#############################################################################

#

# DESCRIPTION

# Functions for determination of the of non-respons R-indicators. A

# short manual of the functions is found in

#

# de Heij, V., Schouten, B., and Shlomo, N. (2010), RISQ manual, Tools in

# SAS and R for the computation of R-indicators and partial

# R-indicators. Work package 8, Deliverable 2.1, 7th Framework

# Programma (FP7) of the European Union.

#

# The manual is available at www.r-indicator.eu.

#

#

# HISTORY

# 2010/05/10 1.0 V. de Heij ---

#

#############################################################################

getRIndicator <-

function(model,

sampleData,

sampleWeights = rep(1, nrow(sampleData)),

sampleStrata = factor(rep('sample', nrow(sampleData))),

withPartials = FALSE,

popTotals = NULL)

{ ## {{{

# Determines the R-indicators and the partial R-indicators for a

# sample.

#

# ARGUMENTS

# model : the respons model which will be used to determine the

# R-indicators; created with the function

# newResponsModel.

#

# sampleData : a data frame containing the sample data;

#

# sampleWeights : (optional) a vector with the inclusion weights of the

# sampling units;

#

# sampleStrata : (optional) a vector with the strata membership of the

# sampling units;

#

# withPartials : (optional) a boolean value, indicating if partial

# R-indicators have to be determined (TRUE) or

# not (FALSE).

#

# VALUE

# getRIndicators returns a list of which the most important components

# are described in the manual.

nSample = nrow(sampleData)

stopifnot(length(sampleWeights) == nSample)

stopifnot(is.numeric(sampleWeights))

stopifnot(length(sampleStrata) == nSample)

stopifnot(is.factor(sampleStrata))

# TODO: Check if the variables used in the formula are part of the data

# frame sampleData.

sampleBased <- isSampleBased(model$formula)

if (sampleBased) {

indicator <- getRSampleBased(

model,

sampleData, sampleWeights, sampleStrata)

if (withPartials)

indicator$partials <- getPartialRs(indicator, sampleData)

} else {

stop('Population-based R-indicators are not yet implemented.')

indicator <- getRPopulationBased(

model,

sampleData, sampleWeights, sampleStrata,

popTotals)

}

return (indicator)

} ## }}}

newResponsModel <-

function(formula,

family = 'binomial')

{ ## {{{

# Creates a list which describes the respons model.

#

# ARGUMENTS

# formula : formula describing the respons model (see details and

# manual);

#

# family : a string either 'binomial' for logistic regression or

# 'gaussian' for lineair regression.

#

# DETAILS

# newResponsModel creates a list which defines the respons model. Part

# of the model is a formula. The left hand side of the formula states

# the respons variabele, the right hand side states the lineair model

# of auxiliary variabeles which will be used to describe the respons.

#

# VALUE

# A list which describes the respons model.

stopifnot(any(family %in% c('binomial', 'gaussian')))

model <- switch(family,

'binomial' = list(

formula = formula,

grad = function(mu) exp(mu) / (1 + exp(mu))^2,

family = binomial(link = 'logit')),

'gaussian' = list(

formula = formula,

grad = function(mu) 1,

family = gaussian(link = 'identity')))

return (model)

} ## }}}

#############################################################################

#

# Private functions for the estimation of the sample-based indicators.

#

#############################################################################

getRSampleBased <-

function(model,

sampleData,

sampleWeights,

sampleStrata)

{ ## {{{

# Determines the sample-based R-indicator.

#

# TODO: Add code to handle errors if t(z) %\*% x is a singular matrix.

#

# TODO: Because the bias can exceed the variance of the propensities,

# propenstities.var - bias can be negative. What then should be the

# bias adjusted R-indicator?

sampleDesign <- getSampleDesign(sampleWeights, sampleStrata)

modelfit <- glm(model$formula, model$family, sampleData)

prop <- predict(modelfit, type = 'response')

propMean <- weighted.mean(prop, sampleWeights)

propVar <- weightedVar(prop, sampleWeights)

# Beacuse estimaters of bias and variance both use the following vectors

# and matrix, they are calculated only once and passed to the functions.

x <- model.matrix(model$formula, sampleData)

z <- model$grad(predict(modelfit, type = 'link')) \* x

sigma <- solve(t(z) %\*% x)

bias <- switch(sampleDesign,

SI = getBiasRSampleBased(

prop, sampleWeights, sampleStrata, z, sigma),

STSI = getBiasRSampleBased(

prop, sampleWeights, sampleStrata, z, sigma),

NA)

variance <- switch(sampleDesign,

SI = getVarianceRSampleBased(

prop, sampleWeights, sampleStrata, z, sigma),

STSI = NA,

NA)

# To simplify formulas the bias correction of the variance will be written

# as a factor, 1 - bias / (variance of propensities).

if (bias > propVar)

biasFactor <- 0

else

biasFactor <- 1 - bias / propVar

indicator <- list(

type = 'R-indicator, sample based',

sampleDesign = sampleDesign,

sampleWeights = sampleWeights,

sampleStrata = sampleStrata,

biasFactor = biasFactor,

prop = prop,

propMean = propMean,

model = model,

modelfit = modelfit,

R = 1 - 2 \* sqrt(propVar \* biasFactor),

RUnadj = 1 - 2 \* sqrt(propVar),

SE = sqrt(variance))

return (indicator)

} ## }}}

getBiasRSampleBased <-

function(prop,

sampleWeights,

sampleStrata,

z,

sigma)

{ ## {{{

# Estimates the bias of the estimator for the variance of the

# propensities.

nPopulation <- sum(sampleWeights)

nStratum <- ave(sampleWeights, sampleStrata, FUN = sum)

prop <- prop \* sqrt(nStratum \* (sampleWeights - 1))

z <- z \* sqrt(sampleWeights)

bias <- numeric()

bias[1] <- sum(sapply(split(prop, sampleStrata), var))

bias[2] <- sum(apply(z, 1, function(zi) return(t(zi) %\*% sigma %\*% zi)))

bias <- (bias[2] - bias[1] / nPopulation) / nPopulation

return (bias)

} ## }}}

getVarianceRSampleBased <-

function(prop,

sampleWeights,

sampleStrata,

z,

sigma)

{ ## {{{

# Estimates the variance of the estimator for the R-indicator.

nSample <- length(sampleWeights)

propVar <- weightedVar(prop, sampleWeights)

factor <- mean(sampleWeights)

A <- (nSample - 1) \* cov(z, prop)

B <- (nSample - 1) \* cov(z)

C <- (nSample - 1) \* var((prop - mean(prop))^2)

variance <- numeric()

variance[1] <- 4 \* t(A) %\*% sigma %\*% A

variance[2] <- 2 \* getTrace(B %\*% sigma %\*% B %\*% sigma)

variance[3] <- (1 - 1 / mean(sampleWeights)) \* C

variance <- sum(variance) / var(prop) / nSample / nSample

return (variance)

} ## }}}

#############################################################################

#

# Private functions for the estimation of the sample-based, partial

# indicators.

#

#############################################################################

getPartialRs <-

function(indicator,

sampleData)

{ ## {{{

# Estimates both unconditional and conditional partial R-indicators.

variables <- getVariables(indicator$model$formula, FALSE)

byVariables <- NULL

byCategories <- list()

for (variable in variables) {

pConditional <-

getPartialRConditional(indicator, sampleData, variable)

pUnconditional <-

getPartialRUnconditional(indicator, sampleData, variable)

byVariable <- data.frame(

variable = variable,

Pu = pUnconditional$Pu,

PuUnadj = pUnconditional$PuUnadj,

Pc = pConditional$Pc,

PcUnadj = pConditional$PcUnadj)

byVariables <- rbind(byVariables, byVariable)

byCategory <- merge(

pUnconditional$byCategory,

pConditional$byCategory)

byCategories <- c(byCategories, list(byCategory))

}

names(byCategories) <- byVariables$variable

partialRs <- list(

byVariables = byVariables,

byCategories = byCategories)

return (partialRs)

} ## }}}

getPartialRUnconditional <-

function(indicator,

sampleData,

variable)

{ ## {{{

# Estimates unconditional partial R-indicators.

modelVariables <- getVariables(indicator$model$formula, FALSE)

stopifnot(variable %in% modelVariables)

stopifnot(variable %in% names(sampleData))

categories <- sampleData[[variable]]

biasFactor <- indicator$biasFactor

nPopulation <- sum(indicator$sampleWeights)

# TODO: propMean is now a componont of the list indicator, so a

# calculation is not needed.

propMean <- with(indicator,

weighted.mean(prop, sampleWeights))

arg <- with(indicator,

data.frame(

n = sampleWeights,

prop = sampleWeights \* prop))

byCategory <- within(

aggregate(arg, list(category = categories), sum), {

prop <- prop / n

propSign <- sign(n \* (prop - propMean))

propVar <- n \* (prop - propMean)^2 / nPopulation

Pu <- propSign \* sqrt(propVar \* biasFactor)

PuUnadj <- propSign \* sqrt(propVar) } )

propVar <- sum(byCategory$propVar)

Pu <- sqrt(propVar \* biasFactor)

PuUnadj <- sqrt(propVar)

partialIndicator <- list(

type = 'Unconditional partial R-indicator, sample based',

variable = variable,

Pu = Pu,

PuUnadj = PuUnadj,

byCategory = byCategory[c("category", "Pu", "PuUnadj")])

return (partialIndicator)

} ## }}}

getPartialRConditional <-

function(indicator,

sampleData,

variable)

{ ## {{{

# Estimates conditional partial R-indicators.

modelVariables <- getVariables(indicator$model$formula, FALSE)

# Some simple checks of the input variables.

stopifnot(variable %in% modelVariables)

stopifnot(variable %in% names(sampleData))

otherVariables <- modelVariables %sub% variable

otherCategories <- as.list(sampleData[otherVariables])

propMeanByOthers <- with(indicator,

ave(sampleWeights \* prop, otherCategories, FUN = sum) /

ave(sampleWeights, otherCategories, FUN = sum))

categories <- sampleData[[variable]]

biasFactor <- indicator$biasFactor

# weights <- with(indicator, sampleWeights / (sum(sampleWeights) - 1))

weights <- with(indicator, sampleWeights / sum(sampleWeights))

arg <- with(indicator,

data.frame(

n = sampleWeights,

propVar = weights \* (prop - propMeanByOthers)^2))

byCategory <- within(

aggregate(arg, list(category = categories), sum), {

Pc <- sqrt(propVar \* biasFactor)

PcUnadj <- sqrt(propVar) } )

propVar <- sum(byCategory$propVar)

Pc <- sqrt(propVar \* biasFactor)

PcUnadj <- sqrt(propVar)

partialIndicator <- list(

type = 'Conditional partial R-indicator, sample based',

variable = variable,

Pc = Pc,

PcUnadj = PcUnadj,

byCategory = byCategory[c("category", "Pc", "PcUnadj")])

return (partialIndicator)

} ## }}}

#############################################################################

#

# Private functions for the estimation of the population-based indicators.

#

#############################################################################

getRPopulationBased <-

function(model,

sampleData,

sampleWeights,

sampleStrata,

popTotals)

{ ## {{{

# Determines the population-based R-indicator.

#

# TODO: Write most of the function.

sampleDesign <- getSampleDesign(sampleWeights, sampleStrata)

modelfit <- NA

prop <- NA

bias <- switch(sampleDesign,

STSI = NA,

SI = getBiasRPopulationBased(),

NA)

variance <- switch(sampleDesign,

STSI = NA,

SI = getVarianceRPopulationBased(),

NA)

R <- NA

RAdjusted <- NA

indicator <- list(

type = 'R-indicator, population based',

sample.design = sampleDesign,

model = model,

modelfit = modelfit,

propensities = propensities,

R = R,

R.adjusted = RAdjusted,

bias = bias,

variance = variance)

return (indicator)

} ## }}}

getBiasRPopulationBased <-

function()

{ ## {{{

# Estimates the bias of the estimator for variance of propensities.

#

# TODO: Write the function.

} ## }}}

getVarianceRPopulationBased <-

function()

{ ## {{{

# Estimates the variance of the estimator for the R-indicator.

#

# TODO: Write the function.

} ## }}}

#############################################################################

#

# Other private functions, ... .

#

#############################################################################

isSampleBased <-

function(formula)

{ ## {{{

# Checks if the left hand-side of the formula is empty. By convention, a

# respons-formule without respons variable -- so just a formula with a

# right hand-side -- implies the calculation of a population-based

# R-indicator.

return (!is.na(getVariables(formula, leftHandSide = TRUE)))

} ## }}}

getSampleDesign <-

function(sampleWeights,

sampleStrata = factor(rep('sample', length(sampleWeights))))

{ ## {{{

# Guesses which type of sample desing is used, using the following rules.

# (1) A single stratum and constant weights implies SI sampling.

# (2) More than one stratum and constant weights per stratum implies STSI

# sampling.

minmax <- sapply(split(sampleWeights, sampleStrata), range)

constantWeights <- all(minmax[1,] == minmax[2,])

nStrata <- length(levels(sampleStrata[, drop = TRUE]))

if (constantWeights && nStrata > 1)

design <- 'STSI'

else if (constantWeights && nStrata == 1)

design <- 'SI'

else

design <- ''

return (design)

} ## }}}

#############################################################################

#

# Other private functions, ... .

#

#############################################################################

getVariables <-

function(formula,

leftHandSide = FALSE)

{ ## {{{

# Returns the names of the variables used either in the left hand side of

# the formula or in the right hand side of the formula.

if (leftHandSide)

formula <- update.formula(formula, . ~ 1)

else

formula <- update.formula(formula, 1 ~ .)

variables <- all.vars(formula)

if (length(variables) == 1 && variables == '.')

variables <- NA

return (variables)

} ## }}}

getTrace <-

function(m)

{ ## {{{

# Returns the trace of the matrix m.

return (trace <- sum(m[col(m) == row(m)]))

} ## }}}

weightedVar <-

function(x,

weights = rep(1, length(x)))

{ ## {{{

# Returns the weighted variance of the vector x.

xMean <- weighted.mean(x, weights)

xVar <- sum(weights \* (x - xMean)^2) / (sum(weights) - 1)

return (xVar)

} ## }}}

'%sub%' <-

function(x,

y)

{ ## {{{

# Returns all elements of the set operation x - y.

# > c(1, 2, 3, 4, 5) %sub% c(2, 4)

# [1] 1 3 5

return (x[! x %in% y])

} ## }}}