Package 'bnlearn'

August 5, 2019

Type Package

Title Bayesian Network Structure Learning, Parameter Learning and Inference

Version 4.5

Date 2019-08-04

Depends R (>= 2.14.0), methods

Suggests parallel, graph, Rgraphviz, lattice, gRain, ROCR, Rmpfr, gmp

Author Marco Scutari [aut, cre], Robert Ness [ctb]

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

Description Bayesian network structure learning, parameter learning and inference.
This package implements constraint-based (PC, GS, IAMB, Inter-IAMB, Fast-IAMB, MMPC, Hiton-PC, HPC), pairwise (ARACNE and Chow-Liu), score-based (Hill-Climbing and Tabu Search) and hybrid (MMHC, RSMAX2, H2PC) structure learning algorithms for discrete, Gaussian and conditional Gaussian networks, along with many score functions and conditional independence tests.
The Naive Bayes and the Tree-Augmented Naive Bayes (TAN) classifiers are also implemented.
Some utility functions (model comparison and manipulation, random data generation, arc orientation testing, simple and advanced plots) are included, as well as support for parameter estimation (maximum likelihood and Bayesian) and inference, conditional probability queries, cross-validation, bootstrap and model averaging.
Development snapshots with the latest bugfixes are available from <http://www.bnlearn.com>.

URL http://www.bnlearn.com/

License GPL (>= 2)

LazyData yes

NeedsCompilation yes

Repository CRAN

Date/Publication 2019-08-05 04:20:02 UTC

R topics documented:

bnlearn-package 3

alarm . . . 5
alpha.star . . . 7
arc operations . . . 8
arc.strength . . . 10
asia . . . 13
BF . . . 14
bn class . . . 15
bn.boot . . . 17
bn.cv . . . 18
bn.fit . . . 22
bn.fit class . . . 25
bn.fit plots . . . 27
bn.fit utilities . . . 28
bn.kcv class . . . 30
bn.strength class . . . 31
choose.direction . . . 32
ci.test . . . 33
clgaussian.test . . . 35
compare . . . 35
configs . . . 37
constraint-based algorithms . . . 38
coronary . . . 40
cpdag . . . 41
cpquery . . . 43
ctsdag . . . 46
dsep . . . 47
foreign files utilities . . . 48
gaussian.test . . . 49
gRain integration . . . 50
graph enumeration . . . 51
graph generation utilities . . . 52
graph integration . . . 54
graph utilities . . . 55
graphviz.chart . . . 56
graphviz.plot . . . 58
hailfinder . . . 59
hybrid algorithms . . . 62
impute . . . 64
independence-tests . . . 66
insurance . . . 67
learning.test . . . 69
lizards . . . 70
lm integration . . . 71
local discovery algorithms . . . 72
marks . . . 73
misc utilities . . . 74
model string utilities . . . 77
naive.bayes . . . 78

network-classifiers . . . 80
network-scores . . . 81
node ordering utilities . . . 82
pcalg integration . . . 84
plot.bn . . . 84
plot.bn.strength . . . 86
preprocess . . . 87
rbn . . . 88
ROCR integration . . . 90
score . . . 91
score-based algorithms . . . 94
single-node local discovery . . . 95
strength.plot . . . 96
structural.em . . . 98
structure-learning . . . 100
test counter . . . 102
whitelists-blacklists . . . 103

Index **105**

`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

Package:	bnlearn
Type:	Package
Version:	4.5
Date:	2019-08-04
License:	GPLv2 or later

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, providing the tools needed for learning and working with discrete Bayesian networks, Gaussian Bayesian networks and conditional linear Gaussian Bayesian networks on real-world data. Incomplete data with missing values are also supported. 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 optimization 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, as they use conditional independence tests (usually to reduce the search space) and network scores (to find the optimal network in the reduced space) at the same time.

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 to learn the structure of Bayesian network classifiers are described in network classifiers. All algorithms support the use of whitelists and blacklists to include and exclude arcs from the networks (see whitelists and blacklists); and many have parallel implementation built on the **parallel** package. Bayesian network scores support the use of graphical priors.

Parameter learning approaches include both frequentist and Bayesian estimators. Inference is implemented using approximate algorithms via particle filters approaches such as likelihood weighting, and covers conditional probability queries, prediction and imputation.

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 `http://www.bnlearn.com/bnrepository`.

Use `citation("bnlearn")` to find out how to cite **bnlearn** in publications and other materials; and visit `http://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 <marco.scutari@gmail.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 http://www.bnlearn.com/bnrepository.

Source

Beinlich I, Suermondt HJ, Chavez RM, Cooper GF (1989). "The ALARM Monitoring System: A Case Study with Two Probabilistic Inference Techniques for Belief Networks". *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 value of the imaginary sample size for the BDe score, assuming a uniform prior and given a network structure and a data set.

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 (also known as 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 (i.e. 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 invisibly an updated copy of `x`.

Author(s)

Marco Scutari

Examples

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

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

---

arc.strength *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 = NULL, R = 200, m = nrow(data),
  algorithm, algorithm.args = list(), cpdag = TRUE, debug = FALSE)
# strength of all possible arcs, from a list of custom networks.
custom.strength(networks, nodes, weights = NULL, cpdag = TRUE, debug = FALSE)
# 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, nodes, threshold)
```

Arguments

x	an object of class bn.strength (for mean()) or of class bn (for all other functions).
networks	a list, containing either object of class bn or arc sets (matrices or data frames with two columns, optionally labeled "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. In averaged.network, it defaults to the set of the unique node labels in the strength argument.
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 (PDAG of) the equivalence class is used instead of the network structure itself. It should make it easier to identify score-equivalent arcs.
algorithm	a character string, the 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 to drop the arc from the network. The only possible additional argument is B, the number of permutations to be generated for each permutation test.

If criterion is the label of a score function, the strength is measured by the score gain/loss which would be caused by the arc's removal. In other words, it is the difference between the score of the network 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 aguments depending on the choice of the score, see score for details.

`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).

`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.

`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 boostrap 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 strength (confidence) significance threshold:

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

See Also

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

Examples

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

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

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

## End(Not run)

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

---

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

Description

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

Usage

```
data(asia)
```

Format

The `asia` data set contains the following variables:

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

Note

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

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

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

The complete BN can be downloaded from http://www.bnlearn.com/bnrepository.

Source

Lauritzen S, Spiegelhalter D (1988). "Local Computation with Probabilities on Graphical Structures and their Application to Expert Systems (with discussion)". *Journal of the Royal Statistical Society: Series B*, **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. 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 not currently supported.
...	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 which is equivalent to the ration of the corresponding posterior probabilities if we assume the uniform prior over all possible DAGs. 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 classic 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's never changed afterward.
  - whitelist: a copy of the whitelist argument (a two-column matrix, whose columns are labeled from and to) as transformed by sanitization functions.
  - blacklist: a copy of the blacklist argument (a two-column matrix, whose columns are labeled from and to) as transformed by sanitization 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 newtork 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, bic and bic-g scores.
    * prob: the probability of each arc to be present in a graph generated by the ordered graph generation algorithm.
    * burn.in: the number of iterations for the ic-dag graph generation algorithm to converge to a stationary (and uniform) probability distribution.
    * max.degree: the maximum degree for any node in a graph generated by the ic-dag graph generation algorithm.
    * max.in.degree: the maximum in-degree for any node in a graph generated by the ic-dag graph generation algorithm.
    * max.out.degree: the maximum out-degree for any node in a graph generated by the ic-dag graph generation algorithm.
    * training: a character string, the label of the training node in a Bayesian network classifier.
    * threshold: the threshold used to determine which arcs are significant when averaging network structures.

- nodes: a list. Each element is named after a node and contains the following elements:
  - mb: the Markov blanket of the node (a vector of character strings).
  - nbr: the neighbourhood of the node (a vector of character strings).
  - parents: the parents of the node (a vector of character strings).
  - children: the children of the node (a vector of character strings).
- arcs: the arcs of the Bayesian network (a two-column matrix, whose columns are labeled from and to). Undirected arcs are stored as two directed arcs with opposite directions between the corresponding incident nodes.

Additional (optional) components under learning:

- optimized: whether additional optimizations have been used in the learning algorithm (a boolean value).
- illegal: arcs that are illegal according to the parametric assumptions used to learn the network structure (a two-column matrix, whose columns are labeled 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.

Author(s)

Marco Scutari

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

Description

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

Usage

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

Arguments

`data`	a data frame containing the variables in the model.
`statistic`	a function or a character string (the name of a function) to be applied to each bootstrap replicate.
`R`	a positive integer, the number of bootstrap replicates.
`m`	a positive integer, the size of each bootstrap replicate.
`sim`	a character string indicating the type of simulation required. Possible values are `"ordinary"` (the default, for nonparametric bootstrap) and `"parametric"`.
`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 `statistics.args` are extracted from the list and passed to `statitstics` as the 2nd, 3rd, etc. arguments.

Value

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

Author(s)

Marco Scutari

References

Friedman N, Goldszmidt M, Wyner A (1999). "Data Analysis with Bayesian Networks: A Bootstrap Approach". *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 = "mle", fit.args = list(), method = "k-fold",
  cluster = NULL, debug = FALSE)

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

```
plot(x, ..., main, xlab, ylab, connect = FALSE)

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 networks 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 newtork. 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 medians 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 and the loss function is then computed using that subset. Loss estimates for each of the k subsets are then combined to give an overall loss for data.

- *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 either cross-validation is used with multiple `runs`, the overall loss is the averge 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 element corresponds to one fold and contains the indices for the observations that are included to 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.
- *Gaussian Log-Likelihood Loss* (`logl-g`): the negated expected log-likelihood for Gaussian Bayesian networks.
- *Classification Error* (`pred`): the *prediction error* for a single node in a discrete network. Frequentist predictions are used, so the values of the target node are predicted using only the information present in its local distribution (from its parents).
- *Posterior Classification Error* (`pred-lw` and `pred-lw-cg`): similar to the above, but predictions are computed from an arbitrary set of nodes using likelihood weighting to obtain Bayesian posterior estimates. `pred-lw` applies to discrete Bayesian networks, `pred-lw-cg` to (discrete nodes in) hybrid networks.
- *Exact Classification Error* (`pred-exact`): closed-form exact posterior predictions are available for Bayesian network classifiers.
- *Predictive Correlation* (`cor`): the *correlation* between the observed and the predicted values for a single node in a Gaussian Bayesian network.
- *Posterior Predictive Correlation* (`cor-lw` and `cor-lw-cg`): similar to the above, but predictions are computed from an arbitrary set of nodes using likelihood weighting to obtain Bayesian posterior estimates. `cor-lw` applies to Gaussian networks and `cor-lw-cg` to (continuous nodes in) hybrid networks.

- *Mean Squared Error* (mse): the *mean squared error* between the observed and the predicted values for a single node in a Gaussian Bayesian network.
- *Posterior Mean Squared Error* (mse-lw and mse-lw-cg): similar to the above, but predictions are computed from an arbitrary set of nodes using likelihood weighting to obtain Bayesian posterior estimates. mse-lw applies to Gaussian networks and mse-lw-cg to (continuous nodes in) hybrid networks.

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

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

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

Plotting Results from Cross-Validation

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

Author(s)

Marco Scutari

References

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

See Also

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

Examples

```
bn.cv(learning.test, 'hc', loss = "pred", loss.args = list(target = "F"))

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 the parameters of a Bayesian network conditional on its structure.

Usage

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

Arguments

x	an object of class bn (for bn.fit() and custom.fit()) or an object of class bn.fit (for bn.net).
data	a data frame containing the variables in the model.
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, either mle for *Maximum Likelihood parameter estimation* or bayes for *Bayesian parameter estimation* (currently implemented only for discrete data).

...	additional arguments for the parameter estimation 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.

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 mle, 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 mximum likelihood estimate of NaN for all parents configurations that are not observed in data. Similarly, regression coefficients are set to NA if the linear regressions correspoding to the local distributions of continuous nodes are singular. Such missing values propagate to the results of functions such as predict().

An in-place replacement method is available to change the parameters of each node in a bn.fit object; see the examples for discrete, continuous and hybrid networks below. For a discrete node (class bn.fit.dnode or bn.fit.onode), the new parameters must be in a table object. For a Gaussian node (class bn.fit.gnode), the new parameters can be defined either by an lm, glm or 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 if the discrete parents of the conditional Gaussian node, stored as a factor.

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

Value

bn.fit() 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 (i.e. there are no undirected arcs). See `set.arc` and `pdag2dag` for two ways of manually setting the direction of one or more arcs.

The `bayes` and `mle` methods in `bn.fit()` handle missing values in `data` by estimating the parameters of each local distribution using the observations that are complete for the variables involved (the node and its parents).

When `method` is set to `mle`, `bn.fit()` produces `NA` parameter estimates for discrete and conditional Gaussian nodes when there are (discrete) parents 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

See Also

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

Examples

```
data(learning.test)

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

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

# discrete Bayesian network from expert knowledge.
net = model2network("[A][B][C|A:B]")
cptA = matrix(c(0.4, 0.6), ncol = 2, dimnames = list(NULL, c("LOW", "HIGH")))
```

```
cptB = matrix(c(0.8, 0.2), ncol = 2, dimnames = list(NULL, c("GOOD", "BAD")))
cptC = c(0.5, 0.5, 0.4, 0.6, 0.3, 0.7, 0.2, 0.8)
dim(cptC) = c(2, 2, 2)
dimnames(cptC) = list("C" = c("TRUE", "FALSE"), "A" =  c("LOW", "HIGH"),
                      "B" = c("GOOD", "BAD"))
cfit = custom.fit(net, dist = list(A = cptA, B = cptB, C = cptC))
# for ordinal nodes it is nearly the same.
cfit = custom.fit(net, dist = list(A = cptA, B = cptB, C = cptC),
         ordinal = c("A", "B"))

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

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

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

Description

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

Details

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

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

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

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

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

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

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

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

Author(s)

Marco Scutari

bn.fit plots *Plot fitted Bayesian networks*

Description

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

Usage

```
## for Gaussian Bayesian networks.
bn.fit.qqplot(fitted, xlab = "Theoretical Quantiles",
  ylab = "Sample Quantiles", main, ...)
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 is plotted using relative frequencies, and the matching normal density is added to the plot.
...	additional arguments to be passed to **lattice** functions.

Details

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

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

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

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

Value

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

Author(s)

Marco Scutari

See Also

bn.fit, bn.fit class.

---

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

Description

Assign, extract or compute various quantities of interest from an object of class bn.fit, bn.fit.dnode, 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, ...)
## S3 method for class 'bn.fit'
AIC(object, data, ..., k = 1)
## S3 method for class 'bn.fit'
BIC(object, data, ...)

## methods available for "bn.fit.dnode"
## S3 method for class 'bn.fit.dnode'
coef(object, 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.
nodes	a vector of character strings, the label of a nodes whose log-likelihood components are to be computed.
data	a data frame containing the variables in the model.
...	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 the log-likelihood of each observations in the sample. If FALSE, logLik() returns a single value, the likelihood of the whole sample.
for.parents	a named list in which each element contains

Details

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

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

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

logLik() returns the log-likelihood for the observations in data.

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` or `bn.fit.onode`).

Author(s)

Marco Scutari

See Also

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

Examples

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

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

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

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

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

- loss: a character string, the label of the loss function.
- mean: the mean of the values of the loss function computed in the k iterations of the cross-validation, 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 (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:

- 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 used to determine if a strength coefficient is significant.

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

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

Author(s)

Marco Scutari

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

Description

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

Usage

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

Arguments

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

Details

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

If criterion is a test or a score function, any node connected to one of the nodes in arc by an undirected arc is treated as a parent of that node (with a warning).

Value

choose.direction returns invisibly an updated copy of x.

Author(s)

Marco Scutari

See Also

score, arc.strength.

Examples

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

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

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

## arcs which introduce cycles are handled correctly.
res = set.arc(res, "A", "B")
# now A -> B -> E -> A is a cycle.
choose.direction(res, learning.test, arc = c("E", "A"), debug = TRUE)

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

## End(Not run)
```

---

ci.test *Independence and conditional independence tests*

Description

Perform an independence or a conditional independence test.

Usage

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

Arguments

x	a character string (the name of a variable), a data frame, a numeric vector or a factor object.
y	a character string (the name of another variable), a numeric vector or a factor object.
z	a vector of character strings (the names of the conditioning variables), a numeric vector, a factor object or a data frame. If NULL an independence test will be executed.
data	a data frame containing the variables to be tested.
test	a character string, the label of the conditional independence test to be used in the algorithm. If none is specified, the default test statistic is the *mutual information* for categorical variables, the Jonckheere-Terpstra test for ordered factors and the *linear correlation* for continuous variables. See independence tests for details.

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

Value

An object of class htest containing the following components:

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

Author(s)

Marco Scutari

See Also

choose.direction, arc.strength.

Examples

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

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

clgaussian.test *Synthetic (mixed) data set to test learning algorithms*

Description

This 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 http://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, ...)

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

```
graphviz.compare(x, ..., groups, layout = "dot", shape = "circle", 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 by blacklist are preserved when constructing the CPDAGs of learned and true.
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 dots, 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.
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, se 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 visualize differences between graphs in various way 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, false negative arcs relative to that first network 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, tp.lwd. In addition, it is possible not to plot the reference network at all by setting show.first to FALSE.

Regardless of the visualization, the nodes are arranged to be in the same position for all the networks to make it easier to compare them.

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.

`graphviz.compare()` plots one or more figures and returns invisibly a list containing the `graph` objects produced by **Rgraphviz**.

Note

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

Author(s)

Marco Scutari

References

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

Examples

```
data(learning.test)

e1 = model2network("[A][B][C|A:B][D|B][E|C][F|A:E]")
e2 = model2network("[A][B][C|A:B][D|B][E|C:F][F|A]")
shd(e2, e1, debug = TRUE)
unlist(compare(e1,e2))
compare(target = e1, current = e2, arcs = TRUE)
## Not run: graphviz.compare(e1, e2, diff = "none")
```

---

configs *Construct configurations of discrete variables*

Description

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

Usage

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

Arguments

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

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

```
iamb.fdr(x, cluster = NULL, whitelist = NULL, blacklist = NULL, test = NULL,
  alpha = 0.05, B = NULL, max.sx = NULL, debug = FALSE, undirected = FALSE)
mmpc(x, cluster = NULL, whitelist = NULL, blacklist = NULL, test = NULL,
  alpha = 0.05, B = NULL, max.sx = NULL, debug = FALSE, undirected = TRUE)
si.hiton.pc(x, cluster = NULL, whitelist = NULL, blacklist = NULL, test = NULL,
  alpha = 0.05, B = NULL, max.sx = NULL, debug = FALSE, undirected = TRUE)
hpc(x, cluster = NULL, whitelist = NULL, blacklist = NULL, test = NULL,
  alpha = 0.05, B = NULL, 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 labeled "from" and "to"), containing a set of arcs to be included in the graph.
`blacklist`	a data frame with two columns (optionally labeled "from" and "to"), containing a set of arcs not to be included in the graph.
`test`	a character string, the label of the conditional independence test to be used in the algorithm. If none is specified, the default test statistic is the *mutual information* for 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.
`B`	a positive integer, the number of permutations considered for each permutation test. It will be ignored with a warning if the conditional independence test specified by the `test` argument is not a permutation test.
`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.
`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.

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, hybrid algorithms.

`coronary` *Coronary heart disease data set*

Description

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

Usage

```
data(coronary)
```

Format

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

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

Source

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

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

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

Examples

```
# This is the undirected graphical model from Whittaker (1990).
data(coronary)
ug = empty.graph(names(coronary))
arcs(ug, 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, moral = TRUE, wlbl = FALSE, debug = FALSE)
cextend(x, strict = TRUE, debug = FALSE)
vstructs(x, arcs = FALSE, moral = FALSE, debug = FALSE)
moral(x, 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.
moral	a boolean value. If FALSE we define a v-structure as in Pearl (2000); if TRUE, as in Koller and Friedman (2009). See below.
wlbl	a boolean value. If TRUE arcs whose directions have been fixed by a whitelist or a by 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

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

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

Note that arcs whose directions are dictated by the parametric assumptions of conditional linear Gaussian networks are preserved as directed arcs in cpdag().

Value

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

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

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

Examples

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

cpquery *Perform conditional probability queries*

Description

Perform conditional probability queries (CPQs).

Usage

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

mutilated(x, evidence)
```

Arguments

fitted	an object of class bn.fit.
x	an object of class bn or 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.

mutilated constructs the mutilated network arising from an ideal intervention setting the nodes involved to the values specified by evidence. In this case evidence must be provided as a list in the same format as for likelihood weighting (see below).

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

Value

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

`cpdist()` returns a data frame containing the samples generated from the conditional distribution of the `nodes` conditional on `evidence()`. The data frame has class `c("bn.cpdist","data.frame")`, and a `meth,-8od` attribute storing the value of the `method` gument. In the case of likelihood weighting, the weights are also attached as an attribute called `weights`.

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

Logic Sampling

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

Three tuning parameters are available:

- `n`: a positive integer number, the number of random samples to generate from `fitted`. The default value is `5000 * log10(nparams(fitted))` for discrete and coditional 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 (e.g. `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`, discarding each batch before generating the next.
- `query.nodes`: a a vector of character strings, the labels of the nodes involved in `event` and `evidence`. Simple queries do not require to generate 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 obervations matching `evidence` (out of the `n` that are generated) are returned, and their number depends on the probability of `evidence`.

Likelihood Weighting

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

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

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

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

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

Author(s)

Marco Scutari

References

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

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

Examples

```
## discrete Bayesian network (it is the same with ordinal nodes).
data(learning.test)
fitted = bn.fit(hc(learning.test), learning.test)
# the result should be around 0.025.
cpquery(fitted, (B == "b"), (A == "a"))
# 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))

## ideal interventions and mutilated networks.
mutilated(fitted, evidence = list(F = 42))
```

---

ctsdag *Equivalence classes in the presence of interventions*

Description

Generate the partially directed acyclic graph representing the equivalence class of a Bayesian network learned using interventions.

Usage

```
ctsdag(x, exp, learning = FALSE, debug = FALSE)
```

Arguments

x	an object of class bn, the network from which to compute the PDAG.
exp	a vector of character strings, the labels of the node that are the targets of the interventions. If no targets are provided, ctsdag() just reverts to cpdag().
learning	a boolean value. If TRUE, interventions, whitelists and blacklists used in learning the structure of x will be taken into account in contructing the PDAG. These interventions will be applied in addition to those provided via the exp argument.
debug	a boolean value. If TRUE a lot of debugging output is printed; otherwise the function is completely silent.

Details

ctsdag() extends cpdag() by incorporating interventions in constructing the partially directed acyclic graph that represents the equivalence class of x; it preserves the directions of arcs that are compelled because they are incident on the target nodes specified by the exp argument. This assumes do-calculus model of targeted interventions with no unknown side-effects.

It also takes into account prior arc probabilities used in structure learning, ensuring that DAGs are equivalent in posterior probability only if they are equivalent in prior probability. This is not the case for graph priors other than the uniform (uniform) and marginal uniform priors (marginal, see bn-class for details).

Value

ctsdag returns an object of class bn, representing the equivalence class. See bn-class for details.

Author(s)

Robert Osazuwa Ness

References

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

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

Ness RO, Sachs K, Mallick P, Vitek O (2017). "A Bayesian Active Learning Experimental Design for Inferring Signaling Networks". *International Conference on Research in Computational Molecular Biology*, 134–156.

Tian J, Pearl J (2001). "Causal Discovery from Changes". *Proceedings of the Seventeenth Conference on Uncertainty in Artificial Intelligence*, 512–521.

See Also

cpdag, score.

---

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

Author(s)

Marco Scutari

References

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

---

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

Description

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

Usage

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

Format

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

Note

The R script to generate data from this network is available from http://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, ...)
## S3 method for class 'bn.fit'
as.grain(x)
## S3 method for class 'grain'
as.bn(x, ..., check.cycles = TRUE)
```

Arguments

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

Note

Conditional probability tables in grain objects must be completely specified; on the other hand, bn.fit allows NaN values for unobserved parents' configurations. Such bn.fit objects will be converted to m codegrain 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 with specific characteristics.

Usage

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

Arguments

type	a character string, the label describing the types of graphs to be counted (see below).
nodes	a vector of positive integers, the graph sizes as given by the numbers of nodes.
...	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.
- dags-given-ordering: all directed acyclic graphs with a specific topological ordering.
- dags-with-k-roots: all directed acyclic graphs with k root nodes.
- dags-with-r-arcs: all directed acyclic graphs with r arcs.

Value

count.graphs() returns an objects 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.

Examples

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

## End(Not run)
```

---

graph generation utilities

Generate empty or random graphs

Description

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

Usage

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

Arguments

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

Details

Available graph generation algorithms are:

- *full ordering* based generation (ordered): generates graphs whose node ordering is given by the order of the labels in the nodes 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.
- *empty graphs* (`empty`): generates graphs without any arc.

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

Value

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

Author(s)

Marco Scutari

References

Ide JS, Cozman FG (2002). "Random Generation of Bayesian Networks". *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.

Note

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

Author(s)

Marco Scutari

Examples

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

---

graph utilities *Utilities to manipulate graphs*

Description

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

Usage

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

Arguments

x	an object of class bn. skeleton(), acyclic(), directed() and path() also accept objects of class bn.fit.
from	a character string, the label of a node.
to	a character string, the label of a node (different from from).
direct	a boolean value. If FALSE ignore any arc between from and to when looking for a path.
underlying.graph	a boolean value. If TRUE the underlying undirected graph is used instead of the (directed) one from the x 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.
debug	a boolean value. If TRUE a lot of debugging output is printed; otherwise the function is completely silent.

Value

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

Author(s)

Marco Scutari

References

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

Examples

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

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

---

graphviz.chart *Plotting networks with probability bars*

Description

Plot a Bayesian network as a graph whose nodes are barplots representing the marginal probability distributions of the corresponding variables. Requires the **Rgraphviz** and **gRain** packages.

Usage

```
graphviz.chart(x, type = "barchart", layout = "dot", draw.levels = 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` representing a discrete Bayesian network.
`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 brachart with probabilities printed over the bars).

layout	a character string, the layout argument that will be passed to **Rgraphviz**. Possible values are dots, neato, twopi, circo and fdp. See **Rgraphviz** documentation for details.
draw.levels	a boolean value, whether to print the labels of the levels 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). If grid is a numeric vector, a verical grid is drawn at the specified probabilities.
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 = "circle", 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 dots, 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.
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 labeled 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 `white`.
- `lwd`: a positive number (the line width of highlighted arcs). It overrides the line width settings in `strength.plot()`. The default value is to use the global settings of **Rgraphviz**.
- `lty`: the line type of highlighted arcs. Possible values are 0, 1, 2, 3, 4, 5, 6, "blank", "solid", "dashed", "dotted", "dotdash", "longdash" and "twodash". The default value is to use the global settings of **Rgraphviz**.

Value

`graphviz.plot()` returns invisibly the `graph` object produced by **Rgraphviz**. It can be further modified using the commands present in 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 http://www.bnlearn.com/bnrepository.

Source

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

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 Maximization (RSMAX2) hybrid algorithms.

Usage

```
rsmax2(x, whitelist = NULL, blacklist = NULL, restrict = "si.hiton.pc",
  maximize = "hc", restrict.args = list(), maximize.args = list(), debug = FALSE)
mmhc(x, whitelist = NULL, blacklist = NULL, restrict.args = list(),
  maximize.args = list(), debug = FALSE)
h2pc(x, whitelist = NULL, blacklist = NULL, 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 labeled "from" and "to"), containing a set of arcs to be included in the graph.
`blacklist`	a data frame with two columns (optionally labeled "from" and "to"), containing a set of arcs not to be included in the graph.
`restrict`	a character string, the constraint-based 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 `hpcand` `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.

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, method = "parents", ..., prob = FALSE,
  debug = FALSE)

impute(object, data, method, ..., 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.
method	a character string, the method used to impute the missing values or predict new ones. The default value is parents.
...	additional arguments for the imputation method. See below.
prob	a boolean value. If TRUE and object is a discrete network, the 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

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 that are used to compute the predicted values can be specified with the from optional argument; the default is to use all the 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.

impute() is based on predict(), and can impute missing values with the same methods (parents and bayes-lw). The latter can take an additional argument n with the number of random samples which are 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 of unobserved discrete parents configurations in the data the parameters were learned from), predict will predict NAs when those parents 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 and conditional Gaussian nodes), a factor (for categorical nodes) or an ordered factor (for ordinal nodes). If prob = TRUE and the network 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*, i.e. 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")
```

`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 $2n$ factor) and is related to the deviance of the tested models. The asymptotic χ^2 test (`mi` and `mi-adf`, with adjusted degrees of freedom), the Monte Carlo permutation test (`mc-mi`), the sequential Monte Carlo permutation test (`smc-mi`), and the semiparametric test (`sp-mi`) are implemented.
- *shrinkage estimator* for the *mutual information* (`mi-sh`): an improved asymptotic 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.
- *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 $2n$ factor). The asymptotic χ^2 test (`mi-g`), the Monte Carlo permutation test (`mc-mi-g`) and the sequential Monte Carlo permutation test (`smc-mi-g`) are implemented.
- *shrinkage estimator* for the *mutual information* (`mi-g-sh`): an improved asymptotic χ^2 test based on the James-Stein estimator for the mutual information.

  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.

Available conditional independence tests (and the respective labels) for *hybrid 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 $2n$ factor). Only the asymptotic χ^2 test (`mi-cg`) is implemented.

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

Description

Insurance is a network for evaluating car insurance risks.

Usage

```
data(insurance)
```

Format

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

- `GoodStudent` (*good student*): a two-level factor with levels `False` and `True`.
- `Age` (*age*): a three-level factor with levels `Adolescent`, `Adult` and `Senior`.
- `SocioEcon` (*socio-economic status*): a four-level factor with levels `Prole`, `Middle`, `UpperMiddle` and `Wealthy`.
- `RiskAversion` (*risk aversion*): a four-level factor with levels `Psychopath`, `Adventurous`, `Normal` and `Cautious`.

- `VehicleYear` (*vehicle age*): a two-level factor with levels `Current` and `older`.
- `ThisCarDam` (*damage to this car*): a four-level factor with levels `None`, `Mild`, `Moderate` and `Severe`.
- `RuggedAuto` (*ruggedness of the car*): a three-level factor with levels `EggShell`, `Football` and `Tank`.
- `Accident` (*severity of the accident*): a four-level factor with levels `None`, `Mild`, `Moderate` and `Severe`.
- `MakeModel` (*car's model*): a five-level factor with levels `SportsCar`, `Economy`, `FamilySedan`, `Luxury` and `SuperLuxury`.
- `DrivQuality` (*driving quality*): a three-level factor with levels `Poor`, `Normal` and `Excellent`.
- `Mileage` (*mileage*): a four-level factor with levels `FiveThou`, `TwentyThou`, `FiftyThou` and `Domino`.
- `Antilock` (*ABS*): a two-level factor with levels `False` and `True`.
- `DrivingSkill` (*driving skill*): a three-level factor with levels `SubStandard`, `Normal` and `Expert`.
- `SeniorTrain` (*senior training*): a two-level factor with levels `False` and `True`.
- `ThisCarCost` (*costs for the insured car*): a four-level factor with levels `Thousand`, `TenThou`, `HundredThou` and `Million`.
- `Theft` (*theft*): a two-level factor with levels `False` and `True`.
- `CarValue` (*value of the car*): a five-level factor with levels `FiveThou`, `TenThou`, `TwentyThou`, `FiftyThou` and `Million`.
- `HomeBase` (*neighbourhood type*): a four-level factor with levels `Secure`, `City`, `Suburb` and `Rural`.
- `AntiTheft` (*anti-theft system*): a two-level factor with levels `False` and `True`.
- `PropCost` (*ratio of the cost for the two cars*): a four-level factor with levels `Thousand`, `TenThou`, `HundredThou` and `Million`.
- `OtherCarCost` (*costs for the other car*): a four-level factor with levels `Thousand`, `TenThou`, `HundredThou` and `Million`.
- `OtherCar` (*other cars involved in the accident*): a two-level factor with levels `False` and `True`.
- `MedCost` (*cost of the medical treatment*): a four-level factor with levels `Thousand`, `TenThou`, `HundredThou` and `Million`.
- `Cushioning` (*cushioning*): a four-level factor with levels `Poor`, `Fair`, `Good` and `Excellent`.
- `Airbag` (*airbag*): a two-level factor with levels `False` and `True`.
- `ILiCost` (*inspection cost*): a four-level factor with levels `Thousand`, `TenThou`, `HundredThou` and `Million`.
- `DrivHist` (*driving history*): a three-level factor with levels `Zero`, `One` and `Many`.

Note

The complete BN can be downloaded from http://www.bnlearn.com/bnrepository.

Source

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

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 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 http://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 about the perching behaviour of two species of lizards in the South Bimini island, from Shoener (1968).

Usage

```
data(lizards)
```

Format

The `lizards` data set contains the following variables:

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

Source

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

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

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

Examples

```
# load the data.
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 makes it possible to use 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 graphs structures from data using pairwise mutual information coefficients.

Usage

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

Arguments

x	a data frame containing the variables in the model.
whitelist	a data frame with two columns (optionally labeled "from" and "to"), containing a set of arcs to be included in the graph.
blacklist	a data frame with two columns (optionally labeled "from" and "to"), containing a set of arcs not to be included in the graph.
mi	a character string, the estimator used for the pairwise (i.e. unconditional) mutual information coefficients in the ARACNE and Chow-Liu algorithms. Possible values are `mi` (discrete mutual information) and `mi-g` (Gaussian mutual information).
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.

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)
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)
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

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

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

Arguments

x,object	an object of class bn or bn.fit. The replacement form of parents, children, arcs and amat requires an object of class bn.
node,Nodes	a character string, the label of a node.
value	either a vector of character strings (for parents and children), an adjacency matrix (for amat) or a data frame with two columns (optionally labeled "from" and "to", for arcs).
data	a data frame containing the data the Bayesian network was learned from. It's only needed if x is an object of class bn.
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.
effective	a boolean value. If TRUE the number of non-zero free parameters is returned, that is, the effective degrees of freedom of the network; otherwise the theoretical number of parameters is returned.
debug	a boolean value. If TRUE a lot of debugging output is printed; otherwise the function is completely silent.

Details

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

Value

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

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

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

amat returns a matrix of 0/1 integer values.

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

Author(s)

Marco Scutari

References

Chickering DM (1995). "A Transformational Characterization of Equivalent Bayesian Network Structures". *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)
res = gs(learning.test)

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

model string utilities

Build a model string from a Bayesian network and vice versa

Description

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

Usage

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

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

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

Arguments

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

Details

The strings returned by modelstringi() 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 label of that node. 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)
res = set.arc(gs(learning.test), "A", "B")
res
modelstring(res)
res2 = model2network(modelstring(res))
res2
all.equal(res, res2)
```

---

naive.bayes *Naive Bayes classifiers*

Description

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

Usage

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

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

Arguments

training	a character string, the label of the training variable.
explanatory	a vector of character strings, the labels of the explanatory variables.
object	an object of class bn.naive, either fitted or not.
x, data	a data frame containing the variables in the model, which must all be factors.
prior	a numeric vector, the prior distribution for the training variable. It is automatically normalized if not already so. The default prior is the probability distribution of the training variable in object.
whitelist	a data frame with two columns (optionally labeled "from" and "to"), containing a set of arcs to be included in the graph.
blacklist	a data frame with two columns (optionally labeled "from" and "to"), containing a set of arcs not to be included in the graph.

mi	a character string, the estimator used for the mutual information coefficients for the Chow-Liu algorithm in TAN. Possible values are 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*, i.e. 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. Morevoer it is not possible to whitelist or blacklist arcs incident on training.

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

Author(s)

Marco Scutari

References

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

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

Examples

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

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

# this is an out-of-sample prediction, from a training test to a separate
# test set.
training.set = learning.test[1:4000, ]
test.set = learning.test[4001:5000, ]
bn = naive.bayes(training.set, "A")
fitted = bn.fit(bn, training.set)
pred = predict(fitted, test.set)
table(pred, test.set[, "A"])
```

---

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 assuming that all classifiers are independent and using the posterior probability of the target variable for classification.

- *Tree-Augmented Naive Bayes* (`tree.bayes`): an improvement over naive Bayes, this algorithms uses Chow-Liu 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

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* score (`aic`).
- the *Bayesian Information Criterion* score (`bic`), which is equivalent to the *Minimum Description Length* (MDL) and is also known as *Schwarz Information Criterion*.

  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 *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*, online first.
- 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 *locally averaged Bayesian Dirichlet* score (`bdla`, not score equivalent).
  Cano A, Gomez-Olmedo M, Masegosa AR, Moral S (2013). "Locally Averaged Bayesian Dirichlet Metrics for Learning the Structure and the Parameters of Bayesian Networks". *International Journal of Approximate Reasoning*, **54**:526–540.
- 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.

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* score (`aic-g`).
- the corresponding *Bayesian Information Criterion* 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 *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 that may be subject to 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.

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

- the conditional linear Gaussian *log-likelihood* (`loglik-cg`) score.
- the corresponding *Akaike Information Criterion* score `aic-cg`).
- the corresponding *Bayesian Information Criterion* score (`bic-cg`).
- the *predictive log-likelihood* (`pred-loglik-cg`) computed on a separate test set. The reference paper is the same as that for `pred-loglik`.

`node ordering utilities`

Utilities dealing with partial node orderings

Description

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

Usage

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

Arguments

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

Details

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

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

Value

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

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

Note

node.ordering() and ordering2blacklist() support only completely directed Bayesian networks.

Author(s)

Marco Scutari

Examples

```
data(learning.test)
res = gs(learning.test)
ntests(res)
res = set.arc(res, "A", "B")
ord = node.ordering(res)
ord
```

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

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

---

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)
res = gs(learning.test)

plot(res)

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

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

plot.bn.strength *Plot arc strengths derived from bootstrap*

Description

Plot arc strengths derived from bootstrap resampling.

Usage

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

Arguments

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

preprocess *Pre-process data to better learn Bayesian networks*

Description

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

Usage

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

Arguments

data	a data frame containing numeric columns (for dedup()) or a combination of numeric or factor columns (for discretize()).
threshold	a numeric value between zero and one, the absolute correlation used a threshold in screening highly correlated pairs.
method	a character string, either interval for *interval discretization*, quantile for *quantile discretization* (the default) or hartemink for *Hartemink's pairwise mutual information* method.
breaks	if method is set to hartemink, an integer number, the number of levels the variables are to be discretized into. Otherwise, a vector of integer numbers, one for each column of the data set, specifying the number of levels for each variable.
ordered	a boolean value. If TRUE the discretized variables are returned as ordered factors instead of unordered ones.
...	additional tuning parameters, see below.
debug	a boolean value. If TRUE a lot of debugging output is printed; otherwise the function is completely silent.

Details

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

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

Value

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

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

Note

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

Author(s)

Marco Scutari

References

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

Examples

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

---

rbn *Simulate random samples from a given Bayesian network*

Description

Simulate random samples from a given Bayesian network.

Usage

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

Arguments

`x`	an object of class `bn` or `bn.fit`.
`n`	a positive integer giving the number of observations to generate.
`data`	a data frame containing the data the Bayesian network was learned from.
`fit`	a character string, the label of the method used to fit the parameters of the newtork. See `bn.fit` for details.

...	additional arguments for the parameter estimation prcoedure, see again bn.fit for details.
debug	a boolean value. If TRUE a lot of debugging output is printed; otherwise the function is completely silent.

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 parents configurations in the data the parameters were learned from), rbn will produce samples that contain NAs when those parents 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 (column names and data types) of the data argument (if x is an object of class bn) or with the same structure as the data originally used to to fit the parameters of the Bayesian network (if x is an object of class bn.fit).

Author(s)

Marco Scutari

References

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

See Also

bn.boot, bn.cv.

Examples

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

`ROCR integration` *Generating a prediction object for ROCR*

Description

Evaluate structure learning accuracy with **ROCR**. This function views the arcs in a `bn.strength` object as a set of predictions and the arcs in a `true` reference graph as a set of labels, and produces a `prediction` object from the **ROCR** package. This facilitates evaluation of structure learning with traditional machine learning metrics such as ROC curves and AUC.

Usage

```
## S3 method for class 'bn.strength'
as.prediction(x, true, ..., consider.direction = TRUE)
```

Arguments

`x`	an object of class `bn.strength` returned by `boot.strength()`, representing learning results targeting the object of class `bn` specified by the `true` argument.
`true`	an object of class `bn`, the target of structure learning.
`...`	additional arguments, currently ignored.
`consider.direction`	a boolean value. If `TRUE` an arc's prediction value is set to the product of its `strength` and `direction` values in `x` (interpreted as the probability an arc is both present and has the specified direction). If `FALSE` the arc's prediction value is set to its `strength` value.

Details

One way of evaluating the overall performance of a network structure learning algorithm is to evaluate how well it detects individual arcs. `as.prediction()` takes each pair of nodes in a ground truth network and labels them with a `1` if an arc exists between them and `0` if not. It uses the arc presence probabilities in a `bn.strength` object returned by `boot.strength()` as the predictions.

Value

An object of class `prediction` from the **ROCR** package.

Author(s)

Robert Ness

Examples

```
## Not run:
library(ROCR)

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]")
true.dag = model2network(modelstring)
strength = boot.strength(alarm, R = 200, m = 30, algorithm = "hc")
pred = as.prediction(strength, true.dag)
perf = performance(pred, "tpr", "fpr")
plot(perf, main = "Arc Detection")
performance(pred, "auc")

## End(Not run)
```

---

score *Score of the Bayesian network*

Description

Compute the score of the Bayesian network.

Usage

```
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, bdj). 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).
- l: the number of scores to average in the locally averaged Bayesian Dirichlet score (bdla). The default value is 5.
- exp: a list of indexes of experimental observations (those that have been artificially manipulated). Each element of the list must be named after one of the nodes, and must contain a numeric vector with indexes of the observations whose value has been manipulated for that node.
- k: the penalty coefficient to be used by the AIC and BIC scores. The default value is 1 for AIC and log(nrow(data))/2 for BIC.
- prior: the prior distribution to be used with the various Bayesian Dirichlet scores (bde, mbde, bds, bdj, bdla) 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) and cs (the Castelo & Siebes prior, which puts an independent prior probability on each arc and direction).
- beta: the parameter associated with prior.
  - If prior is uniform, beta is ignored.
  - If prior is vsp, beta is the probability of inclusion of an additional parent. The default is 1/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.
- 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.

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 probabilties 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

`choose.direction`, `arc.strength`, `alpha.star`.

Examples

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

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

## K2 score on the other hand is not score equivalent.
score(res, learning.test, type = "k2")
score(res2, 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(res, learning.test, type = "bde", prior = "cs", beta = beta)

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

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

score-based algorithms

Score-based structure learning algorithms

Description

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

Usage

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

Arguments

x	a data frame containing the variables in the model.
start	an object of class bn, the preseeded directed acyclic graph used to initialize the algorithm. If none is specified, an empty one (i.e. without any arc) is used.
whitelist	a data frame with two columns (optionally labeled "from" and "to"), containing a set of arcs to be included in the graph.
blacklist	a data frame with two columns (optionally labeled "from" and "to"), containing a set of arcs not to be included in the graph.
score	a character string, the label of the network score to be used in the algorithm. If none is specified, the default score is the *Bayesian Information Criterion* for both discrete and continuous data sets. See 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 for a node. 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

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

---

single-node local discovery

Discover the structure around a single node

Description

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

Usage

```
learn.mb(x, node, method, whitelist = NULL, blacklist = NULL, start = NULL,
  test = NULL, alpha = 0.05, B = NULL, max.sx = NULL, debug = FALSE)
learn.nbr(x, node, method, whitelist = NULL, blacklist = NULL,
  test = NULL, alpha = 0.05, B = NULL, 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.
B	a positive integer, the number of permutations considered for each permutation test. It will be ignored with a warning if the conditional independence test specified by the test argument is not a permutation test.
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()).

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

Non-significant arcs are plotted as dashed lines.

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

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

if the coefficients are computed from conditional independence tests, by

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

for bootstrap estimates or by the quantiles

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

of the significant coefficients if network scores are used.

Value

`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 gs().
res = set.arc(gs(learning.test), "A", "B")
strength = arc.strength(res, learning.test, criterion = "x2")
\dontrun{strength.plot(res, strength)}
# add another (non-significant) arc and plot the network again.
res = set.arc(res, "A", "C")
strength = arc.strength(res, learning.test, criterion = "x2")
\dontrun{strength.plot(res, 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 = "mle",
    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 "maximization" 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 "maximization" 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 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 *maximization* 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.

`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 implementationi, 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 optimized implementation uses score caching, score decomposability and score equivalence to reduce the number of duplicated tests.
- *Tabu Search* (`tabu`): a modified hill-climbing able to escape local optima by selecting a network that minimally decreases the score function.

  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 Maximization* (`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 maximization 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.

---

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()
```

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

∗Topic **classes**
bn class, 15
bn.fit class, 25
bn.kcv class, 30
bn.strength class, 31
∗Topic **classifiers**
naive.bayes, 78
∗Topic **convenience functions**
bn.fit utilities, 28
choose.direction, 32
configs, 37
misc utilities, 74
model string utilities, 77
node ordering utilities, 82
test counter, 102
∗Topic **data preprocessing**
configs, 37
impute, 64
preprocess, 87
∗Topic **datasets**
alarm, 5
asia, 13
clgaussian.test, 35
coronary, 40
gaussian.test, 49
hailfinder, 59
insurance, 67
learning.test, 69
lizards, 70
marks, 73
∗Topic **graphs**
arc operations, 8
compare, 35
cpdag, 41
ctsdag, 46
dsep, 47
graph enumeration, 51
graph utilities, 55
misc utilities, 74
node ordering utilities, 82
∗Topic **import/export to file**
foreign files utilities, 48
∗Topic **independence tests**
arc.strength, 10
choose.direction, 32
ci.test, 33
∗Topic **inference**
cpquery, 43
impute, 64
rbn, 88
∗Topic **interfaces to other packages**
bn.fit plots, 27
compare, 35
gRain integration, 50
graph integration, 54
graphviz.chart, 56
graphviz.plot, 58
lm integration, 71
pcalg integration, 84
ROCR integration, 90
strength.plot, 96
∗Topic **local learning**
local discovery algorithms, 72
single-node local discovery, 95
∗Topic **missing data**
impute, 64
structural.em, 98
∗Topic **network scores**
alpha.star, 7
arc.strength, 10
BF, 14
bn.fit utilities, 28
choose.direction, 32
score, 91
∗Topic **package**
bnlearn-package, 3
independence-tests, 66
network-classifiers, 80

network-scores, 81
structure-learning, 100
whitelists-blacklists, 103
∗Topic **parameter learning**
bn.fit, 22
∗Topic **plots**
bn.fit plots, 27
compare, 35
graphviz.chart, 56
graphviz.plot, 58
plot.bn, 84
plot.bn.strength, 86
strength.plot, 96
∗Topic **resampling**
bn.boot, 17
bn.cv, 18
choose.direction, 32
∗Topic **simulation**
cpquery, 43
graph generation utilities, 52
rbn, 88
∗Topic **structure learning**
alpha.star, 7
arc.strength, 10
BF, 14
constraint-based algorithms, 38
cpdag, 41
hybrid algorithms, 62
naive.bayes, 78
node ordering utilities, 82
score-based algorithms, 94
structural.em, 98
∗Topic **utilities**
gRain integration, 50
graph integration, 54
pcalg integration, 84
$<-.bn.fit *(bn.fit)*, 22

acyclic *(graph utilities)*, 55
AIC.bn *(score)*, 91
AIC.bn.fit *(bn.fit utilities)*, 28
alarm, 5
all.equal.bn *(compare)*, 35
alpha.star, 7, *93*, *95*
amat *(misc utilities)*, 74
amat<- *(misc utilities)*, 74
ancestors *(misc utilities)*, 74
aracne, *102*
aracne *(local discovery algorithms)*, 72
arc operations, 8
arc.strength, 10, *31*, *32*, *34*, *93*
arcs *(misc utilities)*, 74
arcs<- *(misc utilities)*, 74
as.bn *(model string utilities)*, 77
as.bn.fit *(gRain integration)*, 50
as.bn.grain *(gRain integration)*, 50
as.bn.graphAM *(graph integration)*, 54
as.bn.graphNEL *(graph integration)*, 54
as.bn.pcAlgo *(pcalg integration)*, 84
as.character.bn *(model string utilities)*, 77
as.grain *(gRain integration)*, 50
as.graphAM *(graph integration)*, 54
as.graphNEL *(graph integration)*, 54
as.lm *(lm integration)*, 71
as.prediction *(ROCR integration)*, 90
asia, 13
averaged.network, *11*
averaged.network *(arc.strength)*, 10

BF, *11*, 14
bf.strength, *12*, *15*
bf.strength *(arc.strength)*, 10
BIC.bn *(score)*, 91
BIC.bn.fit *(bn.fit utilities)*, 28
blacklist *(misc utilities)*, 74
bn class, 15, *23*
bn-class *(bn class)*, 15
bn.boot, 17, *21*, *89*
bn.cv, *18*, 18, *89*
bn.fit, *19*, 22, *28*, *30*, *65*, *88*, *89*, *99*, *100*
bn.fit class, *23*, 25, *28*
bn.fit plots, *24*, 27
bn.fit utilities, *24*, 28
bn.fit-class *(bn.fit class)*, 25
bn.fit.barchart *(bn.fit plots)*, 27
bn.fit.dnode *(bn.fit class)*, 25
bn.fit.dotplot *(bn.fit plots)*, 27
bn.fit.gnode *(bn.fit class)*, 25
bn.fit.histogram *(bn.fit plots)*, 27
bn.fit.qqplot *(bn.fit plots)*, 27
bn.fit.xyplot *(bn.fit plots)*, 27
bn.kcv class, 30
bn.kcv-class *(bn.kcv class)*, 30
bn.kcv.list class *(bn.kcv class)*, 30
bn.kcv.list-class *(bn.kcv class)*, 30
bn.net *(bn.fit)*, 22
bn.strength *(bn.strength class)*, 31

bn.strength class, *12*, 31
bn.strength-class (bn.strength class), 31
bnlearn (bnlearn-package), 3
bnlearn-package, 3
boot.strength, *12*, *31*
boot.strength (arc.strength), 10

cextend (cpdag), 41
children (misc utilities), 74
children<- (misc utilities), 74
choose.direction, *12*, 32, *34*, *93*
chow.liu, *102*
chow.liu (local discovery algorithms), 72
ci.test, *12*, 33
clgaussian.test, 35
coef.bn.fit (bn.fit utilities), 28
compare, *15*, 35
compelled.arcs (misc utilities), 74
configs, 37
constraint-based algorithms, 38, *63*, *72*, *95*, *96*
coronary, 40
count.graphs (graph enumeration), 51
cpdag, 41, *47*
cpdist (cpquery), 43
cpquery, 43
ctsdag, 46
custom.fit (bn.fit), 22
custom.strength, *12*, *31*
custom.strength (arc.strength), 10

dedup (preprocess), 87
degree (misc utilities), 74
degree,bn-method (misc utilities), 74
degree,bn.fit-method (misc utilities), 74
degree,bn.naive-method (misc utilities), 74
degree,bn.tan-method (misc utilities), 74
descendants (misc utilities), 74
directed (graph utilities), 55
directed.arcs (misc utilities), 74
discretize (preprocess), 87
drop.arc (arc operations), 8
drop.edge (arc operations), 8
dsep, 47

em-based algorithms (structural.em), 98
empty.graph (graph generation utilities), 52

fast.iamb, *100*
fast.iamb (constraint-based algorithms), 38
fitted.bn.fit (bn.fit utilities), 28
foreign files utilities, 48

gaussian.test, 49
gRain integration, 50
graph enumeration, 51
graph generation utilities, 52
graph integration, 54
graph utilities, 55
graphviz.chart, 56
graphviz.compare (compare), 35
graphviz.plot, 58, *85*, *97*
gs, *100*
gs (constraint-based algorithms), 38

h2pc, *101*
h2pc (hybrid algorithms), 62
hailfinder, 59
hamming (compare), 35
hc, *101*
hc (score-based algorithms), 94
hpc, *102*
hpc (constraint-based algorithms), 38
hybrid algorithms, *40*, 62, *72*, *95*

iamb, *100*
iamb (constraint-based algorithms), 38
iamb.fdr, *100*
impute, 64, *99*, *100*
in.degree (misc utilities), 74
incident.arcs (misc utilities), 74
incoming.arcs (misc utilities), 74
increment.test.counter (test counter), 102
independence tests, *4*, *32*, *33*, *39*, *96*
independence tests (independence-tests), 66
independence-tests, 66
insurance, 67
inter.iamb, *100*
inter.iamb (constraint-based algorithms), 38

leaf.nodes (misc utilities), 74
learn.mb (single-node local discovery), 95
learn.nbr (single-node local discovery), 95
learning.test, 69
lizards, 70
lm integration, 71
local discovery algorithms, *40*, *63*, 72, *95*
logLik.bn (score), 91
logLik.bn.fit (bn.fit utilities), 28
loss (bn.cv), 18

marks, 73
mb (misc utilities), 74
mean.bn.strength (arc.strength), 10
misc utilities, 74
mmhc, *101*
mmhc (hybrid algorithms), 62
mmpc, *101*
mmpc (constraint-based algorithms), 38
model string utilities, 77
model2network (model string utilities), 77
modelstring (model string utilities), 77
modelstring<- (model string utilities), 77
moral (cpdag), 41
mutilated (cpquery), 43

naive.bayes, *65*, 78, *80*
narcs (misc utilities), 74
nbr (misc utilities), 74
network classifiers, *4*, *79*
network classifiers (network-classifiers), 80
network scores, *4*, *11*, *32*, *91*, *94*
network scores (network-scores), 81
network-classifiers, 80
network-scores, 81
nnodes (misc utilities), 74
node ordering utilities, 82
node.ordering (node ordering utilities), 82
nodes (misc utilities), 74
nodes,bn-method (misc utilities), 74
nodes,bn.fit-method (misc utilities), 74
nodes,bn.naive-method (misc utilities), 74
nodes,bn.tan-method (misc utilities), 74
nodes<- (misc utilities), 74
nodes<-,bn-method (misc utilities), 74
nodes<-,bn.fit-method (misc utilities), 74
nodes<-,bn.naive-method (misc utilities), 74
nodes<-,bn.tan-method (misc utilities), 74
nparams (misc utilities), 74
ntests (misc utilities), 74

ordering2blacklist (node ordering utilities), 82
out.degree (misc utilities), 74
outgoing.arcs (misc utilities), 74

parents (misc utilities), 74
parents<- (misc utilities), 74
path (graph utilities), 55
pc.stable, *100*
pc.stable (constraint-based algorithms), 38
pcalg integration, 84
pdag2dag, *24*
pdag2dag (graph utilities), 55
plot.bn, *59*, 84
plot.bn.kcv (bn.cv), 18
plot.bn.strength, 86
predict.bn.fit (impute), 64
predict.bn.naive (naive.bayes), 78
predict.bn.tan (naive.bayes), 78
preprocess, 87

random.graph (graph generation utilities), 52
rbn, *18*, *21*, 88
read.bif (foreign files utilities), 48
read.dsc (foreign files utilities), 48
read.net (foreign files utilities), 48
reset.test.counter (test counter), 102
residuals.bn.fit (bn.fit utilities), 28
reverse.arc (arc operations), 8
reversible.arcs (misc utilities), 74
ROCR integration, 90
root.nodes (misc utilities), 74
rsmax2, *101*
rsmax2 (hybrid algorithms), 62

score, *11*, *12*, *14*, *15*, *23*, *32*, *47*, 91, *94*

score-based algorithms, *40*, *63*, *72*, 94, *100*
set.arc, *24*
set.arc *(*arc operations*)*, 8
set.edge *(*arc operations*)*, 8
shd *(*compare*)*, 35
si.hiton.pc, *101*
si.hiton.pc *(*constraint-based algorithms*)*, 38
sigma *(*bn.fit utilities*)*, 28
single-node local discovery, 95
skeleton *(*graph utilities*)*, 55
spouses *(*misc utilities*)*, 74
strength.plot, *12*, *31*, 96
structural.em, 98
structure learning, *4*, *11*, *17*, *39*, *63*, *95*, *99*
structure learning *(*structure-learning*)*, 100
structure-learning, 100
subgraph *(*graph utilities*)*, 55

tabu, *101*
tabu *(*score-based algorithms*)*, 94
test counter, 102
test.counter *(*test counter*)*, 102
tiers2blacklist *(*node ordering utilities*)*, 82
tree.bayes, *81*
tree.bayes *(*naive.bayes*)*, 78

undirected.arcs *(*misc utilities*)*, 74

vstructs *(*cpdag*)*, 41

whitelist *(*misc utilities*)*, 74
whitelists and blacklists, *4*
whitelists and blacklists *(*whitelists-blacklists*)*, 103
whitelists-blacklists, 103
write.bif *(*foreign files utilities*)*, 48
write.dot *(*foreign files utilities*)*, 48
write.dsc *(*foreign files utilities*)*, 48
write.net *(*foreign files utilities*)*, 48