Package 'HydeNet'

July 6, 2020

Type Package

Title Hybrid Bayesian Networks Using R and JAGS

Version 0.10.11

Author Jarrod E. Dalton <daltonj@ccf.org> and Benjamin Nutter <benjamin.nutter@gmail.com>

Maintainer Benjamin Nutter <benjamin.nutter@gmail.com>

Description Facilities for easy implementation of hybrid Bayesian networks using R. Bayesian networks are directed acyclic graphs representing joint probability distributions, where each node represents a random variable and each edge represents conditionality. The full joint distribution is therefore factorized as a product of conditional densities, where each node is assumed to be independent of its non-descendents given information on its parent nodes. Since exact, closed-form algorithms are computationally burdensome for inference within hybrid networks that contain a combination of continuous and discrete nodes, particle-based approximation techniques like Markov Chain Monte Carlo are popular. We provide a user-friendly interface to constructing these networks and running inference using the 'rjags' package. Econometric analyses (maximum expected utility under competing policies, value of information) involving decision and utility nodes are also supported.

Depends R (>= 3.0.0), nnet

Imports checkmate, DiagrammeR (>= 0.9.0), plyr, dplyr, graph, magrittr, pixiedust (>= 0.6.1), rjags, stats, stringr, utils

Suggests knitr, RCurl, rmarkdown, survival, testthat

VignetteBuilder knitr

SystemRequirements JAGS (http://mcmc-jags.sourceforge.net)

LazyLoad yes

LazyData true

URL https://github.com/nutterb/HydeNet,

BugReports https://github.com/nutterb/HydeNet/issues

RoxygenNote 7.1.0

NeedsCompilation no

Repository CRAN

Date/Publication 2020-07-06 15:20:13 UTC

R topics documented:

`Hyde-package` *Hydbrid Decision Networks*

Description

Facilities for easy implementation of hybrid Bayesian networks using R. Bayesian networks are directed acyclic graphs representing joint probability distributions, where each node represents a random variable and each edge represents conditionality. The full joint distribution is therefore factorized as a product of conditional densities, where each node is assumed to be independent of its non-descendants given information on its parent nodes. Since exact, closed-form algorithms are computationally burdensome for inference within hybrid networks that contain a combination of continuous and discrete nodes, particle-based approximation techniques like Markov Chain Monte Carlo are popular. We provide a user-friendly interface to constructing these networks and running inference using rjags. Econometric analyses (maximum expected utility under competing policies, value of information) involving decision and utility nodes are also supported.

`bindSim` *Bind Output From coda Samples*

Description

After determining the simulated distributions are satisfactory, it can be advantageous to bind the simulated distributions together in order to aggregate values and perform other manipulations and analyses.

Usage

```
bindSim(hydeSim, relabel_factor = TRUE)

bindPosterior(hydeSim, relabel_factor = TRUE)
```

Arguments

`hydeSim`	An object of class `HydeSim`
`relabel_factor`	Logical. If `TRUE`, factors that had been converted to integers for the JAGS code can be relabelled as factors for additional analysis in R.

Details

For the purposes of this function, it is assumed that if the simulated distributions are satisfactory, the multiple chains in a run can be bound together. Subsequently, the multiple runs are bound together. Lastly, the factors are relabeled, if requested.

Author(s)

Jarrod Dalton and Benjamin Nutter

Examples

```
#' data(PE, package="HydeNet")
Net <- HydeNetwork(~ wells +
                     pe | wells +
                     d.dimer | pregnant*pe +
                     angio | pe +
                     treat | d.dimer*angio +
                     death | pe*treat,
                     data = PE)


compiledNet <- compileJagsModel(Net, n.chains=5)

#* Generate the simulated distribution
Simulated <- HydeSim(compiledNet,
                     variable.names = c("d.dimer", "death"),
                     n.iter=1000)

Bound <- bindSim(Simulated)

#* Bind a Decision Network
#* Note: angio shouldn't really be a decision node.
#*       We use it here for illustration
Net <- setDecisionNodes(Net, angio, treat)
compiledDecision <- compileDecisionModel(Net, n.chains=5)
SimulatedDecision <- HydeSim(compiledDecision,
                             variable.names = c("d.dimer", "death"),
                             n.iter = 1000)
```

`BJDealer` *Blackjack Dealer Outcome Probabilities*

Description

A dataset containing the conditional probability of various dealer outcomes given the "upcard". (The dealer and player each get two cards; only one of the dealer's cards is shown, and this is called the "upcard")

Usage

```
BJDealer
```

Format

A data frame with 70 rows and 3 variables:

dealerUpcard dealer upcard

dealerOutcome outcome of dealer's hand, under the rule that cards are drawn until the dealer's hand total is at least 17

probability conditional probability of dealerOutcome given dealerUpcard ...

Source

https://www.blackjackinfo.com/dealer-outcome-probabilities

`BlackJack` *Black Jack Hybrid Decision Network*

Description

An object of class `HydeNetwork` establishing a graphical model for a game of Black Jack.

Usage

```
BlackJack
```

Format

A `HydeNetwork` object constructed using the code shown in the example. The network has seven random nodes, three ten deterministic nodes, three decision nodes, and one utility node. This is (almost) the same network used in 'vignette("DecisionNetworks", package="HydeNet")'.

Examples

```
## Not run:
BlackJack <-
       HydeNetwork(~ initialAces | card1*card2
                   + initialPoints | card1*card2
                   + highUpcard | dealerUpcard
                   + hit1 | initialPoints*highUpcard
                   + acesAfterCard3 | initialAces*card3
                   + pointsAfterCard3 | card1*card2*card3*acesAfterCard3
                   + hit2 | pointsAfterCard3*highUpcard
                   + acesAfterCard4 | acesAfterCard3*card4
                   + pointsAfterCard4 | card1*card2*card3*card4*acesAfterCard4
                   + hit3 | pointsAfterCard4*highUpcard
                   + acesAfterCard5 | acesAfterCard4*card5
                   + pointsAfterCard5 | card1*card2*card3*card4*card5*acesAfterCard5
                   + playerFinalPoints | initialPoints*hit1*pointsAfterCard3
                   *hit2*pointsAfterCard4*hit3*pointsAfterCard5
                   + dealerFinalPoints | dealerUpcard
```

```
                  + payoff | playerFinalPoints*dealerFinalPoints)
cardProbs  <- c(rep(1/13,8), 4/13, 1/13)  # probs. for 2, 3, ..., 9, (10-K), A

BlackJack <- setNode(BlackJack, card1, nodeType="dcat",
                     pi=vectorProbs(p=cardProbs, card1))
BlackJack <- setNode(BlackJack, card2, nodeType="dcat",
                     pi=vectorProbs(p=cardProbs, card2))
BlackJack <- setNode(BlackJack, card3, nodeType="dcat",
                     pi=vectorProbs(p=cardProbs, card3))
BlackJack <- setNode(BlackJack, card4, nodeType="dcat",
                     pi=vectorProbs(p=cardProbs, card4))
BlackJack <- setNode(BlackJack, card5, nodeType="dcat",
                     pi=vectorProbs(p=cardProbs, card5))

BlackJack <- setNode(BlackJack, dealerUpcard, nodeType="dcat",
               pi=vectorProbs(p=cardProbs, dealerUpcard))

#Note: node dealerFinalPoints will be defined below, following some discussion
#      about its conditional probability distribution.

####################################
# Deterministic Nodes
####################################
BlackJack <- setNode(BlackJack, highUpcard,
               "determ", define=fromFormula(),
               nodeFormula = highUpcard ~ ifelse(dealerUpcard > 8, 1, 0))

BlackJack <- setNode(BlackJack, initialAces,
               "determ", define=fromFormula(),
               nodeFormula = initialAces ~ ifelse(card1==10,1,0) +
                             ifelse(card2==10,1,0))

BlackJack <- setNode(BlackJack, acesAfterCard3,
               "determ", define=fromFormula(),
               nodeFormula = acesAfterCard3 ~ initialAces + ifelse(card3==10,1,0))

BlackJack <- setNode(BlackJack, acesAfterCard4,
               "determ", define=fromFormula(),
               nodeFormula = acesAfterCard4 ~ acesAfterCard3 + ifelse(card4==10,1,0))

BlackJack <- setNode(BlackJack, acesAfterCard5,
               "determ", define=fromFormula(),
               nodeFormula = acesAfterCard5 ~ acesAfterCard4 + ifelse(card5==10,1,0))

BlackJack <- setNode(BlackJack, initialPoints,
               "determ", define=fromFormula(),
               nodeFormula = initialPoints ~ card1+card2+2)

BlackJack <- setNode(BlackJack, pointsAfterCard3, "determ", define=fromFormula(),
               nodeFormula = pointsAfterCard3 ~
                 ifelse(acesAfterCard3 == 3,
                        13,
                        ifelse(acesAfterCard3 == 2,
```

```
                                card1 + card2 + card3 + 3 - 10,
                                ifelse(acesAfterCard3 == 1,
                                       ifelse(card1 + card2 + card3 + 3 > 22,
                                              card1 + card2 + card3 + 3 - 10,
                                              card1 + card2 + card3 + 3),
                                       card1 + card2 + card3 + 3
                                )
                         )
                  )
)

BlackJack <- setNode(BlackJack, pointsAfterCard4, "determ", define=fromFormula(),
                nodeFormula = pointsAfterCard4 ~
                  ifelse(acesAfterCard4 == 4,
                         14,
                         ifelse(acesAfterCard4 == 3,
                                ifelse(card1 + card2 + card3 + card4 + 4 > 38,
                                       card1 + card2 + card3 + card4 + 4 - 30,
                                       card1 + card2 + card3 + card4 + 4 - 20
                                ),
                                ifelse(acesAfterCard4 > 0,
                                       ifelse(card1 + card2 + card3 + card4 + 4 > 22,
                                              card1 + card2 + card3 + card4 + 4 - 10,
                                              card1 + card2 + card3 + card4 + 4
                                       ),
                                       card1 + card2 + card3 + card4 + 4
                                )
                         )
                  )
)

BlackJack <-
  setNode(BlackJack, pointsAfterCard5, "determ", define=fromFormula(),
          nodeFormula = pointsAfterCard5 ~
            ifelse(acesAfterCard5 == 5,
              15,
              ifelse(acesAfterCard5 == 4,
                ifelse(card1 + card2 + card3 + card4 + card5 + 5 > 51,
                  card1 + card2 + card3 + card4 + card5 + 5 - 40,
                  card1 + card2 + card3 + card4 + card5 + 5 - 30
                ),
                ifelse(acesAfterCard5 == 3,
                  ifelse(card1 + card2 + card3 + card4 + card5 + 5 > 51,
                    card1 + card2 + card3 + card4 + card5 + 5 - 30,
                    card1 + card2 + card3 + card4 + card5 + 5 - 20
                  ),
                  ifelse(acesAfterCard5 == 2,
                    ifelse(card1 + card2 + card3 + card4 + card5 + 5 > 31,
                      card1 + card2 + card3 + card4 + card5 + 5 - 20,
                      card1 + card2 + card3 + card4 + card5 + 5 - 10
                    ),
                    ifelse(acesAfterCard5 > 0,
                      ifelse(card1 + card2 + card3 + card4 + card5 + 5 > 22,
```

```
                        card1 + card2 + card3 + card4 + card5 + 5 - 10,
                        card1 + card2 + card3 + card4 + card5 + 5
                      ),
                      card1 + card2 + card3 + card4 + card5 + 5
                    )
                  )
                )
              )
            )
)

BlackJack <- setNode(BlackJack, playerFinalPoints, "determ", define=fromFormula(),
                nodeFormula = playerFinalPoints ~
                  ifelse(hit1 == 0,
                         initialPoints,
                         ifelse(hit2 == 0,
                                pointsAfterCard3,
                                ifelse(hit3 == 0, pointsAfterCard4, pointsAfterCard5)
                         )
                  )
)

BlackJack <- setDecisionNodes(BlackJack, hit1, hit2, hit3)
BlackJack <- setUtilityNodes(BlackJack, payoff)

## End(Not run)
```

`BlackJackTrain` *Black Jack Network Training Dataset*

Description

These are simulated data on 1,000 Black Jack hands.

Usage

```
BlackJackTrain
```

Format

A data frame with 10000 rows and 7 variables:

dealerUpcard The card in the dealer's hand visible to all players

card1 Value of the first card

card2 Value of the second card

initialPoints Total points with the two cards

hit1 Binary variable indicating if a hit was taken

card3 Value of the third card

pointsAfterCard3 Total points with three cards

hit2 Binary variable indicating if a hit was taken

card4 Value of the fourth card

pointsAfterCard4 Total points with four cards

hit3 Binary variable indicating if a hit was taken

card5 Value of the fifth card

pointsAfterCard5 Total points with five cards

Source

Bicycle Cards, "Blackjack," Retrieved from http://www.bicyclecards.com/card-games/rule/blackjack

---

`compileDecisionModel` *Compile JAGS Models to Evaluate the Effect of Decisions in a Network*

Description

Nodes at which a decision can be made, such as the decision to test or not test; treat or not treat; or use open or robotic surgery may impact the outcome for a subject. These types of decisions may not be truly random and understanding how these decisions may impact downstream outcomes may be beneficial to making the decision. Compiling the decision network permits the network to be evaluated under the conditions of each set of decisions separately.

Usage

```
compileDecisionModel(network, policyMatrix = NULL, ..., data = NULL)
```

Arguments

`network`	A HydeNet object with decision nodes defined.
`policyMatrix`	A data frame of policies to apply to decision nodes for comparing networks under different conditions. See `policyMatrix`.
`...`	Additional arguments to pass to `jags.model`, excepting the `data` argument. The `data` argument is created by `compileDecisionModel`, and cannot be passed manually.
`data`	An optional list of data values to be observed in the nodes. It is passed to the `data` argument of `rjags::jags`. Any values given in `data` will override values provided in `policyMatrix` with a warning.

Details

compileDecisionModel only accepts nodes of type "dbern" (Bernoulli random variable taking either 0 or 1) or "dcat" (categorical variables) as decision nodes. When the node is type "dcat", the decision options are extracted from the JAGS statement returned by writeJagsModel.

The options for each decision node (if there are multiple nodes) are combined via expand.grid to make a table of all possible decisions. Each row of this table is passed as a list to the data argument of jags.model (via compileJagsModel) and a list of JAGS model objects is returned. coda.samples may be run on each of these models.

Value

Returns a list of compiledHydeNetwork objects.

Author(s)

Jarrod Dalton and Benjamin Nutter

See Also

policyMatrix compileJagsModel

Examples

```
data(PE, package="HydeNet")
Net <- HydeNetwork(~ wells +
                     pe | wells +
                     d.dimer | pregnant*pe +
                     angio | pe +
                     treat | d.dimer*angio +
                     death | pe*treat,
                     data = PE)



Net <- setDecisionNodes(Net, treat)
plot(Net)

decision1 <- compileDecisionModel(Net)

#* An effectively equivalent call as the previous
decision2 <- compileDecisionModel(Net, policyMatrix(Net))

#* Using a customized policy matrix
#* Note: this is a bit of nonsense--you can't decide if a test is negative
#*       or positive, but we'll do this for illustration.
custom_policy <- policyMatrix(Net,
                              treat="No",
                              angio = c("Negative", "Positive"))
decision3 <- compileDecisionModel(Net, custom_policy)
```

compileJagsModel *Compile Jags Model from a Hyde Network*

Description

Generates the JAGS code from the Hyde network and uses it to create an object representing a Bayesian graphical model.

Usage

```
compileJagsModel(network, data = NULL, ...)
```

Arguments

network	An object of class HydeNetwork
data	A list of data values to be observed in the nodes. It is passed to the data argument of rjags::jags. Alternatively, a data frame representing a policy matrix may be provided to compile multiple JAGS models.
...	Additional arguments to be passed to jags.model

Details

compileJagsModel is a partial wrapper for jags.model. Running compileJagsModel(network) is equivalent to running jags.model(textConnection(writeNetworkModel(network))).

Value

Returns a compiledHydeNetwork object. The jags element of this object is suitable to pass to coda.samples. Otherwise, the primary function of the object is plotting the network with observed data shown.

Author(s)

Benjamin Nutter

See Also

jags.model

Examples

```
data(PE, package="HydeNet")
Net <- HydeNetwork(~ wells +
                     pe | wells +
                     d.dimer | pregnant*pe +
                     angio | pe +
                     treat | d.dimer*angio +
                     death | pe*treat,
```

```
                    data = PE)


compiledNet <- compileJagsModel(Net, n.chains=5)

#* Generate the posterior distribution
Posterior <- HydeSim(compiledNet,
                    variable.names = c("d.dimer", "death"),
                    n.iter = 1000)
Posterior

#* For a single model (ie, not a decision model), the user may choose to
#* use the \code{rjags} function \code{coda.samples}.
#* However, this does not have a succinct print method
library(rjags)
s <- coda.samples(compiledNet$jags,
                  variable.names = c("d.dimer", "death"),
                  n.iter=1000)
```

`cpt` *Compute a conditional probability table for a factor given other factors*

Description

The function `cpt` operates on sets of factors. Specifically, it computes the conditional probability distribution of one of the factors given other factors, and stores the result in a multidimensional `array`.

`inputCPT()` is a utility function aimed at facilitating the process of populating small conditional probability distributions, i.e., those for which the response variable doesn't have too many levels, there are relatively few independent variables, and the independent variables also don't have too many levels.

Usage

```
cpt(x, data, wt, ...)

## S3 method for class 'formula'
cpt(formula, data, wt, ...)

## S3 method for class 'list'
cpt(x, data, wt, ...)

inputCPT(x, factorLevels, reduce = TRUE, ...)

## S3 method for class 'formula'
inputCPT(formula, factorLevels, reduce = TRUE, ...)
```

```
## S3 method for class 'list'
inputCPT(x, factorLevels, reduce = TRUE, ...)
```

Arguments

x	a list containing the names of the variables used to compute the conditional probability table. See details.
data	a data frame containing all the factors represented by the formula parameter.
wt	(optional) a numeric vector of observation weights.
...	Additional arguments to be passed to other methods.
formula	a formula specifying the relationship between the dependent and independent variables.
factorLevels	(optional) a named list with the following structure: Variable names for the factors specified in vars comprise the names of the list elements, and each list element is a character vector containing the levels of the respective factor. See examples.
reduce	set to TRUE if inputCPT() is to compute probabilities for the first level of the dependent variable as the complement of the inputted probabilities corresponding to the other levels of the dependent variable. For example, reduce = TRUE with a binary dependent variable y (say, with levels 'no' and 'yes') will ask for the probabilities of 'yes' at each combination of the independent variables, and compute the probability of 'no' as their respective complements. See details.

Details

If a formula object is entered for the vars parameter, the formula must have the following structure: *response ~ var1 + var2 + etc.*. The other option is to pass a named list containing two elements y and x. Element y is a character string containing the name of the factor variable in data to be used as the dependent variable, and element x is a character vector containing the name(s) of the factor variable(s) to be used as independent (or conditioning) variables.

In inputCPT(), when the parameter reduce is set to FALSE, any non-negative number (e.g., cell counts) is accepted as input. Conditional probabilities are then calculated via a normalization procedure. However, when reduce is set to TRUE, a) only probabilities in [0,1] are accepted and b) all inputted probabilities for each specific combination of independent variable values must not sum to a value greater than 1 (or the calculated probability for the first level of the dependent variable would be negative).

The cpt() function with a weight vector passed to parameter wt works analogously to inputCPT(reduce = FALSE), i.e., it accepts any non-negative vector, and computes the conditional probability array by normalizing sums of weights.

Author(s)

Jarrod Dalton and Benjamin Nutter

Examples

```
# a very imbalanced dice example

n <- 50000
data <- data.frame(
  di1 = as.factor(1:6 %*% rmultinom(n,1,prob=c(.4,.3,.15,.10,.03,.02))),
  di2 = as.factor(1:6 %*% rmultinom(n,1,prob=rev(c(.4,.3,.15,.10,.03,.02)))),
  di3 = as.factor(1:6 %*% rmultinom(n,1,prob=c(.15,.10,.02,.3,.4,.03)))
)

cpt1 <- cpt(di3 ~ di1 + di2, data)
cpt1[di1 = 1, di2 = 4, ]  # Pr(di3 | di1 = 1, di2 = 4)
cpt1["1","4",]
cpt1[1,4,]

plyr::aaply(cpt1, c(1,2), sum) # card(di1)*card(di2) matrix of ones

l <- list(y = "di3", x = c("di1","di2"))
all(cpt(l, data) == cpt1)

## Not run:
inputCPT(wetGrass ~ rain + morning)

inputCPT(wetGrass ~ rain + morning,
         factorLevels <- list(wetGrass = c("dry","moist","VeryWet"),
                              rain     = c("nope","yep"),
                              morning  = c("NO","YES")),
         reduce = FALSE)

## End(Not run)
```

`expectedVariables` *List Expected Parameter Names and Expected Variables Names*

Description

To assist in formula that defines the relationship to a node, `expectedVariables` returns to the console a sample string that can be pasted into `setNode` and populated with the desired coefficients.

Usage

```
expectedVariables(network, node, returnVector = FALSE)

expectedParameters(network, node, returnVector = FALSE)
```

Arguments

network	A HydeNetwork object.
node	A node name within network
returnVector	Logical. If FALSE, the sample string for use in setNode is returned. Otherwise, the vector of parent names is returned.

Details

Each node is calculated as a model of its parents. If no training data are provided to the network, the user is expected to provide appropriate estimates of the regression coefficients for the model.

returnVector will generally be set to FALSE for most uses, but can be set to TRUE for use in error checking. For example, in setNode, if not all of the parents have been given a coefficient (or if too few coefficients have been given), the vector of names is supplied.

Author(s)

Jarrod Dalton and Benjamin Nutter

Examples

```
data(PE, package="HydeNet")
Net <- HydeNetwork(~ wells +
                     pe | wells +
                     d.dimer | pregnant*pe +
                     angio | pe +
                     treat | d.dimer*angio +
                     death | pe*treat)

expectedVariables(Net, wells)
expectedVariables(Net, treat)
expectedVariables(Net, treat, returnVector=TRUE)

expectedParameters(Net, wells)
expectedParameters(Net, wells, returnVector=TRUE)
```

---

factorFormula *Convert Factor Levels in Formula to Numeric Values*

Description

When working in R, it is often more convenient to work in terms of the factor labels rather than the underlying numeric values. JAGS, however, requires that the numeric values be used. factorFormula permits the user to define formulae to be passed to JAGS using R style coding, and having factor levels translated to the underlying values as determined by the network structure.

Usage

```
factorFormula(form, network)
```

Arguments

form	A formula object.
network	A HydeNetwork object.

Details

It is assumed that factor variables will be used in logical comparisons of the format [variable_name] == '[factor_level]' and only this pattern is recognized in the text search. Single or double quotes may be used around the level, and the spaces around the == are optional.

Author(s)

Jarrod Dalton and Benjamin Nutter

Examples

```
## Not run:
Net <- HydeNetwork(~ wells +
                     pe | wells +
                     d.dimer | pregnant*pe +
                     angio | pe +
                     treat | d.dimer*angio +
                     death | pe*treat,
                   data = PE)
factorFormula(form = payoff ~ (death == 'No') + (pe == 'Yes'),
              network = Net)

## End(Not run)
```

---

factorRegex *Produce Regular Expressions for Extracting Factor Names and Levels*

Description

A utility function to produce a regular expression that can separate factor names and factor levels in the broom::tidy()$term output.

Usage

```
factorRegex(fit)
```

Arguments

fit	a model object

Author(s)

Jarrod Dalton and Benjamin Nutter

Examples

```
data(PE, package = "HydeNet")
g6 <- glm(treat ~ d.dimer + angio, data=PE, family="binomial")
HydeNet:::factorRegex(g6)
```

HydeNetSummaries *HydeNet Summary Objects*

Description

Summaries of HydeNetwork, compiled network, and compiled decision network objects.

Usage

```
## S3 method for class 'HydeNetwork'
summary(object, ...)
```

Arguments

object	A HydeNet object to be summarized
...	Additional arguments.

Author(s)

Jarrod Dalton and Benjamin Nutter

HydeNetwork *Define a Probablistic Graphical Network*

Description

Using either a directed acyclic graph (DAG) or a list of models, define a probabilistic graphical network to serve as the basis of building a model.

Usage

```
HydeNetwork(nodes, ...)

## S3 method for class 'formula'
HydeNetwork(nodes, data = NULL, ...)

## S3 method for class 'list'
HydeNetwork(nodes, ...)
```

Arguments

nodes	Either a formula that defines the network or a list of model objects.
...	additional arguments to other methods. Not currently used.
data	A data frame with the data for estimating node parameters.

Details

The DAG becomes only one element of the object returned by HydeNetwork. The dag object is used to extract the node names and a list of parents for each node. These will be used to help quantify the relationships.

When given a formula, the relationships are defined, but are not quantified until writeNetworkModel is called.

When a list of models is given, rather than refitting models when writeNetworkModel is called, the quantified relationships are placed into the object.

Value

Returns an object of class HydeNetwork. The object is really just a list with the following components:

- nodes a vector of node names
- parents a named list with each element being a vector of parents for the node named.
- nodeType a named list with each element specifying the JAGS distribution type.
- nodeFormula a named list with the formulae specifying the relationships between nodes.
- nodeFitter a named list giving the fitting function for each node.
- nodeFitterArgs A named list with additional arguments to be passed to fitter functions.
- nodeParams A named list. Each element is a vector of parameters that will be expected by JAGS.
- fromData A named list with the logical value of whether parameters should be estimated from the data.
- nodeData A named list with the data for each node. If a node's entry in fromData is TRUE and nodeData is NULL, it will look to the data attribute instead.
- factorLevels If the vector associated with the node is a factor (or character), the levels of the factor are stored here. Although it may seem redundant, it allows factor levels to be specified in cases where the node is not define with data. If data are provided to the node, this element is determined from the data and cannot be manually overwritten.
- nodeModel A list of model objects. This is a storing place for models that have already been fit so that they don't have to be refit again.
- nodeDecision A named list of logical flags for whether the node is a decision node or not.
- nodeUtility A named list of logical flags for whether the node is a utility node or not.
- dag The adjacency matrix defining the network. Most of the plotting utilities will be based on this element.
- data A common data frame for nodes that do not have their own unique data source.

- network_formula The original formula passed to construct the model.

@note These objects can get pretty large. In versions of R earlier than 3.2, it can take a while to print the large network objects if you simply type the object name into the console. It is recommended that you always explicitly invoke the 'print' function (ie, print(Net) instead of just Net) to save yourself some valuable time.

Author(s)

Jarrod Dalton and Benjamin Nutter

Examples

```
#* Formula Input
Net <- HydeNetwork(~ wells +
                     pe | wells +
                     d.dimer | pregnant*pe +
                     angio | pe +
                     treat | d.dimer*angio +
                     death | pe*treat,
                     data = PE)
print(Net)

#* Model Input
g1 <- lm(wells ~ 1, data=PE)
g2 <- glm(pe ~ wells, data=PE, family="binomial")
g3 <- lm(d.dimer ~ pe + pregnant, data=PE)
g4 <- xtabs(~ pregnant, data=PE)
g5 <- glm(angio ~ pe, data=PE, family="binomial")
g6 <- glm(treat ~ d.dimer + angio, data=PE, family="binomial")
g7 <- glm(death ~ pe + treat, data=PE, family="binomial")

bagOfModels <- list(g1,g2,g3,g4,g5,g6,g7)

bagNet <- HydeNetwork(bagOfModels)
print(bagNet)
```

HydeSim *Simulated Distributions of a Decision Network*

Description

The simulated distributions of the decision network can be evaluated to determine the probabilistic outcomes based on the decision inputs in the model as well as subject specific factors.

Usage

```
HydeSim(
  cHN,
  variable.names,
  n.iter,
  thin = 1,
  ...,
  monitor_observed = TRUE,
  bind = TRUE
)

HydePosterior(...)
```

Arguments

cHN	A compiledHydeNetwork object as returned by compileJagsNetwork.
variable.names	a character vector giving the names of variables to be monitored.
n.iter	number of iterations to monitor.
thin	thinning interval for monitors.
...	options arguments that are passed to the update method for jags model objects.
monitor_observed	If TRUE, the observed or fixed variables (those passed to the data argument in compileJagsNetwork) are forced into variable.names if not already provided. This is recommended, especially if you will be binding multiple JAGS runs together.
bind	Logical. If TRUE, simulated distributions will be bound into a single data frame. If FALSE, the standard output from rjags is returned.

Details

This is essentially a wrapper around coda.samples that returns in a list the output for each run of coda.samples over the rows of the policy/decision matrix given in the data argument of compileJagsNetwork.

Value

A list of class HydeSim with elements codas (the MCMC matrices from coda.samples), observed (the values of the variables that were observed), dag (the dag object for convenience in displaying the network), and factorRef (giving the mappings of factor levels to factor variables).

Author(s)

Jarrod Dalton and Benjamin Nutter

Examples

```
data(PE, package="HydeNet")
Net <- HydeNetwork(~ wells +
```

```
                    pe | wells +
                    d.dimer | pregnant*pe +
                    angio | pe +
                    treat | d.dimer*angio +
                    death | pe*treat,
                    data = PE)


compiledNet <- compileJagsModel(Net, n.chains=5)

#* Generate the posterior distribution
Posterior <- HydeSim(compiledNet,
                          variable.names = c("d.dimer", "death"),
                          n.iter = 1000)

#* Posterior Distributions for a Decision Model
Net <- setDecisionNodes(Net, angio, treat)
decisionNet <- compileDecisionModel(Net, n.chains=5)
decisionsPost <- HydeSim(decisionNet,
                             variable.names = c("d.dimer", "death"),
                             n.iter = 1000)
```

HydeUtilities *Hyde Network Utility Functions*

Description

The functions described below are unexported functions that are used internally by HydeNet to prepare and modify network objects and prepare JAGS code.

Usage

```
termName(term, reg)

decisionOptions(node, network)

makeJagsReady(mdl, factorRef, bern)

matchLevelNumber(t, lev)

matchVars(terms, vnames)

nodeFromFunction(node_fn)

policyMatrixValues(node, network)

polyToPow(poly)
```

```
validateParameters(params, dist)

makeFactorRef(network)

dataframeFactors(dataframe)

factor_reference(data)
```

Arguments

term	Usually the term column from the output of broom::tidy()
reg	A regular expression, usually provided by factorRegex
node	Character string indicating a node in a network
network	A Hyde Network Object
mdl	Output from broom::tidy()
factorRef	A list of data frames mapping factors to levels
bern	bernoulli node names.
t	Usually the term_name column generated within makeJagsReady
lev	usually the level_value column generated within makeJagsReady
terms	A vector of term names, usually from a broom::tidy object.
vnames	A vector of term names, usually from network$nodes.
node_fn	A character string representing a function passed in a model formula
poly	A single term for which the polynomial components should be converted to the JAGS pow() function.
params	The list of parameters given in the ... argument of setNode
dist	The JAGS distribution function name. Appropriate names are in the FnName column of the jagsDists data object.
dataframe	A data frame. The data frame will be searched for factors and a reference table (another data frame) is returned.
data	A data frame.

Details

termName: In most model objects, factors in the model are represented as [variable][level]. This utility function pulls out the [variable] component. This is typically called from within makeJagsReady.

decisionOptions: When compiling multiple JAGS models to evaluate the effect of decision nodes, the options for each decision node are extracted by this utility.

makeJagsReady: Manages the presence of factors in interactions and makes sure the proper numeric factor value is given to the JAGS code. This is called from within a writeJagsFormula call.

matchLevelNumber: Assigns the correct numeric value of a level to a factor variable in a model. This is called from within makeJagsReady.

matchVars: Given a list of existing node names, the terms of a formula are matched to the node names. This allows functions to be used in formula defintions. Most commonly, factor(var) would get reduced to var, which is a node name that JAGS will understand. There is still limited ability for translation here, and matchVars assumes that the longest match is the desired match. If you pass a function with two node names, the longer of the two will be taken and JAGS will likely fail.

nodeFromFunction: This is a utility function necessary to make modelToNode work properly. A node vector was needed to pass to matchVars, and this is the mechanism to generate that vector.

polyToPow: converts polynomials generated by the poly function to use the pow function in JAGS.

validateParameters: Users may pass parameters to the JAGS code using the setNode function. If a faulty parameter is given, such as lambda = -10 for a poisson distribution (lambda must be positive in a Poisson distribution), the error returned by JAGS may not clear enough to diagnose a problem that presented several steps earlier in the code. validateParamters allows the user to receive instant feedback on the appropriateness of the code.

Logical expressions for comparisons are stored in the jagsDists data object (data(jagsDists,package='Hyde')). This is a utility function used only within setNode and is not visible to the user.

Author(s)

Jarrod Dalton and Benjamin Nutter

---

inputCPTExample *Example Conditional Probability Table Resulting from the* inputCPT *function.*

Description

This is an example of the output generated by the inputCPT function as illustrated in the article being submitted to JSS. It is saved as an object named h in the article.

Usage

```
inputCPTExample
```

Format

An object of class cpt (inherits from array) with 2 rows and 2 columns.

Source

No Source. It's really just made up.

jagsDists *JAGS Probability Distributions.*

Description

A dataset listing the JAGS probability distributions and their parameters

Usage

```
jagsDists
```

Format

A data frame with 30 rows and 7 variables:

DistName Distribution Name

FnName JAGS Function Name

FnNameR R Function Name

xLow Minimum value for x, the random variable

xHigh Maximum value for x, the random variable

Parameters Names of the JAGS parameters

RParameter R function argument name

paramLimit Limits on the parameter

paramLogic The text of a logical check used in `setNode` to ensure stated parameters are valid.

Rsupport Logical value, indicating if an R equivalent is supported by `HydeNet`

Source

http://people.stat.sc.edu/hansont/stat740/jags_user_manual.pdf

jagsFunctions *JAGS Functions Compatible with R.*

Description

A dataset listing the JAGS functions and their R equivalents.

Usage

```
jagsFunctions
```

Format

A data frame with 30 rows and 3 variables:

jags_function JAGS function name

r_function R function Name

r_package R package where the function is found.

Source

http://people.stat.sc.edu/hansont/stat740/jags_user_manual.pdf

---

mergeDefaultPlotOpts *rdname plot.HydeNetwork*

Description

rdname plot.HydeNetwork

Usage

```
mergeDefaultPlotOpts(network, node_df)
```

Arguments

network	a HydeNetwork object
node_df	A data frame of node attributes.

---

modelToNode *Convert a Model Object to a Network Node*

Description

In cases where model objects may already be fit and established, they can be used to generate a network without having to refit the models or specify the distributions in setNode.

For models built with xtabs, although a data frame may be passed when building the model, it is not stored in the object. Thus, the data used to construct the models are not carried with the node.

Usage

```
modelToNode(model, nodes, ...)

## Default S3 method:
modelToNode(model, nodes, ...)

## S3 method for class 'cpt'
modelToNode(model, nodes, ...)

## S3 method for class 'glm'
modelToNode(model, nodes, ...)

## S3 method for class 'lm'
modelToNode(model, nodes, ...)

## S3 method for class 'multinom'
modelToNode(model, nodes, ...)

## S3 method for class 'survreg'
modelToNode(model, nodes, ...)

## S3 method for class 'xtabs'
modelToNode(model, nodes, ...)
```

Arguments

model	A model object
nodes	A vector of node names, usually as network$nodes
...	Additional arguments to be passed to other functions. Currently ignored.

PE *Pulmonary Embolism Dataset*

Description

These are simulated data on 10,000 cases with suspected pulmonary embolism at a hospital.

Usage

```
PE
```

Format

A data frame with 10000 rows and 7 variables:

wells Wells score (integer ranging from 1 to 10 indicating the degree to which PE is suspected based on clinical review of symptoms)

pregnant Factor indicating pregnancy (No, Yes)

pe Factor indicating pulmonary embolism has occurred (No,Yes)

angio Result of pulmonary angiography test (Negative, Positive)

d.dimer Numeric result of diagnostic blood test called D-Dimer.

treat Factor indicating whether or not treatment for PE was administered (No,Yes)

death Factor indicating patient mortality (No,Yes)

Source

Simulated data - not from real patients.

`plot.HydeNetwork` *Plotting Utilities for Probabilistic Graphical Network*

Description

Generate and customize plots of a `HydeNetwork` class network. `HydeNet` provides some initial defaults for standard variable nodes, deterministic nodes, decision nodes, and utility nodes. Since these nodes are assumed to be of inherent difference and interest, the options are defined in a way to make these easier to identify in a plot. The default options may be altered by the user to their liking by invoking `HydePlotOptions`. Node attributes are more fully explained in the documentation for the `DiagrammeR` package. Individual nodes may be define with `customNode`.

Usage

```
## S3 method for class 'HydeNetwork'
plot(
  x,
  customNodes = NULL,
  customEdges = NULL,
  ...,
  removeDeterm = FALSE,
  useHydeDefaults = TRUE
)

mergeCustomNodes(node_df, customNodes)

mapEdges(n, p, node_df)

mergeCustomEdges(edge_df, customEdges)

customNode(node_id, ...)

HydePlotOptions(
  variable = NULL,
  determ = NULL,
```

```
  decision = NULL,
  utility = NULL,
  restorePackageDefaults = FALSE
)
```

Arguments

x	an object of class HydeNetwork
customNodes	a data frame giving additional specifications for nodes. The customizations provided here will override the default settings.
customEdges	a data frame giving custom settings for edges (arrows) between nodes.
...	for the plot method, additional arguments to be passed to DiagrammeR::render_graph. For customNode, named node attributes to assign to a node's plotting characteristics.
removeDeterm	A logical value. When FALSE (the default), it has no effect. When TRUE, deterministic nodes are removed from the network and a reduced plot with no deterministic nodes is rendered.
useHydeDefaults	A logical value indicating if the default plot parameters in options("Hyde_plotOptions") should be applied to the plot.
node_df	A data frame of node attributes
n	node names from a network object
p	the list of parents from a network object
edge_df	The default edge attribute data frame
node_id	The name of a node in a HydeNetwork object. May be quoted or unquoted.
variable, determ, decision, utility	Named lists of attributes to use as defaults node attribute settings for each variable type.
restorePackageDefaults	A logical value. When TRUE, the original package defaults are restored.

Details

GraphViz is an enormous set of resources for customizing and we cannot adequately describe them all here. See 'Sources' for links to additional documentation from the DiagrammeR package and the GraphViz website.

With its default settings, HydeNet makes use of five node attributes for plotting networks. These are

- style: By default, set to 'filled', but may also take 'striped', 'wedged', or 'radial'.
- fillcolor: The hexadecimal or X11 color name. In styles 'striped', 'wedged', or 'radial', this may take multiple colors separated by a colon (:).
- shape: the node shape. May take the values 'oval', 'diamond', 'egg', 'ellipse', 'square', 'triangle', or 'rect'
- fontcolor: The color of the node label.
- color: The color of the node's border.

HydeNet assumes the GraphViz defaults for edge nodes (arrows).

See the Plotting Hyde Networks vignette (vignette("HydeNetPlots")) for a more thorough explanation of plotting networks.

Author(s)

Jarrod Dalton and Benjamin Nutter

Source

http://rich-iannone.github.io/DiagrammeR/graphviz_and_mermaid.html
See especially the section on Attributes

http://graphviz.org/

Examples

```
## Not run:
#* Plots may open in a browser.
data(BlackJack, package="HydeNet")
plot(BlackJack)

HydePlotOptions(variable=list(shape = "rect", fillcolor = "#A6DBA0"),
                determ = list(shape = "rect", fillcolor = "#E7D4E8",
                              fontcolor = "#1B7837", linecolor = "#1B7837"),
                decision = list(shape = "triangle", fillcolor = "#1B7837",
                                linecolor = "white"),
                utility = list(shape = "circle", fillcolor = "#762A83",
                               fontcolor = "white"))
plot(BlackJack)

HydePlotOptions(restorePackageDefaults = TRUE)

plot(BlackJack,
     customNodes = customNode(payoff,
                              fillcolor = "purple", shape = "circle",
                              fontcolor = "white", height = "2",
                              style="filled"))
plot(BlackJack,
  customNodes =
    dplyr::bind_rows(
      customNode(pointsAfterCard3,
                 shape = "circle",
                 style = "radial",
                 fillcolor = "#1B7837:#762A83",
                 fontcolor = "black",
                 height = "2"),
      customNode(playerFinalPoints,
                 shape = "circle",
                 style = "wedged",
                 height = "3",
                 fillcolor = c("red:orange:yellow:green:blue:purple"))))
```

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

`policyMatrix` *Construct Policy and Decision Matrices*

Description

It may be of interest to compare posterior distributions of variables dependent on differing levels of the decision nodes. For example, how might choosing a different engine in a car affect the fuel consumption. `policyMatrix` provides a quick utility to begin defining the policy matrix on which decisions can be made.

Usage

```
policyMatrix(network, ..., useDefaultPolicies = TRUE)

defaultPolicyMatrix(network)
```

Arguments

`network`	A HydeNetwork object
`...`	Named arguments with vectors of the policy values.
`useDefaultPolicies`	A logical indicating if the default policy values from decision nodes should be used for any decision nodes not named in `....` If a node is named in `...` and also has default policy values, the user supplied values take precedence.

Details

When `...` is not used, the default policy matrix is defined as all possible combinations of the levels in the network's decision nodes. If no decision nodes are defined, an error is returned. Note that the default policy matrix returns JAGS-ready values, which are numeric according to the level number of a factor. In user-defined matrices, character values are supported and will be converted to numerics when the JAGS model is compiled.

Semi-custom policy matrices can be defined by providing the values of each node to be considered. When manually supplying values, the nodes must exist in network, but the requirement that they be decision nodes is not enforced. Thus, it is possible to include numeric values in a decision matrix.

Policy matrices can be passed to `HydeSim` to run posterior distributions on each row of the policy matrix. There is nothing particularly special about the policy matrices returned by `policyMatrix`; they are simply data frame that require names drawn from the nodes in the network. Any data frame can be passed to `HydeSim` and a check is done there to confirm all of the column names match a node in the network.

Whenever a node is identified as a deterministic node, its policy values are forced to `NULL`, regardless of what the user has specified.

Value

Returns a data frame built by `expand.grid` and intended to be used with `HydeSim`.

Author(s)

Jarrod Dalton and Benjamin Nutter

Examples

```
data(PE, package="HydeNet")
Net <- HydeNetwork(~ wells +
                     pe | wells +
                     d.dimer | pregnant*pe +
                     angio | pe +
                     treat | d.dimer*angio +
                     death | pe*treat,
                     data = PE)



Net <- setDecisionNodes(Net, angio, treat)
plot(Net)

policyMatrix(Net)

policyMatrix(Net, treat="No", angio = c("No", "Yes"))
```

---

print.cpt *Print Method for CPT objects*

Description

Just a wrapper to strip the attributes off, change the class, and print the array.

Usage

```
## S3 method for class 'cpt'
print(x, ...)
```

Arguments

`x`	Object of class `cpt`
`...`	Additional arguments to be passed to other methods.

Author(s)

Jarrod Dalton and Benjamin Nutter

`print.HydeNetwork` *Print a HydeNetwork*

Description

Prints a HydeNetwork object with a brief summary of each node.

Usage

```
## S3 method for class 'HydeNetwork'
print(x, ...)
```

Arguments

`x`	a `HydeNetwork object`
`...`	additional arguments to be passed to print methods. Currently none in use.

Details

The summary of each node follows the format
node name | parents
node type (parameter)
estimated from data (or not)
formula

Author(s)

Jarrod Dalton and Benjamin Nutter

Examples

```
data(PE, package="HydeNet")
Net <- HydeNetwork(~ wells +
                     pe | wells +
                     d.dimer | pregnant*pe +
                     angio | pe +
                     treat | d.dimer*angio +
                     death | pe*treat)
print(Net)
print(Net, d.dimer)

Net <- setNode(Net, d.dimer,
                  nodeType='dnorm', mean=fromData(), sd=fromData(),
                  nodeFormula = d.dimer ~ pregnant + pe,
                  nodeFitter='lm')
print(Net, d.dimer)
```

print.HydeSim *Print a Hyde Simulated Distribution Object*

Description

Prints a brief description of a HydeSim object.

Usage

```
## S3 method for class 'HydeSim'
print(x, ...)
```

Arguments

x	a HydeSim object
...	additional arguments to be passed to print methods. Currently none in use.

Details

Prints the number of posterior distributions, chains, and iterations, as well as the observed values.

Author(s)

Jarrod Dalton and Benjamin Nutter

Examples

```
data(PE, package="HydeNet")
Net <- HydeNetwork(~ wells +
                     pe | wells +
                     d.dimer | pregnant*pe +
                     angio | pe +
                     treat | d.dimer*angio +
                     death | pe*treat,
                     data = PE)

Net <- setDecisionNodes(Net, treat)

compiledNet <- compileJagsModel(Net, n.chains=5)

#* Generate the posterior distribution for the model (but not the
#* decision model)
Posterior <- HydeSim(compiledNet,
                           variable.names = c("d.dimer", "death"),
                           n.iter = 1000)
Posterior

#* Generate the posterior for the decision model
Decision <- compileDecisionModel(Net, n.chains=5)
```

```
Posterior_decision <- HydeSim(Decision,
                                    variable.names = c("d.dimer", "death"),
                                    n.iter = 1000)
```

`Resolution.cpt` *Conditional Probability Table for resolution of side effects as a function drugs and emesis.*

Description

This is a conditional probability table used in the emesis example of the JSS article.

Usage

```
Resolution.cpt
```

Format

An object of class `cpt` (inherits from `array`) of dimension 2 x 2 x 2 x 2 x 3.

`rewriteHydeFormula` *Rewrite HydeNetwork Graph Model Formula*

Description

This is a convenience function used to assist in the updating of `HydeNetwork` network objects. It makes it possible to add and subtract individual parent relationships without deleting an entire node.

Usage

```
rewriteHydeFormula(old_form, new_form)
```

Arguments

`old_form`	The current formula in a `HydeNetwork` object.
`new_form`	The formula specifications to be added

Details

To allow changes to be made on the node-parent level, the formulae are broken down into a vector of component where each element identifies a unique parent-child relationship. So if a node has representation `nodeA | nodeB*nodeC`, it will be broken down to `nodeA | nodeB + nodeA | nodeC`.

After decomposing the formulae, any instances of a component in `form1` that are subtracted in `form2` are removed.

Next, all added components in `form2` that do not already exist in `form1` are added.

Lastly, the parents of each node are combined and the specification of the network is written.

Author(s)

Jarrod Dalton and Benjamin Nutter

`SE.cpt` *Conditional Probability Table for side effects as a function of emesis and drug.*

Description

This is a conditional probability table used in the emesis example of the JSS article.

Usage

```
SE.cpt
```

Format

An object of class `cpt` (inherits from `array`) of dimension 2 x 2 x 2.

`setDecisionNodes` *Classify Multiple Nodes as Decision or Utility Nodes*

Description

Depending on how your Hyde Network was built, you may not have had the opportunity to declare which nodes are decision or utlity nodes. For instance, when passing a list of models, there is no way to indicate in the model object that the node should be considered a decision node. As a matter of convenience, these function will set any nodes indicated to decision or utility nodes. It will make no other modifications to a node's definition.

Usage

```
setDecisionNodes(network, ...)

setUtilityNodes(network, ...)
```

Arguments

`network`	A Hyde Network object
`...`	Networks to be classified as decision nodes. These may be quoted or unquoted.

Author(s)

Jarrod Dalton and Benjamin Nutter

setNode *Set Node Relationships*

Description

The relationship between a node and its parents must be defined before the appropriate JAGS model statement can be constructed. setNode is the utility by which a user can define the distribution of the node and its relationship to its parents (usually through a model of some sort).

Usage

```
setNode(
  network,
  node,
  nodeType,
  nodeFitter,
  nodeFormula,
  fitterArgs = list(),
  decision = "current",
  utility = "current",
  fromData = !is.null(network$data),
  ...,
  nodeData = NULL,
  factorLevels = NULL,
  validate = TRUE,
  fitModel = getOption("Hyde_fitModel"),
  policyValues = factorLevels
)

fromData()

fromFormula()
```

Arguments

network	A HydeNetwork.
node	A node within network. This does not have to be quoted.
nodeType	a valid distribution. See the data set in data(jagsDists) for a complete list of available distributions. See "Choosing a Node Type"
nodeFitter	the fitting function, such as lm or glm. This will probably only be needed when fromData = TRUE.
nodeFormula	A formula object specifying the relationship between a node and its parents. It must use as a term every parent of node. This formula will be pushed through the unexported function factorFormula. See "Coding Factor Levels" for more details.

fitterArgs	Additional arguments to be passed to fitter.
decision	A value of either "current" or a logical value. If "current", the current value of the setting is retained. This allows decision nodes set by setDecisionNode to retain the classification as a decision node if setNode is run after setDecisionNode. If TRUE, the node will be considered a decision node in compileDecisionNetwork. This is only a valid option when the node is of type "dbern" or "dcat". Note: if any character value other than "current" is given, setNode will assume you intended "current".
utility	A value of either "current" or a logical value. If "current", the current value of the setting is retained. This allows utility nodes set by setUtilityNode to retain the classification as a utility node if setNode is run after setUtilityNode. If TRUE, the node will be considered a utility node. This is only a valid option when the node is of type "determ" and it has no children. Note: if any character value other than "current" is given, setNode will assume you intended "current".
fromData	Logical. Determines if a node's relationship is calculated from the data object in network. Defaults to TRUE whenever network has a data object.
...	parameters to be passed to the JAGS distribution function. Each parameter in the distribution function must be named. For example, the parameters to pass to dnorm would be mean='',sd=''. The required parameters can be looked up using the expectedParameters function. If parameters are to be estimated from the data, the functions fromData and fromFormula may be used as placeholders.
nodeData	A data frame with the appropriate data to fit the model for the node. Data passed in this argument are applied only to this specific node. No checks are performed to ensure that all of the appropriate variables (the node and its parents) are included.
factorLevels	A character vector used to specify the levels of factors when data are not provided for a node. The order of factors follows the order provided by the user. This argument is only used when the node type is either dcat or dbern, the node Fitter is not cpt, nodeData is NULL, and no variable for the node exists in the network's data element. If any of those conditions is not met, factorLevels is ignored. This proves particularly important when data are specified in order to prevent a user specification from conflicting with expected factors across nodes.
validate	Logical. Toggles validation of parameters given in .... When passing raw JAGS code (ie, character strings), this will be ignored (with a message), as the validation is applicable to numerical/formula values.
fitModel	Logical. Toggles if the model is fit within the function call. This may be set globally using options('Hyde_fitModel'). See Details for more about when to use this option.
policyValues	A vector of values to be used in the policy matrix when the node is decision node. This may be left NULL for factor variables, which will then draw on the factor levels. For numerical variables, it can be more important: if left NULL and data are available for the node, the first, second, and third quartiles will be used to populate the policy values. If no data are available and no values are provided, policyMatrix and compileDecisionModel are likely to return errors when they are called. Policy values may also be set with setPolicyValues after a network has been defined.

Details

The functions `fromFormula()` and `fromData()` help to control how `Hyde` determines the values of parameters passed to JAGS. If the parameters passed in `params` argument are to be calculated from the data or inferred from the formula, these functions may be used as placeholders instead of writing JAGS code in the `params` argument.

By default, `options(Hyde_fitModel=FALSE)`. This prevents `setNode` from fitting any models. Instead, the fitting is delayed until the user calls `writeJagsModel` and all of the models are fit at the same time. When using large data sets that may require time to run, it may be better to leave this option `FALSE` so that the models can all be compiled together (especially if you are working interactively). Using `fitModel=TRUE` will cause the model to be fit and the JAGS code for the parameters to be stored in the `nodeParams` attribute.

Value

Returns the modified `HydeNetwork` object.

Choosing a Node Type

Many of the distribution functions defined in JAGS have an equivalen distribution function in R. You may inspect the `jagsDists` data frame to see the function names in each language. You may specify the distribution function using the R name and it will be translated to the equivalent JAGS function.

You may still use the JAGS names, which allows you to specify a distribution in JAGS that does not have an R equivalent listed. Note, however, that where R functions are supported, `HydeNet` anticipates the parameter names to be given following R conventions (See the `RParameter` column of `jagsDists`.)

Of particular interest are `dbern` and `dcat`, which are functions in JAGS that have no immediate equivalent in R. They provide Bernoulli and Multinomial distributions, respectively.

Coding Factor Levels

The `nodeFormula` argument will accept any valid R formula. If desired, you may use a specific formulation to indicate the presence of factor levels in the formula. For instance, consider the case of a variable `y` with a binary categorical parent `x` coded as 0 = No, and 1 = Yes. JAGS expects the formula `y ~ c * x == 1` (where `c` is a constant). However, in factor variables with a large number of levels, it can be difficult to remember what value corresponds to what level.

`HydeNet` uses an internal (unexported) function within `setNode` to allow an alternate specification: `y ~ c * (x == "Yes")`. So long as the factors in the formula are previously defined within the network structure, `HydeNet` will translate the level into its numeric code.

Note that it is required to write `x == "Yes"`. `"Yes" == x` will not translate correctly.

Validation

The validation of parameters is performed by comparing the values provided with the limits defined in the `jagsDists$paramLogic` variable. (look at `data(jagsDists,data='HydeNet')`. For most node types, validation will be performed for numeric variables. For deterministic variables, the validation will only check that the parameter definition is a formula.

It is possible to pass character strings as definitions, but when this is done, HydeNet assumes you are passing JAGS code. Unfortunately, HydeNet doesn't have to capability to validate JAGS code, so if there is an error in the character string definition, it won't show up until you try to compile the network. If you pass a character string as a parameter and leave validate = TRUE, a message will be printed to indicate that validation is being ignored. This message can be avoided by using validate = FALSE

The two exceptions to this rule are when you pass fromFormula() and fromData() as the parameter definition. These will skip the validation without warning, since the definition will be built by HydeNet and be proper JAGS code (barring any bugs, of course).

Author(s)

Jarrod Dalton and Benjamin Nutter

Examples

```
data(PE, package="HydeNet")
Net <- HydeNetwork(~ wells +
                     pe | wells +
                     d.dimer | pregnant*pe +
                     angio | pe +
                     treat | d.dimer*angio +
                     death | pe*treat,
                     data = PE)
print(Net, d.dimer)

#* Manually change the precision
Net <- setNode(Net, d.dimer, nodeType='dnorm', mean=fromFormula(), sd=sqrt(2.65),
                   nodeFormula = d.dimer ~ pregnant * pe,
                   nodeFitter='lm')
print(Net, d.dimer)
```

---

setNodeModels *Set Node Properties Using Model Objects*

Description

Set node properties using pre-defined model objects. Model objects may be imported from other programs, but need to be valid model objects with the additional restriction that the responses and independent variables must be named nodes in the network. This will NOT create a network from a list of models. For that, see HydeNetwork

Usage

```
setNodeModels(network, ...)
```

Arguments

network	A HydeNetwork object
...	Model objects to be incorporated into network

Author(s)

Jarrod Dalton and Benjamin Nutter

Examples

```
data(PE, package="HydeNet")
Net <- HydeNetwork(~ wells +
                     pe | wells +
                     d.dimer | pregnant*pe +
                     angio | pe +
                     treat | d.dimer*angio +
                     death | pe*treat)
print(Net)

g1 <- lm(wells ~ 1, data=PE)
g2 <- glm(pe ~ wells, data=PE, family="binomial")
g3 <- lm(d.dimer ~ pe + pregnant, data=PE)
g4 <- xtabs(~ pregnant, data=PE)
g5 <- glm(angio ~ pe, data=PE, family="binomial")
g6 <- glm(treat ~ d.dimer + angio, data=PE, family="binomial")
g7 <- glm(death ~ pe + treat, data=PE, family="binomial")

Net2 <- setNodeModels(Net, g1, g2, g3, g4, g5, g6, g7)
print(Net)

writeNetworkModel(Net, pretty=TRUE)
```

---

setPolicyValues *Assign Default Policy Values*

Description

By default, HydeNet uses factor levels for policy values in a decision node, assuming the decision node is a factor variable. In cases where the decision node is a numeric variable, HydeNet will first try to assign the first, second, and third quartiles as policy values. setPolicyValues allows the user flexibility in which values are actually used in the decision network. It can also be used to restrict the levels of a factor variable to a subset of all levels. Policy values may also be set in setNode, but setPolicyValues makes it possible to set the values for multiple nodes in one call.

Usage

```
setPolicyValues(network, ...)
```

Arguments

network	A Hyde Network object
...	arguments named for nodes in the network. The value of each argument will be assigned to the nodePolicyValues element of the HydeNetwork object.

Author(s)

Jarrod Dalton and Benjamin Nutter

TranslateFormula *Translate R Formula to JAGS*

Description

While most functions available in JAGS have equivalents in R, they don't always use the exact same names. R formulas are converted to character strings, function names translated, and the corresponding JAGS formula is returned.

Usage

```
rToJags(f)
```

Arguments

f	R formula object

Details

Only a limited subset of R functions are recognized here, but no attempt is made to restrict the user to functions that will be recognized by JAGS. For now, the user should remain aware of what functions are available in JAGS and only use the corresponding functions in R. The JAGS functions may be referenced in the JAGS user manual (see References). The corresponding R functions are listed in the jagsFunctions data set (use data(jagsFunctions) to review).

Author(s)

Jarrod Dalton and Benjamin Nutter

References

http://people.stat.sc.edu/hansont/stat740/jags_user_manual.pdf

`update.HydeNetwork` *Update Probabilistic Graphical Network*

Description

Add or remove nodes or add parents within a `HydeNetwork` model.

Usage

```
## S3 method for class 'HydeNetwork'
update(object, formula, ...)
```

Arguments

`object`	A `HydeNetwork` object
`formula`	A formula statement indicating the changes to the network.
`...`	Additional arguments to be passed to other methods. Current, none are used.

Details

Adding or removing nodes is fairly straightforward if you are removing a complete node (along with its parents). Removing a parent will generate a warning that the child nodes may need to be redefined.

Author(s)

Jarrod Dalton and Benjamin Nutter

Examples

```
data(PE, package="HydeNet")
Net <- HydeNetwork(~ wells +
                     pe | wells +
                     d.dimer | pregnant*pe +
                     angio | pe +
                     treat | d.dimer*angio +
                     death | pe*treat)

plot(Net)

Net <- update(Net, . ~ . - pregnant)
plot(Net)
```

vectorProbs *Convert a vector to JAGS Probabilities*

Description

Probability vectors can be passed manually to the model, but they must be formatted in code appropriate to JAGS. `vectorProbs` will convert a vector of counts or weights to probabilities and format it into JAGS code.

Usage

```
vectorProbs(p, node, normalize = TRUE)
```

Arguments

`p`	a vector of counts, weights, or probabilities.
`node`	the node for the parameters. this is converted to a character string. It is important that this be given accurately or it will not match with the code written by `writeNetworkModel`.
`normalize`	A logical indicating if the weights in `p` should be normalized (each value is taken as a proportion of the sum).

Author(s)

Jarrod Dalton and Benjamin Nutter

Examples

```
vectorProbs(c(1, 2, 3), "wells")
```

writeJagsFormula *Write the JAGS Formula for a Hyde Node*

Description

Based on the information provided about the node, an appropriate JAGS model is written in text. This is combined with the other node models to generate the complete network.

Usage

```
writeJagsFormula(fit, nodes, ...)

## S3 method for class 'cpt'
writeJagsFormula(fit, nodes, ...)

## S3 method for class 'glm'
writeJagsFormula(fit, nodes, bern = bern, ...)

## S3 method for class 'lm'
writeJagsFormula(fit, nodes, bern, ...)

## S3 method for class 'multinom'
writeJagsFormula(fit, nodes, bern = bern, ...)

## S3 method for class 'survreg'
writeJagsFormula(fit, ..., bern = bern)

## S3 method for class 'xtabs'
writeJagsFormula(fit, ...)
```

Arguments

`fit`	a model object
`nodes`	a vector of node names, usually passed from `network$nodes`
`...`	Additional arguments to be passed to other methods
`bern`	a vector of bernoulli node names

Details

Methods for different model objects can be written so that this function can be extended as desired.

The resulting formulas are based on the coefficient matrix of the fitted model, and the returned result is the JAGS code representing the regression equation of the model.

In the `writeJagsFormula.glm` method, appropriate transformations exist for the following combinations:

1. family = binomial; link = logit
2. family = poisson; link = log
3. family = gaussian; link = identity (calls `writeJagsFormula.lm`)

Author(s)

Jarrod Dalton and Benjamin Nutter

See Also

`writeJagsModel`, `writeNetworkModel`

Examples

```
data(PE, package="HydeNet")
fit <- lm(d.dimer ~ pregnant + pe, data=PE)
writeJagsFormula(fit, nodes=c("d.dimer", "pregnant", "pe"))

fit.glm <- glm(death ~ pe + treat, data=PE, family="binomial")
writeJagsFormula(fit.glm, nodes=c("death", "pe", "treat"))
```

`writeJagsModel` *Write a Node's JAGS Model*

Description

Constructs the JAGS code that designates the model for the node conditioned on its parents. The parameters for the model may be user supplied or estimated from a given data set.

Usage

```
writeJagsModel(network, node)

writeJagsModel_default(network, node_str, node_params)

writeJagsModel_dbern(network, node_str, node_params)

writeJagsModel_dcat(network, node_str, node_params)

writeJagsModel_determ(network, node_str, node_params)

writeJagsModel_dnorm(network, node_str, node_params)

writeJagsModel_dnorm_default(network, node_str, node_params)

writeJagsModel_dpois(network, node_str, node_params)
```

Arguments

`network`	A network of class HydeNetwork
`node`	A node within `network`
`node_str`	A character string giving the name of a node within `network`. This is usually generated within `writeJagsModel` and passed to a specific method.
`node_params`	A vector of parameters for the node. Generated by `writeJagsModel` and passed to a specific method.

Details

The manipulations are performed on the nodeParams element of the Hyde network. A string of JAGS code is returned suitable for inclusion in the Bayesian analysis.

The function will (eventually) travel through a serious of if statements until it finds the right node type. It will then match the appropriate arguments to the inputs based on user supplied values or estimating them from the data.

Author(s)

Jarrod Dalton and Benjamin Nutter

See Also

writeJagsFormula

Examples

```
## Not run:
#* NOTE: writeJagsModel isn't an exported function
data(PE, package='HydeNet')
Net <- HydeNetwork(~ wells +
                     pe | wells +
                     d.dimer | pregnant*pe +
                     angio | pe +
                     treat | d.dimer*angio +
                     death | pe*treat,
                     data = PE)
HydeNet:::writeJagsModel(Net, 'pe')
HydeNet:::writeJagsModel(Net, 'treat')

## End(Not run)
```

---

writeNetworkModel *Generate JAGS Code for a Network's Model*

Description

Based on the parameters given to a network, the code for each node is generated and all of the node models are pasted into a single JAGS model script.

Usage

```
writeNetworkModel(network, pretty = FALSE)
```

Arguments

network	an object of class HydeNetwork
pretty	Logical. When TRUE, the model is printed to the console using the cat function (useful if you wish to copy and paste the code for manual editing). Otherwise, it is returned as a character string.

Author(s)

Jarrod Dalton and Benjamin Nutter

See Also

writeJagsModel, writeJagsFormula

Examples

```
data(PE, package='HydeNet')
Net <- HydeNetwork(~ wells +
                     pe | wells +
                     d.dimer | pregnant*pe +
                     angio | pe +
                     treat | d.dimer*angio +
                     death | pe*treat,
                     data = PE)

#* Default printing
writeNetworkModel(Net)

#* Something a little easier on the eyes.
writeNetworkModel(Net, pretty=TRUE)
```

%>% *Chain together multiple operations.*

Description

This is a copy of the documentation for %>% in magrittr. The copy here is made to conform to CRAN requirements regarding documentation. Please see the magrittr documenation for the complete and current documentation.

This is a copy of the documentation for %$% in magrittr. The copy here is made to conform to CRAN requirements regarding documentation. Please see the magrittr documenation for the complete and current documentation.

Usage

```
lhs %>% rhs

lhs %$% rhs
```

Arguments

`lhs, rhs`	A dataset and function to apply to it

Index