discovery.

- *DirectLiNGAM* (`direct.lingam`): a causal discovery algorithm to learn linear causal networks from non-Gaussian data, based on the LiNGAM principle.

  Shimizu S, Inazumi T, Sogawa Y, Hyvarinen A, Kawahara Y, Washio T, Hoyer PO, Bollen K (2011). "DirectLiNGAM: A Direct Method for Learning a Linear Non-Gaussian Structural Equation Model." *Journal of Machine Learning Research*, **12**:1225–1248.

They also have two implementations (vanilla and parallel).

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()
```

utilities for whitelists and blacklists *Get or create whitelists and blacklists*

Description

Extract whitelists and blacklists from an object of class bn, or create them for use in structure learning.

Usage

```
whitelist(x)
blacklist(x)

ordering2blacklist(nodes)
tiers2blacklist(tiers)
set2blacklist(set)
```

Arguments

x	an object of class bn.
nodes, set	a vector of character strings, the labels of the nodes.
tiers	a vector of character strings or a list, see below.

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 from the graph. In both cases, ordering2blacklist() returns a blacklist containing 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 one list element (sometimes known as *tier*) to a node in previous elements are blacklisted. Arcs between nodes in the same element are not blacklisted.

set2blacklist() creates a blacklist containing all the arcs between any two of the nodes whose labels are passed as the argument set.

Value

whitelist() and blacklist() return a matrix of character strings with two columns, named from and to, if a whitelist or a blacklist has been used to learn the bn object passed as their argument.

ordering2blacklist(), tiers2blacklist() and set2blacklist() return a sanitized blacklist (a two-column matrix, whose columns are labelled from and to).

Author(s)

Marco Scutari

Examples

```
tiers2blacklist(list(LETTERS[1:3], LETTERS[4:6]))
set2blacklist(LETTERS[1:3])
ordering2blacklist(LETTERS[1:6])
```

`whitelists-blacklists` *Whitelists and blacklists in structure learning*

Description

How whitelists and blacklists are used in structure learning.

Constraint-based Algorithms

Constraint-based algorithms support arc whitelisting and blacklisting as follows:

- Blacklisted arcs are never present in the learned graph.
- Arcs whitelisted in one direction only (that is, $A \rightarrow B$ is whitelisted but $B \rightarrow A$ is not) have the respective reverse arcs blacklisted, and are always present in the learned graph.
- Arcs whitelisted in both directions (that is, both $A \rightarrow B$ and $B \rightarrow A$ are whitelisted) are present in the learned 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.

Score-based Algorithms

Score-based algorithms support arc whitelisting and blacklisting as follows:

- Blacklisted arcs are never present in the learned graph.
- Arcs can only be whitelisted in a single direction, and are always present in the learned graph; it is not possible to whitelist arcs in both directions.

Hybrid Algorithms

Hybrid algorithms use constraint-based (or pairwise mutual information) algorithms in the *restrict phase* and score-based algorithms in the *maximise phase*. Hence, whitelists and blacklists are supported as follows:

- Whitelists and blacklists should be specified for the algorithm used in the restrict phase.
- If the whitelist contains any undirected arc, its consistent extension is used instead in the maximise phase.

Pairwise Mutual Information Algorithms

In algorithms that learn undirected graphs, such as ARACNE and Chow-Liu, arcs are treated as being whitelisted or blacklisted in both directions, even if only one direction is listed in the whitelist or blacklist. Again, blacklisted arcs are never present in the learned graph and whitelisted arcs are guaranteed to be present in the learned graph.

Index

∗ **causality**
  causal discovery algorithms, 35
  causal inference, 37
∗ **classes**
  bn class, 16
  bn.fit class, 28
  bn.kcv class, 34
  bn.strength class, 35
  scm class, 100
∗ **classifiers**
  naive.bayes, 86
∗ **convenience functions**
  bn.fit utilities, 31
  configs, 43
  misc utilities, 81
  model string utilities, 84
  node ordering utilities, 93
  test counter, 115
  utilities for whitelists and blacklists, 115
∗ **data preprocessing**
  configs, 43
  data preprocessing, 51
  predict and impute, 97
∗ **datasets**
  alarm, 5
  asia, 13
  clgaussian.test, 40
  coronary, 46
  gaussian.test, 55
  hailfinder, 66
  insurance, 74
  learning.test, 76
  lizards, 77
  marks, 80
∗ **graphs**
  arc operations, 8
  compare, 41
  cpdag, 47
  dsep, 53
  graph enumeration, 57
  graph utilities, 61
  misc utilities, 81
  node operations, 92
  node ordering utilities, 93
∗ **import/export to file**
  foreign files utilities, 54
∗ **independence tests**
  arc.strength, 10
  ci.test, 38
∗ **inference**
  causal inference, 37
  cpquery, 48
  information theoretic quantities, 73
  predict and impute, 97
  rbn, 99
∗ **interfaces to other packages**
  bn.fit plots, 30
  compare, 41
  gRain integration, 56
  graph integration, 60
  graphviz.chart, 63
  graphviz.plot, 64
  igraph integration, 70
  lm integration, 78
  pcalg integration, 94
  strength.plot, 108
∗ **local learning**
  local discovery algorithms, 79
  single-node local discovery, 107
∗ **missing data**
  predict and impute, 97
  structural.em, 110
∗ **network scores**
  alpha.star, 7
  arc.strength, 10
  BF, 15

bn.fit utilities, 31
score, 102
∗ **package**
bnlearn-package, 3
independence-tests, 71
network-classifiers, 89
network-scores, 89
structure-learning, 112
whitelists-blacklists, 117
∗ **parameter learning**
bn.fit, 22
information theoretic quantities, 73
multivariate normal distribution, 85
∗ **plots**
bn.fit plots, 30
compare, 41
graphviz.chart, 63
graphviz.plot, 64
plot.bn, 95
plot.bn.strength, 96
strength.plot, 108
∗ **resampling**
bn.boot, 18
bn.cv, 19
∗ **simulation**
cpquery, 48
graph generation utilities, 58
rbn, 99
∗ **structure learning**
alpha.star, 7
arc.strength, 10
BF, 15
causal discovery algorithms, 35
constraint-based algorithms, 44
cpdag, 47
hybrid algorithms, 69
naive.bayes, 86
score-based algorithms, 105
structural.em, 110
utilities for whitelists and blacklists, 115
∗ **utilities**
gRain integration, 56
graph integration, 60
igraph integration, 70
pcalg integration, 94

$<-.bn.fit *(bn.fit)*, 22

acyclic *(graph utilities)*, 61
add.node *(node operations)*, 92
AIC.bn *(score)*, 102
AIC.bn.fit *(bn.fit utilities)*, 31
alarm, 5
all.equal.bn *(compare)*, 41
alpha.star, 7, *104*, *107*
alst *(misc utilities)*, 81
alst<- *(misc utilities)*, 81
amat *(misc utilities)*, 81
amat<- *(misc utilities)*, 81
ancestors *(misc utilities)*, 81
aracne, *114*
aracne *(local discovery algorithms)*, 79
arc operations, 8
arc.strength, 10, *35*, *40*, *104*
arcs *(misc utilities)*, 81
arcs<- *(misc utilities)*, 81
as.bn *(model string utilities)*, 84
as.bn.fit *(gRain integration)*, 56
as.bn.grain *(gRain integration)*, 56
as.bn.graphAM *(graph integration)*, 60
as.bn.graphNEL *(graph integration)*, 60
as.bn.igraph *(igraph integration)*, 70
as.bn.pcAlgo *(pcalg integration)*, 94
as.character.bn *(model string utilities)*, 84
as.grain *(gRain integration)*, 56
as.graphAM *(graph integration)*, 60
as.graphNEL *(graph integration)*, 60
as.igraph *(igraph integration)*, 70
as.lm *(lm integration)*, 78
as.scm *(scm class)*, 100
asia, 13
averaged.network *(arc.strength)*, 10

BF, *11*, 15
bf.strength, *12*, *15*
bf.strength *(arc.strength)*, 10
BIC.bn *(score)*, 102
BIC.bn.fit *(bn.fit utilities)*, 31
blacklist *(utilities for whitelists and blacklists)*, 115
bn class, 16
bn-class *(bn class)*, 16
bn.boot, 18, *22*
bn.cv, 19, *19*

bn.fit, *20*, 22, *31*, *33*, *82*, *86*, *92*, *98–100*, *110*, *111*
bn.fit class, 28
bn.fit plots, 30
bn.fit utilities, 31
bn.fit-class *(*bn.fit class*)*, 28
bn.fit.barchart *(*bn.fit plots*)*, 30
bn.fit.dnode *(*bn.fit class*)*, 28
bn.fit.dotplot *(*bn.fit plots*)*, 30
bn.fit.gnode *(*bn.fit class*)*, 28
bn.fit.histogram *(*bn.fit plots*)*, 30
bn.fit.qqplot *(*bn.fit plots*)*, 30
bn.fit.xyplot *(*bn.fit plots*)*, 30
bn.kcv class, 34
bn.kcv-class *(*bn.kcv class*)*, 34
bn.kcv.list class *(*bn.kcv class*)*, 34
bn.kcv.list-class *(*bn.kcv class*)*, 34
bn.net *(*bn.fit*)*, 22
bn.strength *(*bn.strength class*)*, 35
bn.strength class, 35
bn.strength-class *(*bn.strength class*)*, 35
bnlearn *(*bnlearn-package*)*, 3
bnlearn-package, 3
boot.strength, *12*, *35*
boot.strength *(*arc.strength*)*, 10

causal discovery algorithms, 35, *45*, *70*, *79*, *107*
causal inference, 37
cextend, *26*, *45*
cextend *(*cpdag*)*, 47
children *(*misc utilities*)*, 81
children<- *(*misc utilities*)*, 81
chow.liu, *114*
chow.liu *(*local discovery algorithms*)*, 79
ci.test, *13*, 38, *45*, *107*
clgaussian.test, 40
coef.bn.fit *(*bn.fit utilities*)*, 31
colliders *(*cpdag*)*, 47
compare, *15*, 41
compelled.arcs *(*misc utilities*)*, 81
complete.graph *(*graph generation utilities*)*, 58
configs, 43
constraint-based algorithms, *36*, 44, *70*, *79*, *107*, *108*
coronary, 46
count.graphs *(*graph enumeration*)*, 57
counterfactual *(*causal inference*)*, 37
cpdag, 47
cpdist, *100*
cpdist *(*cpquery*)*, 48
cpquery, *37*, 48
custom.fit *(*bn.fit*)*, 22
custom.strength, *12*, *35*
custom.strength *(*arc.strength*)*, 10

data preprocessing, 51
dedup *(*data preprocessing*)*, 51
degree *(*misc utilities*)*, 81
degree,bn-method *(*misc utilities*)*, 81
degree,bn.fit-method *(*misc utilities*)*, 81
degree,bn.naive-method *(*misc utilities*)*, 81
degree,bn.tan-method *(*misc utilities*)*, 81
descendants *(*misc utilities*)*, 81
direct.lingam, *114*
direct.lingam *(*causal discovery algorithms*)*, 35
directed *(*graph utilities*)*, 61
directed.arcs *(*misc utilities*)*, 81
discretize *(*data preprocessing*)*, 51
drop.arc *(*arc operations*)*, 8
drop.edge *(*arc operations*)*, 8
dsep, 53

em-based algorithms *(*structural.em*)*, 110
empty.graph *(*graph generation utilities*)*, 58

fast.iamb, *112*
fast.iamb *(*constraint-based algorithms*)*, 44
fitted.bn.fit *(*bn.fit utilities*)*, 31
foreign files utilities, 54

gaussian.test, 55
gbn2mvnorm *(*multivariate normal distribution*)*, 85
gRain integration, 56
graph enumeration, 57
graph generation utilities, 58
graph integration, 60
graph utilities, 61

graphviz.chart, 63
graphviz.compare (compare), 41
graphviz.plot, 64, *95*, *108*
gs, *112*
gs (constraint-based algorithms), 44

H (information theoretic quantities), 73
h2pc, *113*
h2pc (hybrid algorithms), 69
hailfinder, 66
hamming (compare), 41
hc, *113*
hc (score-based algorithms), 105
hpc, *114*
hpc (constraint-based algorithms), 44
hybrid algorithms, *36*, *45*, 69, *79*, *107*

iamb, *112*
iamb (constraint-based algorithms), 44
iamb.fdr, *113*
identifiable (bn.fit utilities), 31
igraph integration, 70
impute, *111*
impute (predict and impute), 97
in.degree (misc utilities), 81
incident.arcs (misc utilities), 81
inclusion.threshold (arc.strength), 10
incoming.arcs (misc utilities), 81
increment.test.counter (test counter), 115
independence tests, *4*
independence tests (independence-tests), 71
independence-tests, 71
information theoretic quantities, 73
insurance, 74
inter.iamb, *112*
inter.iamb (constraint-based algorithms), 44
intervention (causal inference), 37
isolated.nodes (misc utilities), 81

KL (information theoretic quantities), 73

leaf.nodes (misc utilities), 81
learn.mb (single-node local discovery), 107
learn.nbr (single-node local discovery), 107
learning.test, 76
lizards, 77
lm integration, 78
local discovery algorithms, *36*, *45*, *70*, 79, *107*
logLik.bn (score), 102
logLik.bn.fit (bn.fit utilities), 31
loss (bn.cv), 19

marks, 80
mb (misc utilities), 81
mean.bn.strength (arc.strength), 10
misc utilities, 81
mmhc, *113*
mmhc (hybrid algorithms), 69
mmpc, *113*
mmpc (constraint-based algorithms), 44
model string utilities, 84
model2network (model string utilities), 84
modelstring (model string utilities), 84
modelstring<- (model string utilities), 84
moral (cpdag), 47
multivariate normal distribution, 85
mutilated (causal inference), 37
mvnorm2gbn (multivariate normal distribution), 85

naive.bayes, 86, *89*, *98*
narcs (misc utilities), 81
nbr (misc utilities), 81
network classifiers, *4*
network classifiers (network-classifiers), 89
network scores, *4*
network scores (network-scores), 89
network-classifiers, 89
network-scores, 89
nnodes (misc utilities), 81
node operations, 92
node ordering utilities, 93
node.ordering (node ordering utilities), 93
nodes (misc utilities), 81
nodes,bn-method (misc utilities), 81
nodes,bn.fit-method (misc utilities), 81
nodes,bn.naive-method (misc utilities), 81

nodes,bn.tan-method (misc utilities), 81
nodes<- (node operations), 92
nodes<-,bn-method (node operations), 92
nodes<-,bn.fit-method (node operations), 92
nodes<-,bn.naive-method (node operations), 92
nodes<-,bn.tan-method (node operations), 92
nparams (misc utilities), 81
ntests (misc utilities), 81

ordering2blacklist (utilities for whitelists and blacklists), 115
out.degree (misc utilities), 81
outgoing.arcs (misc utilities), 81

parents (misc utilities), 81
parents<- (misc utilities), 81
path (graph utilities), 61
path,bn-method (graph utilities), 61
path,bn.fit-method (graph utilities), 61
path,bn.naive-method (graph utilities), 61
path,bn.tan-method (graph utilities), 61
path.exists (graph utilities), 61
pc.stable, *112*
pc.stable (constraint-based algorithms), 44
pcalg integration, 94
pdag2dag (graph utilities), 61
perturb (graph utilities), 61
plot.bn, *66*, 95
plot.bn.kcv (bn.cv), 19
plot.bn.strength, 96
predict and impute, 97
predict.bn.fit (predict and impute), 97
predict.bn.naive (naive.bayes), 86
predict.bn.tan (naive.bayes), 86

random.graph (graph generation utilities), 58
rbn, *19*, *22*, 99
read.bif (foreign files utilities), 54
read.dsc (foreign files utilities), 54
read.net (foreign files utilities), 54
remove.node (node operations), 92
rename.nodes (node operations), 92
reset.test.counter (test counter), 115
residuals.bn.fit (bn.fit utilities), 31
reverse.arc (arc operations), 8
reversible.arcs (misc utilities), 81
root.nodes (misc utilities), 81
rsmax2, *113*
rsmax2 (hybrid algorithms), 69

scm class, 100
scm-class (scm class), 100
score, *11–13*, *15*, *24*, *92*, 102, *106*
score,bn-method (score), 102
score,bn.naive-method (score), 102
score,bn.tan-method (score), 102
score-based algorithms, *36*, *45*, *70*, *79*, 105, *111*
set.arc, *26*
set.arc (arc operations), 8
set.edge (arc operations), 8
set2blacklist (utilities for whitelists and blacklists), 115
shd (compare), 41
shielded.colliders (cpdag), 47
si.hiton.pc, *114*
si.hiton.pc (constraint-based algorithms), 44
sid (compare), 41
sigma (bn.fit utilities), 31
single-node local discovery, 107
singular (bn.fit utilities), 31
skeleton (graph utilities), 61
spouses (misc utilities), 81
strength.plot, *13*, *35*, 108
structural.em, 110
structure learning, *4*, *107*
structure learning (structure-learning), 112
structure-learning, 112
subgraph (graph utilities), 61

tabu, *113*
tabu (score-based algorithms), 105
test counter, 115
test.counter (test counter), 115
tiers2blacklist (utilities for whitelists and blacklists), 115
tree.bayes, *89*
tree.bayes (naive.bayes), 86
twin (causal inference), 37

undirected.arcs (misc utilities), 81
unshielded.colliders (cpdag), 47
utilities for whitelists and
blacklists, 115

valid.cpdag (graph utilities), 61
valid.dag (graph utilities), 61
valid.ug (graph utilities), 61
vstructs (cpdag), 47

whitelist (utilities for whitelists
and blacklists), 115
whitelists and blacklists, *4*
whitelists and blacklists
(whitelists-blacklists), 117
whitelists-blacklists, 117
write.bif (foreign files utilities), 54
write.dot (foreign files utilities), 54
write.dsc (foreign files utilities), 54
write.net (foreign files utilities), 54