

Package: Peanut (via r-universe)

June 11, 2024

Version 0.9-3

Date 2023/08/20

Title Parameterized Bayesian Networks, Abstract Classes

Author Russell Almond

Maintainer Russell Almond <ralmond@fsu.edu>

Depends R (>= 3.0), CPTtools (>= 0.5), methods

Imports utils, futile.logger, jsonlite

Suggests PNetica, knitr, rmarkdown, tidy, htmltools, shiny, shinyjs

VignetteBuilder knitr

Description This provides support of learning conditional probability tables parameterized using CPTtools. This provides an object oriented layer on top of a CPTtools, to facilitate calculations with Parameterized models for Bayesian networks. Peanut is a collection of abstract classes and generic functions defining a protocol, with the intent that the protocol can be implemented with different Bayes net engines. The companion package PNetica provides an implementation using Netica and RNetica.

License Artistic-2.0

URL <http://pluto.coe.fsu.edu/RNetica>

Support c('Bill & Melinda Gates Foundation grant `` Games as Learning/Assessment: Stealth Assessment' (#0PP1035331, Val Shute, PI)', 'National Science Foundation grant `` DIP: Game-based Assessment and Support of STEM-related Competencies' (#1628937, Val Shute, PI)', 'National Science Foundation grant `` Mathematical Learning via Architectural Design and Modeling Using E-Rebuild.' (#1720533, Fengfeng Ke, PI)', 'Institute of Educational Statistics Grant: `` Exploring adaptive cognitive and affective learning support for next-generation STEM learning games.' (#R305A170376-20, Val Shute and Russell Almond, PIs')

Repository <https://ralmond.r-universe.dev>

RemoteUrl <https://github.com/ralmond/Peanut>

RemoteRef HEAD

RemoteSha d6c4b8d0dd5286ab0e6332e286df5bd8f038676d

Contents

Peanut-package	3
BuildNetManifest	6
BuildNodeManifest	8
BuildTable	12
calcExpTables	14
calcPnetLLike	16
CompensatoryGadget	18
DPCGadget	20
flattenStats	23
flog.try	24
GEMfit	25
is.legal.name	30
isPnodeContinuous	31
maxAllTableParams	33
OffsetGadget	36
Omega2Pnet	38
Pnet	43
Pnet-class	46
Pnet2Omega	48
Pnet2Qmat	52
PnetAdjoin	58
PnetCompile	61
PnetFindNode	62
PnetHub	64
PnetMakeStubNodes	65
PnetName	67
PnetPathname	69
PnetPnodes	70
PnetPriorWeight	72
PnetSerialize	75
PnetTitle	78
PnetWarehouse-class	79
Pnode	82
Pnode-class	85
PnodeBetas	87
PnodeDefaultAlphas	91
PnodeEvidence	92
PnodeLabels	95
PnodeLink	96
PnodeLinkScale	99
PnodeLnAlphas	101
PnodeMin	105

PnodeName	107
PnodeParents	109
PnodeParentTvals	111
PnodePostWeight	113
PnodeProbs	115
PnodeQ	117
PnodeRules	120
PnodeStates	122
PnodeStateTitles	124
PnodeStateValues	126
PnodeStats	128
PnodeTitle	130
PnodeWarehouse-class	131
Qmat2Pnet	134
RegressionGadget	141
Statistic	143
Statistic-class	146
topsort	147
Warehouse	148
WarehouseCopy	152
WarehouseManifest	154

Index**158**

Peanut-package

*Parameterized Bayesian Networks, Abstract Classes***Description**

This provides support of learning conditional probability tables parameterized using CPTtools. This provides an object oriented layer on top of a CPTtools, to facilitate calculations with Parameterized models for Bayesian networks. Peanut is a collection of abstract classes and generic functions defining a protocol, with the intent that the protocol can be implemented with different Bayes net engines. The companion package PNetica provides an implementation using Netica and RNetica.

Details

The DESCRIPTION file: This package was not yet installed at build time.

Peanut (a corruption of Parameterized network or Pnet) is an object oriented layer on top of the tools for constructing conditional probability tables for Bayesian networks in the CPTtools package. In particular, it defines a Pnode (parameterized node) object which stores all of the arguments necessary to use to the calcDPCTable function to build the conditional probability table for the node.

The Pnet object is a Bayesian network containing a number of Pnodes. It supports two key operations, BuildAllTables which sets the values of the conditional probabilities based on current parameters and GEMfit which adjusts the parameters to match a set of cases.

Like the [DBI](#) package, this class consists mostly of generic functions which need to be implemented for specific Bayes net implementations. The package [PNetica](#) provides an implementation of the Peanut generic functions using the [RNetica](#) package. All of the Netica-dependent code is isolated in the [PNetica](#) package, to make it easier to create other implementations.

Index

Index: This package was not yet installed at build time.

Author(s)

Russell Almond

Maintainer: Russell Almond <ralmond@fsu.edu>

References

Almond, R. G. (2015) An IRT-based Parameterization for Conditional Probability Tables. Paper presented at the 2015 Bayesian Application Workshop at the Uncertainty in Artificial Intelligence Conference.

Almond, R. G., Mislevy, R. J., Steinberg, L. S., Yan, D. and Williamson, D. M. (2015) *Bayesian Networks in Educational Assessment*. Springer. (ISBN 978-1-4939-2124-9).

See Also

[PNetica](#) An implementation of the Peanut object model using [RNetica](#).

[CPTtools](#) A collection of implementation independent Bayes net utilities.

Examples

```
## Not run:
library(PNetica) ## Requires implementation
sess <- NeticaSession()
startSession(sess)

## Building CPTs
tNet <- CreateNetwork("TestNet", session=sess)

theta1 <- NewDiscreteNode(tNet, "theta1",
                          c("VH", "High", "Mid", "Low", "VL"))
PnodeStateValues(theta1) <- effectiveThetas(NodeNumStates(theta1))
NodeProbs(theta1) <- rep(1/NodeNumStates(theta1), NodeNumStates(theta1))
theta2 <- NewDiscreteNode(tNet, "theta2",
                          c("VH", "High", "Mid", "Low", "VL"))
PnodeStateValues(theta2) <- effectiveThetas(NodeNumStates(theta2))
NodeProbs(theta2) <- rep(1/NodeNumStates(theta2), NodeNumStates(theta2))

partial3 <- NewDiscreteNode(tNet, "partial3",
                            c("FullCredit", "PartialCredit", "NoCredit"))
```

```

PnodeParents(partial3) <- list(theta1,theta2)

partial3 <- Pnode(partial3,Q=TRUE, link="partialCredit")
PnodePriorWeight(partial3) <- 10
BuildTable(partial3)

## Set up so that first skill only needed for first transition, second
## skill for second transition; adjust alphas to match
PnodeQ(partial3) <- matrix(c(TRUE,TRUE,
                             TRUE,FALSE), 2,2, byrow=TRUE)
PnodeLnAlphas(partial3) <- list(FullCredit=c(-.25,.25),
                               PartialCredit=0)

BuildTable(partial3)
partial4 <- NewDiscreteNode(tNet,"partial4",
                           c("Score4","Score3","Score2","Score1"))
NodeParents(partial4) <- list(theta1,theta2)
partial4 <- Pnode(partial4, link="partialCredit")
PnodePriorWeight(partial4) <- 10

## Skill 1 used for first transition, Skill 2 used for second
## transition, both skills used for the 3rd.

PnodeQ(partial4) <- matrix(c(TRUE,TRUE,
                             FALSE,TRUE,
                             TRUE,FALSE), 3,2, byrow=TRUE)
PnodeLnAlphas(partial4) <- list(Score4=c(.25,.25),
                               Score3=0,
                               Score2=-.25)

BuildTable(partial4)

## Fitting Model to data

irt10.base <- ReadNetworks(system.file("testnets", "IRT10.2PL.base.dne",
                                       package="PNetica"),
                          session=sess)
irt10.base <- as.Pnet(irt10.base) ## Flag as Pnet, fields already set.
irt10.theta <- PnetFindNode(irt10.base,"theta")
irt10.items <- PnetPnodes(irt10.base)
## Flag items as Pnodes
for (i in 1:length(irt10.items)) {
  irt10.items[[i]] <- as.Pnode(irt10.items[[i]])
  ## Add node to list of observed nodes
  PnodeLabels(irt10.items[[1]]) <-
    union(PnodeLabels(irt10.items[[1]]),"onodes")
}

casepath <- system.file("testdat", "IRT10.2PL.200.items.cas", package="PNetica")

BuildAllTables(irt10.base)
PnetCompile(irt10.base) ## Netica requirement

```

```

item1 <- irt10.items[[1]]
priB <- PnodeBetas(item1)
priA <- PnodeAlphas(item1)
pricPT <- NodeProbs(item1)

gemout <- GEMfit(irt10.base,casepath)

DeleteNetwork(irt10.base)
DeleteNetwork(tNet)
stopSession(sess)

## End(Not run)

```

BuildNetManifest	<i>Builds a network manifest from a list of Pnets</i>
------------------	---

Description

A network manifest is a table of meta data about a collection of networks. Each line corresponds to the specific network. This manifest can be used by a network warehouse ([Warehouse](#)) to recreate the network on demand.

Usage

```
BuildNetManifest(Pnetlist)
```

Arguments

Pnetlist A list of [Pnet](#) objects which will appear in the network manifest.

Details

A network manifest is a table (data frame) which describes a collection of networks. It contains meta-data about the networks, and not the information about the nodes, contained in the node manifest ([BuildNodeManifest](#)) or the relationships between the nodes which is contained in the Q -matrix ([Pnet2Qmat](#)) or the Ω -Matrix ([Pnet2Omega](#)). The role of the net manifest is to be used as to create a Net [Warehouse](#) which is an argument to the [Qmat2Pnet](#) and [Omega2Pnet](#) commands, creating networks as they are referenced.

The “Name” column of the table contains the network name and is a key to the table (so it should be unique). It corresponds to [PnetName](#). The “Title” ([PnetTitle](#)) and “Description” ([PnetDescription](#)) columns contain optional meta-data about the node. The “Pathname” ([PnetPathname](#)) column contains the location of the file to which the network should be written and from which it can be read. The “Hub” ([PnetHub](#)) is for spoke models (evidence models) some of whose variables are defined in a hub network. This the network in question is meant to be a spoke, then this field points at the corresponding hub.


```

## Not run:
library(PNetica) ## Example requires PNetica

sess <- NeticaSession()
startSession(sess)

netpath <- System.file("testnets",package="PNetica")
netnames <- paste(c("miniPP-CM", "PPcompEM", "PPconjEM", "PPtwostepEM",
  "PPdurAttEM"), "dne", sep=".")

Nets <- ReadNetworks(file.path(netpath,netnames),
  session=sess)

netman <- BuildNetManifest(Nets)
stopifnot(all.equal(netman,netman1))

## BNWarehouse is the PNetica Net Warehouse.
Nethouse <- BNWarehouse(manifest=netman1,session=sess,key="Name")

stopSession(sess)

## End(Not run)

```

BuildNodeManifest	<i>Builds a table describing a set of Pnodes</i>
-------------------	--

Description

A node manifest is a table where each line describes one state of a node in a Bayesian network. As a node manifest may contain nodes from more than one network, the key for the table is the first two columns: “Model” and “NodeName”. The primary purpose is that this can be given to a [Node Warehouse](#) to create nodes on demand.

Usage

```
BuildNodeManifest(Pnodelist)
```

Arguments

Pnodelist A list of [Pnode](#) objects from which the table will be built.

Details

A node manifest is a table (data frame) which describes a collection of nodes. It contains mostly meta-data about the nodes, and not the information about the relationships between the nodes which is contained in the Q -matrix ([Pnet2Qmat](#)) or the Ω -Matrix ([Pnet20mega](#)). The role of the node manifest is to be used as to create a [Node Warehouse](#) which is an argument to the [Qmat2Pnet](#) and

[Omega2Pnet](#) commands, creating nodes as they are referenced. Hence it contains the information about the node which is not part of the Q or Ω matrix.

The Q -matrix can span multiple Bayesian networks. The same variable can appear with the same name but slightly different definitions in two different networks. Consequently, the key for this table is the “Model” and “NodeName” columns (usually the first two). The function [WarehouseData](#) when applied to a node warehouse should have a key of length 2 (model and node name) and will return multiple lines, one line corresponding to each state of the data frame.

The columns “ModelHub”, “NodeTitle”, “NodeDescription” and “NodeLabels” provide meta-data about the node. They may be missing empty strings, indicating that meta-data is unavailable.

The columns “Nstates” and “StateName” are required. The number of states should be an integer (2 or greater) and there should be as many rows with this model and node name as there are states. Each should have a unique value for “StateName”. The “StateTitle”, “StateDescription” and “StateValue” are optional, although if the variable is to be used as a parent variable, it is strongly recommended to set the state values.

Value

An object of class [data.frame](#) with the following columns.

Node-level Key Fields:

Model	A character value giving the name of the Bayesian network to which this node belongs. Corresponds to the value of PnodeNet .
NodeName	A character value giving the name of the node. All rows with the same value in the model and node name columns are assumed to reference the same node. Corresponds to the value of PnodeName .

Node-level Fields:

ModelHub	If this is a spoke model (meant to be attached to a hub) then this is the name of the hub model (i.e., the name of the proficiency model corresponding to an evidence model). Corresponds to the value of PnetHub(PnodeNet(node)) .
NodeTitle	A character value containing a slightly longer description of the node, unlike the name this is not generally restricted to variable name formats. Corresponds to the value of PnodeTitle .
NodeDescription	A character value describing the node, meant for human consumption (documentation). Corresponds to the value of PnodeDescription .
NodeLabels	A comma separated list of identifiers of sets which this node belongs to. Used to identify special subsets of nodes (e.g., high-level nodes or observable nodes). Corresponds to the value of PnodeLabels .

State-level Key Fields:

Continuous	A logical value. If true, the variable will be continuous, with states corresponding to ranges of values. If false, the variable will be discrete, with named states.
Nstates	The number of states. This should be an integer greater than or equal to 2. Corresponds to the value of PnodeNumStates .

StateName The name of the state. This should be a string value and it should be different for every row within the subset of rows corresponding to a single node. Corresponds to the value of [PnodeStates](#).

State-level Fields:

StateTitle A longer name not subject to variable naming restrictions. Corresponds to the value of [PnodeStateTitles](#).

StateDescription A human readable description of the state (documentation). Corresponds to the value of [PnodeStateDescriptions](#).

StateValue A real numeric value assigned to this state. [PnodeStateValues](#). Note that this has different meaning for discrete and continuous variables. For discrete variables, this associates a numeric value with each level, which is used in calculating the [PnodeEAP](#) and [PnodeSD](#) functions. In the continuous case, this value is ignored and the midpoint between the “LowerBounds” and “UpperBounds” are used instead.

LowerBound This servers as the lower bound for each partition of the continuous variagle. -Inf is a legal value for the first or last row.

UpperBound This is only used for continuous variables, and the value only is needed for one of the states. This servers as the upper bound of range each state. Note the upper bound needs to match the lower bounds of the next state. Inf is a legal value for the first or last row.

Logging

BuildNodeManifest uses the [flog.logger](#) mechanism to log progress. To see progress messages, use [flog.threshold](#)(DEBUG) (or TRACE).

Continuous Variables

Peanut (following Netica) treats continuous variables as discrete variables whose states correspond to ranges of an underlying continuous variable. Unfortunately, this overlays the meaning of [PnodeStateValues](#), and consequently the “StateValue” column.

Discrete Variables. The states of the discrete variables are defined by the “StateName” fields. If values are supplied in “StateValue”, then these values are used in calculating expected a posteriori statistics, [PnodeEAP\(\)](#) and [PnodeSD\(\)](#). The “LowerBound” and “UpperBound” fields are ignored.

Continuous Variables. The states of the continuous variable are defined by breaking the range up into a series of intervals. Right now the intervals must be adjacent (the upper bound of one must match the lower bound of the next) and cannot overlap. This is done by supplying a “LowerBound” and “UpperBound” for each state. If the upper and lower bounds do not match, then an error is signaled.

Author(s)

Russell Almond

References

Almond, R. G. (presented 2017, August). Tabular views of Bayesian networks. In John-Mark Agosta and Tomas Singlair (Chair), *Bayesian Modeling Application Workshop 2017*. Symposium conducted at the meeting of Association for Uncertainty in Artificial Intelligence, Sydney, Australia. (International) Retrieved from <http://bmaw2017.azurewebsites.net/>

See Also

Node functions called to find node meta-data: [PnodeName](#), [PnodeTitle](#), [PnodeNet](#), [PnetHub](#), [PnodeDescription](#), [PnodeLabels](#), [PnodeNumStates](#), [PnodeStateTitles](#), [PnodeStateDescriptions](#), [PnodeStateValues](#).

Used in the construction of Network [Warehouses](#) (see [WarehouseManifest](#)).

Similar to the function [BuildNetManifest](#).

Examples

```
## This expression provides an example Node manifest
nodeman1 <- read.csv(system.file("auxdata", "Mini-PP-Nodes.csv",
                               package="Peanut"),
                    row.names=1, stringsAsFactors=FALSE)

## Not run:
library(PNetica) ## Requires PNetica
sess <- NeticaSession()
startSession(sess)

netpath <- system.file("testnets", package="PNetica")
netnames <- paste(c("miniPP-CM", "PPcompEM", "PPconjEM", "PPtwostepEM",
                   "PPdurAttEM"), "dne", sep=".")

Nets <- ReadNetworks(file.path(netpath, netnames),
                    session=sess)

CM <- Nets[[1]]
EMs <- Nets[-1]

nodeman <- BuildNodeManifest(lapply(NetworkAllNodes(CM), as.Pnode))

for (n in 1:length(EMs)) {
  nodeman <- rbind(nodeman,
                  BuildNodeManifest(lapply(NetworkAllNodes(EMs[[n]]),
                                          as.Pnode)))
}

## Need to ensure that labels are in canonical order only for the
## purpose of testing
nodeman[,6] <- sapply(strsplit(nodeman[,6], ","),
                    function(l) paste(sort(l), collapse=","))
nodeman1[,6] <- sapply(strsplit(nodeman1[,6], ","),
                    function(l) paste(sort(l), collapse=","))
```

```

stopifnot(all.equal(nodeman, nodeman1))

## This is the node warehouse for PNetica
Nodehouse <- NNWarehouse(manifest=nodeman1,
                        key=c("Model", "NodeName"),
                        session=sess)
phyd <- WarehouseData(Nodehouse, c("miniPP_CM", "Physics"))
p3 <- MakePnode.NeticaNode(CM, "Physics", phyd)

attd <- WarehouseData(Nodehouse, c("PPdurAttEM", "Attempts"))
att <- MakePnode.NeticaNode(Nets[[5]], "Attempts", attd)

durd <- WarehouseData(Nodehouse, c("PPdurAttEM", "Duration"))
dur <- MakePnode.NeticaNode(Nets[[5]], "Duration", dur)

stopSession(sess)

## End(Not run)

```

BuildTable

Builds the conditional probability table for a Pnode

Description

The function `BuildTable` builds the conditional probability table for a `Pnode` object, and sets the prior weight for the node using the current values of parameters. It sets these in the Bayesian network object as appropriate for the implementation. The expression `BuildAllTables(net)` builds tables for all of the nodes in `PnetPnodes(net)`.

Usage

```

BuildTable(node)
BuildAllTables(net, debug=FALSE)

```

Arguments

<code>node</code>	A <code>Pnode</code> object whose table is to be built.
<code>net</code>	A <code>Pnet</code> object for whom the tables are needed to be built.
<code>debug</code>	A logical scalar. If true then <code>recover</code> is called after an error, so that the node in question can be inspected.

Details

The fields of the `Pnode` object correspond to the arguments of the `calcDPCTable` function. The output conditional probability table is then set in the node object in an implementation dependent way. Similarly, the current value of `GetPriorWeight` is used to set the weight that the prior table will be given in the EM algorithm.

Value

The node or net argument is returned invisibly. As a side effect the conditional probability table and prior weight of node (or a collection of nodes) is modified.

Logging and Debug Mode

As of version 0.6-2, the meaning of the debug argument is changed. In the new version, the `flog.logger` mechanism is used for progress reports, and error reporting. In particular, setting `flog.threshold`(DEBUG) (or TRACE) will cause progress reports to be sent to the logging output.

The debug argument has been repurposed. It now call `recover` when the error occurs, so that the problem can be debugged.

Note

The function BuildTable is an abstract generic function, and it needs a specific implementation. See the [PNetica-package](#) for an example.

Author(s)

Russell Almond

References

Almond, R. G. (2015) An IRT-based Parameterization for Conditional Probability Tables. Paper presented at the 2015 Bayesian Application Workshop at the Uncertainty in Artificial Intelligence Conference.

See Also

[Pnode](#), [PnodeProbs](#), [PnodeQ](#), [PnodePriorWeight](#), [PnodeRules](#), [PnodeLink](#), [PnodeLnAlphas](#), [PnodeAlphas](#), [PnodeBetas](#), [PnodeLinkScale](#), [GetPriorWeight](#), [calcDPCTable](#)

In many implementations, it will be necessary to run [PnetCompile](#) after building the tables.

Examples

```
## Not run:

## This is the implementation of BuildTable in Netica. The [<- and
## NodeExperience functions are part of the RNetica implementation.

BuildTable.NeticaNode <- function (node) {
  node[] <- calcDPCFrame(ParentStates(node),PnodeStates(node),
                        PnodeLnAlphas(node), PnodeBetas(node),
                        PnodeRules(node),PnodeLink(node),
                        PnodeLinkScale(node),PnodeQ(node),
                        PnodeParentTvals(node))
  NodeExperience(node) <- GetPriorWeight(node)
  invisible(node)
}
```

```
## This is a simplified implementation of BuildAllTables
## (The full implementation adds logging and error handling.)
BuildAllTables <- function (net) {
  lapply(PnetPnodes(net),BuildTable)
  invisible(net)
}

## End(Not run)
```

calcExpTables

Calculate expected tables for a parameterized network

Description

The performs the E-step of the GEM algorithm by running the internal EM algorithm of the host Bayes net package on the cases. After this is run, the posterior parameters for each conditional probability distribution should be the expected cell counts, that is the expected value of the sufficient statistic, for each [Pnode](#) in the net.

Usage

```
calcExpTables(net, cases, Estepit = 1, tol = sqrt(.Machine$double.eps))
```

Arguments

net	A Pnet object
cases	An object representing a set of cases. Note the type of object is implementation dependent. It could be either a data frame providing cases or a filename for a case file.
Estepit	An integer scalar describing the number of steps the Bayes net package should take in the internal EM algorithm.
tol	A numeric scalar giving the stopping tolerance for the Bayes net package internal EM algorithm.

Details

The [GEMfit](#) algorithm uses a generalized EM algorithm to fit the parameterized network to the given data. This loops over the following steps:

E-step Run the internal EM algorithm of the Bayes net package to calculate expected tables for all of the tables being learned. The function `calcExpTables` carries out this step.

M-step Find a set of table parameters which maximize the fit to the expected counts by calling [mapDPC](#) for each table. The function [maxAllTableParams](#) does this step.

Update CPTs Set all the conditional probability tables in the network to the new parameter values. The function [BuildAllTables](#) does this.

Convergence Test Calculate the log likelihood of the cases under the new parameters and stop if no change. The function `calcPnetLLike` calculates the log likelihood.

The function `calcExpTables` performs the E-step. It assumes that the native Bayes net class which `net` represents has a function which does EM learning with hyper-Dirichlet priors. After this internal EM algorithm is run, then the posterior should contain the expected cell counts that are the expected value of the sufficient statistics, i.e., the output of the E-step. Note that the function `maxAllTableParams` is responsible for reading these from the network.

The internal EM algorithm should be set to use the current value of the conditional probability tables (as calculated by `BuildTable(node)` for each node) as a starting point. This starting value is given a prior weight of `GetPriorWeight(node)`. Note that some Bayes net implementations allow a different weight to be given to each row of the table. The prior weight counts as a number of cases, and should be scaled appropriately for the number of cases in cases.

The parameters `Estepit` and `tol` are passed to the internal EM algorithm of the Bayes net. Note that the outer EM algorithm assumes that the expected table counts given the current values of the parameters, so the default value of one is sufficient. (It is possible that a higher value will speed up convergence, the parameter is left open for experimentation.) The tolerance is largely irrelevant as the outer EM algorithm does the tolerance test.

Value

The `net` argument is returned invisibly.

As a side effect, the internal conditional probability tables in the network are updated as are the internal weights given to each row of the conditional probability tables.

Note

The function `calcExpTables` is an abstract generic functions, and it needs specific implementations. See the [PNetica-package](#) for an example.

This function assumes that the host Bayes net implementation (e.g., [RNetica-package](#)): (1) `net` has an EM learning function, (2) the EM learning supports hyper-Dirichlet priors, (3) it is possible to recover the hyper-Dirichlet posteriors after running the internal EM algorithm.

Author(s)

Russell Almond

References

Almond, R. G. (2015) An IRT-based Parameterization for Conditional Probability Tables. Paper presented at the 2015 Bayesian Application Workshop at the Uncertainty in Artificial Intelligence Conference.

See Also

[Pnet](#), [GEMfit](#), [calcPnetLLike](#), [maxAllTableParams](#)

Examples

```

## Not run:

library(PNetica) ## Need a specific implementation
sess <- NeticaSession()
startSession(sess)

irt10.base <- ReadNetworks(system.file("testnets", "IRT10.2PL.base.dne",
                                     package="Peanut"),
                          session=sess)
irt10.base <- as.Pnet(irt10.base) ## Flag as Pnet, fields already set.
irt10.theta <- PnetFindNode(irt10.base, "theta")
irt10.items <- PnetPnodes(irt10.base)
## Flag items as Pnodes
for (i in 1:length(irt10.items)) {
  irt10.items[[i]] <- as.Pnode(irt10.items[[i]])
  ## Add node to list of observed nodes
  PnodeLabels(irt10.items[[1]]) <-
    union(PnodeLabels(irt10.items[[1]]), "onodes")
}
PnetCompile(irt10.base) ## Netica requirement

casepath <- system.file("testdat", "IRT10.2PL.200.items.cas", package="PNetica")

item1 <- irt10.items[[1]]

priorcounts <- sweep(PnodeProbs(item1), 1, GetPriorWeight(item1), "*")

calcExpTables(irt10.base, casepath)

postcounts <- sweep(PnodeProbs(item1), 1, PnodePostWeight(item1), "*")

## Posterior row sums should always be larger.
stopifnot(
  all(apply(postcounts, 1, sum) >= apply(priorcounts, 1, sum))
)

DeleteNetwork(irt10.base)
stopSession(sess)

## End(Not run)

```

calcPnetLLike

Calculates the log likelihood for a set of data under a Pnet model

Description

The function `calcPnetLLike` calculates the log likelihood for a set of data contained in cases using the current values of the conditional probability table in a [Pnet](#). If it is called after a call to

[BuildAllTables](#)(net) this will be the current value of the parameters.

Usage

```
calcPnetLLike(net, cases)
```

Arguments

net	A Pnet object representing a parameterized network.
cases	An object representing a set of cases. Note the type of object is implementation dependent. It could be either a data frame providing cases or a filename for a case file.

Details

This function provides the convergence test for the [GEMfit](#) algorithm. The Pnet represents a model (with parameters set to the value used in the current iteration of the EM algorithm) and cases a set of data. This function gives the log likelihood of the data.

This is a generic function shell. It is assumed that either (a) the native Bayes net implementation provides a way of calculating the log likelihood of a set of cases, or (b) it provides a way of calculating the likelihood of a single case, and the log likelihood of the case set can be calculated through iteration. In either case, the value of cases is implementation dependent. In [PNetica-package](#) the cases argument should be a filename of a Netica case file (see [write.CaseFile](#)).

Value

A numeric scalar giving the log likelihood of the data in the case file.

Note

The function calcPnetLLike is an abstract generic functions, and it needs specific implementations. See the [PNetica-package](#) for an example.

Author(s)

Russell Almond

References

Almond, R. G. (2015) An IRT-based Parameterization for Conditional Probability Tables. Paper presented at the 2015 Bayesian Application Workshop at the Uncertainty in Artificial Intelligence Conference.

See Also

[Pnet](#), [GEMfit](#), [calcExpTables](#), [maxAllTableParams](#)

Examples

```
## Not run:

library(PNetica) ## Need a specific implementation
sess <- NeticaSession()
startSession(sess)

irt10.base <- ReadNetworks(system.file("testnets", "IRT10.2PL.base.dne",
                                     package="PNetica"),
                          session=sess)

irt10.base <- as.Pnet(irt10.base) ## Flag as Pnet, fields already set.
irt10.theta <- PnetFindNode(irt10.base, "theta")
irt10.items <- PnetPnodes(irt10.base)
## Flag items as Pnodes
for (i in 1:length(irt10.items)) {
  irt10.items[[i]] <- as.Pnode(irt10.items[[i]])
  ## Add node to list of observed nodes
  PnodeLabels(irt10.items[[1]]) <-
    union(PnodeLabels(irt10.items[[1]]), "onodes")
}
PnetCompile(irt10.base) ## Netica requirement

casepath <- system.file("testdat", "IRT10.2PL.200.items.cas",
                       package="PNetica")

llike <- calcPnetLLike(irt10.base, casepath)

DeleteNetwork(irt10.base)
stopSession(sess)

## End(Not run)
```

CompensatoryGadget *Shiny gadget for editing compensatory pnodes*

Description

These functions open a shiny application (in a browser window or other location) for editing a [Pnode](#) object. To reduce the complexity, the display assumes that [PnodeLink\(pnode\)](#) is [partialCredit](#) or [gradedResponse](#), and that [PnodeLink\(pnode\)](#) is [Compensatory](#) (Conjunctive or Disjunctive are also possibilities, but usually, the [OffsetGadget](#) is a better parameterization for these rules).

Usage

```
MakeCompensatoryGadget(pnode, color = "firebrick")
CompensatoryGadget(pnode, color="firebrick", viewer=shiny::paneViewer())
```

Arguments

pnode	A Pnode object to be modified.
color	A base color to use for barcharts (see barchart.CPF). Execute <code>colors()</code> for a list of choices.
viewer	This is passed to the viewer argument of <code>shiny::runGadget</code> .

Details

The `CompensatoryGadget` assumes that:

- The link function is [partialCredit](#) or [gradedResponse](#).
- There is a single rule for all states, and `PnodeQ(pnode)=TRUE`.
- One of the multiple-*a* rules: [Compensatory](#), [Conjunctive](#) or [Disjunctive](#) is used, so that there is one alpha for each parent.
- There is one beta for each state except the last, which is a reference state.

It is most useful for compensatory models.

Value

The function `MakeCompensatoryGadget` returns a list of two functions, `ui` and `server`. These are meant to be passed to `shiny::runApp` to generate the actual app.

The function `CompensatoryGadget` will return the `pnode` object or throw a ‘Cancel-Error’.

Note

Although the addition of the ‘`MakeCompensatoryGadget`’ was specifically designed to allow the embedding of the gadget in Rmarkdown shiny documents, the default Rmarkdown layout algorithms do not work properly.

Adding the following markdown code:

```
```{css, echo = FALSE}
.shiny-frame{height: 1000px;}
```
```

will adjust the space allocated for the gadget to 1000 pixels, allowing for sufficient room for display.

Author(s)

Russell Almond

References

Almond, R. G. (2015) An IRT-based Parameterization for Conditional Probability Tables. Paper presented at the 2015 Bayesian Application Workshop at the Uncertainty in Artificial Intelligence Conference.

See Also

[Pnode](#), [calcDPCFrame](#), [barchart.CPF](#)
[OffsetGadget](#), [RegressionGadget](#), [DPCGadget](#)

Examples

```
## Not run:
library(PNetica) ## Requires implementation
sess <- NeticaSession()
startSession(sess)

tNet <- CreateNetwork("TestNet",sess)

theta1 <- NewDiscreteNode(tNet,"theta1",
                          c("VH","High","Mid","Low","VL"))
PnodeStateValues(theta1) <- effectiveThetas(PnodeNumStates(theta1))
PnodeProbs(theta1) <- rep(1/PnodeNumStates(theta1),PnodeNumStates(theta1))
theta2 <- NewDiscreteNode(tNet,"theta2",
                          c("VH","High","Mid","Low","VL"))
PnodeStateValues(theta2) <- effectiveThetas(PnodeNumStates(theta2))
PnodeProbs(theta2) <- rep(1/PnodeNumStates(theta1),PnodeNumStates(theta2))

## CompensatoryGadget

partial3 <- NewDiscreteNode(tNet,"partial3",
                           c("FullCredit","PartialCredit","NoCredit"))
PnodeParents(partial3) <- list(theta1,theta2)

## Usual way to set rules is in constructor
partial3 <- Pnode(partial3,rules="Compensatory", link="partialCredit")
PnodePriorWeight(partial3) <- 10
BuildTable(partial3)

partial3 <- CompensatoryGadget(partial3)

## This expression can be used inside an Rmarkdown document
gadget <- MakeCompensatoryGadget(partial3)
shinyApp(gadget$ui,gadget$server,options(height=2000))

DeleteNetwork(tNet)
stopSession(sess)

## End(Not run)
```

Description

These functions open a shiny application (in a browser window or other location) for editing a `Pnode` object. This is the most complex version taking advantage of all of the complex choices available using the partial credit link (i.e., `PnodeLink(pnode)` is `partialCredit`). In particular, it allows specification of different combination rules for each state transition. The rules can be either the multiple-*a* type (e.g., `Compensatory`) or multiple-*b* type (e.g., `OffsetConjunctive` or `OffsetDisjunctive`). It even allows for a subset of parents to be used in each transition by specifying an inner *Q*-matrix (though `PnodeQ`). To manage this complexity, each transition is displayed on a separate tab of the interface.

Usage

```
MakeDPCGadget(pnode, color = "steelblue")
DPCGadget(pnode, color="steelblue",viewer=shiny::paneViewer())
```

Arguments

| | |
|---------------------|--|
| <code>pnode</code> | A <code>Pnode</code> object to be modified. |
| <code>color</code> | A base color to use for barcharts (see <code>barchart.CPF</code>). Execute <code>colors()</code> for a list of choices. |
| <code>viewer</code> | This is passed to the viewer argument of <code>shiny::runGadget</code> . |

Details

The `DPCGadget` assumes that:

- The link function is `partialCredit`.
- There is a list of rules, one for each state transition (i.e., one for all states except the last and lowest value. Alternatively, if a single value is given it is used for all transitions.
- The value of `PnodeQ(pnode)` is a logical matrix with rows corresponding to state transitions and columns corresponding to parent variables. If any cell value is false, that parent variable is not used to calculate the effective theta for that cell transition. As a special case, if `PnodeQ(pnode)` equals `TRUE`, then it is considered to be a matrix with all elements true.

Both `PnodeAlphas(node)` and `PnodeBetas(node)` should be a list of vectors. The length of each vector should be either one or the number of relevant (after filtering with the inner *Q*-matrix) parent variables. Which it needs to be depends on whether the rule for that transition is a multiple-*a* type (alphas should match number of parents) or multiple-*b* type (betas should match number of parents). In either case, if a single value is given and a longer list is expected, it will be replicated across the parents. The length of the lists should match the state transitions (with the first one corresponding to the transition from the 2nd highest state to the highest, the second the transition to the 2nd highest state and so forth). No entry is needed for the lowest state. Once again, if a single vector is given in place of the list, it will be replicated as needed.

To recap, the outer (list) structure corresponds to state transitions and the inner (vector) structure corresponds to the parent variables. It is recommended to use labeled vectors and lists to annotate the structure.

Value

The function `MakeDPCGadget` returns a list of two functions, `ui` and `server`. These are meant to be passed to `shiny::runApp` to generate the actual app.

The function `DPCGadget` will return the `pnode` object or throw a ‘Cancel-Error’.

Author(s)

Russell Almond

References

Almond, R. G. (2015) An IRT-based Parameterization for Conditional Probability Tables. Paper presented at the 2015 Bayesian Application Workshop at the Uncertainty in Artificial Intelligence Conference.

See Also

[Pnode](#), [calcDPCFrame](#), [barchart.CPF](#)
[OffsetGadget](#), [RegressionGadget](#), [CompensatoryGadget](#)

Examples

```
## Not run:
library(PNetica) ## Requires implementation
sess <- NeticaSession()
startSession(sess)

tNet <- CreateNetwork("TestNet",sess)

theta1 <- NewDiscreteNode(tNet,"theta1",
                          c("VH","High","Mid","Low","VL"))
PnodeStateValues(theta1) <- effectiveThetas(PnodeNumStates(theta1))
PnodeProbs(theta1) <- rep(1/PnodeNumStates(theta1),PnodeNumStates(theta1))
theta2 <- NewDiscreteNode(tNet,"theta2",
                          c("VH","High","Mid","Low","VL"))
PnodeStateValues(theta2) <- effectiveThetas(PnodeNumStates(theta2))
PnodeProbs(theta2) <- rep(1/PnodeNumStates(theta1),PnodeNumStates(theta2))

## DPCGadget

partial3 <- NewDiscreteNode(tNet,"partial3",
                            c("FullCredit","PartialCredit","NoCredit"))
PnodeParents(partial3) <- list(theta1,theta2)

## Usual way to set rules is in constructor
partial3 <- Pnode(partial3,rules="Compensatory", link="partialCredit")
PnodePriorWeight(partial3) <- 10
BuildTable(partial3)

partial3 <- DPCGadget(partial3)
```

```

## This expression can be used inside an Rmarkdown document
gadget <- MakeDPCGadget(partial3)
shinyApp(gadget$ui,gadget$server,options(height=2000))

## More complex example showing off some of the options:
## Set up so that first skill only needed for first transition, second
## skill for second transition; Adjust alphas to match
PnodeQ(partial3) <- matrix(c(TRUE,TRUE,
                             TRUE,FALSE), 2,2, byrow=TRUE)
PnodeLnAlphas(partial3) <- list(FullCredit=c(theta1=-.25,theta2=.25),
                               PartialCredit=0)

partial3 <- DPCGadget(partial3)

DeleteNetwork(tNet)
stopSession(sess)

## End(Not run)

```

flattenStats

Flattens a statistic list into a numeric or character vector.

Description

Margin statistics are list valued, however, it is often useful to break them down into separate statistics for each state. In particular, this facilitates putting the data into a data.frame format for further analysis.

Usage

```
flattenStats(statlist)
```

Arguments

statlist A named list containing the statistics.

Details

Vector valued statistics produced by [PnodeMargin](#) are replaced by multiple statistics with the state name appended (after a dot) to the variable name.

Scalar valued statistics are left as is.

Value

A vector with the statistics. If all statistics are numeric, it will be a numeric vector. If some are character valued (e.g., [PnodeMode](#)), it will be a character vector.

Author(s)

Russell Almond

See Also[PnodeMargin](#), [PnodeEAP](#), [PnodeMode](#),**Examples**

```
slist <- list("Physics_EAP"=.3,
             "Physics_Margin"=c("High"=.5,"Medium"=.3,"Low"=.2),
             "Physics_Mode"="High")
flattenStats(slist)
```

flog.try

*Trys to execute an expression with errors logged.***Description**

This is a version of `try` which logs errors using the `flog.logger` mechanism.

Usage

```
flog.try(expr, context = deparse(substitute(expr)), loggename = flog.namespace(),
         tracelevel = c("WARN", "ERROR", "FATAL"))
```

Arguments

| | |
|-------------------------|---|
| <code>expr</code> | An R expression to be executed. |
| <code>context</code> | A character string defining what was operation is being performed for use in the log message. |
| <code>loggename</code> | A package name defining the logger to be used. See flog.namespace . |
| <code>tracelevel</code> | A character vector. In response to signals of the listed types, a stack trace will be sent to the log file. |

Details

This function behaves like the `try` function, attempt to execute `expr`. If successful, the result is returned, if not an object of class `try-error` is returned, so that the calling function can figure out how to proceed.

It has two important difference from `try`. The first is the `context` argument which provides information about what was happening when the error was generated. In a large problem, this can provide vital debugging information, like the issue was with a particular node in a graph.

The second is that the error message and the stack trace are posted to the logging stream using the `flog.logger` function. This makes the code easier to use in server processes.

Value

Either the result of running `expr` or an object of class `try-error`.

Note

I should move this to the `RGAutils` package as it is generally useful.

Author(s)

Russell Almond

See Also

[try](#), [flog.logger](#)

The function `maxAllTableParams` shows an example of this in use.

Examples

```
## Not run:
maxAllTableParams <- function (net, Mstepit=5,
                               tol=sqrt(.Machine$double.eps),
                               debug=FALSE) {
  Errs <- list()
  netnm <- PnetName(net)
  lapply(PnetPnodes(net),
        function (nd) {
          ndnm <- PnodeName(nd)
          flog.debug("Updating params for node
                    out <- flog.try(maxCPTParam(nd,Mstepit,tol),
                                     context=sprintf("Updating params for node
                                                         ndnm, netnm))
                    if (is(out,'try-error')) {
                      Errs <- c(Errs,out)
                      if (debug) recover()
                    }
          })
  if (length(Errs) > 0L)
    stop("Errors encountered while updating parameters for ",netnm)
  invisible(net)
}

## End(Not run)
```

Description

A [Pnet](#) is a description of a parameterized Bayesian network, with each [Pnode](#) giving the parameterization for its conditional probability table. This function uses a generalized EM algorithm to find the values of the parameters for each Pnode which maximize the posterior probability of the data in cases.

Usage

```
GEMfit(net, cases, tol = sqrt(.Machine$double.eps),
       maxit = 100, Estepit = 1, Mstepit = 30,
       trace=FALSE, debugNo=maxit+1)
```

Arguments

| | |
|---------|--|
| net | A Pnet object |
| cases | An object representing a set of cases. Note the type of object is implementation dependent. It could be either a data frame providing cases or a filename for a case file. |
| tol | A numeric scalar giving the stopping tolerance for the for the EM algorithm. |
| maxit | An integer scalar giving the maximum number of iterations for the outer EM algorithm. |
| Estepit | An integer scalar giving the number of steps the Bayes net package should take in the internal EM algorithm during the E-step. |
| Mstepit | An integer scalar giving the number of steps that should be taken by mapDPC during the M-step. |
| trace | A logical value which indicates whether or not cycle by cycle information should be sent to to the flog.logger . |
| debugNo | An integer scalar. When this iteration is reached, then the flog.threshold (DEBUG) will be set, so more debugging information will be provided. |

Details

The [GEMfit](#) algorithm uses a generalized EM algorithm to fit the parameterized network to the given data. This loops over the following steps:

E-step Run the internal EM algorithm of the Bayes net package to calculate expected tables for all of the tables being learned. The function [calcExpTables](#) carries out this step.

M-step Find a set of table parameters which maximize the fit to the expected counts by calling [mapDPC](#) for each table. The function [maxAllTableParams](#) does this step.

Update CPTs Set all the conditional probability tables in the network to the new parameter values. The function [BuildAllTables](#) does this.

Convergence Test Calculate the log likelihood of the cases under the new parameters and stop if no change. The function [calcPnetLLike](#) calculates the log likelihood.

Note that although `GEMfit` is not a generic function, the four main component functions, `calcExpTables`, `maxAllTableParams`, `BuildAllTables`, and `calcPnetLLike`, are generic functions. In particular, the `cases` argument is passed to `calcExpTables` and `calcPnetLLike` and must be whatever the host Bayes net package regards as a collection of cases. In `PNetica-package` the `cases` argument should be a filename of a Netica case file (see `write.CaseFile`).

The parameter `tol` controls the convergence checking. In particular, the algorithm stops when the difference in log-likelihood (as computed by `calcPnetLLike`) between iterations is less than `tol` in absolute value. If the number of iterations exceeds `maxit` the algorithm will stop and report lack of convergence.

The E-step and the M-step are also both iterative; the parameters `Estepit` and `Mstepit` control the number of iterations taken in each step respectively. As the goal of the E-step is to calculate the expected tables of counts, the default value of 1 should be fine. Although the algorithm should eventually converge for any value of `Mstepit`, different values may affect the convergence rate, and analysts may need to experiment with application specific values of this parameter.

The arguments `trace` and `debugNo` are used to provide extra debugging information. Setting `trace` to `TRUE` means that a message is printed after tables are built but before they are updated. Setting `debugNo` to a certain integer, will begin node-by-node messages for both `BuildAllTables` and `maxAllTableParams`. In particular, setting it to 1 is useful for debugging problems that occur at initialization. If the problem turns up at a later cycle, the `trace` option can be used to figure out when the error occurs.

Value

A list with three elements:

| | |
|------------------------|---|
| <code>converged</code> | A logical flag indicating whether or not the algorithm reached convergence. |
| <code>iter</code> | An integer scalar giving the number of iterations of the outer EM loop taken by the algorithm (plus 1 for the starting point). |
| <code>llikes</code> | A numeric vector of length <code>iter</code> giving the values of the log-likelihood after each iteration. (The first value is the initial log likelihood.) |

As a side effect the `PnodeLnAlphas` and `PnodeBetas` fields of all nodes in `PnetNodes(net)` are updated to better fit the expected tables, and the internal conditional probability tables are updated to match the new parameter values.

Logging and Debug Mode

As of version 0.6-2, the meaning of the `trace` and `debugNo` has changed. In the new version, the `flog.logger` mechanism is used for progress reports, and error reporting.

Setting `trace` to `true` causes information about the steps of the algorithm (including the log likelihood at each step) to be output to the current appender (see `flog.appender`) The logging is done at the `INFO` level. As the default appender is the console, and `INFO` is the default logging level, the meaning of this parameter hasn't changed much.

The meaning of `debugNo` has changed, however. Previously, it would turn on extra debug information when the target iteration was reached. That information is now always logged at the `DEBUG` level. So now if the current iteration reached `debugNo`, then `GEMfit` calls `flog.threshold(DEBUG)` to provide more information. This allows the more detailed `DEBUG`-level messages to be turned on when the EM algorithm is closer to convergence.

Note

Note that although this is not a generic function, the four main component functions: [calcExpTables](#), [maxAllTableParams](#), [BuildAllTables](#), and [calcPnetLLike](#). All four must have specific implementations for this function to work. See the [PNetica-package](#) for an example.

These functions assume that the host Bayes net implementation (e.g., [RNetica-package](#)): (1) net has an EM learning function, (2) the EM learning supports hyper-Dirichlet priors, (3) it is possible to recover the hyper-Dirichlet posteriors after running the internal EM algorithm.

Author(s)

Russell Almond

References

Almond, R. G. (2015) An IRT-based Parameterization for Conditional Probability Tables. Paper presented at the 2015 Bayesian Application Workshop at the Uncertainty in Artificial Intelligence Conference.

See Also

[Pnet](#), [calcExpTables](#), [calcPnetLLike](#), [maxAllTableParams](#), [BuildAllTables](#)

Examples

```
## Not run:

library(PNetica) ## Need a specific implementation
sess <- NeticaSession()
startSession(sess)

irt10.base <- ReadNetworks(system.file(
  "testnets", "IRT10.2PL.base.dne",
  package="PNetica"),
  session=sess)
irt10.base <- as.Pnet(irt10.base) ## Flag as Pnet, fields already set.
irt10.theta <- PnetFindNode(irt10.base, "theta")
irt10.items <- PnetPnodes(irt10.base)
## Flag items as Pnodes
for (i in 1:length(irt10.items)) {
  irt10.items[[i]] <- as.Pnode(irt10.items[[i]])
  ## Add node to list of observed nodes
  PnodeLabels(irt10.items[[1]]) <-
    union(PnodeLabels(irt10.items[[1]]), "onodes")
}

casepath <- system.file("testdat", "IRT10.2PL.200.items.cas", package="PNetica")

BuildAllTables(irt10.base)
PnetCompile(irt10.base) ## Netica requirement
```

```

item1 <- irt10.items[[1]]
priB <- PnodeBetas(item1)
priA <- PnodeAlphas(item1)
priCPT <- PnodeProbs(item1)

gemout <- GEMfit(irt10.base,casepath,trace=TRUE)

postB <- PnodeBetas(item1)
postA <- PnodeAlphas(item1)
postCPT <- PnodeProbs(item1)

## Posterior should be different
stopifnot(
  postB != priB, postA != priA
)

### The network that was used for data generation.
irt10.true <- ReadNetworks(system.file(
  "testnets", "IRT10.2PL.true.dne",
  package="PNetica"),
  session=sess)
irt10.true <- as.Pnet(irt10.true) ## Flag as Pnet, fields already set.
irt10.ttheta <- PnetFindNode(irt10.true,"theta")
irt10.titems <- PnetPnodes(irt10.true)
## Flag titems as Pnodes
for (i in 1:length(irt10.titems)) {
  irt10.titems[[i]] <- as.Pnode(irt10.titems[[i]])
  ## Add node to list of observed nodes
  PnodeLabels(irt10.titems[[1]]) <-
    union(PnodeLabels(irt10.titems[[1]]),"onodes")
}

BuildAllTables(irt10.true)
PnetCompile(irt10.true) ## Netica requirement

## See how close we are.
for (j in 1:length(irt10.titems)) {
  cat("diff[" ,j, "] = ",
      sum(abs(PnodeProbs(irt10.items[[j]])-
             PnodeProbs(irt10.titems[[j]])))/
      length(PnodeProbs(irt10.items[[j]])), "\n")
}

DeleteNetwork(irt10.base)
DeleteNetwork(irt10.true)

## End(Not run)

```

| | |
|---------------|---|
| is.legal.name | <i>Checks to see if names are valid for objects in warehouse.</i> |
|---------------|---|

Description

Objects in a warehouse may have restrictions on the names that are allowed. For example, Netica nodes and nets must have names that follow common variable naming convention (alphanumeric, starts with a letter, no embedded spaces, etc.). The function `is.legal.name` checks the name type, and `as.legal.name` munges the name so that it is legal.

Usage

```
is.legal.name(warehouse, name)
as.legal.name(warehouse, name)
```

Arguments

| | |
|-----------|---|
| warehouse | A warehouse which defines the type of object. |
| name | A character vector giving names to be tested or munged. |

Value

For `is.valid.name`, a logical value returning the result of each test.

For `as.valid.name`, a character vector with the modified names.

Note

The [BNWarehouse](#) and [NNWarehouse](#) have a `prefix` field which is used to ensure that names always start with a letter.

Author(s)

Russell Almond

Examples

```
## Not run:
## Requires PNetica
library(PNetica)

sess <- NeticaSession()
startSession(sess)

## BNWarehouse is the PNetica Net Warehouse.
## This provides an example network manifest.
table.dir <- system.file("auxdata", package="Peanut")
net.dir <- system.file("testnets", package="PNetica")
```

```

netman1 <- read.csv(file.path(table.dir,"Mini-PP-Nets.csv"),
                    row.names=1, stringsAsFactors=FALSE)
Nethouse <- BNWarehouse(manifest=netman1,session=sess,key="Name",
                        address=net.dir,prefix="S")

stopifnot(is.legal.name(Nethouse,c("CamelCase","Embedded Space")) ==
          c(TRUE,FALSE),
          as.legal.name(Nethouse,"100c3") == "S100c3")

## This expression provides an example Node manifest
nodeman1 <- read.csv(file.path(table.dir,"Mini-PP-Nodes.csv"),
                    row.names=1,stringsAsFactors=FALSE)
Nodehouse <- NNWarehouse(manifest=nodeman1,
                          key=c("Model","NodeName"),
                          session=sess,prefix="V")

stopifnot(is.legal.name(Nodehouse,c("Neg1","-1")) ==
          c(TRUE,FALSE),
          as.legal.name(Nodehouse,1) == "V1")

## End(Not run)

```

isPnodeContinuous *Functions for handling continuous nodes.*

Description

Continuous nodes are handled slightly differently from discrete nodes. The function `isPnodeContinuous` returns a logical value indicating whether or not the node is continuous.

Continuous nodes can behave like discrete nodes (for the purposes of building conditional probability tables, see [BuildTable](#)) if states are created from ranges of values. The function `PnodeStateBounds` accesses those ranges.

Usage

```

isPnodeContinuous(node)
PnodeStateBounds(node)
PnodeStateBounds(node) <- value

```

Arguments

`node` A [Pnode](#) object.

`value` A k by 2 numeric matrix giving the upper and lower bound for each state.

Details

Continuous, in this case, covers nodes whose possible states are numeric, either integer or real. The current model supports these nodes in a discrete Bayesian network by discretizing them. In particular, the range is broken up into a number of non-overlapping regions, each region corresponding to a state.

For example, consider a variable which is a count, and the analyst wants to consider the values 0, 1, 2 or 3, and 4 or more. This can be done by setting bounds on these states:

| | | |
|------------|------|-----|
| "Zero" | -0.5 | 0.5 |
| "One" | 0.5 | 1.5 |
| "TwoThree" | 1.5 | 3.5 |
| "FourPlus" | 3.5 | Inf |

This matrix is the NodeStateBounds for the node. Note that the second column is the same as the first (offset by one). Note also that infinite (Inf and -Inf) values are allowed.

Setting the state bounds to a matrix with k rows, will make the variable behave as if it has k states.

Value

The function isPnodeContinuous returns a logical value.

The function PnodeStateBounds returns a k by 2 numeric matrix giving the upper and lower bounds. Note that if bounds have not been set for the node, then it will return a matrix with 0 rows.

Note

This is rather strongly tied to how Netica treats continuous variables. A different mechanism might be necessary as Peanut is expanded to cover more implementations.

Right now, the value is the midpoint of the interval. This cause problems when converting to T-values.

The setter function is very strict about the upper and lower bounds matching. Even a mismatch at the least significant digit will cause a problem.

Author(s)

Russell Almond

See Also

[Pnode](#), [PnodeStateValues](#), [PnodeParentTvals](#)

Examples

```
## Not run:
library(PNetica) ## Requires implementation
sess <- NeticaSession()
startSession(sess)

tNet <- CreateNetwork("TestNet", session=sess)

theta1 <- NewDiscreteNode(tNet, "theta1",
                          c("VH", "High", "Mid", "Low", "VL"))
PnodeStateValues(theta1) <- effectiveThetas(NodeNumStates(theta1))
stopifnot (!isPnodeContinuous(theta1))

## This gives an error
out <- try(PnodeStateBounds(theta1))
stopifnot (is(out, 'try-error'))

theta0 <- NewContinuousNode(tNet, "theta0")
stopifnot(nrow(PnodeStateBounds(theta0)) == 0L)

norm5 <-
  matrix(c(qnorm(c(.001, .2, .4, .6, .8)),
           qnorm(c(.2, .4, .6, .8, .999))), 5, 2,
         dimnames=list(c("VH", "High", "Mid", "Low", "VL"),
                       c("LowerBound", "UpperBound")))
PnodeStateBounds(theta0) <- norm5
PnodeStates(theta0)
PnodeStateBounds(theta0)
PnodeStateValues(theta0) ## Note these are medians not mean wrt normal!

DeleteNetwork(tNet)
stopSession(sess)

## End(Not run)
```

maxAllTableParams

Find optimal parameters of Pnet or Pnode to match expected tables

Description

These functions assume that an expected count contingency table can be built from the network. They then try to find the set of parameters maximizes the probability of the expected contingency table with repeated calls to `mapDPC`. The function `maxCPTParam` maximizes a single `Pnode` and the function `maxAllTableParams` maximizes all `Pnodes` (i.e., the value of `PnetPnodes`(net) in a `Pnet`).

Usage

```
maxAllTableParams(net, Mstepit = 5, tol = sqrt(.Machine$double.eps), debug=FALSE)
maxCPTParam(node, Mstepit = 5, tol = sqrt(.Machine$double.eps))
```

Arguments

| | |
|---------|--|
| net | A Pnet object giving the parameterized network. |
| node | A Pnode object giving the parameterized node. |
| Mstepit | A numeric scalar giving the number of maximization steps to take. Note that the maximization does not need to be run to convergence. |
| tol | A numeric scalar giving the stopping tolerance for the maximizer. |
| debug | A logical scalar. If true then recover is called after an error, so that the node in question can be inspected. |

Details

The [GEMfit](#) algorithm uses a generalized EM algorithm to fit the parameterized network to the given data. This loops over the following steps:

E-step Run the internal EM algorithm of the Bayes net package to calculate expected tables for all of the tables being learned. The function [calcExpTables](#) carries out this step.

M-step Find a set of table parameters which maximize the fit to the expected counts by calling [mapDPC](#) for each table. The function [maxAllTableParams](#) does this step.

Update CPTs Set all the conditional probability tables in the network to the new parameter values. The function [BuildAllTables](#) does this.

Convergence Test Calculate the log likelihood of the cases under the new parameters and stop if no change. The function [calcPnetLLike](#) calculates the log likelihood.

The function [maxAllTableParams](#) performs the M-step of this operation. Under the *global parameter independence* assumption, the parameters for the conditional probability tables for different nodes are independent given the sufficient statistics; that is, the expected contingency tables. The default method of [maxAllTableParams](#) calls [maxCPTParam](#) on each node in [PnetPnodes](#)(net).

After the hyper-Dirichlet EM algorithm is run by [calcExpTables](#), a hyper-Dirichlet prior should be available for each conditional probability table. As the parameter of the Dirichlet distribution is a vector of pseudo-counts, the output of this algorithm should be a table of pseudo counts. Often this is stored as the updated conditional probability table and a vector of row weights indicating the strength of information for each row. Using the [RNetica-package](#), this is calculated as: `sweep(NodeProbs(item1), 1, NodeExperience(item1), "*")`

The function [maxCPTParam](#) is essentially a wrapper which extracts the table of pseudo-counts from the network and then calls [mapDPC](#) to maximize the parameters, updating the parameters of node to the result.

The parameters `Mstepit` and `tol` are passed to [mapDPC](#) to control the gradient descent algorithm used for maximization. Note that for a generalized EM algorithm, the M-step does not need to be run to convergence, a couple of iterations are sufficient. The value of `Mstepit` may influence the speed of convergence, so the optimal value may vary by application. The tolerance is largely irrelevant (if `Mstepit` is small) as the outer EM algorithm does the tolerance test.

Value

The expression `maxCPTParam(node)` returns `node` invisibly. The expression `maxAllTableParams(net)` returns `net` invisibly.

As a side effect the `PnodeLnAlphas` and `PnodeBetas` fields of node (or all nodes in `PnetPnodes(net)`) are updated to better fit the expected tables.

Logging and Debug Mode

As of version 0.6-2, the meaning of the debug argument is changed. In the new version, the `flog.logger` mechanism is used for progress reports, and error reporting. In particular, setting `flog.threshold(DEBUG)` (or `TRACE`) will cause progress reports to be sent to the logging output.

The debug argument has been repurposed. It now call `recover` when the error occurs, so that the problem can be debugged.

Note

The function `maxCPTParam` is an abstract generic function, and it needs specific implementations. See the `PNetica-package` for an example. A default implementation is provided for `maxAllTableParams` which loops through calls to `maxCPTParam` for each node in `PnetPnodes(net)`.

This function assumes that the host Bayes net implementation (e.g., `RNetica-package`): (1) net has an EM learning function, (2) the EM learning supports hyper-Dirichlet priors, (3) it is possible to recover the hyper-Dirichlet posteriors after running the internal EM algorithm.

Author(s)

Russell Almond

References

Almond, R. G. (2015) An IRT-based Parameterization for Conditional Probability Tables. Paper presented at the 2015 Bayesian Application Workshop at the Uncertainty in Artificial Intelligence Conference.

See Also

[Pnet](#), [Pnode](#), [GEMfit](#), [calcPnetLLike](#), [calcExpTables](#), [mapDPC](#)

Examples

```
## Not run:

library(PNetica) ## Need a specific implementation
sess <- NeticaSession()
startSession(sess)

irt10.base <- ReadNetworks(system.file(
  "testnets", "IRT10.2PL.base.dne",
  package="PNetica"),
  session=sess)
irt10.base <- as.Pnet(irt10.base) ## Flag as Pnet, fields already set.
irt10.theta <- NetworkFindNode(irt10.base,"theta")
irt10.items <- PnetPnodes(irt10.base)
## Flag items as Pnodes
```

```

for (i in 1:length(irt10.items)) {
  irt10.items[[i]] <- as.Pnode(irt10.items[[i]])
  ## Add node to list of observed nodes
  PnodeLabels(irt10.items[[1]]) <-
    union(PnodeLabels(irt10.items[[1]]), "onodes")
}

casepath <- system.file("testdat", "IRT10.2PL.200.items.cas", package="PNetica")

BuildAllTables(irt10.base)
PnetCompile(irt10.base) ## Netica requirement

item1 <- irt10.items[[1]]
priB <- PnodeBetas(item1)
priA <- PnodeAlphas(item1)
pricPT <- PnodeProbs(item1)

gemout <- GEMfit(irt10.base, casepath, trace=TRUE)

calcExpTables(irt10.base, casepath)

maxAllTableParams(irt10.base)

postB <- PnodeBetas(item1)
postA <- PnodeAlphas(item1)
BuildTable(item1)
postCPT <- PnodeProbs(item1)

## Posterior should be different
stopifnot(
  postB != priB, postA != priA
)

DeleteNetwork(irt10.base)
stopSession(sess)

## End(Not run)

```

Description

These functions open a shiny application (in a browser window or other location) for editing a `Pnode` object. To reduce the complexity, the display assumes that `PnodeLink(pnode)` is `partialCredit` or `gradedResponse`, and that `PnodeLink(pnode)` is `Compensatory` (Conjunctive or Disjunctive are also possibilities, but usually, the `OffsetGadget` is a better parameterization for these rules).

Usage

```
MakeOffsetGadget(pnode, color = "plum")
OffsetGadget(pnode, color="plum", viewer=shiny::paneViewer())
```

Arguments

| | |
|--------|---|
| pnode | A Pnode object to be modified. |
| color | A base color to use for barcharts (see barchart.CPF). Execute <code>colors()</code> for a list of choices. |
| viewer | This is passed to the viewer argument of <code>shiny::runGadget</code> . |

Details

The `CompensatoryGadget` assumes that:

- The link function is [partialCredit](#)
- There is a single rule for all states, and `PnodeQ(pnode)=TRUE`.
- One of the multiple-*b* rules: `OffsetConjunctive` or `OffsetDisjunctive` is used, so that there is one beta for each parent.
- There is either a single alpha, or one alpha for each state except the last, which is a reference state.

This is the recommended way to model conjunctive and disjunctive models (using a separate difficulty/demand for each parent).

Value

The function `MakeOffsetGadget` returns a list of two functions, `ui` and `server`. These are meant to be passed to `shiny::runApp` to generate the actual app.

The function `OffsetGadget` will return the `pnode` object or throw a ‘Cancel-Error’.

Author(s)

Russell Almond

References

Almond, R. G. (2015) An IRT-based Parameterization for Conditional Probability Tables. Paper presented at the 2015 Bayesian Application Workshop at the Uncertainty in Artificial Intelligence Conference.

See Also

[Pnode](#), [calcDPCFrame](#), [barchart.CPF](#)
[CompensatoryGadget](#), [RegressionGadget](#), [DPCGadget](#)

Examples

```

## Not run:
library(PNetica) ## Requires implementation
sess <- NeticaSession()
startSession(sess)

tNet <- CreateNetwork("TestNet",sess)

theta1 <- NewDiscreteNode(tNet,"theta1",
                          c("VH","High","Mid","Low","VL"))
PnodeStateValues(theta1) <- effectiveThetas(PnodeNumStates(theta1))
PnodeProbs(theta1) <- rep(1/PnodeNumStates(theta1),PnodeNumStates(theta1))
theta2 <- NewDiscreteNode(tNet,"theta2",
                          c("VH","High","Mid","Low","VL"))
PnodeStateValues(theta2) <- effectiveThetas(PnodeNumStates(theta2))
PnodeProbs(theta2) <- rep(1/PnodeNumStates(theta1),PnodeNumStates(theta2))

## CompensatoryGadget

partial3 <- NewDiscreteNode(tNet,"partial3",
                            c("FullCredit","PartialCredit","NoCredit"))
PnodeParents(partial3) <- list(theta1,theta2)

## Usual way to set rules is in constructor
partial3 <- Pnode(partial3,rules="OffsetConjunctive", link="partialCredit")
PnodeLnAlphas(partial3) <- 0
PnodeQ(partial3) <- TRUE
PnodeBetas(partial3) <- c(0,1)
BuildTable(partial3)
PnodePriorWeight(partial3) <- 10
BuildTable(partial3)

partial3 <- OffsetGadget(partial3)

## This expression can be used inside an Rmarkdown document
gadget <- MakeOffsetGadget(partial3)
shinyApp(gadget$ui,gadget$server,options(height=2000))

DeleteNetwork(tNet)
stopSession(sess)

## End(Not run)

```

Omega2Pnet

Constructs a parameterized network from an Omega matrix.

Description

An Omega matrix (represented as a data frame) is a structure which describes a Bayesian network as a series of regressions from the parent nodes to the child nodes. It actually contains two ma-

trixes, one giving the structure and the other the regression coefficients. A skeleton matrix can be constructed through the function [Pnet2Omega](#).

Usage

```
Omega2Pnet(OmegaMat, pn, nodewarehouse, defaultRule = "Compensatory",
  defaultLink = "normalLink", defaultAlpha = 1, defaultBeta = 0,
  defaultLinkScale = 1, defaultPriorWeight=10, debug = FALSE, override =FALSE,
  addTvals = TRUE)
```

Arguments

| | |
|--------------------|--|
| OmegaMat | A data frame containing an Omega matrix (see values section of Pnet2Omega). |
| pn | A (possible empty) Pnet object. This will be modified by the function. |
| nodewarehouse | A Node Warehouse which contains instructions for building nodes referenced in the Omega matrix but not in the network. |
| defaultRule | This should be a character scalar giving the name of a CPTtools combination rule (see Compensatory). With the regression model assumed in the algorithm, currently "Compensatory" is the only value that makes sense. |
| defaultLink | This should be a character scalar giving the name of a CPTtools link function (see normalLink). With the regression model assumed in the algorithm, currently "normalLink" is the only value that makes sense. |
| defaultAlpha | A numeric scalar giving the default value for slope parameters. |
| defaultBeta | A numeric scalar giving the default value for difficulty (negative intercept) parameters. |
| defaultLinkScale | A positive number which gives the default value for the link scale parameter. |
| defaultPriorWeight | A positive number which gives the default value for the node prior weight hyperparameter. |
| debug | A logical scalar. If true then recover is called after an error, so that the node in question can be inspected. |
| override | A logical value. If false, differences between any existing structure in the graph and the Omega matrix will raise an error. If true, the graph will be modified to conform to the matrix. |
| addTvals | A logical value. If true, nodes which do not have state values set, will have those state values set using the function effectiveThetas . |

Details

Whittaker (1990) noted that a normal Bayesian network (one in which all nodes followed a standard normal distribution) could be described using the inverse of the covariance matrix, often denoted Omega. In particular, zeros in the inverse covariance matrix represented variables which were conditionally independent, and therefore reducing the matrix to one with positive and zero values could provide the structure for a graphical model. Almond (2010) proposed using this as the basis

for specifying discrete Bayesian networks for the proficiency model in educational assessments (especially as correlation matrixes among latent variables are a possible output of a factor analysis).

The Omega matrix is represented with a `data.frame` object which contains two square submatrixes and a couple of auxiliary columns. The first column should be named “Node” and contains the names of the nodes. This defines a collection of *nodes* which are defined in the Omega matrix. Let J be the number of nodes (rows in the data frame). The next J columns should have the names the *nodes*. Their values give the structural component of the matrix. The following two columns are “Link” and “Rules” these give the name of the combination rule and link function to use for this row. Next follows another series J “A” columns, each should have a name of the form “A.*node*”. This defines a matrix A containing regression coefficients. Finally, there should be two additional columns, “Intercept” and “PriorWeight”.

Let Q be the logical matrix formed by the J columns after the first and let A be the matrix of coefficients. The matrix Q gives the structure of the graph with $Q[i, j]$ being true when Node j is a parent of node i . By convention, $Q[j, j] = 1$. Note that unlike the inverse covariance matrix from which it gets its name, this matrix is not symmetric. It instead reflects the (possibly arbitrary) directions assigned to the edges. Except for the main diagonal, $Q[i, j]$ and $Q[j, i]$ will not both be 1. Note also, that $A[i, j]$ should be positive only when $Q[i, j] = 1$. This provides an additional check that structures were correctly entered if the Omega matrix is being used for data entry.

When the link function is set to `normalLink` and the rules is set of `Compensatory` the model is described as a series of regressions. Consider Node j which has K parents. Let θ_j be a real value corresponding to that node and let θ_k be a real (standard normal) value representing Parent Node k . a_k represent the corresponding coefficient from the A -table. Let $\sigma_j = a_{j,j}$ that is the diagonal element of the A -table corresponding to the variable under consideration. Let b_j be the value of the intercept column for Node j . Then the model specifies that θ_j has a normal distribution with mean

$$\frac{1}{\sqrt{K}} \sum a_k \theta_k + b_j,$$

and standard deviation σ_j . The regression is discretized to calculate the conditional probability table (see `normalLink` for details).

Note that the parameters are deliberately chosen to look like a regression model. In particular, b_j is a normal intercept and not a difficulty parameter, so that in general `PnodeBetas` applied to the corresponding node will have the opposite sign. The $1/\sqrt{K}$ term is a variance stabilization parameter so that the variance of θ_j will not be affected by number of parents of Node j . The multiple R-squared for the regression model is

$$\frac{1/K \sum a_k^2}{1/K \sum a_k^2 + \sigma_j^2}.$$

This is often a more convenient parameter to elicit than σ_j .

The function `Omega2Pnet` attempts to make adjustments to its `pnet` argument, which should be a `Pnet`, so that it conforms to the information given in the Omega matrix. Nodes are created as necessary using information in the `nodewarehouse` argument, which should be a `Warehouse` object whose `manifest` includes instructions for building the nodes in the network. The warehouse supply function should either return an existing node in `pnet` or create a new node in `pnet`. The structure of the graph is adjusted to correspond to the Q-matrix (structural part of the data frame). If the value of the `override` argument is false, an error is raised if there is existing structure with a different topology. If `override` is true, then the `pnet` is destructively altered to conform to the structural information in the Omega matrix.

The “Link” and “Rules” columns are used to set the values of `PnodeLink(node)` and `PnodeRules(node)`. The off-diagonal elements of the A-matrix are used to set `PnodeAlphas(node)` and the diagonal elements to set `PnodeLinkScale(node)`. The values in the “Intercept” column are the negatives of the values `PnodeBetas(node)`. Finally, the values in the “PriorWeight” column correspond to the values of `PnodePriorWeight(node)`. In any of these cases, if the value in the Omega matrix is missing, then the default value will be supplied instead.

One challenge is setting up a matrix with the correct structure. If the nodes have been defined, the the `Pnet2Omega` can be used to create a blank matrix with the proper format which can then be edited.

Value

The network `pnet` is returned. Note that it is destructively modified by the commands to conform to the Omega matrix.

Omega Matrix Structure

An Omega Matrix should be an object of class `data.frame` with number of rows equal to the number of nodes. Throughout let *node* stand for the name of a node.

Node The name of the node described in this column.

node One column for each node. The value in this column should be 1 if the node in the column is regarded as a parent of the node referenced in the row.

Link The name of a link function. Currently, “`normalLink`” is the only value supported.

Rules The name of the combination rule to use. Currently, “`Compensatory`” is recommended.

A.node One column for each node. This should be a positive value if the corresponding *node* column has a 1. This gives the regression coefficient. If *node* corresponds to the current row, this is the residual standard deviation rather than a regression coefficient. See details.

Intercept A numeric value giving the change in prevalence for the two variables (see details).

PriorWeight The amount of weight which should be given to the current values when learning conditional probability tables. See `PnodePriorWeight`.

Logging and Debug Mode

As of version 0.6-2, the meaning of the debug argument is changed. In the new version, the `flog.logger` mechanism is used for progress reports, and error reporting. In particular, setting `flog.threshold(DEBUG)` (or TRACE) will cause progress reports to be sent to the logging output.

The debug argument has been repurposed. It now call `recover` when the error occurs, so that the problem can be debugged.

Side Effects

This function destructively modifies `pnet` and nodes referenced in the `Qmat` and supplied by the warehouses.

Note that unlike typical R implementations, this is not necessarily safe. In particular, if the `Qmat` references 10 node, and an error is raised when trying to modify the 5th node, the first 4 nodes will be modified, the last 5 will not be and the 5th node may be partially modified. This is different from most R functions where changes are not committed unless the function returns successfully.

Note

While the Omega matrix allows the user to specify both link function and combination rule, the description of the Bayesian network as a series of regressions only really makes sense when the link function is [normalLink](#) and the combination rule is [Compensatory](#). These are included for future expansion.

The representation, using a single row of the data frame for each node in the graph, only works well with the normal link function. In particular, both the partial credit and graded response links require the ability to specify different intercepts for different states of the variable, something which is not supported in the Omega matrix. Furthermore, the [OffsetConjunctive](#) rule requires multiple intercepts. Presumably the [Conjunctive](#) rule could be used, but the interpretation of the slope parameters is then unclear. If the variables need a model other than the compensatory normal model, it might be better to use a Q-matrix (see [Pnet2Qmat](#) to describe the variable.

Author(s)

Russell Almond

References

- Whittaker, J. (1990). *Graphical Models in Applied Multivariate Statistics*. Wiley.
- Almond, R. G. (2010). 'I can name that Bayesian network in two matrixes.' *International Journal of Approximate Reasoning*. **51**, 167-178.
- Almond, R. G. (presented 2017, August). Tabular views of Bayesian networks. In John-Mark Agosta and Tomas Singlair (Chair), *Bayesian Modeling Application Workshop 2017*. Symposium conducted at the meeting of Association for Uncertainty in Artificial Intelligence, Sydney, Australia. (International) Retrieved from <http://bmaw2017.azurewebsites.net/>

See Also

The inverse operation is [Pnet2Omega](#).

See [Warehouse](#) for description of the node warehouse argument.

See [normalLink](#) and [Compensatory](#) for more information about the mathematical model.

The node attributes set from the Omega matrix include: [PnodeParents](#)(*node*), [PnodeLink](#)(*node*), [PnodeLinkScale](#)(*node*), [PnodeRules](#)(*node*), [PnodeAlphas](#)(*node*), [PnodeBetas](#)(*node*), and [PnodePriorWeight](#)(*node*)

Examples

```
## Sample Omega matrix.
omegamat <- read.csv(system.file("auxdata", "miniPP-omega.csv",
                               package="Peanut"),
                    row.names=1, stringsAsFactors=FALSE)

## Not run:
library(PNetica) ## Needs PNetica
sess <- NeticaSession()
startSession(sess)

curd <- getwd()
```

```

netman1 <- read.csv(system.file("auxdata", "Mini-PP-Nets.csv",
                             package="Peanut"),
                  row.names=1,stringsAsFactors=FALSE)

nodeman1 <- read.csv(system.file("auxdata", "Mini-PP-Nodes.csv",
                              package="Peanut"),
                   stringsAsFactors=FALSE)

## Insures we are building nets from scratch
setwd(tempdir())
## Network and node warehouse, to create networks and nodes on demand.
Nethouse <- BNWarehouse(manifest=netman1,session=sess,key="Name")

Nodehouse <- NNWarehouse(manifest=nodeman1,
                        key=c("Model","NodeName"),
                        session=sess)
CM <- WarehouseSupply(Nethouse,"miniPP_CM")
CM1 <- Omega2Pnet(omegamat,CM,Nodehouse,override=TRUE,debug=TRUE)

Om2 <- Pnet2Omega(CM1,NetworkAllNodes(CM1))

DeleteNetwork(CM)
stopSession(sess)
setwd(curd)

## End(Not run)

```

Pnet

A Parameterized Bayesian network

Description

A parameterized Bayesian network. Note that this an abstract class. If an object implements the Pnet protocol, then `is.Pnet(net)` should return TRUE.

Usage

```

is.Pnet(x)
as.Pnet(x)
Pnet(net, priorWeight=10, pnodes=list())
## S4 method for signature 'ANY'
Pnet(net, priorWeight=10, pnodes=list())

```

Arguments

| | |
|-------------|--|
| x | A object to test to see if it a parameterized network, or to coerce into a parameterized network. |
| net | A network object which will become the core of the Pnet. Note that this should probably already be another kind of network, e.g., a NeticaBN object. |
| priorWeight | A numeric vector providing the default prior weight for nodes. |
| pnodes | A list of objects which can be coerced into node objects. Note that the function does not do the coercion. |

Details

The Pnet class is basically a protocol which any Bayesian network net object can follow to work with the tools in the Peanut package. This is really an abstract class (in the java programming language, Pnet would be an interface rather than a class). In particular, a Pnet is any object for which `is.Pnet` returns true. The default method looks for the string "Pnet" in the class list.

A Pnet object has two "fields" (implemented through the accessor methods). The function `PnetPnodes` returns a list of parameterized nodes or [Pnodes](#) associate with the network. The function `PnetPriorWeight` gets (or sets) the default weight to be used for each node.

The default constructor adds "Pnet" to the class of net and then sets the two fields using the accessor functions. There is no default method for the `as.Pnet` function.

In addition to the required fields, there are several optional fields. The methods `PnetName()`, `PnetTitle()`, `PnetDescription()`, and `PnetPathname()` all provide generic setters and getters for mostly self-explanatory properties of the network. For model fragments (such as evidence models) which are meant to be ajoined to other networks, the accessor `PnetHub()` returns the name of the network to which it is to be adjoined (such as a proficiency model). These optional feilds are referenced by the function `BuildNetManifest()` which builds a table of meta-data from which to construct a network.

The Pnet supports hub-and-spoke architectures for Bayes nets. The hub is a complete Bayesian network to which spokes, network fragments are attached. For example, in a typical educational testing application, the central student proficiency model will be the hub, and the evidence models linking the proficiency variables to the observable outcomes, will be the spokes. Only the spokes corresponding to the tasks on a given test form need to be attached to draw inferences. Spoke models are generally model fragments because they contain "stub" nodes, references to nodes in the corresponding hub model. The function `PnetHub()` returns or sets the name of the hub model for a spoke. For a hub net, this function returns `character(0)` or NULL. The function `PnetMakeStubNodes()` will create stub node objects in the spoke model, and the function `PnetRemoveStubNodes()` will remove them. These are called before and after creating graph structures in `Qmat2Pnet`. The functions `PnetAdjoin()` and `PnetDetach()` adjoin a hub and spoke node, matching the stub variables with their real counterparts and detach them (reversing the process).

The importance of the Pnet object is that it supports the `GEMfit` method which adjust the parameters of the Pnode objects to fit a set of case data. In order to be compatible with `GEMfit`, the Pnet object must support four methods: `BuildAllTables`, `calcPnetLLike`, `calcExpTables`, and `maxAllTableParams`.

The generic function `BuildAllTables` builds conditional probability tables from the current values of the parameters in all Pnodes. The default method loops through all of the nodes in `PnetPnodes` and calls the function `BuildTable` on each.

The generic function `calcPnetLLike` calculates the log likelihood of a set of cases given the current values of the parameters. There is no default for this method as its implementation is dependent.

The generic function `calcExpTables` calculates expected cross-tabs for all CPT for the Pnodes given a set of case data. The easiest way to do this is to run the EM algorithm for an unconstrained hyper-Dirichlet model for one or two cycles. There is no default for this as its implementation is dependent.

The generic function `maxAllTableParams` calculates the parameters that maximize the fit to the expected tables for each Pnode. The default method loops over `PnetPnodes(net)` and applies the method `maxCPTParam` to each.

Value

The function `is.Pnet` returns a logical scalar indicating whether or not the object claims to follow the Pnet protocol.

The function `as.Pnet` and `Pnet` convert the argument into a Pnet and return that.

Author(s)

Russell Almond

References

Almond, R. G. (2015) An IRT-based Parameterization for Conditional Probability Tables. Paper presented at the 2015 Bayesian Application Workshop at the Uncertainty in Artificial Intelligence Conference.

See Also

Fields: `PnetPriorWeight`, `PnetPnodes`

Generic Functions: `BuildAllTables`, `calcPnetLLike`, `calcExpTables`, `maxAllTableParams`, `PnetName()`, `PnetTitle()`, `PnetDescription()`, `PnetPathname()`, `PnetAdjoin()`, `PnetDetach()`, `PnetMakeStubNodes()`, `PnetRemoveStubNodes()`, `PnetFindNode()`

Functions: `GEMfit`, `BuildNetManifest`, `Pnet2Qmat`, `Pnet2Omega`, `Qmat2Pnet`, `Omega2Pnet`

Related Classes: `Pnode`, `Warehouse`

Examples

```
## Not run:

library(PNetica) ## Implementation of Peanut protocol
sess <- NeticaSession()
startSession(sess)
## Create network structure using RNetica calls
IRT10.2PL <- CreateNetwork("IRT10_2PL", session=sess)

theta <- NewDiscreteNode(IRT10.2PL, "theta",
                        c("VH", "High", "Mid", "Low", "VL"))
PnodeStateValues(theta) <- effectiveThetas(PnodeNumStates(theta))
PnodeProbs(theta) <- rep(1/PnodeNumStates(theta), PnodeNumStates(theta))
```

```

J <- 10 ## Number of items
items <- NewDiscreteNode(IRT10.2PL,paste("item",1:J,sep=""),
                        c("Correct","Incorrect"))
for (j in 1:J) {
  PnodeParents(items[[j]]) <- list(theta)
  PnodeStateValues(items[[j]]) <- c(1,0)
  PnodeLabels(items[[j]]) <- c("observables")
}

## Convert into a Pnet
IRT10.2PL <- Pnet(IRT10.2PL,priorWeight=10,pnodes=items)

## Draw random parameters
btrue <- rnorm(J)
lnatruue <- rnorm(J)/sqrt(3)
dump(c("btrue","lnatruue"),"IRT10.2PL.params.R")

## Convert nodes to Pnodes
for (j in 1:J) {
  items[[j]] <- Pnode(items[[j]],lnatruue[j],btrue[j])
}
BuildAllTables(IRT10.2PL)
is.Pnet(IRT10.2PL)
WriteNetworks(IRT10.2PL,"IRT10.2PL.true.dne")

DeleteNetwork(IRT10.2PL)
stopSession(sess)

## End(Not run)

```

Pnet-class

Class "Pnet"

Description

This is a virtual class. Classes implementing the Pnet protocol should attach themselves using [setIs](#).

Note that NULL is always considered a member so that uninitialized in containers.

Objects from the Class

A virtual Class: No objects may be created from it.

Classes can register as belonging to this abstract class. The trick for doing this is: `setIs("NetClass", "Pnet")`

Currently [NeticaBN](#) is an example of an object of this class (but requires the PNetica package to provide all of the required functionality).

Methods

No methods defined with class "Pnet" in the signature; however, the following generic functions are available:

- PnetName** signature(net = "Pnet"): Fetches network name.
- PnetName<-** signature(net = "Pnet", value="character"): Sets network name.
- PnetTitle** signature(net = "Pnet"): Fetches network title.
- PnetTitle<-** signature(net = "Pnet", value="character"): Sets network title.
- PnetHub** signature(net = "Pnet"): Fetches name of hub (Proficiency model) if this is a spoke network (Evidence model).
- PnetHub<-** signature(net = "Pnet", value): Sets name of hub model.
- PnetPathname** signature(net = "Pnet"): Fetches name of file in which network is saved.
- PnetPathname<-** signature(net = "Pnet", value): Sets name of file in which network is saved.
- PnetDescription** signature(net = "Pnet"): Fetches documentation string for network.
- PnetDescription<-** signature(net = "Pnet", value="character"): Sets documentation string for network.
- PnetFindNode** signature(net = "Pnet", name="character"): Finds a node by name.
- PnetMakeStubNodes** signature(net = "Pnet", nodes = "list"): Copies nodes from hub model into spoke model.
- PnetRemoveStubNodes** signature(net = "Pnet", nodes = "list"): Removes copied nodes from hub model.
- PnetAdjoin** signature(hub = "Pnet", spoke = "Pnet"): Attaches spoke to hub, matching stub nodes in spoke with their counterparts in the hub.
- PnetDetach** signature(motif = "Pnet", spoke = "Pnet"): Removes the spoke from the motif (combined hub and spoke).
- PnetCompile** signature(net = "Pnet"): Performs topological transformations on the net to make it ready for inference.
- PnetSerialize** signature(net = "Pnet"): Saves the net to a string which can be stored in a database.
- PnetUnserialize** signature(serial = "character"): Reverses the above procedure.
- unserializePnet** signature(factory, data): this is an improved version of unserialize that assumes a store of networks.

Pwned

R really doesn't want me to do this. I'm just having a lot of difficulty creating a class that extends something from a different package.

For now, need to use "ANY" instead of "Pnet" and then do explicit type checking with `is.Pnet`.

Author(s)

Russell Almond

See Also

[Pnet](#).

The class [NeticaBN](#) implements this protocol.

Examples

```
showClass("Pnet")
## Not run:
  setIs("NeticaBN", "Pnet")

## End(Not run)
```

Pnet2Omega

Constructs an Omega matrix from a parameterized network.

Description

An Omega matrix (represented as a data frame) is a structure which describes a Bayesian network as a series of regressions from the parent nodes to the child nodes. It actually contains two matrixes, one giving the structure and the other the regression coefficients. If the parameters have not yet been added to nodes, then the function will use the supplied default values allowing the parameters to later be defined through the use of the function [Pnet2Omega](#).

Usage

```
Pnet2Omega(net, prof, defaultRule = "Compensatory", defaultLink = "normalLink",
           defaultAlpha = 1, defaultBeta = 0, defaultLinkScale = 1, debug = FALSE)
```

Arguments

| | |
|------------------|--|
| net | A Pnet object containing the network to be described. |
| prof | A list of Pnode objects which will become the rows and columns of the matrix. |
| defaultRule | This should be a character scalar giving the name of a CPTtools combination rule (see Compensatory). With the regression model assumed in the algorithm, currently "Compensatory" is the only value that makes sense. |
| defaultLink | This should be a character scalar giving the name of a CPTtools link function (see normalLink). With the regression model assumed in the algorithm, currently "normalLink" is the only value that makes sense. |
| defaultAlpha | A numeric scalar giving the default value for slope parameters. |
| defaultBeta | A numeric scalar giving the default value for difficulty (negative intercept) parameters. |
| defaultLinkScale | A positive number which gives the default value for the link scale parameter. |
| debug | A logical value. If true, extra information will be printed during process of building the Omega matrix. |

Details

Whittaker (1990) noted that a normal Bayesian network (one in which all nodes followed a standard normal distribution) could be described using the inverse of the covariance matrix, often denoted Omega. In particular, zeros in the inverse covariance matrix represented variables which were conditionally independent, and therefore reducing the matrix to one with positive and zero values could provide the structure for a graphical model. Almond (2010) proposed using this as the basis for specifying discrete Bayesian networks for the proficiency model in educational assessments (especially as correlation matrixes among latent variables are a possible output of a factor analysis).

The Omega matrix is represented with a `data.frame` object which contains two square submatrixes and a couple of auxiliary columns. The first column should be named “Node” and contains the names of the nodes. This defines a collection of *nodes* which are defined in the Omega matrix. Let J be the number of nodes (rows in the data frame). The next J columns should have the names the *nodes*. Their values give the structural component of the matrix. The following two columns are “Link” and “Rules” these give the name of the combination rule and link function to use for this row. Next follows another series J “A” columns, each should have a name of the form “A.*node*”. This defines a matrix A containing regression coefficients. Finally, there should be two additional columns, “Intercept” and “PriorWeight”.

Let Q be the logical matrix formed by the J columns after the first and let A be the matrix of coefficients. The matrix Q gives the structure of the graph with $Q[i, j]$ being true when Node j is a parent of node i . By convention, $Q[j, j] = 1$. Note that unlike the inverse covariance matrix from which it gets its name, this matrix is not symmetric. It instead reflects the (possibly arbitrary) directions assigned to the edges. Except for the main diagonal, $Q[i, j]$ and $Q[j, i]$ will not both be 1. Note also, that $A[i, j]$ should be positive only when $Q[i, j] = 1$. This provides an additional check that structures were correctly entered if the Omega matrix is being used for data entry.

When the link function is set to `normalLink` and the rules is set of `Compensatory` the model is described as a series of regressions. Consider Node j which has K parents. Let θ_j be a real value corresponding to that node and let θ_k be a real (standard normal) value representing Parent Node k . Let a_k represent the corresponding coefficient from the A -table. Let $\sigma_j = a_{j,j}$ that is the diagonal element of the A -table corresponding to the variable under consideration. Let b_j be the value of the intercept column for Node j . Then the model specifies that θ_j has a normal distribution with mean

$$\frac{1}{\sqrt{K}} \sum a_k \theta_k + b_j,$$

and standard deviation σ_j . The regression is discretized to calculate the conditional probability table (see `normalLink` for details).

Note that the parameters are deliberately chosen to look like a regression model. In particular, b_j is a normal intercept and not a difficulty parameter, so that in general `PnodeBetas` applied to the corresponding node will have the opposite sign. The $1/\sqrt{K}$ term is a variance stabilization parameter so that the variance of θ_j will not be affected by number of parents of Node j . The multiple R-squared for the regression model is

$$\frac{1/K \sum a_k^2}{1/K \sum a_k^2 + \sigma_j^2}.$$

This is often a more convenient parameter to elicit than σ_j .

The function `Pnet2Omega` builds an Omega matrix from an existing `Pnet`. Only the nodes specified in the `prof` argument are included in the matrix, each row corresponding to a node. The values in

the “Node” column are taken from `PnodeName(node)`. The values in the structural part of the matrix are taken from the graphical structure, specifically `PnodeParents(node)`. The “Link” and “Rules” columns are taken from `PnodeLink(node)` and `PnodeRules(node)`. The off-diagonal elements of the A-matrix are taken from the values of `PnodeAlphas(node)` and the diagonal elements from `PnodeLinkScale(node)`. The values in the “Intercept” column are the negatives of the values `PnodeBetas(node)`. Finally, the values in the “PriorWeight” column correspond to the values of `PnodePriorWeight(node)`; note that a value of NA indicates that the prior weight should be taken from the `Pnet`.

If the nodes do not yet have the various parameters set, then this function will create a blank Omega matrix, with default values (set from various optional arguments) for entries where the parameters have not yet been set. This matrix can then be edited and read back in with `Omega2Pnet` as a way of setting the parameters of the network.

Value

An object of class `(OmegaMat,data.frame)` with number of rows equal to the number of nodes. Throughout let *node* stand for the name of a node.

| | |
|-----------------------------------|---|
| Node | The name of the node described in this column. |
| $\backslash\text{var}\{node\}$ | One column for each node. The value in this column should be 1 if the node in the column is regarded as a parent of the node referenced in the row. |
| Link | The name of a link function. Currently, “ <code>normalLink</code> ” is the only value supported. |
| Rules | The name of the combination rule to use. Currently, “ <code>Compensatory</code> ” is recommended. |
| A. $\backslash\text{var}\{node\}$ | One column for each node. This should be a positive value if the corresponding <i>node</i> column has a 1. This gives the regression coefficient. If <i>node</i> corresponds to the current row, this is the residual standard deviation rather than a regression coefficient. See details. |
| Intercept | A numeric value giving the change in prevalence for the two variables (see details). |
| PriorWeight | The amount of weight which should be given to the current values when learning conditional probability tables. See <code>PnodePriorWeight</code> . |

Note

While the Omega matrix allows the user to specify both link function and combination rule, the description of the Bayesian network as a series of regressions only really makes sense when the link function is `normalLink` and the combination rule is `Compensatory`. These are included for future expansion.

The representation, using a single row of the data frame for each node in the graph, only works well with the normal link function. In particular, both the partial credit and graded response links require the ability to specify different intercepts for different states of the variable, something which is not supported in the Omega matrix. Furthermore, the `OffsetConjunctive` rule requires multiple intercepts. Presumably the `Conjunctive` rule could be used, but the interpretation of the slope parameters is then unclear. If the variables need a model other than the compensatory normal model, it might be better to use a Q-matrix (see `Pnet2Qmat` to describe the variable).

Author(s)

Russell Almond

References

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

Almond, R. G. (2010). 'I can name that Bayesian network in two matrixes.' *International Journal of Approximate Reasoning*. **51**, 167-178.

Almond, R. G. (presented 2017, August). Tabular views of Bayesian networks. In John-Mark Agosta and Tomas Singlair (Chair), *Bayesian Modeling Application Workshop 2017*. Symposium conducted at the meeting of Association for Uncertainty in Artificial Intelligence, Sydney, Australia. (International) Retrieved from <http://bmaw2017.azurewebsites.net/>

See Also

The inverse operation is [Omega2Pnet](#).

See [normalLink](#) and [Compensatory](#) for more information about the mathematical model.

The node functions from which the Omega matrix is populated includes: [PnodeParents\(node\)](#), [PnodeLink\(node\)](#), [PnodeLinkScale\(node\)](#), [PnodeRules\(node\)](#), [PnodeAlphas\(node\)](#), [PnodeBetas\(node\)](#), and [PnodePriorWeight\(node\)](#)

Examples

```
## Sample Omega matrix.
omegamat <- read.csv(system.file("auxdata", "miniPP-omega.csv",
                                package="Peanut"),
                    row.names=1, stringsAsFactors=FALSE)

## Not run:
library(PNetica) ## Needs PNetica
sess <- NeticaSession()
startSession(sess)
curd <- getwd()

netman1 <- read.csv(system.file("auxdata", "Mini-PP-Nets.csv",
                                package="Peanut"),
                    row.names=1, stringsAsFactors=FALSE)

nodeman1 <- read.csv(system.file("auxdata", "Mini-PP-Nodes.csv",
                                package="Peanut"),
                     row.names=1, stringsAsFactors=FALSE)

## Insures we are building nets from scratch
setwd(tempdir())
## Network and node warehouse, to create networks and nodes on demand.
Nethouse <- BNWarehouse(manifest=netman1, session=sess, key="Name")

Nodehouse <- NNWarehouse(manifest=nodeman1,
                          key=c("Model", "NodeName"),
                          session=sess)
```

```

CM <- WarehouseSupply(Nethouse,"miniPP_CM")
CM1 <- Omega2Pnet(omegamat,CM,Nodehouse,override=TRUE,debug=TRUE)

Om2 <- Pnet2Omega(CM1,NetworkAllNodes(CM1))

class(omegamat) <- c("OmegMat","data.frame") # To match Pnet2Omega output.
omegamat$PriorWeight <- rep("10",nrow(omegamat))

stopifnot(all.equal(omegamat,Om2))

DeleteNetwork(CM)
stopSession(sess)
setwd(curd)

## End(Not run)

```

Pnet2Qmat

Makes an augmented Q-matrix from a collection of parameterized nets

Description

In augmented Q -matrix, there is a set of rows for each [Pnode](#) which describes the conditional probability table for that node in terms of the model parameters (see [BuildTable](#)). As the Pnodes could potentially come from multiple nets, the key for the table is (“Model”, “Node”). As there are multiple rows per node, “State” is the third part of the key.

The function `Pnet2` creates an augmented Q -matrix out of a collection of [Pnodes](#), possibly spanning multiple [Pnets](#).

Usage

```

Pnet2Qmat(obs, prof, defaultRule = "Compensatory", defaultLink = "partialCredit",
          defaultAlpha = 1, defaultBeta = NULL, defaultLinkScale = NULL, debug = TRUE)

```

Arguments

| | |
|---------------------------|---|
| <code>obs</code> | A list of <i>observable</i> Pnode objects. These could span multiple Pnet objects. Each element of this list will corresponded to one or more rows in the output Q -matrix. |
| <code>prof</code> | A list of <i>proficiency</i> Pnodes. These are the parents of the Pnodes in the obs list. Usually, these are all in a central proficiency or hub model. |
| <code>defaultRule</code> | This should be a character scalar giving the name of a CPTtools combination rule (see Compensatory). |
| <code>defaultLink</code> | This should be a character scalar giving the name of a CPTtools link function (see partialCredit). |
| <code>defaultAlpha</code> | A numeric scalar giving the default value for slope parameters. |

| | |
|------------------|--|
| defaultBeta | A numeric scalar giving the default value for difficulty (negative intercept) parameters. |
| defaultLinkScale | A positive number which gives the default value for the link scale parameter. |
| debug | A logical value. If true, extra information will be printed during process of building the Pnet. |

Details

A Q -matrix is a 0-1 matrix which describes which proficiency (latent) variables are connected to which observable outcome variables; $q_{jk} = 1$ if and only if proficiency variable k is a parent of observable variable j . Almond (2010) suggested that augmenting the Q -matrix with additional columns representing the combination rules ([PnodeRules](#)), link function ([PnodeLink](#)), link scale parameter (if needed, [PnodeLinkScale](#)) and difficulty parameters ([PnodeBetas](#)). The discrimination parameters ([PnodeAlphas](#)) could be overloaded with the Q -matrix, with non-zero parameters in places where there were 1's in the Q -matrix.

This arrangement worked fine with combination rules (e.g., [Compensatory](#)) which contained multiple alpha (discrimination) parameters, one for each parent variable, and a single beta (difficulty). The introduction of a new type of offset rule (e.g., [OffsetDisjunctive](#)) which uses a multiple difficulty parameters, one for each parent variable, and a single alpha. Almond (2016) suggested a new augmentation which has three matrixes in a single table (a Qmat): the Q -matrix, which contains structural information; the A -matrix, which contains discrimination parameters; and the B -matrix, which contains the difficulty parameters. The names for the columns for these matrixes contain the names of the proficiency variables, prepended with “A.” or “B.” in the case of the A -matrix and B -matrix. There are two additional columns marked “A” and “B” which are used for the discrimination and difficulty parameter in the multiple-beta and multiple-alpha cases. There is some redundancy between the Q , A and B matrixes, but this provides an opportunity for checking the validity of the input.

The introduction of the partial credit link function ([partialCredit](#)) added a further complication. With the partial credit model, there could be a separate set of discrimination or difficulty parameters for each transition for a polytomous item. Even the [gradedResponse](#) link function requires a separate difficulty parameter for each level of the variable save the first. The rows of the Qmat data structure are hence augmented to include one row for every state but the lowest-level state. There should be of fewer rows of associated with the node than the value in the “Nstates” column, and the names of the states (values in the “State” column) should correspond to every state of the target variable except the first. It is an error if the number of states does not match the existing node, or if the state names do not match what is already used for the node or is in the manifest for the node [Warehouse](#).

Note that two nodes in different networks may share the same name, and two states in two different nodes may have the same name as well. Thus, the formal key for the Qmat data frame is (“Model”, “Node”, “State”), however, the rows which share the values for (“Model”, “Node”) form a subtable for that particular node. In particular, the rows of the Q -matrix subtable for that node form the *inner Q-matrix* for that node. The inner Q -matrix shows which variables are relevant for each state transition in a partial credit model. The column-wise maximum of the inner Q -matrix forms the row of the outer Q -matrix for that node. This shows which proficiency nodes are the parent of the observable node. This corresponds to [PnodeQ\(node\)](#).

The function `Qmat2Pnet` creates and sets the parameters of the observable [Pnodes](#) referenced in the `Qmat` argument. As it needs to reference, and possibly create, a number of [Pnets](#) and [Pnodes](#), it

requires both a network and a node `Warehouse`. If the `override` parameter is true, the networks will be modified so that each node has the correct parents, otherwise `Qmat2Pnet` will signal an error if the existing network structure is inconsistent with the Q -matrix.

As there is only one link function for each *node*, the values of `PnodeLink(node)` and `PnodeLinkScale(node)` are set based on the values in the “Link” and “LinkScale” columns and the first row corresponding to *node*. Note that the choice of link functions determines what is sensible for the other values but this is not checked by the code.

The value of `PnodeRules(node)` can either be a single value or a list of rule names. The first value in the sub-Qmat must a character value, but if the other values are missing then a single value is used. If not, all of the entries should be non-missing. If this is a single value, then effectively the same combination rule is used for each transition.

The interpretation of the A -matrix and the B -matrix depends on the value in the “Rules” column. There are two types of rules, multiple-A rules and multiple-B rules (offset rules). The CPTtools function `isOffsetRule` checks to see what kind of a rule it is. The multiple-A rules, of which `Compensatory` is the canonical example, have one discrimination (or slope) parameter for every parent variable (values of 1 in the Q -matrix) and have a single difficulty (negative intercept) parameter which is in the “B” column of the Qmat. The multiple-B or offset rules, of which `OffsetConjunctive` is the canonical example, have a difficulty (negative intercept) parameter for each parent variable and a single discrimination (slope) parameter which is in the “A” column. The function `Qmat2Pnet` uses the value of `isOffsetRule` to determine whether to use the multiple-B (true) or multiple-A (false) paradigm.

A simple example is a binary observable variable which uses the `Compensatory` rule. This is essentially a regression model (logistic regression with `partialCredit` or `gradedResponse` link functions, linear regression with `normalLink` link function) on the parent variables. The linear predictor is:

$$\frac{1}{\sqrt{K}}(a_1\theta_1 + \dots + a_K\theta_K) - b.$$

The values $\theta_1, \dots, \theta_K$ are effective thetas, real values corresponding to the states of the parent variables. The value a_i is stored in the column “A.name i ” where *name i* is the name of the i th proficiency variable; the value of `PnodeAlphas(node)` is the vector a_1, \dots, a_k with names corresponding to the parent variables. The value of b is stored in the “B” column; the value of `PnodeBetas(node)` is b .

The multiple-B pattern replaces the A -matrix with the B -matrix and the column “A” with “B”. Consider binary observable variable which uses the `OffsetConjunctive` rule. The linear predictor is:

$$a \min(\theta_1 - b + 1, \dots, \theta_K - b_K).$$

The value b_i is stored in the column “B.name i ” where *name i* is the name of the i th proficiency variable; the value of `PnodeBetas(node)` is the vector b_1, \dots, b_k with names corresponding to the parent variables. The value of a is stored in the “A” column; the value of `PnodeBetas(node)` is a .

When there are more than two states in the output variable, `PnodeRules`, `PnodeAlphas(node)` and `PnodeBetas(node)` become lists to indicate that a different value should be used for each transition between states. If there is a single value in the “Rules” column, or equivalently the value of `PnodeRules` is a scalar, then the same rule is repeated for each state transition. The same is true for `PnodeAlphas(node)` and `PnodeBetas(node)`. If these values are a list, that indicates that a different value is to be used for each transition. If they are a vector that means that different values (of discriminations for multiple-a rules or difficulties for multiple-b rules) are needed for the parent

variables, but the same set of values is to be used for each state transition. If different values are to be used then the values are a list of vectors.

The necessary configuration of a 's and b 's depends on the type of link function. Here are the rules for the currently existing link functions:

normal (`normalLink`) This link function uses the same linear predictor for each transition, so there should be a single rule, and `PnodeAlphas(node)` and `PnodeBetas(node)` should both be vectors (with b of length 1 for a multiple-a rule). This rule also requires a positive value for the `PnodeLinkScale(node)` in the "LinkScale" column. The values in the "A.name" and "B.name" for rows after the first can be left as NA's to indicate that the same values are reused.

graded response (`gradedResponse`) This link function models the probability of getting at or above each state and then calculates the differences between them to produce the conditional probability table. In order to avoid negative probabilities, the probability of being in a higher state must always be nonincreasing. The surest way to ensure this is to both use the same combination rules at each state and the same set of discrimination parameters for each state. The difficulty parameters must be nondecreasing. Again, values for rows after the first can be left as NAs to indicate that the same value should be reused.

partial credit (`partialCredit`) This link function models the conditional probability from moving from the previous state to the current state. As such, there is no restriction on the rules or parameters. In particular, it can alternate between multiple-a and multiple-b style rules from row to row.

Another restriction that the use of the partial credit rule lifts is the restriction that all parent variable must be used in each transition. Note that there is one row of the Q -matrix (the inner Q -matrix) for each state transition. Only the parent variables with 1's in the particular state row are considered when building the `PnodeAlphas(node)` and `PnodeBetas(node)` for this model. Note that only the partial credit link function can take advantage of the multiple parents, the other two require all parents to be used for every state.

The function `Pnet2Qmat` takes a collection of nodes (in a series of spoke or evidence models) and builds a `Qmat` data structure that can reproduce them. It loops through the nodes and fills out the `Qmat` based on the properties of the `Pnodes`. Note that if the properties are not yet set, then the default values are used, thus applying this to a network for which the structure has been established, but the parameters have not yet been set will build a blank `Qmat` which can be adjusted by experts.

Value

The output augmented Q -matrix is a data frame with the columns described below. The number of columns is variable, with items marked *prof* actually corresponding to a number of columns with names taken from the proficiency variables (the *prof* argument).

| | |
|---------|--|
| Model | The name of the <code>Pnet</code> in which the node in this row lives. |
| Node | The name of the <code>Pnode</code> described in this row. Except for the multiple rows corresponding to the same node, the value of this column needs to be unique within "Model". |
| Nstates | The number of states for this node. Generally, each node should have one fewer rows than this number. |
| State | The name of the state for this row. This should be unique within the ("Model", "Node") combination. |

| | |
|--|---|
| Link | The name of a link function. This corresponds to <code>PnodeLink(node)</code> . |
| LinkScale | Either a positive number giving the link scale parameter or an NA if the link function does not need scale parameters. This corresponds to <code>PnodeLinkScale(node)</code> . |
| $\backslash\text{var}\{\text{prof}\}$ | There is one column for each proficiency variable. This corresponds to the structural part of the Q -matrix. There should be 1 in this column if the named proficiency is used in calculating the transition to this state for this particular node, and a 0 otherwise. |
| Rules | The name of the combination rule to use for this row. This corresponds to <code>PnodeRules(node)</code> . |
| A. $\backslash\text{var}\{\text{prof}\}$ | There is one column for each proficiency with the proficiency name appended to “A.”. If a multiple-alpha style combination rule (e.g., <code>Compensatory</code>) this column should contain the appropriate discriminations, otherwise, its value should be NA. |
| A | If a multiple-beta style combination rule (e.g., <code>OffsetConjunctive</code>) this column should contain the single discrimination, otherwise, its value should be NA. |
| B. $\backslash\text{var}\{\text{prof}\}$ | There is one column for each proficiency with the proficiency name appended to “B.”. If a multiple-beta style combination rule (e.g., <code>OffsetConjunctive</code>) this column should contain the appropriate difficulty (negative intercept), otherwise, its value should be NA. |
| B | If a multiple-beta style combination rule (e.g., <code>Compensatory</code>) this column should contain the single difficulty (negative intercept), otherwise, its value should be NA. |
| PriorWeight | The amount of weight which should be given to the current values when learning conditional probability tables. See <code>PnodePriorWeight</code> . |

Author(s)

Russell Almond

References

Almond, R. G. (2010). ‘I can name that Bayesian network in two matrixes.’ *International Journal of Approximate Reasoning*. **51**, 167-178.

Almond, R. G. (presented 2017, August). Tabular views of Bayesian networks. In John-Mark Agosta and Tomas Singlair (Chair), *Bayesian Modeling Application Workshop 2017*. Symposium conducted at the meeting of Association for Uncertainty in Artificial Intelligence, Sydney, Australia. (International) Retrieved from <http://bmaw2017.azurewebsites.net/>

See Also

The inverse operation is `Qmat2Pnet`.

See `Warehouse` for description of the network and node warehouse arguments

See `partialCredit`, `gradedResponse`, and `normalLink` for currently available link functions. See `Conjunctive` and `OffsetConjunctive` for more information about available combination rules.

The node attributes set from the Omega matrix include: `PnodeParents(node)`, `PnodeLink(node)`, `PnodeLinkScale(node)`, `PnodeRules(node)`, `PnodeQ(node)`, `PnodeAlphas(node)`, `PnodeBetas(node)`, and `PnodePriorWeight(node)`

Examples

```

## Sample Q matrix
Q1 <- read.csv(system.file("auxdata", "miniPP-Q.csv", package="Peanut"),
               stringsAsFactors=FALSE)

## Not run:
library(PNetica) ## Needs PNetica
sess <- NeticaSession()
startSession(sess)
curd <- getwd()

netman1 <- read.csv(system.file("auxdata", "Mini-PP-Nets.csv",
                              package="Peanut"),
                   row.names=1,stringsAsFactors=FALSE)

nodeman1 <- read.csv(system.file("auxdata", "Mini-PP-Nodes.csv",
                                 package="Peanut"),
                    row.names=1,stringsAsFactors=FALSE)

omegamat <- read.csv(system.file("auxdata", "miniPP-omega.csv",
                                 package="Peanut"),
                    row.names=1,stringsAsFactors=FALSE)

## Insures we are building nets from scratch
setwd(tempdir())
## Network and node warehouse, to create networks and nodes on demand.
Nethouse <- BNWarehouse(manifest=netman1,session=sess,key="Name")

Nodehouse <- NNWarehouse(manifest=nodeman1,
                        key=c("Model","NodeName"),
                        session=sess)

## Build the proficiency model first:
CM <- WarehouseSupply(Nethouse,"miniPP_CM")
CM1 <- Omega2Pnet(omegamat,CM,Nodehouse,override=TRUE)

## Build the nets from the Qmat

Qmat2Pnet(Q1, Nethouse,Nodehouse)

## Build the Qmat from the nets
## Generate a list of nodes
obs <-unlist(sapply(list(sess$nets$PPcompEM,sess$nets$PPconjEM,
                        sess$nets$PPtwostepEM,sess$nets$PPdurAttEM),
                    NetworkAllNodes))

Q2 <- Pnet2Qmat(obs,NetworkAllNodes(CM))

## adjust Q1 to match Q2
Q1 <- Q1[,-1] ## Drop unused first column.
class(Q1) <- c("Qmat", "data.frame")

```

```

# Force them into the same order
Q1 <- Q1[order(Q1$Model,Q1$Node),]
Q2 <- Q2[order(Q2$Model,Q2$Node),]
row.names(Q1) <- NULL
row.names(Q2) <- NULL

## Force all NA columns into the right type
Q1$LinkScale <- as.numeric(Q1$LinkScale)
Q1$A.Physics <- as.numeric(Q1$A.Physics)
Q1$A.IterativeD <- as.numeric(Q1$A.IterativeD)
Q1$B.Physics <- as.numeric(Q1$B.Physics)
Q1$B.NTL <- as.numeric(Q1$B.NTL)

## Fix fancy quotes added by some spreadsheets
Q1$Rules <- gsub(intToUtf8(c(91,0x201C,0x201D,93)), "\"", Q1$Rules)

## Insert Default Prior Weights
Q1$PriorWeight <- ifelse(is.na(Q1$NStates), "", "10")
all.equal(Q1, Q2)

stopSession(sess)
setwd(curd)

## End(Not run)

```

PnetAdjoin

Merges (or separates) two Pnets with common variables

Description

In the hub-and-spoke Bayes net construction method, number of spoke models (evidence models in educational applications) are connected to a central hub model (proficiency models in educational applications). The PnetAdjoin operation combines a hub and spoke model to make a motif, replacing references to hub variables in the spoke model with the actual hub nodes. The PnetDetach operation reverses this.

Usage

```

PnetAdjoin(hub, spoke)
PnetDetach(motif, spoke)

```

Arguments

| | |
|-------|---|
| hub | A complete Pnet to which new variables will be added. |
| spoke | An incomplete Pnet which may contain stub nodes, references to nodes in the hub |
| . | |
| motif | The combined Pnet which is formed by joining a hub and spoke together. |

Details

The hub-and-spoke model for Bayes net construction (Almond and Mislevy, 1999; Almond, 2017) divides a Bayes net into a central hub model and a collection of spoke models. The motivation is that the hub model represents the status of a system—in educational applications, the proficiency of the student—and the spoke models are related to collections of evidence that can be collected about the system state. In the educational application, the spoke models correspond to a collection of observable outcomes from a test item or task. A *motif* is a hub plus a collection of spoke model corresponding to a single task.

While the hub model is a complete Bayesian network, the spoke models are fragments. In particular, several hub model variables are parents of variables in the spoke model. These variables are not defined in spoke model, but are rather replaced with *stub nodes*, nodes which reference, but do not define the spoke model.

The PnetAdjoin operation copies the *Pnodes* from the spoke model into the hub model, and connects the stub nodes to the nodes with the same name in the spoke model. The result is a motif consisting of the hub and the spoke. (If this operation is repeated many times it can be used to build an arbitrarily complex motif.)

The PnetDetach operation reverses the adjoin operation. It removes the nodes associated with the spoke model only, leaving the joint probability distribution of the hub model (along with any evidence absorbed by setting values of observable variables in the spoke) intact.

Value

The function PnetAdjoin returns a list of the newly created nodes corresponding to the spoke model nodes. Note that the names may have changed to avoid duplicate names. The names of the list are the spoke node names, so that any name changes can be discovered.

In both cases, the first argument is destructively modified, for PnetAdjoin the hub model becomes the motif. For PnetDetach the motif becomes the hub again.

Note

Node names must be unique within a Bayes net. If several spokes are attached to a hub and those spokes have common names for observable variables, then the names will need to be modified to make them unique. The function PnetAdjoin always returns the new nodes so that any name changes can be noted by the calling program.

I anticipate that there will be considerable variation in how these functions are implemented depending on the underlying implementation of the Bayes net package. In particular, there is no particular need for the PnetDetach function to do anything. While removing variables corresponding to an unneeded spoke model make the network smaller, they are harmless as far as calculations of the posterior distribution.

Author(s)

Russell Almond

References

Almond, R. G. & Mislevy, R. J. (1999) Graphical models and computerized adaptive testing. *Applied Psychological Measurement*, 23, 223–238.

Almond, R., Herskovits, E., Mislevy, R. J., & Steinberg, L. S. (1999). Transfer of information between system and evidence models. In *Artificial Intelligence and Statistics 99, Proceedings* (pp. 181–186). Morgan-Kaufman

Almond, R. G. (presented 2017, August). Tabular views of Bayesian networks. In John-Mark Agosta and Tomas Singlair (Chair), *Bayesian Modeling Application Workshop 2017*. Symposium conducted at the meeting of Association for Uncertainty in Artificial Intelligence, Sydney, Australia. (International) Retrieved from <http://bmaw2017.azurewebsites.net/>

See Also

[Pnet](#), [PnetHub](#), [Qmat2Pnet](#), [PnetMakeStubNodes](#)

Examples

```
## Not run:
library(PNetica) # Requires PNetica
sess <- NeticaSession()
startSession(sess)

PM <- ReadNetworks(system.file("testnets", "miniPP-CM.dne",
                             package="PNetica"), session=sess)
EM1 <- ReadNetworks(system.file("testnets", "PPcompEM.dne",
                              package="PNetica"), session=sess)

Phys <- PnetFindNode(PM,"Physics")

## Prior probability for high level node
PnetCompile(PM)
bel1 <- PnodeMargin(PM, Phys)

## Adjoin the networks.
EM1.obs <- PnetAdjoin(PM,EM1)
PnetCompile(PM)

## Enter a finding
PnodeEvidence(EM1.obs[[1]]) <- "Right"
## Posterior probability for high level node

bel2 <- PnodeMargin(PM,Phys)

PnetDetach(PM,EM1)
PnetCompile(PM)

## Findings are unchanged
bel2a <- PnodeMargin(PM,Phys)
stopifnot(all.equal(bel2,bel2a,tol=1e-6))

DeleteNetwork(list(PM,EM1))
stopSession(sess)

## End(Not run)
```

PnetCompile

Compiles a Parameterized Bayesian Network

Description

This function requests that the Bayes net be compiled—transformed so that inference can be carried out.

Usage

```
PnetCompile(net)
```

Arguments

net A [Pnet](#) object to be compiled.

Details

Many Bayesian network algorithm have two phases. The graph is built as an acyclic directed graph. Before inference is carried out, the graph is transformed into a structure called a *Junction Tree*, *Tree of Cliques* or *Markov Tree* (Almond, 1995).

This function requests that implementation specific processing, particularly, building the appropriate Markov Tree, be done for the net, so that it can be placed in inference mode instead of editing mode.

Value

The compile net argument should be returned.

Note

It should be harmless to call this function on a net which is already compiled.

Author(s)

Russell Almond

References

Almond, R. G. (1995). *Graphical Belief Models*. Chapman and Hall.

See Also

The following functions will likely return errors if the net is not compiled: [PnodeEvidence](#), [calcStat](#), [PnodeMargin](#), [PnodeEAP](#), [PnodeSD](#), [PnodeMedian](#), [PnodeMode](#).

Examples

```

## Not run:

library(PNetica) ## Need a specific implementation
sess <- NeticaSession()
startSession(sess)

irt10.base <- ReadNetworks(system.file("testnets", "IRT10.2PL.base.dne",
                                     package="PNetica"),session=sess)
irt10.base <- as.Pnet(irt10.base) ## Flag as Pnet, fields already set.
irt10.theta <- PnetFindNode(irt10.base,"theta")
irt10.items <- PnetPnodes(irt10.base)
## Flag items as Pnodes
for (i in 1:length(irt10.items)) {
  irt10.items[[i]] <- as.Pnode(irt10.items[[i]])
}
## Make some statistics
marginTheta <- Statistic("PnodeMargin","theta","Pr(theta)")
meanTheta <- Statistic("PnodeEAP","theta","EAP(theta)")
sdTheta <- Statistic("PnodeSD","theta","SD(theta)")
medianTheta <- Statistic("PnodeMedian","theta","Median(theta)")
modeTheta <- Statistic("PnodeMedian","theta","Mode(theta)")

BuildAllTables(irt10.base)
PnetCompile(irt10.base) ## Netica requirement

calcStat(marginTheta,irt10.base)
calcStat(meanTheta,irt10.base)
calcStat(sdTheta,irt10.base)
calcStat(medianTheta,irt10.base)
calcStat(modeTheta,irt10.base)

PnodeEvidence(irt10.items[[1]]) <- "Correct"

calcStat(marginTheta,irt10.base)
calcStat(meanTheta,irt10.base)
calcStat(sdTheta,irt10.base)
calcStat(medianTheta,irt10.base)
calcStat(modeTheta,irt10.base)

DeleteNetwork(irt10.base)
stopSession(sess)

## End(Not run)

```

Description

The function `PnetFindNode` finds a node in a [Pnet](#) with the given name. If no node with the specified name found, it will return `NULL`.

Usage

```
PnetFindNode(net, name)
```

Arguments

| | |
|-------------------|---|
| <code>net</code> | The Pnet to search. |
| <code>name</code> | A character vector giving the name or names of the desired nodes. |

Details

Although each [Pnode](#) belongs to a single network, a network contains many nodes. Within a network, a node is uniquely identified by its name. However, nodes can be renamed (see [PnodeName\(\)](#)).

Value

The [Pnode](#) object or list of `Pnode` objects corresponding to names,

Author(s)

Russell Almond

See Also

[Pnode](#), [Pnet](#)

[PnodeNet](#) retrieves the network for the node.

Examples

```
## Not run:
library(PNetica) # Requires PNetica
sess <- NeticaSession()
startSession(sess)

tnet <- CreateNetwork("TestNet",sess)
nodes <- NewDiscreteNode(tnet,c("A","B","C"))

nodeA <- PnetFindNode(tnet,"A")
stopifnot (nodeA==nodes[[1]])

nodeBC <- PnetFindNode(tnet,c("B","C"))
stopifnot(nodeBC[[1]]==nodes[[2]])
stopifnot(nodeBC[[2]]==nodes[[3]])

DeleteNetwork(tnet)
stopSession(sess)
```

```
## End(Not run)
```

| | |
|---------|--|
| PnetHub | <i>Returns the name of the hub net if this is a spoke net.</i> |
|---------|--|

Description

The hub-and-spoke model divides a complete model up into a central hub model (call a proficiency or competency model in educational applications) and spoke models (or evidence models) which reference variables in the hub network. If a network is a spoke, then the field PnetHub should be set to the name of the corresponding hub network.

Usage

```
PnetHub(net)
PnetHub(net) <- value
```

Arguments

| | |
|-------|---|
| net | A Pnet object whose hub name is to be accessed. |
| value | A character scalar giving the name of the new hub network. |

Value

The getter method returns either a character vector of length 1 giving the name of the hub, or NA or the empty string if no hub is set.

The setter method returns the net argument.

Author(s)

Russell Almond

References

Almond, R. G. & Mislevy, R. J. (1999) Graphical models and computerized adaptive testing. *Applied Psychological Measurement*, 23, 223–238.

Almond, R., Herskovits, E., Mislevy, R. J., & Steinberg, L. S. (1999). Transfer of information between system and evidence models. In *Artificial Intelligence and Statistics 99, Proceedings* (pp. 181–186). Morgan-Kaufman

Almond, R. G. (presented 2017, August). Tabular views of Bayesian networks. In John-Mark Agosta and Tomas Singlair (Chair), *Bayesian Modeling Application Workshop 2017*. Symposium conducted at the meeting of Association for Uncertainty in Artificial Intelligence, Sydney, Australia. (International) Retrieved from <http://bmaw2017.azurewebsites.net/>

See Also

[Pnet](#), [PnetAdjoin](#) (for merging hub and spoke), [Qmat2Pnet](#), [Pnet2Qmat](#)

Examples

```
## Not run:
library(PNetica) # Requires PNetica
sess <- NeticaSession()
startSession(sess)
curd <- setwd(system.file("testnets", package="PNetica"))

PM <- ReadNetworks("miniPP-CM.dne", session=sess)
stopifnot(PnetHub(PM)=="")

EM1 <- ReadNetworks("PPcompEM.dne", session=sess)
stopifnot(PnetHub(EM1)=="miniPP_CM")

foo <- CreateNetwork("foo", sess)
stopifnot(is.na(PnetHub(foo)))
PnetHub(foo) <- PnetName(PM)
stopifnot(PnetHub(foo)=="miniPP_CM")

DeleteNetwork(list(PM, EM1, foo))
stopSession(sess)
setwd(curd)

## End(Not run)
```

PnetMakeStubNodes *Creates (or removes) references to nodes in a network*

Description

A *stub node* is a reference in a *spoke* network to a node in a *hub* network. The function `PnetMakeStubNodes` makes stub nodes in the spoke network. The function `RemoveStubNodes` removes them.

Usage

```
PnetMakeStubNodes(net, nodes)
PnetRemoveStubNodes(net, nodes)
```

Arguments

| | |
|--------------------|---|
| <code>net</code> | A Pnet object in which the stub nodes will be created or removed. This is generally a spoke (evidence model) network. |
| <code>nodes</code> | A list of Pnode objects. In the case of <code>PnetMakeStubNodes</code> these are the nodes in the hub model which are to be copied. In the case of <code>PnetRemoveStubNodes</code> these are the stub nodes to be removed. |

Details

In the hub-and-spoke model, spoke models (evidence models) reference nodes in the central hub model (proficiency model in educational applications). The *stub node* is a node (or pseudo-node) in the spoke model which is actually a reference to a node in the hub model. In the operation `PnetAdjoin` when the spoke model is combined with the hub model, the stubs are replaced with the actual nodes they represent.

The pair of functions `PnetMokeStubNodes` and `PnetRemoveStubNodes` are used inside of `Qmat2Pnet` to create the necessary references to the proficiency nodes (in the columns of the Q -matrix) while building the conditional probability tables for the observable nodes (the rows of the Q -matrix). The function `PnetMakeStubNodes` gets called before the conditional probability tables are built, and the function `PnetRemoveStubNodes` gets called after all conditional probability tables are built.

Value

The function `PnetMakeStubNodes` returns a list of the newly created stub nodes.

The return of the function `PnetRemoveStubNodes` is implementation dependent, and is called mainly for its side effects.

Both functions destructively modify the net argument.

Note

The behavior of these functions will depend a lot on the underlying implementation, and they should be thought of as a pair. The function `PnetMakeStubNodes` gets called before constructing the conditional probability tables, and `PnetRemoveStubNodes`. For example, this could be used to give the nodes the official hub node name while constructing the conditional probability tables and then rename them to something else.

In the `PNetica-package` implementation, the function `PnetMakeStubNodes` copies the nodes from the hub to the spoke, and the function `PnetRemoveStubNodes` deletes them (which if they are attached as a parent, automatically creates a stub node in Netica).

Author(s)

Russell Almond

References

Almond, R. G. (presented 2017, August). Tabular views of Bayesian networks. In John-Mark Agosta and Tomas Singlair (Chair), *Bayesian Modeling Application Workshop 2017*. Symposium conducted at the meeting of Association for Uncertainty in Artificial Intelligence, Sydney, Australia. (International) Retrieved from <http://bmaw2017.azurewebsites.net/>

See Also

`PnetHub(spoke)` give the name of the hub node for a given spoke.

The function `PnetAdjoin(hub, spoke)` merges hub and spoke networks replacing the stubs with the originals in the hub network.

The function `Qmat2Pnet` uses `PnetMakeStubNodes` and `PnetRemoveStubNodes` internally.

Examples

```

## Not run:
library(PNetica) ## Needs PNetica
sess <- NeticaSession()
startSession(sess)

PM <- ReadNetworks(system.file("testnets", "miniPP-CM.dne",
                             package="PNetica"), session=sess)
EM1 <- ReadNetworks(system.file("testnets", "PPcompEM.dne",
                              package="PNetica"), session=sess)

## Find the target node and its parents.
obs <- PnetFindNode(EM1,"CompensatoryObs")
pars <- PnetFindNode(PM,c("NTL","POfMom"))

## Make stub nodes for the parents
stubs <- PnetMakeStubNodes(EM1,pars)
## Set them as the parents
PnodeParents(obs) <- stubs

## Build the CPT
PnodeLink(obs) <- "partialCredit"
PnodeRules(obs) <- "Compensatory"
PnodeAlphas(obs) <- c(NTL=0.9,POfMom=1.1)
PnodeBetas(obs) <- 0.3
PnodeQ(obs) <- TRUE
BuildTable(obs)

##Done, now remove the stubs
PnetRemoveStubNodes(EM1,stubs)

DeleteNetwork(list(PM,EM1))
stopSession(sess)

## End(Not run)

```

PnetName

Gets or Sets the name of a Netica network.

Description

Gets or sets the name of the network. Names must generally conform to the network naming convention of the host Bayesian network system. In particular, they should probably follow the rules for R variable names.

Usage

```

PnetName(net)
## S4 method for signature 'NULL'

```

```
PnetName(net)
PnetName(net) <- value
```

Arguments

| | |
|-------|---|
| net | A Pnet object. |
| value | A character scalar containing the new name. |

Details

Network names must conform to the rules for the host Bayes net system 'q'. Trying to set the network to a name that does not conform to the rules will produce an error, as will trying to set the network name to a name that corresponds to another different network.

The [PnetTitle\(\)](#) function provides another way to name a network which is not subject to the variable restrictions.

Value

The name of the network as a character vector of length 1.

The setter method returns the modified object.

True Names

True names are the names in the secret ancient language which hold power over an object (Le Guin, 1968).

Actually, this is a difficulty with implementations that place restrictions on the name of a network or node. In particular, Netica restricts node names to alphanumeric characters and limits the length. This may make it difficult to match nodes by name with other parts of the system which do not have this restriction. In this case the object may have both a *true name*, which is returned by `PnodeName` and an internal *use name* which is used by the implementation.

Author(s)

Russell Almond

References

Le Guin, U. K. (1968). *A Wizard of Earthsea*. Parnassus Press.

See Also

[Pnet](#), [PnetTitle\(\)](#)

Examples

```
## Not run:
library(PNetica) ## Requires PNetica
sess <- NeticaSession()
startSession(sess)
net <- CreateNetwork("funNet",sess)
stopifnot(PnetName(net)=="funNet")

PnetName(net)<-"SomethingElse"
stopifnot(PnetName(net)=="SomethingElse")

DeleteNetwork(net)
stopSession(sess)

## End(Not run)
```

| | |
|--------------|--|
| PnetPathname | <i>Returns the path associated with a network.</i> |
|--------------|--|

Description

A [Pnet](#) is associated with a filename where it is stored. This value should get set when the network is read or written. Note that this will usually be the name of the network with a implementation file type.

Usage

```
PnetPathname(net)
PnetPathname(net) <- value
```

Arguments

| | |
|-------|---|
| net | A Pnet Bayesian network. |
| value | A character scalar giving the pathname for the network. |

Value

The getter form returns a character vector of length 1. The setter form return the Pnet argument.

Author(s)

Russell Almond

See Also

[Pnet](#)

Examples

```
## Not run:
library(PNetica) # Requires PNetica
sess <- NeticaSession()
startSession(sess)
curd <- setwd(system.file("testnets", package="PNetica"))

PM <- ReadNetworks("miniPP-CM.dne", session=sess)
stopifnot(PnetPathname(PM)=="miniPP-CM.dne")
PnetPathname(PM) <- "StudentModel1.dne"
stopifnot(PnetPathname(PM)=="StudentModel1.dne")

DeleteNetwork(PM)

stopSession(sess)
setwd(curd)

## End(Not run)
```

PnetPnodes

Returns a list of Pnodes associated with a Pnet

Description

Each [Pnet](#) object maintains a list of [Pnode](#) objects which it is intended to set. The function `PnetPnodes` accesses this list. The function `PnodeNet` returns a back pointer to the `Pnet` from the `Pnode`.

Usage

```
PnetPnodes(net)
PnetPnodes(net) <- value
PnodeNet(node)
```

Arguments

| | |
|--------------------|---|
| <code>net</code> | A Pnet object. |
| <code>node</code> | A Pnode object. |
| <code>value</code> | A list of <code>Pnode</code> objects associated with <code>net</code> . |

Details

The primary purpose of `PnetPnodes` is to provide a list of nodes which [GEMfit](#) and [BuildAllTables](#) will iterate to do their function.

The function `PnodeNet` returns the network object associated with the node (this assumes that the implementation has back pointers). Note that `node` may not be in the result of `PnetPnodes` (if for example, the conditional probability table of `node` is to remain fixed during a call to [GEMfit](#)). This function is used by [GetPriorWeight](#) to get the default prior weight if `node` does not have that value set locally.

Value

The function `PnetPnodes` returns a list of `Pnode` objects associated with the net. The expression `PnetPnodes(net) <- value` returns the net.

The function `PnodeNet` returns the network (`Pnet`) object that contains node.

Note

The functions `PnetPnodes` and `PetPnodes<-` and `PnodeNet` are abstract generic functions, and need specific implementations. See the [PNetica-package](#) for an example.

Author(s)

Russell Almond

References

Almond, R. G. (2015) An IRT-based Parameterization for Conditional Probability Tables. Paper presented at the 2015 Bayesian Application Workshop at the Uncertainty in Artificial Intelligence Conference.

See Also

[Pnet](#), [Pnode](#), [GetPriorWeight](#), [BuildAllTables](#), [GEMfit](#)

Examples

```
## Not run:

library(PNetica) ## Implementation of Peanut protocol
sess <- NeticaSession()
startSession(sess)
## Create network structure using RNetica calls
IRT10.2PL <- CreateNetwork("IRT10_2PL",session=sess)

theta <- NewDiscreteNode(IRT10.2PL,"theta",
                        c("VH","High","Mid","Low","VL"))
PnodeStateValues(theta) <- effectiveThetas(PnodeNumStates(theta))
NodeProbs(theta) <- rep(1/PnodeNumStates(theta),PnodeNumStates(theta))

J <- 10 ## Number of items
items <- NewDiscreteNode(IRT10.2PL,paste("item",1:J,sep=""),
                        c("Correct","Incorrect"))
for (j in 1:J) {
  PnodeParents(items[[j]]) <- list(theta)
  PnodeStateValues(items[[j]]) <- c(1,0)
  PnodeLabels(items[[j]]) <- c("observables")
}
## Convert into a Pnet
IRT10.2PL <- Pnet(IRT10.2PL,priorWeight=10,pnode=items[2:J])
for (j in 2:J) {
  items[[j]] <- Pnode(items[[j]])
}
```

```

}

stopifnot(
  length(PnetPnodes(IRT10.2PL)) == J-1, # All except item 1
  PnodeNet(items[[2]]) == IRT10.2PL,
  PnodeNet(items[[1]]) == IRT10.2PL # this is net membership, not
                                     # Pnodes field
)

PnetPnodes(IRT10.2PL) <- items ## Add back item 1
stopifnot(
  length(PnetPnodes(IRT10.2PL)) == J
)
DeleteNetwork(IRT10.2PL)
stopSession(sess)

## End(Not run)

```

| | |
|-----------------|---|
| PnetPriorWeight | <i>Gets the weight to be associated with the prior table during EM learning</i> |
|-----------------|---|

Description

The EM learning algorithm [GEMfit](#) uses the built-in EM learning of the Bayes net to build expected count tables for each [Pnode](#). The expected count tables are a weighted average of the case data and the prior from the parameterized table. This gives the weight, in number of cases, given to the prior.

Usage

```

PnetPriorWeight(net)
PnetPriorWeight(net) <- value
PnodePriorWeight(node)
PnodePriorWeight(node) <- value
GetPriorWeight(node)

```

Arguments

| | |
|-------|---|
| net | A Pnet object whose prior weight is to be accessed. |
| node | A Pnode object whose prior weight is to be accessed. |
| value | A nonnegative numeric vector giving the prior weight. This should either be a scalar or a vector with length equal to the number of rows of the conditional probability table. In the case of <code>PnetPriorWeight</code> using a non-scalar value will produce unpredictable results. |

Details

Suppose that value of the node and all of its parents are fully observed, and let X_{1i}, \dots, X_{ki} be the observed counts for row i , and let p_{1i}, \dots, p_{ki} be the conditional probabilities for row i . Then the posterior probabilities for row i can be found by normalizing $X_{1i} + w_i p_{1i}, \dots, X_{ki} + w_i p_{ki}$. In the EM algorithm, the table is not fully observed but the expected value of X_{1i}, \dots, X_{ki} is used instead.

This function gets or sets the vector w_1, \dots, w_I (where I is the number of rows in the conditional probability table). If value is a scalar this is the same as giving all w_i the same value.

The function `PnodePriorWeight` gets or sets the prior weight for a given node. The function `PnetPriorWeight` gets or sets the default weight for all nodes (a property of the network). Unless all nodes have the name number of parents with the same number of states, this should be a scalar. The expression `GetPriorWeight(node)` gets the prior weight for the node or if that is null, it gets the default prior weight from the net (using the function `PnodeNet`).

Value

A numeric vector or scalar giving the weight or NULL if the default network weight is to be used.

Note

The `GEMfit` algorithm will update the prior weight for each node based on how much information is available for each row. Thus, even if the values are initially the same for each row, after calling `GEMfit` they usually will be different for each row.

The functions `PnetPriorWeight` and `PnodePriorWeight` are abstract generic functions, and they needs specific implementations. See the `PNetica-package` for an example.

Author(s)

Russell Almond

References

Almond, R. G. (2015) An IRT-based Parameterization for Conditional Probability Tables. Paper presented at the 2015 Bayesian Application Workshop at the Uncertainty in Artificial Intelligence Conference.

See Also

[Pnet](#), [Pnode](#), [PnodeNet](#), [BuildTable](#), [GEMfit](#)

Examples

```
## Not run:

library(PNetica) ## Implementation of Peanut protocol
sess <- NeticaSession()
startSession(sess)
## Create network structure using RNetica calls
IRT10.2PL <- CreateNetwork("IRT10_2PL", session=sess)
```

```

theta <- NewDiscreteNode(IRT10.2PL,"theta",
                        c("VH","High","Mid","Low","VL"))
PnodeStateValues(theta) <- effectiveThetas(PnodeNumStates(theta))
PnodeProbs(theta) <- rep(1/PnodeNumStates(theta),PnodeNumStates(theta))

J <- 10 ## Number of items
items <- NewDiscreteNode(IRT10.2PL,paste("item",1:J,sep=""),
                        c("Correct","Incorrect"))
for (j in 1:J) {
  PnodeParents(items[[j]]) <- list(theta)
  PnodeStateValues(items[[j]]) <- c(1,0)
  PnodeLabels(items[[j]]) <- c("observables")
}

## Convert into a Pnet
IRT10.2PL <- as.Pnet(IRT10.2PL)
PnetPriorWeight(IRT10.2PL) <- 10

## Convert nodes to Pnodes
for (j in 1:J) {
  items[[j]] <- Pnode(items[[j]])
}

PnodePriorWeight(items[[2]]) <- 5
## 5 states in parent, so 5 rows
PnodePriorWeight(items[[3]]) <- c(10,7,5,7,10)

stopifnot(
  abs(PnetPriorWeight(IRT10.2PL)-10) < .0001,
  is.null(PnodePriorWeight(items[[1]])),
  abs(GetPriorWeight(items[[1]])-10) < .0001,
  abs(GetPriorWeight(items[[2]])-5) < .0001,
  any(abs(GetPriorWeight(items[[3]])-c(10,7,5,7,10)) < .0001)
)

PnetPriorWeight(IRT10.2PL) <- 15

stopifnot(
  abs(PnetPriorWeight(IRT10.2PL)-15) < .0001,
  is.null(PnodePriorWeight(items[[1]])),
  abs(GetPriorWeight(items[[1]])-15) < .0001,
  abs(GetPriorWeight(items[[2]])-5) < .0001,
  any(abs(GetPriorWeight(items[[3]])-c(10,7,5,7,10)) < .0001)
)

DeleteNetwork(IRT10.2PL)
stopSession(sess)

## End(Not run)

```

| | |
|---------------|---|
| PnetSerialize | <i>Writes/restores network from a string.</i> |
|---------------|---|

Description

The PnetSerialize method writes the network to a string and returns a list containing both the serialized data and type information. The PnetUnserialize method restores the data. Note that the serialized form must contain either the name of the type or the name of the factory used to restore the object (see details).

Usage

```
PnetSerialize(net)
PnetUnserialize(serial)
unserializePnet(factory,data)
WarehouseUnpack(warehouse, serial)
```

Arguments

| | |
|-----------|--|
| net | A Pnet to be serialized. |
| factory | A character scalar containing the name of a global variable which contains a factory object capable of recreating the network from the data. |
| warehouse | A object of the type PnetWarehouse which will contain a link to the appropriate factory. |
| serial | A list containing at least three elements. One is the name of the network. One is the data element which contains the serialized data as a raw vector. The third is either a factory element containing the name of a global symbol containing a factory for reading the object or a type argument giving the name of the constructor. |
| data | A list containing at least two elements. One is the name of the network. One is the data element which contains the serialized data as a raw vector. |

Details

The intention of this function is to serialize the network in such a way that it can be saved to a database and restored. The result of a call to PnetSerialize is a list with three elements. One element is called data and contains the actual serialize data. The second element is called name and it should be an identifier for the network (the result of [PnetName](#)). The last element is either factory or type. In either case, they should be a string. The list may contain other elements, but these may be ignored by other programs.

The intent is to provide a representation that can be saved to a database. The data element should be a raw vector (e.g., the output of `serialize(..., NULL)`) and will be stored as a blob (binary large object) and the other elements should be strings. Document based databases (e.g., [mongo](#)) may handle the additional fields but relational database will have difficulty with them, so implementers should only rely on the three fields.

The function `PnetUnserialize` reverses this operation. If `factory` is supplied, then the factory protocol is used for restoration. If `type` is supplied instead, then the type string protocol is used. If both are supplied, then the factory protocol is preferred, and if neither is supplied, an error is signaled. The function `unserializePnet` is a generic function used by the factory protocol. If a `Pnet` already exists with the given name, then it is replaced, otherwise a new one is created.

Value

The `PnetSerialize` function returns a list with the following elements:

| | |
|----------------------|---|
| <code>name</code> | The name of the network. If this matches an existing network, then it will be replaced on unserialize, otherwise a new network will be created. |
| <code>data</code> | Serialized data for the network. This should be a raw vector. |
| <code>factory</code> | The name of a global object which can restore networks from serialized data. |
| <code>type</code> | The name of a class for which an <code>PnetUnserialize.type</code> method exists. |
| <code>...</code> | There may be other data, but note that programs saving/restoring the serialized representation may not know how to handle these extra fields. |

The `PnetUnserialize` and `unserializePnet` functions return an object of type `Pnet`.

Factory Protocol

A factory is an object of a class for which a method for the `unserializePnet` generic function is defined. This method should return an object of type `Pnet`. Thus the `Peanut` package doesn't need to know the implementation details.

Typically factories are global (static in java language) objects. In this case the factory object should be the name of the factory (as it will need to be serialized). The `get` function is used to retrieve its value, so typically it is stored in `.GlobalEnv`.

The factory protocol allows other kind of flexibility as well, including being able to encapsulate a reference to loaded objects, so this is the preferred method.

Type String Protocol

This mechanism mimics the S3 method dispatch method, although it doesn't really use it. If the argument to `PnetUnserialize` has a `type` field (but no `factory` field) then it will call a function called `PnetUnserialize.type`.

Note

The first use of this function was designed to save/restore a network from a `mongo` database. This format easily supports extra fields in the return list. The something is true if the network is serialized using either JSON/BSON or the normal R `dump` mechanism.

On the other hand, if the network is to be stored in a SQL database, the using program will not have places to store the extra fields.

Author(s)

Russell Almond

See Also

[Pnet, Warehouse](#)

Examples

```
## Not run:
library(mongolite)
library(jsonlite)
library(PNetica)
sess <- NeticaSession()
startSession(sess)

collect <- mongo("studentModels", "test",
                 "mongodb://127.0.0.1:27017/test")
## Or "mongodb://user:pwd@127.0.0.1:27017/test"

## An example network manifest.
netman1 <- read.csv(system.file("auxdata", "Mini-PP-Nets.csv",
                              package="Peanut"),
                  row.names=1, stringsAsFactors=FALSE)
netpath <- system.file("testnets", package="PNetica")
netman1$Pathname <- file.path(netpath, netman1$Pathname)

Nethouse <- BNWarehouse(manifest=netman1, session=sess, key="Name")

pm.net <- WarehouseSupply(Nethouse, "miniPP_CM")
sm.net <- CopyNetworks(pm.net, "Student1")

sm.ser <- PnetSerialize(sm.net)
## base 64 encode the data to make it easier to store.
sm.ser$data <- base64_enc(sm.ser$data)

collect$replace(paste('{"name": "', sm.ser$name, '" }'),
               toJSON(lapply(sm.ser, unbox)),
               upsert=TRUE)

## Use iterator method to find, so we get in list rather than data frame
## representation.

it <- collect$iterate(sprintf('{"name": "%s"}', "Student1"), limit=1)
sm1.ser <- it$one()
## Decode back to the raw vector.
sm1.ser$data <- base64_dec(sm1.ser$data)

DeleteNetwork(sm.net)
sm1 <- WarehouseUnpack(Nethouse, sm1.ser)
stopifnot(PnetName(sm1)=="Student1")

DeleteNetwork(sm1)
sm1a <- unserializePnet(sess, sm1.ser)
stopifnot(PnetName(sm1a)=="Student1")
```

```

DeleteNetwork(sm1a)
#Unserialize needs a reference to the "factory" (in this case session.).
sm1.ser$factory <- "sess"
sm1b <- PnetUnserialize(sm1.ser)
stopifnot(PnetName(sm1b)=="Student1")

stopSession(sess)

## End(Not run)

```

| | |
|-----------|--|
| PnetTitle | <i>Gets the title or comments associated with a parameterized network.</i> |
|-----------|--|

Description

The title is a longer name for a network which is not subject to naming restrictions. The description is free form text used to document the network. Both fields are optional.

Usage

```

PnetTitle(net)
PnetTitle(net) <- value
PnetDescription(net)
PnetDescription(net) <- value

```

Arguments

| | |
|-------|---|
| net | A Pnet object. |
| value | A character object giving the new title or description. |

Details

The title is meant to be a human readable alternative to the name, which is not limited to the network naming restrictions.

The text is any text the user chooses to attach to the network. If value has length greater than 1, the vector is collapsed into a long string with newlines separating the components.

Value

A character vector of length 1 providing the title or description.

Setter methods return the object.

Author(s)

Russell Almond

See Also

[Pnet](#), [PnetName\(\)](#)

Examples

```
## Not run:
library(PNetica) ## Requires PNetica
sess <- NeticaSession()
startSession(sess)

firstNet <- CreateNetwork("firstNet",sess)

PnetTitle(firstNet) <- "My First Bayesian Network"
stopifnot(PnetTitle(firstNet)=="My First Bayesian Network")

now <- date()
PnetDescription(firstNet)<-c("Network created on",now)
## Print here escapes the newline, so is harder to read
cat(PnetDescription(firstNet),"\n")
stopifnot(PnetDescription(firstNet) ==
  paste(c("Network created on",now),collapse="\n"))

DeleteNetwork(firstNet)

stopSession(sess)

## End(Not run)
```

PnetWarehouse-class *Class "PnetWarehouse"*

Description

A [Warehouse](#) object which holds and builds [Pnet](#) objects. In particular, its [WarehouseManifest](#) contains a network manifest (see [BuildNetManifest](#)) which contains information about how to either load the networks from the file system, or build them on demand.

Details

The [PnetWarehouse](#) either supplies prebuilt nets or builds them from the instructions found in the manifest. In particular, the function [WarehouseSupply](#) will attempt to:

1. Find an existing network with name.
2. Try to read the network from the location given in the `Pathname` column of the manifest.
3. Build a blank network, using the metadata in the manifest.

The manifest is an object of type `data.frame` where the columns have the values show below. The key is the “Name” column which should be unique for each row. The `name` argument to `WarehouseData` should be a character scalar corresponding to name, and it will return a `data.frame` with a single row.

Name A character value giving the name of the network. This should be unique for each row and normally must conform to variable naming conventions. Corresponds to the function `PnetName`.

Title An optional character value giving a longer human readable name for the network. Corresponds to the function `PnetTitle`.

Hub If this model is incomplete without being joined to another network, then the name of the hub network. Otherwise an empty character vector. Corresponds to the function `PnetHub`.

Pathname The location of the file from which the network should be read or to which it should be written. Corresponds to the function `PnetPathname`.

Description An optional character value documenting the purpose of the network. Corresponds to the function `PnetDescription`.

The function `BuildNetManifest` will build a manifest for an existing collection of networks.

Objects from the Class

A virtual Class: No objects may be created from it.

Classes can register as belonging to this abstract class. The trick for doing this is: `setIs("NethouseClass", "PnetWarehouse")`

Currently `BNWarehouse` is an example of an object of this class.

Methods

WarehouseSupply signature(`warehouse = "PnetWarehouse"`, `name = "character"`). This finds a network with the appropriate name. If one does not exist, it is created by reading it from the pathname specified in the manifest. If no file exists at the pathname, a new blank network with the properties specified in the manifest is created.

WarehouseFetch signature(`warehouse = "PnetWarehouse"`, `name = "character"`). This fetches the network with the given name, or returns NULL if it has not been built.

WarehouseMake signature(`warehouse = "PnetWarehouse"`, `name = "character"`). This loads the network from a file or builds the network using the data in the Manifest.

WarehouseFree signature(`warehouse = "PnetWarehouse"`, `name = "character"`). This removes the network from the warehouse inventory.

ClearWarehouse signature(`warehouse = "PnetWarehouse"`). This removes all networks from the warehouse inventory.

is.PnetWarehouse signature(`obj = "PnetWarehouse"`). This returns TRUE.

WarehouseManifest signature(`warehouse = "PnetWarehouse"`). This returns the data frame with instructions on how to build networks. (see Details)

WarehouseManifest<- signature(`warehouse = "PnetWarehouse"`, `value="data.frame"`). This sets the data frame with instructions on how to build networks.(see Details)

WarehouseData signature(warehouse = "PnetWarehouse", name="character"). This returns the portion of the data frame with instructions on how to build a particular network. (see Details)

WarehouseUnpack signature(warehouse = "PnetWarehouse", serial="list"). This restores a serialized network, in particular, it is used for saving network state across sessions. See [PnetSerialize](#) for an example.

Note

In the PNetica implementation, the [BNWarehouse](#) implementation contains an embedded [NeticaSession](#) object. When WarehouseSupply is called, it attempts to satisfy the demand by trying in order:

1. Search for the named network in the active networks in the session.
2. If not found in the session, it will attempt to load the network from the Pathname field in the manifest.
3. If the network is not found and there is not file at the target pathname, a new blank network is built and the appropriate fields are set from the metadata.

Author(s)

Russell Almond

See Also

[Warehouse](#), [WarehouseManifest](#), [BuildNetManifest](#)

Implementation in the PNetica package: [BNWarehouse](#), [MakePnet.NeticaBN](#)

Examples

```
## Not run:
library(PNetica) ## Example requires PNetica

sess <- NeticaSession()
startSession(sess)

## BNWarehouse is the PNetica Net Warehouse.
## This provides an example network manifest.
netman1 <- read.csv(system.file("auxdata", "Mini-PP-Nets.csv",
                             package="PNetica"),
                  row.names=1, stringsAsFactors=FALSE)
Nethouse <- BNWarehouse(manifest=netman1, session=sess, key="Name")

CM <- WarehouseSupply(Nethouse, "miniPP_CM")
EM <- WarehouseSupply(Nethouse, "PPcompEM")

DeleteNetwork(list(CM, EM))
stopSession(sess)

## End(Not run)
```

Pnode

A Parameterized Bayesian network node

Description

A node in a parameterized Bayesian network. Note that this is an abstract class. If an object implements the Pnode protocol, then `is.Pnode(node)` should return TRUE.

Usage

```
is.Pnode(x)
as.Pnode(x)
Pnode (node, lnAlphas, betas, rules="Compensatory",
      link="partialCredit", Q=TRUE, linkScale=NULL,
      priorWeight=NULL)
```

Arguments

| | |
|-------------|---|
| x | A object to test to see if it is a parameterized node, or to coerce it to a parameterized node. |
| node | An object that will become the base of the parameterized node. This should already be a parameterized node, e.g., a NeticaNode object. |
| lnAlphas | A numeric vector of list of numeric vectors giving the log slope parameters. See PnodeLnAlphas for a description of this parameter. If missing, the constructor will try to create a pattern of zero values appropriate to the rules argument and the number of parent variables. |
| betas | A numeric vector or list of numeric vectors giving the intercept parameters. See PnodeBetas for a description of this parameter. If missing, the constructor will try to create a pattern of zero values appropriate to the rules argument and the number of parent variables. |
| rules | The combination rule or a list of combination rules. These should either be names of functions or function objects. See PnodeRules for a description of this argument. |
| link | The name of the link function or the link function itself. See PnodeLink for a description of the link function. |
| Q | A logical matrix or the constant TRUE (indicating that the Q-matrix should be a matrix of TRUEs). See PnodeQ for a description of this parameter. |
| linkScale | A numeric vector of link scale parameters or NULL if scale parameters are not needed for the chosen link function. See PnodeLinkScale for a description of this parameter. |
| priorWeight | A numeric vector of weights given to the prior parameter values for each row of the conditional probability table when learning from data (or a scalar if all rows have equal prior weight). See PnodePriorWeight for a description of this parameter. |

Details

The Pnode class is basically a protocol which any Bayesian network node object can follow to work with the tools in the Peanut package. This is really an abstract class (in the java language, Pnode would be an interface rather than a class). In particular, a Pnode is any object for which `is.Pnode` returns true. The default method looks for the string "Pnode" in the class list.

Fields. A Pnode object has eight "fields" (implemented through the accessor methods), which all Pnode objects are meant to support. These correspond to the arguments of the `calcDPCTable` function.

The function `PnodeNet` returns the `Pnet` object which contains the nodes.

The function `PnodeQ` gets or sets a Q-matrix describing which parent variables are relevant for which state transitions. The default value is TRUE which indicates that all parent variables are relevant.

The function `PnodePriorWeight` gets or sets the prior weights associated with the node. This gives the relative weighting of the parameterized table as a prior and the observed data in the `GEMfit` algorithm.

The function `PnodeRules` gets or sets the combination rules used to combine the influence of the parent variables.

The functions `PnodeLnAlphas` and `PnodeAlphas` get or set the slope parameters associated with the combination rules. Note that in many applications, the slope parameters are constrained to be positive and maximization is done over the log of the slope parameter.

The function `PnodeBetas` gets or sets the difficulty (negative intercept) parameter associated with the combination rule.

The function `PnodeLink` gets or sets the link function used to translate between the output of the combination rule and a row of the conditional probability table.

The function `PnodeLinkScale` gets or sets a scale parameter associated with the link function.

There are some additional optional fields which describe metadata about the node and its states. The generic functions `PnodeName()`, `Pnodetitle()`, and `PnodeDescription()` access basic metadata about the node.

The generic function `PnodeLabels()` accesses a set of character labels associated with the node. This is useful for identifying sets of nodes (e.g., observables, high-level proficiency variables.)

The generic functions `PnodeStates()`, `PnodeStateTitles()`, and `PnodeStateDescriptions()` access basic information about the states of the node. The generic function `PnodeNumStates()` returns the number of states. The generic function `PnodeStateValues()` access the numeric values associated with the states.

The generic function `PnodeParents(node)` access the parent set of the *node*. Note that this function has a setter form which changes the topology of the graph. The generic functions `PnodeParentNames()` and `PnodeNumParents()` return the corresponding information about the parent variable.

Generic Functions. The importance of the Pnode object is that it supports the `GEMfit` method which adjust the parameters of the Pnode objects to fit a set of case data. In order to be compatible with `GEMfit`, the Pnode object must support three methods: `PnodeParentTvals`, `BuildTable`, and `maxCPTParam`.

The generic function `PnodeParentTvals` returns a list of effective theta values (vectors of real numbers) associated with the states of the parent variables. These are used to build the conditional probability tables.

The generic function `BuildTable` calls the function `calcDPCTable` to generate a conditional probability table for the node using the current parameter values. It also sets the node experience.

The generic function `maxCPTParam` calls the function `mapDPC` to calculate the optimal parameter values for the CPT for the node and the updates the parameter values.

Value

The function `is.Pnet` returns a logical scalar indicating whether or not the object claims to follow the Pnet protocol.

Author(s)

Russell Almond

References

Almond, R. G. (2015) An IRT-based Parameterization for Conditional Probability Tables. Paper presented at the 2015 Bayesian Application Workshop at the Uncertainty in Artificial Intelligence Conference.

See Also

Parameter Fields: `PnodeQ`, `PnodePriorWeight`, `PnodeRules`, `PnodeLink`, `PnodeLnAlphas`, `PnodeAlphas`, `PnodeBetas`, `PnodeLinkScale`

Metadata fields: `PnodeNet`, `PnodeParents`, `PnodeParentNames`, `PnodeNumParents`, `PnodeName`, `PnodeTitle`, `PnodeDescription`, `PnodeLabels`, `PnodeStates`, `PnodeNumStates`, `PnodeStateTitles`, `PnodeStateDescriptions`, `PnodeStateValues`, `isPnodeContinuous`, `PnodeStateBounds`

Generic Functions: `BuildTable`, `PnodeParentTvals`, `maxCPTParam`

Functions: `GetPriorWeight`, `calcDPCTable`, `mapDPC`

Related Classes: `Pnet`

Examples

```
## Not run:

## These are the implementations of the two key generic functions in
## PNetica

BuildTable.NeticaNode <- function (node) {
  node[] <- calcDPCFrame(ParentStates(node),NodeStates(node),
                        PnodeLnAlphas(node), PnodeBetas(node),
                        PnodeRules(node),PnodeLink(node),
                        PnodeLinkScale(node),PnetQ(node),
                        PnodeParentTvals(node))
  NodeExperience(node) <- GetPriorWeight(node)
  invisible(node)
}

maxCPTParam.NeticaNode <- function (node, Mstepit=3,
```

```

                                tol=sqrt(.Machine$double.eps)) {
  ## Get the posterior pseudo-counts by multiplying each row of the
  ## node's CPT by its experience.
  counts <- sweep(node[[[]],1,NodeExperience(node),"*")
  est <- mapDPC(counts,ParentStates(node),NodeStates(node),
               PnodeLnAlphas(node), PnodeBeta(node),
               PnodeRules(node),PnodeLink(node),
               PnodeLinkScale(node),PnodeQ(node),
               control=list(reltol=tol,maxits=Mstepit)
               )
  PnodeLnAlphas(node) <- est$lnAlphas
  PnodeBetas(node) <- est$betas
  PnodeLinkScale(node) <- est$linkScale
  invisible(node)
}

## End(Not run)

```

Pnode-class

Class "Pnode"

Description

This is a virtual class. Classes implementing the Pnet protocol should attach themselves using [setIs](#).

Note that NULL is always considered a member so that uninitialized in containers.

Objects from the Class

A virtual Class: No objects may be created from it.

Classes can register as belonging to this abstract class. The trick for doing this is: `setIs("NodeClass", "Pnode")`

Currently [NeticaNode](#) is an example of an object of this class (but requires the PNetica package to provide all of the required functionality).

Methods

No methods defined with class "Pnode" in the signature; however, the following generic functions are available:

PnodeName signature(node = "Pnode"): Fetches node name.

PnodeName<- signature(node = "Pnode", value="character"): Sets node name.

PnodeTitle signature(node = "Pnode"): Fetches node title.

- PnodeTitle**<- signature(node = "Pnode", value="character"): Sets node title.
- PnodeDescription** signature(node = "Pnode"): Fetches documentation string for node.
- PnodeDescription**<- signature(node = "Pnode", value="character"): Sets documentation string for node.
- PnodeLabels** signature(node = "Pnode"): Fetches a vector of labels assigned to this node.
- PnodeLabels**<- signature(node = "Pnode", value = "character"): Sets vector of labels assigned to this node. hub model.
- PnodeNumStates** signature(node = "Pnode"): Fetches length of vector of states available for this node.
- PnodeStates** signature(node = "Pnode"): Fetches vector of states available for this node.
- PnodeStates**<- signature(node = "Pnode", value): Sets vector of states for this node.
- PnodeStateTitles** signature(node = "Pnode"): Fetches vector of states available for this node.
- PnodeStateTitles**<- signature(node = "Pnode", value): Sets vector of states for this node.
- PnodeStateDescriptions** signature(node = "Pnode"): Fetches vector of states available for this node.
- PnodeStateDescriptions**<- signature(node = "Pnode", value): Sets vector of states for this node.
- PnodeStateValues** signature(node = "Pnode"): Fetches vector of numeric values associated with states for this node.
- PnodeStateValues**<- signature(node = "Pnode", value): Sets vector of numeric values associated with states for this node.
- PnodeStateBounds** signature(node = "Pnode"): Fetches matrix of upper and lower bounds for discretized states of a continuous node.
- PnodeStateBounds**<- signature(node = "Pnode", value): Sets matrix of upper and lower bounds for discretized states of a continuous node.
- PnodeParents** signature(node = "Pnode"): Fetches a list of the nodes parents.
- PnodeParents**<- signature(node = "Pnode", value = "list"): Sets a list of the nodes parents.
- PnodeParentNames** signature(node = "Pnode"): Lists the names of the parents.
- PnodeNumParents** signature(node = "Pnode"): The length of the parent vector.
- isPnodeContinuous** signature(node = "Pnode"): Copies nodes from hub model into spoke model.
- PnodeProbs** signature(node = "Pnode"): Fetches the conditional probability table for the node.
- PnodeProbs**<- signature(node = "Pnode", value = "array"): Sets the conditional probability table for the node.
- PnodeEvidence** signature(node = "Pnode"): Fetches the current instantiated evidence for this node.
- PnodeEvidence**<- signature(node = "Pnode", value): Sets the instantiated evidence for this node.
- PnodeMargin** signature(node = "Pnode"): Computes the vector of marginal beliefs associated with the state of this node given the evidence.
- PnodeEAP** signature(node = "Pnode"): Computes the expected value of a node given the evidence. This assumes node states are assigned numeric values.

PnodeSD signature(node = "Pnode"): Computes the standard deviation of a node given the evidence. This assumes node states are assigned numeric values.

PnodeMedian signature(node = "Pnode"): Computes the median of a node given the evidence. This assumes node states are ordered.

PnodeMedian signature(node = "Pnode"): Computes the most likely state of a node given the evidence.

Author(s)

Russell Almond

See Also

[Pnode](#).

The class [NeticaNode](#) implements this protocol.

Examples

```
showClass("Pnode")
## Not run:
  setIs("NeticaNode", "Pnode")

## End(Not run)
```

PnodeBetas

Access the combination function slope parameters for a Pnode

Description

In constructing a conditional probability table using the discrete partial credit framework (see [calcDPCTable](#)), the effective thetas for each parent variable are combined into a single effect theta using a combination rule. The expression `PnodeAlphas(node)` accesses the intercept parameters associated with the combination function `PnodeRules(node)`.

Usage

```
PnodeBetas(node)
PnodeBetas(node) <- value
```

Arguments

`node` A [Pnode](#) object.

`value` A numeric vector of intercept parameters or a list of such vectors (see details). The length of the vector depends on the combination rules (see [PnodeRules](#)). If a list, it should have length one less than the number of states in `node`.

Details

Following the framework laid out in Almond (2015), the function `calcDPCTable` calculates a conditional probability table using the following steps:

1. Each set of parent variable states is converted to a set of continuous values called *effective thetas* (see `PnodeParentTvals`). These are built into an array, `eTheta`, using `expand.grid` where each column represents a parent variable and each row a possible configuration of parents.
2. For each state of the node except the last, the set of effective thetas is filtered using the local Q-matrix, `PnodeQ(node) = Q`. Thus, the actual effect thetas for state `s` is `eTheta[,Q[s,]]`.
3. For each state of the node except the last, the corresponding rule is applied to the effective thetas to get a single effective theta for each row of the table. This step is essentially calls the expression: `do.call(rules[[s]], list(eThetas[,Q[s,]], PnodeAlphas(node)[[s]], PnodeBetas(node)[[s]])`.
4. The resulting set of effective thetas are converted into conditional probabilities using the link function `PnodeLink(node)`.

The function `PnodeRules` accesses the function used in step 3. It should be the name of a function or a function with the general signature of a combination function described in `Compensatory`. The compensatory function is a useful model for explaining the roles of the slope parameters, *beta*. Let $\theta_{i,j}$ be the effective theta value for the j th parent variable on the i th row of the effective theta table, and let β_j be the corresponding slope parameter. Then the effective theta for that row is:

$$Z(\theta_i) = (\alpha_1\theta_{i,1} + \dots + \alpha_J\theta_{i,J})/C - \beta,$$

where $C = \sqrt{J}$ is a variance stabilization constant and *alphas* are derived from `PnodeAlphas`. The functions `Conjunctive` and `Disjunctive` are similar replacing the sum with a min or max respectively.

In general, when the rule is one of `Compensatory`, `Conjunctive`, or `Disjunctive`, the the value of `PnodeBetas(node)` should be a scalar.

The rules `OffsetConjunctive`, and `OffsetDisjunctive`, work somewhat differently, in that they assume there is a single slope and multiple intercepts. Thus, the `OffsetConjunctive` has equation:

$$Z(\theta_i) = \text{alphamin}(\theta_{i,1} - \beta_1, \dots, \theta_{i,J} - \beta_J).$$

In this case the assumption is that `PnodeAlphas(node)` will be a scalar and `PnodeBetas(node)` will be a vector of length equal to the number of parents. As a special case, if it is a vector of length 1, then a model with a common slope is used. This looks the same in `calcDPCTable` but has a different implication in `mapDPC` where the parameters are constrained to be the same.

When node has more than two states, there is a different combination function for each transition. (Note that `calcDPCTable` assumes that the states are ordered from highest to lowest, and the transition functions represent transition to the corresponding state, in order.) There are always one fewer transitions than there states. The meaning of the transition functions is determined by the the value of `PnodeLink`, however, both the `partialCredit` and the `gradedResponse` link functions allow for different intercepts for the different steps, and the `gradedResponse` link function requires that the intercepts be in decreasing order (highest first). To get a different intercept for each transition, the value of `PnodeBetas(node)` should be a list.

If the value of `PnodeRules(node)` is a list, then a different combination rule is used for each transition. Potentially, this could require a different number of intercept parameters for each row. Also, if the value of `PnodeQ(node)` is not a matrix of all TRUE values, then the effective number of parents for each state transition could be different. In this case, if the `OffsetConjunctive` or `OffsetDisjunctive` rule is used the value of `PnodeBetas(node)` should be a list of vectors of different lengths (corresponding to the number of true entries in each row of `PnodeQ(node)`).

Value

A list of numeric vectors giving the intercepts for the combination function of each state transition. The vectors may be of different lengths depending on the value of `PnodeRules(node)` and `PnodeQ(node)`. If the intercepts are the same for all transitions then a single numeric vector instead of a list is returned.

Note that the setter form may destructively modify the Pnode object (this depends on the implementation).

Note

The functions `PnodeLnBetas` and `PnodeLnBetas<-` are abstract generic functions, and need specific implementations. See the `PNetica-package` for an example.

The values of `PnodeLink`, `PnodeRules`, `PnodeQ`, `PnodeParentTvals`, `PnodeLnAlphas`, and `PnodeBetas` all need to be consistent for this to work correctly, but no error checking is done on any of the setter methods.

Author(s)

Russell Almond

References

Almond, R. G. (2015) An IRT-based Parameterization for Conditional Probability Tables. Paper presented at the 2015 Bayesian Application Workshop at the Uncertainty in Artificial Intelligence Conference.

Almond, R.G., Mislevy, R.J., Steinberg, L.S., Williamson, D.M. and Yan, D. (2015) *Bayesian Networks in Educational Assessment*. Springer. Chapter 8.

See Also

[Pnode](#), [PnodeQ](#), [PnodeRules](#), [PnodeLink](#), [PnodeLnAlphas](#), [BuildTable](#), [PnodeParentTvals](#), [maxCPTParam](#), [calcDPCTable](#), [mapDPC](#) [Compensatory](#), [OffsetConjunctive](#)

Examples

```
## Not run:
library(PNetica) ## Requires implementation
sess <- NeticaSession()
startSession(sess)

tNet <- CreateNetwork("TestNet", session=sess)
```

```

theta1 <- NewDiscreteNode(tNet,"theta1",
                        c("VH","High","Mid","Low","VL"))
PnodeStateValues(theta1) <- effectiveThetas(PnodeNumStates(theta1))
PnodeProbs(theta1) <- rep(1/PnodeNumStates(theta1),PnodeNumStates(theta1))
theta2 <- NewDiscreteNode(tNet,"theta2",
                        c("VH","High","Mid","Low","VL"))
PnodeStateValues(theta2) <- effectiveThetas(PnodeNumStates(theta2))
PnodeProbs(theta2) <- rep(1/PnodeNumStates(theta2),PnodeNumStates(theta2))

partial3 <- NewDiscreteNode(tNet,"partial3",
                          c("FullCredit","PartialCredit","NoCredit"))
PnodeParents(partial3) <- list(theta1,theta2)

## Usual way to set rules is in constructor
partial3 <- Pnode(partial3,rules="Compensatory", link="gradedResponse")
PnodePriorWeight(partial3) <- 10
BuildTable(partial3)

## increasing intercepts for both transitions
PnodeBetas(partial3) <- list(FullCredit=1,PartialCredit=0)
BuildTable(partial3)

stopifnot(
  all(abs(do.call("c",PnodeBetas(partial3)) -c(1,0) ) <.0001)
)

## increasing intercepts for both transitions
PnodeLink(partial3) <- "partialCredit"
## Full Credit is still rarer than partial credit under the partial
## credit model
PnodeBetas(partial3) <- list(FullCredit=0,PartialCredit=0)
BuildTable(partial3)

stopifnot(
  all(abs(do.call("c",PnodeBetas(partial3)) -c(0,0) ) <.0001)
)

## Switch to rules which use multiple intercepts
PnodeRules(partial3) <- "OffsetConjunctive"

## Make Skill 1 more important for the transition to ParitalCredit
## And Skill 2 more important for the transition to FullCredit
PnodeLnAlphas(partial3) <- 0
PnodeBetas(partial3) <- list(FullCredit=c(-.25,.25),
                          PartialCredit=c(.25,-.25))
BuildTable(partial3)

```

```

## Set up so that first skill only needed for first transition, second
## skill for second transition; Adjust betas to match
PnodeQ(partial3) <- matrix(c(TRUE,TRUE,
                             TRUE,FALSE), 2,2, byrow=TRUE)
PnodeBetas(partial3) <- list(FullCredit=c(-.25,.25),
                             PartialCredit=0)

BuildTable(partial3)

## Can also do this with special parameter values
PnodeQ(partial3) <- TRUE
PnodeBetas(partial3) <- list(FullCredit=c(-.25,.25),
                             PartialCredit=c(0,Inf))

BuildTable(partial3)

DeleteNetwork(tNet)
stopSession(sess)

## End(Not run)

```

PnodeDefaultAlphas *Reshapes alpha or beta vector based on rule and parents*

Description

Combination rules can be sorted into multiple-a rules (e.g., [Compensatory](#)) and multiple-b rules (e.g., [OffsetConjunctive](#)). The function [isOffsetRule](#) distinguishes between the two types. These functions adjust the log alpha or beta matrix to the correct length depending on the rule and parents of the *node* argument.

Usage

```

PnodeDefaultAlphas(node, rule=PnodeRules(node), link=PnodeLink(node) )
PnodeDefaultBetas(node, rule=PnodeRules(node), link=PnodeLink(node))

```

Arguments

| | |
|------|--|
| node | A Pnode object whose PnodeLnAlphas or PnodeBetas field is to be set. |
| rule | A rule (e.g., Compensatory , OffsetConjunctive or the name of a rule. |
| link | A link function (e.g., partialCredit) or the name of a link function. |

Value

A vector of zeros of a suitable length to be used as a default value for [PnodeLnAlphas](#)(*node*) or [PnodeBetas](#)(*node*).

Note

These are used in the PNetica implementation of the [Pnode](#) constructor.

Author(s)

Russell Almond

See Also

[Pnode](#), [PnodeLnAlphas](#), [PnodeBetas](#), [isOffsetRule](#)

Examples

```
## Not run:
library(PNetica) # Requires PNetica
sess <- NeticaSession()
startSession(sess)

EM1 <- ReadNetworks(system.file("testnets", "PPcompEM.dne",
                             package="PNetica"),
                  session=sess)
EM2 <- ReadNetworks(system.file("testnets", "PPconjEM.dne",
                             package="PNetica"),
                  session=sess)

comp <- PnetFindNode(EM1, "CompensatoryObs")
conj <- PnetFindNode(EM2, "ConjunctiveObs")

stopifnot(
  PnodeDefaultAlphas(comp, "Compensatory") == c(0,0),
  PnodeDefaultBetas(comp, "Compensatory") == 0,
  PnodeDefaultAlphas(conj, "OffsetConjunctive") == 0,
  PnodeDefaultBetas(conj, "OffsetConjunctive") == c(0,0)
)

DeleteNetwork(list(EM1, EM2))
stopSession(sess)

## End(Not run)
```

PnodeEvidence

Accesses the value to which a given node has been instantiated.

Description

Inference in a Bayesian network involves setting the state of a particular node to one of its possible states, either because the state has been observed, or because it has been hypothesized. This process is often called *instantiation*. This function returns the value (state) to which the node has been

instantiated, or in the setter form set it. Depending on the implementation logic, the beliefs may be immediately updated or be updated on demand.

Usage

```
PnodeEvidence(node)
PnodeEvidence(node) <- value
```

Arguments

| | |
|-------|---|
| node | A Pnode object whose instantiated value will be accessed. |
| value | The value that the node will be instantiated to, see details. |

Details

Currently, Peanut supports two ways of representing nodes, discrete and continuous (see [isPnodeContinuous](#)). The current [PNetica-package](#) implementation discretizes continuous nodes, using the [PnodeStateBounds](#) to map real numbers to states of the observables. Functions implementing these generic functions may treat these values differently.

The behavior depends on the class of the value argument:

character or factor The character or factor should represent a state of the node. The node will be instantiated to that state.

numeric scalar For continuous nodes, the node will be instantiated to that value. For discretized continuous nodes, the node will be instantiated to the state in which the value lies (see [PnodeStateBounds](#)).

difftime scalar The value is first converted to a numeric value with units of seconds. This can be overridden in the implementation.

numeric vector of length [PnodeNumStates](#) The number should represent likelihoods, and this will enter appropriate virtual evidence for the node.

NULL This will retract any existing evidence associated with the node.

Value

The getter function `PnodeEvidence` will return one of the value forms described in details. If the node is not instantiated, it will return `NULL`.

The setter function `PnodeEvidence<-` returns the node argument invisibly.

Note

The current options for this function make a lot of sense with Netica. There may be other modes that are not covered for other implementations.

Author(s)

Russell Almond

See Also

The function [PnetCompile](#) usually needs to be run before this function has meaning.

The functions [PnodeStates](#) and [PnodeStateBounds](#) define the legal values for the value argument.

Examples

```
## Not run:

library(PNetica) ## Need a specific implementation
sess <- NeticaSession()
startSession(sess)

irt10.base <- ReadNetworks(system.file("testnets", "IRT10.2PL.base.dne",
                                     package="PNetica"),
                          session=sess)
irt10.base <- as.Pnet(irt10.base) ## Flag as Pnet, fields already set.
irt10.theta <- PnetFindNode(irt10.base,"theta")
irt10.items <- PnetPnodes(irt10.base)
## Flag items as Pnodes
for (i in 1:length(irt10.items)) {
  irt10.items[[i]] <- as.Pnode(irt10.items[[i]])
}

BuildAllTables(irt10.base)
PnetCompile(irt10.base) ## Netica requirement

stopifnot (is.na(PnodeEvidence(irt10.items[[1]])))

PnodeEvidence(irt10.items[[1]]) <- "Correct"
stopifnot(PnodeEvidence(irt10.items[[1]])=="Correct")

PnodeEvidence(irt10.items[[1]]) <- NULL
stopifnot (is.na(PnodeEvidence(irt10.items[[1]])))

PnodeEvidence(irt10.items[[1]]) <- c(Correct=.6,Incorrect=.3)
stopifnot(all.equal(PnodeEvidence(irt10.items[[1]]),
                   c(Correct=.6,Incorrect=.3),
                   tol=3*sqrt(.Machine$double.eps) ))

foo <- NewContinuousNode(irt10.base,"foo")

stopifnot(is.na(PnodeEvidence(foo)))

PnodeEvidence(foo) <- 1
stopifnot(PnodeEvidence(foo)==1)

DeleteNetwork(irt10.base)
stopSession(sess)

## End(Not run)
```

| | |
|-------------|---|
| PnodeLabels | <i>Lists or changes the labels associated with a parameterize node.</i> |
|-------------|---|

Description

A label is a character identifier associated with a node which provides information about its role in the models. This function returns or sets the labels associated with a node.

Usage

```
PnodeLabels(node)
PnodeLabels(node) <- value
```

Arguments

| | |
|-------|--|
| node | A Pnode object. |
| value | A character vector containing the names of the labels that <i>node</i> should be associated with. These names should follow the variable naming rules. |

Details

Netica node sets are a collection of string labels that can be associated with various nodes in a network. These have proved to be very useful on writing code as often it is useful to perform some operation on only a certain kind of nodes. One purpose of node sets is to label a set of nodes that play a similar role in the model. For example, "ReportingVariable" or "Observable".

The PnodeLabels function is an attempt to generalize that mechanism. The expression PnodeLabels(*node*) returns the labels currently associated with *node*, thus provides a general mechanism for identifying the roles that a node might play.

The expression PnodeLabels(*node*)<-*value* removes any labels previously associated with *node* and adds the new labels named in *value*. The elements of *value* need not correspond to existing labels, new node sets will be created for new values. (Warning: this implies that if the name of the node set is spelled incorrectly in one of the calls, this will create a new node set. For example, "Observable" and "Observables" would be two distinct labels.)

Two labels have special meaning in the Peanut package. The function [BuildAllTables](#)(*net*) rebuilds the tables for nodes which are labeled "pnode" (i.e., parameterized nodes). The function [GEMfit](#) attempts to fit the parameters for nodes labeled "pnodes", and associates values in the cases argument with the nodes labeled "onodes".

Value

A character vector giving the names of the labels *node* is associated with. The setter form returns *node*.

Author(s)

Russell Almond

See Also

[Pnode](#), [BuildAllTables](#), [GEMfit](#), [PnetPnodes](#)

Examples

```
## Not run:
library(PNetica)##Requires PNetica
sess <- NeticaSession()
startSession(sess)

nsnet <- CreateNetwork("NodeSetExample", session=sess)

Ability <- NewDiscreteNode(nsnet,"Ability",c("High","Med","Low"))

EssayScore <- NewDiscreteNode(nsnet,"EssayScore",paste("level",5:0,sep="_"))

stopifnot(
  length(PnodeLabels(Ability)) == 0L ## Nothing set yet
)

PnodeLabels(Ability) <- "ReportingVariable"
stopifnot(
  PnodeLabels(Ability) == "ReportingVariable"
)
PnodeLabels(EssayScore) <- "Observable"
stopifnot(
  PnodeLabels(EssayScore) == "Observable"
)
## Make EssayScore a reporting variable, too
PnodeLabels(EssayScore) <- c("ReportingVariable",PnodeLabels(EssayScore))
stopifnot(
  setequal(PnodeLabels(EssayScore),c("Observable","ReportingVariable"))
)

## Clear out the node set
PnodeLabels(Ability) <- character()
stopifnot(
  length(PnodeLabels(Ability)) == 0L
)

DeleteNetwork(nsnet)
stopSession(sess)

## End(Not run)
```


Description

In constructing a conditional probability table using the discrete partial credit framework (see [calcDPCTable](#)), the effective thetas for each row of the table is converted into a vector of probabilities using the link function. The function `PnodeLink` accesses the link function associated with a `Pnode`.

Usage

```
PnodeLink(node)
PnodeLink(node) <- value
```

Arguments

| | |
|--------------------|--|
| <code>node</code> | A <code>Pnode</code> object. |
| <code>value</code> | The name of a link function or function object which can serve as the link function. |

Details

Following the framework laid out in Almond (2015), the function `calcDPCTable` calculates a conditional probability table using the following steps:

1. Each set of parent variable states is converted to a set of continuous values called *effective thetas* (see `PnodeParentTvals`). These are built into an array, `eTheta`, using `expand.grid` where each column represents a parent variable and each row a possible configuration of parents.
2. For each state of the node except the last, the set of effective thetas is filtered using the local Q-matrix, `PnodeQ(node) = Q`. Thus, the actual effect thetas for state `s` is `eTheta[, Q[s,]]`.
3. For each state of the node except the last, the corresponding rule is applied to the effective thetas to get a single effective theta for each row of the table. This step is essentially calls the expression: `do.call(rules[[s]], list(eThetas[, Q[s,]], PnodeAlphas(node)[[s]], PnodeBetas(node)[[s]])`.
4. The resulting set of effective thetas are converted into conditional probabilities using the link function.

A link function is a function of three arguments. The first is a matrix of effective theta values with number of rows equal to the number of rows of the conditional probability matrix and number of columns equal to the number of states of node minus one (ordered from highest to lowest). The second is an optional link scale, the third is a set of names for the states which is used to give column names to the output matrix. The second and third both default to `NULL`.

Currently two link functions are `partialCredit` and `gradedResponse`. Note that the function `gradedResponse` assumes that the effective thetas in each row are in increasing order. This puts certain restrictions on the parameter values. Generally, this can only be guaranteed if each state of the variable uses the same combination rules (see `PnodeRules(node)`), slope parameters (see `PnodeAlphas(node)`) and Q-matrix (see `PnodeQ(node)`). Also, the intercepts (see `PnodeBetas(node)`) should be in decreasing order. The `partialCredit` model has fewer restrictions.

The value of `PnodeLinkScale(node)` is fed to the link function. Currently, this is unused; but the DiBello-normal model (see `calcDNTable`) uses it. So the link scale parameter is for future expansion.

Value

A character scalar giving the name of a combination function or a combination function object.

Note that the setter form may destructively modify the Pnode object (this depends on the implementation).

Note

The functions `PnodeLink` and `PnodeLink<-` are abstract generic functions, and need specific implementations. See the `PNetica-package` for an example.

A third normal link function, which would use the scale parameter, is planned but not yet implemented.

Author(s)

Russell Almond

References

Almond, R. G. (2015) An IRT-based Parameterization for Conditional Probability Tables. Paper presented at the 2015 Bayesian Application Workshop at the Uncertainty in Artificial Intelligence Conference.

Almond, R.G., Mislevy, R.J., Steinberg, L.S., Williamson, D.M. and Yan, D. (2015) *Bayesian Networks in Educational Assessment*. Springer. Chapter 8.

See Also

[Pnode](#), [PnodeQ](#), [PnodeRules](#), [PnodeLinkScale](#), [PnodeLnAlphas](#), [PnodeBetas](#), [BuildTable](#), [PnodeParentTvals](#), [maxCPTParam](#), [calcDPCTable](#), [mapDPC](#), [Compensatory](#), [OffsetConjunctive](#)

Examples

```
## Not run:
library(PNetica) ## Requires implementation
sess <- NeticaSession()
startSession(sess)

tNet <- CreateNetwork("TestNet", session=sess)

theta1 <- NewDiscreteNode(tNet, "theta1",
  c("VH", "High", "Mid", "Low", "VL"))
PnodeStateValues(theta1) <- effectiveThetas(PnodeNumStates(theta1))
PnodeProbs(theta1) <- rep(1/PnodeNumStates(theta1), PnodeNumStates(theta1))
theta2 <- NewDiscreteNode(tNet, "theta2",
  c("VH", "High", "Mid", "Low", "VL"))
PnodeStateValues(theta2) <- effectiveThetas(PnodeNumStates(theta2))
```

```

PnodeProbs(theta2) <- rep(1/PnodeNumStates(theta2),PnodeNumStates(theta2))

partial3 <- NewDiscreteNode(tNet,"partial3",
                           c("FullCredit","PartialCredit","NoCredit"))
PnodeParents(partial3) <- list(theta1,theta2)

## Usual way to set link is in constructor
partial3 <- Pnode(partial3,rules="Compensatory", link="gradedResponse")
PnodePriorWeight(partial3) <- 10
PnodeBetas(partial3) <- list(FullCredit=1,PartialCredit=0)
BuildTable(partial3)

## increasing intercepts for both transitions
PnodeLink(partial3) <- "partialCredit"
## Full Credit is still rarer than partial credit under the partial
## credit model
PnodeBetas(partial3) <- list(FullCredit=0,PartialCredit=0)
BuildTable(partial3)

## Can use different slopes with partial credit
## Make Skill 1 more important for the transition to ParitalCredit
## And Skill 2 more important for the transition to FullCredit
PnodeLnAlphas(partial3) <- list(FullCredit=c(-.25,.25),
                               PartialCredit=c(.25,-.25))

BuildTable(partial3)

## Can also use Q-matrix to select skills
## Set up so that first skill only needed for first transition, second
## skill for second transition; Adjust alphas to match
PnodeQ(partial3) <- matrix(c(TRUE,TRUE,
                             TRUE,FALSE), 2,2, byrow=TRUE)
PnodeLnAlphas(partial3) <- list(FullCredit=c(-.25,.25),
                               PartialCredit=0)

BuildTable(partial3)

DeleteNetwork(tNet)

## End(Not run)

```

PnodeLinkScale

Accesses the link function scale parameter associated with a Pnode

Description

In constructing a conditional probability table using the discrete partial credit framework (see [calcDPCTable](#)), the effective thetas for each row of the table is converted into a vector of probabilities using the link function. The function `PnodeLink` accesses the scale parameter of the link function associated with a `Pnode`.

Usage

```
PnodeLinkScale(node)
PnodeLinkScale(node) <- value
```

Arguments

| | |
|-------|---|
| node | A Pnode object. |
| value | A positive numeric value, or NULL if the scale parameter is not used for the link function. |

Details

The link function used in constructing the conditional probability table is controlled by the value of [PnodeLink](#)(node). One of the arguments to the link function is a scale parameter, the expression [PnodeLinkScale](#)(node) provides the link scale parameter associated with the node.

This is mostly for future expansion. Currently, neither of the two link functions defined in the `CPTtools` package, [partialCredit](#) and [gradedResponse](#), require a link scale parameter. However, the DiBello-normal model (see [normalLink](#)) defines the residual variance as a link scale parameter.

Value

The value of the link scale parameter, or NULL if it is not needed.

Note that the setter form may destructively modify the `Pnode` object (this depends on the implementation).

Note

The functions `PnodeLinkScale` and `PnodeLinkScale<-` are abstract generic functions, and need specific implementations. See the [PNetica-package](#) for an example. Even though they are not currently used, they must be defined and return a value (even just NULL).

A third normal link function, which would use the scale parameter, is planned but not yet implemented.

Author(s)

Russell Almond

References

Almond, R. G. (2015) An IRT-based Parameterization for Conditional Probability Tables. Paper presented at the 2015 Bayesian Application Workshop at the Uncertainty in Artificial Intelligence Conference.

Almond, R.G., Mislevy, R.J., Steinberg, L.S., Williamson, D.M. and Yan, D. (2015) *Bayesian Networks in Educational Assessment*. Springer. Chapter 8.

See Also

[Pnode](#), [PnodeQ](#), [PnodeRules](#), [PnodeLinkScale](#), [PnodeLnAlphas](#), [PnodeBetas](#), [BuildTable](#), [PnodeParentTvals](#), [maxCPTParam](#), [calcDPCTable](#), [mapDPC](#), [Compensatory](#), [OffsetConjunctive](#)

Examples

```
## Not run:
library(PNetica) ## Requires implementation
sess <- NeticaSession()
startSession(sess)

tNet <- CreateNetwork("TestNet", session=sess)

theta1 <- NewDiscreteNode(tNet, "theta1",
  c("VH", "High", "Mid", "Low", "VL"))
PnodeStateValues(theta1) <- effectiveThetas(PnodeNumStates(theta1))
PnodeProbs(theta1) <- rep(1/PnodeNumStates(theta1), PnodeNumStates(theta1))
theta2 <- NewDiscreteNode(tNet, "theta2",
  c("VH", "High", "Mid", "Low", "VL"))
PnodeStateValues(theta2) <- effectiveThetas(PnodeNumStates(theta2))
PnodeProbs(theta2) <- rep(1/PnodeNumStates(theta2), PnodeNumStates(theta2))

partial3 <- NewDiscreteNode(tNet, "partial3",
  c("FullCredit", "PartialCredit", "NoCredit"))
PnodeParents(partial3) <- list(theta1, theta2)
partial3 <- Pnode(partial3, rules="Compensatory", link="gradedResponse")
PnodePriorWeight(partial3) <- 10

stopifnot(
  is.null(PnodeLinkScale(partial3))
)

PnodeLinkScale(partial3) <- 1.0

stopifnot(
  all(abs(PnodeLinkScale(partial3)-1)<.0001)
)

DeleteNetwork(tNet)

## End(Not run)
```

PnodeLnAlphas

Access the combination function slope parameters for a Pnode

Description

In constructing a conditional probability table using the discrete partial credit framework (see [calcDPCTable](#)), the effective thetas for each parent variable are combined into a single effect theta

using a combination rule. The expression `PnodeAlphas(node)` accesses the slope parameters associated with the combination function `PnodeRules(node)`. The expression `PnodeLnAlphas(node)` which is used in `mapDPC`.

Usage

```
PnodeLnAlphas(node)
PnodeLnAlphas(node) <- value
PnodeAlphas(node)
PnodeAlphas(node) <- value
## Default S3 method:
PnodeAlphas(node)
## Default S3 replacement method:
PnodeAlphas(node) <- value
```

Arguments

| | |
|--------------------|--|
| <code>node</code> | A <code>Pnode</code> object. |
| <code>value</code> | A numeric vector of (log) slope parameters or a list of such vectors (see details). The length of the vector depends on the combination rules (see <code>PnodeRules</code>). If a list, it should have length one less than the number of states in <code>node</code> . For <code>PnodeAlphas(node) <- value</code> , <code>value</code> should only contain positive numbers. |

Details

Following the framework laid out in Almond (2015), the function `calcDPCTable` calculates a conditional probability table using the following steps:

1. Each set of parent variable states is converted to a set of continuous values called *effective thetas* (see `PnodeParentTvals`). These are built into an array, `eTheta`, using `expand.grid` where each column represents a parent variable and each row a possible configuration of parents.
2. For each state of the node except the last, the set of effective thetas is filtered using the local Q-matrix, `PnodeQ(node) = Q`. Thus, the actual effect thetas for state `s` is `eTheta[, Q[s,]]`.
3. For each state of the node except the last, the corresponding rule is applied to the effective thetas to get a single effective theta for each row of the table. This step is essentially calls the expression: `do.call(rules[[s]], list(eThetas[, Q[s,]], PnodeAlphas(node)[[s]], PnodeBetas(node)[[s]])`.
4. The resulting set of effective thetas are converted into conditional probabilities using the link function `PnodeLink(node)`.

The function `PnodeRules` accesses the function used in step 3. It should be the name of a function or a function with the general signature of a combination function described in `Compensatory`. The compensatory function is a useful model for explaining the roles of the slope parameters, *alpha*. Let $\theta_{i,j}$ be the effective theta value for the j th parent variable on the i th row of the effective theta table, and let α_j be the corresponding slope parameter. Then the effective theta for that row is:

$$Z(\theta_{i,j}) = (\alpha_1 \theta_{i,1} + \dots + \alpha_j \theta_{i,j}) / C - \beta,$$

where $C = \sqrt{J}$ is a variance stabilization constant and β is a value derived from `PnodeBetas`. The functions `Conjunctive` and `Disjunctive` are similar replacing the sum with a min or max respectively.

In general, when the rule is one of `Compensatory`, `Conjunctive`, or `Disjunctive`, the value of `PnodeAlphas(node)` should be a vector of the same length as the number of parents. As a special case, if it is a vector of length 1, then a model with a common slope is used. This looks the same in `calcDPCTable` but has a different implication in `mapDPC` where the parameters are constrained to be the same.

The rules `OffsetConjunctive`, and `OffsetDisjunctive`, work somewhat differently, in that they assume there is a single slope and multiple intercepts. Thus, the `OffsetConjunctive` has equation:

$$Z(\theta_i) = \text{alphamin}(\theta_{i,1} - \beta_1, \dots, \theta_{J,1} - \beta_J).$$

In this case the assumption is that `PnodeAlphas(node)` will be a scalar and `PnodeBetas(node)` will be a vector of length equal to the number of parents.

If the value of `PnodeLink` is `partialCredit`, then the link function can be different for each state of the node. (If it is `gradedResponse` then the curves need to be parallel and the slopes should be the same.) If the value of `PnodeAlphas(node)` is a list (note: list, not numeric vector or matrix), then a different set of slopes is used for each state transition. (This is true whether `PnodeRules(node)` is a single function or a list of functions. Note that if there is a different rule for each transition, they could require different numbers of slope parameters.) The function `calcDPCTable` assumes the states are ordered from highest to lowest, and no transition is needed into the lowest state.

Note that if the value of `PnodeQ(node)` is not a matrix of all TRUE values, then the effective number of parents for each state transition could be different. In this case the value of `PnodeAlphas(node)` should be a list of vectors of different lengths (corresponding to the number of true entries in each row of `PnodeQ(node)`).

Finally, note that if we want the conditional probability table associated with node to be monotonic, then the `PnodeAlphas(node)` must be positive. To ensure this, `mapDPC` works with the log of the slopes, not the raw slopes. Similarly, `calcDPCTable` expects the log slope parameters as its `lnAlphas` argument, not the raw slopes. For that reason `PnodeLnAlphas(node)` is considered the primary function and a default method for `PnodeAlphas(node)` which simply takes exponents (or logs in the setter) is provided. Note that a sensible range for the slope parameters is usually between 1/2 and 2, with 1 (0 on the log scale) as a sensible first pass value.

Value

A list of numeric vectors giving the slopes for the combination function of each state transition. The vectors may be of different lengths depending on the value of `PnodeRules(node)` and `PnodeQ(node)`. If the slopes are the same for all transitions (as is required with the `gradedResponse` link function) then a single numeric vector instead of a list is returned.

Note that the setter form may destructively modify the `Pnode` object (this depends on the implementation).

Note

The functions `PnodeLnAlphas` and `PnodeLnAlphas<-` are abstract generic functions, and need specific implementations. The default methods for the functions `PnodeAlphas` and `PnodeAlphas<-`.

Depend on PnodeLnAlphas and PnodeLnAlphas<-, respectively. See the [PNetica-package](#) for an example.

The values of [PnodeLink](#), [PnodeRules](#), [PnodeQ](#), [PnodeParentTvals](#), [PnodeLnAlphas](#), and [PnodeBetas](#) all need to be consistent for this to work correctly, but no error checking is done on any of the setter methods.

Author(s)

Russell Almond

References

Almond, R. G. (2015) An IRT-based Parameterization for Conditional Probability Tables. Paper presented at the 2015 Bayesian Application Workshop at the Uncertainty in Artificial Intelligence Conference.

Almond, R.G., Mislevy, R.J., Steinberg, L.S., Williamson, D.M. and Yan, D. (2015) *Bayesian Networks in Educational Assessment*. Springer. Chapter 8.

See Also

[Pnode](#), [PnodeQ](#), [PnodeRules](#), [PnodeLink](#), [PnodeBetas](#), [BuildTable](#), [PnodeParentTvals](#), [maxCPTParam](#), [calcDPCTable](#), [mapDPC](#) [Compensatory](#), [OffsetConjunctive](#)

Examples

```
## Not run:
library(PNetica) ## Requires implementation
sess <- NeticaSession()
startSession(sess)

tNet <- CreateNetwork("TestNet",sess)

theta1 <- NewDiscreteNode(tNet,"theta1",
  c("VH","High","Mid","Low","VL"))
PnodeStateValues(theta1) <- effectiveThetas(PnodeNumStates(theta1))
PnodeProbs(theta1) <- rep(1/PnodeNumStates(theta1),PnodeNumStates(theta1))
theta2 <- NewDiscreteNode(tNet,"theta2",
  c("VH","High","Mid","Low","VL"))
PnodeStateValues(theta2) <- effectiveThetas(PnodeNumStates(theta2))
PnodeProbs(theta2) <- rep(1/PnodeNumStates(theta1),PnodeNumStates(theta2))

partial3 <- NewDiscreteNode(tNet,"partial3",
  c("FullCredit","PartialCredit","NoCredit"))
PnodeParents(partial3) <- list(theta1,theta2)

## Usual way to set rules is in constructor
partial3 <- Pnode(partial3,rules="Compensatory", link="partialCredit")
PnodePriorWeight(partial3) <- 10
BuildTable(partial3)
```



```

## slopes of 1 for both transitions
PnodeLnAlphas(partial3) <- c(0,0)
BuildTable(partial3)

## log slope 0 = slope 1
stopifnot(
  all(abs(PnodeAlphas(partial3) -1) <.0001)
)

## Make Skill 1 more important than Skill 2
PnodeLnAlphas(partial3) <- c(.25,-.25)
BuildTable(partial3)

## Make Skill 1 more important for the transition to ParitalCredit
## And Skill 2 more important for the transition to FullCredit
PnodeLnAlphas(partial3) <- list(FullCredit=c(-.25,.25),
                               PartialCredit=c(.25,-.25))
BuildTable(partial3)

## Set up so that first skill only needed for first transition, second
## skill for second transition; Adjust alphas to match
PnodeQ(partial3) <- matrix(c(TRUE,TRUE,
                             TRUE,FALSE), 2,2, byrow=TRUE)
PnodeLnAlphas(partial3) <- list(FullCredit=c(-.25,.25),
                               PartialCredit=0)
BuildTable(partial3)

## Using OffsetConjunctive rule requires single slope
PnodeRules(partial3) <- "OffsetConjunctive"
## Single slope parameter for each transition
PnodeLnAlphas(partial3) <- 0
PnodeQ(partial3) <- TRUE
PnodeBetas(partial3) <- c(0,1)
BuildTable(partial3)

## Separate slope parameter for each transition;
## Note this will only different from the previous transition when
## mapDPC is called. In the former case, it will learn a single slope
## parameter, in the latter, it will learn a different slope for each
## transition.
PnodeLnAlphas(partial3) <- list(0,0)
BuildTable(partial3)

DeleteNetwork(tNet)

## End(Not run)

```

Description

This is an experimental class which contains all of the meta-data of a Pnode, but isn't linked to backing Bayes net software. Its intended use is for assistance in building elicitation user interfaces.

Usage

```
PnodeMin(name, states = c("Yes", "No"), parents = list(), title = title(),
         description = character(), labels = character(), continuous = FALSE,
         tvals = effectiveThetas(length(states)), rules = "Compensatory",
         link = "partialCredit", alphas = 1, betas = 1, linkScale = NA,
         QQ = 1, PriorWeight = NA, CPT = NULL)
```

Arguments

| | |
|-------------|---------------------------------------|
| name | Name of the node. |
| states | Node state names. |
| parents | List of other nodes to use a parents. |
| title | Title of the node. |
| description | Node description. |
| labels | Node labels. |
| continuous | Logical flag for continuous nodes. |
| tvals | Values assigned to node levels. |
| rules | Combination rules. |
| link | Link function. |
| alphas | Slope parameters. |
| betas | Intercept parameters. |
| linkScale | Scale parameter |
| QQ | Inner Q-matrix |
| PriorWeight | Prior weights |
| CPT | Conditional probability table. |

Details

This class is basically just a list containing the necessary fields. See the various accessor functions for the interpretation of the fields for more complete definition.

Value

An [Pnode](#) object.

Note

This is an experimental class. It may be removed in future releases.

Author(s)

Russell Almond

See Also

[Pnode](#).

Examples

```
anode <- PnodeMin("anode")
```

PnodeName

Gets or sets name of a parameterized node.

Description

Gets or sets the name of the node. Rules for names are implementation dependent, but they should generally conform to variable naming conventions (begin with a letter and only contain alphanumeric characters, no embedded spaces.)

Usage

```
PnodeName(node)  
PnodeName(node)<- value
```

Arguments

| | |
|-------|--|
| node | A Pnode object that references the node. |
| value | An character vector of length 1 giving the new name. |

Details

The [PnodeTitle\(\)](#) function provides another way to name a node which is not subject to naming restrictions.

Value

The name of the node as a character vector of length 1.

The setter method returns the node argument.

True Names

True names are the names in the secret ancient language which hold power over an object (Le Guin, 1968).

Actually, this is a difficulty with implementations that place restrictions on the name of a network or node. In particular, Netica restricts node names to alphanumeric characters and limits the length. This may make it difficult to match nodes by name with other parts of the system which do not have this restriction. In this case the object may have both a *true name*, which is returned by PnodeName and an internal *use name* which is used by the implementation.

Author(s)

Russell Almond

References

Le Guin, U. K. (1968). *A Wizard of Earthsea*. Parnassus Press.

See Also

[Pnode](#), [PnetFindNode\(\)](#), [PnodeTitle\(\)](#),

Examples

```
## Not run:
library(PNetica) # Requires PNetica
sess <- NeticaSession()
startSession(sess)
net <- CreateNetwork("funNet", session=sess)

pnode <- NewDiscreteNode(net, "play")

stopifnot(PnodeName(pnode)=="play")
stopifnot(PnetFindNode(net, "play")==pnode)

PnodeName(pnode)<-"work"
stopifnot(PnetFindNode(net, "work")==pnode)

PnodeName(pnode) <- "Non-Netica Name"
stopifnot(PnetFindNode(net, "Non-Netica Name")==pnode)

DeleteNetwork(net)
stopSession(sess)

## End(Not run)
```

| | |
|--------------|--|
| PnodeParents | <i>Gets or sets the parents of a parameterized node.</i> |
|--------------|--|

Description

A parent of a child node is another node which has a link *from* the parent *to* the child. This function returns the list of parents of the node. It allows the list of parents for the node to be set, altering the topology of the network (see details).

Usage

```
PnodeParents(node)
PnodeParents(node) <- value
PnodeNumParents(node)
PnodeParentNames(node)
PnodeParentStates(node)
```

Arguments

| | |
|-------|---|
| node | A Pnode object whose parents are of interest. |
| value | A list of Pnode objects (or NULLs) which will become the new parents. Order of the nodes is important. See details. |

Details

At its most basic level, `PnodeParents()` reports on the topology of a network. Suppose we add the links `A1 --> B`, `A2 --> B`, and `A3 --> B` to the network. Then `PnodeParents(B)` should return `list(A1, A2, A3)`. The order of the inputs is important, because that this determines the order of the dimensions in the conditional probability table ([BuildTable\(\)](#)).

The parent list can be set. This can accomplish a number of different goals: it can replace a parent variable, it can add additional parents, it can remove extra parents, and it can reorder parents. Changing the parents alters the topology of the network. Note that the network must always be acyclic directed graphs. In particular, if changing the parent structure will result in a directed cycle, it will likely raise an error).

Value

`PnodeParents` list of [Pnode](#) objects representing the parents in the order that they will be used to establish dimensions for the conditional probability table.

The setting variant returns the modified *child* object.

The expression `PnodeNumParents(node)` returns an integer scalar giving the number of parents of node.

The expression `PnodeParentNames(node)` is a shortcut for `sapply(PnodeParents(node), PnodeName)` and `PnodeParentStates(node)` is a shortcut for `sapply(PnodeParents(node), PnodeName)`

Author(s)

Russell Almond

See Also[Pnode](#), [PnodeParentTvals](#)**Examples**

```
## Not run:
library(PNetica) ## Requires PNetica
sess <- NeticaSession()
startSession(sess)
abnet <- CreateNetwork("AB", session=sess)

anodes <- NewDiscreteNode(abnet, paste("A",1:3,sep=""))
B <- NewDiscreteNode(abnet,"B")

## Should be empty list
stopifnot(length(PnodeParents(B))==0)

PnodeParents(B) <- anodes
stopifnot(
  length(PnodeParents(B))==3,
  PnodeParents(B)[[2]] == anodes[[2]]
)

## Reorder nodes
PnodeParents(B) <- anodes[c(2:3,1)]
stopifnot(
  length(PnodeParents(B))==3,
  PnodeName(PnodeParents(B)[[2]])=="A3",
  all(nchar(names(PnodeParents(B)))==0)
)

PnodeParentNames(B)
PnodeParentStates(B)

## Remove a node.
PnodeParents(B) <- anodes[2:1]
stopifnot(
  length(PnodeParents(B))==2,
  PnodeName(PnodeParents(B)[[2]])=="A1",
  all(nchar(names(PnodeParents(B)))==0)
)

## Add a node
PnodeParents(B) <- anodes[3:1]
stopifnot(
  length(PnodeParents(B))==3,
  PnodeName(PnodeParents(B)[[3]])=="A1",
  all(nchar(names(PnodeParents(B)))==0)
)
```

```

)

## Remove all parents
PnodeParents(B) <- list()
stopifnot(
  length(PnodeParents(B))==0
)

DeleteNetwork(abnet)
stopSession(sess)

## End(Not run)

```

| | |
|------------------|---|
| PnodeParentTvals | <i>Fetches a list of numeric variables corresponding to parent states</i> |
|------------------|---|

Description

In constructing a conditional probability table using the discrete partial credit framework (see [calcDPCTable](#)), each state of each parent variable is mapped onto a real value called the effective theta. The function `PnodeParentTvals` returns a list of effective theta values for each parent variable.

Usage

```
PnodeParentTvals(node)
```

Arguments

`node` A [Pnode](#) object.

Details

Following the framework laid out in Almond (2015), the function `calcDPCTable` calculates a conditional probability table using the following steps:

1. Each set of parent variable states is converted to a set of continuous values called *effective thetas*. These are built into an array, `eTheta`, using `expand.grid` where each column represents a parent variable and each row a possible configuration of parents.
2. For each state of the node except the last, the set of effective thetas is filtered using the local Q-matrix, `PnodeQ(node) = Q`. Thus, the actual effect thetas for state `s` is `eTheta[,Q[s,]]`. The value of `PnodeRules(node)` determines which combination function is used.
3. For each state of the node except the last, the corresponding rule is applied to the effective thetas to get a single effective theta for each row of the table. This step is essentially calls the expression: `do.call(rules[[s]], list(eThetas[,Q[s,]], PnodeAlphas(node)[[s]], PnodeBetas(node)[[s]])`.
4. The resulting set of effective thetas are converted into conditional probabilities using the link function.

This function is responsible for the first step of this process. `PnodeParentTvals(node)` should return a list corresponding to the parents of node, and each element should be a numeric vector corresponding to the states of the appropriate parent variable. It is passed to `expand.grid` to produce the table of parent variables for each row of the CPT.

Note that in item response theory, ability (theta) values are assumed to have a unit normal distribution in the population of interest. Therefore, appropriate theta values are quantiles of the normal distribution. In particular, they should correspond to the marginal distribution of the parent variable. The function `effectiveThetas` produces equally spaced (wrt the normal measure) theta values (corresponding to a uniform distribution of the parent). Unequally spaced values can be produced by using appropriate values of the `qnorm` function, e.g. `qnorm(c(.875, .5, .125))` will produce effective thetas corresponding to a marginal distribution of (0.25, 0.5, 0.25) (note that each value is in the midpoint of the interval).

Continuous variables are handled

Value

`PnodeParentTvals(node)` should return a list corresponding to the parents of node, and each element should be a numeric vector corresponding to the states of the appropriate parent variable. If there are no parent variables, this will be a list of no elements.

Note

The function `PnodeParentTvals` is an abstract generic functions, and need specific implementations. See the `PNetica-package` for an example.

In particular, it is probably a mistake to using different effective theta values for different parent variables in different contexts, therefor, the cleanest implementation is to associate the effective thetas with the parent variables and simply have `PnodeParentTvals` fetch them on demand. Thus the implementation in `PNetica` is simply, `lapply(NodeParents(node), PnodeStateValues)`.

This is probably less than ideal, as the function `PnodeStateValues` calculates midpoints wrt Lebesgue measure and not normal measure (used by `effectiveTheta`).

Author(s)

Russell Almond

References

Almond, R. G. (2015) An IRT-based Parameterization for Conditional Probability Tables. Paper presented at the 2015 Bayesian Application Workshop at the Uncertainty in Artificial Intelligence Conference.

Almond, R.G., Mislevy, R.J., Steinberg, L.S., Williamson, D.M. and Yan, D. (2015) *Bayesian Networks in Educational Assessment*. Springer. Chapter 8.

See Also

`Pnode`, `PnodeStateValues`, `PnodeStateBounds`, `effectiveThetas`, `PnodeQ`, `PnodeRules`, `PnodeLink`, `PnodeLnAlphas`, `PnodeBetas`, `BuildTable`, `maxCPTParam` `calcDPCTable`, `mapDPC` `expand.grid`, `qnorm`

Examples

```

## Not run:
library(PNetica) ## Requires implementation
sess <- NeticaSession()
startSession(sess)

tNet <- CreateNetwork("TestNet",session=sess)

theta1 <- NewDiscreteNode(tNet,"theta1",
                          c("VH","High","Mid","Low","VL"))
## This next function sets the effective thetas for theta1
PnodeStateValues(theta1) <- effectiveThetas(PnodeNumStates(theta1))
PnodeProbs(theta1) <- rep(1/PnodeNumStates(theta1),PnodeNumStates(theta1))
theta2 <- NewDiscreteNode(tNet,"theta2",
                          c("High","Mid","Low"))
## This next function sets the effective thetas for theta2
PnodeStateValues(theta2) <- effectiveThetas(PnodeNumStates(theta2))
PnodeProbs(theta2) <- rep(1/PnodeNumStates(theta2),PnodeNumStates(theta2))

partial3 <- NewDiscreteNode(tNet,"partial3",
                            c("FullCredit","PartialCredit","NoCredit"))
PnodeParents(partial3) <- list(theta1,theta2)

## Usual way to set rules is in constructor
partial3 <- Pnode(partial3,rules="Compensatory", link="partialCredit")

PnodeParentTvals(partial3)
do.call("expand.grid",PnodeParentTvals(partial3))

DeleteNetwork(tNet)
stopSession(sess)

## End(Not run)

```

PnodePostWeight

Fetches the posterior weight associated with a node

Description

Before running [GEMfit](#), nodes are given a prior weight ([PnodePriorWeight](#)) indicating how much weight should be given to the prior distribution. After running the [calcExpTables](#) step, there will be a posterior weight giving the total weight of the prior plus data.

Usage

```
PnodePostWeight(node)
```

Arguments

node A `Pnode` object.

Details

Let s be a configuration of the parent variables, which corresponds to a row of the CPT of node (`PnodeProbs`(node)). Let $\mathbf{p}_s = (p_{s,1}, \dots, p_{s,K})$ be the corresponding row of the conditional probability table and let n_s be the corresponding prior weight (an element of `PnodePriorWeight`(node)). The corresponding row of the effective Dirichlet prior for that row is $\alpha_s = (\alpha_{s,1}, \dots, \alpha_{s,K})$, where $\alpha_{s,1} = p_{s,1}n_s$. Note that the matrix \mathbf{P} and the vector \mathbf{n} (stacking the conditional probability vectors and the prior weights) are sufficient statistics for the conditional probability distribution of node.

The function `calcExpTables` does the E-step (and some of the M-step) of the `GEMfit` algorithm. Its output is new values for the sufficient statistics, $\hat{\mathbf{P}}$ and $\hat{\mathbf{n}}$. At this point, the function `PnodeProbs` should return $\hat{\mathbf{P}}$ (although possibly as an array rather than a matrix) and `PnodePostWeight`(node) returns $\hat{\mathbf{n}}$.

Although the `PnodePostWeight`(node) is used in the next step, `maxAllTableParams`, it is not retained for the next round of the `GEMfit` algorithm, instead the `PnodePriorWeight`(node) is used for the next time `calcExpTables` is run.

Often, `PnodePriorWeight`(node) is set to a scalar, indicating that every row should be given the same weight, e.g., 10. In this case, `PnodePostWeight`(node) will usually be vector valued as different numbers of data points correspond to each row of the CPT. Furthermore, unless the parent variables are fully observed, the `PnodePostWeight`(node) are unlikely to be integer valued even if the prior weights are integers. However, the posterior weights should always be at least as large as the prior weights.

Value

A vector of numeric values corresponding to the rows of the CPT of node. An error may be produced if `calcExpTables` has not yet been run.

Author(s)

Russell Almond

References

Almond, R. G. (2015) An IRT-based parameterization for conditional probability tables. In Agosta, J. M. and Carvalho, R. N. (Eds.) *Proceedings of the Twelfth UAI Bayesian Modeling Application Workshop (BMAW 2015)*. *CEUR Workshop Proceedings*, **1565**, 14–23. http://ceur-ws.org/Vol-1565/bmaw2015_paper4.pdf.

See Also

`PnodePriorWeight`, `GEMfit`, `calcExpTables`, `maxAllTableParams`

Examples

```

## Not run:
library(PNetica) ## Need a specific implementation
sess <- NeticaSession()
startSession(sess)

irt10.base <- ReadNetworks(system.file("testnets", "IRT10.2PL.base.dne",
                                     package="PNetica"),
                          session=sess)

irt10.base <- as.Pnet(irt10.base) ## Flag as Pnet, fields already set.
irt10.theta <- PnetFindNode(irt10.base,"theta")
irt10.items <- PnetPnodes(irt10.base)
## Flag items as Pnodes
for (i in 1:length(irt10.items)) {
  irt10.items[[i]] <- as.Pnode(irt10.items[[i]])
  ## Add node to list of observed nodes
  PnodeLabels(irt10.items[[1]]) <-
    union(PnodeLabels(irt10.items[[1]]),"onodes")
}
PnetCompile(irt10.base) ## Netica requirement

casepath <- system.file("testdat", "IRT10.2PL.200.items.cas", package="PNetica")

item1 <- irt10.items[[1]]

priorcounts <- sweep(PnodeProbs(item1),1,GetPriorWeight(item1),"*")

calcExpTables(irt10.base,casepath)

postcounts <- sweep(PnodeProbs(item1),1,PnodePostWeight(item1),"*")

## Posterior row sums should always be larger.
stopifnot(
  all(apply(postcounts,1,sum) >= apply(priorcounts,1,sum))
)

DeleteNetwork(irt10.base)
stopSession(sess)

## End(Not run)

```

PnodeProbs

Gets or sets the conditional probability table associated with a Netica node.

Description

A complete Bayesian networks defines a conditional probability distribution for a node given its parents. If all the nodes are discrete, this comes in the form of a conditional probability table a

multidimensional array whose first several dimensions follow the parent variable and whose last dimension follows the child variable.

Usage

```
PnodeProbs(node)
PnodeProbs(node) <- value
```

Arguments

| | |
|-------|--|
| node | An active, discrete Pnode whose conditional probability table is to be accessed. |
| value | The new conditional probability table. See details for the expected dimensions. |

Details

Let `node` be the node of interest and `parent1`, `parent2`, ..., `parent p` , where p is the number of parents. Let `pdim = sapply(PnodeParents(node), PnodeNumStates)` be a vector with the number of states for each parent. A parent configuration is defined by assigning each of the parent values to one of its possible states. Each parent configuration defines a (conditional) probability distribution over the possible states of `node`.

The result of `PnodeProbs(node)` will be an array with dimensions `c(pdim, PnodeNumStates(node))`. The first p dimensions will be named according to the `PnodeParentNames(node)`. The last dimension will be named according to the node itself. The `dimnames` for the resulting array will correspond to the state names.

In the `CPTtools` package, this known as the [CPA](#) format, and tools exist to convert between this form an a two dimensional matrix, or [CPF](#) format.

The setter form expects an array of the same dimensions as an argument, although it does not need to have the `dimnames` set.

Value

A conditional probability array of class `c("CPA", "array")`. See [CPA](#).

Note

All of this assumes that these are discrete nodes, that is `isPnodeContinuous(node)` will return false for both `node` and all of the parents, or that the continuous nodes have been discretized through the use of [PnodeStateBounds](#).

Author(s)

Russell Almond

See Also

[Pnode](#), [BuildTable](#), [CPA](#), [CPF](#), [normalize\(\)](#), [PnodeParents\(\)](#), [PnodeStates\(\)](#)

Examples

```

## Not run: ## Requires implementation
sess <- NeticaSession()
startSession(sess)
abc <- CreateNetwork("ABC", session=sess)
A <- NewDiscreteNode(abc, "A", c("A1", "A2", "A3", "A4"))
B <- NewDiscreteNode(abc, "B", c("B1", "B2", "B3"))
C <- NewDiscreteNode(abc, "C", c("C1", "C2"))

PnodeParents(A) <- list()
PnodeParents(B) <- list(A)
PnodeParents(C) <- list(A,B)

PnodeProbs(A) <- c(.1, .2, .3, .4)
PnodeProbs(B) <- normalize(matrix(1:12, 4, 3))
PnodeProbs(C) <- normalize(array(1:24, c(A=4, B=3, C=2)))

Aprobs <- PnodeProbs(A)
Bprobs <- PnodeProbs(B)
Cprobs <- PnodeProbs(C)
stopifnot(
  CPTtools::is.CPA(Aprobs),
  CPTtools::is.CPA(Bprobs),
  CPTtools::is.CPA(Cprobs)
)

DeleteNetwork(abc)
stopSession(sess)

## End(Not run)

```

PnodeQ

Accesses a state-wise Q-matrix associated with a Pnode

Description

The function `calcDPCTable` has an argument `Q`, which allows the designer to specify that only certain parent variables are relevant for the state transition. The function `PnodeQ` accesses the local `Q`-matrix for the `Pnode` node.

Usage

```

PnodeQ(node)
PnodeQ(node) <- value

```

Arguments

`node` A `Pnode` whose local `Q`-matrix is of interest

value A logical matrix with number of rows equal to the number of outcome states of node minus one and number of columns equal to the number of parents of node. As a special case, if it has the value TRUE this is interpreted as a matrix of true values of the correct shape.

Details

Consider a `partialCredit` model, that is a `Pnode` for which the value of `PnodeLink` is "partialCredit". This model is represented as a series of transitions between the states $s + 1$ and s (in `calcDPCTable` states are ordered from high to low). The log odds of this transition is expressed with a function $Z_s(eTheta)$ where $Z_s()$ is the value of `PnodeRules`(node) and $eTheta$ is the result of the call `PnodeParentTvals`(node).

Let q_{sj} be true if the parent variable x_j is relevant for the transition between states $s + 1$ and s . Thus the function which is evaluated to calculate the transition probabilities is $Z_s(eTheta[, Q[s,]])$; that is, the parent variables for which q_{sj} is false are filtered out. The default value of TRUE means that no values are filtered.

Note that this currently makes sense only for the `partialCredit` link function. The `gradedResponse` link function assumes that the curves are parallel and therefore all of the curves must have the same set of variables (and values for `PnodeAlphas`).

Value

A logical matrix with number of rows equal to the number of outcome states of node minus one and number of columns equal to the number of parents of node, or the logical scalar TRUE if all parent variables are used for all transitions.

Note

The functions `PnodeQ` and `PnodeQ<-` are abstract generic functions, and need specific implementations. See the `PNetica-package` for an example.

The values of `PnodeLink`, `PnodeRules`, `PnodeQ`, `PnodeParentTvals`, `PnodeLnAlphas`, and `PnodeBetas` all need to be consistent for this to work correctly, but no error checking is done on any of the setter methods.

Note that the setter form may destructively modify the `Pnode` object (this depends on the implementation).

Author(s)

Russell Almond

References

Almond, R. G. (2013) Discretized Partial Credit Models for Bayesian Network Conditional Probability Tables. Draft manuscript available from author.

Almond, R. G. (2015) An IRT-based Parameterization for Conditional Probability Tables. Paper presented at the 2015 Bayesian Application Workshop at the Uncertainty in Artificial Intelligence Conference.

See Also

[Pnode](#), [PnodeRules](#), [PnodeLink](#), [PnodeLnAlphas](#), [PnodeAlphas](#), [BuildTable](#), [PnodeParentTvals](#), [maxCPTParam](#) [calcDPCTable](#), [mapDPC](#)

Examples

```
## Not run:
library(PNetica) ## Requires implementation
sess <- NeticaSession()
startSession(sess)

tNet <- CreateNetwork("TestNet",session=sess)

theta1 <- NewDiscreteNode(tNet,"theta1",
                          c("VH","High","Mid","Low","VL"))
PnodeStateValues(theta1) <- effectiveThetas(PnodeNumStates(theta1))
PnodeProbs(theta1) <- rep(1/PnodeNumStates(theta1),PnodeNumStates(theta1))
theta2 <- NewDiscreteNode(tNet,"theta2",
                          c("VH","High","Mid","Low","VL"))
PnodeStateValues(theta2) <- effectiveThetas(PnodeNumStates(theta2))
PnodeProbs(theta2) <- rep(1/PnodeNumStates(theta2),PnodeNumStates(theta2))

partial3 <- NewDiscreteNode(tNet,"partial3",
                            c("FullCredit","PartialCredit","NoCredit"))
PnodeParents(partial3) <- list(theta1,theta2)

partial3 <- Pnode(partial3,Q=TRUE, link="partialCredit")
PnodePriorWeight(partial3) <- 10
BuildTable(partial3)

## Default is all nodes relevant for all transitions
stopifnot(
  length(PnodeQ(partial3)) == 1,
  PnodeQ(partial3) == TRUE
)

## Set up so that first skill only needed for first transition, second
## skill for second transition; adjust alphas to match
PnodeQ(partial3) <- matrix(c(TRUE,TRUE,
                            TRUE,FALSE), 2,2, byrow=TRUE)
PnodeLnAlphas(partial3) <- list(FullCredit=c(-.25,.25),
                              PartialCredit=0)
BuildTable(partial3)

partial4 <- NewDiscreteNode(tNet,"partial4",
                            c("Score4","Score3","Score2","Score1"))
PnodeParents(partial4) <- list(theta1,theta2)
partial4 <- Pnode(partial4, link="partialCredit")
PnodePriorWeight(partial4) <- 10
```

```

## Skill 1 used for first transition, Skill 2 used for second
## transition, both skills used for the 3rd.

PnodeQ(partial4) <- matrix(c(TRUE,TRUE,
                             FALSE,TRUE,
                             TRUE,FALSE), 3,2, byrow=TRUE)
PnodeLnAlphas(partial4) <- list(Score4=c(.25,.25),
                               Score3=0,
                               Score2=-.25)

BuildTable(partial4)

DeleteNetwork(tNet)
stopSession(sess)

## End(Not run)

```

PnodeRules

Accesses the combination rules for a Pnode

Description

In constructing a conditional probability table using the discrete partial credit framework (see [calcDPCTable](#)), the effective thetas for each parent variable are combined into a single effect theta using a combination rule. The function `PnodeRules` accesses the combination function associated with a [Pnode](#).

Usage

```

PnodeRules(node)
PnodeRules(node) <- value

```

Arguments

| | |
|-------|--|
| node | A Pnode object. |
| value | The name of a combination function, the combination function or a list of names or combination functions (see details). If a list, it should have length one less than the number of states in node. |

Details

Following the framework laid out in Almond (2015), the function `calcDPCTable` calculates a conditional probability table using the following steps:

1. Each set of parent variable states is converted to a set of continuous values called *effective thetas* (see [PnodeParentTvals](#)). These are built into an array, `eTheta`, using `expand.grid` where each column represents a parent variable and each row a possible configuration of parents.
2. For each state of the node except the last, the set of effective thetas is filtered using the local Q-matrix, `PnodeQ(node) = Q`. Thus, the actual effect thetas for state `s` is `eTheta[,Q[s,]]`.

3. For each state of the node except the last, the corresponding rule is applied to the effective thetas to get a single effective theta for each row of the table. This step is essentially calls the expression: `do.call(rules[[s]], list(eThetas[,Q[s,]], PnodeAlphas(node)[[s]], PnodeBetas(node)[[s]])`.
4. The resulting set of effective thetas are converted into conditional probabilities using the link function `PnodeLink(node)`.

The function `PnodeRules` accesses the function used in step 3. It should be the name of a function or a function with the general signature of a combination function described in [Compensatory](#). Predefined choices include [Compensatory](#), [Conjunctive](#), [Disjunctive](#), [OffsetConjunctive](#), and [OffsetDisjunctive](#). Note that the first three choices expect that there will be multiple alphas, one for each parent, and the latter two expect that there will be multiple betas, one for each beta. The value of `PnodeAlphas` and `PnodeBetas` should be set to match.

If the value of `PnodeLink` is `partialCredit`, then the link function can be different for state of the node. (If it is `gradedResponse` then the curves need to be parallel and it should be the same.) If the value of `PnodeRules(node)` is a list (note: list, not character vector), then a different rule is used for each state transition. The function `calcDPCTable` assumes the states are ordered from highest to lowest, and no transition is needed into the lowest state.

Value

A character scalar giving the name of a combination function or a combination function object, or a list of the same. If a list, its length is one less than the number of states of node.

Note that the setter form may destructively modify the `Pnode` object (this depends on the implementation).

Note

The functions `PnodeRules` and `PnodeRules<-` are abstract generic functions, and need specific implementations. See the [PNetica-package](#) for an example.

The values of `PnodeLink`, `PnodeRules`, `PnodeQ`, `PnodeParentTvals`, `PnodeLnAlphas`, and `PnodeBetas` all need to be consistent for this to work correctly, but no error checking is done on any of the setter methods.

Author(s)

Russell Almond

References

Almond, R. G. (2015) An IRT-based Parameterization for Conditional Probability Tables. Paper presented at the 2015 Bayesian Application Workshop at the Uncertainty in Artificial Intelligence Conference.

Almond, R.G., Mislevy, R.J., Steinberg, L.S., Williamson, D.M. and Yan, D. (2015) *Bayesian Networks in Educational Assessment*. Springer. Chapter 8.

See Also

[Pnode](#), [PnodeQ](#), [PnodeLink](#), [PnodeLnAlphas](#), [PnodeBetas](#), [BuildTable](#), [PnodeParentTvals](#), [maxCPTParam](#), [calcDPCTable](#), [mapDPC](#), [Compensatory](#), [OffsetConjunctive](#)

Examples

```
## Not run:
library(PNetica) ## Requires implementation
sess <- NeticaSession()
startSession(sess)

tNet <- CreateNetwork("TestNet", session=sess)

theta1 <- NewDiscreteNode(tNet, "theta1",
                          c("VH", "High", "Mid", "Low", "VL"))
PnodeStateValues(theta1) <- effectiveThetas(PnodeNumStates(theta1))
PnodeProbs(theta1) <- rep(1/PnodeNumStates(theta1), PnodeNumStates(theta1))
theta2 <- NewDiscreteNode(tNet, "theta2",
                          c("VH", "High", "Mid", "Low", "VL"))
PnodeStateValues(theta2) <- effectiveThetas(PnodeNumStates(theta2))
PnodeProbs(theta2) <- rep(1/PnodeNumStates(theta2), PnodeNumStates(theta2))

partial3 <- NewDiscreteNode(tNet, "partial3",
                            c("FullCredit", "PartialCredit", "NoCredit"))
PnodeParents(partial3) <- list(theta1, theta2)

## Usual way to set rules is in constructor
partial3 <- Pnode(partial3, rules="Compensatory", link="partialCredit")
PnodePriorWeight(partial3) <- 10
BuildTable(partial3)

stopifnot(
  PnodeRules(partial3) == "Compensatory"
)

## Use different rules for different levels
## Compensatory for 2nd transition, conjunctive for 1st
## Note: Position is important, names are just for documentation.
PnodeRules(partial3) <- list(FullCredit="Compensatory",
                            PartialCredit="Conjunctive")

BuildTable(partial3)

DeleteNetwork(tNet)

## End(Not run)
```

Description

This function returns a list associated with a [Pnode](#). The function `PnodeStates` returns or manipulates them. Depending on the implementation, states may have restrictions on the names to it is best to stick with variable naming conventions (start with a letter, no embedded spaces or punctuation).

Usage

```
PnodeStates(node)
PnodeStates(node) <- value
PnodeNumStates(node)
```

Arguments

| | |
|--------------------|---|
| <code>node</code> | A Pnode object whose states are to be accessed. |
| <code>value</code> | A character vector giving the names of the new states. the names of the states. State names may be restricted by the implementing package and should probably stick to variable naming conventions. |

Details

The states are important when building conditional probability tables (CPTs). In particular, the state names are used to label the columns of the CPT. Thus, state names can be used to address arrays in the same way that `dimnames` can. In particular, the state names can be used to index the vectors returned by `PnodeStates()`, `PnodeStateTitles()`, `PnodeStateTitles()`, and `PnodeStateValues()`.

Value

The function `PnodeStates()` returns a character vector whose values and names are both set to the state names. The setter version of this function invisibly returns the `node` object.

The expression `PnodeNumStates(node)` returns an integer scalar giving the number of states of `node`.

Note

Changing the number of states once a conditional probability table is set will change the dimensions of the table, and hence will likely remove it.

Author(s)

Russell Almond

See Also

[Pnode](#), [PnodeName\(\)](#), [PnodeStateTitles\(\)](#), [PnodeStateValues\(\)](#), [PnodeStateDescriptions\(\)](#),

Examples

```

## Not run:
library(PNetica)##Requires PNetica
sess <- NeticaSession()
startSession(sess)
anet <- CreateNetwork("Annette", session=sess)

## Discrete Nodes
node12 <- NewDiscreteNode(anet,"TwoLevelNode")
stopifnot(
  length(PnodeStates(node12))==2,
  PnodeStates(node12)==c("Yes","No")
)

PnodeStates(node12) <- c("True","False")
stopifnot(
  PnodeNumStates(node12) == 2L,
  PnodeStates(node12)==c("True","False")
)

node13 <- NewDiscreteNode(anet,"ThreeLevelNode",c("High","Med","Low"))
stopifnot(
  PnodeNumStates(node13) == 3L,
  PnodeStates(node13)==c("High","Med","Low"),
  PnodeStates(node13)[2]=="Med"
)

PnodeStates(node13)[2] <- "Median"
stopifnot(
  PnodeStates(node13)[2]=="Median"
)

PnodeStates(node13)["Median"] <- "Medium"
stopifnot(
  PnodeStates(node13)[2]=="Medium"
)

DeleteNetwork(anet)
stopSession(sess)

## End(Not run)

```

Description

Each state of a [Pnode](#) has a short name (which could be restricted by the implementation) and a longer title (which generally can contain emedded spaces and other details to make it more readable). Each state also can have a description associated with it. These functions get or set the state titles or descriptions.

Usage

```
PnodeStateTitles(node)
PnodeStateTitles(node) <- value
PnodeStateDescriptions(node)
PnodeStateDescriptions(node) <- value
```

Arguments

| | |
|-------|---|
| node | A Pnode object whose state titles or descriptions will be accessed. |
| value | A character vector of the same length as the number of states <code>length(PnodeStates(node))</code> which provides the new state titles or descriptions. |

Details

The titles are meant to be a more human readable version of the state names and are not subject the variable naming restrictions. The descriptions are meant to be a longer free form notes.

Both titles and descriptions are returned as a named character vector with names corresponding to the state names. Therefore one can change a single state title or description by accessing it either using the state number or the state name.

Value

Both `PnodeStateTitles()` and `PnodeStateDescriptions()` return a character vector of length `length(PnodeStates(node))` giving the titles or descriptions respectively. The names of this vector are `PnodeStates(node)`.

The setter methods return the modified [Pnode](#) object invisibly.

Author(s)

Russell Almond

See Also

[Pnode](#), [PnodeStates\(\)](#), [PnodeStateValues\(\)](#)

Examples

```
## Not run:
library(PNetica)##Requires PNetica
sess <- NeticaSession()
startSession(sess)
cnet <- CreateNetwork("CreativeNet", session=sess)
```

```

orig <- NewDiscreteNode(cnet,"Originality", c("H","M","L"))
PnodeStateTitles(orig) <- c("High","Medium","Low")
PnodeStateDescriptions(orig)[1] <- "Produces solutions unlike those typically seen."

stopifnot(
  PnodeStateTitles(orig) == c("High","Medium","Low"),
  grep("solutions unlike", PnodeStateDescriptions(orig))==1,
  PnodeStateDescriptions(orig)[3]=="")
)

sol <- NewDiscreteNode(cnet,"Solution",
  c("Typical","Unusual","VeryUnusual"))
stopifnot(
  all(PnodeStateTitles(sol) == ""),
  all(PnodeStateDescriptions(sol) == ""))
)

PnodeStateTitles(sol)["VeryUnusual"] <- "Very Unusual"
PnodeStateDescriptions(sol) <- paste("Distance from typical solution",
  c("<1", "1--2", ">2"))
stopifnot(
  PnodeStateTitles(sol)[3]=="Very Unusual",
  PnodeStateDescriptions(sol)[1] == "Distance from typical solution <1"
)

DeleteNetwork(cnet)
stopSession(sess)

## End(Not run)

```

| | |
|------------------|---|
| PnodeStateValues | <i>Accesses the numeric values associated with the state of a parameterized node.</i> |
|------------------|---|

Description

The values are a numeric value (on a standard normal scale) associated with the levels of a discrete [Pnode](#). This function fetches or retrieves the numeric values for the states of *node*.

Note that the default method for the function [PnodeParentTvals](#) uses the values of [PnodeStateValues](#) on the parent nodes.

Usage

```

PnodeStateValues(node)
PnodeStateValues(node) <- value

```

Arguments

| | |
|-------|--|
| node | A Pnode whose levels are to be accessed. |
| value | A numeric vector of values which should have length <code>length(PnodeStates(node))</code> . |

Details

This function behaves differently for discrete and continuous nodes (see [isPnodeContinuous](#)). For discrete nodes, the states are numeric values associated with the states. These are used in a number of ways, most importantly, as [PnodeParentTvals](#). Note that the first time the `PnodeStateValues()` are set, the entire vector must be set. After that point individual values may be changed.

For continuous nodes, the state values are set by setting the [PnodeStateBounds](#) for the node. The value is the midpoint of each interval. (Note this produces an infinite state value if one of the state bounds is infinite).

Value

A numeric vector of length `length(PnodeStates())`, with names equal to the state names. If levels have not been set, NAs will be returned.

Author(s)

Russell Almond

See Also

[PnodeStateBounds](#), [Pnode](#), [PnodeStates\(\)](#), [PnodeName\(\)](#), [PnodeStateTitles\(\)](#), [PnodeParentTvals\(\)](#)

Examples

```
## Not run:
library(PNetica)##Requires PNetica
sess <- NeticaSession()
startSession(sess)
lnet <- CreateNetwork("LeveledNet", session=sess)

vnode <- NewDiscreteNode(lnet,"volt_switch",c("Off","Reverse","Forwards"))
stopifnot(
  length(PnodeStateValues(vnode))==3,
  names(PnodeStateValues(vnode)) == PnodeStates(vnode),
  all(is.na(PnodeStateValues(vnode)))
)

## Don't run this until the levels for vnode have been set,
## it will generate an error.
try(PnodeStateValues(vnode)[2] <- 0)

PnodeStateValues(vnode) <- 1:3
stopifnot(
  length(PnodeStateValues(vnode))==3,
  names(PnodeStateValues(vnode)) == PnodeStates(vnode),
  PnodeStateValues(vnode)[2]==2
)

PnodeStateValues(vnode)["Reverse"] <- -2

## Continuous nodes get the state values from the bounds.
```

```

theta0 <- NewContinuousNode(lnet, "theta0")
stopifnot(length(PnodeStateValues(theta0))==0L)
norm5 <-
  matrix(c(qnorm(c(.001, .2, .4, .6, .8)),
           qnorm(c(.2, .4, .6, .8, .999))), 5, 2,
         dimnames=list(c("VH", "High", "Mid", "Low", "VL"),
                       c("LowerBound", "UpperBound")))
PnodeStateBounds(theta0) <- norm5
PnodeStateValues(theta0) ## Note these are medians not mean wrt normal!
PnodeStateBounds(theta0)[1,1] <- -Inf
PnodeStateValues(theta0) ## Infinite value!

DeleteNetwork(lnet)
stopSession(sess)

## End(Not run)

```

PnodeStats

Pnode Marginal Statistics

Description

These functions compute statistics of the marginal distribution of the corresponding node. These are designed to be used with [Statistic](#) objects.

Usage

```

PnodeMargin(net, node)
PnodeEAP(net, node)
PnodeSD(net, node)
PnodeMedian(net, node)
PnodeMode(net, node)

```

Arguments

| | |
|------|--|
| net | A Pnet object representing the network. |
| node | A Pnode object describing the node whose statistics are desired. |

Details

These are the functions that implement the statistics. These are typically called by [calcStat](#) which finds the nodes corresponding to the named nodes in the statistics. Both the net and node are passes as arguments as this may be needed in some implementations.

Value

PnodeMargin returns a vector corresponding to the states of node giving the marginal probabilities of the states.

PnodeEAP returns a numeric scalar giving the expected a posteriori value (mean) of the [PnodeStateValues](#) of the node. PnodeSD gives the standard deviation.

PnodeMedian assumes the states are ordered, and returns the state at the 50th percentile. This is a factor (character) value.

PnodeMode returns the most likely state as a factor (character) value.

Author(s)

Russell Almond

References

Almond, R.G., Mislevy, R.J. Steinberg, L.S., Yan, D. and Willamson, D. M. (2015). *Bayesian Networks in Educational Assessment*. Springer. Chapter 13.

See Also

Statistics Class: [Statistic](#)

Constructor function: [Statistic](#)

[calcStat](#)

These statistics will likely produce errors unless [PnetCompile](#) has been run first.

Examples

```
## Not run:

library(PNetica) ## Need a specific implementation
sess <- NeticaSession()
startSession(sess)

irt10.base <- ReadNetworks(system.file("testnets", "IRT10.2PL.base.dne",
                                     package="PNetica"),
                          session=sess)
irt10.base <- as.Pnet(irt10.base) ## Flag as Pnet, fields already set.
irt10.theta <- PnetFindNode(irt10.base,"theta")
irt10.items <- PnetPnodes(irt10.base)
## Flag items as Pnodes
for (i in 1:length(irt10.items)) {
  irt10.items[[i]] <- as.Pnode(irt10.items[[i]])
}
## Make some statistics
marginTheta <- Statistic("PnodeMargin","theta","Pr(theta)")
meanTheta <- Statistic("PnodeEAP","theta","EAP(theta)")
sdTheta <- Statistic("PnodeSD","theta","SD(theta)")
medianTheta <- Statistic("PnodeMedian","theta","Median(theta)")
```

```

modeTheta <- Statistic("PnodeMedian", "theta", "Mode(theta)")

BuildAllTables(irt10.base)
PnetCompile(irt10.base) ## Netica requirement

calcStat(marginTheta, irt10.base)
calcStat(meanTheta, irt10.base)
calcStat(sdTheta, irt10.base)
calcStat(medianTheta, irt10.base)
calcStat(modeTheta, irt10.base)

DeleteNetwork(irt10.base)
stopSession(sess)

## End(Not run)

```

| | |
|------------|---|
| PnodeTitle | <i>Gets the title or Description associated with a parameterized node node.</i> |
|------------|---|

Description

The title is a longer name for a node which is not subject to the naming restrictions. The description is a free form text associated with a node.

Usage

```

PnodeTitle(node)
PnodeTitle(node) <- value
PnodeDescription(node)
PnodeDescription(node) <- value

```

Arguments

| | |
|-------|---|
| node | A Pnode object. |
| value | A character object giving the new title or description. |

Details

The title is meant to be a human readable alternative to the name, which is not limited to the variable name restrictions (i.e., it can contain spaces and punctuation). The title may also affect how the node is displayed.

The description is any text the user chooses to attach to the node. If *value* has length greater than 1, the vector is collapsed into a long string with newlines separating the components.

Value

A character vector of length 1 providing the title or description.

Author(s)

Russell Almond

See Also[Pnode](#), [PnodeName\(\)](#)**Examples**

```
## Not run:
library(PNetica) ## Requires PNetica
sess <- NeticaSession()
startSession(sess)
net2 <- CreateNetwork("secondNet", session=sess)

firstNode <- NewDiscreteNode(net2,"firstNode")

PnodeTitle(firstNode) <- "My First Bayesian Network Node"
stopifnot(PnodeTitle(firstNode)=="My First Bayesian Network Node")

now <- date()
PnodeDescription(firstNode)<-c("Node created on",now)
stopifnot(PnodeDescription(firstNode) ==
  paste(c("Node created on",now),collapse="\n"))

## Print here escapes the newline, so is harder to read
cat(PnodeDescription(firstNode),"\n")

DeleteNetwork(net2)
stopSession(sess)

## End(Not run)
```

PnodeWarehouse-class *Class* "PnodeWarehouse"

Description

A [Warehouse](#) objects which holds and builds [Pnode](#) objects. In particular, its [WarehouseManifest](#) contains a node manifest (see [BuildNodeManifest](#)) which contains information about how to build the nodes if they are not present. Note that the key of the node manifest is the name of both the network and the node.

Details

The PnetWarehouse either supplies prebuilt nodes or builds them from the instructions found in the manifest. Nodes exist inside networks, so the key for a node is a pair (Model,NodeName). Thus, two nodes in different networks can have identical names.

The function WarehouseSupply will attempt to:

1. Find an existing node with name `NodeName` in a network with name `Model`.
2. Build a new node in the named network using the metadata in the manifest.

The manifest is an object of type `data.frame` where the columns have the values show below. The key is the combination of the “Model” and “NodeName” columns. There should be one row with this combination of variables for each state of the variable. In particular, the number of rows should equal the value of the `Nstates` column in the first row with that model–variable combination. The “StateName” column should be unique for each row.

The arguments to `WarehouseData` should be a character vector of length two, (`Model`, `NodeName`). It will return a `data.frame` with one row for each state of the variable.

Node-level Key Fields :

Model A character value giving the name of the Bayesian network to which this node belongs. Corresponds to the value of `PnodeNet`.

NodeName A character value giving the name of the node. All rows with the same value in the model and node name columns are assumed to reference the same node. Corresponds to the value of `PnodeName`.

Node-level Fields :

ModelHub If this is a spoke model (meant to be attached to a hub) then this is the name of the hub model (i.e., the name of the proficiency model corresponding to an evidence model). Corresponds to the value of `PnetHub(PnodeNet(node))`.

NodeTitle A character value containing a slightly longer description of the node, unlike the name this is not generally restricted to variable name formats. Corresponds to the value of `PnodeTitle`.

NodeDescription A character value describing the node, meant for human consumption (documentation). Corresponds to the value of `PnodeDescription`.

NodeLabels A comma separated list of identifiers of sets which this node belongs to. Used to identify special subsets of nodes (e.g., high-level nodes or observable nodes). Corresponds to the value of `PnodeLabels`.

State-level Key Fields :

Continuous A logical value. If true, the variable will be continuous, with states corresponding to ranges of values. If false, the variable will be discrete, with named states.

Nstates The number of states. This should be an integer greater than or equal to 2. Corresponds to the value of `PnodeNumStates`.

StateName The name of the state. This should be a string value and it should be different for every row within the subset of rows corresponding to a single node. Corresponds to the value of `PnodeStates`.

State-level Fields :

StateTitle A longer name not subject to variable naming restrictions. Corresponds to the value of `PnodeStateTitles`.

StateDescription A human readable description of the state (documentation). Corresponds to the value of `PnodeStateDescriptions`.

StateValue A real numeric value assigned to this state. `PnodeStateValues`. Note that this has different meaning for discrete and continuous variables. For discrete variables, this associates a numeric value with each level, which is used in calculating the `PnodeEAP` and `PnodeSD` functions. In the continuous case, this value is ignored and the midpoint between the “LowerBounds” and “UpperBounds” are used instead.

LowerBound This servers as the lower bound for each partition of the continuous variagle. -Inf is a legal value for the first or last row.

UpperBound This is only used for continuous variables, and the value only is needed for one of the states. This servers as the upper bound of range each state. Note the upper bound needs to match the lower bounds of the next state. Inf is a legal value for the first or last row.

Objects from the Class

A virtual Class: No objects may be created from it.

Classes can register as belonging to this abstract class. The trick for doing this is: `setIs("NodehouseClass", "PnodeWareh`

Currently [NNWarehouse](#) is an example of an object of this class.

Methods

Note that for all of these methods, the name should be a vector of two elements, the network name and the node name. Thus each network defines its own namespace for variables.

WarehouseSupply signature(`warehouse = "PnodeWarehouse"`, `name = "character"`). This finds a node with the appropriate name in the specified network. If one does not exist, it is created using the metadata in the manifest.

WarehouseFetch signature(`warehouse = "PnodeWarehouse"`, `name = "character"`). This fetches the node with the given name in the named network, or returns NULL if it has not been built.

WarehouseMake signature(`warehouse = "PnodeWarehouse"`, `name = "character"`). This creates the node using the meta-data in the Manifest.

WarehouseFree signature(`warehouse = "PnodeWarehouse"`, `name = "character"`). This removes the node from the warehouse inventory.

ClearWarehouse signature(`warehouse = "PnodeWarehouse"`). This removes all nodes from the warehouse inventory.

is.PnodeWarehouse signature(`obj = "PnodeWarehouse"`). This returns TRUE.

WarehouseManifest signature(`warehouse = "PnodeWarehouse"`). This returns the data frame with instructions on how to build nodes. (see Details)

WarehouseManifest<- signature(`warehouse = "PnodeWarehouse"`, `value="data.frame"`). This sets the data frame with instructions on how to build nodes.(see Details)

WarehouseData signature(`warehouse = "PnodeWarehouse"`, `name="character"`). This returns the portion of the data frame with instructions on how to build a particular node. This is generally one row for each state of the node. (see Details)

Note

The test for matching upper and lower bounds is perhaps too strict. In particular, if the upper and lower bounds mismatch by the least significant digit (e.g., a rounding difference) they will not match. This is a frequent cause of errors.

Author(s)

Russell Almond

See Also

[Warehouse](#), [WarehouseManifest](#), [BuildNodeManifest](#)

Implementation in the PNetica package: [NNWarehouse](#), [MakePnode.NeticaNode](#)

Examples

```
## This expression provides an example Node manifest
nodeman1 <- read.csv(system.file("auxdata", "Mini-PP-Nodes.csv",
                               package="Peanut"),
                    stringsAsFactors=FALSE)

## Not run:
library(PNetica) ## Requires PNetica
sess <- NeticaSession()
startSession(sess)

## This expression provides an example Node manifest
netman1 <- read.csv(system.file("auxdata", "Mini-PP-Nets.csv",
                               package="PNetica"),
                   row.names=1, stringsAsFactors=FALSE)

## Network and node warehouse, to create networks and nodes on demand.
Nethouse <- BNWarehouse(manifest=netman1, session=sess, key="Name")

Nodehouse <- NNWarehouse(manifest=nodeman1,
                         key=c("Model", "NodeName"),
                         session=sess)

CM <- WarehouseSupply(Nethouse, "miniPP_CM")
WarehouseSupply(Nethouse, "PPdurAttEM")

WarehouseData(Nodehouse, c("miniPP_CM", "Physics"))
WarehouseSupply(Nodehouse, c("miniPP_CM", "Physics"))

WarehouseData(Nodehouse, c("PPdurAttEM", "Attempts"))
WarehouseSupply(Nodehouse, c("PPdurAttEM", "Attempts"))

WarehouseData(Nodehouse, c("PPdurAttEM", "Duration"))
WarehouseSupply(Nodehouse, c("PPdurAttEM", "Duration"))

## End(Not run)
```

Description

In augmented Q -matrix, there is a set of rows for each `Pnode` which describes the conditional probability table for that node in terms of the model parameters (see `BuildTable`). As the `Pnodes` could potentially come from multiple nets, the key for the table is (“Model”, “Node”). As there are multiple rows per node, “State” is the third part of the key.

The function `Qmat2Pnet` adjusts the conditional probability tables of a node to conform to the supplied Q -matrix.

Usage

```
Qmat2Pnet(Qmat, nethouse, nodehouse, defaultRule = "Compensatory",
          defaultLink = "partialCredit", defaultAlpha = 1,
          defaultBeta = NULL, defaultLinkScale = NULL,
          defaultPriorWeight=10, debug = FALSE, override = FALSE)
```

Arguments

| | |
|---------------------------------|---|
| <code>Qmat</code> | A <code>data.frame</code> containing an augmented Q -matrix (See below). |
| <code>nethouse</code> | A <code>Warehouse</code> containing instructions for building the <code>Pnet</code> objects named in the “Model” column of <code>Qmat</code> . |
| <code>nodehouse</code> | A <code>Warehouse</code> containing instructions for building the <code>Pnode</code> objects named in the (“Model”, “Name”) columns of <code>Qmat</code> . |
| <code>defaultRule</code> | This should be a character scalar giving the name of a CPTtools combination rule (see <code>Compensatory</code>). |
| <code>defaultLink</code> | This should be a character scalar giving the name of a CPTtools link function (see <code>partialCredit</code>). |
| <code>defaultAlpha</code> | A numeric scalar giving the default value for slope parameters. |
| <code>defaultBeta</code> | A numeric scalar giving the default value for difficulty (negative intercept) parameters. |
| <code>defaultLinkScale</code> | A positive number which gives the default value for the link scale parameter. |
| <code>defaultPriorWeight</code> | A positive number which gives the default value for the node prior weight hyperparameter. |
| <code>debug</code> | A logical value. If true, extra information will be printed during process of building the <code>Pnet</code> . |
| <code>override</code> | A logical value. If false, differences between any existing structure in the graph and the <code>Qmat</code> will raise an error. If true, the graph will be modified to conform to the matrix. |

Details

A Q -matrix is a 0-1 matrix which describes which proficiency (latent) variables are connected to which observable outcome variables; $q_{jk} = 1$ if and only if proficiency variable k is a parent of observable variable j . Almond (2010) suggested that augmenting the Q -matrix with additional

columns representing the combination rules (`PnodeRules`), link function (`PnodeLink`), link scale parameter (if needed, `PnodeLinkScale`) and difficulty parameters (`PnodeBetas`). The discrimination parameters (`PnodeAlphas`) could be overloaded with the Q -matrix, with non-zero parameters in places where there were 1's in the Q -matrix.

This arrangement worked fine with combination rules (e.g., `Compensatory`) which contained multiple alpha (discrimination) parameters, one for each parent variable, and a single beta (difficulty). The introduction of a new type of offset rule (e.g., `OffsetDisjunctive`) which uses a multiple difficulty parameters, one for each parent variable, and a single alpha. Almond (2016) suggested a new augmentation which has three matrixes in a single table (a Qmat): the Q -matrix, which contains structural information; the A -matrix, which contains discrimination parameters; and the B -matrix, which contains the difficulty parameters. The names for the columns for these matrixes contain the names of the proficiency variables, prepended with "A." or "B." in the case of the A -matrix and B -matrix. There are two additional columns marked "A" and "B" which are used for the discrimination and difficulty parameter in the multiple-beta and multiple-alpha cases. There is some redundancy between the Q , A and B matrixes, but this provides an opportunity for checking the validity of the input.

The introduction of the partial credit link function (`partialCredit`) added a further complication. With the partial credit model, there could be a separate set of discrimination or difficulty parameters for each transition for a polytomous item. Even the `gradedResponse` link function requires a separate difficulty parameter for each level of the variable save the first. The rows of the Qmat data structure are hence augmented to include one row for every state but the lowest-level state. There should be of fewer rows of associated with the node than the value in the "Nstates" column, and the names of the states (values in the "State" column) should correspond to every state of the target variable except the first. It is an error if the number of states does not match the existing node, or if the state names do not match what is already used for the node or is in the manifest for the node `Warehouse`.

Note that two nodes in different networks may share the same name, and two states in two different nodes may have the same name as well. Thus, the formal key for the Qmat data frame is ("Model", "Node", "State"), however, the rows which share the values for ("Model", "Node") form a subtable for that particular node. In particular, the rows of the Q -matrix subtable for that node form the *inner Q-matrix* for that node. The inner Q -matrix shows which variables are relevant for each state transition in a partial credit model. The column-wise maximum of the inner Q -matrix forms the row of the outer Q -matrix for that node. This shows which proficiency nodes are the parent of the observable node. This corresponds to `PnodeQ(node)`.

The function `Qmat2Pnet` creates and sets the parameters of the observable `Pnodes` referenced in the `Qmat` argument. As it needs to reference, and possibly create, a number of `Pnets` and `Pnodes`, it requires both a network and a node `Warehouse`. If the `override` parameter is true, the networks will be modified so that each node has the correct parents, otherwise `Qmat2Pnet` will signal an error if the existing network structure is inconsistent with the Q -matrix.

As there is only one link function for each *node*, the values of `PnodeLink(node)` and `PnodeLinkScale(node)` are set based on the values in the "Link" and "LinkScale" columns and the first row corresponding to *node*. Note that the choice of link functions determines what is sensible for the other values but this is not checked by the code.

The value of `PnodeRules(node)` can either be a single value or a list of rule names. The first value in the sub-Qmat must a character value, but if the other values are missing then a single value is used. If not, all of the entries should be non-missing. If this is a single value, then effectively the same combination rule is used for each transition.

The interpretation of the A -matrix and the B -matrix depends on the value in the “Rules” column. There are two types of rules, multiple-A rules and multiple-B rules (offset rules). The CPTtools function `isOffsetRule` checks to see what kind of a rule it is. The multiple-A rules, of which `Compensatory` is the canonical example, have one discrimination (or slope) parameter for every parent variable (values of 1 in the Q -matrix) and have a single difficulty (negative intercept) parameter which is in the “B” column of the Qmat. The multiple-B or offset rules, of which `OffsetConjunctive` is the canonical example, have a difficulty (negative intercept) parameter for each parent variable and a single discrimination (slope) parameter which is in the “A” column. The function `Qmat2Pnet` uses the value of `isOffsetRule` to determine whether to use the multiple-B (true) or multiple-A (false) paradigm.

A simple example is a binary observable variable which uses the `Compensatory` rule. This is essentially a regression model (logistic regression with `partialCredit` or `gradedResponse` link functions, linear regression with `normalLink` link function) on the parent variables. The linear predictor is:

$$\frac{1}{\sqrt{K}}(a_1\theta_1 + \dots + a_K\theta_K) - b.$$

The values $\theta_1, \dots, \theta_K$ are effective thetas, real values corresponding to the states of the parent variables. The value a_i is stored in the column “A.name i ” where $name_i$ is the name of the i th proficiency variable; the value of `PnodeAlphas(node)` is the vector a_1, \dots, a_k with names corresponding to the parent variables. The value of b is stored in the “B” column; the value of `PnodeBetas(node)` is b .

The multiple-B pattern replaces the A -matrix with the B -matrix and the column “A” with “B”. Consider binary observable variable which uses the `OffsetConjunctive` rule. The linear predictor is:

$$a \min(\theta_1 - b + 1, \dots, \theta_K - b_K).$$

The value b_i is stored in the column “B.name i ” where $name_i$ is the name of the i th proficiency variable; the value of `PnodeBetas(node)` is the vector b_1, \dots, b_k with names corresponding to the parent variables. The value of a is stored in the “A” column; the value of `PnodeBetas(node)` is a .

When there are more than two states in the output variable, `PnodeRules`, `PnodeAlphas(node)` and `PnodeBetas(node)` become lists to indicate that a different value should be used for each transition between states. If there is a single value in the “Rules” column, or equivalently the value of `PnodeRules` is a scalar, then the same rule is repeated for each state transition. The same is true for `PnodeAlphas(node)` and `PnodeBetas(node)`. If these values are a list, that indicates that a different value is to be used for each transition. If they are a vector that means that different values (of discriminations for multiple-a rules or difficulties for multiple-b rules) are needed for the parent variables, but the same set of values is to be used for each state transition. If different values are to be used then the values are a list of vectors.

The necessary configuration of a ’s and b ’s depends on the type of link function. Here are the rules for the currently existing link functions:

normal (`normalLink`) This link function uses the same linear predictor for each transition, so there should be a single rule, and `PnodeAlphas(node)` and `PnodeBetas(node)` should both be vectors (with b of length 1 for a multiple-a rule). This rule also requires a positive value for the `PnodeLinkScale(node)` in the “LinkScale” column. The values in the “A.name” and “B.name” for rows after the first can be left as NA’s to indicate that the same values are reused.

graded response (`gradedResponse`) This link function models the probability of getting at or above each state and then calculates the differences between them to produce the conditional

probability table. In order to avoid negative probabilities, the probability of being in a higher state must always be nonincreasing. The surest way to ensure this is to both use the same combination rules at each state and the same set of discrimination parameters for each state. The difficulty parameters must be nondecreasing. Again, values for rows after the first can be left as NAs to indicate that the same value should be reused.

partial credit (`partialCredit`) This link function models the conditional probability from moving from the previous state to the current state. As such, there is no restriction on the rules or parameters. In particular, it can alternate between multiple-a and multiple-b style rules from row to row.

Another restriction that the use of the partial credit rule lifts is the restriction that all parent variable must be used in each transition. Note that there is one row of the Q -matrix (the inner Q -matrix) for each state transition. Only the parent variables with 1's in the particular state row are considered when building the `PnodeAlphas(node)` and `PnodeBetas(node)` for this model. Note that only the partial credit link function can take advantage of the multiple parents, the other two require all parents to be used for every state.

The function `Qmat2Pnet` takes a data frame containing a `Qmat` sets the properties of the corresponding nodes to match the description in the `Qmat`. It assumes that the proficiency variables have already been built, so it is almost always a good idea to first run `Omega2Pnet` to build the proficiency variables.

The function `Qmat2Pnet` loops through the values in the “Model” column, calling on the network `Warehouse` argument to supply (fetch or build) the requested network. It then loops through the values in the “Node” column, calling on the node `Warehouse` to supply them. First, it attempts to adjust the parents of `node` to match the Q -matrix. If the parent nodes are not in the current model, stub nodes are created by referencing the corresponding nodes in the proficiency model (the model corresponding to `PnetHub`). If `override` is TRUE, the network will be modified so that node has the indicated parents; if it is FALSE an error will be signaled if the pattern in the Q -matrix does not match the network structure. Then the values of various properties of a `Pnode`, in particular, the link function, the combination rules and the parameters, are set based on the values in `Qmat` (as described above).

Value

Invisibly returns a list of models visited.

Q -Matrix (Qmat) Structure

The output augmented Q -matrix is a data frame with the columns described below. The number of columns is variable, with items marked *prof* actually corresponding to a number of columns with names taken from the proficiency variables (the *prof* argument).

Model The name of the `Pnet` in which the node in this row lives.

Node The name of the `Pnode` described in this row. Except for the multiple rows corresponding to the same node, the value of this column needs to be unique within “Model”.

Nstates The number of states for this node. Generally, each node should have one fewer rows than this number.

State The name of the state for this row. This should be unique within the (“Model”, “Node”) combination.

- Link** The name of a link function. This corresponds to `PnodeLink(node)`.
- LinkScale** Either a positive number giving the link scale parameter or an NA if the link function does not need scale parameters. This corresponds to `PnodeLinkScale(node)`.
- prof** There is one column for each proficiency variable. This corresponds to the structural part of the Q -matrix. There should be 1 in this column if the named proficiency is used in calculating the transition to this state for this particular node, and a 0 otherwise.
- Rules** The name of the combination rule to use for this row. This corresponds to `PnodeRules(node)`.
- A.prof** There is one column for each proficiency with the proficiency name appended to “A.”. If a multiple-alpha style combination rule (e.g., `Compensatory`) this column should contain the appropriate discriminations, otherwise, its value should be NA.
- A** If a multiple-beta style combination rule (e.g., `OffsetConjunctive`) this column should contain the single discrimination, otherwise, its value should be NA.
- B.prof** There is one column for each proficiency with the proficiency name appended to “B.”. If a multiple-beta style combination rule (e.g., `OffsetConjunctive`) this column should contain the appropriate difficulty (negative intercept), otherwise, its value should be NA.
- B** If a multiple-beta style combination rule (e.g., `Compensatory`) this column should contain the single difficulty (negative intercept), otherwise, its value should be NA.
- PriorWeight** The amount of weight which should be given to the current values when learning conditional probability tables. See `PnodePriorWeight`.

Side Effects

This function destructively modifies the networks and nodes referenced in the Qmat and supplied by the warehouses.

Note that unlike typical R implementations, this is not necessarily safe. In particular, if the Qmat references 10 models, and an error is raised when trying to modify the 5th model, the first 4 models will be modified, the last 5 will not be and the 5th model may be partially modified. This is different from most R functions where changes are not committed unless the function returns successfully.

Author(s)

Russell Almond

References

Almond, R. G. (2010). ‘I can name that Bayesian network in two matrixes.’ *International Journal of Approximate Reasoning*. **51**, 167-178.

Almond, R. G. (presented 2017, August). Tabular views of Bayesian networks. In John-Mark Agosta and Tomas Singlair (Chair), *Bayesian Modeling Application Workshop 2017*. Symposium conducted at the meeting of Association for Uncertainty in Artificial Intelligence, Sydney, Australia. (International) Retrieved from <http://bmaw2017.azurewebsites.net/>

See Also

The inverse operation is `Pnet2Qmat`.

See [Warehouse](#) for description of the network and node warehouse arguments. The functions [PnetMakeStubNodes](#) and [PnetRemoveStubNodes](#) are used internally to create the stub nodes in evidence models.

See [partialCredit](#), [gradedResponse](#), and [normalLink](#) for currently available link functions. See [Conjunctive](#) and [OffsetConjunctive](#) for more information about available combination rules.

The node attributes set from the Omega matrix include: [PnodeParents\(node\)](#), [PnodeLink\(node\)](#), [PnodeLinkScale\(node\)](#), [PnodeQ\(node\)](#), [PnodeRules\(node\)](#), [PnodeAlphas\(node\)](#), [PnodeBetas\(node\)](#), and [PnodePriorWeight\(node\)](#)

Examples

```
## Sample Q matrix
## Sample Q matrix
Q1 <- read.csv(system.file("auxdata", "miniPP-Q.csv", package="Peanut"),
               stringsAsFactors=FALSE)

## Not run:
library(PNetica) ## Needs PNetica
sess <- NeticaSession()
startSession(sess)
curd <- getwd()

netman1 <- read.csv(system.file("auxdata", "Mini-PP-Nets.csv",
                              package="Peanut"),
                   row.names=1,stringsAsFactors=FALSE)

nodeman1 <- read.csv(system.file("auxdata", "Mini-PP-Nodes.csv",
                                package="Peanut"),
                    row.names=1,stringsAsFactors=FALSE)

omegamat <- read.csv(system.file("auxdata", "miniPP-omega.csv",
                                package="Peanut"),
                    row.names=1,stringsAsFactors=FALSE)

## Insures we are building nets from scratch
setwd(tempdir())
## Network and node warehouse, to create networks and nodes on demand.
Nethouse <- BNWarehouse(manifest=netman1,session=sess,key="Name")

Nodehouse <- NNWarehouse(manifest=nodeman1,
                        key=c("Model","NodeName"),
                        session=sess)

## Build the proficiency model first:
CM <- WarehouseSupply(Nethouse,"miniPP_CM")
CM1 <- Omega2Pnet(omegamat,CM,Nodehouse,override=TRUE,debug=TRUE)

## Build the nets from the Qmat

Qmat2Pnet(Q1, Nethouse,Nodehouse,debug=TRUE)

## Build the Qmat from the nets
```

```
## Generate a list of nodes
obs <-unlist(sapply(list(sess$nets$PPcompEM, sess$nets$PPconjEM,
                        sess$nets$PPtwostepEM, sess$nets$PPdurAttEM),
                    NetworkAllNodes))

Q2 <- Pnet2Qmat(obs, NetworkAllNodes(CM))

stopSession(sess)
setwd(curd)

## End(Not run)
```

RegressionGadget

Shiny gadget for editing compensatory pnodes

Description

These functions open a shiny application (in a browser window or other location) for editing a [Pnode](#) object. To reduce the complexity, the display assumes that [PnodeLink](#)(pnode) is [partialCredit](#) or [gradedResponse](#), and that [PnodeLink](#)(pnode) is [Compensatory](#) (Conjunctive or Disjunctive are also possibilities, but usually, the [OffsetGadget](#) is a better parameterization for these rules).

Usage

```
MakeRegressionGadget(pnode, useR2 = PnodeNumParents(pnode) > 0L,
                     color = "sienna")
RegressionGadget(pnode, useR2 = PnodeNumParents(pnode) > 0L,
                 color="sienna", viewer=shiny::paneViewer())
```

Arguments

| | |
|--------|---|
| pnode | A Pnode object to be modified. |
| color | A base color to use for barcharts (see barchart.CPF). Execute <code>colors()</code> for a list of choices. |
| useR2 | A logical value. If true, the link scale parameter is converted to an R^2 parameter in the gadget. |
| viewer | This is passed to the viewer argument of <code>shiny::runGadget</code> . |

Details

The `RegressionGadget` assumes that:

- The link function is [normallink](#).
- There is a single rule for all states, and that rule is [Compensatory](#).
- There is no inner Q-matrix, i.e., `PnodeQ(pnode)=TRUE`.
- There is a single beta value.

- That `PnodeLinkScale`(`pnode`) has a positive value.

This model is essentially a latent regression of the effective thetas onto the latent theta of the child space. The link scale parameter gives the residual variance (see [normalLink](#)).

An alternative parameterization, often more familiar to experts, uses the multiple R^2 instead of the residual variance. The `useR2` flag determines how the link scale parameter is conveyed on the gadget.

Value

The function `MakeRegressionGadget` returns a list of two functions, `ui` and `server`. These are meant to be passed to `shiny::runApp` to generate the actual app.

The function `RegressionGadget` will return the `pnode` object or throw a ‘Cancel-Error’.

Author(s)

Russell Almond

References

Almond, R. G. (2015) An IRT-based Parameterization for Conditional Probability Tables. Paper presented at the 2015 Bayesian Application Workshop at the Uncertainty in Artificial Intelligence Conference.

See Also

[Pnode](#), [calcDPCFrame](#), [barchart.CPF](#)
[OffsetGadget](#), [CompensatoryGadget](#), [DPCGadget](#)

Examples

```
## Not run:
library(PNetica) ## Requires implementation
sess <- NeticaSession()
startSession(sess)

tNet <- CreateNetwork("TestNet",sess)

theta1 <- NewDiscreteNode(tNet,"theta1",
  c("VH","High","Mid","Low","VL"))
PnodeStateValues(theta1) <- effectiveThetas(PnodeNumStates(theta1))
PnodeProbs(theta1) <- rep(1/PnodeNumStates(theta1),PnodeNumStates(theta1))
theta2 <- NewDiscreteNode(tNet,"theta2",
  c("VH","High","Mid","Low","VL"))
PnodeStateValues(theta2) <- effectiveThetas(PnodeNumStates(theta2))
PnodeProbs(theta2) <- rep(1/PnodeNumStates(theta1),PnodeNumStates(theta2))

## RegressionGadget

partial3 <- NewDiscreteNode(tNet,"partial3",
  c("FullCredit","PartialCredit","NoCredit"))
```

```

PnodeParents(partial3) <- list(theta1,theta2)

## Usual way to set rules is in constructor
partial3 <- Pnode(partial3,rules="Compensatory", link="normalLink")
PnodePriorWeight(partial3) <- 10
PnodeLinkScale(partial3) <- 1.0
BuildTable(partial3)

partial3 <- RegressionGadget(partial3)
partial3 <- RegressionGadget(partial3,useR2=FALSE)

## This expression can be used inside an Rmarkdown document
gadget <- MakeRegressionGadget(partial3)
shinyApp(gadget$ui,gadget$server,options(height=2000))

## No parent case
theta2 <- Pnode(theta2,link="normalLink",linkScale=1)
theta2 <- RegressionGadget(theta2)

DeleteNetwork(tNet)
stopSession(sess)

## End(Not run)

```

Statistic

Key functions for the Statistics class

Description

A **Statistic** is a functional that when applied to a Bayesian network returns a value. Usually, the statistic is a function of the distribution of a single node, but it could also be a function of several nodes. Statistic objects have a **calcStat** method, which when applied to a network, produces the value. Lists of statistics are often maintained by Bayes net engines to report values at designated times (e.g., after new evidence arrives). The Statistic function is the constructor or Statistic objects.

Usage

```

Statistic(fun, node, name = sprintf("%s(%s)", fun, node), ...)
calcStat(stat, net)

```

Arguments

| | |
|-----|--|
| fun | Object of class "character" giving a function to be applied to the nodes. The function should have signature <code>signature(net="Pnet", node)</code> , where node could be either a Pnode or a list of Pnodes (See details). |
|-----|--|

| | |
|------|--|
| node | Object of class "character" giving the name(s) of the node(s) that are referenced by the statistic. Note that these are not the actual node objects, as the network could be different at each call. |
| name | Object of class "character" giving a function to be applied to the nodes. The function should have signature(<code>net="Pnet"</code> , <code>node</code>), where <code>node</code> could be either a Pnode or a list of Pnodes . |
| ... | Other optional arguments for later extension. |
| stat | An object of class Statistic which will be applied to the net |
| net | A Pnet to which the statistic will be applied. |

Details

The [Statistic](#) class represents a functional which can be applied to a Bayes net (a distribution, [Pnet](#)), which returns a value of interest. Usually the functional is a function of the marginal or joint distribution of a number of nodes, [Pnode](#). Some canonical examples are the expected value and the median of the marginal distribution for a node.

Because the functional can be applied to different networks, the nodes are referenced by name instead of actual node objects. The `calcStat` method finds the nodes in the network, and then calls the referenced fun with arguments `signature(net="Pnet", node)`, where `node` can either be a node or list of nodes. (Note that the network object may or may not be needed to calculate the statistic value).

Note that the statistic is free to return any kind of value. The mean of a discrete variable is typically numeric (using [PnodeStateValues](#) to link states of the node with numeric values). The mode and median return a [factor](#) variable, and the margin is a vector of values on the unit simplex.

The current statistics are currently supported are:

[PnodeMargin](#) Provides the marginal distribution of a node.

[PnodeEAP](#) Provides the expected a posteriori (i.e., mean) of a node using numeric values for the state from [PnodeStateValues](#).

[PnodeSD](#) Provides the standard deviation of a node using numeric values for the state from [PnodeStateValues](#).

[PnodeMedian](#) Provides the median value for a node, that is if the states are ordered, the one which is reached at a probability mass of 0.5.

[PnodeMode](#) Returns the most likely state for (the marginal distribution of) node.

Value

The `Statistic` function returns an object of class [Statistic](#).

Author(s)

Russell Almond

References

Almond, R.G., Mislevy, R.J. Steinberg, L.S., Yan, D. and Williamson, D. M. (2015). *Bayesian Networks in Educational Assessment*. Springer. Chapter 13.

See Also

Class: [Statistic](#)

[calcStat](#)

Available Statistic functions: [PnodeMargin](#), [PnodeEAP](#), [PnodeSD](#), [PnodeMedian](#), [PnodeMode](#).

These statistics will likely produce errors unless [PnetCompile](#) has been run first.

Examples

```
## Not run:

library(PNetica) ## Need a specific implementation
sess <- NeticaSession()
startSession(sess)

irt10.base <- ReadNetworks(system.file("testnets", "IRT10.2PL.base.dne",
                                     package="PNetica"),
                          session=sess)
irt10.base <- as.Pnet(irt10.base) ## Flag as Pnet, fields already set.
irt10.theta <- PnetFindNode(irt10.base,"theta")
irt10.items <- PnetPnodes(irt10.base)
## Flag items as Pnodes
for (i in 1:length(irt10.items)) {
  irt10.items[[i]] <- as.Pnode(irt10.items[[i]])
}
## Make some statistics
marginTheta <- Statistic("PnodeMargin","theta","Pr(theta)")
meanTheta <- Statistic("PnodeEAP","theta","EAP(theta)")
sdTheta <- Statistic("PnodeSD","theta","SD(theta)")
medianTheta <- Statistic("PnodeMedian","theta","Median(theta)")
modeTheta <- Statistic("PnodeMedian","theta","Mode(theta)")

BuildAllTables(irt10.base)
CompileNetwork(irt10.base) ## Netica requirement

calcStat(marginTheta,irt10.base)
calcStat(meanTheta,irt10.base)
calcStat(sdTheta,irt10.base)
calcStat(medianTheta,irt10.base)
calcStat(modeTheta,irt10.base)

DeleteNetwork(irt10.base)
stopSession(sess)

## End(Not run)
```

| | |
|-----------------|-------------------|
| Statistic-class | Class "Statistic" |
|-----------------|-------------------|

Description

A statistic is a functional that when applied to a Bayesian network returns a value. Usually, the statistic is a function of the distribution of a single node, but it could also be a function of several nodes. Statistic objects have a `calcStat` method, which when applied to a network, produces the value. Lists of statistics are often maintained by Bayes net engines to report values at designated times (e.g., after new evidence arrives).

Objects from the Class

Objects are created using the function `Statistic(fun, node, name, ...)`.

Slots

name: Object of class "character" giving an identifier for the statistic.

node: Object of class "character" giving the name(s) of the node(s) that are referenced by the statistic. Note that these are not the actual node objects, as the network could be different at each call.

fun: Object of class "character" giving a function to be applied to the nodes. The function should have signature `(net="Pnet", node)`, where `node` could be either a `Pnode` or a list of `Pnodes`.

Methods

calcStat signature(`stat = "Statistic"`, `net`): This method (a) finds the nodes referenced in `node`, (b) applies `fun` (using `do.call` to `net` and the actual nodes).

StatName signature(`x = "Statistic"`): Returns the name of the statistic.

show signature(`obj = "Statistic"`): Returns a printable representation of the statistic.

Author(s)

Russell Almond

References

Almond, R.G., Mislevy, R.J. Steinberg, L.S., Yan, D. and Williamson, D. M. (2015). *Bayesian Networks in Educational Assessment*. Springer. Chapter 13.

See Also

Available Statistic functions: `PnodeMargin`, `PnodeEAP`, `PnodeSD`, `PnodeMedian`, `PnodeMode`.

Constructor function: `Statistic`

`calcStat`

Examples

```

## Not run:

library(PNetica) ## Need a specific implementation
sess <- NeticaSession()
startSession(sess)

irt10.base <- ReadNetworks(system.file("testnets", "IRT10.2PL.base.dne",
                                     package="PNetica"), session=sess)
irt10.base <- as.Pnet(irt10.base) ## Flag as Pnet, fields already set.
irt10.theta <- NetworkFindNode(irt10.base,"theta")
irt10.items <- PnetPnodes(irt10.base)
## Flag items as Pnodes
for (i in 1:length(irt10.items)) {
  irt10.items[[i]] <- as.Pnode(irt10.items[[i]])
}
## Make some statistics
marginTheta <- Statistic("PnodeMargin","theta","Pr(theta)")
meanTheta <- Statistic("PnodeEAP","theta","EAP(theta)")
sdTheta <- Statistic("PnodeSD","theta","SD(theta)")
medianTheta <- Statistic("PnodeMedian","theta","Median(theta)")
modeTheta <- Statistic("PnodeMedian","theta","Mode(theta)")

BuildAllTables(irt10.base)
CompileNetwork(irt10.base) ## Netica requirement

calcStat(marginTheta,irt10.base)
calcStat(meanTheta,irt10.base)
calcStat(sdTheta,irt10.base)
calcStat(medianTheta,irt10.base)
calcStat(modeTheta,irt10.base)

DeleteNetwork(irt10.base)
stopSession(sess)

## End(Not run)

```

topsort

Topologically sorts the rows and columns of an Omega matrix

Description

The structural part of the Ω -matrix is an incidence matrix where the entry is 1 if the node represented by the column is a parent of the node represented by the child. This sorts the rows and columns of the matrix (which should have the same names) so that the ancestors of a node always appear prior to it in the sequence. As a consequence, the values in the upper triangle of the Ω -matrix are always zero after sorting.

Usage

```
topsort(Omega, noisy = FALSE)
```

Arguments

Omega A square matrix of 1's and zeros which corresponds to an acyclic directed graph.
noisy A logical value. If true, details of progress through the algorithm are printed.

Value

An ordering of the rows and columns which will sort the matrix.

Note

This will generate an error if the graph represented by the matrix is cyclic.

Author(s)

Russell Almond

See Also

[Pnet2Omega](#) uses this function to sort the columns in the Omega matrix.

Examples

```
## Sample Omega matrix.
omegamat <- read.csv(system.file("auxdata", "miniPP-omega.csv",
                                package="Peanut"),
                    row.names=1,stringsAsFactors=FALSE)
omega <- as.matrix(omegamat[,2:6])

## omega is already sorted so scramble it.
shuffle <- sample.int(5)
omegas <- omega[shuffle,shuffle]

ord <- topsort(omegas)
omegas[ord,ord]
```

Description

A warehouse is an object which stores a collection of [Pnodes](#) or [Pnets](#). When requested, it will supply the given object. If the object already exists, it is returned. If it does not yet exist, it is built using meta-data in the warehouse's manifest.

Usage

```
WarehouseSupply(warehouse, name, restoreOnly=FALSE)
## S4 method for signature 'ANY'
WarehouseSupply(warehouse, name, restoreOnly=FALSE)
WarehouseFetch(warehouse, name)
WarehouseMake(warehouse, name, restoreOnly=FALSE)
WarehouseFree(warehouse, name)
WarehouseSave(warehouse, obj)
ClearWarehouse(warehouse)
is.valid(warehouse,object)
is.PnetWarehouse(obj)
is.PnodeWarehouse(obj)
```

Arguments

| | |
|-------------|---|
| warehouse | A warehouse object from which the object is to be created. |
| name | A character vector giving the name of the object. Note that for net warehouses, the key is usually has length one, but for node warehouses, this usuall has the form (<i>model,node</i>). |
| obj | An object whose type is to be determined, or a network to be saved. |
| object | An object to be tested to see if it a valid object from this warehouse. |
| restoreOnly | A logical value. If true, then WarehouseMake will restore an object from a file, if available, but will generate an error if the instruction file is not available. |

Details

The warehouse is a combination of a cache and a factory. The idea is that when a Pnet or Pnode object is needed, it is requested from the corresponding warehouse. If the object exists, it is returned. If the object does not exist, then the information in the manifest (see [WarehouseManifest\(\)](#)) is used to create a new object. The key function is `WarehouseSupply(warehouse, name)`; this function looks for an object corresponding to *name* in warehouse. If it exists, it is returned, if not a new one is created.

The generic functions `WarehouseFetch(warehouse, name)` and `WarehouseMake(warehouse, name)` implement the supply protocol. `WarehouseFetch(warehouse, name)` searches for an object corresponding to *name* in the warehouse and returns it if it exists or returns NULL if it does not. The generic function `WarehouseMake(warehouse, name)` creates the object using the data in the manifest.

The function `WarehouseSave` saves the object back out to long term storage. This is particularly used for networks which are often prebuilt and then loaded when needed. Setting the argument `restoreOnly=TRUE` in `WarehouseMake` or `WarehouseSupply` will recreate an object from a save, but not create a new object.

The `WarehouseFree` and `WarehouseClear` functions complete the Warehouse protocol. These respectively remove the named object from the cache, and clear the cache. Note that these may our may not make sense with the implementation. (In the current [PNetica-package](#) implementation, the cache is maintained by the underlying [RNetica](#) objects, and hence it doesn't make sense to free an object without deleting it.)

Each warehouse has a manifest which supplies the necessary data to build a particular object. The generic function `WarehouseManifest()` accesses the manifest, which generally takes the form of a `data.frame` object. The functions `BuildNetManifest()` and `BuildNodeManifest()` build manifests for network and node objects respectively. The generic function `WarehouseData(warehouse, name)` returns the rows of the manifest which correspond to a particular name.

The Peanut package is concerned with two kinds of warehouses: Pnet warehouses and Pnode warehouses. Pnet warehouses contain Pnets, and the key is the name of the network. Each Pnet corresponds to a single line in the manifest, and the *name* is a character scalar. A Pnet warehouse should return true when the generic function `is.PnetWarehouse()` is called.

Pnode warehouses contain Pnodes, and the *name* is a character vector of length 2, with structure `(netname, nodename)`. This is because nodes with the same name will frequently exist in two different networks. Currently the manifest for a node contains one line for each possible state of the node. A Pnode warehouse should return true when the generic function `is.PnodeWarehouse()` is called.

The function `is.valid` checks to see if the object is of a type managed by the warehouse, and that it has valid data. In particular, the `RNetica` package uses pointers to objects in Netica space (as might other implementations). The `is.valid()` function checks that the current `Pnet` and `Pnode` object point to valid objects in the external heap if this is applicable to the application.

The warehouse object is an abstract class, and implementing classes need to provide methods for the generic functions `WarehouseFetch()`, `WarehouseMake()`, `WarehouseFree()`, `WarehouseData()`, `WarehouseManifest()`, `WarehouseCopy()`, `is.legal.name()`, `as.legal.name()`, `is.valid()` and `ClearWarehouse()` as well as one of the generic functions `is.PnetWarehouse` or `is.PnodeWarehouse`.

There are two reference implementations in `BNWarehouse` and `NNWarehouse` (network and node warehouses respectively). Both of these take advantage of the fact that the session and network objects in `RNetica` have built in environments which cache the networks and nodes respectively. The `RGAutils::Warehouse-class` object is a generic implementation that also may be of some use to potential implementors.

Value

The return type of most functions will depend on the type of the warehouse. In most cases, the functions return an object of the type of the warehouse.

Pnet Warehouses

These return TRUE from the function `is.PnetWarehouse()`, and an object of type `Pnet` or NULL from the functions `WarehouseSupply()`, `WarehouseFetch()`, and `WarehouseMake()`. NULL is returned when the requested net is not in the warehouse or the manifest.

Pnode Warehouses

These return TRUE from the function `is.PnodeWarehouse()`, and an object of type `Pnode` or NULL from the functions `WarehouseSupply()`, `WarehouseFetch()`, and `WarehouseMake()`. NULL is returned when the requested net is not in the warehouse or the manifest.

The returns from the functions `WarehouseFree()` and `ClearWarehouse()` are arbitrary depending on the implementation.

Note

There seem to be two use cases for WarehouseMake and WarehouseSupply when working with networks. During model construction, calling this function should create a new blank network. During scoring, it should load a prebuilt network and signal an error if the network is missing. The restoreOnly flag is designed to distinguish between these cases.

Note

The cache part of the warehouse, almost certainly needs to be implemented using the reference class system of Chambers (2016). In particular, an `environment` object provides the kind of persistent storage and object persistence and uniqueness necessary (this breaks the usual functional programming paradigm of R).

Author(s)

Russell G. Almond

References

Almond, R. G. (presented 2017, August). Tabular views of Bayesian networks. In John-Mark Agosta and Tomas Singlair (Chair), *Bayesian Modeling Application Workshop 2017*. Symposium conducted at the meeting of Association for Uncertainty in Artificial Intelligence, Sydney, Australia. (International) Retrieved from <http://bmaw2017.azurewebsites.net/>

Chambers, J. M. (2016) *Extending R*. CRC Press.

See Also

Other warehouse functions: [WarehouseCopy](#), [is.legal.name](#)

These functions support the manifest process: [WarehouseManifest\(\)](#), [WarehouseData\(\)](#)

These functions construct manifests: [BuildNetManifest\(\)](#), [BuildNodeManifest\(\)](#)

These functions use the warehouse to build networks: [Omega2Pnet](#) [Qmat2Pnet](#)

Examples

```
## Not run:
## Requires PNetica package
library(PNetica)
sess <- NeticaSession()
startSession(sess)

### This tests the manifest and factory protocols.

netman1 <- read.csv(system.file("auxdata", "Mini-PP-Nets.csv",
                              package="PNetica"),
                  row.names=1, stringsAsFactors=FALSE)

nodeman1 <- read.csv(system.file("auxdata", "Mini-PP-Nodes.csv",
                              package="PNetica"),
                  stringsAsFactors=FALSE)
```

```

### Test Net building
Nethouse <- BNWarehouse(manifest=netman1,session=sess,key="Name")
stopifnot(is.PnetWarehouse(Nethouse))

curd <- setwd(system.file("testnets",package="PNetica"))
CM <- WarehouseSupply(Nethouse,"miniPP_CM")
stopifnot(is.null(WarehouseFetch(Nethouse,"PPcompEM")))
EM1 <- WarehouseMake(Nethouse,"PPcompEM")

EMs <- lapply(c("PPcompEM","PPconjEM", "PPtwostepEM", "PPdurAttEM"),
             function(nm) WarehouseSupply(Nethouse,nm))

### Test Node Building with already loaded nets

Nodehouse <- NNWarehouse(manifest=nodeman1,
                        key=c("Model","NodeName"),
                        session=sess)
stopifnot(is.PnodeWarehouse(Nodehouse))

phyd <- WarehouseData(Nodehouse,c("miniPP_CM","Physics"))

p3 <- MakePnode.NeticaNode(CM,"Physics",phyd)

phys <- WarehouseSupply(Nodehouse,c("miniPP_CM","Physics"))
stopifnot(p3==phys)

for (n in 1:nrow(nodeman1)) {
  name <- as.character(nodeman1[n,c("Model","NodeName")])
  if (is.null(WarehouseFetch(Nodehouse,name))) {
    cat("Building Node ",paste(name,collapse=":"),"\n")
    WarehouseSupply(Nodehouse,name)
  }
}

WarehouseFree(Nethouse,PnetName(EM1))
stopifnot(!is.valid(Nethouse,EM1))

setwd(curd)
stopSession(sess)

## End(Not run)

```


Description

The creates an object with a new name that shares the appropriate properties with the original object. How this is handled depends on the type of objects conatined in the Warehouse. The new object is placed in the warehouse inventory, although not in the manifest.

Usage

```
WarehouseCopy(warehouse, obj, newname)
```

Arguments

| | |
|-----------|---|
| warehouse | A Warehouse object. |
| obj | An object generated from the warehouse. |
| newname | A new name for the warehouse object. |

Details

For the [PnetWarehouse](#), this is essentially a copy net function. The newname is the name of the new network, and it will be an error if this is the same as the name of an existing network.

For the [PnodeWarehouse](#), this is essentially a copy node function. Note that in this case, the new name must be a character vector with two element: the new net name, and the new node name. If the new net name is not the same as the net containing the obj argument, then the node will be copied into the new net. If it is the same, then a new node will be added to the existing net. Note that at least one part of the name (the net name or the node name) must be different.

Value

A new object of the same type as obj, that is a new object managed by the warehouse.

Author(s)

Russell Almond

See Also

[Warehouse](#), [WarehouseFree](#)

Examples

```
## Not run: ## Requires implementation
library(PNetica)

sess <- NeticaSession()
startSession(sess)

## BNWarehouse is the PNetica Net Warehouse.
## This provides an example network manifest.
table.dir <- system.file("auxdata", package="Peanut")
net.dir <- system.file("testnets", package="PNetica")
```

```

netman1 <- read.csv(file.path(table.dir,"Mini-PP-Nets.csv"),
                    row.names=1, stringsAsFactors=FALSE)
Nethouse <- BNWarehouse(manifest=netman1,session=sess,key="Name",
                        address=net.dir)

CM <- WarehouseSupply(Nethouse, "miniPP_CM")
SM1 <- WarehouseCopy(Nethouse, CM, "Student1")

stopifnot(length(NetworkAllNodes(CM))==length(NetworkAllNodes(SM1)))

## This expression provides an example Node manifest
nodeman1 <- read.csv(file.path(table.dir,"Mini-PP-Nodes.csv"),
                    row.names=1,stringsAsFactors=FALSE)
Nodehouse <- NNWarehouse(manifest=nodeman1,
                        key=c("Model","NodeName"),
                        session=sess)
NNodes <- length(NetworkAllNodes(CM))

IterD <- NetworkAllNodes(CM)$IterativeD

## Copy within net.
Explain <- WarehouseCopy(Nodehouse,IterD,c("miniPP_CM","Explanation"))
stopifnot(PnodeName(Explain)=="Explanation",
          PnetName(PnodeNet(Explain))=="miniPP_CM",
          all.equal(PnodeStates(IterD),PnodeStates(Explain)))

## Copy between nets.
Explain1 <- WarehouseCopy(Nodehouse,Explain,c("Student1","Explanation"))
stopifnot(PnodeName(Explain1)=="Explanation",
          PnetName(PnodeNet(Explain1))=="Student1",
          all.equal(PnodeStates(IterD),PnodeStates(Explain1)))

## End(Not run)

```

WarehouseManifest

Manipulates the manifest for a warehouse

Description

A [Warehouse](#) is an object which can either retrieve an existing object or create a new one on demand. The *manifest* is a [data.frame](#) which contains data used for building the objects managed by the warehouse on demand. The function `WarehouseManifest` access the entire manifest and `WarehouseData` extracts the warehouse data for a single item. `WarehouseInventory` returns a list of objects which have already been built.

Usage

```
WarehouseManifest(warehouse)
WarehouseManifest(warehouse) <- value
WarehouseData(warehouse, name)
WarehouseInventory(warehouse)
```

Arguments

| | |
|-----------|--|
| warehouse | A Warehouse object |
| value | A <code>data.frame</code> which provides the new manifest data. The required columns depend on the type of data managed by the warehouse |
| name | A character vector which provides a key for a single object in the warehouse. |

Details

The [Warehouse](#) design pattern is a combination of a factory and a cache. The idea is that if an object is needed, the warehouse will search the cache and return it if it already exists. If it does not exist, the warehouse will create it using the data in the *manifest*. The manifest is a `data.frame` with one or more columns serving as keys. The function `ManifestData` extracts the data necessary to create a given object.

Two kinds of warehouses are needed in the Peanut interface: *net warehouses* and *node warehouses*.

Net Warehouse. A network warehouse will return an already existing network, read the network from disk, or build it from scratch as needed. The required fields for a network warehouse manifest are given in the documentation for [BuildNetManifest](#). The key is the “Name” column which should be unique for each row. The *name* argument to `WarehouseData` should be a character scalar corresponding to name, and it will return a `data.frame` with a single row.

Node Warehouse. A network warehouse will return an already existing node in a network, or build it from scratch as needed. The required fields for a network warehouse manifest are given in the documentation for [BuildNodeManifest](#). Note that node names are only unique within a network, so the key is the pair of columns “Model” and “NodeName”. If the variable has more than 2 states, there may be more than two rows of the manifest which correspond to that node. These should have unique values for the field “StateName”. The *name* argument to `WarehouseData` should be a character vector with the first element being the model name and the section the node name. That function will return a `data.frame` with multiple rows (depending on the number of states).

Value

The function `WarehouseManifest` returns a `data.frame` giving the complete warehouse manifest. The function `WarehouseData` returns selected rows from that `data.frame`.

The setter function returns the warehouse object.

The function `WarehouseInventory` returns a data frame where each row corresponds to the key of an object which has been built.

Note

The best way to build a manifest is probably to call `BuildNetManifest` or `BuildNodeManifest` on a couple of objects and use that to build a skeleton, which can then be edited with the specific needed data.

Author(s)

Russell Almond

References

Almond, R. G. (presented 2017, August). Tabular views of Bayesian networks. In John-Mark Agosta and Tomas Singlair (Chair), *Bayesian Modeling Application Workshop 2017*. Symposium conducted at the meeting of Association for Uncertainty in Artificial Intelligence, Sydney, Australia. (International) Retrieved from <http://bmaw2017.azurewebsites.net/>

See Also

[Warehouse](#), [BuildNetManifest](#), [BuildNodeManifest](#)

Examples

```
## This provides an example network manifest.
netman1 <- read.csv(system.file("auxdata", "Mini-PP-Nets.csv",
                             package="Peanut"),
                  row.names=1, stringsAsFactors=FALSE)

## This provides an example node manifest
nodeman1 <- read.csv(system.file("auxdata", "Mini-PP-Nodes.csv",
                              package="Peanut"),
                   row.names=1, stringsAsFactors=FALSE)

## Not run:
library(PNetica) ## Example requires PNetica
sess <- NeticaSession()
startSession(sess)

## BNWarehouse is the PNetica Net Warehouse.
Nethouse <- BNWarehouse(manifest=netman1, session=sess, key="Name")
stopifnot(all.equal(WarehouseManifest(Nethouse), netman1))

stopifnot(all.equal(WarehouseData(Nethouse, "miniPP_CM"),
                  netman1["miniPP_CM", ]))

netman2 <- netman1
netman2["miniPP_CM", "Pathname"] <- "mini_CM.dne"
WarehouseManifest(Nethouse) <- netman2

stopifnot(all.equal(WarehouseData(Nethouse, "miniPP_CM"),
```

```
netman2["miniPP_CM",]))

Nodehouse <- NNWarehouse(manifest=nodeman1,
                          key=c("Model", "NodeName"),
                          session=sess)

WarehouseData(Nodehouse, c("miniPP_CM", "Physics"))

stopSession(sess)

## End(Not run)
```

Index

* CPT

- CompensatoryGadget, 18
- DPCGadget, 20
- OffsetGadget, 36
- RegressionGadget, 141

* Compensatory

- CompensatoryGadget, 18
- DPCGadget, 20
- RegressionGadget, 141

* Conjunctive

- DPCGadget, 20
- OffsetGadget, 36

* Disjunctive

- DPCGadget, 20
- OffsetGadget, 36

* IO

- PnetSerialize, 75

* attributes

- PnodeLabels, 95

* attribute

- PnetHub, 64
- PnetName, 67
- PnetPathname, 69
- PnetTitle, 78
- PnodeName, 107
- PnodeStateTitles, 124
- PnodeStateValues, 126
- PnodeTitle, 130

* attrib

- PnetPnodes, 70
- PnetPriorWeight, 72
- PnodeBetas, 87
- PnodeDefaultAlphas, 91
- PnodeLink, 96
- PnodeLinkScale, 99
- PnodeLnAlphas, 101
- PnodeParentTvals, 111
- PnodeQ, 117
- PnodeRules, 120

* classes

- Pnet, 43
- Pnet-class, 46
- PnetWarehouse-class, 79
- Pnode, 82
- Pnode-class, 85
- PnodeMin, 105
- PnodeWarehouse-class, 131
- Statistic, 143
- Statistic-class, 146

* class

- Warehouse, 148

* distribution

- BuildTable, 12
- Omega2Pnet, 38
- Pnet2Omega, 48
- Pnet2Qmat, 52
- Qmat2Pnet, 134

* gradedResponse

- CompensatoryGadget, 18
- DPCGadget, 20

* graphs

- BuildNetManifest, 6
- BuildNodeManifest, 8
- calcPnetLLike, 16
- GEMfit, 25
- Peanut-package, 3
- Pnet, 43
- PnetAdjoin, 58
- PnetCompile, 61
- PnetFindNode, 62
- PnetSerialize, 75
- PnetWarehouse-class, 79
- Pnode, 82
- PnodeEvidence, 92
- PnodeParents, 109
- PnodeStates, 122
- PnodeStats, 128
- Statistic, 143

- Statistic-class, [146](#)
- WarehouseManifest, [154](#)
- * **graph**
 - Omega2Pnet, [38](#)
 - Pnet2Omega, [48](#)
 - Pnet2Qmat, [52](#)
 - PnetHub, [64](#)
 - PnetMakeStubNodes, [65](#)
 - PnetName, [67](#)
 - PnetPathname, [69](#)
 - PnetTitle, [78](#)
 - PnodeDefaultAlphas, [91](#)
 - PnodePostWeight, [113](#)
 - Qmat2Pnet, [134](#)
 - topsort, [147](#)
- * **interface**
 - BuildNetManifest, [6](#)
 - BuildNodeManifest, [8](#)
 - CompensatoryGadget, [18](#)
 - DPCGadget, [20](#)
 - OffsetGadget, [36](#)
 - Pnet, [43](#)
 - PnetFindNode, [62](#)
 - PnetHub, [64](#)
 - PnetName, [67](#)
 - PnetPathname, [69](#)
 - PnetTitle, [78](#)
 - Pnode, [82](#)
 - PnodeLabels, [95](#)
 - PnodeName, [107](#)
 - PnodeParents, [109](#)
 - PnodeProbs, [115](#)
 - PnodeStates, [122](#)
 - PnodeStateTitles, [124](#)
 - PnodeStateValues, [126](#)
 - PnodeTitle, [130](#)
 - RegressionGadget, [141](#)
 - Warehouse, [148](#)
 - WarehouseManifest, [154](#)
- * **language**
 - flog.try, [24](#)
- * **manip**
 - calcExpTables, [14](#)
 - CompensatoryGadget, [18](#)
 - DPCGadget, [20](#)
 - flattenStats, [23](#)
 - GEMfit, [25](#)
 - is.legal.name, [30](#)
 - isPnodeContinuous, [31](#)
 - maxAllTableParams, [33](#)
 - OffsetGadget, [36](#)
 - PnetAdjoin, [58](#)
 - PnetMakeStubNodes, [65](#)
 - RegressionGadget, [141](#)
 - WarehouseCopy, [152](#)
- * **model**
 - PnodeProbs, [115](#)
- * **normalLink**
 - RegressionGadget, [141](#)
- * **objects**
 - WarehouseCopy, [152](#)
- * **package**
 - Peanut-package, [3](#)
- * **partialCredit**
 - CompensatoryGadget, [18](#)
 - DPCGadget, [20](#)
 - OffsetGadget, [36](#)
- * **utilities**
 - PnetFindNode, [62](#)
- * **util**
 - flog.try, [24](#)
- as.legal.name, [150](#)
- as.legal.name (is.legal.name), [30](#)
- as.Pnet (Pnet), [43](#)
- as.Pnode (Pnode), [82](#)
- barchart.CPF, [19–22](#), [37](#), [141](#), [142](#)
- BNWarehouse, [30](#), [80](#), [81](#), [150](#)
- BuildAllTables, [3](#), [14](#), [17](#), [26–28](#), [34](#), [44](#), [45](#), [70](#), [71](#), [95](#), [96](#)
- BuildAllTables (BuildTable), [12](#)
- BuildNetManifest, [6](#), [11](#), [44](#), [45](#), [79–81](#), [150](#), [151](#), [155](#), [156](#)
- BuildNodeManifest, [6](#), [7](#), [8](#), [131](#), [134](#), [150](#), [151](#), [155](#), [156](#)
- BuildTable, [12](#), [15](#), [31](#), [44](#), [52](#), [73](#), [83](#), [84](#), [89](#), [98](#), [101](#), [104](#), [109](#), [112](#), [116](#), [119](#), [122](#), [135](#)
- calcDNTable, [98](#)
- calcDPCFrame, [20](#), [22](#), [37](#), [142](#)
- calcDPCTable, [3](#), [12](#), [13](#), [83](#), [84](#), [87–89](#), [97–99](#), [101–104](#), [111](#), [112](#), [117–122](#)
- calcExpTables, [14](#), [17](#), [26–28](#), [34](#), [35](#), [44](#), [45](#), [113](#), [114](#)
- calcPnetLLike, [15](#), [16](#), [26–28](#), [34](#), [35](#), [44](#), [45](#)

- calcStat, [61](#), [128](#), [129](#), [143–146](#)
- calcStat (Statistic), [143](#)
- calcStat, Statistic-method
(Statistic-class), [146](#)
- ClearWarehouse, [80](#), [133](#)
- ClearWarehouse (Warehouse), [148](#)
- Compensatory, [18](#), [19](#), [21](#), [36](#), [39–42](#), [48–54](#),
[56](#), [88](#), [89](#), [91](#), [98](#), [101–104](#), [121](#),
[122](#), [135–137](#), [139](#), [141](#)
- CompensatoryGadget, [18](#), [22](#), [37](#), [142](#)
- Conjunctive, [42](#), [50](#), [56](#), [88](#), [103](#), [121](#), [140](#)
- CPA, [116](#)
- CPF, [116](#)
- CPTtools, [3](#), [4](#)
- data.frame, [7](#), [9](#), [40](#), [41](#), [49](#), [50](#), [80](#), [132](#), [135](#),
[150](#), [154](#), [155](#)
- DBI, [4](#)
- dimnames, [123](#)
- Disjunctive, [88](#), [103](#), [121](#)
- DPCGadget, [20](#), [20](#), [37](#), [142](#)
- dump, [76](#)
- effectiveThetas, [39](#), [112](#)
- environment, [151](#)
- expand.grid, [88](#), [97](#), [102](#), [111](#), [112](#), [120](#)
- factor, [144](#)
- flattenStats, [23](#)
- flog.appender, [27](#)
- flog.logger, [7](#), [10](#), [13](#), [24–27](#), [35](#), [41](#)
- flog.namespace, [24](#)
- flog.threshold, [7](#), [10](#), [13](#), [26](#), [27](#), [35](#), [41](#)
- flog.try, [24](#)
- GEMfit, [3](#), [14](#), [15](#), [17](#), [25](#), [26](#), [34](#), [35](#), [44](#), [45](#),
[70–73](#), [83](#), [95](#), [96](#), [113](#), [114](#)
- get, [76](#)
- GetPriorWeight, [12](#), [13](#), [15](#), [70](#), [71](#), [84](#)
- GetPriorWeight (PnetPriorWeight), [72](#)
- gradedResponse, [18](#), [19](#), [36](#), [53–56](#), [88](#), [97](#),
[100](#), [103](#), [118](#), [121](#), [136](#), [137](#), [140](#),
[141](#)
- is.legal.name, [30](#), [150](#), [151](#)
- is.Pnet, [47](#)
- is.Pnet (Pnet), [43](#)
- is.PnetWarehouse, [80](#)
- is.PnetWarehouse (Warehouse), [148](#)
- is.Pnode (Pnode), [82](#)
- is.PnodeWarehouse, [133](#)
- is.PnodeWarehouse (Warehouse), [148](#)
- is.valid (Warehouse), [148](#)
- isOffsetRule, [54](#), [91](#), [92](#), [137](#)
- isPnodeContinuous, [31](#), [84](#), [86](#), [93](#), [116](#), [127](#)
- MakeCompensatoryGadget
(CompensatoryGadget), [18](#)
- MakeDPCGadget (DPCGadget), [20](#)
- MakeOffsetGadget (OffsetGadget), [36](#)
- MakePnet.NeticaBN, [81](#)
- MakePnode.NeticaNode, [134](#)
- MakeRegressionGadget
(RegressionGadget), [141](#)
- mapDPC, [14](#), [26](#), [33–35](#), [84](#), [88](#), [89](#), [98](#),
[101–104](#), [112](#), [119](#), [122](#)
- maxAllTableParams, [14](#), [15](#), [17](#), [25–28](#), [33](#),
[44](#), [45](#), [114](#)
- maxCPTParam, [45](#), [83](#), [84](#), [89](#), [98](#), [101](#), [104](#),
[112](#), [119](#), [122](#)
- maxCPTParam (maxAllTableParams), [33](#)
- mongo, [75](#), [76](#)
- NeticaBN, [44](#), [46](#), [48](#)
- NeticaNode, [82](#), [85](#), [87](#)
- NeticaSession, [81](#)
- NNWarehouse, [30](#), [133](#), [134](#), [150](#)
- NodeExperience, [34](#)
- NodeProbs, [34](#)
- normalize, [116](#)
- normalLink, [39–42](#), [48–51](#), [54–56](#), [100](#), [137](#),
[140–142](#)
- OffsetConjunctive, [21](#), [42](#), [50](#), [54](#), [56](#), [88](#),
[89](#), [91](#), [98](#), [101](#), [103](#), [104](#), [121](#), [122](#),
[137](#), [139](#), [140](#)
- OffsetDisjunctive, [21](#), [53](#), [88](#), [89](#), [103](#), [121](#),
[136](#)
- OffsetGadget, [18](#), [20](#), [22](#), [36](#), [36](#), [141](#), [142](#)
- Omega2Pnet, [6](#), [9](#), [38](#), [45](#), [50](#), [51](#), [138](#), [151](#)
- partialCredit, [18](#), [19](#), [21](#), [36](#), [37](#), [52–56](#), [88](#),
[91](#), [97](#), [100](#), [103](#), [118](#), [121](#), [135–138](#),
[140](#), [141](#)
- Peanut (Peanut-package), [3](#)
- Peanut-package, [3](#)
- Pnet, [3](#), [6](#), [12](#), [14–17](#), [26](#), [28](#), [33–35](#), [39](#), [40](#),
[43](#), [48–50](#), [52](#), [53](#), [55](#), [58](#), [60](#), [61](#),

- 63–65, 68–73, 75–79, 83, 84, 128,
 135, 136, 138, 144, 148, 150
 Pnet, ANY-method (Pnet), 43
 Pnet-class, 46
 Pnet2Omega, 6, 8, 39, 41, 42, 45, 48, 48, 148
 Pnet2Qmat, 6, 8, 42, 45, 50, 52, 64, 139
 PnetAdjoin, 44, 45, 47, 58, 64, 66
 PnetCompile, 13, 47, 61, 94, 129, 145
 PnetDescription, 6, 7, 44, 45, 47, 80
 PnetDescription (PnetTitle), 78
 PnetDescription<-, 47
 PnetDescription<- (PnetTitle), 78
 PnetDetach, 44, 45, 47
 PnetDetach (PnetAdjoin), 58
 PnetFindNode, 45, 47, 62, 108
 PnetHub, 6, 7, 9, 11, 44, 47, 60, 64, 66, 80,
 132, 138
 PnetHub<-, 47
 PnetHub<- (PnetHub), 64
 PNetica, 4
 PNetica-package, 149
 PnetMakeStubNodes, 44, 45, 47, 60, 65, 140
 PnetName, 6, 7, 44, 45, 47, 67, 75, 79, 80
 PnetName, NULL-method (PnetName), 67
 PnetName<-, 47
 PnetName<- (PnetName), 67
 PnetPathname, 6, 7, 44, 45, 47, 69, 80
 PnetPathname<-, 47
 PnetPathname<- (PnetPathname), 69
 PnetPnodes, 12, 27, 33–35, 44, 45, 70, 96
 PnetPnodes<- (PnetPnodes), 70
 PnetPriorWeight, 44, 45, 72
 PnetPriorWeight<- (PnetPriorWeight), 72
 PnetRemoveStubNodes, 44, 45, 47, 140
 PnetRemoveStubNodes
 (PnetMakeStubNodes), 65
 PnetSerialize, 47, 75, 81
 PnetTitle, 6, 7, 44, 45, 47, 68, 78, 80
 PnetTitle<-, 47
 PnetTitle<- (PnetTitle), 78
 PnetUnserialize, 47
 PnetUnserialize (PnetSerialize), 75
 PnetWarehouse, 75, 153
 PnetWarehouse (PnetWarehouse-class), 79
 PnetWarehouse-class, 79
 Pnode, 3, 8, 12–14, 18–22, 26, 31–37, 44, 45,
 48, 52, 53, 55, 59, 63, 65, 70–73, 82,
 87, 89, 91–93, 95–102, 104,
 106–112, 114, 116–120, 122, 123,
 125–128, 130, 131, 135, 136, 138,
 141–144, 146, 148, 150
 Pnode-class, 85
 PnodeAlphas, 13, 21, 41, 42, 50, 51, 53–56,
 83, 84, 88, 97, 118, 119, 121,
 136–138, 140
 PnodeAlphas (PnodeLnAlphas), 101
 PnodeAlphas<- (PnodeLnAlphas), 101
 PnodeBetas, 13, 21, 27, 35, 40–42, 49–51,
 53–56, 82–84, 87, 89, 91, 92, 97, 98,
 101, 103, 104, 112, 118, 121, 122,
 136–138, 140
 PnodeBetas<- (PnodeBetas), 87
 PnodeDefaultAlphas, 91
 PnodeDefaultBetas (PnodeDefaultAlphas),
 91
 PnodeDescription, 9, 11, 83, 84, 86, 132
 PnodeDescription (PnodeTitle), 130
 PnodeDescription<-, 86
 PnodeDescription<- (PnodeTitle), 130
 PnodeEAP, 10, 24, 61, 86, 132, 144–146
 PnodeEAP (PnodeStats), 128
 PnodeEvidence, 61, 86, 92
 PnodeEvidence<-, 86
 PnodeEvidence<- (PnodeEvidence), 92
 PnodeLabels, 9, 11, 83, 84, 86, 95, 132
 PnodeLabels<-, 86
 PnodeLabels<- (PnodeLabels), 95
 PnodeLink, 13, 18, 21, 36, 41, 42, 50, 51, 53,
 54, 56, 82–84, 88, 89, 96, 100,
 102–104, 112, 118, 119, 121, 122,
 136, 139–141
 PnodeLink<- (PnodeLink), 96
 PnodeLinkScale, 13, 41, 42, 50, 51, 53–56,
 82–84, 98, 99, 101, 136, 137, 139,
 140, 142
 PnodeLinkScale<- (PnodeLinkScale), 99
 PnodeLnAlphas, 13, 27, 35, 82–84, 89, 91, 92,
 98, 101, 101, 104, 112, 118, 119,
 121, 122
 PnodeLnAlphas<- (PnodeLnAlphas), 101
 PnodeMargin, 23, 24, 61, 86, 144–146
 PnodeMargin (PnodeStats), 128
 PnodeMedian, 61, 87, 144–146
 PnodeMedian (PnodeStats), 128
 PnodeMin, 105
 PnodeMode, 23, 24, 61, 144–146

- PnodeMode (PnodeStats), 128
 PnodeName, 9, 11, 50, 63, 83–85, 107, 123, 127, 131, 132
 PnodeName<-, 85
 PnodeName<- (PnodeName), 107
 PnodeNet, 9, 11, 63, 73, 83, 84, 132
 PnodeNet (PnetPnodes), 70
 PnodeNumParents, 83, 84, 86
 PnodeNumParents (PnodeParents), 109
 PnodeNumStates, 9, 11, 83, 84, 86, 93, 116, 132
 PnodeNumStates (PnodeStates), 122
 PnodeParentNames, 83, 84, 86, 116
 PnodeParentNames (PnodeParents), 109
 PnodeParents, 42, 50, 51, 56, 83, 84, 86, 109, 116, 140
 PnodeParents<-, 86
 PnodeParents<- (PnodeParents), 109
 PnodeParentStates (PnodeParents), 109
 PnodeParentTvals, 32, 83, 84, 88, 89, 97, 98, 101, 102, 104, 110, 111, 118–122, 126, 127
 PnodePostWeight, 113
 PnodePriorWeight, 13, 41, 42, 50, 51, 56, 82–84, 113, 114, 139, 140
 PnodePriorWeight (PnetPriorWeight), 72
 PnodePriorWeight<- (PnetPriorWeight), 72
 PnodeProbs, 13, 86, 114, 115
 PnodeProbs<-, 86
 PnodeProbs<- (PnodeProbs), 115
 PnodeQ, 13, 19, 21, 37, 53, 56, 82–84, 88, 89, 97, 98, 101–104, 111, 112, 117, 118, 120–122, 136, 140, 141
 PnodeQ<- (PnodeQ), 117
 PnodeRules, 13, 41, 42, 50, 51, 53, 54, 56, 82–84, 87–89, 97, 98, 101–104, 111, 112, 118, 119, 120, 121, 136, 137, 139, 140
 PnodeRules<- (PnodeRules), 120
 PnodeSD, 10, 61, 87, 132, 144–146
 PnodeSD (PnodeStats), 128
 PnodeStateBounds, 84, 86, 93, 94, 112, 116, 127
 PnodeStateBounds (isPnodeContinuous), 31
 PnodeStateBounds<-, 86
 PnodeStateBounds<- (isPnodeContinuous), 31
 PnodeStateDescriptions, 10, 11, 83, 84, 86, 123, 132
 PnodeStateDescriptions (PnodeStateTitles), 124
 PnodeStateDescriptions<-, 86
 PnodeStateDescriptions<- (PnodeStateTitles), 124
 PnodeStates, 10, 83, 84, 86, 94, 116, 122, 123, 125–127, 132
 PnodeStates<-, 86
 PnodeStates<- (PnodeStates), 122
 PnodeStateTitles, 10, 11, 83, 84, 86, 123, 124, 127, 132
 PnodeStateTitles<-, 86
 PnodeStateTitles<- (PnodeStateTitles), 124
 PnodeStateValues, 10, 11, 32, 83, 84, 86, 112, 123, 125, 126, 129, 132, 144
 PnodeStateValues<-, 86
 PnodeStateValues<- (PnodeStateValues), 126
 PnodeStats, 128
 PnodeTitle, 9, 11, 83–85, 107, 108, 130, 132
 PnodeTitle<-, 86
 PnodeTitle<- (PnodeTitle), 130
 PnodeWarehouse, 153
 PnodeWarehouse (PnodeWarehouse-class), 131
 PnodeWarehouse-class, 131
 Qmat2Pnet, 6, 8, 44, 45, 56, 60, 64, 66, 134, 151
 qnorm, 112
 recover, 12, 13, 34, 35, 39, 41
 RegressionGadget, 20, 22, 37, 141
 RNetica, 4, 149, 150
 runApp, 19, 22, 37, 142
 runGadget, 19, 21, 37, 141
 serialize, 75
 setIs, 46, 85
 show, Statistic-method (Statistic-class), 146
 Statistic, 128, 129, 143, 143, 144–146
 Statistic-class, 146
 StatName (Statistic-class), 146
 StatName, Statistic-method (Statistic-class), 146
 topsort, 147

try, [24](#), [25](#)

unserializePnet, [47](#)
unserializePnet (PnetSerialize), [75](#)

Warehouse, [6–8](#), [11](#), [39](#), [40](#), [42](#), [45](#), [53](#), [54](#), [56](#),
[77](#), [79](#), [81](#), [131](#), [134–136](#), [138](#), [140](#),
[148](#), [153–156](#)

WarehouseCopy, [150](#), [151](#), [152](#)
WarehouseCopy, default-method
(WarehouseCopy), [152](#)

WarehouseData, [9](#), [81](#), [133](#), [150](#), [151](#)
WarehouseData (WarehouseManifest), [154](#)

WarehouseFetch, [80](#), [133](#)
WarehouseFetch (Warehouse), [148](#)

WarehouseFree, [80](#), [133](#), [153](#)
WarehouseFree (Warehouse), [148](#)

WarehouseInventory (WarehouseManifest),
[154](#)

WarehouseMake, [80](#), [133](#)
WarehouseMake (Warehouse), [148](#)

WarehouseManifest, [7](#), [11](#), [79–81](#), [131](#), [133](#),
[134](#), [149–151](#), [154](#)

WarehouseManifest<-, [80](#), [133](#)
WarehouseManifest<-
(WarehouseManifest), [154](#)

WarehouseSave (Warehouse), [148](#)

WarehouseSupply, [80](#), [133](#)
WarehouseSupply (Warehouse), [148](#)

WarehouseSupply, ANY-method (Warehouse),
[148](#)

WarehouseUnpack, [81](#)
WarehouseUnpack (PnetSerialize), [75](#)

write.CaseFile, [17](#), [27](#)