Package 'DHARMa'

June 5, 2018

Title Residual Diagnostics for Hierarchical (Multi-Level / Mixed) Regression Models

Version 0.2.0

Date 2018-06-06

Description The 'DHARMa' package uses a simulation-based approach to create readily interpretable scaled (quantile) residuals for fitted (generalized) linear mixed models. Currently supported are (generalized) linear mixed models from 'lme4' (classes 'lmerMod', 'glmerMod') and 'glmmTMB', generalized additive models ('gam' from 'mgcv'), 'glm' (including 'negbin' from 'MASS', but excluding quasi-distributions) and 'lm' model classes. Moreover, externally created simulations, e.g. posterior predictive simulations from Bayesian software such as 'JAGS', 'STAN', or 'BUGS' can be processed as well. The resulting residuals are standardized to values between 0 and 1 and can be interpreted as intuitively as residuals from a linear regression. The package also provides a number of plot and test functions for typical model misspecification problems, such as over/underdispersion, zero-inflation, and residual spatial and temporal autocorrelation.

Depends R (>= 3.0.2)

Imports stats, graphics, utils, grDevices, parallel, doParallel, foreach, gap, qrnn, lmtest, ape, sfsmisc, MASS, lme4, mgcv, glmmTMB (>= 0.2.1)

Suggests knitr, testthat

License GPL (>= 3)

URL http://florianhartig.github.io/DHARMa/

BugReports https://github.com/florianhartig/DHARMa/issues

LazyData true

RoxygenNote 6.0.1

VignetteBuilder knitr

NeedsCompilation no

Author Florian Hartig [aut, cre] (Theoretical Ecology, University of Regensburg, Regensburg, Germany)

Maintainer Florian Hartig <florian.hartig@biologie.uni-regensburg.de>

Repository CRAN

Date/Publication 2018-06-05 21:59:37 UTC

R topics documented:

`createData` *Simulate test data*

Description

This function creates synthetic dataset with various problems such as overdispersion, zero-inflation, etc.

Usage

```
createData(sampleSize = 10, intercept = 0, fixedEffects = 1,
  quadraticFixedEffects = NULL, numGroups = 10, randomEffectVariance = 1,
  overdispersion = 0, family = poisson(), scale = 1, cor = 0,
  roundPoissonVariance = NULL, pZeroInflation = 0, binomialTrials = 1,
  temporalAutocorrelation = 0, spatialAutocorrelation = 0,
  factorResponse = F, replicates = 1)
```

Arguments

sampleSize	sample size of the dataset
intercept	intercept (linear scale)
fixedEffects	vector of fixed effects (linear scale)
quadraticFixedEffects	vector of quadratic fixed effects (linear scale)
numGroups	number of groups for the random effect
randomEffectVariance	variance of the random effect (intercept)
overdispersion	if this is a numeric value, it will be used as the sd of a random normal variate that is added to the linear predictor. Alternatively, a random function can be provided that takes as input the linear predictor.
family	family
scale	scale if the distribution has a scale (e.g. sd for the Gaussian)
cor	correlation between predictors
roundPoissonVariance	if set, this creates a uniform noise on the possion response. The aim of this is to create heteroscedasticity
pZeroInflation	probability to set any data point to zero
binomialTrials	Number of trials for the binomial. Only active if family == binomial
temporalAutocorrelation	strength of temporalAutocorrelation
spatialAutocorrelation	strength of spatial Autocorrelation
factorResponse	should the response be transformed to a factor (inteded to be used for 0/1 data)
replicates	number of datasets to create

Examples

```
testData = createData(sampleSize = 500, intercept = 2, fixedEffects = c(1),
  overdispersion = 0, family = poisson(), quadraticFixedEffects = c(-3),
  randomEffectVariance = 0)

par(mfrow = c(1,2))
plot(testData$Environment1, testData$observedResponse)
```

```
hist(testData$observedResponse)

# with zero-inflation

testData = createData(sampleSize = 500, intercept = 2, fixedEffects = c(1),
  overdispersion = 0, family = poisson(), quadraticFixedEffects = c(-3),
  randomEffectVariance = 0, pZeroInflation = 0.6)

par(mfrow = c(1,2))
plot(testData$Environment1, testData$observedResponse)
hist(testData$observedResponse)

# binomial with multiple trials

testData = createData(sampleSize = 40, intercept = 2, fixedEffects = c(1),
                  overdispersion = 0, family = binomial(), quadraticFixedEffects = c(-3),
                      randomEffectVariance = 0, binomialTrials = 20)

plot(observedResponse1 / observedResponse0 ~ Environment1, data = testData, ylab = "Proportion 1")


# spatial / temporal correlation

testData = createData(sampleSize = 100, family = poisson(), spatialAutocorrelation = 3,
                      temporalAutocorrelation = 3)

plot(log(observedResponse) ~ time, data = testData)
plot(log(observedResponse) ~ x, data = testData)
```

`createDHARMa` *Convert simulated residuals or posterior predictive simulations to a DHARMa object*

Description

Convert simulated residuals or posterior predictive simulations to a DHARMa object

Usage

```
createDHARMa(scaledResiduals = NULL, simulatedResponse = NULL,
  observedResponse = NULL, fittedPredictedResponse = NULL,
  integerResponse = F)
```

Arguments

`scaledResiduals`
: optional scaled residuals from a simulation, e.g. Bayesian p-values. If those are not provided, simulated and true observations have to be provided.

simulatedResponse
: matrix of observations simulated from the fitted model - row index for observations and colum index for simulations

observedResponse
: true observations

fittedPredictedResponse
: fitted predicted response. Optional, but will be neccessary for some plots. If scaled residuals are Bayesian p-values, using the median posterior prediction as fittedPredictedResponse is recommended.

integerResponse
: if T, noise will be added at to the residuals to maintain a uniform expectations for integer responses (such as Poisson or Binomial). Unlike in `simulateResiduals`, the nature of the data is not automatically detected, so this MUST be set by the user appropriately

Details

The use of this function is to convert simulated residuals (e.g. from a point estimate, or Bayesian p-values) to a DHARMa object, to make use of the plotting / test functions in DHARMa

Note

Either scaled residuals or (simulatedResponse AND observed response) have to be provided

Examples

```
## Not run:

  # This example shows how to check the residuals for a
  # Bayesian fit of a process-based vegetation model, using
  # THe BayesianTools package

  library(BayesianTools)

  # Create input data for the model
  PAR <- VSEMcreatePAR(1:1000)
  plotTimeSeries(observed = PAR)

  # load reference parameter definition (upper, lower prior)
  refPars <- VSEMgetDefaults()
  # this adds one additional parameter for the likelihood standard deviation (see below)
  refPars[12,] <- c(2, 0.1, 4)
  rownames(refPars)[12] <- "error-sd"

  # create some simulated test data
  # generally recommended to start with simulated data before moving to real data
 referenceData <- VSEM(refPars$best[1:11], PAR) # model predictions with reference parameters
  referenceData[,1] = 1000 * referenceData[,1]
  # this adds the error - needs to conform to the error definition in the likelihood
  obs <- referenceData + rnorm(length(referenceData), sd = refPars$best[12])
```

```
parSel = c(1:6, 12) # parameters to calibrate

# here is the likelihood
likelihood <- function(par, sum = TRUE){
  # set parameters that are not calibrated on default values
  x = refPars$best
  x[parSel] = par
  predicted <- VSEM(x[1:11], PAR) # replace here VSEM with your model
  predicted[,1] = 1000 * predicted[,1] # this is just rescaling
  diff <- c(predicted[,1:4] - obs[,1:4]) # difference betweeno observed and predicted
 # univariate normal likelihood. Note that there is a parameter involved here that is fit
  llValues <- dnorm(diff, sd = x[12], log = TRUE)
  if (sum == FALSE) return(llValues)
  else return(sum(llValues))
}

# optional, you can also directly provide lower, upper in the createBayesianSetup, see help
prior <- createUniformPrior(lower = refPars$lower[parSel],
                            upper = refPars$upper[parSel], best = refPars$best[parSel])

bayesianSetup <- createBayesianSetup(likelihood, prior, names = rownames(refPars)[parSel])

# settings for the sampler, iterations should be increased for real applicatoin
settings <- list(iterations = 10000, nrChains = 2)

out <- runMCMC(bayesianSetup = bayesianSetup, sampler = "DEzs", settings = settings)

plot(out)
summary(out)
gelmanDiagnostics(out) # should be below 1.05 for all parameters to demonstrate convergence

# Posterior predictive simulations

# Create a function to create posterior predictive simulations
createPredictions <- function(par){
  # set the parameters that are not calibrated on default values
  x = refPars$best
  x[parSel] = par
  predicted <- VSEM(x[1:11], PAR) * 1000
  out = rnorm(length(predicted), mean = predicted, sd = par[7])
  return(out)
}

posteriorSample = getSample(out, numSamples = 1000)
posteriorPredictiveSims = apply(posteriorSample, 1, createPredictions)

dim(posteriorPredictiveSims)
library(DHARMa)
x = createDHARMa(t(posteriorPredictiveSims))
plot(x)

## End(Not run)
```

DHARMa *DHARMa - Residual Diagnostics for HierArchical (Multi-level / Mixed) Regression Models*

Description

The 'DHARMa' package uses a simulation-based approach to create readily interpretable scaled (quantile) residuals for fitted generalized linear mixed models. Currently supported are generalized linear mixed models from 'lme4' (classes 'lmerMod', 'glmerMod') and 'glmmTMB', generalized additive models ('gam' from 'mgcv'), 'glm' (including 'negbin' from 'MASS', but excluding quasi-distributions) and 'lm' model classes. Alternatively, externally created simulations, e.g. posterior predictive simulations from Bayesian software such as 'JAGS', 'STAN', or 'BUGS' can be processed as well. The resulting residuals are standardized to values between 0 and 1 and can be interpreted as intuitively as residuals from a linear regression. The package also provides a number of plot and test functions for typical model misspecification problems, such as over/underdispersion, zero-inflation, and residual spatial and temporal autocorrelation.

Details

See index / vignette for details

See Also

simulateResiduals

Examples

```
vignette("DHARMa", package="DHARMa")
```

---

fitted.gam *This function overwrites the standard fitted function for GAM*

Description

This function overwrites the standard fitted function for GAM

Usage

```
## S3 method for class 'gam'
fitted(object, ...)
```

Arguments

object	fitted model
...	arguments to be passed on to stats::fitted

Note

See explanation at

`getRandomState` *Record and restore a random state*

Description

The aim of this function is to record, manipualate and restor a random state

Usage

```
getRandomState(seed = NULL)
```

Arguments

`seed`	seed argument to set.seed(). NULL = no seed, but random state will be restored. F = random state will not be restored

Details

This function is intended for two (not mutually exclusive tasks)

a) record the current random state

b) change the current random state in a way that the previous state can be restored

Value

a list with various infos about the random state that after function execution, as well as a function to restore the previous state before the function execution

Author(s)

Florian Hartig

Examples

```
# testing the function in standard settings

set.seed(13)
runif(1)
x = getRandomState(123)
runif(1)
x$restoreCurrent()
runif(1)

# values outside set /restore are identical to

set.seed(13)
```

```
runif(2)

# if no seed is set, this will also be restored

rm(.Random.seed)

x = getRandomState(123)
runif(1)
x$restoreCurrent()
exists(".Random.seed")

# with false

rm(.Random.seed)
x = getRandomState(seed = FALSE)
exists(".Random.seed")
runif(1)
x$restoreCurrent()
exists(".Random.seed")
```

`plot.DHARMa` *DHARMa standard residual plots*

Description

This function creates standard plots for the simulated residuals

Usage

```
## S3 method for class 'DHARMa'
plot(x, rank = TRUE, ...)
```

Arguments

`x`	an object with simualted residuals created by `simulateResiduals`
`rank`	if T (default), the values of pred will be rank transformed. This will usually make patterns easier to spot visually, especially if the distribution of the predictor is skewed.
`...`	further options for `plotResiduals`. Consider in particular parameters quantreg, rank and asFactor. xlab, ylab and main cannot be changed when using plotSimulatedResiduals, but can be changed when using plotResiduals.

Details

The function creates two plots. To the left, a qq-uniform plot to detect deviations from overall uniformity of the residuals (calling `plotQQunif`), and to the right, a plot of residuals against predicted values (calling `plotResiduals`). For a correctly specified model, we would expect

a) a straight 1-1 line in the uniform qq-plot -> evidence for an overal uniform (flat) distribution of the residuals

b) uniformity of residuals in the vertical direction in the res against predictor plot

Deviations of this can be interpreted as for a liner regression. See the vignette for detailed examples.

To provide a visual aid in detecting deviations from uniformity in y-direction, the plot of the residuals against the predited values also performs an (optional) quantile regression, which provides 0.25, 0.5 and 0.75 quantile lines across the plots. These lines should be straight, horizontal, and at y-values of 0.25, 0.5 and 0.75. Note, however, that some deviations from this are to be expected by chance, even for a perfect model, especially if the sample size is small. See further comments on this plot, and options, in plotResiduals

The quantile regression can take some time to calculate, especially for larger datasets. For that reason, quantreg = F can be set to produce a smooth spline instead. This is default for n > 2000.

See Also

plotResiduals, plotQQunif

Examples

```
testData = createData(sampleSize = 200, family = poisson(),
                      randomEffectVariance = 0, numGroups = 5)
fittedModel <- glm(observedResponse ~ Environment1,
                   family = "poisson", data = testData)
simulationOutput <- simulateResiduals(fittedModel = fittedModel)

######### main plotting function #############

# for all functions, quantreg = T will be more
# informative, but slower

plot(simulationOutput, quantreg = FALSE)

#############  qq plot  ######################

plotQQunif(simulationOutput = simulationOutput)

#############  residual plots  ###############

# rank transformation, using a simulationOutput
plotResiduals(simulationOutput, rank = TRUE, quantreg = FALSE)

# residual vs predictors, using explicit values for pred, residual
plotResiduals(pred = testData$Environment1,
              residuals = simulationOutput$scaledResiduals, quantreg = FALSE)

# if pred is a factor, or asFactor = T, will produce a boxplot
plotResiduals(pred = testData$group, residuals = simulationOutput$scaledResiduals,
              quantreg = FALSE, asFactor = TRUE)

# All these options can also be provided to the main plotting function
```

```
plot(simulationOutput, quantreg = FALSE, rank = FALSE)

# If you want to plot summaries per group, use
simulationOutput = recalculateResiduals(simulationOutput, group = testData$group)
plot(simulationOutput, asFactor = TRUE) # we see one residual point per RE
```

`plotConventionalResiduals` *Conventional residual plot*

Description

Convenience function to draw conventional residual plots

Usage

```
plotConventionalResiduals(fittedModel)
```

Arguments

`fittedModel`	a fitted model object

`plotQQunif` *Quantile-quantile plot for a uniform distribution*

Description

The function produces a uniform quantile-quantile plot from a DHARMa output

Usage

```
plotQQunif(simulationOutput, testUniformity = T)
```

Arguments

`simulationOutput`	a DHARMa simulation output (class DHARMa)
`testUniformity`	if T, the function `testUniformity` will be called and the result will be added to the plot

Details

the function calls qqunif from the R package gap to create a quantile-quantile plot for a uniform distribution.

See Also

plotSimulatedResiduals, plotResiduals

Examples

```
testData = createData(sampleSize = 200, family = poisson(),
                      randomEffectVariance = 0, numGroups = 5)
fittedModel <- glm(observedResponse ~ Environment1,
                   family = "poisson", data = testData)
simulationOutput <- simulateResiduals(fittedModel = fittedModel)

######### main plotting function #############

# for all functions, quantreg = T will be more
# informative, but slower

plot(simulationOutput, quantreg = FALSE)

#############  qq plot  ######################

plotQQunif(simulationOutput = simulationOutput)

#############  residual plots  ###############

# rank transformation, using a simulationOutput
plotResiduals(simulationOutput, rank = TRUE, quantreg = FALSE)

# residual vs predictors, using explicit values for pred, residual
plotResiduals(pred = testData$Environment1,
              residuals = simulationOutput$scaledResiduals, quantreg = FALSE)

# if pred is a factor, or asFactor = T, will produce a boxplot
plotResiduals(pred = testData$group, residuals = simulationOutput$scaledResiduals,
              quantreg = FALSE, asFactor = TRUE)

# All these options can also be provided to the main plotting function
plot(simulationOutput, quantreg = FALSE, rank = FALSE)

# If you want to plot summaries per group, use
simulationOutput = recalculateResiduals(simulationOutput, group = testData$group)
plot(simulationOutput, asFactor = TRUE) # we see one residual point per RE
```

---

plotResiduals *Generic residual plot with either spline or quantile regression*

Description

The function creates a generic residual plot with either spline or quantile regression

Usage

```
plotResiduals(pred, residuals = NULL, quantreg = NULL, rank = FALSE,
  asFactor = FALSE, ...)
```

Arguments

pred	either the predictor variable against which the residuals should be plotted, or a DHARMa object
residuals	residuals values. Leave empty if pred is a DHARMa object
quantreg	whether to perform a quantile regression on 0.25, 0.5, 0.75 on the residuals. If F, a spline will be created instead. Default NULL chooses T for nObs < 2000, and F otherwise.
rank	if T, the values of pred will be rank transformed. This will usually make patterns easier to spot visually, especially if the distribution of the predictor is skewed. If pred is a factor, this has no effect.
asFactor	should the predictor variable converted into a factor
...	additional arguments to plot

Details

For a correctly specified model, we would expect uniformity in y direction when plotting against any predictor.

To provide a visual aid in detecting deviations from uniformity in y-direction, the plot of the residuals against the predited values also performs an (optional) quantile regression, which provides 0.25, 0.5 and 0.75 quantile lines across the plots. These lines should be straight, horizontal, and at y-values of 0.25, 0.5 and 0.75. Note, however, that some deviations from this are to be expected by chance, even for a perfect model, especially if the sample size is small.

The quantile regression can take some time to calculate, especially for larger datasets. For that reason, quantreg = F can be set to produce a smooth spline instead.

Note

if pred is a factor, a boxplot will be plotted instead of a scatter plot. The distribution for each factor level should be uniformly distributed, so the box should go from 0.25 to 0.75, with the median line at 0.5. Again, chance deviations from this will increases when the sample size is smaller. You can run null simulations to test if the deviations you see exceed what you would expect from random variation. If you want to create box plots for categorical predictors (e.g. because you only have a small number of unique numberic predictor values), you can convert your predictor with as.factor(pred)

See Also

plotSimulatedResiduals, plotQQunif

Examples

```
testData = createData(sampleSize = 200, family = poisson(),
                      randomEffectVariance = 0, numGroups = 5)
fittedModel <- glm(observedResponse ~ Environment1,
                   family = "poisson", data = testData)
simulationOutput <- simulateResiduals(fittedModel = fittedModel)

######### main plotting function #############

# for all functions, quantreg = T will be more
# informative, but slower

plot(simulationOutput, quantreg = FALSE)

#############  qq plot  ######################

plotQQunif(simulationOutput = simulationOutput)

#############  residual plots  ###############

# rank transformation, using a simulationOutput
plotResiduals(simulationOutput, rank = TRUE, quantreg = FALSE)

# residual vs predictors, using explicit values for pred, residual
plotResiduals(pred = testData$Environment1,
              residuals = simulationOutput$scaledResiduals, quantreg = FALSE)

# if pred is a factor, or asFactor = T, will produce a boxplot
plotResiduals(pred = testData$group, residuals = simulationOutput$scaledResiduals,
              quantreg = FALSE, asFactor = TRUE)

# All these options can also be provided to the main plotting function
plot(simulationOutput, quantreg = FALSE, rank = FALSE)

# If you want to plot summaries per group, use
simulationOutput = recalculateResiduals(simulationOutput, group = testData$group)
plot(simulationOutput, asFactor = TRUE) # we see one residual point per RE
```

`plotSimulatedResiduals` *DHARMa standard residual plots*

Description

DEPRECATED, use plot() instead

Usage

```
plotSimulatedResiduals(simulationOutput, ...)
```

Arguments

simulationOutput	an object with simualted residuals created by simulateResiduals
...	further options for plotResiduals. Consider in particular parameters quantreg, rank and asFactor. xlab, ylab and main cannot be changed when using plotSimulatedResiduals, but can be changed when using plotResiduals.

Note

THis function is deprecated. Use plot.DHARMa

See Also

plotResiduals, plotQQunif

---

print.DHARMa *Print simulated residuals*

Description

Print simulated residuals

Usage

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

Arguments

x	an object with simulated residuals created by simulateResiduals
...	optional arguments for compatibility with the generic function, no function implemented

`recalculateResiduals` *Recalculate residuals with grouping*

Description

The purpose of this function is to recalculate scaled residuals per group, based on the simulations done by `simulateResiduals`

Usage

```
recalculateResiduals(simulationOutput, group = NULL, aggregateBy = sum)
```

Arguments

`simulationOutput`	an object with simualted residuals created by `simulateResiduals`
`group`	group of each data point
`aggregateBy`	function for the aggregation. Default is sum. This should only be changed if you know what you are doing. Note in particular that the expected residual distribution might not be flat any more if you choose general functions, such as sd etc.

Value

an object of class DHARMa, similar to what is returned by `simulateResiduals`, but with additional outputs for the new grouped calculations. Note that the relevant outputs are 2x in the object, the first is the grouped calculations (which is returned by $name access), and later another time, under identical name, the original output. Moreover, there is a function 'aggregateByGroup', which can be used to aggregate predictor variables in the same way as the variables calculated here

Examples

```
library(lme4)

testData = createData(sampleSize = 200, overdispersion = 0.5, family = poisson())
fittedModel <- glmer(observedResponse ~ Environment1 + (1|group),
                     family = "poisson", data = testData,
                     control=glmerControl(optCtrl=list(maxfun=20000) ))

simulationOutput <- simulateResiduals(fittedModel = fittedModel)

# plot residuals, quantreg = T is better but costs more time
plot(simulationOutput, quantreg = FALSE)

# the calculated residuals can be accessed via
residuals(simulationOutput)
simulationOutput$scaledResiduals
```

```
# calculating summaries per group
simulationOutput = recalculateResiduals(simulationOutput, group = testData$group)
plot(simulationOutput, quantreg = FALSE)

# create simulations with refitting, n=5 is very low, set higher when using this
simulationOutput <- simulateResiduals(fittedModel = fittedModel,
                                      n = 10, refit = TRUE)
plot(simulationOutput, quantreg = FALSE)

# grouping per random effect group works as above
simulationOutput = recalculateResiduals(simulationOutput, group = testData$group)
plot(simulationOutput, quantreg = FALSE)
```

`refit.glmmTMB` *Refit a Model with a Different Response*

Description

Refit a Model with a Different Response

Usage

```
## S3 method for class 'glmmTMB'
refit(object, newresp, ...)
```

Arguments

`object`	a fitted model
`newresp`	a new response
`...`	further arguments, no effect implemented for this S3 class

Examples

```
testData = createData(sampleSize = 200, family = poisson())

# examples of refit with different model classes
library(lme4)
library(mgcv)
library(glmmTMB)

fittedModel <- lm(observedResponse ~ Environment1 , data = testData)
newResponse = simulate(fittedModel)
refit(fittedModel, newResponse[,1])

fittedModel <- glm(observedResponse ~ Environment1 , data = testData, family = "poisson")
newResponse = simulate(fittedModel)
refit(fittedModel, newResponse[,1])
```

```
fittedModel <- mgcv::gam(observedResponse ~ s(Environment1) , data = testData, family = "poisson")
newResponse = simulate(fittedModel)
refit(fittedModel, newResponse[,1])

fittedModel <- lme4::lmer(observedResponse ~ Environment1 + (1|group) , data = testData)
newResponse = simulate(fittedModel)
refit(fittedModel, newResponse[,1])

fittedModel <- lme4::glmer(observedResponse ~ Environment1 + (1|group) , data = testData,
                           family = "poisson")
newResponse = simulate(fittedModel)
refit(fittedModel, newResponse[,1])

fittedModel <- glmmTMB::glmmTMB(observedResponse ~ Environment1 + (1|group) , data = testData)
newResponse = simulate(fittedModel)
refit(fittedModel, newResponse[,1])
```

`refit.lm` *Refit a Model with a Different Response*

Description

Refit a Model with a Different Response

Usage

```
## S3 method for class 'lm'
refit(object, newresp, ...)
```

Arguments

`object`	a fitted model
`newresp`	a new response
`...`	further arguments, no effect implemented for this S3 class

Examples

```
testData = createData(sampleSize = 200, family = poisson())

# examples of refit with different model classes
library(lme4)
library(mgcv)
library(glmmTMB)

fittedModel <- lm(observedResponse ~ Environment1 , data = testData)
newResponse = simulate(fittedModel)
refit(fittedModel, newResponse[,1])

fittedModel <- glm(observedResponse ~ Environment1 , data = testData, family = "poisson")
```

```
newResponse = simulate(fittedModel)
refit(fittedModel, newResponse[,1])

fittedModel <- mgcv::gam(observedResponse ~ s(Environment1) , data = testData, family = "poisson")
newResponse = simulate(fittedModel)
refit(fittedModel, newResponse[,1])

fittedModel <- lme4::lmer(observedResponse ~ Environment1 + (1|group) , data = testData)
newResponse = simulate(fittedModel)
refit(fittedModel, newResponse[,1])

fittedModel <- lme4::glmer(observedResponse ~ Environment1 + (1|group) , data = testData,
                           family = "poisson")
newResponse = simulate(fittedModel)
refit(fittedModel, newResponse[,1])

fittedModel <- glmmTMB::glmmTMB(observedResponse ~ Environment1 + (1|group) , data = testData)
newResponse = simulate(fittedModel)
refit(fittedModel, newResponse[,1])
```

`residuals.DHARMa` *Return residuals of a DHARMa simulation*

Description

Return residuals of a DHARMa simulation

Usage

```
## S3 method for class 'DHARMa'
residuals(object, ...)
```

Arguments

`object`	an object with simulated residuals created by `simulateResiduals`
`...`	optional arguments for compatibility with the generic function, no function implemented

Details

the function accesses the slot $scaledResiduals in a fitted DHARMa object

Examples

```
library(lme4)

testData = createData(sampleSize = 200, overdispersion = 0.5, family = poisson())
fittedModel <- glmer(observedResponse ~ Environment1 + (1|group),
                     family = "poisson", data = testData,
```

```
                    control=glmerControl(optCtrl=list(maxfun=20000) ))

simulationOutput <- simulateResiduals(fittedModel = fittedModel)

# plot residuals, quantreg = T is better but costs more time
plot(simulationOutput, quantreg = FALSE)

# the calculated residuals can be accessed via
residuals(simulationOutput)
simulationOutput$scaledResiduals

# calculating summaries per group
simulationOutput = recalculateResiduals(simulationOutput, group = testData$group)
plot(simulationOutput, quantreg = FALSE)

# create simulations with refitting, n=5 is very low, set higher when using this
simulationOutput <- simulateResiduals(fittedModel = fittedModel,
                                      n = 10, refit = TRUE)
plot(simulationOutput, quantreg = FALSE)

# grouping per random effect group works as above
simulationOutput = recalculateResiduals(simulationOutput, group = testData$group)
plot(simulationOutput, quantreg = FALSE)
```

`runBenchmarks` *Benchmark calculations*

Description

This function runs statistical benchmarks, including Power / Type I error simulations for an arbitrary test with a control parameter

Usage

```
runBenchmarks(calculateStatistics, controlValues = NULL, nRep = 10,
  alpha = 0.05, parallel = F, ...)
```

Arguments

`calculateStatistics`	the statistics to be benchmarked. Should return one value, or a vector of values. If controlValues are given, must accept a paramteter control
`controlValues`	a vector with a control parameter (e.g. to vary the strength of a problem the test should be specific to)
`nRep`	number of replicates per level of the controlValues
`alpha`	significance level

parallel	whether to use parallel computations. Possible values are F, T (sets the cores automatically to number of available cores -1), or an integer number for the number of cores that should be used for the cluster
...	additional parameters to calculateStatistics

Note

The benchmark function in DHARMa are intended for development purposes, and for users that want to test / confirm the properties of functions in DHARMa. If you are running an applied data analysis, they are probably of little use.

---

simulateResiduals *Create simulated residuals*

Description

The function creates scaled residuals by simulating from the fitted model

Usage

```
simulateResiduals(fittedModel, n = 250, refit = F, integerResponse = NULL,
  plot = F, seed = 123, ...)
```

Arguments

fittedModel	fitted model object. Supported are generalized linear mixed models from 'lme4' (classes 'lmerMod', 'glmerMod'), generalized additive models ('gam' from 'mgcv', excluding extended families from 'mgcv'), 'glm' (including 'negbin' from 'MASS', but excluding quasi-distributions) and 'lm' model classes.
n	integer number > 1, number of simulations to run. If possible, set to at least 250, better 1000. Smaller number > 50 can be chose if runtime is prohbitie, but discretization artefacts can occur at some point.
refit	if F, new data will be simulated and scaled residuals will be created by comparing observed data with new data. If T, the model will be refit on the simulated data (parametric bootstrap), and scaled residuals will be created by comparing observed with refitted residuals.
integerResponse	if T, noise will be added at to the residuals to maintain a uniform expectations for integer responses (such as Poisson or Binomial). Usually, the model will automatically detect the appropriate setting, so there is no need to adjust this setting.
plot	if T, plotSimulatedResiduals will be directly run after the simulations have terminated

seed	the random seed. The default setting, recommended for any type of data analysis, is to reset the random number generator each time the function is run, meaning that you will always get the same result when running the same code. NULL = no new seed is set, but previous random state will be restored after simulation. F = no seed is set, and random state will not be restored. The latter two options are only recommended for simulation experiments. See vignette for details.
...	parameters to pass to the simulate function of the model object. An important use of this is to specify whether simulations should be conditional on the current random effect estimates. See details.

Details

There are a number of important considerations when simulating from a more complex (hierarchical) model:

Re-simulating random effects / hierarchical structure: the first is that in a hierarchical model, several layers of stochasticity are aligned on top of each other. Specifically, in a GLMM, we have a lower level stochastic process (random effect), whose result enters into a higher level (e.g. Poisson distribution). For other hierarchical models such as state-space models, similar considerations apply. When simulating, we have to decide if we want to re-simulate all stochastic levels, or only a subset of those. For example, in a GLMM, it is common to only simulate the last stochastic level (e.g. Poisson) conditional on the fitted random effects.

For controlling how many levels should be re-simulated, the simulateResidual function allows to pass on parameters to the simulate function of the fitted model object. Please refer to the help of the different simulate functions (e.g. ?simulate.merMod) for details. For merMod (lme4) model objects, the relevant parameters are parameters are use.u, and re.form

If the model is correctly specified, the simulated residuals should be flat regardles how many hierarchical levels we re-simulate. The most thorough procedure would therefore be to test all possible options. If testing only one option, I would recommend to re-simulate all levels, because this essentially tests the model structure as a whole. This is the default setting in the DHARMa package. A potential drawback is that re-simulating the lower-level random effects creates more variability, which may reduce power for detecing problems in the upper-level stochatic processes.

Integer responses: a second complication is the treatment of inter responses. Imaging we have observed a 0, and we predict 30% zeros - what is the quantile that we should display for the residual? To deal with this problem and maintain a unifor response, the option integerResponse adds a uniform noise from -0.5 to 0.5 on the simulated and observed response. Note that this works because the expected distribution of this is flat - you can see this via hist(ecdf(runif(10000))(runif(10000)))

Refitting or not: a third issue is how residuals are calculated. simulateResiduals has two options that are controlled by the refit parameter:

1. if refit = F (default), new data is simulated from the fitted model, and residuals are calculated by comparing the observed data to the new data

2. if refit = T, a parametric bootstrap is performed, meaning that the model is refit on the new data, and residuals are created by comparing observed residuals against refitted residuals

The second option is much slower, and only important for running tests that rely on comparing observed to simulated residuals, e.g. the `testOverdispersion` function

Residuals per group: In many situations, it can be useful to look at residuals per group, e.g. to see how much the model over / underpredicts per plot, year or subject. To do this, use `recalculateResiduals`, together with a grouping variable (see also help)

Value

An S3 class of type "DHARMa", essentially a list with various elements. Implemented S3 functions include plot, print and `residuals.DHARMa`. Residuals returns the calculated scaled residuals, which can also be accessed via $scaledResiduals. The returned object additionally contains an element 'scaledResidualsNormal', which contains the scaled residuals transformed to a normal distribution (for stability reasons not recommended)

Note

See `testResiduals` for an overview of residual tests, `plot.DHARMa` for an overview of available plots.

See Also

`testResiduals`, `plot.DHARMa`, `print.DHARMa`, `residuals.DHARMa`, `recalculateResiduals`

Examples

```
library(lme4)

testData = createData(sampleSize = 200, overdispersion = 0.5, family = poisson())
fittedModel <- glmer(observedResponse ~ Environment1 + (1|group),
                     family = "poisson", data = testData,
                     control=glmerControl(optCtrl=list(maxfun=20000) ))

simulationOutput <- simulateResiduals(fittedModel = fittedModel)

# plot residuals, quantreg = T is better but costs more time
plot(simulationOutput, quantreg = FALSE)

# the calculated residuals can be accessed via
residuals(simulationOutput)
simulationOutput$scaledResiduals

# calculating summaries per group
simulationOutput = recalculateResiduals(simulationOutput, group = testData$group)
plot(simulationOutput, quantreg = FALSE)

# create simulations with refitting, n=5 is very low, set higher when using this
simulationOutput <- simulateResiduals(fittedModel = fittedModel,
                                      n = 10, refit = TRUE)
plot(simulationOutput, quantreg = FALSE)

# grouping per random effect group works as above
simulationOutput = recalculateResiduals(simulationOutput, group = testData$group)
plot(simulationOutput, quantreg = FALSE)
```

`testDispersion` *DHARMa dispersion tests*

Description

This function performs a simulation-based test for over/underdispersion

Usage

```
testDispersion(simulationOutput, alternative = c("two.sided", "greater",
  "less"), plot = T, ...)
```

Arguments

`simulationOutput`	a DHARMa object with simulated residuals created with `simulateResiduals`
`alternative`	a character string specifying whether the test should test if observations are "greater", "less" or "two.sided" compared to the simulated null hypothesis. Greate corresponds to overdispersion.
`plot`	whether to plot output
`...`	arguments to pass on to `testGeneric`

Details

The function implements two tests, depending on whether it is applied on a simulation with refit = F, or refit = T.

If refit = F (not recommended), the function tests if the IQR of the scaled residuals deviate from the null hypothesis of a uniform distribution. Simulations show that this option is not properly calibrated and much less powerful than the parametric alternative `testOverdispersionParametric` and even the simple `testUniformity`, and therefore it's use is not recommended. A warning will be returned if the function is called.

If refit = T, the function compares the approximate deviance (via squared pearson residuals) with the same quantity from the models refitted with simulated data. It is much slower than the parametric alternative `testOverdispersionParametric`, but simulations show that it is slightly more powerful than the latter, and more powerful than any other non-parametric test in DHARMa, and it doesn't make any parametric assumptions. However, given the computational cost, I would suggest that most users will be satisfied with the parametric overdispersion test.

Author(s)

Florian Hartig

See Also

`testResiduals`, `testUniformity`, `testZeroInflation`, `testGeneric`, `testTemporalAutocorrelation`, `testSpatialAutocorrelation`

Examples

```
# creating test data

testData = createData(sampleSize = 200, overdispersion = 0.5, randomEffectVariance = 0)
fittedModel <- glm(observedResponse ~ Environment1 , family = "poisson", data = testData)
simulationOutput <- simulateResiduals(fittedModel = fittedModel)

plot(simulationOutput, quantreg = FALSE)

###### Distribution tests #####

testUniformity(simulationOutput)

###### Dispersion tests #######

testDispersion(simulationOutput, alternative = "less") # underdispersion

###### Both together###########

testResiduals(simulationOutput)

###### Special tests ##########

# testing zero inflation
testZeroInflation(simulationOutput)

# testing generic summaries
countOnes <- function(x) sum(x == 1)  # testing for number of 1s
testGeneric(simulationOutput, summary = countOnes) # 1-inflation
testGeneric(simulationOutput, summary = countOnes, alternative = "less") # 1-deficit

means <- function(x) mean(x) # testing if mean prediction fits
testGeneric(simulationOutput, summary = means)

spread <- function(x) sd(x) # testing if mean sd fits
testGeneric(simulationOutput, summary = spread)

###### Refited ##############

# if model is refitted, a different test will be called

simulationOutput <- simulateResiduals(fittedModel = fittedModel, refit = TRUE, seed = 12)
testDispersion(simulationOutput)

###### Test per group ##############

simulationOutput = recalculateResiduals(simulationOutput, group = testData$group)
testDispersion(simulationOutput)
```

`testGeneric` *Generic simulation test of a summary statistic*

Description

This function tests if a user-defined summary differs when applied to simulated / observed data.

Usage

```
testGeneric(simulationOutput, summary, alternative = c("two.sided", "greater",
  "less"), plot = T, methodName = "DHARMa generic simulation test")
```

Arguments

`simulationOutput`	a DHARMa object with simulated residuals created with `simulateResiduals`
`summary`	a function that can be applied to simulated / observed data. See examples below
`alternative`	a character string specifying whether the test should test if observations are "greater", "less" or "two.sided" compared to the simulated null hypothesis
`plot`	whether to plot the simulated summary
`methodName`	name of the test (will be used in plot)

Details

This function tests if a user-defined summary differs when applied to simulated / observed data. the function can easily be remodeled to apply summaries on the residuals, by simply defining f = function(x) summary (x - predictions), as done in `testDispersion`

Note

The function that you supply is applied on the data as it is represented in your fitted model, which may not always correspond to how you think. This is important in particular when you use k/n binomial data, and want to test for 1-inflation. As an example, if have k/20 observations, and you provide your data via cbind (y, y-20), you have to test for 20-inflation (because this is how the data is represented in the model). However, if you provide data via y/20, and weights = 20, you should test for 1-inflation. In doubt, check how the data is internally represented in model.frame(model), or via simulate(model)

Author(s)

Florian Hartig

See Also

`testResiduals`, `testUniformity`, `testDispersion`, `testZeroInflation`, `testTemporalAutocorrelation`, `testSpatialAutocorrelation`

Examples

```
# creating test data

testData = createData(sampleSize = 200, overdispersion = 0.5, randomEffectVariance = 0)
fittedModel <- glm(observedResponse ~ Environment1 , family = "poisson", data = testData)
simulationOutput <- simulateResiduals(fittedModel = fittedModel)

plot(simulationOutput, quantreg = FALSE)

###### Distribution tests #####

testUniformity(simulationOutput)

###### Dispersion tests #######

testDispersion(simulationOutput, alternative = "less") # underdispersion

###### Both together###########

testResiduals(simulationOutput)

###### Special tests ##########

# testing zero inflation
testZeroInflation(simulationOutput)

# testing generic summaries
countOnes <- function(x) sum(x == 1)  # testing for number of 1s
testGeneric(simulationOutput, summary = countOnes) # 1-inflation
testGeneric(simulationOutput, summary = countOnes, alternative = "less") # 1-deficit

means <- function(x) mean(x) # testing if mean prediction fits
testGeneric(simulationOutput, summary = means)

spread <- function(x) sd(x) # testing if mean sd fits
testGeneric(simulationOutput, summary = spread)

###### Refited ##############

# if model is refitted, a different test will be called

simulationOutput <- simulateResiduals(fittedModel = fittedModel, refit = TRUE, seed = 12)
testDispersion(simulationOutput)

###### Test per group ##############

simulationOutput = recalculateResiduals(simulationOutput, group = testData$group)
testDispersion(simulationOutput)
```

testOverdispersion *Simulated overdisperstion tests*

Description

Simulated overdisperstion tests

Usage

```
testOverdispersion(simulationOutput, ...)
```

Arguments

simulationOutput	a DHARMa object with simulated residuals created with simulateResiduals
...	additional arguments to testDispersion

Details

Deprecated, switch your code to using the testDispersion function

testOverdispersionParametric *Parametric overdisperstion tests*

Description

Parametric overdisperstion tests

Usage

```
testOverdispersionParametric(...)
```

Arguments

...	arguments will be ignored, the parametric tests is no longer recommend

Details

Deprecated, switch your code to using the testDispersion function. The function will do nothing, arguments will be ignored, the parametric tests is no longer recommend

testPDistribution *Plot distribution of p-values*

Description

Plot distribution of p-values

Usage

```
testPDistribution(x, plot = T,
  main = "p distribution \n expected is flat at 1", ...)
```

Arguments

x	vector of p values
plot	should the values be plottet
main	title for the plot
...	additional arguments to hist

Author(s)

Florian Hartig

testResiduals *DHARMa general residual test*

Description

Calls both uniformity and dispersion test

Usage

```
testResiduals(simulationOutput)
```

Arguments

simulationOutput	a DHARMa object with simulated residuals created with simulateResiduals

Details

This function is a wrapper for the various test functions implemented in DHARMa. Currently, this function calls the testUniformity and the testDispersion functions. All other tests (see below) have to be called by hand.

Author(s)

Florian Hartig

See Also

testUniformity, testDispersion, testZeroInflation, testGeneric, testTemporalAutocorrelation, testSpatialAutocorrelation

---

testSimulatedResiduals

Residual tests

Description

Residual tests

Usage

```
testSimulatedResiduals(simulationOutput)
```

Arguments

simulationOutput	a DHARMa object with simulated residuals created with simulateResiduals

Details

Deprecated, switch your code to using the testResiduals function

Author(s)

Florian Hartig

---

testSpatialAutocorrelation

Test for spatial autocorrelation

Description

This function performs a standard test for spatial autocorrelation on the simulated residuals

Usage

```
testSpatialAutocorrelation(simulationOutput, x = NULL, y = NULL,
  distMat = NULL, alternative = c("two.sided", "greater", "less"),
  plot = T)
```

Arguments

simulationOutput	a DHARMa object with simulated residuals created with simulateResiduals
x	the x coordinate, in the same order as the data points. If not provided, random values will be created
y	the x coordinate, in the same order as the data points. If not provided, random values will be created
distMat	optional distance matrix. If not provided, a distance matrix will be calculated based on x and y. See details for explanation
alternative	a character string specifying whether the test should test if observations are "greater", "less" or "two.sided" compared to the simulated null hypothesis
plot	whether to plot output

Details

The function performs Moran.I test from the package ape, based on the provided distance matrix of the data points.

There are several ways to specify this distance. If a distance matrix (distMat) is provided, calculations will be based on this distance matrix, and x,y coordinates will only used for the plotting (if provided) If distMat is not provided, the function will calculate the euclidian distances between x,y coordinates, and test Moran.I based on these distances.

The sense of being able to run the test with x/y = NULL (random values) is to test the rate of false positives under the current residual structure (random x/y corresponds to H0: no spatial autocorrelation), e.g. to check if the test has noninal error rates for particular residual structures.

Author(s)

Florian Hartig

See Also

testResiduals, testUniformity, testDispersion, testZeroInflation, testGeneric, testTemporalAutocorrelatio

Examples

```
testData = createData(sampleSize = 40, family = gaussian())
fittedModel <- lm(observedResponse ~ Environment1, data = testData)
res = simulateResiduals(fittedModel)

# Standard use
testSpatialAutocorrelation(res, x =  testData$x, y = testData$y)

# If x and y is not provided, random values will be created
testSpatialAutocorrelation(res)

# Alternatively, one can provide a distance matrix
dM = as.matrix(dist(cbind(testData$x, testData$y)))
testSpatialAutocorrelation(res, distMat = dM)
```

testTemporalAutocorrelation *Test for temporal autocorrelation*

Description

This function performs a standard test for temporal autocorrelation on the simulated residuals

Usage

```
testTemporalAutocorrelation(simulationOutput, time = NULL,
  alternative = c("two.sided", "greater", "less"), plot = T)
```

Arguments

simulationOutput	an object with simulated residuals created by simulateResiduals
time	the time, in the same order as the data points. If set to "random", random values will be created
alternative	a character string specifying whether the test should test if observations are "greater", "less" or "two.sided" compared to the simulated null hypothesis
plot	whether to plot output

Details

The function performs a Durbin-Watson test on the uniformly scaled residuals, and plots the residuals against time. The DB test was originally be designed for normal residuals. In simulations, I didn't see a problem with this setting though. The alternative is to transform the uniform residuals to normal residuals and perform the DB test on those.

Note

The sense of being able to run the test with time = NULL (random values) is to test the rate of false positives under the current residual structure (random time corresponds to H0: no spatial autocorrelation), e.g. to check if the test has noninal error rates for particular residual structures (note that Durbin-Watson originally assumes normal residuals, error rates seem correct for uniform residuals, but may not be correct if there are still other residual problems).

Author(s)

Florian Hartig

See Also

testResiduals, testUniformity, testDispersion, testZeroInflation, testGeneric, testSpatialAutocorrelation

Examples

```
testData = createData(sampleSize = 40, family = gaussian())
fittedModel <- lm(observedResponse ~ Environment1, data = testData)
res = simulateResiduals(fittedModel)

# Standard use
testTemporalAutocorrelation(res, time =  testData$time)

# If no time is provided, random values will be created
testTemporalAutocorrelation(res)
```

---

testUniformity *Test for overall uniformity*

Description

This function tests the overall uniformity of the simulated residuals in a DHARMa object

Usage

```
testUniformity(simulationOutput, alternative = c("two.sided", "less",
  "greater"), plot = T)
```

Arguments

simulationOutput	a DHARMa object with simulated residuals created with simulateResiduals
alternative	a character string specifying whether the test should test if observations are "greater", "less" or "two.sided" compared to the simulated null hypothesis
plot	if T, plots calls plotQQunif as well

Details

The function applies a KS test for uniformity on the simulated residuals

Author(s)

Florian Hartig

See Also

testResiduals, testDispersion, testZeroInflation, testGeneric, testTemporalAutocorrelation, testSpatialAutocorrelation

`testZeroInflation` *Tests for zero-inflation*

Description

This function compares the observed number of zeros with the zeros expected from simulations.

Usage

```
testZeroInflation(simulationOutput, ...)
```

Arguments

`simulationOutput`	a DHARMa object with simulated residuals created with `simulateResiduals`
`...`	further arguments to `testGeneric`

Details

shows the expected distribution of zeros against the observed

Author(s)

Florian Hartig

See Also

`testResiduals`, `testUniformity`, `testDispersion`, `testGeneric`, `testTemporalAutocorrelation`, `testSpatialAutocorrelation`

Examples

```
# creating test data

testData = createData(sampleSize = 200, overdispersion = 0.5, randomEffectVariance = 0)
fittedModel <- glm(observedResponse ~ Environment1 , family = "poisson", data = testData)
simulationOutput <- simulateResiduals(fittedModel = fittedModel)

plot(simulationOutput, quantreg = FALSE)

###### Distribution tests #####

testUniformity(simulationOutput)

###### Dispersion tests #######

testDispersion(simulationOutput, alternative = "less") # underdispersion

###### Both together###########
```

```
testResiduals(simulationOutput)

###### Special tests ##########

# testing zero inflation
testZeroInflation(simulationOutput)

# testing generic summaries
countOnes <- function(x) sum(x == 1)  # testing for number of 1s
testGeneric(simulationOutput, summary = countOnes) # 1-inflation
testGeneric(simulationOutput, summary = countOnes, alternative = "less") # 1-deficit

means <- function(x) mean(x) # testing if mean prediction fits
testGeneric(simulationOutput, summary = means)

spread <- function(x) sd(x) # testing if mean sd fits
testGeneric(simulationOutput, summary = spread)

###### Refited ##############

# if model is refitted, a different test will be called

simulationOutput <- simulateResiduals(fittedModel = fittedModel, refit = TRUE, seed = 12)
testDispersion(simulationOutput)

###### Test per group ##############

simulationOutput = recalculateResiduals(simulationOutput, group = testData$group)
testDispersion(simulationOutput)
```

Index