Package ‘regressinator’

August 11, 2023

Type Package

Title Simulate and Diagnose (Generalized) Linear Models

Version 0.1.2

Date 2023-08-10

Description Simulate samples from populations with known covariate distributions, generate response variables according to common linear and generalized linear model families, draw from sampling distributions of regression estimates, and perform visual inference on diagnostics from model fits.

URL https://www.refsmmat.com/regressinator/,
https://github.com/capnrefsmmat/regressinator

BugReports https://github.com/capnrefsmmat/regressinator/issues

License MIT + file LICENSE

Encoding UTF-8

RoxygenNote 7.2.3

Depends R (>= 4.1)

Imports broom, cli, dplyr, ggplot2, insight, nullabor, purrr, rlang, tibble, tidyr, tidyselect

Suggests rmarkdown, knitr, mvtnorm, palmerpenguins, patchwork, testthat (>= 3.0.0)

VignetteBuilder knitr

Config/testthat/edition 3

NeedsCompilation no

Author Alex Reinhart [aut, cre] (<https://orcid.org/0000-0002-6658-514X>)

Maintainer Alex Reinhart <areinhar@stat.cmu.edu>

Repository CRAN

Date/Publication 2023-08-11 13:40:02 UTC

R topics documented:

`augment_longer` *Augment a model fit with residuals, in "long" format*

Description

Use `broom::augment()` to augment a model fit with residual and fit information, then reformat the resulting data frame into a "long" format with one row per predictor per observation, to facilitate plotting of the result.

Usage

```
augment_longer(x, ...)
```

Arguments

`x`	A model fit object, such as those returned by `lm()` or `glm()`. See the broom documentation for the full list of model types supported.
`...`	Additional arguments passed to `broom::augment()`.

Details

When there are factor predictors, this function is less useful. Because a data frame column can contain values of only one type, factor or character values will force all values of the `.predictor_value` column, including for other predictors, to be converted.

The name comes by analogy to `tidyr::pivot_longer()`, and the concept of long versus wide data formats.

Value

A data frame (tibble) in similar form to those produced by `broom::augment()`, but expanded to have one row per predictor per observation. Columns `.predictor_name` and `.predictor_value` identify the predictor and its value. An additional column `.obs` records the original observation numbers so results can be matched to observations in the original model data.

Examples

```
fit <- lm(mpg ~ cyl + disp + hp, data = mtcars)

# each observation appears 3 times, once per predictor:
augment_longer(fit)
```

`binned_residuals` *Obtain binned residuals for a model*

Description

Construct a data frame by binning the fitted values or predictors of a model into discrete bins of equal width, and calculating the average value of the residuals within each bin.

Usage

```
binned_residuals(fit, predictors = !".fitted", breaks = NULL, ...)
```

Arguments

`fit`	The model to obtain residuals for. This can be a model fit with `lm()` or `glm()`, or any model that has `residuals()` and `fitted()` methods.
`predictors`	Predictors to calculate binned residuals for. Defaults to all predictors, skipping factors. Predictors can be specified using tidyselect syntax; see `help("language", package = "tidyselect")` and the examples below. Specify `predictors = .fitted` to obtain binned residuals versus fitted values.
`breaks`	Number of bins to create. If `NULL`, a default number of breaks is chosen based on the number of rows in the data.
`...`	Additional arguments passed on to `residuals()`. The most useful additional argument is typically `type`, to select the type of residuals to produce (such as standardized residuals or deviance residuals).

Details

In many generalized linear models, the residual plots (Pearson or deviance) are not useful because the response variable takes on very few possible values, causing strange patterns in the residuals. For instance, in logistic regression, plotting the residuals versus covariates usually produces two curved lines.

If we first bin the data, i.e. divide up the observations into `breaks` bins based on their fitted values, we can calculate the average residual within each bin. This can be more informative: if a region has 20 observations and its average residual value is large, this suggests those observations are collectively poorly fit. We can also bin each predictor and calculate averages within those bins, allowing the detection of misspecification for specific model terms.

Value

Data frame (tibble) with one row per bin *per selected predictor*, and the following columns:

`.bin` Bin number.

`n` Number of observations in this bin.

`predictor_name` Name of the predictor that has been binned.

`predictor_min`, `predictor_max`, `predictor_mean`, `predictor_sd`
Minimum, maximum, mean, and standard deviation of the predictor (or fitted values).

`resid_mean` Mean residual in this bin.

`resid_sd` Standard deviation of residuals in this bin.

Limitations

Factor predictors (as factors, logical, or character vectors) are detected automatically and omitted. However, if a numeric variable is converted to factor in the model formula, such as with `y ~ factor(x)`, the function cannot determine the appropriate type and will raise an error. Create factors as needed in the source data frame *before* fitting the model to avoid this issue.

References

Gelman, A., Hill, J., and Vehtari, A. (2021). *Regression and Other Stories*. Section 14.5. Cambridge University Press.

See Also

`partial_residuals()` for the related partial residuals; `vignette("logistic-regression-diagnostics")` and `vignette("other-glm-diagnostics")` for examples of use and interpretation of binned residuals in logistic regression and GLMs; `bin_by_interval()` and `bin_by_quantile()` to bin data and calculate other values in each bin

Examples

```
fit <- lm(mpg ~ disp + hp, data = mtcars)

# Automatically bins both predictors:
binned_residuals(fit, breaks = 5)

# Just bin one predictor, selected with tidyselect syntax. Multiple could be
# selected with c().
binned_residuals(fit, disp, breaks = 5)
```

```
# Bin the fitted values:
binned_residuals(fit, predictors = .fitted)

# Bins are made using the predictor, not regressors derived from it, so here
# disp is binned, not its polynomial
fit2 <- lm(mpg ~ poly(disp, 2), data = mtcars)
binned_residuals(fit2)
```

`bin_by_interval` *Group a data frame into bins*

Description

Groups a data frame (similarly to `dplyr::group_by()`) based on the values of a column, either by dividing up the range into equal pieces or by quantiles.

Usage

```
bin_by_interval(.data, col, breaks = NULL)

bin_by_quantile(.data, col, breaks = NULL)
```

Arguments

`.data`	Data frame to bin
`col`	Column to bin by
`breaks`	Number of bins to create. `bin_by_interval()` also accepts a numeric vector of two or more unique cut points to use. If `NULL`, a default number of breaks is chosen based on the number of rows in the data. In `bin_by_quantile()`, if the number of unique values of the column is smaller than `breaks`, fewer bins will be produced.

Details

`bin_by_interval()` breaks the numerical range of that column into equal-sized intervals, or into intervals specified by `breaks`. `bin_by_quantile()` splits the range into pieces based on quantiles of the data, so each interval contains roughly an equal number of observations.

Value

Grouped data frame, similar to those returned by `dplyr::group_by()`. An additional column `.bin` indicates the bin number for each group. Use `dplyr::summarize()` to calculate values within each group, or other dplyr operations that work on groups.

Examples

```
suppressMessages(library(dplyr))
cars |>
  bin_by_interval(speed, breaks = 5) |>
  summarize(mean_speed = mean(speed),
            mean_dist = mean(dist))

cars |>
  bin_by_quantile(speed, breaks = 5) |>
  summarize(mean_speed = mean(speed),
            mean_dist = mean(dist))
```

---

by_level *Convert factor levels to numeric values*

Description

Replace each entry in a vector with its corresponding numeric value, for instance to use a factor variable to specify intercepts for different groups in a regression model.

Usage

```
by_level(x, ...)
```

Arguments

`x`	Vector of factor values
`...`	Mapping from factor levels to values. Can be provided either as a series of named arguments, whose names correspond to factor levels, or as a single named vector.

Value

Named vector of same length as `x`, with values replaced with those specified. Names are the original factor level name.

See Also

`rfactor()` to draw random factor levels, and the `forcats` package https://forcats.tidyverse.org/ for additional factor manipulation tools

Examples

```
foo <- factor(c("spam", "ham", "spam", "ducks"))

by_level(foo, spam = 4, ham = 10, ducks = 16.7)

by_level(foo, c("spam" = 4, "ham" = 10, "ducks" = 16.7))
```

```
# to define a population with a factor that affects the regression intercept
intercepts <- c("foo" = 2, "bar" = 30, "baz" = 7)
pop <- population(
  group = predictor("rfactor",
                    levels = c("foo", "bar", "baz"),
                    prob = c(0.1, 0.6, 0.3)),
  x = predictor("runif", min = 0, max = 10),
  y = response(by_level(group, intercepts) + 0.3 * x,
               error_scale = 1.5)
)
sample_x(pop, 5)
```

---

custom_family *Family representing a GLM with custom distribution and link function*

Description

Allows specification of the random component and link function for a response variable. In principle this could be used to specify any GLM family, but it is usually easier to use the predefined families, such as `gaussian()` and `binomial()`.

Usage

```
custom_family(distribution, inverse_link)
```

Arguments

`distribution`	The distribution of the random component. This should be in the form of a function taking one argument, the vector of values on the inverse link scale, and returning a vector of draws from the distribution.
`inverse_link`	The inverse link function.

Details

A GLM is specified by a combination of:

- Random component, i.e. the distribution that Y is drawn from
- Link function relating the mean of the random component to the linear predictor
- Linear predictor

Using `custom_family()` we can specify the random component and link function, while the linear predictor is set in `population()` when setting up the population relationships. A family specified this way can be used to specify a population (via `population()`), but can't be used to estimate a model (such as with `glm()`).

Value

A family object representing this family

See Also

ols_with_error() for the special case of linear regression with custom error distribution

Examples

```
# A zero-inflated Poisson family
rzeroinfpois <- function(ys) {
  n <- length(ys)
  rpois(n, lambda = ys * rbinom(n, 1, prob = 0.4))
}

custom_family(rzeroinfpois, exp)
```

---

decrypt — *Decrypt message giving the location of the true plot in a lineup*

Description

Decrypts the message printed by `model_lineup()` indicating the location of the true diagnostics in the lineup.

Usage

```
decrypt(...)
```

Arguments

...	Message to decrypt, specifying the location of the true diagnostics

Value

The decrypted message.

---

empirical_link — *Empirically estimate response values on the link scale*

Description

Calculates the average value of the response variable, and places this on the link scale. Plotting these against a predictor (by dividing the dataset into bins) can help assess the choice of link function.

Usage

```
empirical_link(response, family, na.rm = FALSE)
```

Arguments

`response`	Vector of response variable values.
`family`	Family object representing the response distribution and link function. Only the link function will be used.
`na.rm`	Should `NA` values of the response be stripped? Passed to `mean()` when calculating the mean of the response.

Value

Mean response value, on the link scale.

Examples

```
suppressMessages(library(dplyr))
suppressMessages(library(ggplot2))

mtcars |>
  bin_by_interval(disp, breaks = 5) |>
  summarize(
    mean_disp = mean(disp),
    link = empirical_link(am, binomial())
  ) |>
  ggplot(aes(x = mean_disp, y = link)) +
  geom_point()
```

`model_lineup` *Produce a lineup for a fitted model*

Description

A lineup hides diagnostics among "null" diagnostics, i.e. the same diagnostics calculated using models fit to data where all model assumptions are correct. For each null diagnostic, `model_lineup()` simulates new responses from the model using the fitted covariate values and the model's error distribution, link function, and so on. Hence the new response values are generated under ideal conditions: the fitted model is true and all assumptions hold. `decrypt()` reveals which diagnostics are the true diagnostics.

Usage

```
model_lineup(fit, fn = augment, nsim = 20, ...)
```

Arguments

`fit`	A model fit to data, such as by `lm()` or `glm()`

fn
: A diagnostic function. The function's first argument should be the fitted model, and it must return a data frame. Defaults to `broom::augment()`, which produces a data frame containing the original data and additional columns `.fitted`, `.resid`, and so on. To see a list of model types supported by `broom::augment()`, and to find documentation on the columns reported for each type of model, load the `broom` package and use `methods(augment)`.

nsim
: Number of total diagnostics. For example, if `nsim = 20`, the diagnostics for `fit` are hidden among 19 null diagnostics.

...
: Additional arguments passed to `fn` each time it is called.

Details

To generate different kinds of diagnostics, the user can provide a custom `fn`. The `fn` should take a model fit as its argument and return a data frame. For instance, the data frame might contain one row per observation and include the residuals and fitted values for each observation; or it might be a single row containing a summary statistic or test statistic.

`fn` will be called on the original `fit` provided. Then `parametric_boot_distribution()` will be used to simulate data from the model fit `nsim - 1` times, refit the model to each simulated dataset, and run `fn` on each refit model. The null distribution is conditional on X, i.e. the covariates used will be identical, and only the response values will be simulated. The data frames are concatenated with an additional `.sample` column identifying which fit each row came from.

When called, this function will print a message such as `decrypt("sD0f gCdC En JP2EdEPn ZY")`. This is how to get the location of the true diagnostics among the null diagnostics: evaluating this in the R console will produce a string such as `"True data in position 5"`.

Value

A data frame (tibble) with columns corresponding to the columns returned by `fn`. The additional column `.sample` indicates which set of diagnostics each row is from. For instance, if the true data is in position 5, selecting rows with `.sample == 5` will retrieve the diagnostics from the original model fit.

References

Buja et al. (2009). Statistical inference for exploratory data analysis and model diagnostics. *Philosophical Transactions of the Royal Society A*, 367 (1906), pp. 4361-4383. doi:10.1098/rsta.2009.0120

Wickham et al. (2010). Graphical inference for infovis. *IEEE Transactions on Visualization and Computer Graphics*, 16 (6), pp. 973-979. doi:10.1109/TVCG.2010.161

See Also

`parametric_boot_distribution()` to simulate draws by using the fitted model to draw new response values; `sampling_distribution()` to simulate draws from the population distribution, rather than from the model

Examples

```
fit <- lm(dist ~ speed, data = cars)
model_lineup(fit, nsim = 5)

resids_vs_speed <- function(f) {
  data.frame(resid = residuals(f),
             speed = model.frame(f)$speed)
}
model_lineup(fit, fn = resids_vs_speed, nsim = 5)
```

---

ols_with_error *Family representing a linear relationship with non-Gaussian errors*

Description

The `ols_with_error()` family can represent any non-Gaussian error, provided random variates can be drawn by an R function. A family specified this way can be used to specify a population (via `population()`), but can't be used to estimate a model (such as with `glm()`).

Usage

```
ols_with_error(error, ...)
```

Arguments

`error`	Function that can draw random variables from the non-Gaussian distribution, or a string giving the name of the function. For example, `rt` draws *t*-distributed random variates. The function must take an argument `n` indicating how many random variates to draw (as all random generation functions built into R do).
`...`	Further arguments passed to the `error` function to draw random variates, such as to specify degrees of freedom, shape parameters, or other parameters of the distribution. These arguments are evaluated with the model data in the environment, so they can be expressions referring to model data, such as values of the predictors.

Value

A family object representing this family.

See Also

`custom_family()` for fully custom families, including for GLMs

Examples

```
# t-distributed errors with 3 degrees of freedom
ols_with_error(rt, df = 3)

# A linear regression with t-distributed error, using error_scale to make
# errors large
population(
  x1 = predictor("rnorm", mean = 4, sd = 10),
  x2 = predictor("runif", min = 0, max = 10),
  y = response(0.7 + 2.2 * x1 - 0.2 * x2,
               family = ols_with_error(rt, df = 4),
               error_scale = 2.5)
)

# Cauchy-distributed errors
ols_with_error(rcauchy, scale = 3)

# A contaminated error distribution, where
# 95% of observations are Gaussian and 5% are Cauchy
rcontaminated <- function(n) {
  contaminant <- rbinom(n, 1, prob = 0.05)

  return(ifelse(contaminant == 1,
                rcauchy(n, scale = 20),
                rnorm(n, sd = 1)))
}
ols_with_error(rcontaminated)
```

---

parametric_boot_distribution

Simulate the distribution of estimates by parametric bootstrap

---

Description

Repeatedly simulates new response values by using the fitted model, holding the covariates fixed. By default, refits the same model to each simulated dataset, but an alternative model can be provided. Estimates, confidence intervals, or other quantities are extracted from each fitted model and returned as a tidy data frame.

Usage

```
parametric_boot_distribution(
  fit,
  alternative_fit = fit,
  fn = tidy,
  nsim = 100,
  ...
)
```

Arguments

fit
: A model fit to data, such as by `lm()` or `glm()`, to simulate new response values from.

alternative_fit
: A model fit to data, to refit to the data sampled from `fit`. Defaults to `fit`, but an alternative model can be provided to examine its behavior when `fit` is the true model.

fn
: Function to call on each new model fit to produce a data frame of estimates. Defaults to `broom::tidy()`, which produces a tidy data frame of coefficients, estimates, standard errors, and hypothesis tests.

nsim
: Number of total simulations to run.

...
: Additional arguments passed to `fn` each time it is called.

Details

The default behavior samples from a model and refits the same model to the sampled data; this is useful when, for example, exploring how model diagnostics look when the model is well-specified. Another common use of the parametric bootstrap is hypothesis testing, where we might simulate from a null model and fit an alternative model to the data, to obtain the null distribution of a particular estimate or statistic. Provide `alternative_fit` to have a specific model fit to each simulated dataset, rather than the model they are simulated from.

Only the response variable from the `fit` (or `alternative_fit`, if given) is redrawn; other response variables in the population are left unchanged.

Because `model_lineup()` uses the S3 generic methods `model.frame()`, `simulate()`, and `update()`, it can be used with any model fit for which methods are provided. In base R, this includes `lm()` and `glm()`.

Value

A data frame (tibble) with columns corresponding to the columns returned by `fn`. The additional column `.sample` indicates which fit each row is from.

See Also

`model_lineup()` to use resampling to aid in regression diagnostics; `sampling_distribution()` to simulate draws from the population distribution, rather than the null

Examples

```
# Bootstrap distribution of estimates:
fit <- lm(dist ~ speed, data = cars)
parametric_boot_distribution(fit, nsim = 5)

# Bootstrap distribution of estimates for a quadratic model, when true
# relationship is linear:
quad_fit <- lm(dist ~ poly(speed, 2), data = cars)
parametric_boot_distribution(fit, quad_fit, nsim = 5)
```

partial_residuals *Augment a model fit with partial residuals for all terms*

Description

Construct a data frame containing the model data, partial residuals for all quantitative predictors, and predictor effects, for use in residual diagnostic plots and other analyses. The result is in tidy form (one row per predictor per observation), allowing it to be easily manipulated for plots and simulations.

Usage

```
partial_residuals(fit, predictors = everything())
```

Arguments

`fit`	The model to obtain residuals for. This can be a model fit with `lm()` or `glm()`, or any model with a `predict()` method that accepts a `newdata` argument.
`predictors`	Predictors to calculate partial residuals for. Defaults to all predictors, skipping factors. Predictors can be specified using tidyselect syntax; see `help("language", package = "tidyselect")` and the examples below.

Value

Data frame (tibble) containing the model data and residuals in tidy form. There is one row *per selected predictor* per observation. All predictors are included as columns, plus the following additional columns:

`.obs`	Row number of this observation in the original model data frame.
`.predictor_name`	Name of the predictor this row gives the partial residual for.
`.predictor_value`	Value of the predictor this row gives the partial residual for.
`.partial_resid`	Partial residual for this predictor for this observation.
`.predictor_effect`	Predictor effect $\hat{\mu}(X_{if}, 0)$ for this observation.

Predictors and regressors

To define partial residuals, we must distinguish between the *predictors*, the measured variables we are using to fit our model, and the *regressors*, which are calculated from them. In a simple linear model, the regressors are equal to the predictors. But in a model with polynomials, splines, or other nonlinear terms, the regressors may be functions of the predictors.

For example, in a regression with a single predictor X, the regression model $Y = \beta_0 + \beta_1 X + e$ has one regressor, X. But if we choose a polynomial of degree 3, the model is $Y = \beta_0 + \beta_1 X + \beta_2 X^2 + \beta_3 X^3$, and the regressors are $\{X, X^2, X^3\}$.

Similarly, if we have predictors X_1 and X_2 and form a model with main effects and an interaction, the regressors are $\{X_1, X_2, X_1 X_2\}$.

Partial residuals are defined in terms of the predictors, not the regressors, and are intended to allow us to see the shape of the relationship between a particular predictor and the response, and to compare it to how we have chosen to model it with regressors. Partial residuals are not useful for categorical (factor) predictors, and so these are omitted.

Linear models

Consider a linear model where $\mathbb{E}[Y \mid X = x] = \mu(x)$. The mean function $\mu(x)$ is a linear combination of regressors. Let $\hat{\mu}$ be the fitted model and $\hat{\beta}_0$ be its intercept.

Choose a predictor X_f, the *focal* predictor, to calculate partial residuals for. Write the mean function as $\mu(X_f, X_o)$, where X_f is the value of the focal predictor, and X_o represents all other predictors.

If e_i is the residual for observation i, the partial residual is

$$r_{if} = e_i + (\hat{\mu}(x_{if}, 0) - \hat{\beta}_0).$$

Setting $X_o = 0$ means setting all other numeric predictors to 0; factor predictors are set to their first (baseline) level.

Generalized linear models

Consider a generalized linear model where $g(\mathbb{E}[Y \mid X = x]) = \mu(x)$, where g is a link function. Let $\hat{\mu}$ be the fitted model and $\hat{\beta}_0$ be its intercept.

Let e_i be the *working residual* for observation i, defined to be

$$e_i = (y_i - g^{-1}(x_i)) g'(x_i).$$

Choose a predictor X_f, the *focal* predictor, to calculate partial residuals for. Write μ as $\mu(X_f, X_o)$, where X_f is the value of the focal predictor, and X_o represents all other predictors. Hence $\mu(X_f, X_o)$ gives the model's prediction on the link scale.

The partial residual is again

$$r_{if} = e_i + (\hat{\mu}(x_{if}, 0) - \hat{\beta}_0).$$

Interpretation

In linear regression, because the residuals e_i should have mean zero in a well-specified model, plotting the partial residuals against x_f should produce a shape matching the modeled relationship μ. If the model is wrong, the partial residuals will appear to deviate from the fitted relationship. Provided the regressors are uncorrelated or approximately linearly related to each other, the plotted trend should approximate the true relationship between x_f and the response.

In generalized linear models, this is approximately true if the link function g is approximately linear over the range of observed x values.

Additionally, the function $\mu(X_f, 0)$ can be used to show the relationship between the focal predictor and the response. In a linear model, the function is linear; with polynomial or spline regressors, it is nonlinear. This function is the *predictor effect function*, and the estimated predictor effects $\hat{\mu}(X_{if}, 0)$ are included in this function's output.

Limitations

Factor predictors (as factors, logical, or character vectors) are detected automatically and omitted. However, if a numeric variable is converted to factor in the model formula, such as with `y ~ factor(x)`, the function cannot determine the appropriate type and will raise an error. Create factors as needed in the source data frame *before* fitting the model to avoid this issue.

References

R. Dennis Cook (1993). "Exploring Partial Residual Plots", *Technometrics*, 35:4, 351-362. doi:10.1080/00401706.1993.10485350

Cook, R. Dennis, and Croos-Dabrera, R. (1998). "Partial Residual Plots in Generalized Linear Models." *Journal of the American Statistical Association* 93, no. 442: 730–39. doi:10.2307/2670123

Fox, J., & Weisberg, S. (2018). "Visualizing Fit and Lack of Fit in Complex Regression Models with Predictor Effect Plots and Partial Residuals." *Journal of Statistical Software*, 87(9). doi:10.18637/jss.v087.i09

See Also

`binned_residuals()` for the related binned residuals; `vignette("linear-regression-diagnostics")`, `vignette("logistic-regression-diagnostics")`, and `vignette("other-glm-diagnostics")` for examples of plotting and interpreting partial residuals

Examples

```
fit <- lm(mpg ~ cyl + disp + hp, data = mtcars)
partial_residuals(fit)

# You can select predictors with tidyselect syntax:
partial_residuals(fit, c(disp, hp))

# Predictors with multiple regressors are supported:
fit2 <- lm(mpg ~ poly(disp, 2), data = mtcars)
partial_residuals(fit2)

# Allowing an interaction by number of cylinders is fine, but partial
# residuals are not generated for the factor. Notice the factor must be
# created first, not in the model formula:
mtcars$cylinders <- factor(mtcars$cyl)
fit3 <- lm(mpg ~ cylinders * disp + hp, data = mtcars)
partial_residuals(fit3)
```

`population` *Define the population generalized regression relationship*

Description

Specifies a hypothetical infinite population of cases. Each case has some predictor variables and one or more response variables. The relationship between the variables and response variables are defined, as well as the population marginal distribution of each predictor variable.

Usage

```
population(...)
```

Arguments

`...`	A sequence of named arguments defining predictor and response variables. These are evaluated in order, so later response variables may refer to earlier predictor and response variables. All predictors should be provided first, before any response variables.

Value

A population object.

See Also

`predictor()` and `response()` to define the population; `sample_x()` and `sample_y()` to draw samples from it

Examples

```
# A population with a simple linear relationship
linear_pop <- population(
  x1 = predictor("rnorm", mean = 4, sd = 10),
  x2 = predictor("runif", min = 0, max = 10),
  y = response(0.7 + 2.2 * x1 - 0.2 * x2, error_scale = 1.0)
)

# A population whose response depends on local variables
slope <- 2.2
intercept <- 0.7
sigma <- 2.5
variable_pop <- population(
  x = predictor("rnorm"),
  y = response(intercept + slope * x, error_scale = sigma)
)

# Response error scale is heteroskedastic and depends on predictors
heteroskedastic_pop <- population(
  x1 = predictor("rnorm", mean = 4, sd = 10),
  x2 = predictor("runif", min = 0, max = 10),
  y = response(0.7 + 2.2 * x1 - 0.2 * x2,
               error_scale = 1 + x2 / 10)
)

# A binary outcome Y, using a binomial family with logistic link
binary_pop <- population(
  x1 = predictor("rnorm", mean = 4, sd = 10),
  x2 = predictor("runif", min = 0, max = 10),
  y = response(0.7 + 2.2 * x1 - 0.2 * x2,
               family = binomial(link = "logit"))
```

```
)

# A binomial outcome Y, with 10 trials per observation, using a logistic link
# to determine the probability of success for each trial
binomial_pop <- population(
  x1 = predictor("rnorm", mean = 4, sd = 10),
  x2 = predictor("runif", min = 0, max = 10),
  y = response(0.7 + 2.2 * x1 - 0.2 * x2,
               family = binomial(link = "logit"),
               size = 10)
)

# Another binomial outcome, but the number of trials depends on another
# predictor
binom_size_pop <- population(
  x1 = predictor("rnorm", mean = 4, sd = 10),
  x2 = predictor("runif", min = 0, max = 10),
  trials = predictor("rpois", lambda = 20),
  y = response(0.7 + 2.2 * x1 - 0.2 * x2,
               family = binomial(link = "logit"),
               size = trials)
)

# A population with a simple linear relationship and collinearity. Because X
# is bivariate, there will be two predictors, named x1 and x2.
library(mvtnorm)
collinear_pop <- population(
  x = predictor("rmvnorm", mean = c(0, 1),
                sigma = matrix(c(1, 0.8, 0.8, 1), nrow = 2)),
  y = response(0.7 + 2.2 * x1 - 0.2 * x2, error_scale = 1.0)
)
```

predictor *Specify the distribution of a predictor variable*

Description

Predictor variables can have any marginal distribution as long as a function is provided to sample from the distribution. Multivariate distributions are also supported: if the random generation function returns multiple columns, multiple random variables will be created, successively numbered.

Usage

```
predictor(dist, ...)
```

Arguments

`dist`	Name (as character vector) of the function to generate draws from this predictor's distribution.
`...`	Additional arguments to pass to `dist` when generating draws.

Details

The random generation function must take an argument named n specifying the number of draws. For univariate distributions, it should return a vector of length n; for multivariate distributions, it should return an array or matrix with n rows and a column per variable.

Multivariate predictors are successively numbered. For instance, if predictor X is specified with

```
library(mvtnorm)
predictor(dist = "rmvnorm", mean = c(0, 1),
          sigma = matrix(c(1, 0.5, 0.5, 1), nrow = 2))
```

then the population predictors will be named X1 and X2, and will have covariance 0.5.

Value

A predictor_dist object, to be used in population() to specify a population distribution

Examples

```
# Univariate normal distribution
predictor(dist = "rnorm", mean = 10, sd = 2.5)

# Multivariate normal distribution
library(mvtnorm)
predictor(dist = "rmvnorm", mean = c(0, 1, 7))
```

---

response *Specify a response variable in terms of predictors*

Description

Response variables are related to predictors (and other response variables) through a link function and response distribution. First the expression provided is evaluated using the predictors, to give this response variable's value on the link scale; then the inverse link function and response distribution are used to get the response value. See Details for more information.

Usage

```
response(expr, family = gaussian(), error_scale = NULL, size = 1L)
```

Arguments

expr	An expression, in terms of other predictor or response variables, giving this predictor's value on the link scale.
family	The family of this response variable, e.g. gaussian() for an ordinary Gaussian linear relationship.

error_scale
: Scale factor for errors. Used only for linear families, such as gaussian() and ols_with_error(). Errors drawn while simulating the response variable will be multiplied by this scale factor. The scale factor can be a scalar value (such as a fixed standard deviation), or an expression in terms of the predictors, which will be evaluated when simulating response data. For generalized linear models, leave as NULL.

size
: When the family is binomial(), this is the number of trials for each observation. Defaults to 1, as in logistic regression. May be specified either as a vector of the same length as the number of observations or as a scalar. May be written terms of other predictor or response variables. For other families, size is ignored.

Details

Response variables are drawn based on a typical generalized linear model setup. Let Y represent the response variable and X represent the predictor variables. We specify that

$$Y \mid X \sim \text{SomeDistribution},$$

where

$$\mathbb{E}[Y \mid X = x] = g^{-1}(\mu(x)).$$

Here $\mu(X)$ is the expression expr, and both the distribution and link function g are specified by the family provided. For instance, if the family is gaussian(), the distribution is Normal and the link is the identity function; if the family is binomial(), the distribution is binomial and the link is (by default) the logistic link.

Response families:

The following response families are supported.

gaussian() The default family is gaussian() with the identity link function, specifying the relationship

$$Y \mid X \sim \text{Normal}(\mu(X), \sigma^2),$$

where σ^2 is given by error_scale.

ols_with_error() Allows specification of custom non-Normal error distributions, specifying the relationship

$$Y = \mu(X) + e,$$

where e is drawn from an arbitrary distribution, specified by the error argument to ols_with_error().

binomial() Binomial responses include binary responses (as in logistic regression) and responses giving a total number of successes out of a number of trials. The response has distribution

$$Y \mid X \sim \text{Binomial}(N, g^{-1}(\mu(X))),$$

where N is set by the size argument and g is the link function. The default link is the logistic link, and others can be chosen with the link argument to binomial(). The default N is 1, representing a binary outcome.

`poisson()` Poisson-distributed responses with distribution

$$Y \mid X \sim \text{Poisson}(g^{-1}(\mu(X))),$$

where g is the link function. The default link is the log link, and others can be chosen with the `link` argument to `poisson()`.

`custom_family()` Responses drawn from an arbitrary distribution with arbitrary link function, i.e.

$$Y \mid X \sim \text{SomeDistribution}(g^{-1}(\mu(X))),$$

where both g and SomeDistribution are specified by arguments to `custom_family()`.

Evaluation and scoping:

The `expr`, `error_scale`, and `size` arguments are evaluated only when simulating data for this response variable. They are evaluated in an environment with access to the predictor variables and the preceding response variables, which they can refer to by name. Additionally, these arguments can refer to variables in scope when the enclosing `population()` was defined. See the Examples below.

Value

A `response_dist` object, to be used in `population()` to specify a population distribution

See Also

`predictor()` and `population()` to define populations; `ols_with_error()` and `custom_family()` for custom response distributions

Examples

```
# Defining a binomial response. The expressions can refer to other predictors
# and to the environment where the `population()` is defined:
slope1 <- 2.5
slope2 <- -3
intercept <- -4.6
size <- 10
population(
  x1 = predictor("rnorm"),
  x2 = predictor("rnorm"),
  y = response(intercept + slope1 * x1 + slope2 * x2,
               family = binomial(), size = size)
)
```

rfactor *Draw random values from a factor variable*

Description

To specify the population distribution of a factor variable, specify the probability for each of its factor levels. When drawn from the population, factor levels are drawn with replacement according to their probability.

Usage

```
rfactor(n, levels, prob = rep_len(1/length(levels), length(levels)))
```

Arguments

n	Number of values to draw
levels	Character vector specifying the levels for the factor
prob	Vector specifying the probability for each factor level

Value

Sample of `n` values from `levels`, drawn in proportion to their probabilities. By default, levels are equally likely.

See Also

`by_level()` to assign numeric values based on factor levels, such as to set population regression coefficients by factor level

Examples

```
rfactor(5, c("foo", "bar", "baz"), c(0.4, 0.3, 0.3))
```

sample_x *Draw a data frame from the specified population.*

Description

Sampling is split into two steps, for predictors and for response variables, to allow users to choose which to simulate. `sample_x()` will only sample predictor variables, and `sample_y()` will augment a data frame of predictors with columns for response variables, overwriting any already present. Hence one can use `sample_y()` as part of a simulation with fixed predictors, for instance.

Usage

```
sample_x(population, n)

sample_y(xs)
```

Arguments

`population`	Population, as defined by `population()`.
`n`	Number of observations to draw from the population.
`xs`	Data frame of predictor values drawn from the population, as obtained from `sample_x()`.

Value

Data frame (tibble) of `n` rows, with columns matching the variables specified in the population.

Examples

```
# A population with a simple linear relationship
pop <- population(
  x1 = predictor("rnorm", mean = 4, sd = 10),
  x2 = predictor("runif", min = 0, max = 10),
  y = response(0.7 + 2.2 * x1 - 0.2 * x2, error_scale = 1.0)
)

xs <- pop |>
  sample_x(5)

xs

xs |>
  sample_y()
```

`sampling_distribution` *Simulate the sampling distribution of estimates from a population*

Description

Repeatedly refits the model to new samples from the population, calculates estimates for each fit, and compiles a data frame of the results.

Usage

```
sampling_distribution(fit, data, fn = tidy, nsim = 100, fixed_x = TRUE, ...)
```

Arguments

`fit`	A model fit to data, such as by `lm()` or `glm()`, to refit to each sample from the population.
`data`	Data drawn from a `population()`, using `sample_x()` and possibly `sample_y()`. The `population()` specification is used to draw the samples.
`fn`	Function to call on each new model fit to produce a data frame of estimates. Defaults to `broom::tidy()`, which produces a tidy data frame of coefficients, estimates, standard errors, and hypothesis tests.
`nsim`	Number of simulations to run.
`fixed_x`	If `TRUE`, the default, the predictor variables are held fixed and only the response variables are redrawn from the population. If `FALSE`, the predictor and response variables are drawn jointly.
`...`	Additional arguments passed to `fn` each time it is called.

Details

To generate sampling distributions of different quantities, the user can provide a custom `fn`. The `fn` should take a model fit as its argument and return a data frame. For instance, the data frame might contain one row per estimated coefficient and include the coefficient and its standard error; or it might contain only one row of model summary statistics.

Refitting is done using the S3 generic `update()`, so this function can be used with any model fit that supports `update()`. In base R, this includes `lm()` and `glm()`, and many other model fits.

Value

Data frame (tibble) of `nsim + 1` simulation results, formed by concatenating together the data frames returned by `fn`. The `.sample` column identifies which simulated sample each row came from. Rows with `.sample == 0` come from the original `fit`.

See Also

`parametric_boot_distribution()` to simulate draws from a fitted model, rather than from the population

Examples

```
pop <- population(
  x1 = predictor("rnorm", mean = 4, sd = 10),
  x2 = predictor("runif", min = 0, max = 10),
  y = response(0.7 + 2.2 * x1 - 0.2 * x2, error_scale = 4.0)
)

d <- sample_x(pop, n = 20) |>
  sample_y()

fit <- lm(y ~ x1 + x2, data = d)
# using the default fn = broom::tidy(). conf.int argument is passed to
# broom::tidy()
```

```
samples <- sampling_distribution(fit, d, conf.int = TRUE)
samples

suppressMessages(library(dplyr))
# the model is correctly specified, so the estimates are unbiased:
samples |>
  group_by(term) |>
  summarize(mean = mean(estimate),
            sd = sd(estimate))

# instead of coefficients, get the sampling distribution of R^2
rsquared <- function(fit) {
  data.frame(r2 = summary(fit)$r.squared)
}
sampling_distribution(fit, d, rsquared, nsim = 10)
```

Index