Package 'PNetica'

June 2, 2021

Version 0.8-5

Date 2021/06/01

Title Parameterized Bayesian Networks Netica Interface

Author Russell Almond

Maintainer Russell Almond <ralmond@fsu.edu>

Depends R (>= 3.0), RNetica (>= 0.7), CPTtools (>= 0.5), Peanut (>= 0.8), futile.logger, methods

Description This package provides RNetica implementation of Peanut interface.

License Artistic-2.0

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

R topics documented:

`PNetica-package` *Parameterized Bayesian Networks Netica Interface*

Description

This package provides RNetica implementation of Peanut interface.

Details

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

The `Peanut` package provides a set of generic functions for manipulation parameterized networks, in particular, for the abstract `Pnet` and `Pnode` classes. This package provides concrete implementations of those classes using the built in classes of `RNetica`. In particular, `Pnet.NeticaBN` extends `NeticaBN` and `Pnode.NeticaNode` extends `NeticaNode`. The documentation object `Pnode.States` documents additional fields of this object.

The properties of the `Pnet` and `Pnode` objects are stored as serialized Netica user fields (see `NetworkUserObj` and `NodeUserObj`). The documentation object `Pnode.Properties` documents the methods.

The `as.Pnet` (`as.Pnode`) method for a `NeticaBN` (`NeticaNode`) merely adds "Pnet" ("Pnode") to `class(net)` (`class(node)`). All of the methods in the `PNetica` are defined for either the `NeticaBN` or `NeticaNode` object, so strictly speaking, adding the "Pnet" or "Pnode" class is not necessary, but it is recommended in case this is used in the future.

PNetica Specific Implementation Details

Here are some Netica specific details which may not be apparent from the description of the generic functions in the `Peanut` package.

1. The `cases` argument to `calcPnetLLike`, `calcExpTables` and `GEMfit` all expect the pathname of a Netica case file (see `write.CaseFile`).
2. The methods `calcPnetLLike`, `calcExpTables`, and therefore `GEMfit` when called with a `Pnet` as the first argument, expect that there exists a node set (see `NetworkNodesInSet`) called "onodes" corresponding to the observable variables in the case file `cases`.
3. The function `CompileNetwork` needs to be called before calls to `calcPnetLLike`, `calcExpTables` and `GEMfit`.
4. The method `PnetPnodes` stores its value in a nodeset called "pnodes". It is recommended that the accessor function be used for modifying this field.
5. The `PnetPriorWeight` field of the `Pnet.NeticaBN` object and all of the fields of the `Pnode.NeticaNode` are stored in serialized user fields with somewhat obvious names (see `NetworkUserObj` and `NodeUserObj`). These fields should not be used for other purposes.

Creating and Restoring Pnet.NeticaBN objects

As both the nodesets and and user fields are serialized when Netica serializes a network (`WriteNetworks`) the fields of the `Pnet.NeticaBN` and `Pnode.NeticaNode` objects should be properly saved and restored.

The first time the network and nodes are created, it is recommended that `Pnet` and `Pnode.NeticaNode` (or simply the generic functions `Pnet` and `Pnode`. Note that calling `Pnode` will calculate defaults for the `PnodeLnAlphas` and `PnodeBetas` based on the current value of `NodeParents(node)`, so this should be set before calling this function. (See examples).

Index

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

Legal Stuff

Netica and Norsys are registered trademarks of Norsys, LLC (`http://www.norsys.com/`), used by permission.

Extensive use of `PNetica` will require a Netica API license from Norsys. This is basically a requirement of the `RNetica` package, and details are described more fully there. Without a license, `RNetica` and `PNetica` will work in a student/demonstration mode which limits the size of the network.

Although Norsys is generally supportive of the RNetica project, it does not officially support RNetica, and all questions should be sent to the package maintainers.

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.

See Also

`PNetica` depends on the following other packages.

`RNetica` A binding of the Netica C API into R.

`Peanut` An the generic functions for which this package provides implementations.

`CPTtools` A collection of implementation independent Bayes net utilities.

Examples

```
sess <- NeticaSession()
startSession(sess)

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


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

partial3 <- NewDiscreteNode(tNet,"partial3",
                            c("FullCredit","PartialCredit","NoCredit"))
NodeParents(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(file.path(library(help="PNetica")$path,
                            "testnets","IRT10.2PL.base.dne"), 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]])

}


casepath <- file.path(library(help="PNetica")$path,
                            "testdat","IRT10.2PL.200.items.cas")
## Record which nodes in the casefile we should pay attention to
NetworkNodesInSet(irt10.base,"onodes") <-
   NetworkNodesInSet(irt10.base,"observables")


BuildAllTables(irt10.base)
CompileNetwork(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)
```

BNWarehouse — *Constructor for the* BNWarehosue *class.*

Description

This is the constructor for the BNWarehouse class. This produces NeticaBN objects, which are instances of the Pnet abstract class.

Usage

```
BNWarehouse(manifest = data.frame(), session = getDefaultSession(), address = ".", key = c("Name"), pre
```

Arguments

`manifest`	A data frame containing instructions for building the nets. See BuildNetManifest.
`session`	A link to a NeticaSession object for managing the nets.

`address`	A character scalar giving the path in which the ".dne" files containing the networks are stored.
`key`	A character scalar giving the name of the column in the manifest which contains the network name.
`prefix`	A character scaler used in front of numeric names to make legal Netica names. (See `as.IDname`).

Value

An object of class `BNWarehouse`.

Author(s)

Russell Almond

See Also

`Warehouse` for the general warehouse protocol.

Examples

```
sess <- NeticaSession()
startSession(sess)

### This tests the manifest and factory protocols.

nodeman1 <- read.csv(paste(library(help="Peanut")$path, "auxdata",
                           "Mini-PP-Nodes.csv", sep=.Platform$file.sep),
                     row.names=1,stringsAsFactors=FALSE)

netman1 <- read.csv(paste(library(help="Peanut")$path, "auxdata",
                          "Mini-PP-Nets.csv", sep=.Platform$file.sep),
                    row.names=1, stringsAsFactors=FALSE)


### Test Net building
Nethouse <- BNWarehouse(manifest=netman1,session=sess,key="Name")
stopifnot(is.PnetWarehouse(Nethouse))

setwd(paste(library(help="PNetica")$path, "testnets",sep=.Platform$file.sep))
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))
```

BNWarehouse-class *Class* "BNWarehouse"

Description

A Warehouse (specifically a PnetWarehouse) object which holds and builds NeticaBN 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 BNWarehouse either supplies prebuilt (i.e., already in the Netica session) 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 in the session.
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 netowrk. 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

Objects can be created by calls of the form BNWarehouse(...).

This class is a subclass of PnetWarehouse in the Peanut-package.

This is a reference object and typically there is only one instance per project.

Methods

WarehouseSupply `signature(warehouse = "BNWarehouse",name = "character")`. This finds a network with the appropriate name in the session. 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 properities specified in the manifest is created.

WarehouseFetch `signature(warehouse = "BNWarehouse",name = "character")`. This fetches the network with the given name from the session object, or returns `NULL` if it has not been built in Netica yet.

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

WarehouseFree `signature(warehouse = "BNWarehouse",name = "character")`. This removes the network from the warehouse inventory. *Warning*: This deletes the network.

ClearWarehouse `signature(warehouse = "BNWarehouse")`. This removes all networks from the warehouse inventory. *Warning*: This deletes all the networks.

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

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

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

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

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

as.legal.name `signature(warehouse = "BNWarehouse")`: If necessary, mangles a node name to follow the Netica IDname conventions.

is.legal.name `signature(warehouse = "BNWarehouse")`: Checks to see if a node name follows the Netica IDname conventions.

WarehouseCopy `signature(warehouse = "BNWarehouse",obj = "NeticaBN")`: Makes a copy of a network.

is.valid `signature(warehouse = "BNWarehouse")`: Checks an object to see if it is a valid Netica Network.

WarehouseSave `signature(warehouse = "NNWarehouse",obj = "NeticaBN")`: Saves the network to the pathname in the `PnetPathname` property.

WarehouseSave `signature(warehouse = "NNWarehouse",obj = "character")`: Saves the network with the given name.

Slots

`manifest`**:** A `data.frame` which consists of the manifest. (see details).

`session`**:** Object of class `NeticaSession`. This is the session in which the nets are created.

`address`**:** Object of class "`character`" which gives the path to the directory in which written descriptions of the nets are stored.

`key`**:** Object of class "`character`" giving the name of the column which has the key for the manifest. This is usually "`Name`".

`prefix`**:** Object of class "`character`" giving a short string to insert in front of numeric names to make legal Netica names (see `as.IDname`).

Extends

Class "`PnetWarehouse`", directly.

Note

The `BNWarehouse` implementatation 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 pathename, a new blank network is built and the appropriate fields are set from the metadata.

Author(s)

Russell Almond

References

The following is a Google sheet where an example network manifest can be found on the `nets` tab. `https://docs.google.com/spreadsheets/d/1SiHQTLBNHQ-FUPnNzf9jPm9ifUG-c8f_6ljOrEcdl9M/`

See Also

In Peanut Package: `Warehouse`, `WarehouseManifest`, `BuildNetManifest`

Implementation in the `PNetica` package: `BNWarehouse`, `MakePnet.NeticaBN`

Examples

```
sess <- NeticaSession()
startSession(sess)

## BNWarehouse is the PNetica Net Warehouse.
## This provides an example network manifest.
netman1 <- read.csv(paste(library(help="Peanut")$path, "auxdata",
                          "Mini-PP-Nets.csv", sep=.Platform$file.sep),
                    row.names=1, stringsAsFactors=FALSE)
Nethouse <- BNWarehouse(manifest=netman1,session=sess,key="Name")

## is.PnetWarehouse -- tests for PnetWarehouse.
stopifnot(is.PnetWarehouse(Nethouse))
```

```
## WarehouseManifest
stopifnot(all.equal(WarehouseManifest(Nethouse),netman1))

## WarehouseData
stopifnot(all.equal(WarehouseData(Nethouse,"miniPP_CM")[-4],
   netman1["miniPP_CM",-4]),
   ## Pathname has leading address prefix instered.
   basename(WarehouseData(Nethouse,"miniPP_CM")$Pathname) ==
   basename(netman1["miniPP_CM","Pathname"]))

## WarehouseManifest<-
netman2 <- netman1
netman2["miniPP_CM","Pathname"] <- "mini_CM.dne"
WarehouseManifest(Nethouse) <- netman2

stopifnot(all.equal(WarehouseData(Nethouse,"miniPP_CM")[,-4],
   netman2["miniPP_CM",-4]),
   basename(WarehouseData(Nethouse,"miniPP_CM")$Pathname) ==
   basename(netman2["miniPP_CM","Pathname"]))
WarehouseManifest(Nethouse) <- netman1

## Usually way to access nets is through warehouse supply
CM <- WarehouseSupply(Nethouse, "miniPP_CM")
EM <- WarehouseSupply(Nethouse, "PPcompEM")
stopifnot(is.active(CM),is.active(EM))

## WarehouseFetch -- Returns NULL if does not exist
stopifnot(is.null(WarehouseFetch(Nethouse,"PPconjEM")))

## WarehouseMake -- Make the net anew.
EM1 <- WarehouseMake(Nethouse,"PPconjEM")
EM1a <- WarehouseFetch(Nethouse,"PPconjEM")
stopifnot(PnetName(EM1)==PnetName(EM1a))

## WarehouseFree -- Deletes the Net
WarehouseFree(Nethouse,"PPconjEM")
stopifnot(!is.active(EM1))

## ClearWarehouse -- Deletes all nets
ClearWarehouse(Nethouse)
stopifnot(!is.active(EM),!is.active(CM))

stopSession(sess)
```

BuildTable.NeticaNode *Builds the conditional probability table for a Pnode*

Description

The function `BuildTable` calls `calcDPCFrame` to calculate the conditional probability for a `Pnode` object, and sets the current conditional probability table of `node` to the resulting value. It also sets the `NodeExperience(node)` to the current value of `GetPriorWeight(node)`.

Usage

```
## S4 method for signature 'NeticaNode'
BuildTable(node)
```

Arguments

`node`	A `Pnode` and `NeticaNode` object whose table is to be built.

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 using the `[]` (Extract.NeticaNode) operator.

In addition to setting the CPT, the weight given to the nodes in the EM algorithm are set to `GetPriorWeight(node)`, which will extract the value of `PnodePriorWeight(node)` or if that is null, the value of `PnetPriorWeight(NodeParents(node))` and set `NodeExperience(node)` to the resulting value.

Value

The `node` argument is returned invisibly. As a side effect the conditional probability table and experience of `node` is modified.

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.NeticaNode`, `Pnode`, `PnodeQ`, `PnodePriorWeight`, `PnodeRules`, `PnodeLink`, `PnodeLnAlphas`, `PnodeAlphas`, `PnodeBetas`, `PnodeLinkScale`,`GetPriorWeight`, `calcDPCTable`, `NodeExperience(node)`, `Extract.NeticaNode` (`[`)

Examples

```
sess <- NeticaSession()
startSession(sess)

## Network with two proficiency variables and observables for each
## different type of rule

binAll <- CreateNetwork("binAll", session=sess)
PnetPriorWeight(binAll) <- 11           #Give it something to see.

## Set up Proficiency Model.
thetas <- NewDiscreteNode(binAll,paste("theta",0:1,sep=""),
                          c("Low","Med","High")) # Create the variable with 3 levels
names(thetas) <- paste("theta",0:1,sep="")
NodeParents(thetas[[2]]) <- thetas[1]

for (nd in thetas) {
  NodeLevels(nd) <- effectiveThetas(NodeNumStates(nd))
  PnodeRules(nd) <- "Compensatory"
  PnodeLink(nd) <- "normalLink"
  PnodeBetas(nd) <- 0 # A numeric vector of intercept parameters
  PnodeQ(nd) <- TRUE # All parents are relevant.
 NodeSets(nd) <- c("pnodes","Proficiency") # A character vector containing the names of the node sets
}

## Standard normal prior.
PnodeAlphas(thetas[[1]]) <- numeric() # A numeric vector of (log) slope parameters
PnodeLinkScale(thetas[[1]]) <- 1 # A positive numeric value, or NULL
                                 # if the scale parameter is not used
                                 # for the link function.
## Regression with a correlation of .6
PnodeAlphas(thetas[[2]]) <- .6
PnodeLinkScale(thetas[[2]]) <- .8

BuildTable(thetas[[1]])
BuildAllTables(binAll)

DeleteNetwork(binAll)
stopSession(sess)
```

`calcExpTables.NeticaBN` *Calculate expected tables for a Pnet.NeticaBN*

Description

The performs the E-step of the GEM algorithm by running the Netica EM algorithm (see `LearnCPTs`) using the data in cases. After this is run, the conditional probability table for each `Pnode.NeticaNode` should be the mean of the Dirichlet distribution and the scale parameter should be the value of `NodeExperience(node)`.

Usage

```
## S4 method for signature 'NeticaBN'
calcExpTables(net, cases, Estepit = 1,
                            tol = sqrt(.Machine$double.eps))
```

Arguments

`net`	A `Pnet.NeticaBN` object representing a parameterized network.
`cases`	A character scalar giving the file name of a Netica case file (see `write.CaseFile`).
`Estepit`	An integer scalar describing the number of steps the Netica should take in the internal EM algorithm.
`tol`	A numeric scalar giving the stopping tolerance for the internal Netica EM algorithm.

Details

The key to this method is realizing that the EM algorithm built into the Netica (see `LearnCPTs`) can perform the E-step of the outer `GEMfit` generalized EM algorithm. It does this in every iteration of the algorithm, so one can stop after the first iteration of the internal EM algorithm.

This method expects the `cases` argument to be a pathname pointing to a Netica cases file containing the training or test data (see `write.CaseFile`). Also, it expects that there is a nodeset (see `NetworkNodesInSet`) attached to the network called "onodes" which references the observable variables in the case file.

Before calling this method, the function `BuildTable` needs to be called on each `Pnode` to both ensure that the conditional probability table is at a value reflecting the current parameters and to reset the value of `NodeExperience(node)` to the starting value of `GetPriorWeight(node)`.

Note that Netica does allow `NodeExperience(node)` to have a different value for each row the the conditional probability table. However, in this case, each node must have its own prior weight (or exactly the same number of parents). 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 `LearnCPTs`. 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.

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, Pnet.NeticaBN, GEMfit, calcPnetLLike, maxAllTableParams, calcExpTables, NetworkNodesInSet write.CaseFile, LearnCPTs

Examples

```
sess <- NeticaSession()
startSession(sess)

irt10.base <- ReadNetworks(file.path(library(help="PNetica")$path,
                           "testnets","IRT10.2PL.base.dne"), 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]])
}
CompileNetwork(irt10.base) ## Netica requirement

casepath <- file.path(library(help="PNetica")$path,
                           "testdat","IRT10.2PL.200.items.cas")
## Record which nodes in the casefile we should pay attention to
NetworkNodesInSet(irt10.base,"onodes") <-
   NetworkNodesInSet(irt10.base,"observables")

item1 <- irt10.items[[1]]

priorcounts <- sweep(NodeProbs(item1),1,NodeExperience(item1),"*")

calcExpTables(irt10.base,casepath)

postcounts <- sweep(NodeProbs(item1),1,NodeExperience(item1),"*")

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

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

`calcPnetLLike.NeticaBN` *Calculates the log likelihood for a set of data under a Pnet.NeticaBN model*

Description

The method `calcPnetLLike.NeticaBN` calculates the log likelihood for a set of data contained in `cases` using the current conditional probability tables in a `Pnet.NeticaBN`. Here `cases` should be the filename of a Netica case file (see `write.CaseFile`).

Usage

```
## S4 method for signature 'NeticaBN'
calcPnetLLike(net, cases)
```

Arguments

`net`	A `Pnet.NeticaBN` object representing a parameterized network.
`cases`	A character scalar giving the file name of a Netica case file (see `write.CaseFile`).

Details

This function provides the convergence test for the `GEMfit` algorithm. The `Pnet.NeticaBN` 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 method expects the `cases` argument to be a pathname pointing to a Netica cases file containing the training or test data (see `write.CaseFile`). Also, it expects that there is a nodeset (see `NetworkNodesInSet`) attached to the network called “onodes” which references the observable variables in the case file.

As Netica does not have an API function to directly calculate the log-likelihood of a set of cases, this method loops through the cases in the case set and calls `FindingsProbability(net)` for each one. Note that if there are frequencies in the case file, each case is weighted by its frequency.

Value

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

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, Pnet.NeticaBN, GEMfit, calcExpTables, BuildAllTables, maxAllTableParams NetworkNodesInSet, FindingsProbability, write.CaseFile

Examples

```
sess <- NeticaSession()
startSession(sess)

irt10.base <- ReadNetworks(file.path(library(help="PNetica")$path,
                           "testnets","IRT10.2PL.base.dne"), 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]])
}
CompileNetwork(irt10.base) ## Netica requirement

casepath <- file.path(library(help="PNetica")$path,
                           "testdat","IRT10.2PL.200.items.cas")
## Record which nodes in the casefile we should pay attention to
NetworkNodesInSet(irt10.base,"onodes") <-
   NetworkNodesInSet(irt10.base,"observables")

llike <- calcPnetLLike(irt10.base,casepath)

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

MakePnet.NeticaBN *Creates a NeticaBN object which is also a Pnet*

Description

This does the actual work of making a Pnet from the manifest description. It is typically called from WarehouseMake.

Usage

```
MakePnet.NeticaBN(sess, name, data)
```

Arguments

`sess`	The Netica session (`NeticaSession`) object in which the net will be created.
`name`	A character scalar with the name of the network. This should follow the `IDname` rules.
`data`	A list providing data and metadata about the network. See details.

Details

This is a key piece of the `Warehouse` infrastructure. The idea is that a network can be constructed given a session, a name, and a collection of metadata. The metadata can be stored in a table which is the the manifest of the warehouse.

The current system expects the following fields in the `data` argument.

Hub For a network which represents an evidence model (spoke), this is the name of the network to which it should be attached (the *hub*).

Title This is a longer unconstrained name for the network.

Pathname This is the location in which the `.neta` or `.dne` file which stores the network.

Description This is a longer string describing the network.

These correspond to fields in the `RNetica{NeticaBN}` object.

Value

An object of class `NeticaBN` which is also in the `Pnet` abtract class.

Names and Truenames

The truename system is designed to implement the name restrictions inherent in Netica (see IDname) without imposing the same limits on the Peanut framework. This is done by adding a `Truename` field to the net object and then mangling the actual name to follow the Netica rules using the `as.IDname` function.

The object should be available from the warehouse via its truename, but it is best to stick to the Netica naming conventions for networks and nodes.

Author(s)

Russell Almond

See Also

RNetica Package: `CreateNetwork`, `NeticaBN`, `IDname`

Peanut Package: `Warehouse`, `WarehouseMake`

PNetica Pacakge `BNWarehouse`

Examples

```
sess <- NeticaSession()
startSession(sess)

anet <- MakePnet.NeticaBN(sess,"Anet",
                          list(Title="A Network",Hub="",
                               Description="A Sample Network."))

DeleteNetwork(anet)

netman1 <- read.csv(paste(library(help="Peanut")$path, "auxdata",
                          "Mini-PP-Nets.csv", sep=.Platform$file.sep),
                    row.names=1, stringsAsFactors=FALSE)
## Build the first network (proficiency model)
miniPP <- MakePnet.NeticaBN(sess,"miniPP",netman1[1,,drop=FALSE])

DeleteNetwork(miniPP)
stopSession(sess)
```

MakePnode.NeticaNode *Makes a Pnode which is also a Netica Node*

Description

This does the actual work of making a node from a warehose manifest. It is typically called from WarehouseMake.

Usage

```
MakePnode.NeticaNode(net, name, data)
```

Arguments

net	A NeticaBN object in which the node will be created.
name	The name of the node. Ideally, this should follow the Netica IDname rules.
data	A data.frame with one for each state of contains data and meta-data about the node and states (See details).

Details

This is a key piece of the Warehouse infrastructure. If a node of the designated name does not exist, it will be created. If it does exist, the metadata fields of the node will be adjusted to match the fields in the data object.

Some of the fields of the data object apply to the whole node. In these fields, the value in the first row is used and the rest are ignored.

NStates A integer giving the number of states for a discrete variable or the discritzation of a continuous one. The number of rows of the data frame should match this.

Continuous A logical value telling whether or not the node should be regarded as continuous.

NodeTitle This is a longer unconstrained name for the node.

NodeDescription This is a longer string describing the node.

NodeLabels This is a comma separated list of tags identifying sets to which the node belongs. See `PnodeLabels`.

These fields are repeated for each of the states in the node, as they are different for each state.

StateName The name of the state, this should follow the Netica IDname conventions.

StateTitle This is a longer unconstrained name for the state.

StateDescription This is a longer string describing the state.

Additionally, the following field is used only for discrete nodes:

StateValue This is a numeric value assigned to the state. This value is used when calculating the node expected value.

The StateValue plays two important roles. First, when used with the `PnodeEAP` and `PnodeSD` functions, it is the value assigned to the node. Second, when constructing CPTs using the DiBello framework, it is used at the effective thetas. See `PnodeParentTvals` and `PnodeStateValues`

Continuous nodes in Netica are handled by breaking the interval up into pieces. This is the function `PnodeStateBounds`. Note that the bounds should be either monotonically increasing or decreasing and that the lower bound for one category should match lower bound for the next to within a tolerance of .002. The values `Inf` and `-Inf` can be used where appropriate.

LowerBound This is a numeric value giving the lower bound for the range for the discritization of the node.

UpperBound This is a numeric value giving the upper bound for the range for the

Value

An object of class `NeticaNode` which is also in the `Pnode` abtract class.

Names and Truenames

The truename system is designed to implement the name restrictions inherent in Netica (see IDname) without imposing the same limits on the Peanut framework. This is done by adding a `Truename` field to the net object and then mangling the actual name to follow the Netica rules using the `as.IDname` function.

The object should be available from the warehouse via its truename, but it is best to stick to the Netica naming conventions for networks and nodes.

Note that the truename convention is used for node names, but not for state names, which are restricted to Netica conventions.

Author(s)

Russell Almond

See Also

RNetica Package: NeticaNode, NewContinuousNode, NewDiscreteNode, IDname

Peanut Package: Warehouse, WarehouseMake

PNetica Pacakge PnodeWarehouse

Examples

```
sess <- NeticaSession()
startSession(sess)

### This tests the manifest and factory protocols.

netman1 <- read.csv(paste(library(help="Peanut")$path, "auxdata",
                          "Mini-PP-Nets.csv", sep=.Platform$file.sep),
                    row.names=1, stringsAsFactors=FALSE)
## Build the first network (proficiency model)
miniPP <- MakePnet.NeticaBN(sess,"miniPP",netman1[1,,drop=FALSE])

nodeman1 <- read.csv(paste(library(help="Peanut")$path, "auxdata",
                           "Mini-PP-Nodes.csv", sep=.Platform$file.sep),
                     row.names=1,stringsAsFactors=FALSE)

## Discrete Example
phys.dat <- nodeman1[nodeman1$NodeName=="Physics",]

Physics <- MakePnode.NeticaNode(miniPP,"Physics",phys.dat)

## Continuous Example
dur.dat <- nodeman1[nodeman1$NodeName=="Duration",]

Duration <- MakePnode.NeticaNode(miniPP,"Duration",dur.dat)


DeleteNetwork(miniPP)
stopSession(sess)
```

---

maxCPTParam.NeticaNode

Find optimal parameters of a Pnode.NeticaNode to match expected tables

---

Description

These function assumes that an expected count contingency table can be built from the network; i.e., that LearnCPTs has been recently called. They then try to find the set of parameters maximizes the probability of the expected contingency table with repeated calls to mapDPC. This describes the method for maxCPTParam when the Pnode is a NeticaNode.

Usage

```
## S4 method for signature 'NeticaNode'
maxCPTParam(node, Mstepit = 5, tol = sqrt(.Machine$double.eps))
```

Arguments

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.

Details

This method is called on on a Pnode.NeticaNode object during the M-step of the EM algorithm (see GEMfit and maxAllTableParams for details). Its purpose is to extract the expected contingency table from Netica and pass it along to mapDPC.

When doing EM learning with Netica, the resulting conditional probability table (CPT) is the mean of the Dirichlet posterior. Going from the mean to the parameter requires multiplying the CPT by row counts for the number of virtual observations. In Netica, these are call NodeExperience. Thus, the expected counts are calculated with this expression: `sweep(node[[]],1,NodeExperience(node),"*")`.

What remains is to take the table of expected counts and feed it into mapDPC and then take the output of that routine and update the parameters.

The parameters `Mstepit` and `tol` are passed to mapDPC to control the gradient decent 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. 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.

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, Pnode.NeticaNode, GEMfit, maxAllTableParams mapDPC

Examples

```
## This method is mostly a wrapper for CPTtools::mapDPC
getMethod(maxCPTParam,"NeticaNode")


sess <- NeticaSession()
startSession(sess)

irt10.base <- ReadNetworks(paste(library(help="PNetica")$path,
                           "testnets","IRT10.2PL.base.dne",
                           sep=.Platform$file.sep),
                           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 <- paste(library(help="PNetica")$path,
                           "testdat","IRT10.2PL.200.items.cas",
                           sep=.Platform$file.sep)


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)
```

NNWarehouse — *Constructor for the* `NNWarehosue` *class.*

Description

This is the constructor for the `NNWarehouse` class. This produces `NeticaNode` objects, which are instances of the `Pnode` abstract class.

Usage

```
NNWarehouse(manifest = data.frame(), session = getDefaultSession(),
             key = c("Model","NodeName"), prefix = "V")
```

Arguments

`manifest`	A data frame containing instructions for building the nodes. See `BuildNodeManifest`.
`session`	A link to a `NeticaSession` object for managing the nets.
`key`	A character vector giving the name of the column in the manifest which contains the network name and the node name.
`prefix`	A character scaler used in front of numeric names to make legal Netica names. (See `as.IDname`).

Details

Each network defines its own namespace for nodes, so the key to the node manifest is a pair (*Model*,*NodeName*) where *Model* is the name of the net and `NodeName` is the name of the node.

Value

An object of class `NNWarehouse`.

Author(s)

Russell Almond

See Also

`Warehouse` for the general warehouse protocol.

Examples

```
sess <- NeticaSession()
startSession(sess)

### This tests the manifest and factory protocols.

nodeman1 <- read.csv(paste(library(help="Peanut")$path, "auxdata",
                           "Mini-PP-Nodes.csv", sep=.Platform$file.sep),
                     row.names=1,stringsAsFactors=FALSE)

netman1 <- read.csv(paste(library(help="Peanut")$path, "auxdata",
                          "Mini-PP-Nets.csv", sep=.Platform$file.sep),
                    row.names=1, stringsAsFactors=FALSE)


### Test Net building
Nethouse <- BNWarehouse(manifest=netman1,session=sess,key="Name")
stopifnot(is.PnetWarehouse(Nethouse))

setwd(paste(library(help="PNetica")$path, "testnets",sep=.Platform$file.sep))
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))
```

NNWarehouse-class *Class* "NNWarehouse"

Description

This is a container for node objects, which are instances of the `Pnode` class. If a requested node is not already built, it can be built from the description found in the warehouse. In implements the `Warehouse` protocol.

Details

The `NNWarehouse` generally works with a paired `BNWarehouse` which supplies the network. It assumes that the referenced network already exists or has been loaded from a file. If the node already exists in the network, it simply returns it. If not, it creates it 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 pair of columns (“Model”, “NodeName”), with each pair identifying a set of rows correpsonding to the possible states of the node. The *name* argument to `WarehouseData` should be a character vector of length 2 with the first component corresonding to the network name and the second to the node name; it will return a `data.frame` with multiple rows.

Some of the fields of the manifest data apply to the whole node. In these fields, the value in the first row is used and the rest are ignored.

NStates A integer giving the number of states for a discrete variable or the discritzation of a continuous one. The number of rows of the manifest data for this node should match this.

Continuous A logical value telling whether or not the node should be regarded as continuous.

NodeTitle This is a longer unconstrained name for the node.

NodeDescription This is a longer string describing the node.

NodeLabels This is a comma separated list of tags identifying sets to which the node belongs. See `PnodeLabels`.

These fields are repeated for each of the states in the node, as they are different for each state. The “StateName” field is required and must be unique for each row.

StateName The name of the state, this should follow the Netica IDname conventions.

StateTitle This is a longer unconstrained name for the state.

StateDescription This is a longer string describing the state.

Additionally, the following field is used only for discrete nodes:

StateValue This is a numeric value assigned to the state. This value is used when calculating the node expected value.

The StateValue plays two important roles. First, when used with the `PnodeEAP` and `PnodeSD` functions, it is the value assigned to the node. Second, when constructing CPTs using the DiBello framework, it is used at the effective thetas. See `PnodeParentTvals` and `PnodeStateValues`

Continuous nodes in Netica are handled by breaking the interval up into pieces. This is the function `PnodeStateBounds`. Note that the bounds should be either monotonically increasing or decreasing and that the lower bound for one category should match lower bound for the next to within a tolerance of .002. The values `Inf` and `-Inf` can be used where appropriate.

LowerBound This is a numeric value giving the lower bound for the range for the discritization of the node.

UpperBound This is a numeric value giving the upper bound for the range for the

Objects from the Class

Objects can be using the constructor `NNWarehouse`.

This class is a subclass of `PnodeWarehouse` in the `Peanut-package`.

This is a reference object and typically there is only one instance per project.

Slots

`manifest`: A data frame that gives details of how to build the nodes.

`session`: Object of class `NeticaSession`, which is a pointer back to the Netica user space.

`key`: A character vector of length two, which gives the name of the fields in the manifest which which identify the network and variable names.

`prefix`: Object of class `"character"` which is used as a prefix if the name needs to be mangled to fit Netica IDname conventions.

Extends

Class `"PnodeWarehouse"`, directly.

Methods

For all of these methods, the `name` argument is expected to be a vector of length 2 with the first component specifying the network and the second the node.

WarehouseSupply `signature(warehouse = "NNWarehouse",name = "character")`. In this case the name is expected to be a vector of length 2 with the first component identifying the network and the second the node within the network. This finds a node with the appropriate name in the referenced network. If one does not exist, it is created with the properities specified in the manifest.

WarehouseFetch `signature(warehouse = "NNWarehouse",name="character")`: Fetches the node if it already exists, or returns `NULL` if it does not.

WarehouseMake `signature(warehouse = "NNWarehouse")`: Makes a new node, calling `MakePnode.NeticaNode`.

as.legal.name `signature(warehouse = "NNWarehouse")`: If necessary, mangles a node name to follow the Netica IDname conventions.

ClearWarehouse `signature(warehouse = "NNWarehouse")`: Removes prebuilt objects from the warehouse.

is.legal.name `signature(warehouse = "NNWarehouse")`: Checks to see if a node name follows the Netica IDname conventions.

is.PnodeWarehouse `signature(obj = "NNWarehouse")`: Returns true.

is.valid `signature(warehouse = "NNWarehouse")`: Checks an object to see if it is a valid Netica Node.

WarehouseCopy `signature(warehouse = "NNWarehouse",obj = "NeticaNode")`: Makes a copy of a node.

WarehouseData `signature(warehouse = "NNWarehouse")`: Returns the hunk of manifest for a single node.

WarehouseFree `signature(warehouse = "NNWarehouse")`: Deletes the node.

WarehouseInventory `signature(warehouse = "NNWarehouse")`: Returns a list of all nodes which have already been built.

WarehouseManifest `signature(warehouse = "NNWarehouse")`: Returns the current warehous manifest

WarehouseManifest<- `signature(warehouse = "NNWarehouse",value = "data.frame")`: sets the manifest

WarehouseSave `signature(warehouse = "NNWarehouse",obj = "ANY")`: Does nothing. Saving is done at the netowrk level.

Extends

Class `"PnodeWarehouse"`, directly.

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

References

The following is a Google sheet where an example node manifest can be found on the nodes tab. `https://docs.google.com/spreadsheets/d/1SiHQTLBNHQ-FUPnNzf9jPm9ifUG-c8f_6ljOrEcdl9M/`

See Also

In Peanut Package: `Warehouse`, `WarehouseManifest`, `BuildNodeManifest`

Implementation in the `PNetica` package: `NNWarehouse`, `MakePnode.NeticaNode`

Examples

```
sess <- NeticaSession()
startSession(sess)

## BNWarehouse is the PNetica Net Warehouse.
## This provides an example network manifest.
netman1 <- read.csv(paste(library(help="Peanut")$path, "auxdata",
                          "Mini-PP-Nets.csv", sep=.Platform$file.sep),
                    row.names=1, stringsAsFactors=FALSE)
Nethouse <- BNWarehouse(manifest=netman1,session=sess,key="Name")

nodeman1 <- read.csv(paste(library(help="Peanut")$path, "auxdata",
                           "Mini-PP-Nodes.csv", sep=.Platform$file.sep),
                     row.names=1,stringsAsFactors=FALSE)

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"))

WarehouseFree(Nethouse,"miniPP_CM")
WarehouseFree(Nethouse,"PPdurAttEM")
stopSession(sess)
```

Pnet.NeticaBN   *Class* "NeticaBN" *as a* "Pnet"

Description

The PNetica package supplies the needed methods so that the RNetica::NeticaBN object is an instance of the Peanut::Pnet object.

Extends

See NeticaBN for a description of the Netica class.

With these methods, NeticaBN now extends Pnet.

All reference classes extend and inherit methods from "envRefClass".

Methods

PnetCompile `signature(net = "NeticaBN")`: Compiles the network.

PnetName `signature(net = NeticaBN)`: Gets the name of the network.

PnetName<- `signature(net = NeticaBN)`: Sets the name of the network.

PnetTitle `signature(net = NeticaBN)`: Gets the title of the network.

PnetTitle<- `signature(net = NeticaBN)`: Sets the title of the network.

PnetDescription `signature(net = NeticaBN)`: Gets the description of the network.

PnetDescription<- `(signature(net = NeticaBN)`: Sets the description of the network.

PnetPathname `signature(net = NeticaBN)`: Gets the pathname where the network is stored.

PnetPathname<- `signature(net = NeticaBN)`: Sets the pathname where the network is stored.

PnetHub `signature(net = NeticaBN)`: Returns the name of the hub (competency/proficiency model) associated with an spoke (evidence model) network.

PnetHub<- `signature(net = NeticaBN)`: Sets the name of the hub.

PnetPriorWeight `signature(net = NeticaNode)`: Returns the default prior weight associated with nodes in this network.

PnetPriorWeight<- `signature(net = NeticaNode)`: Sets the default prior weight associated with nodes in this network.

as.Pnet `signature(x = NeticaBN)`: Forces x to be a `Pnet`.

is.Pnet `signature(x = NeticaBN)`: Returns true.

Author(s)

Russell Almond

See Also

Base class: `NeticaBN`.

Mixin class: `Pnet`.

Methods (from Peanut package.):

`PnetCompile`, `PnetHub`, `PnetName`, `PnetTitle`, `PnetDescription`, `PnetPathname`, `as.Pnet`, `is.Pnet`.

Examples

```
sess <- NeticaSession()
startSession(sess)
curd <- getwd()
setwd(file.path(library(help="PNetica")$path, "testnets"))

## PnetHub
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")

## PnetCompile
PnetCompile(PM)
marginPhysics <- Statistic("PnodeMargin","Physics","Pr(Physics)")
calcStat(marginPhysics,PM)

net <- CreateNetwork("funNet",sess)
stopifnot(PnetName(net)=="funNet")

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

## PnetPathname
stopifnot(PnetPathname(PM)=="miniPP-CM.dne")
PnetPathname(PM) <- "StudentModel1.dne"
stopifnot(PnetPathname(PM)=="StudentModel1.dne")

##PnetTitle and PnetDescirption
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(list(PM,EM1,foo,net,firstNet))
stopSession(sess)
setwd(curd)
```

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

```
## S4 method for signature 'NeticaBN'
PnetAdjoin(hub, spoke)
## S4 method for signature 'NeticaBN'
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.

Known Bugs

Netica version 5.04 has a bug that when nodes with no graphical information (e.g., position) are absorbed in a net in which some of the nodes have graphical information, it will generate an error. This was found and fixed in version 6.07 (beta) of the API. However, the function `PnetDetach` may generate internal Netica errors in this condition.

Right now they are logged, but nothing is done. Hopefully, they are harmless.

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 varation 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), *Bayeisan 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

```
sess <- NeticaSession()
startSession(sess)

PM <- ReadNetworks(file.path(library(help="PNetica")$path, "testnets",
     "miniPP-CM.dne"), session=sess)
EM1 <- ReadNetworks(file.path(library(help="PNetica")$path, "testnets",
      "PPcompEM.dne"), 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)
```

`PnetFindNode` *Finds nodes in a Netica Pnet.*

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`.

The function `PnetPnodes` returns nodes which have been marked as pnodes, that is nodes that have "pnodes" in their `PnodeLabels`.

Usage

```
## S4 method for signature 'NeticaBN'
PnetFindNode(net, name)
## S4 method for signature 'NeticaBN'
PnetPnodes(net)
## S4 replacement method for signature 'NeticaBN'
PnetPnodes(net) <- value
```

Arguments

`net`	The `Pnet` to search.
`name`	A character vector giving the name or names of the desired nodes. Names must follow the `IDname` protocol.
`value`	A list of `NeticaNode` objects in the network to be marked as Pnodes.

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 `NodeName()`).

A `NeticaNode` is also a `Pnode` if it has the label (node set) "pnodes".

The function `PnetPnodes()` returns all the Pnodes in the network, however, the order of the nodes in the network could be different in different calls to this function.

The form `PnetPnodes(net)<-value` sets the list of nodes in `value` to be the set of Pnodes; removing nodes which are not in the `value` from the set of Pndoes.

The Pnodes are not necesarily all of the nodes in the Netica network. The complete list of ndoes can be found through the `RNetica::NetworkAllNodes` function.

Value

The `Pnode` object or list of `Pnode` objects corresponding to `names`, or a list of all node objects for `PnetPnodes()`. In the latter case, the names will be set to the node names.

Note

`NeticaNode` objects do not survive the life of a Netica session (or by implication an R session). So the safest way to "save" a `NeticaNode` object is to recreate it using `PnetFindNode()` after the network is reloaded.

Author(s)

Russell Almond

References

`http://norsys.com/onLineAPIManual/index.html`, GetNodeNamed_bn(), GetNetNodes_bn()

See Also

Generic functions: `PnetPnodes()`, `PnetFindNode()`,

Related functions in RNetica package: `NetworkFindNode`, `NetworkAllNodes`

Examples

```
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]])

allnodes <- PnetPnodes(tnet)
stopifnot(length(allnodes)==0)

## Need to mark nodes a Pnodes before they will be seen.
nodes <- lapply(nodes,as.Pnode)
allnodes <- PnetPnodes(tnet)
stopifnot(length(allnodes)==3)
stopifnot(any(sapply(allnodes,"==",nodeA))) ## NodeA in there somewhere.

DeleteNetwork(tnet)
```

`PnetName` *Gets or Sets the name of a Netica network.*

Description

Gets or sets the name of the network. Names must conform to the `IDname` rules

Usage

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

Arguments

`net`	A `NeticaBN` object which links to the network.
`value`	A character scalar containing the new name.

Details

Network names must conform to the `IDname` rules for Netica identifiers. 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 `IDname` restrictions.

Value

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

The setter method returns the modified object.

Note

`NeticaBN` objects are internally implemented as character vectors giving the name of the network. If a network is renamed, then it is possible that R will hold onto an old reference that still using the old name. In this case, `PnetName(net)` will give the correct name, and `GetNamedNets(PnetName(net))` will return a reference to a corrected object.

Author(s)

Russell Almond

References

`http://norsys.com/onLineAPIManual/index.html`: GetNetName_bn(), SetNetName_bn()

See Also

`CreateNetwork()`, `NeticaBN`, `GetNamedNetworks()`, `PnetTitle()`

Examples

```
sess <- NeticaSession()
startSession(sess)

net <- CreateNetwork("funNet",sess)
netcached <- net

stopifnot(PnetName(net)=="funNet")

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

stopifnot(PnetName(net)==PnetName(netcached))

DeleteNetwork(net)
```

`PnetSerialize` *Methods for (un)serializing a Netica Network*

Description

Methods for functions `PnetSerialize` and `unserializePnet` in package **Peanut**, which serialize `NeticaBN` objects. Note that in this case, the factory is the `NeticaSession` object. These methods assume that there is a global variable with the name of the session object which points to the Netica session.

Methods

PnetSerialize, signature(net = "NeticaBN") Returns a vector with three components. The name field is the name of the network. The data component is a raw vector produced by calling serialize(...,NULL) on the output of a WriteNetworks operation. The factory component is the name of the NeticaSession object. Note that the PnetUnserialize function assumes that there is a global variable with name given by the factory argument which contains an appropriate NeticaSession object for the restoration.

unserializePnet, signature(factory = "NeticaSession") This method reverses the previous one. In particular, it applies ReadNetworks to the serialized object.

Examples

```
## Need to create session whose name is is the same a the symbol it is
## stored in.
MySession <- NeticaSession(SessionName="MySession")
startSession(MySession)

irt5 <- ReadNetworks(file.path(library(help="RNetica")$path,
                           "sampleNets","IRT5.dne"), session=MySession)
NetworkAllNodes(irt5)
CompileNetwork(irt5) ## Ready to enter findings
NodeFinding(irt5$nodes$Item_1) <- "Right"
NodeFinding(irt5$nodes$Item_2) <- "Wrong"

## Serialize the network
irt5.ser <- PnetSerialize(irt5)
stopifnot (irt5.ser$name=="IRT5",irt5.ser$factory=="MySession")

NodeFinding(irt5$nodes$Item_3) <- "Right"


## now revert by unserializing.
irt5 <- PnetUnserialize(irt5.ser)
NetworkAllNodes(irt5)
stopifnot(NodeFinding(irt5$nodes$Item_1)=="Right",
          NodeFinding(irt5$nodes$Item_2)=="Wrong",
          NodeFinding(irt5$nodes$Item_3)=="@NO FINDING")

DeleteNetwork(irt5)
stopSession(MySession)
```

PnetTitle *Gets the title or comments associated with a Netica network.*

Description

The title is a longer name for a network which is not subject to the Netica IDname restrictions. The comment is a free form text associated with a network.

Usage

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

Arguments

`net`	A `NeticaBN` object.
`value`	A character object giving the new title or comment.

Details

The title is meant to be a human readable alternative to the name, which is not limited to the `IDname` restrictions. The title also affects how the network is displayed in the Netica GUI.

The comment 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 comment.

Author(s)

Russell Almond

References

`http://norsys.com/onLineAPIManual/index.html`: GetNetTitle_bn(), SetNetTitle_bn(), GetNetComments_bn(), SetNetComments_bn()

See Also

`NeticaBN`, `NetworkName()`

Examples

```
sess <- NeticaSession()
startSession(sess)

firstNet <- CreateNetwork("firstNet",sess)

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

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

DeleteNetwork(firstNet)
```

`Pnode.NeticaNode` *Class* "NeticaNode" *as a* "Pnode"

Description

The PNetica package supplies the needed methods so that the RNetica::NeticaNode object is an instance of the Peanut::Pnode object. As a Pnode is nominally parameterized, the are given the special label "pnode" to indicate that this note has parametric information.

Extends

See NeticaNode for a description of the Netica class.

With these methods, NeticaNode now extends Pnode.

All reference classes extend and inherit methods from "envRefClass".

Methods

All methods are implementations of generic functions in the Peanut package. The following methods are related to the basic node structures and they should operate on all NeticaNode objects, whether they are Pnodes or not.

PnodeNet `signature(net = NeticaNode)`: Returns the NeticaBN (also Pnet) which contains the node.

PnodeName `signature(net = NeticaNode)`: Gets the name of the node.

PnodeName<- `signature(net = NeticaNode)`: Sets the name of the node.

PnodeTitle `signature(net = NeticaNode)`: Gets the title of the node.

PnodeTitle<- `signature(net = NeticaNode)`: Sets the title of the node.

PnodeDescription `signature(net = NeticaNode)`: Gets the description of the node.

PnodeProbs `signature(net = NeticaNode)`: Gets the conditional probability table for a node..

PnodeProbs<- `signature(net = NeticaNode)`: Sets the conditional probability table for a node.

PnodeDescription<- `(signature(net = NeticaNode)`: Sets the description of the node.

PnodeLabels `signature(net = NeticaNode)`: Gets the vector of names of the sets to which this node belongs.

PnodeLabels<- `signature(net = NeticaNode)`: Sets the vector of sets to which the node belongs.

isPnodeContinuous `signature(net = NeticaNode)`: Returns true or false, depending on whether or not node is continuous.

Documentation for other methods of the Pnode generic functions for NeticaNode objects can be found in the documentation objects Pnode.Properties and Pnode.States.

Note

The “Pnode properies”, `lnAlphas`, `betas`, `Q`, `rules`, `link`, `linkScale`, and `priorWeight` are stored in user fields (`NodeUserObj`) of the Netica node. A `NeticaNode` object which has those fields behaves as a `Pnode` and is suitable for the use with `Peanut`. The function `Pnode` will add default values for these fields if they are not set.

To mark a node as a `Pnode`, it is added to the node set “pnode”. The `is.Pnode` function checks for this method.

Author(s)

Russell Almond

See Also

Other methods of this class `Pnode.States`, `Pnode.Properties`.

Base class: `NeticaNode`.

Mixin class: `Pnode`.

Generic functions from `Peanut` package:

`Pnode`, `PnodeNet`, `PnodeName`, `PnodeTitle`, `PnodeDescription`, `PnodeLabels`, `PnodeNumParents`, `PnodeParentNames`, `PnodeParents`, `PnodeProbs`, `as.Pnode`, `is.Pnode`, `isPnodeContinuous`.

Examples

```
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="_"))
Duration <- NewContinuousNode(nsnet,"Duration")

## Pnode, is.Pnode, as.Pnode
stopifnot(!is.Pnode(EssayScore),!is.Pnode(Duration))
EssayScore <- Pnode(EssayScore)
Duration <- as.Pnode(Duration)
stopifnot(is.Pnode(EssayScore),is.Pnode(Duration))

## PnodeNet

stopifnot(PnodeNet(Ability)==nsnet)

## PnodeName, PnodeTitle, PnodeDescription
PnodeTitle(Ability) <- "Student Ability"
PnodeDescription(Ability) <-
"Students who have more ability will have more success on the exam."
stopifnot(PnodeTitle(Ability) == "Student Ability",
PnodeDescription(Ability) ==
"Students who have more ability will have more success on the exam."
)
```

```
## PnodeLabels
stopifnot(
  length(PnodeLabels(Ability)) == 0L ## Nothing set yet
)
PnodeLabels(Ability) <- "ReportingVariable"
stopifnot(
  PnodeLabels(Ability) == "ReportingVariable"
)
PnodeLabels(EssayScore) <- c("Observable",PnodeLabels(EssayScore))
stopifnot(
  !is.na(match("Observable",PnodeLabels(EssayScore)))
)
## Make EssayScore a reporting variable, too
PnodeLabels(EssayScore) <- c("ReportingVariable",PnodeLabels(EssayScore))
stopifnot(
  setequal(PnodeLabels(EssayScore),c("Observable","ReportingVariable","pnodes"))
)

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

## PnodeNumParents, PnodeParents

stopifnot(PnodeNumParents(Ability)==0L, PnodeParents(Ability)==list())
PnodeParents(EssayScore) <- list(Ability)
stopifnot(PnodeNumParents(EssayScore)==1L,
          PnodeParents(EssayScore)[[1]]==Ability,
          PnodeParentNames(EssayScore)=="Ability")

DeleteNetwork(nsnet)

## Node Probs
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)
```

`Pnode.Properties` *Properties of class* `"NeticaNode"` *as a* `"Pnode"`

Description

The `PNetica` package supplies the needed methods so that the `RNetica::NeticaNode` object is an instance of the `Peanut::Pnode` object. As a Pnode is nominally parameterized, the are given the special label "pnode" to indicate that this note has parametric information. This document describes the extra properties of `Pnode`s that are added by `PNetica`.

Extends

See `NeticaNode` for a description of the Netica class.

With these methods, `NeticaNode` now extends `Pnode`.

All reference classes extend and inherit methods from `"envRefClass"`.

Methods

All methods are implementations of generic functions in the `Peanut` package. These methods are related to the parameteric information which makes a node a `Pnode`. To inidcate that a node has this extra information, it should have the ""pnode"" label. The functions `Pnode` and `as.Pnode` will do this.

Pnode `signature(node = "NeticaNode",lnAlphas,betas,rules = "Compensatory",link = "partialCredit",Q = TRUE,linkScale = NULL,priorWeight = NULL)`: This function forces a `NeticaNode` into a `Pnode` by initializing the Pnode-specific fields.

PnodeLnAlphas `signature(node = NeticaNode)`: Returns the log of discrimination parameters associated with the node.

PnodeLnAlphas<- `signature(node = NeticaNode)`: Sets the log of discrimination parameters associated with the node.

PnodeBetas `signature(node = NeticaNode)`: Returns the difficulty parameters associated with the node.

PnodeBetas<- `signature(node = NeticaNode)`: Sets the difficulty parameters associated with the node.

PnodeQ `signature(node = NeticaNode)`: Returns the local Q matrix associated with the node.

PnodeQ<- `signature(node = NeticaNode)`: Sets the local Q matrix associated with the node.

PnodeRules `signature(node = NeticaNode)`: Returns the names of the combination rules associated with the node.

PnodeRules<- `signature(node = NeticaNode)`: Sets the names of the combination rules associated with the node.

PnodeLink `signature(node = NeticaNode)`: Returns the link function associated with the node.

PnodeLink<- `signature(node = NeticaNode)`: Sets the link function associated with the node.

PnodeLinkScale `signature(node = NeticaNode)`: Returns the link function scale parameter associated with the node.

PnodeLinkScale<- `signature(node = NeticaNode)`: Sets the link function scale parameter associated with the node.

PnodePriorWeight `signature(node = NeticaNode)`: Returns the weight or weights assigned to prior information associated with the node.

PnodePriorWeight<- `signature(node = NeticaNode)`: Sets the weight or weights assigned to prior information associated with the node.

PnodePostWeight `signature(node = NeticaNode)`: Returns the combined prior and data weights associated with the node.

as.Pnode `signature(x = NeticaNode)`: Forces `x` to be a `Pnode`; in particular, it adds the lable "pnode".

is.Pnode `signature(x = NeticaNode)`: Returns true if the node has the special label "pnode".

Documentation for other methods of the `Pnode` generic functions for `NeticaNode` objects can be found in the documentation objects Pnode.NeticaNode and Pnode.States.

Note

The “Pnode properies”, `lnAlphas`, `betas`, `Q`, `rules`, `link`, `linkScale`, and `priorWeight` are stored in user fields (`NodeUserObj`) of the Netica node. A `NeticaNode` object which has those fields behaves as a `Pnode` and is suitable for the use with `Peanut`. The function `Pnode` will add default values for these fields if they are not set.

To mark a node as a `Pnode`, it is added to the node set “pnode”. The `is.Pnode` function checks for this method.

Author(s)

Russell Almond

See Also

Other methods of this class `Pnode.NeticaNode`, `Pnode.Properties`.

Base class: `NeticaNode`.

Mixin class: `Pnode`.

Generic functions from `Peanut` package:

`PnodeLnAlphas`, `PnodeBetas`, `PnodeQ`, `PnodeRules`, `PnodeLink`, `PnodeLinkScale`, `PnodePostWeight`, `PnodePriorWeight`.

Examples

```
sess <- NeticaSession()
startSession(sess)
curd <- getwd()
setwd(file.path(library(help="PNetica")$path, "testnets"))

tNet <- CreateNetwork("TestNet",sess)

## Alphas
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)

## increasing intercepts for both transitions
PnodeLink(partial3) <- "gradedResponse"
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)
)


## 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)

## 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)


## 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)

## 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)

## The normal link function is the only one which takes a scale parameter
PnodeLink(partial3) <- "normalLink"
PnodeLinkScale(partial3) <- 1.0
PnodeLnAlphas(partial3) <- 0
PnodeBetas(partial3) <- c(0,1)
BuildTable(partial3)
stopifnot(
  all(abs(PnodeLinkScale(partial3)-1)<.0001)
)

DeleteNetwork(tNet)

stopSession(sess)
setwd(curd)
```

Pnode.States   *States of the* "NeticaNode" *as a* "Pnode"

Description

The PNetica package supplies the needed methods so that the RNetica::NeticaNode object is an instance of the Peanut::Pnode object. As a Pnode is nominally parameterized, the are given the special label “pnode” to indicate that this note has parametric information. This page documents the methods which access the states.

Extends

See NeticaNode for a description of the Netica class.

With these methods, NeticaNode now extends Pnode.

All reference classes extend and inherit methods from "envRefClass".

Methods

All methods are implementations of generic functions in the Peanut package. The following functions work with the states associated with the node. Each of the values returned or set is a vector whose length should match the number of states of the node.

PnodeNumStates signature(node = NeticaNode): Returns the number of states of the node.

PnodeStates signature(node = NeticaNode): Gets the names of the states of the node.

PnodeStates<- `signature(node = NeticaNode)`: Sets the number and names of the states of the node. Note that node state names must follow the IDname conventions.

PnodeStateTitles `signature(node = NeticaNode)`: Gets the titles of the states.

PnodeStateTitles<- `signature(node = NeticaNode)`: Sets the titles of the states.

PnodeStateDescriptions `signature(node = NeticaNode)`: Gets the description of the states.

PnodeStateDescriptions<- `(signature(node = NeticaNode)`: Sets the description of the states.

PnodeStateValues `signature(node = NeticaNode)`: Gets the vector of real values associated with the states. For continuous nodes, these are calculated from the bounds.

PnodeStateValues<- `signature(node = NeticaNode)`: Sets the vector of real values associated with the states; the node must be discrete.

PnodeStateBounds `signature(node = NeticaNode)`: Gets the K by 2 matrix of upper and lower bounds for a continuous node.

PnodeStateBounds<- `signature(node = NeticaNode)`: Sets the K by 2 matrix of upper and lower bounds for a continuous node. The upper and lower bounds must match, see the documentation in the Peanut package for more information.

Documentation for other methods of the `Pnode` generic functions for `NeticaNode` objects can be found in the documentation objects Pnode.NeticaNode and Pnode.Properties.

Note

Netica overrides the `NodeLevels` to do different things whether the node is continuous or discrete. The functions `PnodeStateValues` and `PnodeStateBounds` attempt to untangle these two different use cases. In particular, `NodeLevels` for a continuous node assumes that the range of the node is chopped into a number of contiguous segments, and what is fed to the function is a list of cut points. Thus, it will encouter problems if the lower bound of one state does not match the upper of the preious one.

Author(s)

Russell Almond

See Also

Other methods of this class `Pnode.NeticaNode`, `Pnode.Properties`.

Base class: `NeticaNode`.

Mixin class: `Pnode`.

Generic functions from `Peanut` package:

`PnodeNumStates`, `PnodeStates`, `PnodeStateTitles`, `PnodeStateDescriptions`, `PnodeStateValues`, `PnodeStateBounds`.

Examples

```
sess <- NeticaSession()
startSession(sess)
curd <- getwd()
setwd(file.path(library(help="PNetica")$path, "testnets"))

## Making states
anet <- CreateNetwork("Annette", session=sess)

## Discrete Nodes
nodel2 <- NewDiscreteNode(anet,"TwoLevelNode")
stopifnot(
  length(PnodeStates(nodel2))==2,
  PnodeStates(nodel2)==c("Yes","No")
)

PnodeStates(nodel2) <- c("True","False")
stopifnot(
  PnodeNumStates(nodel2) == 2L,
  PnodeStates(nodel2)==c("True","False")
)

nodel3 <- NewDiscreteNode(anet,"ThreeLevelNode",c("High","Med","Low"))
stopifnot(
  PnodeNumStates(nodel3) == 3L,
  PnodeStates(nodel3)==c("High","Med","Low"),
  PnodeStates(nodel3)[2]=="Med"
)

PnodeStates(nodel3)[2] <- "Median"
stopifnot(
  PnodeStates(nodel3)[2]=="Median"
)

PnodeStates(nodel3)["Median"] <- "Medium"
stopifnot(
  PnodeStates(nodel3)[2]=="Medium"
)


DeleteNetwork(anet)

## State Metadata (Titles and Descriptions)

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)

## State Values
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)
setwd(curd)
```

PnodeEvidence.NeticaNode

Gets or sets the value of a Pnode.

Description

Adding evidence to a Bayesian network is done by setting the value of the node to one of its states. The generic function `Peanut::PnodeEvidence` (and the method for a `NeticaNode`) simply returns the to which it is set, or `NA` if the node is not set. There are a number of different ways of setting the state depending on the type of the value argument (see Details).

Usage

```
## S4 method for signature 'NeticaNode'
PnodeEvidence(node)
## S4 replacement method for signature 'NeticaNode'
PnodeEvidence(node) <- value
```

Arguments

`node`	A `NeticaNode` object whose value is to be set.
`value`	A value representing the new type of the argument. See details.

Details

The generic function `PnodeEvidence` is defined in the `Peanut` package. It returns either the name of a state (discrete node), a numeric value (continuous node) or `NA` if the node has not been set.

There are different methods for different classes for the `value` argument (the RHS of the assignment operator).

ANY If no other method is appropriate, does nothing and issues a warning.

NULL The value of the node is retracted (`RetractNodeFinding`).

character If the `value` is the name of a state, then the node will be set to that state (`NodeFinding`). Otherwise, nothing will be done and a warning will be issued.

factor The character value of the `value` is uses (see character method).

logical This method assumes that the node has exactly two states, and that those states have values (`PnodeStateValues`, `NodeLevels`) 0 and 1. These levels are used to determine the mapping of TRUE and FALSE to states. If node state values are not set, then the character method is called using "TRUE" or "FALSE" as the value.

numeric If the `value` is of length 1, then the value of the node is set (`NodeValue`) to the argument. If the `value` is a vector of the same length as the number of states of the node, then it is regarded as virtual evidence, and the likelihood is set (`NodeLikelihood`).

difftime Difftime `values` are converted to real numbers in seconds, then the node value is set (see numeric method).

Value

`PnodeEvidence`: For all node types, if the node is not set, `PnodeEvidence` returns `NA`.

If the node is continuous, its currently set value is returned as a numeric scalar (`NA` if not set).

If the node is discrete, usually a character value giving the current state (or `NA`) is returned. However, if the node was assigned a likelihood instead of exact evidence, the likelihood vector is returned.

`PnodeEvidence<-` returns the node argument invisibly.

Note

For continuous nodes, `PnodeEvidence` is equivalent to `NodeValue`. For discrete nodes, it maps to either `NodeFinding` or `NodeLikelihood`

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

```
sess <- NeticaSession()
startSession(sess)

irt10.base <- ReadNetworks(paste(library(help="PNetica")$path,
                           "testnets","IRT10.2PL.base.dne",
                           sep=.Platform$file.sep),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)
```

PnodeParentTvals.NeticaNode

Fetches a list of numeric variables corresponding to parent states

---

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 `PnodeParentTvals` method for Netica nodes returns the result of applying `NodeLevels` to each of the nodes in `NodeParents(node)`.

Usage

```
## S4 method for signature 'NeticaNode'
PnodeParentTvals(node)
```

Arguments

node	A `Pnode` which is also a `NeticaNode`.

Details

While the best practices for assigning values to the states of the parent nodes is probably to assign equal spaced values (using the function `effectiveThetas` for this purpose), this method needs to retain some flexibility for other possibilities. However, in general, the best choice should depend on the meaning of the parent variable, and the same values should be used everywhere the parent variable occurs.

Netica already provides the `NodeLevels` function which allows the states of a `NeticaNode` to be associated with numeric values. This method merely gathers them together. The method assumes that all of the parent variables have had their `NodeLevels` set and will generate an error if that is not true.

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 implementation is merely: `lapply(NodeParents(node),NodeLevels)`.

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.NeticaNode`, `Pnode`, `effectiveThetas`, `BuildTable,NeticaNode-method`, `maxCPTParam,NeticaNode-method`

Examples

```
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
NodeLevels(theta1) <- effectiveThetas(NodeNumStates(theta1))
NodeProbs(theta1) <- rep(1/NodeNumStates(theta1),NodeNumStates(theta1))
theta2 <- NewDiscreteNode(tNet,"theta2",
                            c("High","Mid","Low"))
## This next function sets the effective thetas for theta2
NodeLevels(theta2) <- effectiveThetas(NodeNumStates(theta2))
NodeProbs(theta2) <- rep(1/NodeNumStates(theta2),NodeNumStates(theta2))

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

Statistic.NeticaNode *Statistic methods for* "NeticaBN" *class.*

Description

These are the implementation for the basic statistic calculation methods.

Methods

All methods have signature `signature(net = "NeticaBN",node = "NeticaNode")` and `signature(net = "NeticaBN",node = "character")`. The later form is more often used, and takes the name of the node and finds the appropriate node in the network.

PnodeEAP Calculates the marginal distribution of the node. Statistic returns a named vector of values.

PnodeEAP Calculates the expected value of the node; assumes numeric values have been set with `PnodeStateValues`.

PnodeSD Calculates the standard deviation of the node; assumes numeric values have been set with `PnodeStateValues`.

PnodeMedian Calculates the median state (state whose cumulative probability covers .5) of the node. Statistic returns the name of the state.

PnodeMode Calculates the modal (most likely) state of the node. Statistic returns the name of the state.

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

```
sess <- NeticaSession()
startSession(sess)

irt10.base <- ReadNetworks(paste(library(help="PNetica")$path,
                           "testnets","IRT10.2PL.base.dne",
                           sep=.Platform$file.sep),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)
```

WarehouseDirectory *Gets or sets the directory associated with an BNWarehouse*

Description

If a network is not available, a BNWarehouse will look in the specified directory to find the .dne or .neta files associated with the Bayesian networks.

Usage

```
WarehouseDirectory(warehouse)
WarehouseDirectory(warehouse) <- value
```

Arguments

warehouse	An object of type BNWarehouse.
value	A character scalar giving the new pathname for the net directory.

Value

A character string giving the path associated with a Warehouse.

Author(s)

Russell Almond

See Also

BNWarehouse, MakePnet.NeticaBN

Examples

```
sess <- NeticaSession()
startSession(sess)

netman1 <- read.csv(paste(library(help="Peanut")$path, "auxdata",
                          "Mini-PP-Nets.csv", sep=.Platform$file.sep),
                    row.names=1, stringsAsFactors=FALSE)

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

## Set up to use a temporary directory (all networks will be built fresh)
td <- tempdir()
WarehouseDirectory(Nethouse) <- td
stopifnot(WarehouseDirectory(Nethouse)==td)
```

Index