Package ‘rdecision’

February 1, 2021

Title Decision Analytic Modelling in Health Economics

Version 1.0.0

Description Classes and functions for modelling healthcare interventions using cohort models (decision trees, Markov models and extended Markov models). It draws on terminology and examples from Briggs, Claxton and Sculpher, ``Decision Modelling for Health Economic Evaluation", Oxford University Press, 2006.

Depends R (>= 3.1.0)

Imports R6, rlang (>= 0.4.2), stats, utils

Suggests rmarkdown, knitr, grid, pander, testthat

License GPL-3

LazyData true

Encoding UTF-8

RoxygenNote 7.1.0

VignetteBuilder knitr, rmarkdown

NeedsCompilation no

Author Andrew Sims [aut, cre] (<https://orcid.org/0000-0002-9553-7278>), Kim Fairbairn [aut] (<https://orcid.org/0000-0001-5108-6279>)

Maintainer Andrew Sims <andrew.sims@newcastle.ac.uk>

R topics documented:

`Action` *Action*

Description

An R6 class to represent an action (choice) edge in a decision tree.

Details

A specialism of class Arrow which is used in a decision tree to represent edges with source nodes joined to `DecisionNodes`.

Super classes

`rdecision::Edge` -> `rdecision::Arrow` -> `Action`

Methods

Public methods:

- `Action$new()`
- `Action$modvars()`
- `Action$cost()`
- `Action$benefit()`
- `Action$clone()`

Method `new()`**:** Create an object of type 'Action'. Optionally, a cost and a benefit may be associated with traversing the edge. A *payoff* (benefit-cost) is sometimes used in edges of decision trees; the parametrization used here is more general.

Usage:

```
Action$new(source, target, label, cost = 0, benefit = 0)
```

Arguments:

`source` Decision node from which the arrow leaves.

`target` Node which the arrow enters.

`label` Character string containing the arrow label. This

`cost` Cost associated with traversal of this edge.

`benefit` Benefit associated with traversal of the edge. must be defined for an action because the label is used in tabulation of strategies.

Returns: A new `Action` object.

Method `modvars()`**:** Find all the model variables of type ModVar that have been specified as values associated with this Action. Includes operands of these `ModVar`s, if they are expressions.

Usage:

`Action$modvars()`

Returns: A list of `ModVars`.

Method `cost()`**:** Return the cost associated with traversing the edge.

Usage:

`Action$cost()`

Returns: Cost.

Method `benefit()`**:** Return the benefit associated with traversing the edge.

Usage:

`Action$benefit()`

Returns: Benefit.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

`Action$clone(deep = FALSE)`

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Andrew J. Sims `<andrew.sims@newcastle.ac.uk>`

`Arborescence` *Arborescence*

Description

An R6 class to represent an arborescence (rooted directed tree).

Details

Class to encapsulate a directed rooted tree specialization of a digraph. An arboresecence must be a directed tree with exactly one root and the directed paths from the root must be unique.

Super classes

`rdecision::Graph` -> `rdecision::Digraph` -> `Arborescence`

Methods

Public methods:

- `Arborescence$new()`
- `Arborescence$is_parent()`
- `Arborescence$is_leaf()`
- `Arborescence$root()`
- `Arborescence$root_to_leaf_paths()`

- `Arborescence$clone()`

Method `new()`**:** Create a new Arborescence object from sets of nodes and edges.

Usage:

```
Arborescence$new(V, A)
```

Arguments:

`V` A list of Nodes.

`A` A list of Arrows.

Returns: An Arborescence object.

Method `is_parent()`**:** Test whether the given node is a parent (has child nodes).

Usage:

```
Arborescence$is_parent(v)
```

Arguments:

`v` Node to test

Returns: TRUE if v has one or more child nodes, FALSE otherwise.

Method `is_leaf()`**:** Test whether the given vertex is a leaf. In an arborescence, `is_parent()` and `is_leaf()` will be mutually exclusive.

Usage:

```
Arborescence$is_leaf(v)
```

Arguments:

`v` Vertex to test.

Returns: TRUE if v has no child nodes, FALSE otherwise.

Method `root()`**:** Find the root vertex of the arborescence.

Usage:

```
Arborescence$root()
```

Returns: The root vertex.

Method `root_to_leaf_paths()`**:** Find all directed paths from the root of the tree to the leaves.

Usage:

```
Arborescence$root_to_leaf_paths()
```

Returns: A list of ordered node lists.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Arborescence$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Andrew Sims <`andrew.sims@newcastle.ac.uk`>

`Arrow` *Arrow*

Description

An R6 class to represent an directed edge in a digraph.

Details

Arrows are the formal term for links between pairs of nodes in a directed graph. Inherits from class Edge.

Super class

`rdecision::Edge` -> `Arrow`

Methods

Public methods:

- `Arrow$new()`
- `Arrow$source()`
- `Arrow$target()`
- `Arrow$clone()`

Method `new()`**:** Create an object of type 'Arrow'.

Usage:

```
Arrow$new(source, target, label = "")
```

Arguments:

`source` Node from which the arrow leaves.

`target` second Node to which the arrow enters.

`label` Character string containing the arrow label.

Returns: A new ‘Arrow‘ object.

Method `source()`**:** Access source node.

Usage:

```
Arrow$source()
```

Returns: ‘Node‘ from which the arrow leads.

Method `target()`**:** Access target node.

Usage:

```
Arrow$target()
```

Returns: ‘Node‘ to which the arrow points.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Arrow$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Andrew J. Sims <andrew.sims@newcastle.ac.uk>

ChanceNode *ChanceNode*

Description

An R6 class to represent a chance node in a decision tree.

Details

An R6 class to represent a chance node in a decision tree. The node is associated with at least two branches to other nodes, each of which has a conditional probability (the probability of following that branch given that the node has been reached) and a cost.

Super class

`rdecision::Node` -> `ChanceNode`

Methods

Public methods:

- `ChanceNode$new()`
- `ChanceNode$clone()`

Method `new()`**:** Create a new ChanceNode object

Usage:

```
ChanceNode$new(label = "")
```

Arguments:

`label` An optional label for the chance node.

Returns: A new ‘ChanceNode‘ object

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
ChanceNode$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Andrew Sims <andrew.sims@newcastle.ac.uk>

`ConstModVar` *ConstModVar*

Description

An R6 class for a constant in a model

Details

A ModVar with no uncertainty in its value. It has no distribution and there are no hyperparameters. Its benefit over using a regular 'numeric' variable in a model is that it will appear in automatic tabulations of the model variables associated with a model and therefore be explicitly documented as a model input.

Super class

`rdecision::ModVar` -> `ConstModVar`

Methods

Public methods:

- `ConstModVar$new()`
- `ConstModVar$is_probabilistic()`
- `ConstModVar$distribution()`
- `ConstModVar$mode()`
- `ConstModVar$mean()`
- `ConstModVar$r()`
- `ConstModVar$SD()`
- `ConstModVar$quantile()`
- `ConstModVar$clone()`

Method `new()`**:** Create a new constant model variable

Usage:

```
ConstModVar$new(description, units, const)
```

Arguments:

`description` A character string description of the variable and its role in the model. This description will be used in a tabulation of the variables linked to a model.

`units` A character string description of the units, e.g. 'GBP', 'per year'.

`const` The constant numerical value of the object.

Returns: A new ConstModVar object.

Method `is_probabilistic()`**:** Tests whether the model variable is probabilistic, i.e. a random variable that follows a distribution, or an expression involving random variables, some of which follow distributions.

Usage:

```
ConstModVar$is_probabilistic()
```

Returns: TRUE if probabilistic

Method `distribution()`**:** Accessor function for the name of the uncertainty distribution.

Usage:

`ConstModVar$distribution()`

Returns: Distribution name as character string.

Method `mode()`**:** Return the mode of the distribution.

Usage:

`ConstModVar$mode()`

Returns: Value of the constant.

Method `mean()`**:** Return the expected value of the distribution.

Usage:

`ConstModVar$mean()`

Returns: Expected value as a numeric value.

Method `r()`**:** Return a random sample from the distribution.

Usage:

`ConstModVar$r(n = 1)`

Arguments:

`n` Number of samples to draw.

Returns: Constant value as a numeric value.

Method `SD()`**:** Return the standard deviation of the distribution.

Usage:

`ConstModVar$SD()`

Returns: Standard deviation as a numeric value

Method `quantile()`**:** Quantiles of the uncertainty distribution; for a constant all quantiles are returned as the value of the constant.

Usage:

`ConstModVar$quantile(probs)`

Arguments:

`probs` Numeric vector of probabilities, each in range [0,1].

Returns: Vector of numeric values of the same length as ‘probs‘.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

`ConstModVar$clone(deep = FALSE)`

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Andrew Sims <`andrew.sims@newcastle.ac.uk`>

`DecisionNode` *DecisionNode*

Description

An R6 class for a decision node in a decision tree

Details

A class to represent a decision node in a decision tree. The node is associated with one or more branches to child nodes.

Super class

`rdecision::Node` -> `DecisionNode`

Methods

Public methods:

- `DecisionNode$new()`
- `DecisionNode$clone()`

Method `new()`**:** Create a new decision node.

Usage:

```
DecisionNode$new(label)
```

Arguments:

`label` A label for the node. Must be defined because the label is used in tabulation of strategies. The label is automatically converted to a syntactically valid (in R) name to ensure it can be used as a column name in a data frame.

Returns: A new DecisionNode object

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
DecisionNode$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Andrew J. Sims `<andrew.sims@newcastle.ac.uk>`

`DecisionTree` *DecisionTree*

Description

An R6 class to represent a decision tree

Details

A class to represent a decision tree. An object contains a tree of decision nodes, chance nodes and leaf nodes, connected by edges (either actions or reactions) and which satisfies the following conditions:

1. Nodes and edges must form a tree with a single root and there must be a unique path from the root to each node. In graph theory terminology, the directed graph formed by the nodes and edges must be an *arborescence*.
2. Each node must inherit from one of `DecisionNode`, `ChanceNode` or `LeafNode`. Formally the set of vertices must be a disjoint union of sets of decision nodes, chance nodes and leaf nodes.
3. All and only leaf nodes must have no children.
4. Each edge must inherit from either `Action` or `Reaction`.
5. All and only edges that have source endpoints joined to decision nodes must inherit from `Action`.
6. All and only edges that have source endpoints joined to chance nodes must inherit from `Reaction`.
7. The sum of probabilities of each set of reaction edges with a common source endpoint must be 1.
8. Each `DecisionNode` must have a label, and the labels of all `DecisionNodes` must be unique within the model.
9. Each `Action` must have a label, and the labels of `Actions` that share a common source endpoint must be unique.

Super classes

`rdecision::Graph` -> `rdecision::Digraph` -> `rdecision::Arborescence` -> `DecisionTree`

Methods

Public methods:

- `DecisionTree$new()`
- `DecisionTree$decision_nodes()`
- `DecisionTree$chance_nodes()`
- `DecisionTree$leaf_nodes()`
- `DecisionTree$actions()`
- `DecisionTree$modvars()`
- `DecisionTree$modvar_table()`
- `DecisionTree$paths_in_strategy()`
- `DecisionTree$strategies()`

- DecisionTree$evaluate_strategy()
- DecisionTree$evaluate()
- DecisionTree$clone()

Method `new()`**:** Create a new decision tree. The tree must consist of a set of nodes and a set of edges which satisfy the conditions given in the details section of this class.

Usage:

```
DecisionTree$new(V, E)
```

Arguments:

`V` A list of nodes.

`E` A list of edges.

Returns: A DecisionTree object

Method `decision_nodes()`**:** Find the decision nodes in the tree.

Usage:

```
DecisionTree$decision_nodes(what = "node")
```

Arguments:

`what` A character string defining what to return. Must be one of "node", "label" or "index".

Returns: A list of `DecisionNode` objects (for what="node"); a list of character strings (for what="label"); or a list of integer indexes of the decision nodes (for what="index").

Method `chance_nodes()`**:** Find the chance nodes in the tree.

Usage:

```
DecisionTree$chance_nodes()
```

Returns: A list of `ChanceNode` objects.

Method `leaf_nodes()`**:** Find the leaf nodes in the tree.

Usage:

```
DecisionTree$leaf_nodes()
```

Returns: A list of `LeafNode` objects.

Method `actions()`**:** Return the edges that have the specified decision node as their source.

Usage:

```
DecisionTree$actions(d)
```

Arguments:

`d` A decision node.

Returns: A list of Action edges.

Method `modvars()`**:** Find all the model variables of type ModVar that have been specified as values associated with the nodes and edges of the tree.

Usage:

```
DecisionTree$modvars()
```

Returns: A list of `ModVar`s.

Method `modvar_table()`**:** Tabulate the model variables.

Usage:

```
DecisionTree$modvar_table()
```

Returns: Data frame with one row per model variable, as follows:

Label The label given to the variable on creation.

Description As given at initialization.

Units Units of the variable.

Distribution Either the uncertainty distribution, if it is a regular model variable, or the expression used to create it, if it is an ExprModVar.

Mean Mean; calculated from means of operands if an expression.

E Expectation; estimated from random sample if expression, mean otherwise.

SD Standard deviation; estimated from random sample if expression, exact value otherwise.

Q2.5 p=0.025 quantile; estimated from random sample if expression, exact value otherwise.

Q97.5 p=0.975 quantile; estimated from random sample if expression, exact value otherwise.

Est TRUE if the quantiles and SD have been estimated by random sampling.

Method `paths_in_strategy()`**:** Find all the root to leaf paths traversable under the specified strategy. A strategy is a unanimous prescription of an action in each decision node.

Usage:

```
DecisionTree$paths_in_strategy(strategy)
```

Arguments:

`strategy` A list of Actions, with one action per decision node.

Returns: A list of root to leaf paths.

Method `strategies()`**:** Find all unique strategies for the decision tree. A strategy is a unanimous prescription of the actions at each decision node. In trees where there are decision nodes that are descendants of other decision nodes, not all decision nodes are reachable in each strategy. Equivalently, different strategies involve the traversal of an identical set of paths and are considered non- unique. Only unique strategies are returned.

Usage:

```
DecisionTree$strategies(what = "index")
```

Arguments:

`what` A character string defining what to return. Must be one of "label" or "index".

Returns: A table (data frame) where each row is a strategy traversed by a unique set of paths, and each column is a Decision Node. Values are either the index of each action edge, or their label.

Method `evaluate_strategy()`**:** Evaluate the components of payoff associated with the paths in the decision tree. For each path, the strategy, probability, cost, benefit and utility are calculated.

Usage:

```
DecisionTree$evaluate_strategy(strategy)
```

Arguments:

`strategy` A list of Actions, with one action per decision node.

Returns: A data frame (payoff table) with one row per path and columns organized as follows:

<label of decision node> One column for each decision node in the mode. Each column is named with the label of the node. For each row (path) the value is the label of the Action edge taken from the decision node.

Leaf The label of the leaf node on which the pathway ends; normally the clinical outcome.

Probability The probability of traversing the pathway. The total probability of each strategy should sum to unity.

Path.Cost The cost of traversing the pathway.

Path.Benefit The benefit derived from traversing the pathway.

Path.Utility The utility associated with the outcome (leaf node).

Cost Path.Cost $*$ probability of traversing the pathway.

Benefit Path.Benefit $*$ probability of traversing the pathway.

Utility Path.Utility $*$ probability of traversing the pathway.

Method `evaluate()`**:** Evaluate each strategy. Starting with the root, the function works though all possible paths to leaf nodes and computes the probability, cost, benefit and utility of each, then aggregates by strategy.

Usage:

```
DecisionTree$evaluate(expected = TRUE, N = 1)
```

Arguments:

`expected` If TRUE, evaluate each model variable as its mean value, otherwise sample each one from their uncertainty distribution.

`N` Number of replicates. Intended for use with PSA (expected=F); use with expected=T will be repetitive and uninformative.

Returns: A data frame with one row per strategy per run and columns organized as follows:

Run The run number

Strategy The strategy.

Cost Aggregate cost of the strategy.

Benefit Aggregate benefit of the strategy.

Utility Aggregate utility of the strategy.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
DecisionTree$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Andrew J. Sims `<andrew.sims@newcastle.ac.uk>`

`Digraph` *Digraph*

Description

An R6 class to represent a digraph (a directed graph).

Details

Encapulates, and provides methods for computation and checking of directed graphs (digraphs). Inherits from class Graph.

Super class

`rdecision::Graph` -> `Digraph`

Methods

Public methods:

- `Digraph$new()`
- `Digraph$adjacency_matrix()`
- `Digraph$incidence_matrix()`
- `Digraph$topological_sort()`
- `Digraph$is_connected()`
- `Digraph$is_weakly_connected()`
- `Digraph$is_acyclic()`
- `Digraph$is_tree()`
- `Digraph$is_polytree()`
- `Digraph$is_arborescence()`
- `Digraph$direct_successors()`
- `Digraph$direct_predecessors()`
- `Digraph$paths()`
- `Digraph$walk()`
- `Digraph$clone()`

Method `new()`**:** Create a new Digraph object from sets of nodes and edges.

Usage:

```
Digraph$new(V, A)
```

Arguments:

`V` A list of Nodes.

`A` A list of Arrows.

Returns: A Digraph object.

Method `adjacency_matrix()`**:** Compute the adjacency matrix for the digraph. Each cell contains the number of edges from the row vertex to the column vertex, with the convention of self loops being counted once, unless 'boolean' is TRUE when cells are either FALSE (not adjacent) or TRUE (adjacent).

Usage:

```
Digraph$adjacency_matrix(boolean = FALSE)
```

Arguments:

`boolean` If TRUE, the adjacency matrix is logical, each cell is FALSE,TRUE.

Returns: A square numeric matrix with the number of rows and columns equal to the order of the graph. The rows and columns are in the same order as V. If the nodes have defined and unique labels the dimnames of the matrix are the labels of the nodes.

Method `incidence_matrix()`**:** Compute the incidence matrix for the graph. Each row is a vertex and each column is an edge. Edges leaving a vertex have value -1 and edges entering have value +1. if all vertexes have defined and unique labels and all edges have defined and unique labels, the dimnames of the matrix are the labels of the vertexes and edges.

Usage:

`Digraph$incidence_matrix()`

Returns: The incidence matrix.

Method `topological_sort()`**:** Attempt to topologically sort the vertexes in the directed graph using Kahn's algorithm (https://doi.org/10.1145

Usage:

`Digraph$topological_sort()`

Returns: A list of vertexes, topologically sorted. If the digraph has cycles, the returned ordered list will not contain all the vertexes in the graph, but no error will be raised.

Method `is_connected()`**:** Test whether the graph is connected. For digraphs this will always return FALSE because "connected" is not defined. Function `weakly_connected` calculates whether the underlying graph is connected.

Usage:

`Digraph$is_connected()`

Returns: TRUE if connected, FALSE if not.

Method `is_weakly_connected()`**:** Test whether the digraph is weakly connected, i.e. if the underlying graph is connected.

Usage:

`Digraph$is_weakly_connected()`

Returns: TRUE if connected, FALSE if not.

Method `is_acyclic()`**:** Checks for the presence of a cycle in the graph by attempting to do a topological sort. If the sort does not contain all vertexes, the digraph contains at least one cycle. This method overrides 'is_acyclic' in Graph.

Usage:

`Digraph$is_acyclic()`

Returns: TRUE if no cycles detected.

Method `is_tree()`**:** Compute whether the digraph's underlying graph is a tree (connected and acyclic).

Usage:

`Digraph$is_tree()`

Returns: TRUE if the underlying graph is a tree; FALSE if not.

Method `is_polytree()`**:** Compute whether the digraph's underlying graph is a tree (connected and acyclic). Synonymous with 'is_graph'.

Usage:

`Digraph$is_polytree()`

Returns: TRUE if the underlying graph is a tree; FALSE if not.

Method `is_arborescence()`**:** Check whether the digraph is an arborescence (a tree with a single root and unique paths from the root).

Usage:

`Digraph$is_arborescence()`

Returns: TRUE if the digraph is an arborescence; FALSE if not.

Method `direct_successors()`**:** Find the direct successors of a node.

Usage:

```
Digraph$direct_successors(v)
```

Arguments:

`v` The index vertex.

Returns: A list of nodes or an empty list if the specified node has no successors.

Method `direct_predecessors()`**:** Find the direct predecessors of a node.

Usage:

```
Digraph$direct_predecessors(v)
```

Arguments:

`v` The index vertex.

Returns: A list of nodes or an empty list if the specified node has no predecessors.

Method `paths()`**:** Find all directed paths from source node 's' to target node 't'. In this definition, 'path' is a simple path, i.e. all vertexes are unique. Uses a recursive depth-first search algorithm.

Usage:

```
Digraph$paths(s, t)
```

Arguments:

`s` Source node.

`t` Target node.

Returns: A list of ordered node lists.

Method `walk()`**:** Construct the sequence of edges which joins the specified sequence of vertexes in this graph.

Usage:

```
Digraph$walk(P)
```

Arguments:

`P` A list of Nodes

Returns: A list of Edges

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Digraph$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Andrew Sims <`andrew.sims@newcastle.ac.uk`>

Edge *Edge*

Description

An R6 class to represent an edge in a graph.

Details

Edges are the formal term for links between pairs of nodes in a graph.

Methods

Public methods:

- `Edge$new()`
- `Edge$is_same_edge()`
- `Edge$endpoints()`
- `Edge$label()`
- `Edge$clone()`

Method `new()`**:** Create an object of type 'Edge'.

Usage:

```
Edge$new(v1, v2, label = "")
```

Arguments:

`v1` Node at one endpoint of the edge.

`v2` Node at the other endpoint of the edge.

`label` Character string containing the edge label.

Returns: A new 'Edge' object.

Method `is_same_edge()`**:** Is this edge the same as the argument? (DOM-style)

Usage:

```
Edge$is_same_edge(e)
```

Arguments:

`e` edge to compare with this one

Returns: TRUE if 'e' is also this one.

Method `endpoints()`**:** Retrieve the endpoints of the edge.

Usage:

```
Edge$endpoints()
```

Returns: List of two nodes to which the edge is connected.

Method `label()`**:** Access label.

Usage:

```
Edge$label()
```

Returns: Label of the edge; character string.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Edge$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Andrew J. Sims <andrew.sims@newcastle.ac.uk>

`ExprModVar` *ExprModVar*

Description

An R6 class for a model variable constructed from an expression involving other model variables.

Details

A class to support expressions involving objects of base class ModVar, which itself behaves like a model variable. For example, if `A` and `B` are variables with base class `ModVar` and `c` is a variable of type `numeric`, then it is not possible to write, for example, `x <-42*A/B + c`, because R cannot manipulate class variables using the same operators as regular variables. But such forms of expression may be desirable in constructing a model and this class provides a mechanism for doing so.

Super class

`rdecision::ModVar` -> `ExprModVar`

Methods

Public methods:

- `ExprModVar$new()`
- `ExprModVar$is_probabilistic()`
- `ExprModVar$operands()`
- `ExprModVar$distribution()`
- `ExprModVar$r()`
- `ExprModVar$mean()`
- `ExprModVar$mode()`
- `ExprModVar$SD()`
- `ExprModVar$quantile()`
- `ExprModVar$mu_hat()`
- `ExprModVar$sigma_hat()`
- `ExprModVar$q_hat()`
- `ExprModVar$set()`
- `ExprModVar$get()`
- `ExprModVar$clone()`

Method `new()`**:** Create a Model Variable formed from an expression involving other model variables.

Usage:

```
ExprModVar$new(description, units, quo)
```

Arguments:

`description` Name for the model variable expresssion. In a complex model it may help to tabulate how model variables are combined into costs, probablities and rates.

`units` Units in which the variable is expressed.

`quo` A quosure (see package rlang), which contains an expression and its environment. The usage is ‘quo(x+y)‘ or ‘rlang::quo(x+y)‘.

Returns: An object of type ExprModVar

Method `is_probabilistic()`**:** Tests whether the model variable is probabilistic, i.e. a random variable that follows a distribution, or an expression involving random variables, at least one of which follows a distribution.

Usage:

```
ExprModVar$is_probabilistic()
```

Returns: TRUE if probabilistic

Method `operands()`**:** Return a list of operands that are themselves ModVars given in the expression.

Usage:

```
ExprModVar$operands()
```

Returns: A list of model variables.

Method `distribution()`**:** Accessor function for the name of the expression model variable.

Usage:

```
ExprModVar$distribution()
```

Returns: Expression as a character string with all control characters having been removed.

Method `r()`**:** Draw a random sample from the model variable.

Usage:

```
ExprModVar$r(n = 1)
```

Arguments:

`n` Number of samples to draw.

Returns: A sample drawn at random.

Method `mean()`**:** Return the value of the expression when its operands take their mean value (i.e. value returned by call to `mean` or their value, if numeric). See notes on this class for further explanation.

Usage:

```
ExprModVar$mean()
```

Returns: Mean value as a numeric value.

Method `mode()`**:** Return the mode of the variable. By default returns NA, which will be the case for most ExprModVar variables, because an arbitrary expression is not guaranteed to be unimodel.

Usage:

`ExprModVar$mode()`

Returns: Mode as a numeric value.

Method `SD()`**:** Return the standard deviation of the distribution as NA because the variance is not available as a closed form for all functions of distributions.

Usage:

`ExprModVar$SD()`

Returns: Standard deviation as a numeric value

Method `quantile()`**:** Find quantiles of the uncertainty distribution. Not available as a closed form, and returned as NA.

Usage:

`ExprModVar$quantile(probs)`

Arguments:

`probs` Numeric vector of probabilities, each in range [0,1].

Returns: Vector of numeric values of the same length as ‘probs‘.

Method `mu_hat()`**:** Return the estimated expected value of the expression variable. This is computed by numerical simulation because there is, in general, no closed form expressions for the mean of a function of distributions.

Usage:

`ExprModVar$mu_hat(nest = 1000)`

Arguments:

`nest` Sample size to be used to estimate the mean. Values less than 1000 (default) are unlikely to return meaningful estimates and will be rejected.

Returns: Expected value as a numeric value.

Method `sigma_hat()`**:** Return the estimated standard deviation of the distribution. This is computed by numerical simulation because there is, in general, no closed form expressions for the SD of a function of distributions.

Usage:

`ExprModVar$sigma_hat(nest = 1000)`

Arguments:

`nest` Sample size to be used to estimate the SD. Values less than 1000 (default) are unlikely to return meaningful estimates and will be rejected.

Returns: Standard deviation as a numeric value.

Method `q_hat()`**:** Return the estimated quantiles by sampling the variable. This is computed by numerical simulation because there is, in general, no closed form expressions for the quantiles of a function of distributions.

Usage:

`ExprModVar$q_hat(probs, nest = 1000)`

Arguments:

`probs` Vector of probabilities, in range [0,1].

`nest` Sample size to be used to estimate the SD. Values less than 1000 (default) are unlikely to return meaningful estimates and will be rejected.

Returns: Vector of quantiles.

Method `set()`**:** Sets the value of the ExprModVar that will be returned by subsequent calls to get() until set() is called again. Because an ExprModVar can be considered the LHS of an equation, the idea of `setting` a value is meaningless, and calls to this method have no effect. To affect the value returned by the next call to `get`, call `set` for each of the operands of this expression.

Usage:

```
ExprModVar$set(expected = FALSE)
```

Arguments:

`expected` Logical; for compatibility with non-expression ModVars only; not used.

Returns: Updated ExprModVar.

Method `get()`**:** Gets the value of the ExprModVar that was set by the most recent call to set() to each operand of the expression.

Usage:

```
ExprModVar$get()
```

Returns: Value determined by last set().

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
ExprModVar$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Note

For many expressions involving model variables there will be no closed form expressions for the mean, standard deviation and the quantiles. Therefore they are obtained by simulation, via functions `mu_hat`, `sigma_hat` and `q_hat`.

For consistency with `ModVars` which are not expressions, the function `mean` returns the value of the expression when all its operands take their mean values. This will, in general, not be the mean of the expression distribution (which can be obtained via `mu_hat`), but is the value normally used in the base case of a model as the point estimate. As Briggs *et al* note (section 4.1.1) "in all but the most nonlinear models, the difference between the expectation over the output of a probabilistic model and that model evaluated at the mean values of the input parameters, is likely to be modest."

Functions `SD`, `mode` and `quantile` return NA because they do not necessarily have a closed form. The standard deviation can be estimated by calling `sigma_hat` and the quantiles by `q_hat`. Because a unimodal distribution is not guaranteed, there is no estimator provided for the mode.

Method `distribution` returns the string representation of the expression used to create the model variable.

Author(s)

Andrew J. Sims <`andrew.sims@newcastle.ac.uk`>

`GammaModVar` *GammaModVar*

Description

An R6 class for a model variable with Gamma function uncertainty

Details

A model variable for which the uncertainty in the point estimate can be modelled with a Gamma distribution. The hyperparameters of the distribution are the shape (‘alpha‘) and the rate (‘beta‘) of the uncertainty distribution. Note that this is the conventional parametrization used in Bayesian statistics; in econometrics the shape/scale (‘k‘/‘theta‘) parametrization is more common (and the one used in this implementation). Note, however, that although Briggs et al use the shape/scale formulation, they use ‘alpha‘/‘beta‘ as parameter names.

Super class

`rdecision::ModVar` -> `GammaModVar`

Methods

Public methods:

- `GammaModVar$new()`
- `GammaModVar$is_probabilistic()`
- `GammaModVar$distribution()`
- `GammaModVar$mean()`
- `GammaModVar$mode()`
- `GammaModVar$SD()`
- `GammaModVar$r()`
- `GammaModVar$quantile()`
- `GammaModVar$clone()`

Method `new()`**:** Create an object of class GammaModVar.

Usage:

```
GammaModVar$new(description, units, shape, scale)
```

Arguments:

`description`	A character string describing the variable.
`units`	Units of the variable, as character string.
`shape`	shape parameter of the Gamma distribution.
`scale`	scale parameter of the Gamma distribution.

Returns: An object of class GammaModVar.

Method `is_probabilistic()`**:** Tests whether the model variable is probabilistic, i.e. a random variable that follows a distribution, or an expression involving random variables, some of which follow distributions.

Usage:

```
GammaModVar$is_probabilistic()
```

Returns: TRUE if probabilistic

Method `distribution()`**:** Accessor function for the name of the uncertainty distribution.

Usage:

```
GammaModVar$distribution()
```

Returns: Distribution name as character string.

Method `mean()`**:** Return the expected value of the distribution.

Usage:

```
GammaModVar$mean()
```

Returns: Expected value as a numeric value.

Method `mode()`**:** Return the mode of the distribution (if shape >= 1)

Usage:

```
GammaModVar$mode()
```

Returns: mode as a numeric value.

Method `SD()`**:** Return the standard deviation of the distribution.

Usage:

```
GammaModVar$SD()
```

Returns: Standard deviation as a numeric value

Method `r()`**:** Draw a random sample from the model variable. Normally accessed by a call to value(what="r").

Usage:

```
GammaModVar$r(n = 1)
```

Arguments:

`n` Number of samples to draw.

Returns: Samples drawn at random.

Method `quantile()`**:** Return the quantiles of the Gamma uncertainty distribution.

Usage:

```
GammaModVar$quantile(probs)
```

Arguments:

`probs` Vector of probabilities, in range [0,1].

Returns: Vector of quantiles.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
GammaModVar$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Andrew J. Sims <andrew.sims@newcastle.ac.uk>

Graph *Graph*

Description

An R6 class to represent a graph (from discrete mathematics).

Details

Encapulates and provides methods for computation and checking of undirected graphs. Graphs are systems of vertices connected in pairs by edges.

Methods

Public methods:

- `Graph$new()`
- `Graph$has_vertex()`
- `Graph$has_edge()`
- `Graph$has_element()`
- `Graph$element_index()`
- `Graph$order()`
- `Graph$size()`
- `Graph$adjacency_matrix()`
- `Graph$is_simple()`
- `Graph$is_connected()`
- `Graph$is_acyclic()`
- `Graph$is_tree()`
- `Graph$degree()`
- `Graph$neighbours()`
- `Graph$clone()`

Method `new()`**:** Create a new Graph object from sets of nodes and edges.

Usage:

```
Graph$new(V, E)
```

Arguments:

`V` A list of Nodes.

`E` A list of Edges.

Returns: A Graph object.

Method `has_vertex()`**:** Test whether a vertex an element of the graph.

Usage:

```
Graph$has_vertex(v)
```

Arguments:

`v` Subject vertex.

Returns: TRUE if v is an element of V(G).

Method `has_edge()`**:** Test whether an edge is element of the graph.

Usage:

```
Graph$has_edge(e)
```

Arguments:

`e` Subject edge.

Returns: TRUE if e is an element of E(G).

Method `has_element()`**:** Test whether an edge is an element of the graph.

Usage:

```
Graph$has_element(x)
```

Arguments:

`x` Subject vertex or edge

Returns: TRUE if x is an element of V(G), the vertex set, or x is an element of E(G), the edge set.

Method `element_index()`**:** Find the index of element x in the vertices or edges of the graph. The vertices and edges are normally stored internally in the same order they were defined in the call to $new(), but this cannot be guaranteed. The index returned by this function will be same as the index of a vertex or edge returned by other methods, e.g. adjacency_matrix.

Usage:

```
Graph$element_index(x)
```

Arguments:

`x` The subject element (a Node or Edge).

Returns: The index of the element (integer).

Method `order()`**:** Return the order of the graph (number of vertices).

Usage:

```
Graph$order()
```

Returns: Order of the graph (integer).

Method `size()`**:** Return the size of the graph (number of edges).

Usage:

```
Graph$size()
```

Returns: Size of the graph (integer).

Method `adjacency_matrix()`**:** Compute the adjacency matrix for the graph. Each cell contains the number of edges joining the two vertexes, with the convention of self loops being counted twice, unless 'binary' is TRUE when cells are either 0 (not adjacent) or 1 (adjacent).

Usage:

```
Graph$adjacency_matrix(boolean = FALSE)
```

Arguments:

`boolean` If TRUE, the adjacency matrix is logical, each cell is FALSE,TRUE.

Returns: A square numeric matrix with the number of rows and columns equal to the order of the graph. The rows and columns are in the same order as V. If the nodes have defined and unique labels the dimnames of the matrix are the labels of the nodes.

Method `is_simple()`**:** A simple graph has no self loops or multi-edges.

Usage:

```
Graph$is_simple()
```

Returns: TRUE if simple, FALSE if not.

Method `is_connected()`**:** Test whether the graph is connected. Graphs with no vertices are considered unconnected; graphs with 1 vertex are considered connected. Otherwise a graph is connected if all nodes can be reached from an arbitrary starting point. Uses a depth first search.

Usage:

```
Graph$is_connected()
```

Returns: TRUE if connected, FALSE if not.

Method `is_acyclic()`**:** Checks for the presence of a cycle in the graph using a depth-first search from each node to detect the presence of back edges. A back edge is an edge from the current node joining a previously detected (visited) node, that is not the parent node of the current one.

Usage:

```
Graph$is_acyclic()
```

Returns: TRUE if no cycles detected.

Method `is_tree()`**:** Compute whether the graph is connected and acyclic.

Usage:

```
Graph$is_tree()
```

Returns: TRUE if the graph is a tree; FALSE if not.

Method `degree()`**:** The degree of a vertex in the graph, or number of incident edges.

Usage:

```
Graph$degree(v)
```

Arguments:

`v` The subject node.

Returns: Degree of the vertex, integer.

Method `neighbours()`**:** Find the neighbours of a node. A property of the graph, not the node. Does not include self, even in the case of a loop to self.

Usage:

```
Graph$neighbours(v)
```

Arguments:

`v` The subject node.

Returns: A list of nodes which are joined to the subject.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Graph$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Andrew Sims <`andrew.sims@newcastle.ac.uk`>

`LeafNode` *LeafNode*

Description

An R6 class for a leaf node in a decision tree representing a clinical state.

Details

It represents a state of being, and is associated with an incremental utility.

Super class

`rdecision::Node` -> `LeafNode`

Methods

Public methods:

- `LeafNode$new()`
- `LeafNode$modvars()`
- `LeafNode$utility()`
- `LeafNode$clone()`

Method `new()`**:** Create a new `LeafNode` object; synonymous with a clinical outcome.

Usage:

```
LeafNode$new(
  label,
  utility = 1,
  interval = as.difftime(365.25, units = "days")
)
```

Arguments:

`label` Character string; a label for the state; must be defined because it is used in tabulations. The label is automatically converted to a syntactically valid (in R) name to ensure it can be used as a column name in a data frame.

`utility` The incremental utility that a user associates with being in the health state (range -Inf to 1) for the interval. Intended for use with cost benefit analysis.

`interval` The time interval over which the `utility` parameters apply, expressed as an R `difftime` object; default 1 year.

Returns: A new `LeafNode` object

Method `modvars()`**:** Find all the model variables of type ModVar that have been specified as values associated with this LeafNode. Includes operands of these `ModVars`, if they are expressions.

Usage:

```
LeafNode$modvars()
```

Returns: A list of `ModVars`.

Method `utility()`**:** Return the incremental utility associated with being in the state for the interval.

Usage:

`LeafNode$utility(expected)`

Arguments:

`expected` Parameter passed to the `value` method of the model variable used to define utility; ignored otherwise.

Returns: Incremental utility (numeric value).

Method `clone()`: The objects of this class are cloneable with this method.

Usage:

`LeafNode$clone(deep = FALSE)`

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Andrew J. Sims <`andrew.sims@newcastle.ac.uk`>

`LogNormModVar` *LogNormModVar*

Description

An R6 class for a model variable with Log Normal uncertainty

Details

A model variable for which the uncertainty in the point estimate can be modelled with a Log Normal distribution. ProbOnto defines seven parametrizations of the log normal distribution. These are linked, allowing the parameters of any one to be derived from any other. All 7 parameterizations require two parameters; their meanings are as follows:

LN1 $p_1 = \mu$, $p_2 = \sigma$, where μ and σ are the mean and standard deviation, both on the log scale.

LN2 $p_1 = \mu$, $p_2 = v$, where μ and v are the mean and variance, both on the log scale.

LN3 $p_1 = m$, $p_2 = \sigma$, where m is the median on the natural scale and σ is the standard deviation on the log scale.

LN4 $p_1 = m$, $p_2 = c_v$, where m is the median on the natural scale and c_v is the coefficient of variation on the natural scale.

LN5 $p_1 = \mu$, $p_2 = \tau$, where μ is the mean on the log scale and τ is the precision on the log scale.

LN6 $p_1 = m$, $p_2 = \sigma_g$, where m is the median on the natural scale and σ_g is the geometric standard deviation on the natural scale.

LN7 $p_1 = \mu_N$, $p1 = \sigma_N$, where μ_N is the mean on the natural scale and σ_N is the standard deviation on the natural scale.

Super class

`rdecision::ModVar` -> `LogNormModVar`

Methods

Public methods:

- `LogNormModVar$new()`
- `LogNormModVar$is_probabilistic()`
- `LogNormModVar$distribution()`
- `LogNormModVar$r()`
- `LogNormModVar$mean()`
- `LogNormModVar$mode()`
- `LogNormModVar$SD()`
- `LogNormModVar$quantile()`
- `LogNormModVar$clone()`

Method `new()`**:** Create a model variable with log normal uncertainty.

Usage:

```
LogNormModVar$new(description, units, p1, p2, parametrization = "LN1")
```

Arguments:

`description` A character string describing the variable.

`units` Units of the quantity; character string.

`p1` First hyperparameter, a measure of location. See 'Details'.

`p2` Second hyperparameter, a measure of spread. See 'Details'.

`parametrization` A character string taking one of the values 'LN1' (default) through 'LN7' (see 'Details').

Returns: A LogNormModVar object.

Method `is_probabilistic()`**:** Tests whether the model variable is probabilistic, i.e. a random variable that follows a distribution, or an expression involving random variables, some of which follow distributions.

Usage:

```
LogNormModVar$is_probabilistic()
```

Returns: TRUE if probabilistic

Method `distribution()`**:** Accessor function for the name of the uncertainty distribution.

Usage:

```
LogNormModVar$distribution()
```

Returns: Distribution name as character string (LN1, LN2 etc).

Method `r()`**:** Draw a random sample from the model variable.

Usage:

```
LogNormModVar$r(n = 1)
```

Arguments:

`n` Number of samples to draw.

Returns: A sample drawn at random.

Method `mean()`**:** Return the expected value of the distribution.

Usage:

```
LogNormModVar$mean()
```

Returns: Expected value as a numeric value.

Method `mode()`**:** Return the point estimate of the variable.

Usage:

```
LogNormModVar$mode()
```

Returns: Point estimate (mode) of the LN distribution.

Method `SD()`**:** Return the standard deviation of the distribution.

Usage:

```
LogNormModVar$SD()
```

Returns: Standard deviation as a numeric value

Method `quantile()`**:** Return the quantiles of the logNormal uncertainty distribution.

Usage:

```
LogNormModVar$quantile(probs)
```

Arguments:

`probs` Vector of probabilities, in range [0,1].

Returns: Vector of quantiles.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
LogNormModVar$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Andrew J. Sims `<andrew.sims@newcastle.ac.uk>`

`ModVar` *ModVar*

Description

An R6 class for a variable in an health economic model

Details

Base class for a variable used in a health economic model. The base class, which is not intended to be directly instantiated by model applications, wraps a numerical value which is used in calculations. The base class provides a framework for creating classes of model variables whose uncertainties are described by statistical distributions parametrized with hyperparameters.

Methods

Public methods:

- `ModVar$new()`
- `ModVar$is_expression()`
- `ModVar$is_probabilistic()`
- `ModVar$description()`
- `ModVar$units()`
- `ModVar$distribution()`
- `ModVar$r()`
- `ModVar$mean()`
- `ModVar$mode()`
- `ModVar$SD()`
- `ModVar$quantile()`
- `ModVar$set()`
- `ModVar$get()`
- `ModVar$clone()`

Method `new()`**:** Create an object of type ‘ModVar‘

Usage:

```
ModVar$new(description, units)
```

Arguments:

`description` A character string description of the variable and its role in the model. This description will be used in a tabulation of the variables linked to a model.

`units` A character string description of the units, e.g. ’GBP’, ’per year’.

Returns: A new ModVar object.

Method `is_expression()`**:** Is this ModVar an expression?

Usage:

```
ModVar$is_expression()
```

Returns: TRUE if it inherits from ExprModVar, FALSE otherwise.

Method `is_probabilistic()`**:** Tests whether the model variable is probabilistic, i.e. a random variable that follows a distribution, or an expression involving random variables, some of which follow distributions.

Usage:

```
ModVar$is_probabilistic()
```

Returns: TRUE if probabilistic #' @description #' Return the current value of the model variable. This will be the #' expected value if the argument to the most recent call to ‘sample‘ #' was TRUE or after creation of the object; otherwise it will return #' a value sampled from the uncertainty distribution. #' @param what Determines what is returned (a character string). Options #' are as follows: #'

#'Point estimate #' A single random sample from the uncertainty distribution #' Mean of the uncertainty distribution #'

#' @return Numeric value of the model variable. value = function(what="pe") # check argument
if (!is.character(what)) rlang::abort("Argument ’what’ must be a character string", class="what_not_string")
# returned requested value v <- as.numeric(NA) if (what=="pe") v <- self$point_estimate() else
if (what=="mean") v <- self$mean() else if (what=="r") v <- self$r(1) else rlang::abort("Argument
’what’ must be (pe|r|mean)", class="unknown_what")
return(v) ,

"pe"mean" **Method** `description()`**:** Accessor function for the description.

Usage:

```
ModVar$description()
```

Returns: Description of model variable as character string.

Method `units()`**:** Accessor function for units.

Usage:

```
ModVar$units()
```

Returns: Description of units as character string.

Method `distribution()`**:** Accessor function for the name of the uncertainty distribution.

Usage:

```
ModVar$distribution()
```

Returns: Distribution name as character string.

Method `r()`**:** Draw a random sample from the model variable.

Usage:

```
ModVar$r(n = 1)
```

Arguments:

`n` Number of samples to draw.

Returns: A sample drawn at random.

Method `mean()`**:** Return the mean value of the distribution.

Usage:

```
ModVar$mean()
```

Returns: Mean value as a numeric value.

Method `mode()`**:** Return the mode of the variable. By default returns NA, which will be the case for most ExprModVar variables, because an arbitrary expression is not guaranteed to be unimodel.

Usage:

```
ModVar$mode()
```

Returns: Mode as a numeric value.

Method `SD()`**:** Return the standard deviation of the distribution.

Usage:

```
ModVar$SD()
```

Returns: Standard deviation as a numeric value

Method `quantile()`**:** Find quantiles of the uncertainty distribution.

Usage:

```
ModVar$quantile(probs)
```

Arguments:

`probs` Numeric vector of probabilities, each in range [0,1].

Returns: Vector of numeric values of the same length as ‘probs‘.

Method `set()`**:** Sets the value of the ModVar that will be returned by subsequent calls to get() until set() is called again.

Usage:

`ModVar$set(expected = FALSE)`

Arguments:

`expected` Logical; TRUE to set the value to the mean of the model variable.

Returns: Updated ModVar.

Method `get()`**:** Gets the value of the ExprModVar that was set by the most recent call to set().

Usage:

`ModVar$get()`

Returns: Value determined by last set().

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

`ModVar$clone(deep = FALSE)`

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Andrew Sims <`andrew.sims@newcastle.ac.uk`>

`Node` *Node*

Description

An R6 class to represent a node in a decision tree

Details

Base class to represent a single node in a decision tree. Objects of base class Node are not expected to be created as model objects.

Methods

Public methods:

- `Node$new()`
- `Node$label()`
- `Node$type()`
- `Node$clone()`

Method `new()`**:** Create new Node object.

Usage:

`Node$new(label = "")`

Arguments:

`label` An optional label for the node.

Returns: A new Node object.

Method `label()`**:** Return the label of the node.

Usage:

`Node$label()`

Returns: Label as a character string. #' @description #' Return list of child nodes (DOM-style) #' @return #' list of child Nodes child_nodes = function() children = list() for (e in private$edges) children <- c(children, e$target())
return(children) , #' @description #' Return list of descendent nodes. #' @return List of descendent nodes, including self. descendantNodes = function() nodes <- list() toLeaf <- function(node) # push current node to path nodes[[length(nodes)+1]] «- node # process child nodes if not leaf if (node$has_child_nodes()) for (child in node$child_nodes()) toLeaf(child)
toLeaf(self) return(nodes) , #' @description #' Is this node the same as the argument? (DOM-style) #' @param otherNode node to compare with this one #' @return TRUE if ‘otherNode‘ is also this one is_same_node = function(otherNode) return(identical(self, otherNode)) ,

Method `type()`**:** node type

Usage:

`Node$type()`

Returns: Node class, as character string #' @description #' Trace and list all pathways ending on leaf nodes which start #' with this node. #' @param choice Name of choice. All pathways are returned if NA. #' @return A list of Path objects. Each member of the list is a #' path from this node to a leaf, limited to those associated #' with choice, if defined. getPathways = function(choice=NA) path <- list() rc <- list() toLeaf <- function(node) # push current node to path path[[length(path)+1]] «- node # leaf reached; store the path if (node$has_child_nodes()) # process child nodes for (child in node$child_nodes()) toLeaf(child)
else p <- Path$new(path) if (!is.na(choice)) if (p$getChoice()==choice) rc[[length(rc)+1]] «- p
else rc[[length(rc)+1]] «- p
# pop current node from path path «- path[1:(length(path)-1)]
toLeaf(self) return(rc) , #' @description #' Return label of edge which links to specified child node #' @param childNode child node to which find label of linking edge #' @return label as character string get_edge_label = function(childNode) rv <- NA ie <- private$whichArrow(childNode)
if (!is.na(ie)) edge <- private$edges[[ie]] rv <- edge$label()
return(rv) , #' @description #' Function to return a list of model variables associated with this node. #' @return #' List of model variables associated with this node. get_modvars = function() return(list()) , #' @description #' Tabulate all model variables associated with this node. #' @param include.descendants If TRUE, model variables associated #' with this node and its descendants are tabulated; otherwise only #' the ones that are associated with this node. #' @param include.operands If TRUE, recursively add model variables which are #' included in expressions in ExprModVars. Default is #' FALSE. #' @return Data frame with one row per model variable, as follows: #'

#'The label given to the variable on creation. #' As given at initialization. #' Units of the variable. #' The uncertainty distribution or an expression. #' Expected value. #' Standard deviation. #' p=0.025 quantile. #' p=0.975 quantile. #' Asterisk if the quantiles and SD were estimated by random sampling. #'

tabulate_modvars = function(include.descendants=FALSE, include.operands=FALSE) # create list of nodes if (include.descendants) nodes <- self$descendantNodes()
else nodes <- list(self)
# list model variables associated with these nodes mvlist <- list() sapply(nodes, FUN=function(n)
mv <- n$get_modvars() if (length(mv) > 0) mvlist «- c(mvlist, unlist(mv))
) # tabulate the model variables DF <- do.call('rbind', lapply(mvlist, FUN=function(x)x$tabulate(include.operands))
) DF <- DF[!duplicated(DF),] # order the table if (nrow(DF) > 0) DF <- DF[order(DF$Label),]

# return the tabulated variables return(DF) , #' @description #' Sample the model variables associated with the node. #' @param expected if TRUE, use the expected value of the model variables in #' the node; otherwise sample from their uncertainty distributions. #' @return Updated Node object sample_modvars = function(expected=FALSE) # get the model variables associated with this node mvlist <- self$get_modvars() # sample them sapply(mvlist, FUN=function(mv) mv$sample(expected)) return(invisible(self))

LaDescriptionDistributionMe$Q9Q5hat Method `clone()`: The objects of this class are cloneable with this method.

Usage:

`Node$clone(deep = FALSE)`

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Andrew Sims `<andrew.sims@newcastle.ac.uk>`

`NormModVar` *NormModVar*

Description

An R6 class for a model variable with Normal uncertainty

Details

A model variable for which the uncertainty in the point estimate can be modelled with a Normal distribution. The hyperparameters of the distribution are the mean (‘mu‘) and the standard deviation (‘sd‘) of the uncertainty distribution. The value of ‘mu‘ is the expected value of the variable.

Super class

`rdecision::ModVar` -> `NormModVar`

Methods

Public methods:

- `NormModVar$new()`
- `NormModVar$is_probabilistic()`
- `NormModVar$distribution()`
- `NormModVar$r()`
- `NormModVar$mean()`
- `NormModVar$SD()`
- `NormModVar$quantile()`
- `NormModVar$clone()`

Method `new()`: Create a model variable with normal uncertainty.

Usage:

`NormModVar$new(description, units, mu, sigma)`

Arguments:

`description` A character string describing the variable.

`units` Units of the quantity; character string.

`mu` Hyperparameter with mean of the Normal distribution for the uncertainty of the variable.

`sigma` Hyperparameter equal to the standard deviation of the normal distribution for the uncertainty of the variable.

Returns: A NormModVar object.

Method `is_probabilistic()`**:** Tests whether the model variable is probabilistic, i.e. a random variable that follows a distribution, or an expression involving random variables, some of which follow distributions.

Usage:

`NormModVar$is_probabilistic()`

Returns: TRUE if probabilistic

Method `distribution()`**:** Accessor function for the name of the uncertainty distribution.

Usage:

`NormModVar$distribution()`

Returns: Distribution name as character string.

Method `r()`**:** Draw a random sample from the model variable. Normally accessed by a call to value(what="r").

Usage:

`NormModVar$r(n = 1)`

Arguments:

`n` Number of samples to draw.

Returns: A sample drawn at random.

Method `mean()`**:** Return the mean value of the distribution.

Usage:

`NormModVar$mean()`

Returns: Expected value as a numeric value.

Method `SD()`**:** Return the standard deviation of the distribution.

Usage:

`NormModVar$SD()`

Returns: Standard deviation as a numeric value

Method `quantile()`**:** Return the quantiles of the Normal uncertainty distribution.

Usage:

`NormModVar$quantile(probs)`

Arguments:

`probs` Vector of probabilities, in range [0,1].

Returns: Vector of quantiles.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

`NormModVar$clone(deep = FALSE)`

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Andrew J. Sims <andrew.sims@newcastle.ac.uk>

`rdecision` *rdecision: Decision Analytic Modelling in Health Economics.*

Description

Classes and functions for modelling healthcare interventions using cohort models (decision trees, Markov models and extended Markov models). It draws on terminology from Briggs, Claxton and Sculpher, "Decision Modelling for Health Economic Evaluation", Oxford University Press, 2006.

`Reaction` *Reaction*

Description

An R6 class to represent a reaction (chance) edge in a decision tree.

Details

A specialism of class Arrow which is used in a decision tree to represent edges with source nodes joined to `ChanceNode`s.

Super classes

`rdecision::Edge` -> `rdecision::Arrow` -> `Reaction`

Methods

Public methods:

- `Reaction$new()`
- `Reaction$modvars()`
- `Reaction$p()`
- `Reaction$cost()`
- `Reaction$benefit()`
- `Reaction$clone()`

Method `new()`**:** Create an object of type 'Reaction'. A probability must be assigned to the edge. Optionally, a cost and a benefit may be associated with traversing the edge. A *payoff* (benefit-cost) is sometimes used in edges of decision trees; the parametrization used here is more general.

Usage:

```
Reaction$new(source, target, p, cost = 0, benefit = 0, label = "")
```

Arguments:

`source` Chance node from which the arrow leaves.

`target` Node which the arrow enters.

`p` Probability

`cost` Cost associated with traversal of this edge.

`benefit` Benefit associated with traversal of the edge.

`label` Character string containing the arrow label.

Returns: A new `Reaction` object.

Method `modvars()`**:** Find all the model variables of type ModVar that have been specified as values associated with this Action. Includes operands of these `ModVars`, if they are expressions.

Usage:

```
Reaction$modvars()
```

Returns: A list of `ModVars`.

Method `p()`**:** Return the current value of the edge probability.

Usage:

```
Reaction$p()
```

Returns: Numeric value in range [0,1].

Method `cost()`**:** Return the cost associated with traversing the edge.

Usage:

```
Reaction$cost()
```

Returns: Cost.

Method `benefit()`**:** Return the benefit associated with traversing the edge.

Usage:

```
Reaction$benefit()
```

Returns: Benefit.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Reaction$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Andrew J. Sims <`andrew.sims@newcastle.ac.uk`>

`Stack` *A stack class.*

Description

An R6 class to represent a stack of objects of any type.

Details

Conventional implementation of a stack. Used extensively in graph algorithms and offered as a separate class for ease of programming and to ensure that implementations of stacks are optimized. By intention, there is only minimal checking of method arguments. This is to maximize performance and because the class is mainly intended for use internally to 'rdecision'.

Methods

Public methods:

- `Stack$new()`
- `Stack$push()`
- `Stack$pop()`
- `Stack$size()`
- `Stack$as_list()`
- `Stack$clone()`

Method `new()`**:** Create a stack.

Usage:

```
Stack$new()
```

Returns: A new Stack object.

Method `push()`**:** Push an item onto the stack.

Usage:

```
Stack$push(x)
```

Arguments:

`x` The item to push onto the top of the stack. It should be of the same class as items previously pushed on to the stack. It is not checked.

Returns: An updated Stack object

Method `pop()`**:** Pop an item from the stack. Note that stack underflow is not checked.

Usage:

```
Stack$pop()
```

Returns: The item previously at the top of the stack.

Method `size()`**:** Gets the number of items on the stack.

Usage:

```
Stack$size()
```

Returns: Number of items.

Method `as_list()`**:** Inspect items in the stack.

Usage:

```
Stack$as_list()
```

Returns: A list of items.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Stack$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Andrew Sims `<andrew.sims@newcastle.ac.uk>`

Index