Package 'ggeffects'

January 20, 2025

Type Package

Encoding UTF-8

Title Create Tidy Data Frames of Marginal Effects for 'ggplot' from Model Outputs

Version 2.1.0

Maintainer Daniel Lüdecke <d.luedecke@uke.de>

Description Compute marginal effects and adjusted predictions from statistical models and returns the result as tidy data frames. These data frames are ready to use with the 'ggplot2'-package. Effects and predictions can be calculated for many different models. Interaction terms, splines and polynomial terms are also supported. The main functions are ggpredict(), ggemmeans() and ggeffect(). There is a generic plot()-method to plot the results using 'ggplot2'.

Depends R (>= 3.6)

Imports graphics, insight (>= 1.0.1), datawizard (>= 1.0.0), stats, utils

Suggests AER, afex, aod, bayestestR, betareg, brglm, brglm2, brms, broom, car, carData, clubSandwich, dfidx, effects (>= 4.2-2), emmeans (>= 1.8.9), fixest, gam, gamlss, gamm4, gee, geepack, ggplot2, ggrepel, GLMMadaptive, glmmTMB (>= 1.1.7), gridExtra, gt, haven, htmltools, httr2, jsonlite, knitr, lme4 (>= 1.1-35), logistf, logitr, marginaleffects (>= 0.24.0), MASS, Matrix, mice, MCMCglmm, MuMIn, mgcv, mclogit, mlogit, nestedLogit (>= 0.3.0), nlme, nnet, ordinal, parameters, parsnip, patchwork, pscl, plm, quantreg, rmarkdown, rms, robustbase, rstanarm, rstantools, sandwich, sdmTMB (>= 0.4.0), see, sjlabelled (>= 1.1.2), sjstats, speedglm, survey, survival, testthat, tibble, tinytable (>= 0.1.0), vdiffr, withr, VGAM

URL https://strengejacke.github.io/ggeffects/

BugReports https://github.com/strengejacke/ggeffects/issues/

RoxygenNote 7.3.2

VignetteBuilder knitr

Config/testthat/edition 3

License MIT + file LICENSE

LazyData true

NeedsCompilation no

Author Daniel Lüdecke [aut, cre] (<https://orcid.org/0000-0002-8895-3206>),
Frederik Aust [ctb] (<https://orcid.org/0000-0003-4900-788X>),
Sam Crawley [ctb] (<https://orcid.org/0000-0002-7847-0411>),
Mattan S. Ben-Shachar [ctb] (<https://orcid.org/0000-0002-4287-4801>),
Sean C. Anderson [ctb] (<https://orcid.org/0000-0001-9563-1937>)

Repository CRAN

Date/Publication 2025-01-20 10:50:02 UTC

Contents

`as.data.frame.ggeffects` *Adjusted predictions from regression models*

Description

After fitting a model, it is useful generate model-based estimates (expected values, or *adjusted predictions*) of the response variable for different combinations of predictor values. Such estimates can be used to make inferences about relationships between variables.

The **ggeffects** package computes marginal means and adjusted predicted values for the response, at the margin of specific values or levels from certain model terms. The package is built around three core functions: `predict_response()` (understanding results), `test_predictions()` (testing results for statistically significant differences) and `plot()` (communicate results).

By default, adjusted predictions or marginal means are by returned on the *response* scale, which is the easiest and most intuitive scale to interpret the results. There are other options for specific models as well, e.g. with zero-inflation component (see documentation of the `type`-argument). The result is returned as consistent data frame, which is nicely printed by default. `plot()` can be used to easily create figures.

The main function to calculate marginal means and adjusted predictions is `predict_response()`. In previous versions of **ggeffects**, the functions `ggpredict()`, `ggemmeans()`, `ggeffect()` and `ggaverage()` were used to calculate marginal means and adjusted predictions. These functions are still available, but `predict_response()` as a "wrapper" around these functions is the preferred way to do this now.

- `ggpredict()` calls `get_predictions()` (which in turn calls `stats::predict()`)
- `ggemmeans()` calls `emmeans::emmeans()`
- `ggaverage()` calls `marginaleffects::avg_predictions()`
- `ggeffect()` calls `effects::Effect()`

Usage

```
## S3 method for class 'ggeffects'
as.data.frame(
  x,
  row.names = NULL,
  optional = FALSE,
  ...,
  stringsAsFactors = FALSE,
  terms_to_colnames = FALSE
)

ggaverage(
  model,
  terms,
  ci_level = 0.95,
```

```
  type = "fixed",
  typical = "mean",
  condition = NULL,
  back_transform = TRUE,
  vcov = NULL,
  vcov_args = NULL,
  weights = NULL,
  verbose = TRUE,
  ...
)

ggeffect(
  model,
  terms,
  ci_level = 0.95,
  bias_correction = FALSE,
  verbose = TRUE,
  ...
)

ggemmeans(
  model,
  terms,
  ci_level = 0.95,
  type = "fixed",
  typical = "mean",
  condition = NULL,
  interval = "confidence",
  back_transform = TRUE,
  vcov = NULL,
  vcov_args = NULL,
  bias_correction = FALSE,
  weights = NULL,
  verbose = TRUE,
  ...
)

ggpredict(
  model,
  terms,
  ci_level = 0.95,
  type = "fixed",
  typical = "mean",
  condition = NULL,
  interval = "confidence",
  back_transform = TRUE,
  vcov = NULL,
  vcov_args = NULL,
```

```
  bias_correction = FALSE,
  verbose = TRUE,
  ...
)
```

Arguments

x	An object of class ggeffects, as returned by predict_response(), ggpredict(), ggeffect(), ggaverage() or ggemmeans().
row.names	NULL or a character vector giving the row names for the data frame. Missing values are not allowed.
optional	logical. If TRUE, setting row names and converting column names (to syntactic names: see make.names) is optional. Note that all of R's **base** package as.data.frame() methods use optional only for column names treatment, basically with the meaning of data.frame(*, check.names = !optional). See also the make.names argument of the matrix method.
...	Arguments are passed down to ggpredict() (further down to predict()) or ggemmeans() (and thereby to emmeans::emmeans()), If type = "simulate", ... may also be used to set the number of simulation, e.g. nsim = 500. When calling ggeffect(), further arguments passed down to effects::Effect().
stringsAsFactors	logical: should the character vector be converted to a factor?
terms_to_colnames	Logical, if TRUE, standardized column names (like "x", "group" or "facet") are replaced by the variable names of the focal predictors specified in terms.
model	A model object, or a list of model objects.
terms	Names of those terms from model, for which predictions should be displayed (so called *focal terms*). Can be: • A character vector, specifying the names of the focal terms. This is the preferred and probably most flexible way to specify focal terms, e.g. terms = "x [40:60]", to calculate predictions for the values 40 to 60. • A list, where each element is a named vector, specifying the focal terms and their values. This is the "classical" R way to specify focal terms, e.g. list(x = 40:60). • A formula, e.g. terms = ~ x + z, which is internally converted to a character vector. This is probably the least flexible way, as you cannot specify representative values for the focal terms. • A data frame representing a "data grid" or "reference grid". Predictions are then made for all combinations of the variables in the data frame. terms at least requires one variable name. The maximum length is five terms, where the second to fifth term indicate the groups, i.e. predictions of the first term are grouped at meaningful values or levels of the remaining terms (see values_at()). It is also possible to define specific values for focal terms, at which adjusted predictions should be calculated (see details below). All remaining covariates that are not specified in terms are "marginalized", see the margin argument in ?predict_response. See also argument condition to fix

non-focal terms to specific values, and argument `typical` for `ggpredict()` or `ggemmeans()`.

`ci_level` Numeric, the level of the confidence intervals. Use `ci_level = NA` if confidence intervals should not be calculated (for instance, due to computation time). Typically, confidence intervals are based on the returned standard errors for the predictions, assuming a t- or normal distribution (based on the model and the available degrees of freedom, i.e. roughly `+/- 1.96 * SE`). See introduction of this vignette for more details.

`type` Character, indicating whether predictions should be conditioned on specific model components or not, or whether population or unit-level predictions are desired. Consequently, most options only apply for survival models, mixed effects models and/or models with zero-inflation (and their Bayesian counter-parts); only exception is `type = "simulate"`, which is available for some other model classes as well (which respond to `simulate()`).

Note 1: For `brmsfit`-models with zero-inflation component, there is no `type = "zero_inflated"` nor `type = "zi_random"`; predicted values for these models *always* condition on the zero-inflation part of the model. The same is true for `MixMod`-models from **GLMMadaptive** with zero-inflation component (see 'Details').

Note 2: If `margin = "empirical"`, or when calling `ggaverage()` respectively, (i.e. counterfactual predictions), the `type` argument is handled differently. It is set to `"response"` by default, but usually accepts all possible options from the `type`-argument of the model's respective `predict()` method. E.g., passing a `glm` object would allow the options `"response"`, `"link"`, and `"terms"`. For models with zero-inflation component, the below mentioned options `"fixed"`, `"zero_inflated"` and `"zi_prob"` can also be used and will be "translated" into the corresponding `type` option of the model's respective `predict()`-method.

Note 3: If `margin = "marginalmeans"`, or when calling `ggemmeans()` respectively, `type = "random"` and `type = "zi_random"` are not available, i.e. no unit-level predictions are possible.

- `"fixed"` (or `"count"`)
  Predicted values are conditioned on the fixed effects or conditional model only. For mixed models, predicted values are on the *population-level*, i.e. `re.form = NA` when calling `predict()`. For models with zero-inflation component, this type would return the predicted mean from the count component (without conditioning on the zero-inflation part).
- `"random"`
  This only applies to mixed models, and `type = "random"` does not condition on the zero-inflation component of the model. Use this for *unit-level* predictions, i.e. predicted values for each level of the random effects groups. Add the name of the related random effect term to the `terms`-argument (for more details, see this vignette).
- `"zero_inflated"` (or `"zi"`)
  Predicted values are conditioned on the fixed effects and the zero-inflation component, returning the expected value of the response (`mu*(1-p)`). For For mixed models with zero-inflation component (e.g. from package **glmmTMB**),

this would return the expected response `mu*(1-p)` on the *population-level*. See 'Details'.

- `"zi_random"` (or `"zero_inflated_random"`)
  This only applies to mixed models. Predicted values are conditioned on the fixed effects and the zero-inflation component. Use this for *unit-level* predictions, i.e. predicted values for each level of the random effects groups. Add the name of the related random effect term to the `terms`-argument (for more details, see this vignette).
- `"zi_prob"`
  Returns the predicted zero-inflation probability, i.e. probability of a structural or "true" zero (see this vignette for a short introduction into zero-inflation models).
- `"simulate"`
  Predicted values and confidence resp. prediction intervals are based on simulations, i.e. calls to `simulate()`. This type of prediction takes all model uncertainty into account. Currently supported models are objects of class `lm`, `glm`, `glmmTMB`, `wbm`, `MixMod` and `merMod`. Use `nsim` to set the number of simulated draws (see `...` for details).
- `"survival"`, `"cumulative_hazard"` and `"quantile"`
  `"survival"` and `"cumulative_hazard"` apply only to `coxph`-objects from the **survial**-package. These options calculate the survival probability or the cumulative hazard of an event. `type = "quantile"` only applies to `survreg`-objects from package **survival**, which returns the predicted quantiles. For this option, the `p` argument is passed to `predict()`, so that quantiles for different probabilities can be calculated, e.g. `predict_response(..., type = "quantile", p = c(0.2, 0.5, 0.8))`.

When `margin = "empirical"` (or when calling `ggaverage()`), the `type` argument accepts all values from the `type`-argument of the model's respective `predict()`-method.

`typical` Character vector, naming the function to be applied to the covariates (non-focal terms) over which the effect is "averaged". The default is `"mean"`. Can be `"mean"`, `"weighted.mean"`, `"median"`, `"mode"` or `"zero"`, which call the corresponding R functions (except `"mode"`, which calls an internal function to compute the most common value); `"zero"` simply returns 0. By default, if the covariate is a factor, only `"mode"` is applicable; for all other values (including the default, `"mean"`) the reference level is returned. For character vectors, only the mode is returned. You can use a named vector to apply different functions to integer, numeric and categorical covariates, e.g. `typical = c(numeric = "median", factor = "mode")`. If `typical` is `"weighted.mean"`, weights from the model are used. If no weights are available, the function falls back to `"mean"`. **Note** that this argument is ignored for `predict_response()`, because the `margin` argument takes care of this.

`condition` Named character vector, which indicates covariates that should be held constant at specific values. Unlike `typical`, which applies a function to the covariates to determine the value that is used to hold these covariates constant, `condition` can be used to define exact values, for instance `condition = c(covariate1 = 20, covariate2 = 5)`. See 'Examples'.

back_transform Logical, if TRUE (the default), predicted values for log-, log-log, exp, sqrt and similar transformed responses will be back-transformed to original response-scale. See insight::find_transformation() for more details.

vcov Variance-covariance matrix used to compute uncertainty estimates (e.g., for confidence intervals based on robust standard errors). This argument accepts a covariance matrix, a function which returns a covariance matrix, or a string which identifies the function to be used to compute the covariance matrix.

- A covariance matrix
- A function which returns a covariance matrix (e.g., stats::vcov())
- A string which indicates the kind of uncertainty estimates to return.
  - Heteroskedasticity-consistent: "HC", "HC0", "HC1", "HC2", "HC3", "HC4", "HC4m", "HC5". See ?sandwich::vcovHC
  - Cluster-robust: "vcovCR", "CR0", "CR1", "CR1p", "CR1S", "CR2", "CR3". See ?clubSandwich::vcovCR.
  - Bootstrap: "BS", "xy", "fractional", "jackknife", "residual", "wild", "mammen", "norm", "webb". See ?sandwich::vcovBS
  - Other sandwich package functions: "HAC", "PC", "CL", or "PL".

If NULL, standard errors (and confidence intervals) for predictions are based on the standard errors as returned by the predict()-function. **Note** that probably not all model objects that work with predict_response() are also supported by the **sandwich** or **clubSandwich** packages.

See details in this vignette.

vcov_args List of arguments to be passed to the function identified by the vcov argument. This function is typically supplied by the **sandwich** or **clubSandwich** packages. Please refer to their documentation (e.g., ?sandwich::vcovHAC) to see the list of available arguments. If no estimation type (argument type) is given, the default type for "HC" equals the default from the **sandwich** package; for type "CR" the default is set to "CR3". For other defaults, refer to the documentation in the **sandwich** or **clubSandwich** package.

weights This argument is used in two different ways, depending on the margin argument.

- When margin = "empirical" (or when calling ggaverag()), weights can either be a character vector, naming the weigthing variable in the data, or a vector of weights (of same length as the number of observations in the data). This variable will be used to weight adjusted predictions.
- When margin = "marginalmeans" (or when calling ggemmeans()), weights must be a character vector and is passed to emmeans::emmeans(), specifying weights to use in averaging non-focal categorical predictors. Options are "equal", "proportional", "outer", "cells", "flat", and "show.levels". See ?emmeans::emmeans for details.

verbose Toggle messages or warnings.

bias_correction

Logical, if TRUE, adjusts for bias-correction when back-transforming the predicted values (to the response scale) for non-Gaussian *mixed models*. Back-transforming the the population-level predictions ignores the effect of the variation around the population mean, so the result on the original data scale is

biased due to *Jensen's inequality*. That means, when `type = "fixed"` (the default) and population level predictions are returned, it is recommended to set `bias_correction = TRUE`. To apply bias-correction, a valid value of sigma is required, which is extracted by default using `insight::get_variance_residual()`. Optionally, to provide own estimates of uncertainty, use the `sigma` argument. Note that `bias_correction` currently only applies to mixed models, where there are additive random components involved and where that bias-adjustment can be appropriate. If `ggemmeans()` is called, bias-correction can also be applied to GEE-models.

`interval` Type of interval calculation, can either be `"confidence"` (default) or `"prediction"`. May be abbreviated. Unlike *confidence intervals*, *prediction intervals* include the residual variance (sigma^2) to account for the uncertainty of predicted values. Note that prediction intervals are not available for all models, but only for models that work with `insight::get_sigma()`. For Bayesian models, when `interval = "confidence"`, predictions are based on posterior draws of the linear predictor `rstantools::posterior_epred()`. If `interval = "prediction"`, `rstantools::posterior_predict()` is called.

Details

Please see `?predict_response` for details and examples.

Value

A data frame (with `ggeffects` class attribute) with consistent data columns:

- `"x"`: the values of the first term in `terms`, used as x-position in plots.
- `"predicted"`: the predicted values of the response, used as y-position in plots.
- `"std.error"`: the standard error of the predictions. *Note that the standard errors are always on the link-scale, and not back-transformed for non-Gaussian models!*
- `"conf.low"`: the lower bound of the confidence interval for the predicted values.
- `"conf.high"`: the upper bound of the confidence interval for the predicted values.
- `"group"`: the grouping level from the second term in `terms`, used as grouping-aesthetics in plots.
- `"facet"`: the grouping level from the third term in `terms`, used to indicate facets in plots.

The estimated marginal means (or predicted values) are always on the response scale!

For proportional odds logistic regression (see `?MASS::polr`) resp. cumulative link models (e.g., see `?ordinal::clm`), an additional column `"response.level"` is returned, which indicates the grouping of predictions based on the level of the model's response.

Note that for convenience reasons, the columns for the intervals are always named `"conf.low"` and `"conf.high"`, even though for Bayesian models credible or highest posterior density intervals are returned.

There is an `as.data.frame()` method for objects of class `ggeffects`, which has an `terms_to_colnames` argument, to use the term names as column names instead of the standardized names `"x"` etc.

coffee_data *Sample dataset from a course about analysis of factorial designs*

Description

A sample data set from a course about the analysis of factorial designs, by Mattan S. Ben-Shachar. See following link for more information: https://github.com/mattansb/Analysis-of-Factorial-Designs-foR-Psychologists

The data consists of five variables from 120 observations:

- `ID`: A unique identifier for each participant
- `sex`: The participant's sex
- `time`: The time of day the participant was tested (morning, noon, or afternoon)
- `coffee`: Group indicator, whether participant drank coffee or not (`"coffee"` or `"control"`).
- `alertness`: The participant's alertness score.

Examples

```
# Attach coffee-data
data(coffee_data)
```

collapse_by_group *Collapse raw data by random effect groups*

Description

This function extracts the raw data points (i.e. the data that was used to fit the model) and "averages" (i.e. "collapses") the response variable over the levels of the grouping factor given in `collapse_by`. Only works with mixed models.

Usage

```
collapse_by_group(grid, model, collapse_by = NULL, residuals = FALSE)
```

Arguments

`grid`	A data frame representing the data grid, or an object of class `ggeffects`, as returned by `predict_response()`.
`model`	The model for which to compute partial residuals. The data grid `grid` should match to predictors in the model.
`collapse_by`	Name of the (random effects) grouping factor. Data is collapsed by the levels of this factor.
`residuals`	Logical, if `TRUE`, collapsed partial residuals instead of raw data by the levels of the grouping factor.

Value

A data frame with raw data points, averaged over the levels of the given grouping factor from the random effects. The group level of the random effect is saved in the column "random".

Examples

```
library(ggeffects)
data(efc, package = "ggeffects")
efc$e15relat <- as.factor(efc$e15relat)
efc$c161sex <- as.factor(efc$c161sex)
levels(efc$c161sex) <- c("male", "female")
model <- lme4::lmer(neg_c_7 ~ c161sex + (1 | e15relat), data = efc)
me <- predict_response(model, terms = "c161sex")
head(attributes(me)$rawdata)
collapse_by_group(me, model, "e15relat")
```

---

efc *Sample dataset from the EUROFAMCARE project*

Description

An SPSS sample data set, imported with the `sjlabelled::read_spss()` function. Consists of 28 variables from 908 observations. The data set is part of the EUROFAMCARE project, a cross-national survey on informal caregiving in Europe.

Examples

```
# Attach EFC-data
data(efc)

# Show structure
str(efc)

# show first rows
head(efc)
```

---

fish *Sample data set*

Description

A sample data set, used in tests and some examples. Useful for demonstrating count models (with or without zero-inflation component). It consists of nine variables from 250 observations.

format.ggeffects *Print and format ggeffects-objects*

Description

A generic print-method for ggeffects-objects.

Usage

```
## S3 method for class 'ggeffects'
format(
  x,
  variable_labels = FALSE,
  value_labels = FALSE,
  group_name = FALSE,
  row_header_separator = ", ",
  digits = 2,
  collapse_ci = FALSE,
  collapse_tables = FALSE,
  n,
  ...
)

## S3 method for class 'ggcomparisons'
format(x, collapse_ci = FALSE, collapse_p = FALSE, combine_levels = FALSE, ...)

## S3 method for class 'ggeffects'
print(x, group_name = TRUE, digits = 2, verbose = TRUE, ...)

## S3 method for class 'ggeffects'
print_md(x, group_name = TRUE, digits = 2, ...)

## S3 method for class 'ggeffects'
print_html(
  x,
  group_name = TRUE,
  digits = 2,
  theme = NULL,
  engine = c("tt", "gt"),
  ...
)

## S3 method for class 'ggcomparisons'
print(x, collapse_tables = FALSE, ...)

## S3 method for class 'ggcomparisons'
print_html(
```

```
  x,
  collapse_ci = FALSE,
  collapse_p = FALSE,
  theme = NULL,
  engine = c("tt", "gt"),
  ...
)

## S3 method for class 'ggcomparisons'
print_md(x, collapse_ci = FALSE, collapse_p = FALSE, theme = NULL, ...)
```

Arguments

x	An object of class ggeffects, as returned by the functions from this package.
variable_labels	Logical, if TRUE variable labels are used as column headers. If FALSE, variable names are used.
value_labels	Logical, if TRUE, value labels are used as values in the table output. If FALSE, the numeric values or factor levels are used.
group_name	Logical, if TRUE, the name of further focal terms are used in the sub-headings of the table. If FALSE, only the values of the focal terms are used.
row_header_separator	Character, separator between the different subgroups in the table output.
digits	Number of digits to print.
collapse_ci	Logical, if TRUE, the columns with predicted values and confidence intervals are collapsed into one column, e.g. Predicted (95% CI).
collapse_tables	Logical, if TRUE, all tables are combined into one. The tables are not split by further focal terms, but rather are added as columns. Only works when there is more than one focal term.
n	Number of rows to print per subgroup. If NULL, a default number of rows is printed, depending on the number of subgroups.
...	Further arguments passed down to format.ggeffects(), some of them are also passed down further to insight::format_table() or insight::format_value().
collapse_p	Logical, if TRUE, the columns with predicted values and p-values are collapsed into one column, where significant p-values are indicated as asterisks.
combine_levels	Logical, if TRUE, the levels of the first comparison of each focal term against the second are combined into one column. This is useful when comparing multiple focal terms, e.g. education = low-high and gender = male-female are combined into first = low-male and second = high-female.
verbose	Toggle messages.
theme	The theme to apply to the table. One of "grid", "striped", "bootstrap", or "darklines".
engine	The engine to use for printing. One of "tt" (default) or "gt". "tt" uses the *tinytable* package, "gt" uses the *gt* package.

Value

`format()` return a formatted data frame, `print()` prints a formatted data frame to the console. `print_html()` returns a `tinytable` object by default (unless changed with `engine = "gt"`), which is printed as HTML, markdown or LaTeX table (depending on the context from which `print_html()` is called, see `tinytable::tt()` for details).

Global Options to Customize Tables when Printing

The `verbose` argument can be used to display or silence messages and warnings. Furthermore, `options()` can be used to set defaults for the `print()` and `print_html()` method. The following options are available, which can simply be run in the console:

- `ggeffects_ci_brackets`: Define a character vector of length two, indicating the opening and closing parentheses that encompass the confidence intervals values, e.g. `options(ggeffects_ci_brackets = c("[", "]"))`.
- `ggeffects_collapse_ci`: Logical, if `TRUE`, the columns with predicted values (or contrasts) and confidence intervals are collapsed into one column, e.g. `options(ggeffects_collapse_ci = TRUE)`.
- `ggeffects_collapse_p`: Logical, if `TRUE`, the columns with predicted values (or contrasts) and p-values are collapsed into one column, e.g. `options(ggeffects_collapse_p = TRUE)`. Note that p-values are replaced by asterisk-symbols (stars) or empty strings when `ggeffects_collapse_p = TRUE`, depending on the significance level.
- `ggeffects_collapse_tables`: Logical, if `TRUE`, multiple tables for subgroups are combined into one table. Only works when there is more than one focal term, e.g. `options(ggeffects_collapse_tables = TRUE)`.
- `ggeffects_output_format`: String, either `"text"`, `"markdown"` or `"html"`. Defines the default output format from `predict_response()`. If `"html"`, a formatted HTML table is created and printed to the view pane. `"markdown"` creates a markdown-formatted table inside Rmarkdown documents, and prints a text-format table to the console when used interactively. If `"text"` or `NULL`, a formatted table is printed to the console, e.g. `options(ggeffects_output_format = "html")`.
- `ggeffects_html_engine`: String, either `"tt"` or `"gt"`. Defines the default engine to use for printing HTML tables. If `"tt"`, the *tinytable* package is used, if `"gt"`, the *gt* package is used, e.g. `options(ggeffects_html_engine = "gt")`.

Use `options(<option_name> = NULL)` to remove the option.

Examples

```
data(efc, package = "ggeffects")
fit <- lm(barthtot ~ c12hour + e42dep, data = efc)

# default print
predict_response(fit, "e42dep")

# surround CI values with parentheses
print(predict_response(fit, "e42dep"), ci_brackets = c("(", ")"))
# you can also use `options(ggeffects_ci_brackets = c("[", "]"))`
```

```
# to set this globally

# collapse CI columns into column with predicted values
print(predict_response(fit, "e42dep"), collapse_ci = TRUE)

# include value labels
print(predict_response(fit, "e42dep"), value_labels = TRUE)

# include variable labels in column headers
print(predict_response(fit, "e42dep"), variable_labels = TRUE)

# include value labels and variable labels
print(predict_response(fit, "e42dep"), variable_labels = TRUE, value_labels = TRUE)

data(iris)
m <- lm(Sepal.Length ~ Species * Petal.Length, data = iris)

# default print with subgroups
predict_response(m, c("Petal.Length", "Species"))

# omit name of grouping variable in subgroup table headers
print(predict_response(m, c("Petal.Length", "Species")), group_name = FALSE)

# collapse tables into one
print(predict_response(m, c("Petal.Length", "Species")), collapse_tables = TRUE, n = 3)

# increase number of digits
print(predict_response(fit, "e42dep"), digits = 5)
```

`get_predictions` *S3-class definition for the ggeffects package*

Description

`get_predictions()` is the core function to return adjusted predictions for a model, when calling `ggpredict()` or `predict_response()` with `margin = "mean_reference"` (the default option for `margin`). Basically, the input contains the model object and a data grid that is typically used for the `newdata` argument of the `predict()` method. `get_predictions()` can be used as S3-method for own classes, to add support for new models in **ggeffects** and is only relevant for package developers.

There are no S3-class definitions for `ggemmeans()` or `ggaverage()`, because these functions simply call methods from the **emmeans** or **marginaleffects** packages. Hence, methods should be written for those packages, too, if a model-object should work with `ggemmeans()` or `ggaverage()`.

Usage

```
get_predictions(model, ...)

## Default S3 method:
```

```
get_predictions(
  model,
  data_grid = NULL,
  terms = NULL,
  ci_level = 0.95,
  type = NULL,
  typical = NULL,
  vcov = NULL,
  vcov_args = NULL,
  condition = NULL,
  interval = "confidence",
  bias_correction = FALSE,
  link_inverse = insight::link_inverse(model),
  model_info = NULL,
  verbose = TRUE,
  ...
)
```

Arguments

`model`, `terms`, `ci_level`, `type`, `typical`, `vcov`, `vcov_args`, `condition`, `interval`, `bias_correction`, `verbose`
: Arguments from the call to `predict_response()` that are passed down to `get_predictions()`. Note that `bias_correction` is usally already processed in `predict_response()` and thus doesn't need further handling in `get_predictions()`, unless you need to re-calculate the link-inverse-function (argument `link_inverse`) inside the `get_predictions()` method.

`...`
: Further arguments, passed to `predict()` or other methods used in `get_predictions()`.

`data_grid`
: A data frame containing the data grid (or reference grid) with all relevant values of predictors for which the adjusted predictions should be made. Typically the data frame that is passed to the `newdata` argument in `predict()`. A data grid can be created with functions like `data_grid()` or `insight::get_datagrid()`.

`link_inverse`
: The model's family link-inverse function. Can be retrieved using `insight::link_inverse()`.

`model_info`
: An object returned by `insight::model_info()`.

Details

Adding support for **ggeffects** is quite easy. The user-level function is `predict_response()`, which either calls `ggpredict()`, `ggemmeans()` or `ggaverage()`. These function, in turn, call `predict()`, `emmeans::emmeans()` or `marginaleffects::avg_predictions()`. Following needs to be done to add support for new model classes:

- **emmeans**: if your model is supported by emmeans, it is automatically supported by `ggemmeans()`. Thus, you need to add the corresponding methods to your package so that your model class is supported by **emmeans.
- **marginaleffects**: similar to **emmeans**, if your package is supported by the **marginaleffects** package, it works with `ggaverage()`.

- **predict**: in order to make your model class work with ggpredict(), you need to add a get_predictions() method. The here documented arguments are *all* passed from predict_response() to get_predictions(), no matter if they are required to calculate predictions or not. Thus, it is not necessary to process all of those arguments, but they can be used to modulate certain settings when calculating predictions. Note that if your method does not define all mentioned arguments, these are still passed via ... - make sure that further methods in your get_predictions() method still work when they process the .... It is important that the function returns a data frame with a specific structure, namely the data grid and the columns predicted, conf.low, and conf.high. Predictions and intervals should be on the response scale.

A simple example for an own class-implementation for Gaussian-alike models could look like this:

```
get_predictions.own_class <- function(model, data_grid, ci_level = 0.95, ...) {
  predictions <- predict(
    model,
    newdata = data_grid,
    type = "response",
    se.fit = !is.na(ci_level),
    ...
  )

  # do we have standard errors?
  if (is.na(ci_level)) {
    # copy predictions
    data_grid$predicted <- as.vector(predictions)
  } else {
    # copy predictions
    data_grid$predicted <- predictions$fit

    # calculate CI
    data_grid$conf.low <- predictions$fit - qnorm(0.975) * predictions$se.fit
    data_grid$conf.high <- predictions$fit + qnorm(0.975) * predictions$se.fit

    # optional: copy standard errors
    attr(data_grid, "std.error") <- predictions$se.fit
  }

  data_grid
}
```

A simple example for an own class-implementation for non-Gaussian-alike models could look like this (note the use of the link-inverse function link_inverse(), which is passed to the link_inverse argument):

```
get_predictions.own_class <- function(model,
                                      data_grid,
                                      ci_level = 0.95,
                                    link_inverse = insight::link_inverse(model),
```

```
                                        ...) {
  predictions <- predict(
    model,
    newdata = data_grid,
    type = "link", # for non-Gaussian, return on link-scale
    se.fit = !is.na(ci_level),
    ...
  )

  # do we have standard errors?
  if (is.na(ci_level)) {
    # copy predictions
    data_grid$predicted <- link_inverse(as.vector(predictions))
  } else {
    # copy predictions, use link-inverse to back-transform
    data_grid$predicted <- link_inverse(predictions$fit)

    # calculate CI
    data_grid$conf.low <- link_inverse(
      predictions$fit - qnorm(0.975) * predictions$se.fit
    )
    data_grid$conf.high <- link_inverse(
      predictions$fit + qnorm(0.975) * predictions$se.fit
    )

    # optional: copy standard errors
    attr(data_grid, "std.error") <- predictions$se.fit
  }

  data_grid
}
```

Value

A data frame that contains

- the data grid (from the argument `data_grid`)
- the columns `predicted`, `conf.low`, and `conf.high`
- optionally, the attribute `"std.error"` with the standard errors.

Note that predictions and confidence intervals should already be transformed to the *response* scale (e.g., by using `insight::link_inverse()`). The *standard errors* are always on the link scale (not transformed).

If values are not available (for example, confidence intervals), set their value to `NA`.

get_title *Get titles and labels from data*

Description

Get variable and value labels from ggeffects-objects. `predict_response()` saves information on variable names and value labels as additional attributes in the returned data frame. This is especially helpful for labelled data (see **sjlabelled**), since these labels can be used to set axis labels and titles.

Usage

```
get_title(x, case = NULL)

get_x_title(x, case = NULL)

get_y_title(x, case = NULL)

get_legend_title(x, case = NULL)

get_legend_labels(x, case = NULL)

get_x_labels(x, case = NULL)

get_complete_df(x, case = NULL)
```

Arguments

x	An object of class ggeffects, as returned by any ggeffects-function; for `get_complete_df()`, must be a list of ggeffects-objects.
case	Desired target case. Labels will automatically converted into the specified character case. See `?sjlabelled::convert_case` for more details on this argument.

Value

The titles or labels as character string, or NULL, if variables had no labels; `get_complete_df()` returns the input list x as single data frame, where the grouping variable indicates the predicted values for each term.

Examples

```
library(ggeffects)
library(ggplot2)
data(efc, package = "ggeffects")
efc$c172code <- datawizard::to_factor(efc$c172code)
fit <- lm(barthtot ~ c12hour + neg_c_7 + c161sex + c172code, data = efc)

mydf <- predict_response(fit, terms = c("c12hour", "c161sex", "c172code"))
```

```
ggplot(mydf, aes(x = x, y = predicted, colour = group)) +
  stat_smooth(method = "lm") +
  facet_wrap(~facet, ncol = 2) +
  labs(
    x = get_x_title(mydf),
    y = get_y_title(mydf),
    colour = get_legend_title(mydf)
  )

# adjusted predictions, a list of data frames (one data frame per term)
eff <- ggeffect(fit)
eff
get_complete_df(eff)

# adjusted predictions for education only, and get x-axis-labels
mydat <- eff[["c172code"]]
ggplot(mydat, aes(x = x, y = predicted, group = group)) +
  stat_summary(fun = sum, geom = "line") +
  scale_x_discrete(labels = get_x_labels(mydat))
```

`install_latest` *Update latest ggeffects-version from R-universe (GitHub) or CRAN*

Description

This function can be used to install the latest package version of *ggeffects*, either the development version (from R-universe/GitHub) or the current version from CRAN.

Usage

```
install_latest(
  source = c("development", "cran"),
  force = FALSE,
  verbose = TRUE
)
```

Arguments

`source`	Character. Either `"development"` or `"cran"`. If `"cran"`, *ggeffects* will be installed from the default CRAN mirror returned by `getOption("repos")['CRAN']`. If `"development"` (the default), *ggeffects* is installed from the r-universe repository (https://strengejacke.r-universe.dev/).
`force`	Logical, if `FALSE`, the update will only be installed if a newer version is available. Use `force=TRUE` to force installation, even if the version number for the locally installed package is identical to the latest development-version. Only applies when `source="development"`.
`verbose`	Toggle messages.

Value

Invisible NULL.

Examples

```
# install latest development-version of ggeffects from the
# r-universe repository
install_latest()
```

---

johnson_neyman *Spotlight-analysis: Create Johnson-Neyman confidence intervals and plots*

Description

Function conduct a spotlight-analysis to create so-called Johnson-Neyman intervals. The plot() method can be used to visualize the results of the Johnson-Neyman test.

Usage

```
johnson_neyman(x, precision = 500, p_adjust = NULL, ...)

spotlight_analysis(x, precision = 500, p_adjust = NULL, ...)

## S3 method for class 'ggjohnson_neyman'
plot(
  x,
  colors = c("#f44336", "#2196F3"),
  show_association = TRUE,
  show_rug = FALSE,
  verbose = TRUE,
  ...
)
```

Arguments

x	An object of class ggeffects, as returned by the functions from this package.
precision	Number of values used for the range of the moderator variable to calculate the Johnson-Neyman interval. This argument is passed down to pretty(..., n = precision). Usually, the default value of 500 is sufficient. Increasing this value will result in a smoother plot and more accurate values for the interval bounds, but can also slightly increase the computation time.
p_adjust	Character vector, if not NULL, indicates the method to adjust p-values. See stats::p.adjust() or stats::p.adjust.methods for details. Further possible adjustment methods are "tukey" or "sidak", and for johnson_neyman(),

	"fdr" (or "bh") and "esarey" (or its short-cut "es") are available options. Some caution is necessary when adjusting p-value for multiple comparisons. See also section *P-value adjustment* below.
...	Arguments passed down to test_predictions() (and then probably further to marginaleffects::slopes()). See ?test_predictions for further details.
colors	Colors used for the plot. Must be a vector with two color values. Only used if show_association = TRUE.
show_association	Logical, if TRUE, highlights the range where values of the moderator are positively or negtatively associated with the outcome.
show_rug	Logical, if TRUE, adds a rug with raw data of the moderator variable to the plot. This helps visualizing its distribution.
verbose	Show/hide printed message for plots.

Details

The Johnson-Neyman intervals help to understand where slopes are significant in the context of interactions in regression models. Thus, the interval is only useful if the model contains at least one interaction term. The function accepts the results of a call to predict_response(). The *first* and the *last* focal term used in the terms argument of predict_response() must be numeric. The function will then test the slopes of the first focal terms against zero, for different moderator values of the last focal term. If only one numeric focal term is given, the function will create contrasts by levels of the categorical focal term. Use plot() to create a plot of the results.

To avoid misleading interpretations of the plot, we speak of "positive" and "negative" associations, respectively, and "no clear" associations (instead of "significant" or "non-significant"). This should prevent the user from considering a non-significant range of values of the moderator as "accepting the null hypothesis".

Value

A data frame including contrasts of the test_predictions() for the given interaction terms; for plot(), returns a Johnson-Neyman plot.

P-value adjustment for multiple comparisons

Note that p-value adjustment for methods supported by p.adjust() (see also p.adjust.methods), each row is considered as one set of comparisons, no matter which test was specified. That is, for instance, when test_predictions() returns eight rows of predictions (when test = NULL), and p_adjust = "bonferroni", the p-values are adjusted in the same way as if we had a test of pairwise comparisons (test = "pairwise") where eight rows of comparisons are returned. For methods "tukey" or "sidak", a rank adjustment is done based on the number of combinations of levels from the focal predictors in terms. Thus, the latter two methods may be useful for certain tests only, in particular pairwise comparisons.

For johnson_neyman(), the only available adjustment methods are "fdr" (or "bh") (*Benjamini & Hochberg (1995)*) and "esarey" (or "es") (*Esarey and Sumner 2017*). These usually return similar results. The major difference is that "fdr" can be slightly faster and more stable in edge cases, however, confidence intervals are not updated. Only the p-values are adjusted. "esarey" is slower, but confidence intervals are updated as well.

References

Bauer, D. J., & Curran, P. J. (2005). Probing interactions in fixed and multilevel regression: Inferential and graphical techniques. Multivariate Behavioral Research, 40(3), 373-400. doi: 10.1207/s15327906mbr4003_5

Esarey, J., & Sumner, J. L. (2017). Marginal effects in interaction models: Determining and controlling the false positive rate. Comparative Political Studies, 1–33. Advance online publication. doi: 10.1177/0010414017730080

Johnson, P.O. & Fay, L.C. (1950). The Johnson-Neyman technique, its theory and application. Psychometrika, 15, 349-367. doi: 10.1007/BF02288864

McCabe CJ, Kim DS, King KM. Improving Present Practices in the Visual Display of Interactions. Advances in Methods and Practices in Psychological Science. 2018;1(2):147-165. doi:10.1177/2515245917746792

Spiller, S. A., Fitzsimons, G. J., Lynch, J. G., & McClelland, G. H. (2013). Spotlights, Floodlights, and the Magic Number Zero: Simple Effects Tests in Moderated Regression. Journal of Marketing Research, 50(2), 277–288. doi:10.1509/jmr.12.0420

Examples

```
## Not run:
data(efc, package = "ggeffects")
efc$c172code <- as.factor(efc$c172code)
m <- lm(neg_c_7 ~ c12hour * barthtot * c172code, data = efc)

pr <- predict_response(m, c("c12hour", "barthtot"))
johnson_neyman(pr)
plot(johnson_neyman(pr))

pr <- predict_response(m, c("c12hour", "c172code", "barthtot"))
johnson_neyman(pr)
plot(johnson_neyman(pr))

# robust standard errors
if (requireNamespace("sandwich")) {
  johnson_neyman(pr, vcov = sandwich::vcovHC)
}

## End(Not run)
```

---

lung2 *Sample data set*

Description

A sample data set, used in tests and examples for survival models. This dataset is originally included in the **survival** package, but for convenience reasons it is also available in this package.

new_data *Create a data frame from all combinations of predictor values*

Description

Create a data frame for the "newdata"-argument that contains all combinations of values from the terms in questions. Similar to `expand.grid()`. The `terms`-argument accepts all shortcuts for representative values as in `predict_response()`.

Usage

```
new_data(model, terms, typical = "mean", condition = NULL, ...)

data_grid(model, terms, typical = "mean", condition = NULL, ...)
```

Arguments

`model`	A fitted model object.
`terms`	Character vector with the names of those terms from `model` for which all combinations of values should be created. This argument works in the same way as the `terms` argument in `predict_response()`. See also this vignette.
`typical`	Character vector, naming the function to be applied to the covariates (non-focal terms) over which the effect is "averaged". The default is `"mean"`. Can be `"mean"`, `"weighted.mean"`, `"median"`, `"mode"` or `"zero"`, which call the corresponding R functions (except `"mode"`, which calls an internal function to compute the most common value); `"zero"` simply returns 0. By default, if the covariate is a factor, only `"mode"` is applicable; for all other values (including the default, `"mean"`) the reference level is returned. For character vectors, only the mode is returned. You can use a named vector to apply different functions to integer, numeric and categorical covariates, e.g. `typical = c(numeric = "median", factor = "mode")`. If `typical` is `"weighted.mean"`, weights from the model are used. If no weights are available, the function falls back to `"mean"`. **Note** that this argument is ignored for `predict_response()`, because the `margin` argument takes care of this.
`condition`	Named character vector, which indicates covariates that should be held constant at specific values. Unlike `typical`, which applies a function to the covariates to determine the value that is used to hold these covariates constant, `condition` can be used to define exact values, for instance `condition = c(covariate1 = 20, covariate2 = 5)`. See 'Examples'.
`...`	Currently not used.

Value

A data frame containing one row for each combination of values of the supplied variables.

Examples

```
data(efc, package = "ggeffects")
fit <- lm(barthtot ~ c12hour + neg_c_7 + c161sex + c172code, data = efc)
new_data(fit, c("c12hour [meansd]", "c161sex"))

nd <- new_data(fit, c("c12hour [meansd]", "c161sex"))
pr <- predict(fit, type = "response", newdata = nd)
nd$predicted <- pr
nd

# compare to
predict_response(fit, c("c12hour [meansd]", "c161sex"))
```

---

plot *Plot ggeffects-objects*

Description

plot is a generic plot-method for ggeffects-objects. ggeffects_palette() returns show_palettes()

Usage

```
## S3 method for class 'ggeffects'
plot(
  x,
  show_ci = TRUE,
  ci_style = c("ribbon", "errorbar", "dash", "dot"),
  show_data = FALSE,
  show_residuals = FALSE,
  show_residuals_line = FALSE,
  data_labels = FALSE,
  limit_range = FALSE,
  collapse_group = FALSE,
  show_legend = TRUE,
  show_title = TRUE,
  show_x_title = TRUE,
  show_y_title = TRUE,
  case = NULL,
  colors = NULL,
  alpha = 0.15,
  dot_size = NULL,
  dot_alpha = 0.35,
  dot_shape = NULL,
  line_size = NULL,
  jitter = NULL,
  dodge = 0.25,
```

```
  use_theme = TRUE,
  log_y = FALSE,
  connect_lines = FALSE,
  facets,
  grid,
  one_plot = TRUE,
  n_rows = NULL,
  verbose = TRUE,
  ...
)

theme_ggeffects(base_size = 11, base_family = "")

ggeffects_palette(palette = "metro", n = NULL)

show_palettes()
```

Arguments

x	An object of class ggeffects, as returned by the functions from this package.
show_ci	Logical, if TRUE, confidence bands (for continuous variables at x-axis) resp. error bars (for factors at x-axis) are plotted.
ci_style	Character vector, indicating the style of the confidence bands. May be either "ribbon", "errorbar", "dash" or "dot", to plot a ribbon, error bars, or dashed or dotted lines as confidence bands.
show_data	Logical, if TRUE, a layer with raw data from response by predictor on the x-axis, plotted as point-geoms, is added to the plot. Note that if the model has a transformed response variable, and the predicted values are *not* back-transformed (i.e. if back_transform = FALSE), the raw data points are plotted on the transformed scale, i.e. same scale as the predictions.
show_residuals	Logical, if TRUE, a layer with partial residuals is added to the plot. See vignette Effect Displays with Partial Residuals. from **effects** for more details on partial residual plots.
show_residuals_line	Logical, if TRUE, a loess-fit line is added to the partial residuals plot. Only applies if residuals is TRUE.
data_labels	Logical, if TRUE and row names in data are available, data points will be labelled by their related row name.
limit_range	Logical, if TRUE, limits the range of the prediction bands to the range of the data.
collapse_group	For mixed effects models, name of the grouping variable of random effects. If collapse_group = TRUE, data points "collapsed" by the first random effect groups are added to the plot. Else, if collapse_group is a name of a group factor, data is collapsed by that specific random effect. See collapse_by_group() for further details.
show_legend	Logical, shows or hides the plot legend.
show_title	Logical, shows or hides the plot title-

show_x_title — Logical, shows or hides the plot title for the x-axis.

show_y_title — Logical, shows or hides the plot title for the y-axis.

case — Desired target case. Labels will automatically converted into the specified character case. See ?sjlabelled::convert_case for more details on this argument.

colors — Character vector with color values in hex-format, valid color value names (see demo("colors")) or a name of a ggeffects-color-palette (see ggeffects_palette()).

Following options are valid for colors:

- If not specified, the color brewer palette "Set1" will be used.
- If "gs", a greyscale will be used.
- If "bw", the plot is black/white and uses different line types to distinguish groups.
- There are some pre-defined color-palettes in this package that can be used, e.g. colors = "metro". See show_palettes() to show all available palettes.
- Else specify own color values or names as vector (e.g. colors = c("#f00000", "#00ff00")).

alpha — Alpha value for the confidence bands.

dot_size — Numeric, size of the point geoms.

dot_alpha — Alpha value for data points, when show_data = TRUE.

dot_shape — Shape of data points, when show_data = TRUE.

line_size — Numeric, size of the line geoms.

jitter — Numeric, between 0 and 1. If not NULL and show_data = TRUE, adds a small amount of random variation to the location of data points dots, to avoid overplotting. Hence the points don't reflect exact values in the data. May also be a numeric vector of length two, to add different horizontal and vertical jittering. For binary outcomes, raw data is not jittered by default to avoid that data points exceed the axis limits.

dodge — Value for offsetting or shifting error bars, to avoid overlapping. Only applies, if a factor is plotted at the x-axis (in such cases, the confidence bands are replaced by error bars automatically), or if ci_style = "errorbars".

use_theme — Logical, if TRUE, a slightly tweaked version of ggplot's minimal-theme, theme_ggeffects(), is applied to the plot. If FALSE, no theme-modifications are applied.

log_y — Logical, if TRUE, the y-axis scale is log-transformed. This might be useful for binomial models with predicted probabilities on the y-axis.

connect_lines — Logical, if TRUE and plot has point-geoms with error bars (this is usually the case when the x-axis is discrete), points of same groups will be connected with a line.

facets, grid — Logical, defaults to TRUE if x has a column named facet, and defaults to FALSE if x has no such column. Set facets = TRUE to wrap the plot into facets even for grouping variables (see 'Examples'). grid is an alias for facets.

one_plot — Logical, if TRUE and x has a grid column (i.e. when five terms were used), a single, integrated plot is produced.

n_rows	Number of rows to align plots. By default, all plots are aligned in one row. For facets, or multiple panels, plots can also be aligned in multiiple rows, to avoid that plots are too small.
verbose	Logical, toggle warnings and messages.
...	Further arguments passed down to ggplot::scale_y*(), to control the appearance of the y-axis.
base_size	Base font size.
base_family	Base font family.
palette	Name of a pre-defined color-palette as string. See show_palettes() to show all available palettes. Use NULL to return a list with names and color-codes of all avaibale palettes.
n	Number of color-codes from the palette that should be returned.

Details

For proportional odds logistic regression (see ?MASS::polr) or cumulative link models in general, plots are automatically facetted by response.level, which indicates the grouping of predictions based on the level of the model's response.

Value

A ggplot2-object.

Partial Residuals

For **generalized linear models** (glms), residualized scores are computed as inv.link(link(Y) + r) where Y are the predicted values on the response scale, and r are the *working* residuals.

For (generalized) linear **mixed models**, the random effect are also partialled out.

Note

Load library(ggplot2) and use theme_set(theme_ggeffects()) to set the **ggeffects**-theme as default plotting theme. You can then use further plot-modifiers, e.g. from **sjPlot**, like legend_style() or font_size() without losing the theme-modifications.

There are pre-defined colour palettes in this package. Use show_palettes() to show all available colour palettes as plot, or ggeffects_palette(palette = NULL) to show the color codes.

Examples

```
library(sjlabelled)
data(efc)
efc$c172code <- as_label(efc$c172code)
fit <- lm(barthtot ~ c12hour + neg_c_7 + c161sex + c172code, data = efc)

dat <- predict_response(fit, terms = "c12hour")
plot(dat)
```

```
# facet by group, use pre-defined color palette
dat <- predict_response(fit, terms = c("c12hour", "c172code"))
plot(dat, facet = TRUE, colors = "hero")

# don't use facets, b/w figure, w/o confidence bands
dat <- predict_response(fit, terms = c("c12hour", "c172code"))
plot(dat, colors = "bw", show_ci = FALSE)

# factor at x axis, plot exact data points and error bars
dat <- predict_response(fit, terms = c("c172code", "c161sex"))
plot(dat)

# for three variables, automatic facetting
dat <- predict_response(fit, terms = c("c12hour", "c172code", "c161sex"))
plot(dat)


# show color codes of specific palette
ggeffects_palette("okabe-ito")

# show all color palettes
show_palettes()
```

`pool_comparisons` *Pool contrasts and comparisons from* `test_predictions()`

Description

This function "pools" (i.e. combines) multiple ggcomparisons objects, returned by `test_predictions()`, in a similar fashion as `mice::pool()`.

Usage

```
pool_comparisons(x, ...)
```

Arguments

x	A list of ggcomparisons objects, as returned by `test_predictions()`.
...	Currently not used.

Details

Averaging of parameters follows Rubin's rules (*Rubin, 1987, p. 76*).

Value

A data frame with pooled comparisons or contrasts of predictions.

References

Rubin, D.B. (1987). Multiple Imputation for Nonresponse in Surveys. New York: John Wiley and Sons.

Examples

```
data("nhanes2", package = "mice")
imp <- mice::mice(nhanes2, printFlag = FALSE)
comparisons <- lapply(1:5, function(i) {
  m <- lm(bmi ~ age + hyp + chl, data = mice::complete(imp, action = i))
  test_predictions(m, "age")
})
pool_comparisons(comparisons)
```

---

pool_predictions *Pool Predictions or Estimated Marginal Means*

Description

This function "pools" (i.e. combines) multiple ggeffects objects, in a similar fashion as mice::pool().

Usage

```
pool_predictions(x, ...)
```

Arguments

x	A list of ggeffects objects, as returned by predict_response().
...	Currently not used.

Details

Averaging of parameters follows Rubin's rules (*Rubin, 1987, p. 76*). Pooling is applied to the predicted values on the scale of the *linear predictor*, not on the response scale, in order to have accurate pooled estimates and standard errors. The final pooled predicted values are then transformed to the response scale, using insight::link_inverse().

Value

A data frame with pooled predictions.

References

Rubin, D.B. (1987). Multiple Imputation for Nonresponse in Surveys. New York: John Wiley and Sons.

Examples

```
# example for multiple imputed datasets
data("nhanes2", package = "mice")
imp <- mice::mice(nhanes2, printFlag = FALSE)
predictions <- lapply(1:5, function(i) {
  m <- lm(bmi ~ age + hyp + chl, data = mice::complete(imp, action = i))
  predict_response(m, "age")
})
pool_predictions(predictions)
```

---

predict_response *Adjusted predictions and estimated marginal means from regression models*

Description

After fitting a model, it is useful generate model-based estimates (expected values, or *adjusted predictions*) of the response variable for different combinations of predictor values. Such estimates can be used to make inferences about relationships between variables.

The **ggeffects** package computes marginal means and adjusted predicted values for the response, at the margin of specific values or levels from certain model terms. The package is built around three core functions: `predict_response()` (understanding results), `test_predictions()` (importance of results) and `plot()` (communicate results).

By default, adjusted predictions or marginal means are returned on the *response* scale, which is the easiest and most intuitive scale to interpret the results. There are other options for specific models as well, e.g. with zero-inflation component (see documentation of the `type`-argument). The result is returned as structured data frame, which is nicely printed by default. `plot()` can be used to easily create figures.

The main function to calculate marginal means and adjusted predictions is `predict_response()`, which returns adjusted predictions, marginal means or averaged counterfactual predictions depending on value of the `margin`-argument.

In previous versions of **ggeffects**, the functions `ggpredict()`, `ggemmeans()`, `ggeffect()` and `ggaverage()` were used to calculate marginal means and adjusted predictions. These functions are still available, but `predict_response()` as a "wrapper" around these functions is the preferred way to calculate marginal means and adjusted predictions now.

Usage

```
predict_response(
  model,
  terms,
  margin = "mean_reference",
  ci_level = 0.95,
  type = "fixed",
  condition = NULL,
```

```
  interval = "confidence",
  back_transform = TRUE,
  vcov = NULL,
  vcov_args = NULL,
  weights = NULL,
  bias_correction = FALSE,
  verbose = TRUE,
  ...
)
```

Arguments

model	A model object.
terms	Names of those terms from model, for which predictions should be displayed (so called *focal terms*). Can be: • A character vector, specifying the names of the focal terms. This is the preferred and probably most flexible way to specify focal terms, e.g. terms = "x [40:60]", to calculate predictions for the values 40 to 60. • A list, where each element is a named vector, specifying the focal terms and their values. This is the "classical" R way to specify focal terms, e.g. list(x = 40:60). • A formula, e.g. terms = ~ x + z, which is internally converted to a character vector. This is probably the least flexible way, as you cannot specify representative values for the focal terms. • A data frame representing a "data grid" or "reference grid". Predictions are then made for all combinations of the variables in the data frame. terms at least requires one variable name. The maximum length is five terms, where the second to fifth term indicate the groups, i.e. predictions of the first term are grouped at meaningful values or levels of the remaining terms (see values_at()). It is also possible to define specific values for focal terms, at which adjusted predictions should be calculated (see details below). All remaining covariates that are not specified in terms are "marginalized", see the margin argument in ?predict_response. See also argument condition to fix non-focal terms to specific values, and argument typical for ggpredict() or ggemmeans().
margin	Character string, indicating how to marginalize over the *non-focal* predictors, i.e. those variables that are *not* specified in terms. Possible values are "mean_reference", "mean_mode", "marginalmeans" and "empirical" (or one of its aliases, "counterfactual" or "average", aka average "counterfactual" predictions). You can set a default-option for the margin argument via options(), e.g. options(ggeffects_margin = "empirical"), so you don't have to specify your preferred marginalization method each time you call predict_response(). See details in the documentation below.
ci_level	Numeric, the level of the confidence intervals. Use ci_level = NA if confidence intervals should not be calculated (for instance, due to computation time). Typically, confidence intervals are based on the returned standard errors for the predictions, assuming a t- or normal distribution (based on the model and the

available degrees of freedom, i.e. roughly `+/- 1.96 * SE`). See introduction of this vignette for more details.

`type` Character, indicating whether predictions should be conditioned on specific model components or not, or whether population or unit-level predictions are desired. Consequently, most options only apply for survival models, mixed effects models and/or models with zero-inflation (and their Bayesian counter-parts); only exception is `type = "simulate"`, which is available for some other model classes as well (which respond to `simulate()`).

Note 1: For `brmsfit`-models with zero-inflation component, there is no `type = "zero_inflated"` nor `type = "zi_random"`; predicted values for these models *always* condition on the zero-inflation part of the model. The same is true for `MixMod`-models from **GLMMadaptive** with zero-inflation component (see 'Details').

Note 2: If `margin = "empirical"`, or when calling `ggaverage()` respectively, (i.e. counterfactual predictions), the `type` argument is handled differently. It is set to `"response"` by default, but usually accepts all possible options from the `type`-argument of the model's respective `predict()` method. E.g., passing a `glm` object would allow the options `"response"`, `"link"`, and `"terms"`. For models with zero-inflation component, the below mentioned options `"fixed"`, `"zero_inflated"` and `"zi_prob"` can also be used and will be "translated" into the corresponding `type` option of the model's respective `predict()`-method.

Note 3: If `margin = "marginalmeans"`, or when calling `ggemmeans()` respectively, `type = "random"` and `type = "zi_random"` are not available, i.e. no unit-level predictions are possible.

- `"fixed"` (or `"count"`)
  Predicted values are conditioned on the fixed effects or conditional model only. For mixed models, predicted values are on the *population-level*, i.e. `re.form = NA` when calling `predict()`. For models with zero-inflation component, this type would return the predicted mean from the count component (without conditioning on the zero-inflation part).
- `"random"`
  This only applies to mixed models, and `type = "random"` does not condition on the zero-inflation component of the model. Use this for *unit-level* predictions, i.e. predicted values for each level of the random effects groups. Add the name of the related random effect term to the `terms`-argument (for more details, see this vignette).
- `"zero_inflated"` (or `"zi"`)
  Predicted values are conditioned on the fixed effects and the zero-inflation component, returning the expected value of the response (`mu*(1-p)`). For For mixed models with zero-inflation component (e.g. from package **glmmTMB**), this would return the expected response `mu*(1-p)` on the *population-level*. See 'Details'.
- `"zi_random"` (or `"zero_inflated_random"`)
  This only applies to mixed models. Predicted values are conditioned on the fixed effects and the zero-inflation component. Use this for *unit-level* predictions, i.e. predicted values for each level of the random effects groups. Add the name of the related random effect term to the `terms`-argument (for more details, see this vignette).

- `"zi_prob"`
  Returns the predicted zero-inflation probability, i.e. probability of a structural or "true" zero (see this vignette for a short introduction into zero-inflation models).
- `"simulate"`
  Predicted values and confidence resp. prediction intervals are based on simulations, i.e. calls to `simulate()`. This type of prediction takes all model uncertainty into account. Currently supported models are objects of class `lm`, `glm`, `glmmTMB`, `wbm`, `MixMod` and `merMod`. Use `nsim` to set the number of simulated draws (see `...` for details).
- `"survival"`, `"cumulative_hazard"` and `"quantile"`
  `"survival"` and `"cumulative_hazard"` apply only to coxph-objects from the **survial**-package. These options calculate the survival probability or the cumulative hazard of an event. `type = "quantile"` only applies to `survreg`-objects from package **survival**, which returns the predicted quantiles. For this option, the `p` argument is passed to `predict()`, so that quantiles for different probabilities can be calculated, e.g. `predict_response(..., type = "quantile", p = c(0.2, 0.5, 0.8))`.

When `margin = "empirical"` (or when calling `ggaverage()`), the `type` argument accepts all values from the `type`-argument of the model's respective `predict()`-method.

`condition` Named character vector, which indicates covariates that should be held constant at specific values. Unlike `typical`, which applies a function to the covariates to determine the value that is used to hold these covariates constant, `condition` can be used to define exact values, for instance `condition = c(covariate1 = 20, covariate2 = 5)`. See 'Examples'.

`interval` Type of interval calculation, can either be `"confidence"` (default) or `"prediction"`. May be abbreviated. Unlike *confidence intervals*, *prediction intervals* include the residual variance (sigma^2) to account for the uncertainty of predicted values. Note that prediction intervals are not available for all models, but only for models that work with `insight::get_sigma()`. For Bayesian models, when `interval = "confidence"`, predictions are based on posterior draws of the linear predictor `rstantools::posterior_epred()`. If `interval = "prediction"`, `rstantools::posterior_predict()` is called.

`back_transform` Logical, if `TRUE` (the default), predicted values for log-, log-log, exp, sqrt and similar transformed responses will be back-transformed to original response-scale. See `insight::find_transformation()` for more details.

`vcov` Variance-covariance matrix used to compute uncertainty estimates (e.g., for confidence intervals based on robust standard errors). This argument accepts a covariance matrix, a function which returns a covariance matrix, or a string which identifies the function to be used to compute the covariance matrix.

- A covariance matrix
- A function which returns a covariance matrix (e.g., `stats::vcov()`)
- A string which indicates the kind of uncertainty estimates to return.
  - Heteroskedasticity-consistent: `"HC"`, `"HC0"`, `"HC1"`, `"HC2"`, `"HC3"`, `"HC4"`, `"HC4m"`, `"HC5"`. See `?sandwich::vcovHC`

- Cluster-robust: "vcovCR", "CR0", "CR1", "CR1p", "CR1S", "CR2", "CR3". See ?clubSandwich::vcovCR.
- Bootstrap: "BS", "xy", "fractional", "jackknife", "residual", "wild", "mammen", "norm", "webb". See ?sandwich::vcovBS
- Other sandwich package functions: "HAC", "PC", "CL", or "PL".

If NULL, standard errors (and confidence intervals) for predictions are based on the standard errors as returned by the predict()-function. **Note** that probably not all model objects that work with predict_response() are also supported by the **sandwich** or **clubSandwich** packages.

See details in this vignette.

vcov_args

List of arguments to be passed to the function identified by the vcov argument. This function is typically supplied by the **sandwich** or **clubSandwich** packages. Please refer to their documentation (e.g., ?sandwich::vcovHAC) to see the list of available arguments. If no estimation type (argument type) is given, the default type for "HC" equals the default from the **sandwich** package; for type "CR" the default is set to "CR3". For other defaults, refer to the documentation in the **sandwich** or **clubSandwich** package.

weights

This argument is used in two different ways, depending on the margin argument.

- When margin = "empirical" (or when calling ggaverag()), weights can either be a character vector, naming the weigthing variable in the data, or a vector of weights (of same length as the number of observations in the data). This variable will be used to weight adjusted predictions.
- When margin = "marginalmeans" (or when calling ggemmeans()), weights must be a character vector and is passed to emmeans::emmeans(), specifying weights to use in averaging non-focal categorical predictors. Options are "equal", "proportional", "outer", "cells", "flat", and "show.levels". See ?emmeans::emmeans for details.

bias_correction

Logical, if TRUE, adjusts for bias-correction when back-transforming the predicted values (to the response scale) for non-Gaussian *mixed models*. Back-transforming the the population-level predictions ignores the effect of the variation around the population mean, so the result on the original data scale is biased due to *Jensen's inequality*. That means, when type = "fixed" (the default) and population level predictions are returned, it is recommended to set bias_correction = TRUE. To apply bias-correction, a valid value of sigma is required, which is extracted by default using insight::get_variance_residual(). Optionally, to provide own estimates of uncertainty, use the sigma argument. Note that bias_correction currently only applies to mixed models, where there are additive random components involved and where that bias-adjustment can be appropriate. If ggemmeans() is called, bias-correction can also be applied to GEE-models.

verbose

Toggle messages or warnings.

...

If margin is set to "mean_reference" or "mean_mode", arguments are passed down to ggpredict() (further down to predict()); for margin = "marginalmeans", further arguments passed down to ggemmeans() and thereby to emmeans::emmeans(); if margin = "empirical", further arguments are passed down to marginaleffects::avg_predictions(

If type = "simulate", ... may also be used to set the number of simulation, e.g. nsim = 500. When calling ggeffect(), further arguments passed down to effects::Effect().

Value

A data frame (with ggeffects class attribute) with consistent data columns:

- "x": the values of the first term in terms, used as x-position in plots.
- "predicted": the predicted values of the response, used as y-position in plots.
- "std.error": the standard error of the predictions. *Note that the standard errors are always on the link-scale, and not back-transformed for non-Gaussian models!*
- "conf.low": the lower bound of the confidence interval for the predicted values.
- "conf.high": the upper bound of the confidence interval for the predicted values.
- "group": the grouping level from the second term in terms, used as grouping-aesthetics in plots.
- "facet": the grouping level from the third term in terms, used to indicate facets in plots.

The estimated marginal means (or predicted values) are always on the response scale!

For proportional odds logistic regression (see ?MASS::polr) resp. cumulative link models (e.g., see ?ordinal::clm), an additional column "response.level" is returned, which indicates the grouping of predictions based on the level of the model's response.

Note that for convenience reasons, the columns for the intervals are always named "conf.low" and "conf.high", even though for Bayesian models credible or highest posterior density intervals are returned.

There is an as.data.frame() method for objects of class ggeffects, which has an terms_to_colnames argument, to use the term names as column names instead of the standardized names "x" etc.

Supported Models

A list of supported models can be found at the package website. Support for models varies by marginalization method (the margin argument), i.e. although predict_response() supports most models, some models are only supported exclusively by one of the four downstream functions (ggpredict(), ggemmeans(), ggeffect() or ggaverage()). This means that not all models work for every margin option of predict_response().

Holding covariates at constant values, or how to marginalize over the *non-focal* predictors

predict_response() is a wrapper around ggpredict(), ggemmeans() and ggaverage(). Depending on the value of the margin argument, predict_response() calls one of those functions. The margin argument indicates how to marginalize over the *non-focal* predictors, i.e. those variables that are *not* specified in terms. Possible values are:

- "mean_reference" and "mean_mode": For "mean_reference", non-focal predictors are set to their mean (numeric variables), reference level (factors), or "most common" value (mode) in case of character vectors. For "mean_mode", non-focal predictors are set to their mean (numeric variables) or mode (factors, or "most common" value in case of character vectors).

  These predictons represent a rather "theoretical" view on your data, which does not necessarily exactly reflect the characteristics of your sample. It helps answer the question, "What is the

predicted (or: expected) value of the response at meaningful values or levels of my focal terms for a 'typical' observation in my data?", where 'typical' refers to certain characteristics of the remaining predictors.

- `"marginalmeans"`: non-focal predictors are set to their mean (numeric variables) or averaged over the levels or "values" for factors and character vectors. Averaging over the factor levels of non-focal terms computes a kind of "weighted average" for the values at which these terms are hold constant. Thus, non-focal categorical terms are conditioned on "weighted averages" of their levels. There are different weighting options that can be altered using the `weights` argument.

  These predictions come closer to the sample, because all possible values and levels of the non-focal predictors are taken into account. It would answer the question, "What is the predicted (or: expected) value of the response at meaningful values or levels of my focal terms for an 'average' observation in my data?". It refers to randomly picking a subject of your sample and the result you get on average.

- `"empirical"` (or `"counterfactual"` or `"average"`): non-focal predictors are averaged over the observations in the sample. The response is predicted for each subject in the data and predicted values are then averaged across all subjects, aggregated/grouped by the focal terms. In particular, averaging is applied to *counterfactual predictions* (Dickerman and Hernan 2020). There is a more detailed description in this vignette.

  Counterfactual predictions are useful, insofar as the results can also be transferred to other contexts. It answers the question, "What is the predicted (or: expected) value of the response at meaningful values or levels of my focal terms for the 'average' observation in the population?". It does not only refer to the actual data in your sample, but also "what would be if" we had more data, or if we had data from a different population. This is where "counterfactual" refers to.

You can set a default-option for the `margin` argument via `options()`, e.g. `options(ggeffects_margin = "empirical")`, so you don't have to specify your "default" marginalization method each time you call `predict_response()`. Use `options(ggeffects_margin = NULL)` to remove that setting.

The `condition` argument can be used to fix non-focal terms to specific values.

Marginal Means and Adjusted Predictions at Specific Values

Meaningful values of focal terms can be specified via the `terms` argument. Specifying meaningful or representative values as string pattern is the preferred way in the **ggeffects** package. However, it is also possible to use a `list()` for the focal terms if prefer the "classical" R way. `terms` can also be a data (or reference) grid provided as data frame. All options are described in this vignette.

Indicating levels in square brackets allows for selecting only certain groups or values resp. value ranges. The term name and the start of the levels in brackets must be separated by a whitespace character, e.g. `terms = c("age", "education [1,3]")`. Numeric ranges, separated with colon, are also allowed: `terms = c("education", "age [30:60]")`. The stepsize for ranges can be adjusted using `by`, e.g. `terms = "age [30:60 by=5]"`.

The `terms` argument also supports the same shortcuts as the `values` argument in `values_at()`. So `terms = "age [meansd]"` would return predictions for the values one standard deviation below the mean age, the mean age and one SD above the mean age. `terms = "age [quart2]"` would calculate predictions at the value of the lower, median and upper quartile of age.

Furthermore, it is possible to specify a function name. Values for predictions will then be transformed, e.g. `terms = "income [exp]"`. This is useful when model predictors were transformed for fitting the model and should be back-transformed to the original scale for predictions. It is also possible to define own functions (see this vignette).

Instead of a function, it is also possible to define the name of a variable with specific values, e.g. to define a vector `v = c(1000, 2000, 3000)` and then use `terms = "income [v]"`.

You can take a random sample of any size with `sample=n`, e.g `terms = "income [sample=8]"`, which will sample eight values from all possible values of the variable `income`. This option is especially useful for plotting predictions at certain levels of random effects group levels, where the group factor has too many levels to be completely plotted. For more details, see this vignette.

Finally, numeric vectors for which no specific values are given, a "pretty range" is calculated (see `pretty_range()`), to avoid memory allocation problems for vectors with many unique values. If a numeric vector is specified as second or third term (i.e. if this focal term is used for "stratification"), representative values (see `values_at()`) are chosen (unless other values are specified), which are typically the mean value, as well as one standard deviation below and above the mean. If all values for a numeric vector should be used to compute predictions, you may use e.g. `terms = "age [all]"`. See also package vignettes.

To create a pretty range that should be smaller or larger than the default range (i.e. if no specific values would be given), use the `n` tag, e.g. `terms="age [n=5]"` or `terms="age [n=12]"`. Larger values for `n` return a larger range of predicted values.

Bayesian Regression Models

`predict_response()` also works with **Stan**-models from the **rstanarm** or **brms**-packages. The predicted values are the median value of all drawn posterior samples. Standard errors are the median absolute deviation of the posterior samples. The confidence intervals for Stan-models are Bayesian predictive intervals. By default, the predictions are based on `rstantools::posterior_epred()` and hence have the limitations that the uncertainty of the error term (residual variance) is not taken into account. The recommendation is to use the posterior predictive distribution (`rstantools::posterior_predict()`), i.e. setting `interval = "prediction"`.

Mixed (multilevel) Models

For mixed models, following options are possible:

- Predictions can be made on the population-level (`type = "fixed"`, the default) or for each level of the grouping variable (unit-level). If unit-level predictions are requested, you need to set `type = "random"`` and specify the grouping variable(s) in the terms‘ argument.
- Population-level predictions can be either *conditional* (predictions for a "typical" group) or *marginal* (average predictions across all groups). The default in `predict_response()` calculated *conditional* predictions. Set `margin = "empirical"` for marginal predictions.
- Prediction intervals, i.e. when `interval = "predictions"` also account for the uncertainty in the random effects.

See more details in this vignette.

Zero-Inflated and Zero-Inflated Mixed Models with brms

Models of class `brmsfit` always condition on the zero-inflation component, if the model has such a component. Hence, there is no `type = "zero_inflated"` nor `type = "zi_random"` for `brmsfit`-models, because predictions are based on draws of the posterior distribution, which already account for the zero-inflation part of the model.

Zero-Inflated and Zero-Inflated Mixed Models with glmmTMB

If `model` is of class `glmmTMB`, `hurdle`, `zeroinfl` or `zerotrunc`, and `margin` is *not* set to `"empirical"`, simulations from a multivariate normal distribution (see `?MASS::mvrnorm`) are drawn to calculate `mu*(1-p)`. Confidence intervals are then based on quantiles of these results. For `type = "zi_random"`, prediction intervals also take the uncertainty in the random-effect paramters into account (see also *Brooks et al. 2017*, pp.391-392 for details).

An alternative for models fitted with **glmmTMB** that take all model uncertainties into account are simulations based on `simulate()`, which is used when `type = "simulate"` (see *Brooks et al. 2017*, pp.392-393 for details).

Finally, if `margin = "empirical"`, the returned predictions are already conditioned on the zero-inflation part (and possible random effects) of the model, thus these are most comparable to the `type = "simulate"` option. In other words, if all model components should be taken into account for predictions, you should consider using `margin = "empirical"`.

MixMod-models from GLMMadaptive

Predicted values for the fixed effects component (`type = "fixed"` or `type = "zero_inflated"`) are based on `predict(..., type = "mean_subject")`, while predicted values for random effects components (`type = "random"` or `type = "zi_random"`) are calculated with `predict(..., type = "subject_specific")` (see `?GLMMadaptive::predict.MixMod` for details). The latter option requires the response variable to be defined in the `newdata`-argument of `predict()`, which will be set to its typical value (see `values_at()`).

Multinomial Models

`polr`, `clm` models, or more generally speaking, models with ordinal or multinominal outcomes, have an additional column `response.level`, which indicates with which level of the response variable the predicted values are associated.

Averaged model predictions (package MuMIn)

For averaged model predictions, i.e. when the input model is an object of class `"averaging"` (`MuMIn::model.avg()`), predictions are made with the full averaged coefficients.

Note

Printing Results

The `print()` method gives a clean output (especially for predictions by groups), and indicates at which values covariates were held constant. Furthermore, the `print()` method has several arguments to customize the output. See this vignette for details.

Limitations

The support for some models, for example from package **MCMCglmm**, is not fully tested and may fail for certain models. If you encounter any errors, please file an issue at Github.

References

- Brooks ME, Kristensen K, Benthem KJ van, Magnusson A, Berg CW, Nielsen A, et al. glmmTMB Balances Speed and Flexibility Among Packages for Zero-inflated Generalized Linear Mixed Modeling. The R Journal. 2017;9: 378-400.
- Johnson PC. 2014. Extension of Nakagawa & Schielzeth's R2GLMM to random slopes models. Methods Ecol Evol, 5: 944-946.
- Dickerman BA, Hernan, MA. Counterfactual prediction is not only for causal inference. Eur J Epidemiol 35, 615–617 (2020).

Examples

```
library(sjlabelled)
data(efc)
fit <- lm(barthtot ~ c12hour + neg_c_7 + c161sex + c172code, data = efc)

predict_response(fit, terms = "c12hour")
predict_response(fit, terms = c("c12hour", "c172code"))
# more compact table layout for printing
out <- predict_response(fit, terms = c("c12hour", "c172code", "c161sex"))
print(out, collapse_table = TRUE)

# specified as formula
predict_response(fit, terms = ~ c12hour + c172code + c161sex)

# only range of 40 to 60 for variable 'c12hour'
predict_response(fit, terms = "c12hour [40:60]")

# terms as named list
predict_response(fit, terms = list(c12hour = 40:60))

# covariate "neg_c_7" is held constant at a value of 11.84 (its mean value).
# To use a different value, use "condition"
predict_response(fit, terms = "c12hour [40:60]", condition = c(neg_c_7 = 20))

# to plot ggeffects-objects, you can use the 'plot()'-function.
# the following examples show how to build your ggplot by hand.


# plot predicted values, remaining covariates held constant
library(ggplot2)
mydf <- predict_response(fit, terms = "c12hour")
ggplot(mydf, aes(x, predicted)) +
  geom_line() +
  geom_ribbon(aes(ymin = conf.low, ymax = conf.high), alpha = 0.1)

# three variables, so we can use facets and groups
mydf <- predict_response(fit, terms = c("c12hour", "c161sex", "c172code"))
```

```
ggplot(mydf, aes(x = x, y = predicted, colour = group)) +
  stat_smooth(method = "lm", se = FALSE) +
  facet_wrap(~facet, ncol = 2)

# select specific levels for grouping terms
mydf <- predict_response(fit, terms = c("c12hour", "c172code [1,3]", "c161sex"))
ggplot(mydf, aes(x = x, y = predicted, colour = group)) +
  stat_smooth(method = "lm", se = FALSE) +
  facet_wrap(~facet) +
  labs(
    y = get_y_title(mydf),
    x = get_x_title(mydf),
    colour = get_legend_title(mydf)
  )

# level indication also works for factors with non-numeric levels
# and in combination with numeric levels for other variables
data(efc)
efc$c172code <- sjlabelled::as_label(efc$c172code)
fit <- lm(barthtot ~ c12hour + neg_c_7 + c161sex + c172code, data = efc)
predict_response(fit, terms = c(
  "c12hour",
  "c172code [low level of education, high level of education]",
  "c161sex [1]"
))

# when "terms" is a named list
predict_response(fit, terms = list(
  c12hour = seq(0, 170, 30),
  c172code = c("low level of education", "high level of education"),
  c161sex = 1
))

# use categorical value on x-axis, use axis-labels, add error bars
dat <- predict_response(fit, terms = c("c172code", "c161sex"))
ggplot(dat, aes(x, predicted, colour = group)) +
  geom_point(position = position_dodge(0.1)) +
  geom_errorbar(
    aes(ymin = conf.low, ymax = conf.high),
    position = position_dodge(0.1)
  ) +
  scale_x_discrete(breaks = 1:3, labels = get_x_labels(dat))

# 3-way-interaction with 2 continuous variables
data(efc)
# make categorical
efc$c161sex <- as_factor(efc$c161sex)
fit <- lm(neg_c_7 ~ c12hour * barthtot * c161sex, data = efc)
# select only levels 30, 50 and 70 from continuous variable Barthel-Index
dat <- predict_response(fit, terms = c("c12hour", "barthtot [30,50,70]", "c161sex"))
ggplot(dat, aes(x = x, y = predicted, colour = group)) +
  stat_smooth(method = "lm", se = FALSE, fullrange = TRUE) +
  facet_wrap(~facet) +
```

```
  labs(
    colour = get_legend_title(dat),
    x = get_x_title(dat),
    y = get_y_title(dat),
    title = get_title(dat)
  )

# or with ggeffects' plot-method
plot(dat, show_ci = FALSE)


# predictions for polynomial terms
data(efc)
fit <- glm(
  tot_sc_e ~ c12hour + e42dep + e17age + I(e17age^2) + I(e17age^3),
  data = efc,
  family = poisson()
)
predict_response(fit, terms = "e17age")
```

`pretty_range` *Create a pretty sequence over a range of a vector*

Description

Creates an evenly spaced, pretty sequence of numbers for a range of a vector.

Usage

```
pretty_range(x, n = NULL, length = NULL)
```

Arguments

`x`	A numeric vector.
`n`	Numeric value, indicating the size of how many values are used to create a pretty sequence. If `x` has a large value range (> 100), `n` could be something between 1 to 5. If `x` has a rather small amount of unique values, `n` could be something between 10 to 20. If `n = NULL`, `pretty_range()` automatically tries to find a pretty sequence.
`length`	Integer value, as alternative to `n`, defines the number of intervals to be returned.

Value

A numeric vector with a range corresponding to the minimum and maximum values of `x`. If `x` is missing, a function, pre-programmed with `n` and `length` is returned. See examples.

Examples

```
data(iris)
# pretty range for vectors with decimal points
pretty_range(iris$Petal.Length)

# pretty range for large range, increasing by 50
pretty_range(1:1000)

# increasing by 20
pretty_range(1:1000, n = 7)

# return 10 intervals
pretty_range(1:1000, length = 10)

# same result
pretty_range(1:1000, n = 2.5)

# function factory
range_n_5 <- pretty_range(n = 5)
range_n_5(1:1000)
```

---

residualize_over_grid *Compute partial residuals from a data grid*

Description

This function computes partial residuals based on a data grid, where the data grid is usually a data frame from all combinations of factor variables or certain values of numeric vectors. This data grid is usually used as `newdata` argument in `predict()`, and can be created with `new_data()`.

Usage

```
residualize_over_grid(grid, model, ...)

## S3 method for class 'data.frame'
residualize_over_grid(grid, model, predictor_name, ...)

## S3 method for class 'ggeffects'
residualize_over_grid(grid, model, protect_names = TRUE, ...)
```

Arguments

`grid`	A data frame representing the data grid, or an object of class `ggeffects`, as returned by `predict_response()`.
`model`	The model for which to compute partial residuals. The data grid `grid` should match to predictors in the model.
`...`	Currently not used.

predictor_name The name of the focal predictor, for which partial residuals are computed.

protect_names Logical, if TRUE, preserves column names from the ggeffects objects that is used as grid.

Value

A data frame with residuals for the focal predictor.

Partial Residuals

For **generalized linear models** (glms), residualized scores are computed as inv.link(link(Y) + r) where Y are the predicted values on the response scale, and r are the *working* residuals.

For (generalized) linear **mixed models**, the random effect are also partialled out.

References

Fox J, Weisberg S. Visualizing Fit and Lack of Fit in Complex Regression Models with Predictor Effect Plots and Partial Residuals. Journal of Statistical Software 2018;87.

Examples

```
library(ggeffects)
set.seed(1234)
x <- rnorm(200)
z <- rnorm(200)
# quadratic relationship
y <- 2 * x + x^2 + 4 * z + rnorm(200)

d <- data.frame(x, y, z)
model <- lm(y ~ x + z, data = d)

pr <- predict_response(model, c("x [all]", "z"))
head(residualize_over_grid(pr, model))
```

---

test_predictions *(Pairwise) comparisons between predictions (marginal effects)*

Description

Function to test differences of adjusted predictions for statistical significance. This is usually called contrasts or (pairwise) comparisons, or "marginal effects". hypothesis_test() is an alias.

Usage

```
test_predictions(object, ...)

hypothesis_test(object, ...)

## Default S3 method:
test_predictions(
  object,
  terms = NULL,
  by = NULL,
  test = "pairwise",
  test_args = NULL,
  equivalence = NULL,
  scale = "response",
  p_adjust = NULL,
  df = NULL,
  ci_level = 0.95,
  collapse_levels = FALSE,
  margin = "mean_reference",
  condition = NULL,
  engine = "marginaleffects",
  verbose = TRUE,
  ...
)

## S3 method for class 'ggeffects'
test_predictions(
  object,
  by = NULL,
  test = "pairwise",
  equivalence = NULL,
  scale = "response",
  p_adjust = NULL,
  df = NULL,
  collapse_levels = FALSE,
  engine = "marginaleffects",
  verbose = TRUE,
  ...
)
```

Arguments

object	A fitted model object, or an object of class ggeffects. If object is of class ggeffects, arguments terms, margin and ci_level are taken from the ggeffects object and don't need to be specified.
...	Arguments passed down to data_grid() when creating the reference grid and to marginaleffects::predictions() resp. marginaleffects::slopes(). For instance, arguments type or transform can be used to back-transform

comparisons and contrasts to different scales. vcov can be used to calculate heteroscedasticity-consistent standard errors for contrasts. See examples at the bottom of this vignette for further details.

To define a heteroscedasticity-consistent variance-covariance matrix, you can either use the same arguments as for predict_response() etc., namely vcov and vcov_args. These are then transformed into a matrix and passed down to the vcov argument in **marginaleffects**. Or you directly use the vcov argument. See ?marginaleffects::slopes for further details.

terms

If object is an object of class ggeffects, the same terms argument is used as for the predictions, i.e. terms can be ignored. Else, if object is a model object, terms must be a character vector with the names of the focal terms from object, for which contrasts or comparisons should be displayed. At least one term is required, maximum length is three terms. If the first focal term is numeric, contrasts or comparisons for the *slopes* of this numeric predictor are computed (possibly grouped by the levels of further categorical focal predictors).

by

Character vector specifying the names of predictors to condition on. Hypothesis test is then carried out for focal terms by each level of by variables. This is useful especially for interaction terms, where we want to test the interaction within "groups". by is only relevant for categorical predictors.

test

Hypothesis to test, defined as character string, formula, or data frame. Can be one of:

- String:
  - "pairwise" (default), to test pairwise comparisons.
  - "trend" (or "slope") to test for the linear trend/slope of (usually) continuous predictors. These options are just aliases for setting trend = NULL.
  - "contrast" to test simple contrasts (i.e. each level is tested against the average over *all* levels).
  - "exclude" to test simple contrasts (i.e. each level is tested against the average over *all other* levels, excluding the contrast that is being tested).
  - "interaction" to test interaction contrasts (difference-in-difference contrasts). More flexible interaction contrasts can be calcualted using the test_args argument.
  - "consecutive" to test contrasts between consecutive levels of a predictor.
  - "polynomial" to test orthogonal polynomial contrasts, assuming equally-spaced factor levels.
- String equation:
  A character string with a custom hypothesis, e.g. "b2 = b1". This would test if the second level of a predictor is different from the first level. Custom hypotheses are very flexible. It is also possible to test interaction contrasts (difference-in-difference contrasts) with custom hypotheses, e.g. "(b2 - b1) = (b4 - b3)". See also section *Introduction into contrasts and pairwise comparisons*.

- Formula:
  A formula, where the left-hand side indicates the type of comparison and the right-hand side which pairs to compare. Optionally, grouping variables can be specified after a vertical bar. See also 'Examples'.
  - For the left-hand side, comparisons can be `difference` or `ratio`.
  - For the right-hand side, pairs can be `reference`, `sequential`, or `meandev`. For `reference`, all factor levels are compared to the reference level. `sequential` compares consecutive levels of a predictor. `meandev` compares each factor level against the "average" factor level.
  - If a variable is specified after `|`, comparisons will be grouped by that variable.
- A data frame with custom contrasts. See 'Examples'.
- `NULL`, in which case simple contrasts are computed.

Technical details about the packages used as back-end to calculate contrasts and pairwise comparisons are provided in the section *Packages used as back-end to calculate contrasts and pairwise comparisons* below.

`test_args` Optional arguments passed to `test`, typically provided as named list. Only applies to those options that use the **emmeans** package as backend, e.g. if `test = "interaction"`, `test_args` will be passed to `emmeans::contrast(interaction = test_args)`. For other *emmeans* options (like `"contrast"`, `"exclude"`, `"consecutive"` and so on), `test_args` will be passed to the `option` argument in `emmeans::contrast()`.

`equivalence` ROPE's lower and higher bounds. Should be `"default"` or a vector of length two (e.g., `c(-0.1, 0.1)`). If `"default"`, `bayestestR::rope_range()` is used. Instead of using the `equivalence` argument, it is also possible to call the `equivalence_test()` method directly. This requires the **parameters** package to be loaded. When using `equivalence_test()`, two more columns with information about the ROPE coverage and decision on H0 are added. Furthermore, it is possible to `plot()` the results from `equivalence_test()`. See `bayestestR::equivalence_test()` resp. `parameters::equivalence_test.lm()` for details.

`scale` Character string, indicating the scale on which the contrasts or comparisons are represented. Can be one of:

- `"response"` (default), which would return contrasts on the response scale (e.g. for logistic regression, as probabilities);
- `"link"` to return contrasts on scale of the linear predictors (e.g. for logistic regression, as log-odds);
- `"probability"` (or `"probs"`) returns contrasts on the probability scale, which is required for some model classes, like `MASS::polr()`;
- `"oddsratios"` to return contrasts on the odds ratio scale (only applies to logistic regression models);
- `"irr"` to return contrasts on the odds ratio scale (only applies to count models);
- or a transformation function like `"exp"` or `"log"`, to return transformed (exponentiated respectively logarithmic) contrasts; note that these transformations are applied to the *response scale*.

Note: If the `scale` argument is not supported by the provided `object`, it is automatically changed to a supported scale-type (a message is printed when `verbose = TRUE`).

`p_adjust` Character vector, if not `NULL`, indicates the method to adjust p-values. See `stats::p.adjust()` or `stats::p.adjust.methods` for details. Further possible adjustment methods are `"tukey"` or `"sidak"`, and for `johnson_neyman()`, `"fdr"` (or `"bh"`) and `"esarey"` (or its short-cut `"es"`) are available options. Some caution is necessary when adjusting p-value for multiple comparisons. See also section *P-value adjustment* below.

`df` Degrees of freedom that will be used to compute the p-values and confidence intervals. If `NULL`, degrees of freedom will be extracted from the model using `insight::get_df()` with `type = "wald"`.

`ci_level` Numeric, the level of the confidence intervals. If `object` is an object of class `ggeffects`, the same `ci_level` argument is used as for the predictions, i.e. `ci_level` can be ignored.

`collapse_levels`
Logical, if `TRUE`, term labels that refer to identical levels are no longer separated by "-", but instead collapsed into a unique term label (e.g., `"level a-level a"` becomes `"level a"`). See 'Examples'.

`margin` Character string, indicates the method how to marginalize over non-focal terms. See `predict_response()` for details. If `object` is an object of class `ggeffects`, the same `margin` argument is used as for the predictions, i.e. `margin` can be ignored.

`condition` Named character vector, which indicates covariates that should be held constant at specific values, for instance `condition = c(covariate1 = 20, covariate2 = 5)`.

`engine` Character string, indicates the package to use for computing contrasts and comparisons. Usually, this argument can be ignored, unless you want to explicitly use another package than *marginaleffects* to calculate contrasts and pairwise comparisons. `engine` can be either `"marginaleffects"` (default) or `"emmeans"`. The latter is useful when the **marginaleffects** package is not available, or when the **emmeans** package is preferred. Note that using **emmeans** as back-end is currently not as feature rich as the default (**marginaleffects**). Setting `engine = "emmeans"` provides some additional test options: `"interaction"` to calculate interaction contrasts, `"consecutive"` to calculate contrasts between consecutive levels of a predictor, or a data frame with custom contrasts (see also `test`). There is a third option as well, `engine = "ggeffects"`. However, this option offers less features as the default engine, `"marginaleffects"`. It can be faster in some cases, though, and works for comparing predicted random effects in mixed models, or predicted probabilities of the zero-inflation component. If the **marginaleffects** package is not installed, the **emmeans** package is used automatically. If this package is not installed as well, `engine = "ggeffects"` is used.

`verbose` Toggle messages and warnings.

Value

A data frame containing predictions (e.g. for `test = NULL`), contrasts or pairwise comparisons of adjusted predictions or estimated marginal means.

Introduction into contrasts and pairwise comparisons

There are many ways to test contrasts or pairwise comparisons. A detailed introduction with many (visual) examples is shown in this vignette.

Simple workflow for pairwise comparisons

A simple workflow includes calculating adjusted predictions and passing the results directly to `test_predictions()`, e.g.:

```
# 1. fit your model
model <- lm(mpg ~ hp + wt + am, data = mtcars)
# 2. calculate adjusted predictions
pr <- predict_response(model, "am")
pr
# 3. test pairwise comparisons
test_predictions(pr)
```

See also this vignette.

Packages used as back-end to calculate contrasts and pairwise comparisons

The `test` argument is used to define which kind of contrast or comparison should be calculated. The default is to use the **marginaleffects** package. Here are some technical details about the packages used as back-end. When `test` is...

- `"pairwise"` (default), pairwise comparisons are based on the **marginaleffects** package.
- `"trend"` or `"slope"` also uses the **marginaleffects** package.
- `"contrast"` uses the **emmeans** package, i.e. `emmeans::contrast(method = "eff")` is called.
- `"exclude"` relies on the **emmeans** package, i.e. `emmeans::contrast(method = "del.eff")` is called.
- `"polynomial"` relies on the **emmeans** package, i.e. `emmeans::contrast(method = "poly")` is called.
- `"interaction"` uses the **emmeans** package, i.e. `emmeans::contrast(interaction = ...)` is called.
- `"consecutive"` also relies on the **emmeans** package, i.e. `emmeans::contrast(method = "consec")` is called.
- a character string with a custom hypothesis, the **marginaleffects** package is used.
- a data frame with custom contrasts, **emmeans** is used again.
- for formulas, the **marginaleffects** package is used.
- `NULL` calls functions from the **marginaleffects** package with `hypothesis = NULL`.

- If all focal terms are only present as random effects in a mixed model, or if predicted probabilities for the zero-inflation component of a model should be tested, functions from the **ggeffects** package are used. There is an example for pairwise comparisons of random effects in this vignette.

P-value adjustment for multiple comparisons

Note that p-value adjustment for methods supported by `p.adjust()` (see also `p.adjust.methods`), each row is considered as one set of comparisons, no matter which `test` was specified. That is, for instance, when `test_predictions()` returns eight rows of predictions (when `test = NULL`), and `p_adjust = "bonferroni"`, the p-values are adjusted in the same way as if we had a test of pairwise comparisons (`test = "pairwise"`) where eight rows of comparisons are returned. For methods `"tukey"` or `"sidak"`, a rank adjustment is done based on the number of combinations of levels from the focal predictors in `terms`. Thus, the latter two methods may be useful for certain tests only, in particular pairwise comparisons.

For `johnson_neyman()`, the only available adjustment methods are `"fdr"` (or `"bh"`) (*Benjamini & Hochberg (1995)*) and `"esarey"` (or `"es"`) (*Esarey and Sumner 2017*). These usually return similar results. The major difference is that `"fdr"` can be slightly faster and more stable in edge cases, however, confidence intervals are not updated. Only the p-values are adjusted. `"esarey"` is slower, but confidence intervals are updated as well.

Global options to choose package for calculating comparisons

`ggeffects_test_engine` can be used as option to either use the **marginaleffects** package for computing contrasts and comparisons (default), or the **emmeans** package (e.g. `options(ggeffects_test_engine = "emmeans")`). The latter is useful when the **marginaleffects** package is not available, or when the **emmeans** package is preferred. You can also provide the engine directly, e.g. `test_predictions(..., engine = "emmeans")`. Note that using **emmeans** as backend is currently not as feature rich as the default (**marginaleffects**).

If `engine = "emmeans"`, the `test` argument can also be `"interaction"` to calculate interaction contrasts (difference-in-difference contrasts), `"consecutive"` to calculate contrasts between consecutive levels of a predictor, or a data frame with custom contrasts. If `test` is one of the latter options, and `engine` is not specified, the `engine` is automatically set to `"emmeans"`. Additionally, the `test_args` argument can be used to specify further options for those contrasts. See 'Examples' and documentation of `test_args`.

If the **marginaleffects** package is not installed, the **emmeans** package is used automatically. If this package is not installed as well, `engine = "ggeffects"` is used.

Global Options to Customize Tables when Printing

The `verbose` argument can be used to display or silence messages and warnings. Furthermore, `options()` can be used to set defaults for the `print()` and `print_html()` method. The following options are available, which can simply be run in the console:

- `ggeffects_ci_brackets`: Define a character vector of length two, indicating the opening and closing parentheses that encompass the confidence intervals values, e.g. `options(ggeffects_ci_brackets = c("[", "]"))`.

- ggeffects_collapse_ci: Logical, if TRUE, the columns with predicted values (or contrasts) and confidence intervals are collapsed into one column, e.g. options(ggeffects_collapse_ci = TRUE).
- ggeffects_collapse_p: Logical, if TRUE, the columns with predicted values (or contrasts) and p-values are collapsed into one column, e.g. options(ggeffects_collapse_p = TRUE). Note that p-values are replaced by asterisk-symbols (stars) or empty strings when ggeffects_collapse_p = TRUE, depending on the significance level.
- ggeffects_collapse_tables: Logical, if TRUE, multiple tables for subgroups are combined into one table. Only works when there is more than one focal term, e.g. options(ggeffects_collapse_tables = TRUE).
- ggeffects_output_format: String, either "text", "markdown" or "html". Defines the default output format from predict_response(). If "html", a formatted HTML table is created and printed to the view pane. "markdown" creates a markdown-formatted table inside Rmarkdown documents, and prints a text-format table to the console when used interactively. If "text" or NULL, a formatted table is printed to the console, e.g. options(ggeffects_output_format = "html").
- ggeffects_html_engine: String, either "tt" or "gt". Defines the default engine to use for printing HTML tables. If "tt", the *tinytable* package is used, if "gt", the *gt* package is used, e.g. options(ggeffects_html_engine = "gt").

Use options(<option_name> = NULL) to remove the option.

References

Esarey, J., & Sumner, J. L. (2017). Marginal effects in interaction models: Determining and controlling the false positive rate. Comparative Political Studies, 1–33. Advance online publication. doi: 10.1177/0010414017730080

See Also

There is also an equivalence_test() method in the **parameters** package (parameters::equivalence_test.lm()), which can be used to test contrasts or comparisons for practical equivalence. This method also has a plot() method, hence it is possible to do something like:

```
library(parameters)
predict_response(model, focal_terms) |>
  equivalence_test() |>
  plot()
```

Examples

```
data(efc)
efc$c172code <- as.factor(efc$c172code)
efc$c161sex <- as.factor(efc$c161sex)
levels(efc$c161sex) <- c("male", "female")
m <- lm(barthtot ~ c12hour + neg_c_7 + c161sex + c172code, data = efc)

# direct computation of comparisons
```

```
test_predictions(m, "c172code")

# passing a `ggeffects` object
pred <- predict_response(m, "c172code")
test_predictions(pred)

# test for slope
test_predictions(m, "c12hour")

# interaction - contrasts by groups
m <- lm(barthtot ~ c12hour + c161sex * c172code + neg_c_7, data = efc)
test_predictions(m, c("c161sex", "c172code"), test = NULL)

# interaction - pairwise comparisons by groups
test_predictions(m, c("c161sex", "c172code"))

# equivalence testing
test_predictions(m, c("c161sex", "c172code"), equivalence = c(-2.96, 2.96))

# equivalence testing, using the parameters package
pr <- predict_response(m, c("c161sex", "c172code"))
parameters::equivalence_test(pr)

# interaction - collapse unique levels
test_predictions(m, c("c161sex", "c172code"), collapse_levels = TRUE)

# p-value adjustment
test_predictions(m, c("c161sex", "c172code"), p_adjust = "tukey")

# not all comparisons, only by specific group levels
test_predictions(m, "c172code", by = "c161sex")

# specific comparisons
test_predictions(m, c("c161sex", "c172code"), test = "b2 = b1")

# interaction - slope by groups
m <- lm(barthtot ~ c12hour + neg_c_7 * c172code + c161sex, data = efc)
test_predictions(m, c("neg_c_7", "c172code"))

# Interaction and consecutive contrasts -----------------
# -------------------------------------------------------
data(coffee_data, package = "ggeffects")
m <- lm(alertness ~ time * coffee + sex, data = coffee_data)

# consecutive contrasts
test_predictions(m, "time", by = "coffee", test = "consecutive")

# same as (using formula):
pr <- predict_response(m, c("time", "coffee"))
test_predictions(pr, test = difference ~ sequential | coffee)

# interaction contrasts - difference-in-difference comparisons
pr <- predict_response(m, c("time", "coffee"), margin = "marginalmeans")
```

```
test_predictions(pr, test = "interaction")

# Ratio contrasts -------------------------------------
# -----------------------------------------------------
test_predictions(test = ratio ~ reference | coffee)

# Custom contrasts ------------------------------------
# -----------------------------------------------------
wakeup_time <- data.frame(
  "wakeup vs later" = c(-2, 1, 1) / 2, # make sure each "side" sums to (+/-)1!
  "start vs end of day" = c(-1, 0, 1)
)
test_predictions(m, "time", by = "coffee", test = wakeup_time)

# Example: marginal effects ---------------------------
# -----------------------------------------------------
data(iris)
m <- lm(Petal.Width ~ Petal.Length + Species, data = iris)

# we now want the marginal effects for "Species". We can calculate
# the marginal effect using the "marginaleffects" package
marginaleffects::avg_slopes(m, variables = "Species")

# finally, test_predictions() returns the same. while the previous results
# report the marginal effect compared to the reference level "setosa",
# test_predictions() returns the marginal effects for all pairwise comparisons
test_predictions(m, "Species")
```

`values_at` *Calculate representative values of a vector*

Description

This function calculates representative values of a vector, like minimum/maximum values or lower, median and upper quartile etc., which can be used for numeric vectors to plot adjusted predictions at these representative values.

Usage

```
values_at(x, values = "meansd")

representative_values(x, values = "meansd")
```

Arguments

`x`	A numeric vector.
`values`	Character vector, naming a pattern for which representative values should be calculcated.

- "minmax": (default) minimum and maximum values (lower and upper bounds) of x.
- "meansd": uses the mean value of x as well as one standard deviation below and above mean value to plot the effect of the moderator on the independent variable.
- "zeromax": is similar to the "minmax" option, however, 0 is always used as minimum value for x. This may be useful for predictors that don't have an empirical zero-value, but absence of moderation should be simulated by using 0 as minimum.
- "fivenum": calculates and uses the Tukey's five number summary (minimum, lower-hinge, median, upper-hinge, maximum) of x. This is equivalent to "quartiles".
- "threenum": calculates a three number summary (lower-hinge, median, and upper-hinge) of x. This is equivalent to "quartiles2".
- "terciles": calculates and uses the terciles (lower and upper third) of x, *including* minimum and maximum value.
- "terciles2": calculates and uses the terciles (lower and upper third) of x, *excluding* minimum and maximum value.
- an option to compute a range of percentiles is also possible, using "percentile", followed by the percentage of the range. For example, "percentile95" will calculate the 95% range of x.
- "all": uses all values of x.

Value

A numeric vector, representing the required values from x, like minimum/maximum value or mean and +/- 1 SD. If x is missing, a function, pre-programmed with n and length is returned. See examples.

Examples

```
data(efc)
values_at(efc$c12hour)
values_at(efc$c12hour, "quartiles2")

mean_sd <- values_at(values = "meansd")
mean_sd(efc$c12hour)
```

---

vcov *Calculate variance-covariance matrix for adjusted predictions*

Description

Returns the variance-covariance matrix for the predicted values from object.

Usage

```
## S3 method for class 'ggeffects'
vcov(object, vcov = NULL, vcov_args = NULL, verbose = TRUE, ...)
```

Arguments

object	An object of class "ggeffects", as returned by predict_response().
vcov	Variance-covariance matrix used to compute uncertainty estimates (e.g., for confidence intervals based on robust standard errors). This argument accepts a covariance matrix, a function which returns a covariance matrix, or a string which identifies the function to be used to compute the covariance matrix. • A covariance matrix • A function which returns a covariance matrix (e.g., stats::vcov()) • A string which indicates the kind of uncertainty estimates to return. – Heteroskedasticity-consistent: "HC", "HC0", "HC1", "HC2", "HC3", "HC4", "HC4m", "HC5". See ?sandwich::vcovHC – Cluster-robust: "vcovCR", "CR0", "CR1", "CR1p", "CR1S", "CR2", "CR3". See ?clubSandwich::vcovCR. – Bootstrap: "BS", "xy", "fractional", "jackknife", "residual", "wild", "mammen", "norm", "webb". See ?sandwich::vcovBS – Other sandwich package functions: "HAC", "PC", "CL", or "PL". If NULL, standard errors (and confidence intervals) for predictions are based on the standard errors as returned by the predict()-function. **Note** that probably not all model objects that work with predict_response() are also supported by the **sandwich** or **clubSandwich** packages. See details in this vignette.
vcov_args	List of arguments to be passed to the function identified by the vcov argument. This function is typically supplied by the **sandwich** or **clubSandwich** packages. Please refer to their documentation (e.g., ?sandwich::vcovHAC) to see the list of available arguments. If no estimation type (argument type) is given, the default type for "HC" equals the default from the **sandwich** package; for type "CR" the default is set to "CR3". For other defaults, refer to the documentation in the **sandwich** or **clubSandwich** package.
verbose	Toggle messages or warnings.
...	Currently not used.

Details

The returned matrix has as many rows (and columns) as possible combinations of predicted values from the predict_response() call. For example, if there are two variables in the terms-argument of predict_response() with 3 and 4 levels each, there will be 3*4 combinations of predicted values, so the returned matrix has a 12x12 dimension. In short, nrow(object) is always equal to nrow(vcov(object)). See also 'Examples'.

Value

The variance-covariance matrix for the predicted values from object.

Examples

```
data(efc)
model <- lm(barthtot ~ c12hour + neg_c_7 + c161sex + c172code, data = efc)
result <- predict_response(model, c("c12hour [meansd]", "c161sex"))

vcov(result)

# compare standard errors
sqrt(diag(vcov(result)))
as.data.frame(result)

# only two predicted values, no further terms
# vcov() returns a 2x2 matrix
result <- predict_response(model, "c161sex")
vcov(result)

# 2 levels for c161sex multiplied by 3 levels for c172code
# result in 6 combinations of predicted values
# thus vcov() returns a 6x6 matrix
result <- predict_response(model, c("c161sex", "c172code"))
vcov(result)
```

Index