Package ‘Mediana’

October 12, 2022

Type Package

Title Clinical Trial Simulations

Version 1.0.8

Date 2019-05-08

Author Gautier Paux, Alex Dmitrienko.

Maintainer Gautier Paux <gautier@paux.fr>

BugReports https://github.com/gpaux/Mediana/issues

Description Provides a general framework for clinical trial simulations based on the Clinical Scenario Evaluation (CSE) approach. The package supports a broad class of data models (including clinical trials with continuous, binary, survival-type and count-type endpoints as well as multivariate outcomes that are based on combinations of different endpoints), analysis strategies and commonly used evaluation criteria.

Imports doParallel, doRNG, foreach, MASS, mvtmvnorm, stats, survival, utils

License GPL-2

URL http://gpaux.github.io/Mediana/

RoxygenNote 6.1.1

Encoding UTF-8

Suggests flextable, knitr, officer, rmarkdown, pander

VignetteBuilder knitr

NeedsCompilation no

Repository CRAN

Date/Publication 2019-05-08 13:20:03 UTC

R topics documented:

Mediana-package .................................................. 2
AdjustCIs .......................................................... 6
Description

Provides a general framework for clinical trial simulations based on the Clinical Scenario Evaluation (CSE) approach. The package supports a broad class of data models (including clinical trials with continuous, binary, survival-type and count-type endpoints as well as multivariate outcomes that are based on combinations of different endpoints), analysis strategies and commonly used evaluation criteria.
Details

Package: Mediana
Type: Package
Version: 1.0.8
Date: 2019-05-08
License: GPL-2

Author(s)
Gautier Paux, Alex Dmitrienko
Maintainer: Gautier Paux <gautier@paux.fr>

References
http://gpaux.github.io/Mediana/

Examples
## Not run:
# Clinical trial in patients with rheumatoid arthritis

# Variable types
var.type = parameters("BinomDist", "NormalDist")

# Outcome distribution parameters
plac.par = parameters(parameters(prop = 0.3),
parameters(mean = -0.10, sd = 0.5))

dosel.par1 = parameters(parameters(prop = 0.40),
parameters(mean = -0.20, sd = 0.5))
dosel.par2 = parameters(parameters(prop = 0.45),
parameters(mean = -0.25, sd = 0.5))

dosel.par3 = parameters(parameters(prop = 0.50),
parameters(mean = -0.30, sd = 0.5))

doseh.par1 = parameters(parameters(prop = 0.50),
parameters(mean = -0.30, sd = 0.5))

doseh.par2 = parameters(parameters(prop = 0.55),
parameters(mean = -0.35, sd = 0.5))

doseh.par3 = parameters(parameters(prop = 0.60),
parameters(mean = -0.40, sd = 0.5))

# Correlation between two endpoints
corr.matrix = matrix(c(1.0, 0.5,
0.5, 1.0), 2, 2)

# Outcome parameter set 1
outcome1.plac = parameters(type = var.type,
par = plac.par,
corr = corr.matrix)
outcome1.dosel = parameters(type = var.type,
par = dosel.par1,
corr = corr.matrix)
outcome1.doseh = parameters(type = var.type,
par = doseh.par1,
corr = corr.matrix)

# Outcome parameter set 2
outcome2.plac = parameters(type = var.type,
par = plac.par,
corr = corr.matrix)
outcome2.dosel = parameters(type = var.type,
par = dosel.par2,
corr = corr.matrix)
outcome2.doseh = parameters(type = var.type,
par = doseh.par2,
corr = corr.matrix)

# Outcome parameter set 3
outcome3.plac = parameters(type = var.type,
par = plac.par,
corr = corr.matrix)
outcome3.doseh = parameters(type = var.type,
par = doseh.par3,
corr = corr.matrix)
outcome3.dosel = parameters(type = var.type,
par = dosel.par3,
corr = corr.matrix)

# Data model
data.model = DataModel() +
OutcomeDist(outcome.dist = "MVMixedDist") +
SampleSize(c(100, 120)) +
Sample(id = list("Plac ACR20", "Plac HAQ-DI"),
outcome.par = parameters(outcome1.plac, outcome2.plac, outcome3.plac)) +
Sample(id = list("DoseL ACR20", "DoseL HAQ-DI"),
outcome.par = parameters(outcome1.dosel, outcome2.dosel, outcome3.dosel)) +
Sample(id = list("DoseH ACR20", "DoseH HAQ-DI"),
outcome.par = parameters(outcome1.doseh, outcome2.doseh, outcome3.doseh))

family = families(family1 = c(1, 2), family2 = c(3, 4))
component.procedure = families(family1 = "HolmAdj", family2 = "HolmAdj")
gamma = families(family1 = 0.8, family2 = 1)

# Tests to which the multiplicity adjustment will be applied
test.list = tests("Pl vs DoseH - ACR20",
"Pl vs DoseL - ACR20",
"Pl vs DoseH - HAQ-DI",
"Pl vs DoseL - HAQ-DI")

# Analysis model
analysis.model = AnalysisModel() +
  MultAdjProc(proc = "MultipleSequenceGatekeepingAdj",
               par = parameters(family = family,
                                proc = component.procedure,
                                gamma = gamma),
               tests = test.list) +
  Test(id = "Pl vs DoseL - ACR20",
       method = "PropTest",
       samples = samples("Plac ACR20", "DoseL ACR20")) +
  Test(id = "Pl vs DoseH - ACR20",
       method = "PropTest",
       samples = samples("Plac ACR20", "DoseH ACR20")) +
  Test(id = "Pl vs DoseL - HAQ-DI",
       method = "TTest",
       samples = samples("DoseL HAQ-DI", "Plac HAQ-DI")) +
  Test(id = "Pl vs DoseH - HAQ-DI",
       method = "TTest",
       samples = samples("DoseH HAQ-DI", "Plac HAQ-DI"))

# Evaluation model
evaluation.model = EvaluationModel() +
  Criterion(id = "Marginal power",
            method = "MarginalPower",
            tests = tests("Pl vs DoseL - ACR20",
                           "Pl vs DoseH - ACR20",
                           "Pl vs DoseL - HAQ-DI",
                           "Pl vs DoseH - HAQ-DI"),
            labels = c("Pl vs DoseL - ACR20",
                       "Pl vs DoseH - ACR20",
                       "Pl vs DoseL - HAQ-DI",
                       "Pl vs DoseH - HAQ-DI"),
            par = parameters(alpha = 0.025)) +
  Criterion(id = "Disjunctive power - ACR20",
            method = "DisjunctivePower",
            tests = tests("Pl vs DoseL - ACR20",
                           "Pl vs DoseH - ACR20"),
            par = parameters(alpha = 0.025))
AdjustCIs function

Description

Computation of simultaneous confidence intervals for selected multiple testing procedures based on univariate p-values (Bonferroni, Holm and fixed-sequence procedures) and commonly used parametric multiple testing procedures (single-step and step-down Dunnett procedures)

Usage

AdjustCIs(est, proc, par = NA)
Arguments

- `est`: defines the point estimates.
- `proc`: defines the multiple testing procedure. Several procedures are already implemented in the Mediana package (listed below, along with the required or optional parameters to specify in the `par` argument):
- `par`: defines the parameters associated to the multiple testing procedure.

Details

This function computes one-sided simultaneous confidence limits for the Bonferroni, Holm (Holm, 1979) and fixed-sequence (Westfall and Krishen, 2001) procedures in general one-sided hypothesis testing problems (equally or unequally weighted null hypotheses), as well as for the single-step Dunnett procedure (Dunnett, 1955) and step-down Dunnett procedure (Naik, 1975; Marcus, Peritz and Gabriel, 1976) in one-sided hypothesis testing problems with a balanced one-way layout and equally weighted null hypotheses.

For non-parametric procedure, the simultaneous confidence intervals are computed using the methods developed in Hsu and Berger (1999), Strassburger and Bretz (2008) and Guilbaud (2008). For more information on the algorithms used in the function, see Dmitrienko et al. (2009, Section 2.6).

For the Dunnett procedures, the simultaneous confidence intervals are computed using the methods developed in Bofinger (1987) and Stefansson, Kim and Hsu (1988). For more information on the algorithms used in the function, see Dmitrienko et al. (2009, Section 2.7).

Value

Return a vector of lower simultaneous confidence limits.

References

http://gpaux.github.io/Mediana/


**See Also**

See Also `MultAdjProc` and `AdjustPvalues`.

**Examples**

# Consider a clinical trial conducted to evaluate the effect of three
# doses of a treatment compared to a placebo with respect to a normally
# distributed endpoint

# Three null hypotheses of no effect are tested in the trial:
# Null hypothesis H1: No difference between Dose 1 and Placebo
# Null hypothesis H2: No difference between Dose 2 and Placebo
# Null hypothesis H3: No difference between Dose 3 and Placebo

# Null hypotheses of no treatment effect are equally weighted
weight<-c(1/3,1/3,1/3)

# Treatment effect estimates (mean dose-placebo differences)
est<-c(2.3,2.5,1.9)

# Pooled standard deviation
sd<-rep(9.5,3)

# Study design is balanced with 180 patients per treatment arm
n<-180

# Bonferroni, Holm, Hochberg, Hommel and Fixed-sequence procedure
proc = c("BonferroniAdj", "HolmAdj", "FixedSeqAdj", "DunnettAdj", "StepDownDunnettAdj")

# Equally weighted
sapply(proc, function(x) {AdjustCIs(est, 
proc = x,
par = parameters(sd = sd, 
n = n, 
covprob = 0.975, 
weight = weight))})
AdjustPvalues

AdjustPvalues function

Description

Computation of adjusted p-values for commonly used multiple testing procedures based on univariate p-values (Bonferroni, Holm, Hommel, Hochberg, fixed-sequence and Fallback procedures), commonly used parametric multiple testing procedures (single-step and step-down Dunnett procedures) and multistage gatepeaking procedure.

Usage

AdjustPvalues(pval, proc, par = NA)

Arguments

pval

defines the raw p-values.

proc

defines the multiple testing procedure. Several procedures are already implemented in the Mediana package (listed below, along with the required or optional parameters to specify in the par argument):

- FixedSeqAdj: Fixed-sequence procedure.
- MultipleSequenceGatekeepingAdj: Family of multiple-sequence gatekeeping procedures. Required parameters: family, proc, gamma.

par
defines the parameters associated to the multiple testing procedure
Details

This function can be used to adjust p-values according to a multiple testing procedure defined in the `proc` argument.

This function computes adjusted p-values and generates decision rules for the Bonferroni, Holm (Holm, 1979), Hommel (Hommel, 1988), Hochberg (Hochberg, 1988), fixed-sequence (Westfall and Krishen, 2001) and Fallback (Wiens, 2003; Wiens and Dmitrienko, 2005) procedures. The adjusted p-values are computed using the closure principle (Marcus, Peritz and Gabriel, 1976) in general hypothesis testing problems (equally or unequally weighted null hypotheses). For more information on the algorithms used in the function, see Dmitrienko et al. (2009, Section 2.6).

This function computes adjusted p-values for the single-step Dunnett procedure (Dunnett, 1955) and step-down Dunnett procedure (Naik, 1975; Marcus, Peritz and Gabriel, 1976) in one-sided hypothesis testing problems with a balanced one-way layout and equally weighted null hypotheses. For the Dunnett procedures, it is assumed that the test statistics follow a t distribution. For more information on the algorithms used in the function, see Dmitrienko et al. (2009, Section 2.7).

This function computes adjusted p-values and generates decision rules for multistage parallel gatekeeping procedures in hypothesis testing problems with multiple families of null hypotheses (null hypotheses are assumed to be equally weighted within each family) based on the methodology presented in Dmitrienko, Tamhane and Wiens (2008), Dmitrienko, Kordzakhia and Tamhane (2011) and Dmitrienko, Kordzakhia and Brechenmacher (2016). For more information on parallel gatekeeping procedures (computation of adjusted p-values, independence condition, etc), see Dmitrienko and Tamhane (2009, Section 5.4).

Value

Return a vector of adjusted p-values.

References

http://gpaux.github.io/Mediana/


AdjustPvalues


See Also

See Also MultAdjProc and AdjustCIs.

Examples

# Bonferroni, Holm, Hochberg, Hommel and Fixed-sequence procedure
proc = c("BonferroniAdj", "HolmAdj", "HochbergAdj", "HommelAdj", "FixedSeqAdj", "FallbackAdj")
rawp = c(0.012, 0.009, 0.023)

# Equally weighted
sapply(proc, function(x) {AdjustPvalues(rawp, proc = x)})

# Unequally weighted (no effect on the fixed-sequence procedure)
sapply(proc, function(x) {AdjustPvalues(rawp, proc = x,
par = parameters(weight = c(1/2, 1/4, 1/4))}))

# Dunnett procedures
# Compute one-sided adjusted p-values for the single-step Dunnett procedure
AdjustPvalues

# Three null hypotheses of no effect are tested in the trial:
# Null hypothesis H1: No difference between Dose 1 and Placebo
# Null hypothesis H2: No difference between Dose 2 and Placebo
# Null hypothesis H3: No difference between Dose 3 and Placebo

# Treatment effect estimates (mean dose-placebo differences)
est = c(2.3, 2.5, 1.9)

# Pooled standard deviation
sd = 9.5

# Study design is balanced with 180 patients per treatment arm
n = 180

# Standard errors
stderr = rep(sd*sqrt(2/n), 3)

# T-statistics associated with the three dose-placebo tests
stat = est/stderror

# One-sided pvalue
rawp = 1-pt(stat, 2*(n-1))

# Adjusted p-values based on the Dunnett procedures
# (assuming that each test statistic follows a t distribution)
AdjustPvalues(rawp, proc = "DunnettAdj", par = parameters(n = n))
AdjustPvalues(rawp, proc = "StepDownDunnettAdj", par = parameters(n = n))

# Parallel gatekeeping
# Consider a clinical trial with two families of null hypotheses
# Family 1: Primary null hypotheses (one-sided p-values)
# H1 (Endpoint 1), p1=0.0082
# H2 (Endpoint 2), p2=0.0174
# Family 2: Secondary null hypotheses (one-sided p-values)
# H3 (Endpoint 3), p3=0.0042
# H4 (Endpoint 4), p4=0.0180

# Define raw p-values
rawp <- c(0.0082, 0.0174, 0.0042, 0.0180)

# Define hypothesis included in each family
family = families(family1 = c(1, 2),
                   family2 = c(3, 4))

# Define component procedure of each family
component.procedure = families(family1 = "HolmAdj",
                               family2 = "HolmAdj")

# Truncation parameter of each family
gamma = families(family1 = 0.5,
                 family2 = 1)

adjustp = AdjustPvalues(rawp,
proc = "ParallelGatekeepingAdj",
par = parameters(family = family,
    proc = component.procedure,
    gamma = gamma))

---

**Description**

`AnalysisModel()` initializes an object of class `AnalysisModel`.

**Usage**

`AnalysisModel(...)`

**Arguments**

`...` defines the arguments passed to create the object of class `AnalysisModel`.

**Details**

Analysis models define statistical methods that are applied to the study data in a clinical trial. `AnalysisModel()` is used to create an object of class `AnalysisModel` incrementally, using the `+` operator to add objects to the existing `AnalysisModel` object. The advantage is to explicitly define which objects are added to the `AnalysisModel` object. Initialization with `AnalysisModel()` is highly recommended.

Objects of class `Test`, `MultAdjProc`, `MultAdjStrategy`, `MultAdj` and `Statistic` can be added to an object of class `AnalysisModel`.

**References**

[http://gpaux.github.io/Mediana/](http://gpaux.github.io/Mediana/)

**See Also**

See Also `Test`, `MultAdjProc`, `MultAdjStrategy`, `MultAdj` and `Statistic`.

**Examples**

```r
## Initialize an AnalysisModel and add objects to it
analysis.model = AnalysisModel() +
    Test(id = "Placebo vs treatment",
         samples = samples("Placebo", "Treatment"),
         method = "TTest")
```
AnalysisStack

**Description**

This function generates analysis results according to the specified data and analysis models.

**Usage**

```r
AnalysisStack(data.model,
              analysis.model,
              sim.parameters)
```

**Arguments**

- `data.model`: defines a `DataModel` object.
- `analysis.model`: defines a `AnalysisModel` object.
- `sim.parameters`: defines a `SimParameters` object.

**Value**

This function generates an analysis stack according to the data and analysis models and the simulation parameters objects. The object returned by the function is a `AnalysisStack` object containing:

- `description`: a description of the object.
- `analysis.set`: a list of size n.sims defined in the `SimParameters` object. This list contains the analysis results generated for each data scenario (first level), and for each test and statistic defined in the `AnalysisModel` object. The results generated for the ith simulation runs and the jth data scenario are stored in `analysis.stack$analysis.set[[i]][[j]]$result` (where `analysis.stack` is a `AnalysisStack` object). Then, this list is composed of three lists:
  - `tests`: return the unadjusted p-values for to the tests defined in the `AnalysisModel` object.
  - `statistic`: return the statistic defined in the `AnalysisModel` object.
  - `test.adjust`: return a list of adjusted p-values according to the multiple testing procedure defined in the `AnalysisModel` object. The length of this list corresponds to the number of `MultAdjProc` objects defined in the `AnalysisModel` object. Note that if no `MultAdjProc` objects have been defined, this list contains the unadjusted p-values.
- `analysis.scenario.grid`: a data frame indicating all data and analysis scenarios according to the `DataModel` and `AnalysisModel` objects.
- `analysis.structure`: a list containing the analysis structure according to the `AnalysisModel` object.
- `sim.parameters`: a list containing the simulation parameters according to `SimParameters` object.
A specific analysis set of a AnalysisStack object can be extracted using the ExtractAnalysisStack function.

References

http://gpaux.github.io/Mediana/

See Also

See Also DataModel, AnalysisModel and SimParameters and ExtractAnalysisStack.

Examples

```r
## Not run:  
# Generation of an AnalysisStack object  
###################################################################
# Outcome parameter set 1  
outcome1.placebo = parameters(mean = 0, sd = 70)  
outcome1.treatment = parameters(mean = 40, sd = 70)

# Outcome parameter set 2  
outcome2.placebo = parameters(mean = 0, sd = 70)  
outcome2.treatment = parameters(mean = 50, sd = 70)

# Data model  
case.study1.data.model = DataModel() +  
  OutcomeDist(outcome.dist = "NormalDist") +  
  SampleSize(c(50, 55, 60, 65, 70)) +  
  Sample(id = "Placebo",  
    outcome.par = parameters(outcome1.placebo, outcome2.placebo)) +  
  Sample(id = "Treatment",  
    outcome.par = parameters(outcome1.treatment, outcome2.treatment))

# Analysis model  
case.study1.analysis.model = AnalysisModel() +  
  Test(id = "Placebo vs treatment",  
    samples = samples("Placebo", "Treatment"),  
    method = "TTest") +  
  Statistic(id = "Mean Treatment",  
    method = "MeanStat",  
    samples = samples("Treatment"))

# Simulation Parameters  
case.study1.sim.parameters = SimParameters(n.sims = 1000,  
  proc.load = 2,  
  seed = 42938001)

# Generate results  
case.study1.analysis.stack = AnalysisStack(data.model = case.study1.data.model,  
  analysis.model = case.study1.analysis.model,  
  simulation.parameters = case.study1.sim.parameters)
```
Criterion

Criterion object

Description

This function creates an object of class Criterion which can be added to an object of class EvaluationModel.

Usage

Criterion(id, method, tests = NULL, statistics = NULL, par = NULL, labels)

Arguments

id defines the ID of the Criterion object.
method defines the method used by the Criterion object.
tests defines the test(s) used by the Criterion object.
statistics defines the statistic(s) used by the Criterion object.
par defines the parameter(s) of the method argument of the Criterion object.
labels defines the label(s) of the results.

Details

Objects of class Criterion are used in objects of class EvaluationModel to specify the criteria that will be applied to the Clinical Scenario. Several objects of class Criterion can be added to an object of class EvaluationModel.

Mandatory arguments are id, method, labels and tests and/or statistics.

method argument defines the criterion’s method. Several methods are already implemented in the Mediana package (listed below, along with the required parameters to define in the par parameter):

sim.parameters = case.study1.sim.parameters)

# Print the analysis results generated in the 100th simulation run
# for the 2nd data scenario for both samples
case.study1.analysis.stack$analysis.set[[100]][[2]]

# Extract the same set of data
case.study1.extracted.analysis.stack =
  ExtractAnalysisStack(analysis.stack = case.study1.analysis.stack,
                        data.scenario = 2,
                        simulation.run = 100)

# A careful attention should be paid on the index of the result.
# As only one data.scenario has been requested
# the result for data.scenario = 2 is now in the first position ($analysis.set[[1]][[1]]).

## End(Not run)
CSE

- **MarginalPower**: generate the marginal power of all tests defined in the test argument. Required parameter: alpha.
- **WeightedPower**: generate the weighted power of all tests defined in the test argument. Required parameters: alpha and weight.
- **DisjunctivePower**: generate the disjunctive power (probability to reject at least one hypothesis defined in the test argument). Required parameter: alpha.
- **ConjunctivePower**: generate the conjunctive power (probability to reject all hypotheses defined in the test argument). Required parameter: alpha.
- **ExpectedRejPower**: generate the expected number of rejected hypotheses. Required parameter: alpha.

References

[http://gpaux.github.io/Mediana/](http://gpaux.github.io/Mediana/)

See Also

See Also `AnalysisModel`.

Examples

```r
## Add a Criterion to an EvaluationModel object
evaluation.model = EvaluationModel() +
  Criterion(id = "Marginal power",
            method = "MarginalPower",
            tests = tests("Placebo vs treatment"),
            labels = c("Placebo vs treatment"),
            par = parameters(alpha = 0.025))
```

CSE Clinical Scenario Evaluation

Description

This function is used to perform the Clinical Scenario Evaluation according to the objects of class `DataModel, AnalysisModel` and `EvaluationModel` specified respectively in the arguments `data, analysis` and `evaluation` of the function.

Usage

```r
CSE(data, analysis, evaluation, simulation)
```

Arguments

- `data` defines a `DataModel` or a `DataStack` object
- `analysis` defines an `AnalysisModel` object
- `evaluation` defines an `EvaluationModel` object
- `simulation` defines a `SimParameters` object
Value

The CSE function returns a list containing:

- simulation.results
  a data frame containing the results of the simulations for each scenario.
- analysis.scenario.grid
  a data frame containing the grid of the combination of data and analysis scenarios.
- data.structure
  a list containing the data structure according to the DataModel object.
- analysis.structure
  a list containing the analysis structure according to the AnalysisModel object.
- evaluation.structure
  a list containing the evaluation structure according to the EvaluationModel object.
- sim.parameters
  a list containing the simulation parameters according to SimParameters object.
- timestamp
  a list containing information about the start time, end time and duration of the simulation runs.

References

http://gpaux.github.io/Mediana/

See Also

See Also DataModel, DataStack, AnalysisModel, EvaluationModel, SimParameters.

Examples

```r
# Not run:
# Outcome parameter set 1
outcome1.placebo = parameters(mean = 0, sd = 70)
outcome1.treatment = parameters(mean = 40, sd = 70)

# Outcome parameter set 2
outcome2.placebo = parameters(mean = 0, sd = 70)
outcome2.treatment = parameters(mean = 50, sd = 70)

# Data model
case.study1.data.model = DataModel() +
  OutcomeDist(outcome.dist = "NormalDist") +
  SampleSize(c(50, 55, 60, 65, 70)) +
  Sample(id = "Placebo",
         outcome.par = parameters(outcome1.placebo, outcome2.placebo)) +
  Sample(id = "Treatment",
         outcome.par = parameters(outcome1.treatment, outcome2.treatment))
```
# Analysis model
case.study1.analysis.model = AnalysisModel() +
    Test(id = "Placebo vs treatment",
         samples = samples("Placebo", "Treatment"),
         method = "TTest")

# Evaluation model
case.study1.evaluation.model = EvaluationModel() +
    Criterion(id = "Marginal power",
              method = "MarginalPower",
              tests = tests("Placebo vs treatment"),
              labels = c("Placebo vs treatment"),
              par = parameters(alpha = 0.025))

# Simulation Parameters
case.study1.sim.parameters = SimParameters(n.sims = 1000, proc.load = 2, seed = 42938001)

# Perform clinical scenario evaluation
case.study1.results = CSE(case.study1.data.model,
                            case.study1.analysis.model,
                            case.study1.evaluation.model,
                            case.study1.sim.parameters)

# Summary of the simulation results
summary(case.study1.results)

# Get the data generated for the simulation
case.study1.data.stack = DataStack(data.model = case.study1.data.model,
                                    sim.parameters = case.study1.sim.parameters)

## End(Not run)

## Not run:
# Alternatively, a DataStack object can be used in the CSE function
# (not recommended as the computational time is increased)

# Generate data
case.study1.data.stack = DataStack(data.model = case.study1.data.model,
                                    sim.parameters = case.study1.sim.parameters)

# Perform clinical scenario evaluation with data stack
case.study1.results = CSE(case.study1.data.stack,
                            case.study1.analysis.model,
                            case.study1.evaluation.model,
                            case.study1.sim.parameters)

## End(Not run)
CustomLabel

CustomLabel object

Description

This function creates an object of class CustomLabel which can be added to an object of class PresentationModel.

Usage

CustomLabel(param, label)

Arguments

param defines a parameter for which the labels will be assigned.
lable defines the label(s) to assign to the parameter.

Details

Objects of class CustomLabel are used in objects of class PresentationModel to specify the labels that will be assigned to the parameter. Several objects of class CustomLabel can be added to an object of class PresentationModel.

The argument param only accepts the following values:

- "sample.size"
- "event"
- "outcome.parameter"
- "design.parameter"
- "multiplicity.adjustment"

References

http://gpaux.github.io/Mediana/

See Also

See Also PresentationModel.

Examples

```r
# Create a PresentationModel object with customized label
presentation.model = PresentationModel() +
    Section(by = "outcome.parameter") +
    Table(by = "sample.size") +
    CustomLabel(param = "sample.size",
                label= paste0("N = ",c(50, 55, 60, 65, 70))) +
    CustomLabel(param = "outcome.parameter", label=c("Standard 1", "Standard 2"))
```
DataModel

DataModel object

Description

DataModel() initializes an object of class DataModel.

Usage

DataModel(...)

Arguments

... defines the arguments passed to create the object of class DataModel.

Details

Data models define the process of generating patients data in a clinical trial.

DataModel() is used to create an object of class DataModel incrementally, using the ‘+’ operator to add objects to the existing DataModel object. The advantage is to explicitly define which objects are added to the DataModel object. Initialization with DataModel() is highly recommended.

Objects of class OutcomeDist, SampleSize, Sample, Event and Design can be added to an object of class DataModel.

References

http://gpaux.github.io/Mediana/

See Also

See Also OutcomeDist, SampleSize, Sample and Design.

Examples

# Outcome parameter set 1
outcome1.placebo = parameters(mean = 0, sd = 70)
outcome1.treatment = parameters(mean = 40, sd = 70)

# Outcome parameter set 2
outcome2.placebo = parameters(mean = 0, sd = 70)
outcome2.treatment = parameters(mean = 50, sd = 70)

# Data model
data.model = DataModel() +
    OutcomeDist(outcome.dist = "NormalDist") +
    SampleSize(c(50, 55, 60, 65, 70)) +
    Sample(id = "Placebo", outcome.par = parameters(outcome1.placebo, outcome2.placebo)) +
Sample(id = "Treatment",
          outcome.par = parameters(outcome1.treatment, outcome2.treatment))

DataStack

DataStack object

Description

This function generates data according to the specified data model.

Usage

DataStack(data.model,
           sim.parameters)

Arguments

data.model defines a DataModel object.
sim.parameters defines a SimParameters object.

Value

This function generates a data stack according to the data model and the simulation parameters objects. The object returned by the function is a DataStack object containing:

description a description of the object.
data.set a list of size n.sims defined in the sim.parameters object. This list contains the data generated for each data scenario (data.scenario level) and each sample (sample level). The data generated for the i-th simulation run, the j-th data scenario and the k-th sample is stored in data.stack$data.set[[i]]$data.scenario[[j]]$sample[[k]] where data.stack is a DataStack object.
data.scenario.grid a data frame indicating all data scenarios according to the DataModel object.
data.structure a list containing the data structure according to the DataModel object.
sim.parameters a list containing the simulation parameters according to SimParameters object.

A specific data.set of a DataStack object can be extracted using the ExtractDataStack function.

References

http://gpaux.github.io/Mediana/

See Also

See Also DataModel and SimParameters and ExtractDataStack.
### Examples

```r
## Not run:
# Generation of a DataStack object
#################################
# Outcome parameter set 1
outcome1.placebo = parameters(mean = 0, sd = 70)
outcome1.treatment = parameters(mean = 40, sd = 70)
# Outcome parameter set 2
outcome2.placebo = parameters(mean = 0, sd = 70)
outcome2.treatment = parameters(mean = 50, sd = 70)

# Data model
case.study1.data.model = DataModel() +
  OutcomeDist(outcome.dist = "NormalDist") +
  SampleSize(c(50, 55, 60, 65, 70)) +
  Sample(id = "Placebo",
    outcome.par = parameters(outcome1.placebo, outcome2.placebo)) +
  Sample(id = "Treatment",
    outcome.par = parameters(outcome1.treatment, outcome2.treatment))

# Simulation Parameters
case.study1.sim.parameters = SimParameters(n.sims = 1000,
  proc.load = 2,
  seed = 42938001)

# Generate data
case.study1.data.stack = DataStack(data.model = case.study1.data.model,
  sim.parameters = case.study1.sim.parameters)

# Print the data set generated in the 100th simulation run
# for the 2nd data scenario for both samples
case.study1.data.stack$data.set[[100]]$data.scenario[[2]]

# Extract the same set of data
case.study1.extracted.data.stack = ExtractDataStack(data.stack = case.study1.data.stack,
  data.scenario = 2,
  simulation.run = 100)

# The same dataset can be obtained using
case.study1.extracted.data.stack$data.set[[1]]$data.scenario[[1]]$sample
# A careful attention should be paid on the index of the result.
# As only one data.scenario has been requested
# the result for data.scenario = 2 is now in the first position (data.scenario[[1]]).

## End(Not run)
```

## Not run:
# Use of a DataStack object in the CSE function
##############################################
```r
# Outcome parameter set 1
outcome1.placebo = parameters(mean = 0, sd = 70)
ooutcome1.treatment = parameters(mean = 40, sd = 70)

# Outcome parameter set 2
outcome2.placebo = parameters(mean = 0, sd = 70)
ooutcome2.treatment = parameters(mean = 50, sd = 70)

# Data model
case.study1.data.model = DataModel() +
  OutcomeDist(outcome.dist = "NormalDist") +
  SampleSize(c(50, 55, 60, 65, 70)) +
  Sample(id = "Placebo",
    outcome.par = parameters(outcome1.placebo, outcome2.placebo)) +
  Sample(id = "Treatment",
    outcome.par = parameters(outcome1.treatment, outcome2.treatment))

# Simulation Parameters
case.study1.sim.parameters = SimParameters(n.sims = 1000, proc.load = 2, seed = 42938001)

# Generate data
case.study1.data.stack = DataStack(data.model = case.study1.data.model,
  sim.parameters = case.study1.sim.parameters)

# Analysis model
case.study1.analysis.model = AnalysisModel() +
  Test(id = "Placebo vs treatment",
    samples = samples("Placebo", "Treatment"),
    method = "TTest")

# Evaluation model
case.study1.evaluation.model = EvaluationModel() +
  Criterion(id = "Marginal power",
    method = "MarginalPower",
    tests = tests("Placebo vs treatment"),
    labels = c("Placebo vs treatment"),
    par = parameters(alpha = 0.025))

# Simulation Parameters

# Perform clinical scenario evaluation
case.study1.results = CSE(case.study1.data.stack,
  case.study1.analysis.model,
  case.study1.evaluation.model,
  case.study1.sim.parameters)

## End(Not run)
```
Design

Design object

Description

This function creates an object of class Design which can be added to an object of class DataModel.

Usage

Design(enroll.period = NULL,
enroll.dist = NULL,
enroll.dist.par = NULL,
followup.period = NULL,
study.duration = NULL,
dropout.dist = NULL,
dropout.dist.par = NULL)

Arguments

enroll.period defines the length of the enrollment period.
enroll.dist defines the enrollment distribution.
enroll.dist.par defines the parameters of the enrollment distribution (optional).
followup.period defines the length of the follow-up period for each patient in study designs
with a fixed follow-up period, i.e., the length of time from the enrollment to
planned discontinuation is constant across patients. The user must specify ei-
ther followup.period or study.duration.
study.duration defines the total study duration in study designs with a variable follow-up period.
The total study duration is defined as the length of time from the enrollment of
the first patient to the discontinuation of the last patient.
dropout.dist defines the dropout distribution.
dropout.dist.par defines the parameters of the dropout distribution.

Details

Objects of class Design are used in objects of class DataModel to specify the design parameters
used in event-driven designs if the user is interested in modeling the enrollment (or accrual) and
dropout (or loss to follow up) processes that will be applied to the Clinical Scenario. Several objects
of class Design can be added to an object of class DataModel.

Note that the length of the enrollment period, total study duration and follow-up periods are mea-
sured using the same time units.

If enroll.dist = "UniformDist", the enroll.dist.par should be let to NULL (then enrollment
distribution will be uniform during the enrollment period).
If enroll.dist = "BetaDist", the enroll.dist.par should contain the parameter of the beta distribution (a and b). These parameters must be derived according to the expected enrollment at a specific timepoint. For example, if half the patients are expected to be enrolled at 75% of the enrollment period, the beta distribution is a Beta(log(0.5)/log(0.75), 1). Generally, let q be the proportion of enrolled patients at p% of the enrollment period, the Beta distribution can be derived as follows:

- If q > p, the Beta distribution is Beta(a,1) with \( a = \frac{\log(p)}{\log(q)} \)
- If q < p, the Beta distribution is Beta (1,b) with \( b = \frac{\log(1-p)}{\log(1-q)} \)
- Otherwise the Beta distribution is Beta(1,1)

If dropout.dist = "UniformDist", the dropout.dist.par should contain the dropout rate. This parameter must be specified using the prop parameter, such as dropout.dist.par = parameters(prop = 0.1) for a 10% dropout rate.

**References**

http://gpaux.github.io/Mediana/

**See Also**

See Also DataModel.

**Examples**

```r
## Create DataModel object with a Design Object
data.model = DataModel() +
  Design(enroll.period = 9,
         study.duration = 21,
         enroll.dist = "UniformDist",
         dropout.dist = "ExpoDist",
         dropout.dist.par = parameters(rate = 0.0115))

## Create DataModel object with several Design Objects
design1 = Design(enroll.period = 9,
                 study.duration = 21,
                 enroll.dist = "UniformDist",
                 dropout.dist = "ExpoDist",
                 dropout.dist.par = parameters(rate = 0.0115))

design2 = Design(enroll.period = 18,
                 study.duration = 24,
                 enroll.dist = "UniformDist",
                 dropout.dist = "ExpoDist",
                 dropout.dist.par = parameters(rate = 0.0115))

data.model = DataModel() +
            design1 +
            design2
```
Description

`EvaluationModel()` initializes an object of class `EvaluationModel`.

Usage

`EvaluationModel(...)`

Arguments

... defines the arguments passed to create the object of class `EvaluationModel`.

Details

Evaluation models are used within the Mediana package to specify the measures (metrics) for evaluating the performance of the selected clinical scenario (combination of data and analysis models).

`EvaluationModel()` is used to create an object of class `EvaluationModel` incrementally, using the ` '+' ` operator to add objects to the existing `EvaluationModel` object. The advantage is to explicitly define which objects are added to the `EvaluationModel` object. Initialization with `EvaluationModel()` is highly recommended.

Object of Class `Criterion` can be added to an object of class `EvaluationModel`.

References

[http://gpaux.github.io/Mediana/](http://gpaux.github.io/Mediana/)

See Also

See Also `Criterion`.

Examples

```r
## Initialize a EvaluationModel and add objects to it
evaluation.model = EvaluationModel() +
    Criterion(id = "Marginal power",
        method = "MarginalPower",
        tests = tests("Placebo vs treatment"),
        labels = c("Placebo vs treatment"),
        par = parameters(alpha = 0.025))
```
**Event**

**Event object**

**Description**

This function creates an object of class Event which can be added to an object of class DataModel.

**Usage**

Event(n.events, rando.ratio = NULL)

**Arguments**

n.events defines a vector of number of events required.
rando.ratio defines a vector of randomization ratios for each Sample object defined in the DataModel.

**Details**

This function can be used if the number of events needs to be fixed in an event-driven clinical trial. Either objects of class Event or SampleSize can be added to an object of class DataModel but not both.

**References**

http://gpaux.github.io/Mediana/

**See Also**

See Also DataModel.

**Examples**

# In this case study, the randomization ratio is 2:1 (Treatment:Placebo).

# Sample size parameters
event.count.total = c(390, 420)
randomization.ratio = c(1,2)

# Outcome parameters
median.time.placebo = 6
rate.placebo = log(2)/median.time.placebo
outcome.placebo = list(rate = rate.placebo)
median.time.treatment = 9
rate.treatment = log(2)/median.time.treatment
outcome.treatment = list(rate = rate.treatment)

# Dropout parameters
dropout.par = parameters(rate = 0.0115)
# Data model
```r
data.model = DataModel() +
   OutcomeDist(outcome.dist = "ExpoDist") +
   Event(n.events = event.count.total, rando.ratio = randomization.ratio) +
   Design(enroll.period = 9,
      study.duration = 21,
      enroll.dist = "UniformDist",
      dropout.dist = "ExpoDist",
      dropout.dist.par = dropout.par) +
   Sample(id = "Placebo",
      outcome.par = parameters(outcome.placebo)) +
   Sample(id = "Treatment",
      outcome.par = parameters(outcome.treatment))
```

**ExtractAnalysisStack**  
*ExtractAnalysisStack function*

**Description**

This function extracts data stack according to the data scenario, sample id and simulation run specified.

**Usage**

```r
ExtractAnalysisStack(analysis.stack, 
   data.scenario = NULL, 
   simulation.run = NULL)
```

**Arguments**

- `analysis.stack` defines a `AnalysisStack` object.
- `data.scenario` defines the data scenario index to extract. By default all data scenarios will be extracted.
- `simulation.run` defines the simulation run index. By default all simulation runs will be extracted.

**Value**

This function extract a particular set of analysis stack according to the data scenario and simulation runs index. The object returned by the function is a list having the same structure as the `analysis.set` argument of a `AnalysisStack` object:

- `analysis.set` a list of size corresponding to the index number of simulation runs specified by the user defined in the `simulation.run` argument. This list contains the results generated for each data scenario (`data.scenario` argument).

**References**

http://gpaux.github.io/Mediana/
See Also

See Also `AnalysisStack`.

Examples

```r
## Not run:
# Generation of an AnalysisStack object

# Outcome parameter set 1
outcome1.placebo = parameters(mean = 0, sd = 70)
outcome1.treatment = parameters(mean = 40, sd = 70)

# Outcome parameter set 2
outcome2.placebo = parameters(mean = 0, sd = 70)
outcome2.treatment = parameters(mean = 50, sd = 70)

# Data model
case.study1.data.model = DataModel() +
  OutcomeDist(outcome.dist = "NormalDist") +
  SampleSize(c(50, 55, 60, 65, 70)) +
  Sample(id = "Placebo",
         outcome.par = parameters(outcome1.placebo, outcome2.placebo)) +
  Sample(id = "Treatment",
         outcome.par = parameters(outcome1.treatment, outcome2.treatment))

# Analysis model
case.study1.analysis.model = AnalysisModel() +
  Test(id = "Placebo vs treatment",
       samples = samples("Placebo", "Treatment"),
       method = "TTest") +
  Statistic(id = "Mean Treatment",
           method = "MeanStat",
           samples = samples("Treatment"))

# Simulation Parameters
case.study1.sim.parameters = SimParameters(n.sims = 1000,
                                           proc.load = 2,
                                           seed = 42938001)

# Generate data
case.study1.analysis.stack = AnalysisStack(data.model = case.study1.data.model,
                                           analysis.model = case.study1.analysis.model,
                                           sim.parameters = case.study1.sim.parameters)

# Print the analysis results generated in the 100th simulation run
# for the 2nd data scenario for both samples
case.study1.analysis.stack$analysis.set[[100]][[2]]

# Extract the same set of data
case.study1.extracted.analysis.stack =
```

ExtractDataStack

ExtractAnalysisStack(analysis.stack = case.study1.analysis.stack,
data.scenario = 2,
simulation.run = 100)

# A careful attention should be paid on the index of the result.
# As only one data.scenario has been requested
# the result for data.scenario = 2 is now in the first position (analysis.set[[1]][[1]]).

# End(Not run)

ExtractDataStack | ExtractDataStack function

Description
This function extracts data stack according to the data scenario, sample id and simulation run specified.

Usage
ExtractDataStack(data.stack,
data.scenario = NULL,
sample.id = NULL,
simulation.run = NULL)

Arguments
data.stack defines a DataStack object.
data.scenario defines the data scenario index to extract. By default all data scenarios will be extracted.
sample.id defines the sample id to extract. By default all sample ids will be extracted.
simulation.run defines the simulation run index. By default all simulation runs will be extracted.

Value
This function extract a particular set of data stack according to the data scenario, sample id and simulation runs index. The object returned by the function is a list having the same structure as the data.set argument of a DataStack object:
data.set a list of size corresponding to the number of simulation runs specified by the user defined in the simulation.run argument. This list contains the data generated for each data scenario (data.scenario argument) and each sample specified by the user (sample.id argument).

References
http://gpaux.github.io/Mediana/
## Not run:

```r
# Generation of a DataStack object
##################################
# Outcome parameter set 1
outcome1.placebo = parameters(mean = 0, sd = 70)
outcome1.treatment = parameters(mean = 40, sd = 70)

# Outcome parameter set 2
outcome2.placebo = parameters(mean = 0, sd = 70)
outcome2.treatment = parameters(mean = 50, sd = 70)

# Data model
case.study1.data.model = DataModel() +
  OutcomeDist(outcome.dist = "NormalDist") +
  SampleSize(c(50, 55, 60, 65, 70)) +
  Sample(id = "Placebo",
         outcome.par = parameters(outcome1.placebo, outcome2.placebo)) +
  Sample(id = "Treatment",
         outcome.par = parameters(outcome1.treatment, outcome2.treatment))

# Simulation Parameters
case.study1.sim.parameters = SimParameters(n.sims = 1000,
                                          proc.load = 2,
                                          seed = 42938001)

# Generate data
case.study1.data.stack = DataStack(data.model = case.study1.data.model,
                                    sim.parameters = case.study1.sim.parameters)

# Print the data set generated in the 100th simulation run
# for the 2nd data scenario for both samples
(case.study1.data.stack$data.set[[100]]$data.scenario[[2]]$sample

# Extract the same set of data
(case.study1.extracted.data.stack = ExtractDataStack(data.stack = case.study1.data.stack,
                                                     data.scenario = 2,
                                                     simulation.run = 100)

# A careful attention should be paid on the index of the result.
# As only one data.scenario has been requested
# the result for data.scenario = 2 is now in the first position (data.scenario[[1]])

## End(Not run)
```
families

Create list of character strings

**Description**

This function is used mostly for user’s convenience. It simply creates a list of character strings.

**Usage**

```r
tests(...)  
samples(...)  
statistics(...)  
parameters(...)  
families(...)  
```

**Arguments**

... defines character strings to be passed into the function.

**References**

http://gpaux.github.io/Mediana/

---

**GenerateData**

Generate data

**Description**

This function generates data according to the specified data model.

**Usage**

```r
GenerateData(data.model, sim.parameters)
```

**Arguments**

- `data.model` defines a DataModel object.
- `sim.parameters` defines a SimParameters object.
Value

This function generates a data stack according to the data model and the simulation parameters objects. The object returned by the function is a DataStack object containing:

- **description**: a description of the object.
- **data.set**: a list of size \( n \text{sims} \) defined in the sim.parameters object.
- **data.data.scenario.grid**: a data frame indicating all data scenario according to the DataModel object.
- **data.structure**: a list containing the data structure according to the DataModel object.
- **sim.parameters**: a list containing the simulation parameters according to SimParameters object.

References

http://gpaux.github.io/Mediana/

See Also

See Also DataModel and SimParameters.

Examples

```r
## Not run:
# Generation of a DataStack object
#################################
# Outcome parameter set 1
outcome1.placebo = parameters(mean = 0, sd = 70)
outcome1.treatment = parameters(mean = 40, sd = 70)

# Outcome parameter set 2
outcome2.placebo = parameters(mean = 0, sd = 70)
outcome2.treatment = parameters(mean = 50, sd = 70)

# Data model
case.study1.data.model = DataModel() +
    OutcomeDist(outcome.dist = "NormalDist") +
    SampleSize(c(50, 55, 60, 65, 70)) +
    Sample(id = "Placebo",
        outcome.par = parameters(outcome1.placebo, outcome2.placebo)) +
    Sample(id = "Treatment",
        outcome.par = parameters(outcome1.treatment, outcome2.treatment))

# Simulation Parameters
case.study1.sim.parameters = SimParameters(n.sims = 1000,
    proc.load = 2,
    seed = 42938001)
```
# Generate data

```r
case.study1.data.stack = GenerateData(data.model = case.study1.data.model, 
  sim.parameters = case.study1.sim.parameters)
```

# Print the data set generated in the 100th simulation run for the 2nd data scenario

```r
case.study1.data.stack$data.set[[100]]$data.scenario[[2]]
```

## End(Not run)

## Not run:

# Use of a DataStack object in the CSE function

### Outcome parameter set 1

```r
outcome1.placebo = parameters(mean = 0, sd = 70)
outcome1.treatment = parameters(mean = 40, sd = 70)
```

### Outcome parameter set 2

```r
outcome2.placebo = parameters(mean = 0, sd = 70)
outcome2.treatment = parameters(mean = 50, sd = 70)
```

# Data model

```r
case.study1.data.model = DataModel() +
  OutcomeDist(outcome.dist = "NormalDist") +
  SampleSize(c(50, 55, 60, 65, 70)) +
  Sample(id = "Placebo",
         outcome.par = parameters(outcome1.placebo, outcome2.placebo)) +
  Sample(id = "Treatment",
         outcome.par = parameters(outcome1.treatment, outcome2.treatment))
```

# Simulation Parameters

```r
case.study1.sim.parameters = SimParameters(n.sims = 1000, 
  proc.load = 2, 
  seed = 42938001)
```

# Generate data

```r
case.study1.data.stack = GenerateData(data.model = case.study1.data.model, 
  sim.parameters = case.study1.sim.parameters)
```

# Analysis model

```r
case.study1.analysis.model = AnalysisModel() +
  Test(id = "Placebo vs treatment",
       samples = samples("Placebo", "Treatment"),
       method = "TTest")
```

# Evaluation model

```r
case.study1.evaluation.model = EvaluationModel() +
  Criterion(id = "Marginal power",
            method = "MarginalPower",
            tests = tests("Placebo vs treatment"),
            labels = c("Placebo vs treatment"),
            ...)```
GenerateReport

Clinical Scenario Evaluation Report

Description

This function generates a Word-based report to present a detailed description of the simulation parameters (data, analysis and evaluation models) and results.

Usage

GenerateReport(presentation.model, cse.results, report.filename, report.template = NULL)

Arguments

- **presentation.model**
  - defines a PresentationModel object.
- **cse.results**
  - defines a CSE object returned by the CSE function.
- **report.filename**
  - defines the output filename of the word-based document generated.
- **report.template**
  - defines a word-based template (optional).

Details

This function requires the package officer. A customized template can be specified in the argument report.template (optional), which consists in a Word document to place in the working directory folder.

References

http://gpaux.github.io/Mediana/

See Also

See Also CSE and PresentationModel.
## Not run:
# Outcome parameter set 1
outcome1.placebo = parameters(mean = 0, sd = 70)
outcome1.treatment = parameters(mean = 40, sd = 70)

# Outcome parameter set 2
outcome2.placebo = parameters(mean = 0, sd = 70)
outcome2.treatment = parameters(mean = 50, sd = 70)

# Data model
case.study1.data.model = DataModel() +
  OutcomeDist(outcome.dist = "NormalDist") +
  SampleSize(c(50, 55, 60, 65, 70)) +
  Sample(id = "Placebo",
    outcome.par = parameters(outcome1.placebo, outcome2.placebo)) +
  Sample(id = "Treatment",
    outcome.par = parameters(outcome1.treatment, outcome2.treatment))

# Analysis model
case.study1.analysis.model = AnalysisModel() +
  Test(id = "Placebo vs treatment",
    samples = samples("Placebo", "Treatment"),
    method = "TTest")

# Evaluation model
case.study1.evaluation.model = EvaluationModel() +
  Criterion(id = "Marginal power",
    method = "MarginalPower",
    tests = tests("Placebo vs treatment"),
    labels = c("Placebo vs treatment"),
    par = parameters(alpha = 0.025))

# Simulation Parameters
case.study1.sim.parameters = SimParameters(n.sims = 1000, proc.load = 2, seed = 42938001)

# Perform clinical scenario evaluation
case.study1.results = CSE(case.study1.data.model,
  case.study1.analysis.model,
  case.study1.evaluation.model,
  case.study1.sim.parameters)

# Reporting
case.study1.presentation.model = PresentationModel() +
  Section(by = "outcome.parameter") +
  Table(by = "sample.size") +
  CustomLabel(param = "sample.size",
    label= paste0("N = ",c(50, 55, 60, 65, 70))) +
  CustomLabel(param = "outcome.parameter",
    label=c("Standard 1", "Standard 2"))
# Report Generation
GenerateReport(presentation.model = case.study1.presentation.model,
cse.results = case.study1.results,
report.filename = "Case study 1 (normally distributed endpoint).docx")

## End(Not run)

## MultAdj

### MultAdj object

**Description**

This function creates an object of class `MultAdj` which can be added to an object of class `AnalysisModel`.

**Usage**

MultAdj(...)

**Arguments**

... defines the arguments passed to create the object of class `MultAdj`.

**Details**

This function can be used to wrap-up several objects of class `MultAdjProc` or `MultAdjStrategy` and add them to an object of class `AnalysisModel`. Its use is optional as objects of class `MultAdjProc` or `MultAdjStrategy` can be added to an object of class `AnalysisModel` incrementally using the `+' operator.

Objects of class `MultAdjProc` or `MultAdjStrategy` can be added to an object of class `AnalysisModel`.

**References**

http://gpaux.github.io/Mediana/

**See Also**

See Also `MultAdjStrategy, MultAdjProc` and `AnalysisModel`.

**Examples**

# Multiplicity adjustments
mult.adj1 = MultAdjProc(proc = NA)
mult.adj2 = MultAdjProc(proc = "BonferroniAdj")
mult.adj3 = MultAdjProc(proc = "HolmAdj", par = parameters(weight = rep(1/3,3)))
mult.adj4 = MultAdjProc(proc = "HochbergAdj", par = parameters(weight = c(1/4,1/4,1/2)))
# Analysis model
```r
analysis.model = AnalysisModel() +
    MultAdj(mult.adj1, mult.adj2, mult.adj3, mult.adj4) +
    Test(id = "Pl vs Dose L",
         samples = samples("Placebo", "Dose L"),
         method = "TTest") +
    Test(id = "Pl vs Dose M",
         samples = samples("Placebo", "Dose M"),
         method = "TTest") +
    Test(id = "Pl vs Dose H",
         samples = samples("Placebo", "Dose H"),
         method = "TTest")
```

# Equivalent to:
```r
analysis.model = AnalysisModel() +
    mult.adj1 +
    mult.adj2 +
    mult.adj3 +
    mult.adj4 +
    Test(id = "Pl vs Dose L",
         samples = samples("Placebo", "Dose L"),
         method = "TTest") +
    Test(id = "Pl vs Dose M",
         samples = samples("Placebo", "Dose M"),
         method = "TTest") +
    Test(id = "Pl vs Dose H",
         samples = samples("Placebo", "Dose H"),
         method = "TTest")
```

---

**MultAdjProc**  
*MultAdjProc object*

**Description**

This function creates an object of class `MultAdjProc` which can be added to objects of class `AnalysisModel`, `MultAdj` or `MultAdjStrategy`.

**Usage**

`MultAdjProc(proc, par = NULL, tests = NULL)`

**Arguments**

- `proc` defines a multiplicity adjustment procedure.
- `par` defines the parameters of the multiplicity adjustment procedure (optional).
- `tests` defines the tests taken into account in the multiplicity adjustment procedure.
Objects of class MultAdjProc are used in objects of class AnalysisModel to specify a Multiplicity Adjustment Procedure that will be applied to the statistical tests to protect the overall Type I error rate. Several objects of class MultAdjProc can be added to an object of class AnalysisModel, using the ‘+’ operator or by grouping them into a MultAdj object.

proc argument defines the multiplicity adjustment procedure. Several procedures are already implemented in the Mediana package (listed below, along with the required or optional parameters to specify in the par argument):

- FixedSeqAdj: Fixed-sequence procedure.
- MultipleSequenceGatekeepingAdj: Family of multiple-sequence gatekeeping procedures. Required parameters: family, proc, gamma.

If no tests are defined, the multiplicity adjustment procedure will be applied to all tests defined in the AnalysisModel.

References

http://gpaux.github.io/Mediana/

See Also

See Also MultAdj, MultAdjStrategy and AnalysisModel.

Examples

# Parameters of the chain procedure (fixed-sequence procedure)
# Vector of hypothesis weights
chain.weight = c(1, 0)
# Matrix of transition parameters
chain.transition = matrix(c(0, 1,
                          0, 0), 2, 2, byrow = TRUE)

# Analysis model
analysis.model = AnalysisModel() +
MultAdjStrategy

MultAdjProc(proc = "ChainAdj",
            par = parameters(weight = chain.weight,
                             transition = chain.transition)) +
Test(id = "PFS test",
     samples = samples("Plac PFS", "Treat PFS"),
     method = "LogrankTest") +
Test(id = "OS test",
     samples = samples("Plac OS", "Treat OS"),
     method = "LogrankTest")

Description

This function creates an object of class MultAdjStrategy which can be added to objects of class AnalysisModel, MultAdj or MultAdjStrategy.

Usage

MultAdjStrategy(...)

Arguments

... defines an object of class MultAdjProc.

Details

This function can be used when several multiplicity adjustment procedures are used within a single Clinical Scenario Evaluation, for example when several case studies are simulated into the same Clinical Scenario Evaluation.

Objects of class MultAdjStrategy are used in objects of class AnalysisModel to define a Multiplicity Adjustment Procedure Strategy that will be applied to the statistical tests to protect the overall Type I error rate. Several objects of class MultAdjStrategy can be added to an object of class AnalysisModel, using the '+' operator or by grouping them into a MultAdj object.

References

http://gpaux.github.io/Mediana/

See Also

See Also MultAdj, MultAdjProc and AnalysisModel.
Examples

# Parallel gatekeeping procedure parameters
family = families(family1 = c(1), family2 = c(2, 3))
component.procedure = families(family1 = "HolmAdj", family2 = "HolmAdj")
gamma = families(family1 = 1, family2 = 1)

# Multiple sequence gatekeeping procedure parameters for Trial A
mult.adj.trialA = MultAdjProc(proc = "ParallelGatekeepingAdj",
    par = parameters(family = family,
        proc = component.procedure,
        gamma = gamma),
    tests = tests("Trial A Pla vs Trt End1",
        "Trial A Pla vs Trt End2",
        "Trial A Pla vs Trt End3")
)

mult.adj.trialB = MultAdjProc(proc = "ParallelGatekeepingAdj",
    par = parameters(family = family,
        proc = component.procedure,
        gamma = gamma),
    tests = tests("Trial B Pla vs Trt End1",
        "Trial B Pla vs Trt End2",
        "Trial B Pla vs Trt End3")
)

mult.adj.pooled = MultAdjProc(proc = "ParallelGatekeepingAdj",
    par = parameters(family = family,
        proc = component.procedure,
        gamma = gamma),
    tests = tests("Pooled Pla vs Trt End1",
        "Pooled Pla vs Trt End2",
        "Pooled Pla vs Trt End3")
)

# Analysis model
analysis.model = AnalysisModel() +
    MultAdjStrategy(mult.adj.trialA, mult.adj.trialB, mult.adj.pooled) +
    # Tests for study A
    Test(id = "Trial A Pla vs Trt End1",
        method = "PropTest",
        samples = samples("Trial A Plac End1", "Trial A Trt End1") +
    Test(id = "Trial A Pla vs Trt End2",
        method = "TTest",
        samples = samples("Trial A Plac End2", "Trial A Trt End2") +
    Test(id = "Trial A Pla vs Trt End3",
        method = "TTest",
        samples = samples("Trial A Plac End3", "Trial A Trt End3") +
    # Tests for study B
    Test(id = "Trial B Pla vs Trt End1",
        method = "PropTest",
        samples = samples("Trial B Plac End1", "Trial B Trt End1") +
    Test(id = "Trial B Pla vs Trt End2",}
OutcomeDist

method = "TTest",
samples = samples("Trial B Plac End2", "Trial B Trt End2")+ Test(id = "Trial B Pla vs Trt End3",
method = "TTest",
samples = samples("Trial B Plac End3", "Trial B Trt End3")+ # Tests for pooled studies
Test(id = "Pooled Pla vs Trt End1",
method = "PropTest",
samples = samples(samples("Trial A Plac End1","Trial B Plac End1"),
samples("Trial A Trt End1","Trial B Trt End1"))) +
Test(id = "Pooled Pla vs Trt End2",
method = "TTest",
samples = samples(samples("Trial A Plac End2","Trial B Plac End2"),
samples("Trial A Trt End2","Trial B Trt End2"))) +
Test(id = "Pooled Pla vs Trt End3",
method = "TTest",
samples = samples(samples("Trial A Plac End3","Trial B Plac End3"),
samples("Trial A Trt End3","Trial B Trt End3")))

OutcomeDist

Description

This function creates an object of class OutcomeDist which can be added to an object of class DataModel.

Usage

OutcomeDist(outcome.dist, outcome.type = NULL)

Arguments

outcome.dist  defines the outcome distribution.
outcome.type  defines the outcome type.

Details

Objects of class OutcomeDist are used in objects of class DataModel to specify the outcome distribution of the generated data. A single object of class OutcomeDist can be added to an object of class DataModel.

Several distribution are already implemented in the Mediana package (listed below, along with the required parameters to specify in the outcome.par argument of the Sample object) to be used in the outcome.dist argument:

- UniformDist: generate data following a univariate distribution. Required parameter: max.
- NormalDist: generate data following a normal distribution. Required parameters: mean and sd.
OutcomeDist

- **BinomDist**: generate data following a binomial distribution. Required parameter: prop.
- **BetaDist**: generate data following a beta distribution. Required parameter: a. and b.
- **ExpoDist**: generate data following an exponential distribution. Required parameter: rate.
- **WeibullDist**: generate data following a weibull distribution. Required parameter: shape and scale.
- **TruncatedExpoDist**: generate data following a truncated exponential distribution. Required parameter: rate and trunc.
- **PoissonDist**: generate data following a Poisson distribution. Required parameter: lambda.
- **NegBinomDist**: generate data following a negative binomial distribution. Required parameters: dispersion and mean.
- **MultinomialDist**: generate data following a multinomial distribution. Required parameter: prob.
- **MVNormalDist**: generate data following a multivariate normal distribution. Required parameters: par and corr. For each generated endpoint, the par parameter must contain the required parameters mean and sd. The corr parameter specifies the correlation matrix for the endpoints.
- **MVBinomDist**: generate data following a multivariate binomial distribution. Required parameters: par and corr. For each generated endpoint, the par parameter must contain the required parameter prop. The corr parameter specifies the correlation matrix for the endpoints.
- **MVEypoDist**: generate data following a multivariate exponential distribution. Required parameters: par and corr. For each generated endpoint, the par parameter must contain the required parameter rate. The corr parameter specifies the correlation matrix for the endpoints.
- **MVEypoPFSOSDist**: generate data following a multivariate exponential distribution to generate PFS and OS endpoints. The PFS value is imputed to the OS value if the latter occurs earlier. Required parameters: par and corr. For each generated endpoint, the par parameter must contain the required parameter rate. The corr parameter specifies the correlation matrix for the endpoints.
- **MVMixedDist**: generate data following a multivariate mixed distribution. Required parameters: type, par and corr. The type parameter can take the following values:
  - NormalDist
  - BinomDist
  - ExpoDist

  For each generated endpoint, the par parameter must contain the required parameters according to the type of distribution. The corr parameter specifies the correlation matrix for the endpoints.

The outcome.type argument defines the outcome’s type. This argument accepts only two values:

- **standard**: for fixed design setting.
- **event**: for event-driven design setting.

The outcome’s type must be defined for each endpoint in case of multivariate distribution, e.g. c("event", "event") in case of multivariate exponential distribution.
# Simple example with a univariate distribution
# Outcome parameter set 1
outcome1.placebo = parameters(mean = 0, sd = 70)
outcome1.treatment = parameters(mean = 40, sd = 70)

# Outcome parameter set 2
outcome2.placebo = parameters(mean = 0, sd = 70)
outcome2.treatment = parameters(mean = 50, sd = 70)

# Data model
data.model = DataModel() +
  OutcomeDist(outcome.dist = "NormalDist") +
  SampleSize(c(50, 55, 60, 65, 70)) +
  Sample(id = "Placebo",
         outcome.par = parameters(outcome1.placebo, outcome2.placebo)) +
  Sample(id = "Treatment",
         outcome.par = parameters(outcome1.treatment, outcome2.treatment))

# Complex example with multivariate distribution following a Binomial and a Normal distribution
# Variable types
var.type = list("BinomDist", "NormalDist")

# Outcome distribution parameters
plac.par = list(list(prop = 0.3), list(mean = -0.10, sd = 0.5))
dosel.par1 = list(list(prop = 0.40), list(mean = -0.20, sd = 0.5))
dosel.par2 = list(list(prop = 0.45), list(mean = -0.25, sd = 0.5))
dosel.par3 = list(list(prop = 0.50), list(mean = -0.30, sd = 0.5))
doseh.par1 = list(list(prop = 0.50), list(mean = -0.30, sd = 0.5))
doseh.par2 = list(list(prop = 0.55), list(mean = -0.35, sd = 0.5))
doseh.par3 = list(list(prop = 0.60), list(mean = -0.40, sd = 0.5))

corr.matrix = matrix(c(1.0, 0.5,
                      0.5, 1.0), 2, 2)

# Correlation between two endpoints
corr.matrix = matrix(c(1.0, 0.5,
                      0.5, 1.0), 2, 2)

# Outcome parameter set 1
outcome1.plac = list(type = var.type, par = plac.par, corr = corr.matrix)
outcome1.dosel = list(type = var.type, par = dosel.par1, corr = corr.matrix)
outcome1.doseh = list(type = var.type, par = doseh.par1, corr = corr.matrix)

# Outcome parameter set 2
outcome2.plac = list(type = var.type, par = plac.par, corr = corr.matrix)
outcome2.dosel = list(type = var.type, par = dosel.par2, corr = corr.matrix)
outcome2.doseh = list(type = var.type, par = doseh.par2, corr = corr.matrix)

# Outcome parameter set 3
outcome3.plac = list(type = var.type, par = plac.par, corr = corr.matrix)
outcome3.doseh = list(type = var.type, par = doseh.par3, corr = corr.matrix)
outcome3.dosel = list(type = var.type, par = dosel.par3, corr = corr.matrix)

# Data model
data.model = DataModel() +
   OutcomeDist(outcome.dist = "MVMixedDist") +
   SampleSize(c(100, 120)) +
   Sample(id = list("Plac ACR20", "Plac HAQ-DI"),
          outcome.par = parameters(outcome1.plac, outcome2.plac, outcome3.plac)) +
   Sample(id = list("DoseL ACR20", "DoseL HAQ-DI"),
          outcome.par = parameters(outcome1.dosel, outcome2.dosel, outcome3.dosel)) +
   Sample(id = list("DoseH ACR20", "DoseH HAQ-DI"),
          outcome.par = parameters(outcome1.doseh, outcome2.doseh, outcome3.doseh))

---

**PresentationModel**  
*PresentationModel object*

**Description**

`PresentationModel()` initializes an object of class `PresentationModel`.

**Usage**

`PresentationModel(...)`

**Arguments**

... defines the arguments passed to create the object of class `PresentationModel`.

**Details**

Presentation models can be used to create a customized structure to report the results. Project information, structure of the sections and subsections, as well as sorting the results tables and labeling of scenarios can be defined.

`PresentationModel()` is used to create an object of class `PresentationModel` incrementally, using the `+` operator to add objects to the existing `PresentationModel` object. The advantage is to explicitly define which objects are added to the `PresentationModel` object. Initialization with `PresentationModel()` is highly recommended.

Objects of class `Project`, `Section`, `Subsection`, `Table` and `CustomLabel` can be added to an object of class `PresentationModel`. 
**Project**

**References**

http://gpaux.github.io/Mediana/

**See Also**

See Also `Project`, `Section`, `Subsection`, `Table` and `CustomLabel`.

**Examples**

```r
presentation.model = PresentationModel() +
  Project(username = "Gautier Paux",
          title = "Clinical trial",
          description = "Simulation report for my clinical trial") +
  Section(by = "outcome.parameter") +
  Table(by = "sample.size") +
  CustomLabel(param = "sample.size",
              label = paste0("N = ", c(50, 55, 60, 65, 70))) +
  CustomLabel(param = "outcome.parameter",
              label = c("Standard 1", "Standard 2"))
```

---

**Project** | **Project object**
--- | ---

**Description**

This function creates an object of class `Project` which can be added to an object of class `PresentationModel`.

**Usage**

```r
Project(username = "[Unknown User]",
        title = "[Unknown title]",
        description = "[No description]")
```

**Arguments**

- `username` defines the username to be printed in the report.
- `title` defines the title of the project to be printed in the report.
- `description` defines the description of the project to be printed in the report.

**Details**

Objects of class `Project` are used in objects of class `PresentationModel` to add more details on the project, such as the author, a title and a description of the project. This information will be added in the report generated using the `GenerateReport` function. A single object of class `Project` can be added to an object of class `PresentationModel`. 
Sample

References

http://gpaux.github.io/Mediana/

See Also

See Also `PresentationModel` and `GenerateReport`.

Examples

```r
# Reporting
presentation.model = PresentationModel() +
  Project(username = "[Mediana's User]",
          title = "Case study 1",
          description = "Clinical trial in patients with pulmonary arterial hypertension") +
  Section(by = "outcome.parameter") +
  Table(by = "sample.size") +
  CustomLabel(param = "sample.size",
              label= paste0("N = ",c(50, 55, 60, 65, 70))) +
  CustomLabel(param = "outcome.parameter",
              label = c("Standard 1", "Standard 2"))
```

Sample object

Description

This function creates an object of class `Sample` which can be added to an object of class `DataModel`.

Usage

```r
Sample(id, outcome.par, sample.size = NULL)
```

Arguments

- `id` defines the ID of the sample.
- `outcome.par` defines the parameters of the outcome distribution of the sample.
- `sample.size` defines the sample size of the sample (optional).

Details

Objects of class `Sample` are used in objects of class `DataModel` to specify a sample. Several objects of class `Sample` can be added to an object of class `DataModel`.

Mandatory arguments are `id` and `outcome.par`. The `sample.size` argument is optional but must be used to define the sample size if unbalance samples have to be defined. The sample size must be either defined in the `Sample` object or in the `SampleSize` object, but not in both.

`outcome.par` defines the sample-specific parameters of the `OutcomeDist` object. Required parameters according to the distribution can be found in `OutcomeDist`. 
SampleSize

References

http://gpaux.github.io/Mediana/

See Also

See Also DataModel and OutcomeDist.

Examples

```r
# Outcome parameter set 1
outcome1.placebo = parameters(mean = 0, sd = 70)
outcome1.treatment = parameters(mean = 40, sd = 70)

# Outcome parameter set 2
outcome2.placebo = parameters(mean = 0, sd = 70)
outcome2.treatment = parameters(mean = 50, sd = 70)

# Data model
case.study1.data.model = DataModel() +
  OutcomeDist(outcome.dist = "NormalDist") +
  SampleSize(c(50, 55, 60, 65, 70)) +
  Sample(id = "Placebo",
    outcome.par = parameters(outcome1.placebo, outcome2.placebo)) +
  Sample(id = "Treatment",
    outcome.par = parameters(outcome1.treatment, outcome2.treatment))
```

SampleSize

SampleSize object

Description

This function creates an object of class SampleSize which can be added to an object of class DataModel.

Usage

SampleSize(sample.size)

Arguments

sample.size a list or vector of sample size(s).

Details

Objects of class SampleSize are used in objects of class DataModel to specify the sample size in case of balanced design (all samples will have the same sample size). A single object of class SampleSize can be added to an object of class DataModel.

Either objects of class Event or SampleSize can be added to an object of class DataModel, but not both.
References

http://gpaux.github.io/Mediana/

See Also

See Also DataModel.

Examples

```r
# Outcome parameter set 1
outcome1.placebo = parameters(mean = 0, sd = 70)
outcome1.treatment = parameters(mean = 40, sd = 70)

# Outcome parameter set 2
outcome2.placebo = parameters(mean = 0, sd = 70)
outcome2.treatment = parameters(mean = 50, sd = 70)

# Data model
case.study1.data.model = DataModel() +
  OutcomeDist(outcome.dist = "NormalDist") +
  SampleSize(c(50, 55, 60, 65, 70)) +
  Sample(id = "Placebo",
         outcome.par = parameters(outcome1.placebo, outcome2.placebo)) +
  Sample(id = "Treatment",
         outcome.par = parameters(outcome1.treatment, outcome2.treatment))

# Equivalent to:
case.study1.data.model = DataModel() +
  OutcomeDist(outcome.dist = "NormalDist") +
  SampleSize(seq(50, 70, 5)) +
  Sample(id = "Placebo",
         outcome.par = parameters(outcome1.placebo, outcome2.placebo)) +
  Sample(id = "Treatment",
         outcome.par = parameters(outcome1.treatment, outcome2.treatment))
```

Section

<table>
<thead>
<tr>
<th>Section</th>
<th>Section object</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Description

This function creates an object of class Section which can be added to an object of class PresentationModel.

Usage

Section(by)

Arguments

by defines the parameter to create the section in the report.
**SimParameters**

**Details**

Objects of class `Section` are used in objects of class `PresentationModel` to define how the results will be presented in the report. If a `Section` object is added to a `PresentationModel` object, the report will have sections according to the parameter defined in the `by` argument. A single object of class `Section` can be added to an object of class `PresentationModel`.

One or several parameters can be defined in the `by` argument:

- "sample.size"
- "event"
- "outcome.parameter"
- "design.parameter"
- "multiplicity.adjustment"

**References**

[http://gpaux.github.io/Mediana/](http://gpaux.github.io/Mediana/)

**See Also**

See Also `PresentationModel`.

**Examples**

```r
# Reporting
presentation.model = PresentationModel() +
  Section(by = "outcome.parameter") +
  Table(by = "sample.size") +
  CustomLabel(param = "sample.size",
              label= paste0("N = ", c(50, 55, 60, 65, 70))) +
  CustomLabel(param = "outcome.parameter",
              label=c("Standard 1", "Standard 2"))

# In this report, one section will be created for each outcome parameter assumption.
```

---

**SimParameters**  
*SimParameters object*

**Description**

This function creates an object of class `SimParameters` to be passed into the CSE function.

**Usage**

`SimParameters(n.sims, seed, proc.load = 1)`
Arguments

n.sims   defines the number of simulations.
seed     defines the seed for the simulations.
proc.load defines the load of the processor (parallel computation).

Details

Objects of class SimParameters are used in the CSE function to define the simulation parameters. The proc.load argument is used to define the number of clusters dedicated to the simulations. Numeric value can be defined as well as character value which automatically detect the number of cores:

• low: 1 processor core.
• med: Number of available processor cores / 2.
• high: Number of available processor cores - 1.
• full: All available processor cores.

References

http://gpaux.github.io/Mediana/

See Also

See Also CSE.

Examples

```
sim.parameters = SimParameters(n.sims = 1000, proc.load = "full", seed = 42938001)
```

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Statistic object</th>
</tr>
</thead>
</table>

Description

This function creates an object of class Statistic which can be added to an object of class AnalysisModel.

Usage

```
Statistic(id, method, samples, par = NULL)
```

Arguments

id    defines the ID of the statistic.
method defines the type of statistics/method for computing the statistic.
samples defines a list of sample(s) (defined in the data model) to be used by the statistic method.
par    defines the parameter(s) of the method for computing the statistic.
Details

Objects of class Statistic are used in objects of class AnalysisModel to define the statistics to produce. Several objects of class Statistic can be added to an object of class AnalysisModel. The method argument defines the statistical method. Several methods are already implemented in the Mediana package (listed below, along with the required parameters to define in the par parameter):

- MedianStat: compute the median of the sample defined in the samples argument.
- MeanStat: compute the mean of the sample defined in the samples argument.
- SdStat: compute the standard deviation of the sample defined in the samples argument.
- MinStat: compute the minimum of the sample defined in the samples argument.
- MaxStat: compute the maximum of the sample defined in the samples argument.
- DiffMeanStat: compute the difference of means between the two samples defined in the samples argument. Two samples must be defined.
- EffectSizeContStat: compute the effect size for a continuous endpoint. Two samples must be defined.
- RatioEffectSizeContStat: compute the ratio of two effect sizes for a continuous endpoint. Four samples must be defined.
- PropStat: compute the proportion of the sample defined in the samples argument.
- DiffPropStat: compute the difference of the proportions between the two samples defined in the samples argument. Two samples must be defined.
- EffectSizePropStat: compute the effect size for a binary endpoint. Two samples must be defined.
- RatioEffectSizePropStat: compute the ratio of two effect sizes for a binary endpoint. Four samples must be defined.
- HazardRatioStat: compute the hazard ratio of the two samples defined in the samples argument. Two samples must be defined. By default the Log-Rank method is used. Optional argument: method taking as value Log-Rank or Cox.
- EffectSizeEventStat: compute the effect size for a survival endpoint (log of the HR). Two samples must be defined. Two samples must be defined. By default the Log-Rank method is used. Optional argument: method taking as value Log-Rank or Cox.
- RatioEffectSizeEventStat: compute the ratio of two effect sizes for a survival endpoint. Four samples must be defined. By default the Log-Rank method is used. Optional argument: method taking as value Log-Rank or Cox.
- EventCountStat: compute the number of events observed in the sample(s) defined in the samples argument.
- PatientCountStat: compute the number of patients observed in the sample(s) defined in the samples argument.

References

http://gpaux.github.io/Mediana/

See Also

See Also AnalysisModel.
Examples

```r
# Analysis model
analysis.model = AnalysisModel() +
    Test(id = "Placebo vs treatment",
         samples = samples("Placebo", "Treatment"),
         method = "TTest") +
    Statistic(id = "Mean Treatment",
              method = "MeanStat",
              samples = samples("Treatment"))
```

Subsection object

Description

This function creates an object of class `Subsection` which can be added to an object of class `PresentationModel`.

Usage

```r
Subsection(by)
```

Arguments

by defines the parameter to create the subsection in the report.

Details

Objects of class `Subsection` are used in objects of class `PresentationModel` to define how the results will be presented in the report. If a `Subsection` object is added to a `PresentationModel` object, the report will have subsections according to the parameter defined in the by argument. A single object of class `Subsection` can be added to an object of class `PresentationModel`.

One or several parameters can be defined in the by argument:

- "sample.size"
- "event"
- "outcome.parameter"
- "design.parameter"
- "multiplicity.adjustment"

A object of class `Subsection` must be added to an object of class `PresentationModel` only if a Section object has been defined.

References

http://gpaux.github.io/Mediana/
Table

See Also

See Also `PresentationModel`.

Examples

```r
# Reporting
presentation.model = PresentationModel() +
  Section(by = "outcome.parameter") +
  Subsection(by = "sample.size") +
  CustomLabel(param = "sample.size",
              label= paste0("N = ",c(50, 55, 60, 65, 70))) +
  CustomLabel(param = "outcome.parameter",
              label=c("Standard 1", "Standard 2"))

# In this report, one section will be created for each outcome parameter assumption
# and within each section, a subsection will be created for each sample size.
```

---

### Table

<table>
<thead>
<tr>
<th>Table object</th>
</tr>
</thead>
</table>

#### Description

This function creates an object of class `Table` which can be added to an object of class `PresentationModel`.

#### Usage

```
Table(by)
```

#### Arguments

- `by`: defines the parameter to sort the table in the report.

#### Details

Objects of class `Table` are used in objects of class `PresentationModel` to define how the results will be sorted in the results tables of the report. If a `Table` object is added to a `PresentationModel` object, the report will generate tables sorted according to the parameter defined in the `by` argument. A single object of class `Table` can be added to an object of class `PresentationModel`.

One or several parameters can be defined in the `by` argument:

- "sample.size"
- "event"
- "outcome.parameter"
- "design.parameter"
- "multiplicity.adjustment"
# Reporting

```r
presentation.model = PresentationModel() +
  Section(by = "outcome.parameter") +
  Table(by = "sample.size") +
  CustomLabel(param = "sample.size",
    label= paste0("N = ",c(50, 55, 60, 65, 70)) +
  CustomLabel(param = "outcome.parameter",
    label=c("Standard 1", "Standard 2"))
```

# In this report, one section will be created for each outcome parameter assumption.
# The tables presented within each section will be sorted by sample size.

## Test

### Test object

**Description**

This function creates an object of class `Test` which can be added to an object of class `AnalysisModel`.

### Usage

```r
Test(id, method, samples, par = NULL)
```

### Arguments

- **id**: defines the ID of the Test object.
- **method**: defines the method of the Test object.
- **samples**: defines a list of samples defined in the data model to be used within the selected Test object method.
- **par**: defines the parameter(s) of the selected Test object method.

### Details

Objects of class `Test` are used in objects of class `AnalysisModel` to define the statistical test to produce. Several objects of class `Test` can be added to an object of class `AnalysisModel`.

### method argument defines the statistical test method. Several methods are already implemented in the Mediana package (listed below, along with the required parameters to define in the par parameter):
Test

- **TTest**: perform a two-sample t-test between the two samples defined in the `samples` argument. Optional parameter: `larger` (Larger value is expected in the second sample (TRUE or FALSE)). Two samples must be defined.

- **TTestNI**: perform a non-inferiority two-sample t-test between the two samples defined in the `samples` argument. Required parameter: `margin`. Optional parameter: `larger` (Larger value is expected in the second sample (TRUE or FALSE)). Two samples must be defined.

- **WilcoxTest**: perform a Wilcoxon-Mann-Whitney test between the two samples defined in the `samples` argument. Optional parameter: `larger` (Larger value is expected in the second sample (TRUE or FALSE)). Two samples must be defined.

- **PropTest**: perform a two-sample test for proportions between the two samples defined in the `samples` argument. Optional parameter: `yates` (Yates’ continuity correction TRUE or FALSE) and `larger` (Larger value is expected in the second sample (TRUE or FALSE)). Two samples must be defined.

- **PropTestNI**: perform a non-inferiority two-sample test for proportions between the two samples defined in the `samples` argument. Required parameter: `margin`. Optional parameter: `yates` (Yates’ continuity correction TRUE or FALSE) and `larger` (Larger value is expected in the second sample (TRUE or FALSE)). Two samples must be defined.

- **FisherTest**: perform a Fisher exact test between the two samples defined in the `samples` argument. Optional parameter: `larger` (Larger value is expected in the second sample (TRUE or FALSE)). Two samples must be defined.

- **GLMPoissonTest**: perform a Poisson regression test between the two samples defined in the `samples` argument. Optional parameter: `larger` (Larger value is expected in the second sample (TRUE or FALSE)). Two samples must be defined.

- **GLMNegBinomTest**: perform a Negative-binomial regression test between the two samples defined in the `samples` argument. Optional parameter: `larger` (Larger value is expected in the second sample (TRUE or FALSE)). Two samples must be defined.

- **LogrankTest**: perform a Log-rank test between the two samples defined in the `samples` argument. Optional parameter: `larger` (Larger value is expected in the second sample (TRUE or FALSE)). Two samples must be defined.

- **OrdinalLogisticRegTest**: perform an Ordinal logistic regression test between the two samples defined in the `samples` argument. Optional parameter: `larger` (Larger value is expected in the second sample (TRUE or FALSE)). Two samples must be defined.

It is to be noted that the statistical tests implemented are one-sided and thus the sample order in the `samples` argument is important. In particular, the Mediana package assumes by default that a numerically larger value of the endpoint is expected in Sample 2 compared to Sample 1. Suppose, for example, that a higher treatment response indicates a beneficial effect (e.g., higher improvement rate). In this case Sample 1 should include control patients whereas Sample 2 should include patients allocated to the experimental treatment arm. The sample order needs to be reversed if a beneficial treatment effect is associated with a lower value of the endpoint (e.g., lower blood pressure), or alternatively (from version 1.0.6), the optional parameters `larger` must be set to `FALSE` to indicate that a larger value is expected on the first Sample.

**References**

http://gpaux.github.io/Mediana/
See Also

See Also `AnalysisModel`.

Examples

```r
# Analysis model
analysis.model = AnalysisModel() +
  Test(id = "Placebo vs treatment",
       samples = samples("Placebo", "Treatment"),
       method = "TTest")
```
Index

* package
  Mediana-package, 2
AdjustCIs, 6, 11
AdjustPvalues, 8, 9
AnalysisModel, 13, 15, 17, 18, 38, 40, 41, 53, 58
AnalysisStack, 14, 30
Criterion, 16, 27
CSE, 17, 36, 52
CustomLabel, 20, 47
DataModel, 15, 18, 21, 22, 26, 28, 34, 45, 49, 50
DataStack, 18, 22, 32
Design, 21, 25
EvaluationModel, 18, 27
Event, 28
ExtractAnalysisStack, 15, 29
ExtractDataStack, 22, 31
families, 33
GenerateData, 33
GenerateReport, 36, 48
Mediana (Mediana-package), 2
Mediana-package, 2
MultAdj, 13, 38, 40, 41
MultAdjProc, 8, 11, 13, 38, 39, 41
MultAdjStrategy, 13, 38, 40, 41
OutcomeDist, 21, 43, 48, 49
parameters (families), 33
PresentationModel, 20, 36, 46, 48, 51, 55, 56
Project, 47, 47
Sample, 21, 48
samples (families), 33
SampleSize, 21, 49
Section, 47, 50
SimParameters, 15, 18, 22, 34, 51
Statistic, 13, 52
statistics (families), 33
Subsection, 47, 54
Table, 47, 55
test, 13, 56
tests (families), 33