Package ‘DHARMa’

November 18, 2019

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

Version 0.2.5

Date 2019-11-18

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 linear and generalized linear (mixed) models from 'lme4' (classes 'lmerMod', 'glmerMod'), 'glmmTMB' and 'spaMM', 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, qrmn, lmtest, ape, sfsmisc, MASS, lme4, mgcv, glmmTMB (>= 0.2.1), spaMM (>= 2.6.0)

Suggests knitr, testthat

License GPL (>= 3)

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

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

LazyData true

RoxygenNote 6.1.1

VignetteBuilder knitr

Encoding UTF-8

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**  2019-11-18 18:10:02 UTC

---

**R topics documented:**

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>createData</td>
<td>3</td>
</tr>
<tr>
<td>createDHARMa</td>
<td>4</td>
</tr>
<tr>
<td>DHARMa</td>
<td>7</td>
</tr>
<tr>
<td>fitted.gam</td>
<td>8</td>
</tr>
<tr>
<td>getRandomState</td>
<td>8</td>
</tr>
<tr>
<td>getResponse</td>
<td>10</td>
</tr>
<tr>
<td>getSimulations</td>
<td>10</td>
</tr>
<tr>
<td>hist.DHARMa</td>
<td>11</td>
</tr>
<tr>
<td>plot.DHARMa</td>
<td>12</td>
</tr>
<tr>
<td>plotConventionalResiduals</td>
<td>14</td>
</tr>
<tr>
<td>plotQQunif</td>
<td>14</td>
</tr>
<tr>
<td>plotResiduals</td>
<td>16</td>
</tr>
<tr>
<td>plotSimulatedResiduals</td>
<td>18</td>
</tr>
<tr>
<td>print.DHARMa</td>
<td>19</td>
</tr>
<tr>
<td>refit.glmmTMB</td>
<td>20</td>
</tr>
<tr>
<td>refit.lm</td>
<td>21</td>
</tr>
<tr>
<td>residuals.DHARMa</td>
<td>22</td>
</tr>
<tr>
<td>runBenchmarks</td>
<td>23</td>
</tr>
<tr>
<td>simulateResiduals</td>
<td>24</td>
</tr>
<tr>
<td>testDispersion</td>
<td>27</td>
</tr>
<tr>
<td>testGeneric</td>
<td>29</td>
</tr>
<tr>
<td>testModel</td>
<td>31</td>
</tr>
<tr>
<td>testOutliers</td>
<td>31</td>
</tr>
<tr>
<td>testOverdispersion</td>
<td>32</td>
</tr>
<tr>
<td>testOverdispersionParametric</td>
<td>33</td>
</tr>
<tr>
<td>testPDistribution</td>
<td>33</td>
</tr>
<tr>
<td>testResiduals</td>
<td>34</td>
</tr>
<tr>
<td>testSimulatedResiduals</td>
<td>34</td>
</tr>
<tr>
<td>testSpatialAutocorrelation</td>
<td>35</td>
</tr>
<tr>
<td>testTemporalAutocorrelation</td>
<td>38</td>
</tr>
<tr>
<td>testUniformity</td>
<td>40</td>
</tr>
<tr>
<td>testZeroInflation</td>
<td>41</td>
</tr>
<tr>
<td>transformQuantiles</td>
<td>42</td>
</tr>
</tbody>
</table>

**Index**  44
createData

Simulate test data

description

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

usage

createData(sampleSize = 100, 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 poission 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 (intended 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(simulatedResponse, observedResponse,
  fittedPredictedResponse = NULL, integerResponse = F)
createDHARMa

Arguments

- **simulatedResponse**: matrix of observations simulated from the fitted model - row index for observations and column index for simulations.
- **observedResponse**: true observations.
- **fittedPredictedResponse**: fitted predicted response. Optional, but will be necessary 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

```r
## READING IN HAND-CODED SIMULATIONS

testData = createData(sampleSize = 50, randomEffectVariance = 0)
fittedModel <- glm(observedResponse ~ Environment1, data = testData, family = "poisson")

# in DHARMa, using the simulate.glm function of glm
sims = simulateResiduals(fittedModel)
plot(sims, quantreg = FALSE)

# Doing the same with a handcode simulate function. 
# of course this code will only work with a 1-par glm model
simulateMyfit <- function(n=10, fittedModel){
  int = coef(fittedModel)[1]
  slo = coef(fittedModel)[2]
  pred = exp(int + slo * testData$Environment1)
  predSim = replicate(n, rpois(length(pred), pred))
  return(predSim)
}
sims = simulateMyfit(250, fittedModel)
dharmaRes <- createDHARMa(simulatedResponse = sims,
                            observedResponse = testData$observedResponse,
                            ...)
```
fittedPredictedResponse = predict(fittedModel, type = "response"),
   integer = TRUE)

plot(dharmaRes, quantreg = FALSE)

## A BAYESIAN EXAMPLE

## 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$best[1:11], PAR) # model predictions with reference parameters
referenceData[,1] = 1000 * referenceData[,1] # this is just rescaling
obs <- referenceData + rnorm(length(referenceData), sd = refPars$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 between 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 applications
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 linear and generalized linear (mixed) models from 'lme4' (classes 'lmerMod', 'glmerMod'), 'glmmTMB' and 'spaMM', 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.

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

```r
## 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, manipulate and restore a random state

Usage

```r
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
getRandomState

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")
getResponse

*Get model response*

**Description**

Extract the response of a fitted model

**Usage**

```r
getResponse(object, ...)
```

**Arguments**

- `object`: a fitted model
- `...`: additional parameters

**Details**

The purpose of this function is to safely extract the response (dependent variable) of the fitted model classes.

**Author(s)**

Florian Hartig

getSimulations

*Get model simulations*

**Description**

Wrapper to simulate from a fitted model

**Usage**

```r
getSimulations(object, ...)
```

**Arguments**

- `object`: a fitted model
- `...`: additional parameters to be passed on, usually to the simulate function of the respective model class

**Details**

The purpose of this wrapper is to standardize the simulations from a model in a standardized way.
hist.DHARMa

Author(s)
Florian Hartig

hist.DHARMa
Histogram of DHARMa residuals

Description
The function produces a histogram from a DHARMa output

Usage
## S3 method for class 'DHARMa'
hist(x, ...)

Arguments
x          a DHARMa simulation output (class DHARMa)
...        arguments to be passed on to hist. Breaks and col are fixed.

See Also
plotSimulatedResiduals, plotResiduals

Examples
testData = createData(sampleSize = 200, family = poisson(),
 randomEffectVariance = 1, 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)

########### Distribution ########################
plotQQunif(simulationOutput = simulationOutput)
hist(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

---

**plot.DHARMa**

**DHARMa standard residual plots**

**Description**

This function creates standard plots for the simulated residuals

**Usage**

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

**Arguments**

- `x` an object with simulated 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`). Outliers are highlighted in red (for more on outliers, see `testOutliers`). 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 linear 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 predicted 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, its interpretation 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 = 1, 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)

############# Distribution ######################
plotQQunif(simulationOutput = simulationOutput)
hist(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, testOutliers = 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
testOutliers if T, the function testOutliers will be called and the result will be added to the plot
... arguments to be passed on to qqunif
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 = 1, 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)

############# Distribution ######################
plotQQunif(simulationOutput = simulationOutput)
hist(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 to highlight patterns in the residuals. Outliers are highlighted in red.

**Usage**

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

**Arguments**

- `pred`:
  - either the predictor variable against which the residuals should be plotted, or a DHARMa object, in which case res ~ pred is plotted.

- `residuals`:
  - residuals values. Can be either numerical values, or a DHARMa object, from which residual values can be extracted.

- `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 be treated as a factor. Default is to choose this for <10 unique predictions, as long as enough predictions are available to draw a boxplot.

- `...`:
  - additional arguments to plot / boxplot.

**Details**

The function plots residuals against a predictor (e.g. fitted value, or any other predictor).

Outliers are highlighted in red (for information on definition and interpretation of outliers, see `testOutliers`).

To provide a visual aid in detecting deviations from uniformity in y-direction, the plot function calculates an (optional) quantile regression, which compares the empirical 0.25, 0.5 and 0.75 quantiles in y direction (red solid lines) with the theoretical 0.25, 0.5 and 0.75 quantiles (dashed black line). Assymptotically (i.e. for lots of data / residuals), if the model is correct, theoretical and the empirical quantiles should be identical (i.e. dashed and solid lines should match).

In practice, however, there will be only a finite and often small number of residuals. If the model is correct, these residuals are drawn from the theoretical (uniform) distribution, but because of the limited sample size, the empirical quantiles of these residuals will never perfectly match the theoretical quantiles. It’s the same as in a normal linear regression - even if the model is entirely
correct, the qq-plot (or any other residual plot) for a few data points will never perfectly match the theoretical quantiles.

Thus, for a limited amount of data, the question one has to ask is if the deviation of the empirical (red) from the expected (dashed) distribution is strong enough so that one can reject the null hypothesis that the residuals are drawn from a uniform distribution. To answer this question, DHARMa has various tests implemented (see later). Unfortunately, there is not yet a dedicated test for trends in the red quantile lines, so at the moment it’s up to the user to make the call of a deviation in the residual pattern is is still acceptable, i.e. could appear do to random variation.

Note

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.

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 numeric predictor values), you can convert your predictor with as.factor(pred)

See Also

plotSimulatedResiduals, plotQQunif

Examples

testData = createData(sampleSize = 200, family = poisson(),
  randomEffectVariance = 1, 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)

############# Distribution  #####################
plotQQunif(simulationOutput = simulationOutput)
hist(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

DHAR Ma standard residual plots

Description

DEPRECATED, use plot() instead

Usage

plotSimulatedResiduals(simulationOutput, ...)

Arguments

  simulationOutput
  an object with simulated 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.DHAR Ma

See Also

plotResiduals, plotQQunif
print.DHARMa

## Description
Print simulated residuals

## Usage
```r
# 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
```r
recalculateResiduals(simulationOutput, group = NULL, aggregateBy = sum)
```

## Arguments
- **simulationOutput**: an object with simulated 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

```r
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**

```r
## 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
runBenchmarks

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, ...)
simulateResiduals

Arguments

- **calculateStatistics**: the statistics to be benchmarked. Should return one value, or a vector of values. If controlValues are given, must accept a parameter 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

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

Arguments

- **fittedModel**: a fitted model of a class supported by DHARMa
- **n**: number of simulations. Default is 100. A more save value would be 250 or even 1000. The smaller the number, the higher the stochastic error on the residuals. Also, for very small n, discretization artefacts can influence the tests.
- **refit**: if FALSE, new data will be simulated and scaled residuals will be created by comparing observed data with new data. If TRUE, 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 TRUE, 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.
simulateResiduals

- **plot**: if TRUE, `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. FALSE = 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 also 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 `use.u` and `re.form`.

If the model is correctly specified, the simulated residuals should be flat regardless 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 detecting problems in the upper-level stochastic processes.

**Integer responses**: a second complication is the treatment of integer responses. Imagining 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 uniform response, the option `integerResponse` adds a uniform noise from -0.5 to 0.5 on the simulated and observed response, which creates a uniform distribution - you can see this via `hist(ecdf(runif(10000))(runif(10000)))`.

DHARMa will try to automatically if the fitted model has an integer or discrete distribution via the `family` argument. However, in some cases the family does not allow to uniquely identify the distribution type. For example, a tweedie distribution can be inter or continuous. Therefore, DHARMa will additionally check the simulation results for repeated values, and will change the distribution type if repeated values are found (a message is displayed in this case).

**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 = FALSE (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 = TRUE, 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**

```r
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
```
testDispersion <- simulateResiduals(fittedModel = fittedModel, n = 10, refit = TRUE)
plot(testDispersion, quantreg = FALSE)

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

---

**testDispersion**  

*DHARMA dispersion tests*

**Description**

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

**Usage**

```r
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. Greater 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, the function tests the sd of the data against the sd of the simulated data.

If refit = T, the function compares the approximate deviance (via squared pearson residuals) with the same quantity from the models refitted with simulated data. Applying this is much slower than the previous alternative, but simulations show that it is slightly more powerful as well.

However, given the computational cost, I would suggest that most users will be satisfied with the standard dispersion test.

**Author(s)**

Florian Hartig
See Also

testResiduals, testUniformity, testOutliers, 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)
**Description**

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

**Usage**

```r
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, testOutliers, testDispersion, testZeroInflation, testTemporalAutocorrelation, testSpatialAutocorrelation

Examples

# creating test data

testData = createData(sampleSize = 200, overdispersion = 0.5, randomEffectVariance = 0)
fittedModel <- glm(observableResponse ~ 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)

# Refitted

# 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)

testModel  Test DHARMa compatibility

Description
This helper function tests the compatibility of a model with DHARMa by trying to run various functions that are needed.

Usage
testModel(fittedModel)

Arguments
fittedModel  the fitted model

Author(s)
Florian Hartig

testOutliers  Test for outliers

Description
This function tests if the number of observations that are strictly greater / smaller than all simulations are larger than expected.

Usage
testOutliers(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
plot  if T, the function will create an additional plot
DHARMa residuals are created by simulating from the fitted model, and comparing the simulated values to the observed data. It can occur that all simulated values are higher or smaller than the observed data, in which case they get the residual value of 0 and 1, respectively. I refer to these values as simulation outliers, or simply outliers.

Because no data was simulated in the range of the observed value, we actually don’t know “how much” these values deviate from the model expectation, so the term “outlier” should be used with a grain of salt - it’s not a judgement about the probability of a deviation from an expectation, but denotes that we are outside the simulated range. The number of outliers would usually decrease if the number of DHARMa simulations is increased.

The probability of an outlier depends on the number of simulations (in fact, it is 1/(nSim +1) for each side), so whether the existence of outliers is a reason for concern depends also on the number of simulations. The expected number of outliers is therefore binomially distributed, and we can calculate a p-value from that.

Author(s)
Florian Hartig

See Also
testResiduals, testUniformity, testDispersion, testZeroInflation, testGeneric, testTemporalAutocorrelation, testSpatialAutocorrelation

testOverdispersion Simulated overdispersion tests

Description
Simulated overdispersion 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 overdispersion tests

Description

Parametric overdispersion 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
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, testOutliers, testDispersion, testZeroInflation, testGeneric, testTemporalAutocorrelation, testSpatialAutocorrelation

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**

```r
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 y 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.

If no x/y values are provided, random values will be created. 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 nominal error rates for particular residual structures.

Testing for spatial autocorrelation requires unique x,y values - if you have several observations per location, either use the recalculateResiduals function to aggregate residuals per location, or extract the residuals from the fitted object, and plot/test each of them independently for spatially repeated subgroups (a typical scenario would repeated spatial observations, in which case one could plot/test each time step separately for temporal autocorrelation). Note that the latter must be done by hand, outside testSpatialAutocorrelation.

Note

Important to note for all autocorrelation tests (spatial/temporal): the autocorrelation tests are valid to check for residual autocorrelation in models that don’t assume such a correlation (in this case, you can use conditional or unconditional simulations), or if there is remaining residual autocorrelation after accounting for it in a spatial/temporal model (in that case, you have to use conditional simulations), but if checking unconditional simulations from a model with an autocorrelation structure on data that corresponds to this model, they will be significant, even if the model fully accounts for this structure.

This behavior is not really a bug, but rather originates from the definition of the quantile residuals: quantile residuals are calculated independently per data point, i.e. without consideration of any correlation structure between data points that may exist in the simulations. As a result, the simulated distributions from an unconditional simulation will typically not reflect the correlation structure that is present in each single simulation, and the same is true for the subsequently calculated quantile residuals.

The bottomline here is that spatial/temporal/other autoregressive models should either be tested based on conditional simulations, or (ideally) custom tests should be used that are not based on quantile residuals, but rather compare the correlation structure in the simulated data with the correlation structure in the observed data.

Author(s)

Florian Hartig

See Also

testResiduals, testUniformity, testOutliers, testDispersion, testZeroInflation, testGeneric, testTemporalAutocorrelation

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)

# if there are multiple observations with the same x values,
# create first ar group with unique values for each location
# then aggregate the residuals per location, and calculate
# spatial autocorrelation on the new group
res2 = recalculateResiduals(res, group = testData$group)
testSpatialAutocorrelation(res)

# careful with clustered data and conditional / unconditional simulations
# this originates from https://github.com/florianhartig/DHARMa/issues/81

# Assume our data is divided into clusters, and we use a RE to take out cluster effects
clusters = 100
subsamples = 10
size = clusters * subsamples
testData = createData(sampleSize = size, family = gaussian(), numGroups = clusters )
testData$x = rnorm(clusters)[testData$group] + rnorm(size, sd = 0.01)
testData$y = rnorm(clusters)[testData$group] + rnorm(size, sd = 0.01)

library(lme4)
fittedModel <- lmer(observedResponse ~ Environment1 + (1|group), data = testData)

# DHARMa default is to re-simulted REs - this means spatial pattern remains
# because residuals are still clustered
res = simulateResiduals(fittedModel)
testSpatialAutocorrelation(res, x = testData$x, y = testData$y)

# However, it should disappear if you just calculate an aggregate residuals per cluster
# Because at least how the data are simulated, cluster are spatially independent
res2 = recalculateResiduals(res, group = testData$group)
testSpatialAutocorrelation(res2, 
x = aggregate(testData$x, list(testData$group), mean)$x,
y = aggregate(testData$y, list(testData$group), mean)$x)

# For lme4, possible to simulated residuals conditional on fitted REs (re.form)
# This takes out most of the RSA - a remainder is probably due the shrinkage
# of the REs
res = simulateResiduals(fittedModel, re.form = NULL)
testSpatialAutocorrelation(res, x = testData$x, y = testData$y)
testTemporalAutocorrelation

Test for temporal autocorrelation

Description

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

Usage

```r
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 not provided, 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 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.

If no time values are provided, random values will be created. 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 nominal 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).

Testing for temporal autocorrelation requires unique time values - if you have several observations per time value, either use the `recalculateResiduals` function to aggregate residuals per time step, or extract the residuals from the fitted object, and plot / test each of them independently for temporally repeated subgroups (typical choices would be location / subject etc.). Note that the latter must be done by hand, outside `testSpatialAutocorrelation`.

Note

Important to note for all autocorrelation tests (spatial / temporal): the autocorrelation tests are valid to check for residual autocorrelation in models that don’t assume such a correlation (in this case,
You can use conditional or unconditional simulations, or if there is remaining residual autocorrelation after accounting for it in a spatial/temporal model (in that case, you have to use conditional simulations), but if checking unconditional simulations from a model with an autocorrelation structure on data that corresponds to this model, they will be significant, even if the model fully accounts for this structure.

This behavior is not really a bug, but rather originates from the definition of the quantile residuals: quantile residuals are calculated independently per data point, i.e. without consideration of any correlation structure between data points that may exist in the simulations. As a result, the simulated distributions from a unconditional simulation will typically not reflect the correlation structure that is present in each single simulation, and the same is true for the subsequently calculated quantile residuals.

The bottomline here is that spatial / temporal / other autoregressive models should either be tested based on conditional simulations, or (ideally) custom tests should be used that are not based on quantile residuals, but rather compare the correlation structure in the simulated data with the correlation structure in the observed data.

**Author(s)**

Florian Hartig

**See Also**

testResiduals, testUniformity, testOutliers, 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)

# If you have several observations per time step
timeSeries1 = createData(sampleSize = 40, family = gaussian())
timeSeries1$location = 1
timeSeries2 = createData(sampleSize = 40, family = gaussian())
timeSeries2$location = 2
testData = rbind(timeSeries1, timeSeries2)
fittedModel <- lm(observedResponse ~ Environment1, data = testData)
res = simulateResiduals(fittedModel)

# for this, you cannot do testTemporalAutocorrelation(res, time = testData$time)
# because here we would have observations with the same time, i.e.
# zero difference in time. We have two options a) aggregate observations
# b) calculate / test per subset. Testing per subset might also be useful
# if you have several locations, regardless of whether the times are
# identical, because you would expect the autocorrelation structure to be
# independent per location

# testing grouped residuals
res = recalculateResiduals(res, group = testData$time)
testTemporalAutocorrelation(res, time = unique(testData$time))

# plotting and testing per subgroup

# extract subgroup
testData$Residuals = res$scaledResiduals
temp = testData[testData$location == 1,]

# plots and tests
plot(Residuals ~ time, data = temp)
lmtest::dwtest(temp$Residuals ~ 1, order.by = temp$time)

---

testUniformity

*Test for overall uniformity*

**Description**

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

**Usage**

```r
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. See `ks.test` for details
- `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
### testZeroInflation

**Tests for zero-inflation**

**Description**

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

**Usage**

```r
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`, `testOutliers`, `testDispersion`, `testZeroInflation`, `testGeneric`, `testTemporalAutocorrelation`, `testSpatialAutocorrelation`

**Examples**

```r
# 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)
```
transformQuantiles

Transform quantiles to pdf

Description

The purpose of this function is to transform the DHARMa quantile residuals (which have a uniform distribution) to a particular pdf.

Usage

transformQuantiles(res, quantileFunction = qnorm, outlierValue = 7)
transformQuantiles

Arguments

- **res**: DHARMa residuals
- **quantileFunction**: the quantile function of the desired distribution
- **outlierValue**: the value that should be assigned to residuals that are 0/1 and thus typically mapped to -Inf / Inf for continuous distributions with infinite support

Details

Some of the papers on simulated quantile residuals transforming the residuals (which are natively uniform) back to a normal distribution. I presume this is because of the larger familiarity of most users with normal residuals. Personally, I never considered this desirable, for the reasons explained in https://github.com/florianhartig/DHARMa/issues/39, but with this function, I wanted to give users the option to plot normal residuals if they so wish.

Examples

```r
set.seed(1)
testData = createData(sampleSize = 200, family = poisson())
fittedModel <- glm(observeResponse ~ Environment1,
                   family = "poisson", data = testData)
res <- simulateResiduals(fittedModel = fittedModel)
plot(res$fittedPredictedResponse, transformQuantiles(res))
```
Index

createData, 3
createDHARMa, 4

DHARMa, 7
DHARMa-package (DHARMa), 7

fitted.gam, 8

getResponse, 10
getSimulations, 10

hist.DHARMa, 11

ks.test, 40

plot.DHARMa, 12, 18, 26
plotConventionalResiduals, 14
plotQQunif, 12, 13, 14, 17, 18, 40
plotResiduals, 11–13, 15, 16, 18
plotSimulatedResiduals, 11, 15, 17, 18, 25
print.DHARMa, 19, 26

qqunif, 14

recalculateResiduals, 19, 26
refit.glmmTMB, 20
refit.lm, 21
residuals.DHARMa, 22, 26
runBenchmarks, 23

simulateResiduals, 5, 8, 12, 18, 19, 22, 24,
27, 29, 31, 32, 34, 35, 38, 40, 41

testDispersion, 27, 29, 30, 32–34, 36, 39, 41
testGeneric, 27, 28, 29, 32, 34, 36, 39, 41
testModel, 31
testOutliers, 12, 14, 16, 28, 30, 31, 34, 36,
39, 41
testOverdispersion, 26, 32
testOverdispersionParametric, 33