Package 'Peanut'

March 30, 2017

Version 0.2-2

Date 2016/11/07

Title Parameterized Bayesian Networks, Abstract Classes

Author Russell Almond

Maintainer Russell Almond <ralmond@fsu.edu>

Depends R (>= 3.0), CPTtools (>= 0.4)

Description
This provides support of learning conditional probability tables parameterized using CPTtools

License Artistic-2.0

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

R topics documented:

Peanut-package *Parameterized Bayesian Networks, Abstract Classes*

Description

This provides support of learning conditional probability tables parameterized using CPTtools

Details

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

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

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

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

Index

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

Author(s)

Russell Almond

Maintainer: Russell Almond <ralmond@fsu.edu>

References

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

See Also

PNetica An implementation of the Peanut object model using RNetica.

CPTtools A collection of implementation independent Bayes net utilities.

Examples

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


## Building CPTs
tNet <- CreateNetwork("TestNet")


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

## End(Not run)
```

BNgenerics *Aliases for generic Bayesian network functions.*

Description

These are all mostly self-explantory functions which almost any Bayesian network implementation will support. These are alias of these functions so that generic functions can be written using Peanut which will support almost all Bayes net implementation.

Usage

```
PnodeName(node)
```

```
PnodeStates(node)
PnodeNumStates(node)
PnodeParents(node)
PnodeNumParents(node)
PnodeParentNames(node)
```

Arguments

node	A object of a type which could be a Pnode, athough it does not necessarily need Pnode special properties.

Details

The general idea is to find a minimal set of common Bayes net functions that any reasonable Bayes net package is likely to support so that basic code can be written which is generic across Bayes net packages. For example, if nd is a RNetica{NeticaNode} object, then PnodeName(nd) is a synonym for NodeName(nd). However, using PnodeName(nd) is more portable as it could expand to another function if using a different Bayes net package.

The goal is to be able to write simple loops based on things like number of parents and number of states which are common to most implementations.

Note that setter methods are not supported, thus although NodeName(nd) <- newname is valid in RNetica.package, PnodeName(nd) <- newname will return an error.

Value

The expression PnodeName(node) returns a character scalar giving the name of node.

The expression PnodeStates(node) returns a character vector giving the names of the states of node.

The expression PnodeNumStates(node) returns an integer scalar giving the number of states of node.

The expression PnodeParents(node) returns a list giving the parent objects.

The expression PnodeNumParents(node) returns an integer scalar giving the number of parents of node.

The expression PnodeStates(node) returns a character vector giving the names of the parents of node.

Author(s)

Russell Almond

Examples

```
## Not run:
PnodeName.NeticaNode <- function (node)
  NodeName(node)

PnodeStates.NeticaNode <- function (node)
  NodeStates(node)
```

```
PnodeNumStates.NeticaNode <- function (node)
  NodeNumStates(node)

PnodeParents.NeticaNode <- function (node)
  NodeParents(node)

PnodeParentNames.NeticaNode <- function (node)
  sapply(NodeParents(node),NodeName)

PnodeNumParents.NeticaNode <- function (node)
  length(NodeParents(node))


## End(Not run)
```

BuildTable *Builds the conditional probability table for a Pnode*

Description

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

Usage

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

Arguments

node	A Pnode object whose table is to be built.
net	A Pnet object for whom the tables are needed to be built.
debug	A logical scalar. If true the names of the nodes are printed as each one is built to help determine where errors are occuring.

Details

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

Value

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

Note

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

Author(s)

Russell Almond

References

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

See Also

Pnode, PnodeQ, PnodePriorWeight, PnodeRules, PnodeLink, PnodeLnAlphas, PnodeAlphas, PnodeBetas, PnodeLinkScale,GetPriorWeight, calcDPCTable

Examples

```
## Not run:

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

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

## This is the implementation of BuildAllTables
BuildAllTables <- function (net) {
  lapply(PnetPnodes(net),BuildTable)
  invisible(net)
}



## End(Not run)
```

calcExpTables *Calculate expected tables for a parameterized network*

Description

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

Usage

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

Arguments

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

Details

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

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

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

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

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

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

The internal EM algorithm should be set to use the current value of the conditional probability tables (as calculated by BuildTable(node) for each node) as a starting point. This starting value is

given a prior weight of GetPriorWeight(node). Note that some Bayes net implementations allow a different weight to be given to each row of the table. The prior weight counts as a number of cases, and should be scaled appropriately for the number of cases in cases.

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

Value

The net argument is returned invisibly.

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

Note

The function calcExpTables is an abstract generic functions, and it needs specific implementations. See the PNetica-package for an example.

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

Author(s)

Russell Almond

References

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

See Also

Pnet, GEMfit, calcPnetLLike, maxAllTableParams

Examples

```
## Not run:

library(PNetica) ## Need a specific implementation

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


## End(Not run)
```

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

Description

The function `calcPnetLLike` calculates the log likelihood for a set of data contained in `cases` using the current values of the conditional probability table in a `Pnet`. If it is called after a call to `BuildAllTables(net)` this will be the current value of the parameters.

Usage

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

Arguments

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

Details

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

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

Value

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

Note

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

Author(s)

Russell Almond

References

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

See Also

Pnet, GEMfit, calcExpTables, maxAllTableParams

Examples

```
## Not run:

library(PNetica) ## Need a specific implementation

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


## End(Not run)
```

GEMfit — *Fits the parameters of a parameterized network using the GEM algorithm*

Description

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

Usage

```
GEMfit(net, cases, tol = sqrt(.Machine$double.eps),
       maxit = 100, Estepit = 1, Mstepit = 3,
       trace=FALSE, debugNo=maxit+1)
```

Arguments

net	A Pnet object
cases	An object representing a set of cases. Note the type of object is implementation dependent. It could be either a data frame providing cases or a filename for a case file.
tol	A numeric scalar giving the stopping tolerance for the for the EM algorithm.
maxit	An integer scalar giving the maximum number of iterations for the outer EM algorithm.
Estepit	An integer scalar giving the number of steps the Bayes net package should take in the internal EM algorithm during the E-step.
Mstepit	An integer scalar giving the number of steps that should be taken by mapDPC during the M-step.

trace	A logical value which indicates whether or not cycle by cycle information should be sent to standard output.
debugNo	An integer scalar. Node-by-node status updates will start when this iteration is reached. Default value provides no debugging information.

Details

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

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

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

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

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

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

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

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

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

Value

A list with three elements:

converged	A logical flag indicating whether or not the algorithm reached convergence.
iter	An integer scalar giving the number of iterations of the outer EM loop taken by the algorithm (plus 1 for the starting point).

`llikes` A numeric vector of length `iter` giving the values of the log-likelihood after each iteration. (The first value is the initial log likelihood.)

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

Note

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

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

Author(s)

Russell Almond

References

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

See Also

`Pnet`, `calcExpTables`, `calcPnetLLike`, `maxAllTableParams`, `BuildAllTables`

Examples

```
## Not run:

library(PNetica) ## Need a specific implementation

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

```
                          "testdat","IRT10.2PL.200.items.cas",
                          sep=.Platform$file.sep)
## 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)

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

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


### The network that was used for data generation.
irt10.true <- ReadNetworks(paste(library(help="PNetica")$path,
                          "testnets","IRT10.2PL.true.dne",
                          sep=.Platform$file.sep))
irt10.true <- as.Pnet(irt10.true)  ## Flag as Pnet, fields already set.
irt10.ttheta <- NetworkFindNode(irt10.true,"theta")
irt10.titems <- PnetPnodes(irt10.true)
## Flag titems as Pnodes
for (i in 1:length(irt10.titems)) {
  irt10.titems[[i]] <- as.Pnode(irt10.titems[[i]])
}
NetworkNodesInSet(irt10.true,"onodes") <-
   NetworkNodesInSet(irt10.true,"observables")

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




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


## End(Not run)
```

maxAllTableParams *Find optimal parameters of Pnet or Pnode to match expected tables*

Description

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

Usage

```
maxAllTableParams(net, Mstepit = 5, tol = sqrt(.Machine$double.eps), debug=TRUE)
## Default S3 method:
maxAllTableParams(net, Mstepit = 5, tol = sqrt(.Machine$double.eps), debug=TRUE)
maxCPTParam(node, Mstepit = 5, tol = sqrt(.Machine$double.eps))
```

Arguments

net	A Pnet object giving the parameterized network.
node	A Pnode object giving the parameterized node.
Mstepit	A numeric scalar giving the number of maximization steps to take. Note that the maximization does not need to be run to convergence.
tol	A numeric scalar giving the stopping tolerance for the maximizer.
debug	A logical scalar. If true, information about which node is being worked on is printed.

Details

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

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

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

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

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

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

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

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

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

Value

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

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

Note

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

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

Author(s)

Russell Almond

References

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

See Also

Pnet, Pnode, GEMfit, calcPnetLLike, calcExpTables, mapDPC

Examples

```
## Not run:

library(PNetica) ## Need a specific implementation

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

calcExpTables(irt10.base,casepath)

maxAllTableParams(irt10.base)

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

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


DeleteNetwork(irt10.base)
```

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

Pnet *A Parameterized Bayesian network*

Description

A parameterized Bayesian network. Note that this is a "mixin" class, it is meant be added to another Bayesian network class to indicate that it contains parameterized nodes.

Usage

```
is.Pnet(x)
as.Pnet(x)
Pnet(net, priorWeight=10, pnodes=list())
## Default S3 method:
Pnet(net, priorWeight=10, pnodes=list())
```

Arguments

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

Details

The Pnet class is basically a protocol which any Bayesian network net object can follow to work with the tools in the Peanut package. This is really an abstract class (in the java programming language, Pnet would be an interface rather than a class) which exploits the rather loose S3 object system. In particular, a Pnet is any object for which is.Pnet returns true. The default method looks for the string "Pnet" in the class list.

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

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

The importance of the Pnet object is that it supports the GEMfit method which adjust the parameters of the Pnode objects to fit a set of case data. In order to be compatible with GEMfit, the

Pnet object must support four methods: BuildAllTables, calcPnetLLike, calcExpTables, and maxAllTableParams.

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

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

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

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

Value

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

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

Author(s)

Russell Almond

References

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

See Also

Fields: PnetPriorWeight, PnetPnodes

Generic Functions: BuildAllTables, calcPnetLLike, calcExpTables, maxAllTableParams

Functions: GEMfit

Related Classes: Pnode

Examples

```
 ## Not run:

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

theta <- NewDiscreteNode(IRT10.2PL,"theta",
```

```
                          c("VH","High","Mid","Low","VL"))
NodeLevels(theta) <- effectiveThetas(NodeNumStates(theta))
NodeProbs(theta) <- rep(1/NodeNumStates(theta),NodeNumStates(theta))

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

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

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

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

DeleteNetwork(IRT10.2PL)

## End(Not run)
```

PnetPnodes *Returns a list of Pnodes associated with a Pnet*

Description

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

Usage

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

Arguments

net	A Pnet object.
node	A Pnode object.
value	A list of Pnode objects associated with net.

Details

The primary purpose of PnetPnodes is to provide a list of nodes which GEMfit and BuildAllTables will iterate to do their function.

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

Value

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

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

Note

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

Author(s)

Russell Almond

References

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

See Also

Pnet, Pnode, GetPriorWeight, BuildAllTables, GEMfit

Examples

```
 ## Not run:

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

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

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

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

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

## End(Not run)
```

`PnetPriorWeight` *Gets the weight to be associated with the prior table during EM learning*

Description

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

Usage

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

Arguments

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

Details

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

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

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

Value

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

Note

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

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

Author(s)

Russell Almond

References

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

See Also

Pnet, Pnode, PnodeNet, BuildTable, GEMfit

Examples

```
 ## Not run:

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

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

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

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

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

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

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

PnetPriorWeight(IRT10.2PL) <- 15

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

```
DeleteNetwork(IRT10.2PL)

## End(Not run)
```

Pnode *A Parameterized Bayesian network node*

Description

A node in a parameterized Bayesian network. Note that this is a "mixin" class, it is meant be added to another Bayesian network node class to indicate that it is parameterized using the discrete partial credit framework.

Usage

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

Arguments

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

priorWeight — A numeric vector of weights given to the prior parameter values for each row of the conditional probability table when learning from data (or a scalar if all rows have equal prior weight). See PnodePriorWeight for a description of this parameter.

Details

The Pnode class is basically a protocol which any Bayesian network node object can follow to work with the tools in the Peanut package. This is really an abstract class (in the java language, Pnode would be an interface rather than a class) which exploits the rather loose S3 object system. In particular, a Pnode is any object for which is.Pnode returns true. The default method looks for the string "Pnode" in the class list.

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

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

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

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

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

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

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

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

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

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

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

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

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

Value

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

Author(s)

Russell Almond

References

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

See Also

Fields: PnodeNet, PnodeQ, PnodePriorWeight, PnodeRules, PnodeLink, PnodeLnAlphas, PnodeAlphas, PnodeBetas, PnodeLinkScale

Generic Functions: BuildTable, PnodeParentTvals, maxCPTParam

Functions: GetPriorWeight, calcDPCTable, mapDPC

Related Classes: Pnet

Examples

```
 ## Not run:

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

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

maxCPTParam.NeticaNode <- function (node, Mstepit=3,
                                    tol=sqrt(.Machine$double.eps)) {
  ## Get the posterior pseudo-counts by multiplying each row of the
  ## node's CPT by its experience.
  counts <- sweep(node[[]],1,NodeExperience(node),"*")
  est <- mapDPC(counts,ParentStates(node),NodeStates(node),
                PnodeLnAlphas(node), PnodeBeta(node),
                PnodeRules(node),PnodeLink(node),
                PnodeLinkScale(node),PnodeQ(node),
                control=list(reltol=tol,maxits=Mstepit)
```

```
                )
  PnodeLnAlphas(node) <- est$lnAlphas
  PnodeBetas(node) <- est$betas
  PnodeLinkScale(node) <- est$linkScale
  invisible(node)
}



## End(Not run)
```

PnodeBetas *Access the combination function slope parameters for a Pnode*

Description

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

Usage

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

Arguments

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

Details

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

1. Each set of parent variable states is converted to a set of continuous values called *effective thetas* (see PnodeParentTvals). These are built into an array, eTheta, using expand.grid where each column represents a parent variable and each row a possible configuration of parents.
2. For each state of the node except the last, the set of effective thetas is filtered using the local Q-matrix, PnodeQ(node) = Q. Thus, the actual effect thetas for state s is eTheta[,Q[s,]].

3. For each state of the node except the last, the corresponding rule is applied to the effective thetas to get a single effective theta for each row of the table. This step is essentially calls the expression: `do.call(rules[[s]], list(eThetas[,Q[s,]]), PnodeAlphas(node)[[s]], PnodeBetas(node)[[s]])`.
4. The resulting set of effective thetas are converted into conditional probabilities using the link function `PnodeLink(node)`.

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

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

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

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

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

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

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

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

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

Value

A list of numeric vectors giving the intercepts for the combination function of each state transition. The vectors may be of different lengths depending on the value of `PnodeRules(node)` and

PnodeQ(node). If the intercepts are the same for all transitions then a single numeric vector instead of a list is returned.

Note

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

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

Author(s)

Russell Almond

References

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

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

See Also

Pnode, PnodeQ, PnodeRules, PnodeLink, PnodeLnAlphas, BuildTable, PnodeParentTvals, maxCPTParam calcDPCTable, mapDPC Compensatory, OffsetConjunctive

Examples

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

tNet <- CreateNetwork("TestNet")


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)

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

```
PnodePriorWeight(partial3) <- 10
BuildTable(partial3)


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

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


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


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

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

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


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


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


DeleteNetwork(tNet)
```

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

PnodeLink *Accesses the link function associated with a Pnode*

Description

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

Usage

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

Arguments

node	A Pnode object.
value	The name of a link function or function object which can serve as the link function.

Details

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

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

A link function is a function of three arguments. The first is a matrix of effective theta values with number of rows equal to the number of rows of the conditional probability matrix and number of columns equal to the number of states of node minus one (ordered from highest to lowest). The second is an optional link scale, the third is a set of names for the states which is used to give column names to the output matrix. The second and third both default to NULL.

Currently two link functions are partialCredit and gradedResponse. Note that the function gradedResponse assumes that the effective thetas in each row are in increasing order. This puts certain restrictions on the parameter values. Generally, this can only be guaranteed if each state of the variable uses the same combination rules (see PnodeRules(node)), slope parameters (see PnodeAlphas(node)) and Q-matrix (see PnodeQ(node)). Also, the intercepts (see PnodeBetas(node)) should be in decreasing order. The partialCredit model has fewer restrictions.

The value of PnodeLinkScale(node) is fed to the link function. Currently, this is unused; but the DiBello-normal model (see calcDNTable) uses it. So the link scale parameter is for future expansion.

Value

A character scalar giving the name of a combination function or a combination function object.

Note

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

A third normal link function, which would use the scale parameter, is planned but not yet implemented.

Author(s)

Russell Almond

References

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

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

See Also

Pnode, PnodeQ, PnodeRules PnodeLinkScale, PnodeLnAlphas, PnodeBetas, BuildTable, PnodeParentTvals, maxCPTParam calcDPCTable, mapDPC Compensatory, OffsetConjunctive

Examples

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

tNet <- CreateNetwork("TestNet")

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)

## Usual way to set link is in constructor
partial3 <- Pnode(partial3,rules="Compensatory", link="gradedResponse")
PnodePriorWeight(partial3) <- 10
PnodeBetas(partial3) <- list(FullCredit=1,PartialCredit=0)
BuildTable(partial3)

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

## Can use different slopes with partial credit
## Make Skill 1 more important for the transition to ParitalCredit
## And Skill 2 more important for the transition to FullCredit
PnodeLnAlphas(partial3) <- list(FullCredit=c(-.25,.25),
                                PartialCredit=c(.25,-.25))
BuildTable(partial3)

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

DeleteNetwork(tNet)

## End(Not run)
```

PnodeLinkScale *Accesses the link function scale parameter associated with a Pnode*

Description

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

Usage

```
PnodeLinkScale(node)
PnodeLinkScale(node) <- value
```

Arguments

node	A Pnode object.
value	A positive numeric value, or NULL if the scale parameter is not used for the link function.

Details

The link function used in constructing the conditional probability table is controlled by the value of PnodeLink(node). One of the arguments to the link function is a scale parameter, the expression PnodeLinkScale(node) provides the link scale parameter associated with the node.

This is mostly for future expansion. Currently, neither of the two link functions defined in the CPTtools package, partialCredit and gradedResponse, require a link scale parameter. However, the DiBello-normal model (see calcDNTable) uses a link scale parameter so it may be useful in the future.

Value

The value of the link scale parameter, or NULL if it is not needed.

Note

The functions PnodeLinkScale and PnodeLinkScale<- are abstract generic functions, and need specific implementations. See the PNetica-package for an example. Even though they are not currently used, they must be defined and return a value (even just NULL).

A third normal link function, which would use the scale parameter, is planned but not yet implemented.

Author(s)

Russell Almond

References

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

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

See Also

Pnode, PnodeQ, PnodeRules, PnodeLinkScale, PnodeLnAlphas, PnodeBetas, BuildTable, PnodeParentTvals, maxCPTParam, calcDPCTable, mapDPC, Compensatory, OffsetConjunctive

Examples

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

tNet <- CreateNetwork("TestNet")

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,rules="Compensatory", link="gradedResponse")
PnodePriorWeight(partial3) <- 10

stopifnot(
  is.null(PnodeLinkScale(partial3))
)

PnodeLinkScale(partial3) <- 1.0

stopifnot(
  all(abs(PnodeLinkScale(partial3)-1)<.0001)
)

DeleteNetwork(tNet)

## End(Not run)
```

PnodeLnAlphas *Access the combination function slope parameters for a Pnode*

Description

In constructing a conditional probability table using the discrete partial credit framework (see calcDPCTable), the effective thetas for each parent variable are combined into a single effect theta using a combination rule. The expression PnodeAlphas(node) accesses the slope parameters associated with the combination function PnodeRules(node). The expression PnodeLnAlphas(node) which is used in mapDPC.

Usage

```
PnodeLnAlphas(node)
PnodeLnAlphas(node) <- value
PnodeAlphas(node)
PnodeAlphas(node) <- value
## Default S3 method:
PnodeAlphas(node)
## Default S3 replacement method:
PnodeAlphas(node) <- value
```

Arguments

node	A Pnode object.
value	A numeric vector of (log) slope parameters or a list of such vectors (see details). The length of the vector depends on the combination rules (see PnodeRules). If a list, it should have length one less than the number of states in node. For PnodeAlphas(node) <- value, value should only contain positive numbers.

Details

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

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

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

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

where $C = \sqrt{(J)}$ is a variance stabilization constant and $beta$ is a value derived from PnodeBetas. The functions Conjunctive and Disjunctive are similar replacing the sum with a min or max respectively.

In general, when the rule is one of Compensatory, Conjunctive, or Disjunctive, the the value of PnodeAlphas(node) should be a vector of the same length as the number of parents. As a special case, if it is a vector of length 1, then a model with a common slope is used. This looks the same in calcDPCTable but has a different implication in mapDPC where the parameters are constrained to be the same.

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

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

In this case the assumption is that PnodeAlphas(node) will be a scalar and PnodeBetas(node) will be a vector of length equal to the number of parents.

If the value of PnodeLink is partialCredit, then the link function can be different for each state of the node. (If it is gradedResponse then the curves need to be parallel and the slopes should be the same.) If the value of PnodeAlphas(node) is a list (note: list, not numeric vector or matrix), then a different set of slopes is used for each state transition. (This is true whether PnodeRules(node) is a single function or a list of functions. Note that if there is a different rule for each transition, they could require different numbers of slope parameters.) The function calcDPCTable assumes the states are ordered from highest to lowest, and no transition is needed into the lowest state.

Node that if the value of PnodeQ(node) is not a matrix of all TRUE values, then the effective number of parents for each state transition could be different. In this case the value of PnodeAlphas(node) should be a list of vectors of different lengths (corresponding to the number of true entries in each row of PnodeQ(node)).

Finally, note that if we want the conditional probability table associated with node to be monotonic, then the PnodeAlphas(node) must be positive. To ensure this, mapDPC works with the log of the slopes, not the raw slopes. Similarly, calcDPCTable expects the log slope parameters as its lnAlphas argument, not the raw slopes. For that reason PnodeLnAlphas(node) is considered the primary function and a default method for PnodeAlphas(node) which simply takes exponents (or logs in the setter) is provided. Note that a sensible range for the slope parameters is usually between 1/2 and 2, with 1 (0 on the log scale) as a sensible first pass value.

Value

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

Note

The functions PnodeLnAlphas and PnodeLnAlphas<- are abstract generic functions, and need specific implementations. The default methods for the functions PnodeAlphas and PnodeAlphas<-. Depend on PnodeLnAlphas and PnodeLnAlphas<-, respectively. See the PNetica-package for an example.

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

Author(s)

Russell Almond

References

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

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

See Also

Pnode, PnodeQ, PnodeRules, PnodeLink, PnodeBetas, BuildTable, PnodeParentTvals, maxCPTParam calcDPCTable, mapDPC Compensatory, OffsetConjunctive

Examples

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

tNet <- CreateNetwork("TestNet")


theta1 <- NewDiscreteNode(tNet,"theta1",
                          c("VH","High","Mid","Low","VL"))
NodeLevels(theta1) <- effectiveThetas(NodeNumStates(theta1))
NodeProbs(theta1) <- rep(1/NodeNumStates(theta),NodeNumStates(theta1))
theta2 <- NewDiscreteNode(tNet,"theta2",
                          c("VH","High","Mid","Low","VL"))
NodeLevels(theta2) <- effectiveThetas(NodeNumStates(theta2))
NodeProbs(theta2) <- rep(1/NodeNumStates(theta),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")
PnodePriorWeight(partial3) <- 10
BuildTable(partial3)

## slopes of 1 for both transitions
PnodeLnAlphas(partial3) <- c(0,0)
BuildTable(partial3)

## log slope 0 = slope 1
stopifnot(
   all(abs(PnodeAlphas(partial3) -1) <.0001)
)
```

```
## Make Skill 1 more important than Skill 2
PnodeLnAlphas(partial3) <- c(.25,-.25)
BuildTable(partial3)

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


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

## Using OffsetConjunctive rule requires single slope
PnodeRules(partial3) <- "OffsetConjunctive"
## Single slope parameter for each transition
PnodeLnAlphas(partial3) <- 0
PnodeQ(partial3) <- TRUE
PnodeBetas(partial3) <- c(0,1)
BuildTable(partial3)

## Separate slope parameter for each transition;
## Note this will only different from the previous transition when
## mapDPC is called.  In the former case, it will learn a single slope
## parameter, in the latter, it will learn a different slope for each
## transition.
PnodeLnAlphas(partial3) <- list(0,0)
BuildTable(partial3)


DeleteNetwork(tNet)

## End(Not run)
```

PnodeParentTvals *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 function PnodeParentTvals returns a list of effective theta values for each parent variable.

Usage

```
PnodeParentTvals(node)
```

Arguments

node	A Pnode object.

Details

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

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

This function is responsible for the first step of this process. PnodeParentTvals(node) should return a list corresponding to the parents of node, and each element should be a numeric vector corresponding to the states of the appropriate parent variable. It is passed to expand.grid to produce the table of parent variables for each row of the CPT.

Note that in item response theory, ability (theta) values are assumed to have a unit normal distribution in the population of interest. Therefore, appropriate theta values are quantiles of the normal distribution. In particular, they should correspond to the marginal distribution of the parent variable. The function effectiveThetas produces equally spaced (wrt the normal measure) theta values (corresponding to a uniform distribution of the parent). Unequally spaced values can be produced by using appropriate values of the qnorm function, e.g. qnorm(c(.875,.5,.125)) will produce effective thetas corresponding to a marginal distribution of (0.25, 0.5, 0.25) (note that each value is in the midpoint of the interval).

Value

PnodeParentTvals(node) should return a list corresponding to the parents of node, and each element should be a numeric vector corresponding to the states of the appropriate parent variable. If there are no parent variables, this will be a list of no elements.

Note

The function PnodeParentTvals is an abstract generic functions, and need specific implementations. See the PNetica-package for an example.

In particular, it is probably a mistake to using different effective theta values for different parent variables in different contexts, therefor, the cleanest implementation is to associate the effective thetas with the parent variables and simply have `PnodeParentTvals` fetch them on demand. Thus the implementation in `PNetica` is simply, `lapply(NodeParents(node), 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`, `PnodeQ`, `PnodeRules`, `PnodeLink`, `PnodeLnAlphas`, `PnodeBetas`, `BuildTable`, `PnodeParentTvals`, `maxCPTParam` `calcDPCTable`, `mapDPC` `effectiveThetas`, `expand.grid`, `qnorm`

Examples

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

tNet <- CreateNetwork("TestNet")

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)

## End(Not run)
```

PnodeQ *Accesses a state-wise Q-matrix associated with a Pnode*

Description

The function calcDPCTable has an argument Q, which allows the designer to specify that only certain parent variables are relevant for the state transition. The function PnodeQ accesses the local Q-matrix for the Pnode node.

Usage

```
PnodeQ(node)
PnodeQ(node) <- value
```

Arguments

node	A Pnode whose local Q-matrix is of interest
value	A logical matrix with number of rows equal to the number of outcome states of node minus one and number of columns equal to the number of parents of node. As a special case, if it has the value TRUE this is interpreted as a matrix of true values of the correct shape.

Details

Consider a partialCredit model, that is a Pnode for which the value of PnodeLink is "partialCredit". This model is represented as a series of transitions between the states $s+1$ and s (in calcDPCTable states are ordered from high to low). The log odds of this transition is expressed with a function $Z_s(eTheta)$ where $Z_s()$ is the value of PnodeRules(node) and $eTheta$ is the result of the call PnodeParentTvals(node).

Let q_{sj} be true if the parent variable x_j is relevant for the transition between states $s+1$ and s. Thus the function which is evaluated to calculate the transition probabilities is $Z_s(eTheta[, Q[s,]])$; that is, the parent variables for which q_{sj} is false are filtered out. The default value of TRUE means that no values are filtered.

Note that this currently makes sense only for the partialCredit link function. The gradedResponse link function assumes that the curves are parallel and therefore all of the curves must have the same set of variables (and values for PnodeAlphas.

Value

A logical matrix with number of rows equal to the number of outcome states of node minus one and number of columns equal to the number of parents of node, or the logical scalar TRUE if all parent variables are used for all transitions.

Note

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

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

Author(s)

Russell Almond

References

Almond, R. G. (2013) Discretized Partial Credit Models for Bayesian Network Conditional Probability Tables. Draft manuscript available from author.

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

See Also

Pnode, PnodeRules, PnodeLink, PnodeLnAlphas, PnodeAlphas, BuildTable, PnodeParentTvals, maxCPTParam calcDPCTable, mapDPC

Examples

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

tNet <- CreateNetwork("TestNet")


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)
```

```
## Default is all nodes relevant for all transitions
stopifnot(
   length(PnodeQ(partial3)) == 1,
   PnodeQ(partial3) == TRUE
)

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


partial4 <- NewDiscreteNode(tNet,"partial4",
                            c("Score4","Score3","Score2","Score1"))
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)

DeleteNetwork(tNet)


## End(Not run)
```

PnodeRules *Accesses the combination rules for a Pnode*

Description

In constructing a conditional probability table using the discrete partial credit framework (see calcDPCTable), the effective thetas for each parent variable are combined into a single effect theta using a combination rule. The function PnodeRules accesses the combination function associated with a Pnode.

Usage

```
PnodeRules(node)
PnodeRules(node) <- value
```

Arguments

node	A Pnode object.
value	The name of a combination function, the combination function or a list of names or combination functions (see details). If a list, it should have length one less than the number of states in node.

Details

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

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

The function PnodeRules accesses the function used in step 3. It should should be the name of a function or a function with the general signature of a combination function described in Compensatory. Predefined choices include Compensatory, Conjunctive, Disjunctive, OffsetConjunctive, and OffsetDisjunctive. Note that the first three choices expect that there will be multiple alphas, one for each parent, and the latter two expect that there will be multiple betas, one for each beta. The value of PnodeAlphas and PnodeBetas should be set to match.

If the value of PnodeLink is partialCredit, then the link function can be different for state of the node. (If it is gradedResponse then the curves need to be parallel and it should be the same.) If the value of PnodeRules(node) is a list (note: list, not character vector), then a different rule is used for each state transition. The function calcDPCTable assumes the states are ordered from highest to lowest, and no transition is needed into the lowest state.

Value

A character scalar giving the name of a combination function or a combination function object, or a list of the same. If a list, its length is one less than the number of states of node.

Note

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

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

Author(s)

Russell Almond

References

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

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

See Also

Pnode, PnodeQ, PnodeLink, PnodeLnAlphas, PnodeBetas, BuildTable, PnodeParentTvals, maxCPTParam, calcDPCTable, mapDPC, Compensatory, OffsetConjunctive

Examples

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

tNet <- CreateNetwork("TestNet")

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)

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

stopifnot(
   PnodeRules(partial3) == "Compensatory"
)

## Use different rules for different levels
## Compensatory for 2nd transition, conjunctive for 1st
## Note:  Position is important, names are just for documentation.
PnodeRules(partial3) <- list(FullCredit="Compensatory",
                             PartialCredit="Conjunctive")
```

```
BuildTable(partial3)

DeleteNetwork(tNet)

## End(Not run)
```

Index