

Package: bartisan (via r-universe)

October 28, 2026

Type Package

Title Generalized Bayesian Additive Regression Trees

Version 0.1.0

Description Fits Bayesian additive regression trees (BART) models for likelihoods outside the conditionally conjugate Gaussian case, using the Laplace-approximation reversible-jump sampler of Linero (2025) <[doi:10.1080/01621459.2024.2337156](https://doi.org/10.1080/01621459.2024.2337156)>. Supported response families include Gaussian, binomial, Poisson, negative binomial, gamma, beta, ordered beta, ordinal (cumulative link), multinomial, zero-inflated counts, a Tweedie compound Poisson for a non-negative response with a point mass at zero, accelerated failure time and proportional hazards models for right-censored survival data, location-scale regression, and a Dirichlet process mixture for the error distribution. Decision rules may be hard, as in standard BART, or soft, as in the SoftBart model of Linero and Yang (2018) <[doi:10.1111/rssb.12293](https://doi.org/10.1111/rssb.12293)>, which yields smoother fits. The interface follows that of glm(), so that a model is specified with a formula, a data frame, and a family.

License GPL-2

Encoding UTF-8

Depends R (>= 4.1.0)

Imports arg (>= 0.2.1), cli (>= 3.6.4), rlang (>= 1.2.0), Matrix, Rcpp (>= 1.1.2), parallel, reformulas (>= 0.4.4), ggplot2 (>= 4.0.1), stats, utils

LinkingTo Rcpp, RcppArmadillo

Suggests bayesplot, cobalt, collapse, future, future.apply, insight, knitr, loo, MASS, marginaleffects, nlme, lme4, patrick (>= 0.3.0), performance, posterior, progressr, rmarkdown, rstanarm, rstantools, survival, testthat (>= 3.0.0), WeightIt

VignetteBuilder knitr, rmarkdown

URL <https://github.com/ngreifer/bartisan>,
<https://ngreifer.github.io/bartisan/>

BugReports <https://github.com/ngreifer/bartisan/issues>

Config/testthat/edition 3

Config/roxygen2/version 8.1.0

LazyData true

NeedsCompilation yes

Author Noah Greifer [aut, cre, cph] (ORCID:
<https://orcid.org/0000-0003-3067-7154>), Antonio R. Linero
 [ctb, cph] (Author of FlexBart, the reference implementation of
 Linero (2025), from which the MCMC engine is adapted)

Maintainer Noah Greifer <noah.greifer@gmail.com>

Repository <https://cran.r-universe.dev>

Date/Publication 2026-10-28 11:40:07 UTC

RemoteUrl <https://github.com/cran/bartisan>

RemoteRef HEAD

RemoteSha c8e7b6d230744473bff7dd9488f0f3bfb44e1f3c

Contents

bartisan-package	3
bartisan	5
bartisan-families	11
bartisan-interop	20
bartisan-marginaleffects	26
bartisan_control	30
bcf	39
coef.bartisan_fit	43
custom_family	45
diagnose	49
error_density	53
estimate_effect	54
partial_dependence	58
predict.bartisan_fit	61
print.bartisan_fit	65
print.bcf_fit	67
ranef.bartisan_fit	68
rhc	70
variable_importance	71
vc	74

Index	79
--------------	-----------

Description

Fits Bayesian additive regression trees (BART) models for likelihoods outside the conditionally conjugate Gaussian case, using the Laplace-approximation reversible-jump sampler of Linero (2025) [doi:10.1080/01621459.2024.2337156](https://doi.org/10.1080/01621459.2024.2337156). Supported response families include Gaussian, binomial, Poisson, negative binomial, gamma, beta, ordered beta, ordinal (cumulative link), multinomial, zero-inflated counts, a Tweedie compound Poisson for a non-negative response with a point mass at zero, accelerated failure time and proportional hazards models for right-censored survival data, location-scale regression, and a Dirichlet process mixture for the error distribution. Decision rules may be hard, as in standard BART, or soft, as in the 'SoftBart' model of Linero and Yang (2018) [doi:10.1111/rssb.12293](https://doi.org/10.1111/rssb.12293), which yields smoother fits. The interface follows that of `glm()`, so that a model is specified with a formula, a data frame, and a family.

Details

Fits Bayesian additive regression trees (BART) the way `stats::glm()` fits a generalized linear model: a formula, a data frame, and a family. The forest replaces the linear predictor, so nothing has to be said about which terms enter the model, which are curved, or which interact.

```
fit <- bartisan(y ~ x1 + x2 + x3, data = d)
```

Common Tasks:

Most of what a fitted model is used for is handled by a package that already does that job well, and **bartisan** registers the methods those packages need. This table is the map.

Task	Use
fit a model	<code>bartisan()</code>
choose a likelihood	<code>bartisan-families</code> , <code>vignette("families")</code>
change the sampler's settings	<code>bartisan_control()</code>
predict for new data	<code>predict.bartisan_fit()</code>
a prediction with an interval	<code>marginaleffects::predictions()</code>
an interval for a new observation, noise included	<code>posterior_predict()</code>
how much a predictor moves the outcome	<code>marginaleffects::avg_comparisons()</code> , <code>vignette("effects")</code>
a partial dependence plot	<code>partial_dependence()</code>
which predictors the forest uses	<code>variable_importance()</code>
to tell the prior which predictors matter	<code>split_prior</code> in <code>bartisan_control()</code>
to see what prior a fit was given	<code>prior_summary()</code>
to see what that prior implies about the outcome	<code>prior_only</code> in <code>bartisan()</code>
to give one forest its own predictors or settings	a list of formulas, and per-forest arguments; see <code>bartisan-families</code>
a coefficient that varies with the other predictors	<code>vc()</code> in the formula, <code>vignette("varying")</code>
a treatment effect with its own prior	<code>bcb()</code> , <code>vignette("causal")</code>
the ATE, ATT or the effect for each unit	<code>estimate_effect()</code>
to check it converged and mixed	<code>diagnose()</code> , then <code>as_draws()</code> with <code>bayesplot</code> , <code>vignette("diagnostics")</code>

to check it fits
 to compare two models
 survival data
 a likelihood of one's own

```
pp_check(), residuals()
loo(), vignette("comparison")
ph(), dpm_aft(), vignette("survival")
custom_family()
```

Before the First Fit:

There are a few things worth knowing first, and none of them requires knowing anything about Bayesian statistics.

A forest has no slope to read off, so the effect of a predictor is found by asking the fitted model what it predicts under one value of that predictor and under another, and taking the difference. `marginalEffects::avg_comparisons(fit, variables = "x")` does exactly that, and reports an interval with it. This is a better habit than reading coefficients even where coefficients exist (i.e., in a linear model), and here it is the only habit available.

A 95% interval from any of the above is the range the model considers most plausible given the data and the model. It already includes the uncertainty from not knowing the shape of the relationship, which is the part a linear model leaves out by assuming it away.

The defaults are meant to be used. The settings in `bartisan_control()` are there for the analyses that need them; the priors and the number of trees are chosen to work across a wide range of problems, and tuning them is rarely where the gains are. Choosing the right `family` matters much more.

Flexibility and Its Limits:

The model is flexible about the shape of the relationship and about nothing else. Whether a predictor is a cause, whether the sample represents the population, and whether the outcome was measured well are assumptions the analysis brings to the model rather than things it can supply: a forest fitted to confounded data returns a confounded answer with a tight interval around it.

Author(s)

Maintainer: Noah Greifer <noah.greifer@gmail.com> ([ORCID](#)) [copyright holder]

Authors:

- Noah Greifer <noah.greifer@gmail.com> ([ORCID](#)) [copyright holder]

Other contributors:

- Antonio R. Linero (Author of FlexBart, the reference implementation of Linero (2025), from which the MCMC engine is adapted) [contributor, copyright holder]

See Also

`bartisan()` to fit a model; `bartisan-families` to choose a likelihood; `bartisan-marginalEffects` and `bartisan-interop` for the packages that read a fit; `vignette("bartisan")` for a worked analysis and `vignette("implementation")` for how the sampler works.

bartisan

Fit a generalized Bayesian additive regression trees (BART) model

Description

Fits a BART model in which the response distribution is arbitrary rather than restricted to the conditionally conjugate cases, using the Laplace-approximation reversible-jump sampler of Linero (2025). Decision rules may be soft, as in Linero and Yang (2018), which gives smoother fits than the step functions of standard BART. The interface mirrors that of `stats::glm()`: a formula, a data frame, and a family, with the families that `glm()` has no counterpart for documented at [bartisan-families](#).

Usage

```
bartisan(
  formula,
  data,
  family = NULL,
  weights = NULL,
  offset = NULL,
  subset = NULL,
  na.action = stats::na.pass,
  control = bartisan_control(),
  prior_only = FALSE,
  ...
)
```

Arguments

formula	a model formula (or a list thereof for families with multiple additive predictors). Survival families take a <code>survival::Surv()</code> object on the left. A (1 group) term adds a group-level random intercept, in the notation of lme4 ; see Details.
data	a data frame containing the variables named in formula.
family	the response distribution, given as a <code>stats::family</code> object, as one of the families in bartisan-families , or as the name of either. A family object is accepted when the distribution it names is one this package implements, since the likelihood is the package's rather than the object's: a family object carries a link and a variance function and not a density, so one naming anything else (e.g., <code>stats::inverse.gaussian</code> , or a Tweedie from another package) is an error rather than something a likelihood can be built from, and <code>custom_family()</code> is the route for those. Links are used as supplied, and a link the package does not compile is composed onto the scale its family works on. Default is <code>NULL</code> , in which case the family is read off the response and a message reports the choice; see Details for the rules.
weights	optional; prior weights, one per observation. For a binomial response given as proportions, these are the numbers of trials, as in <code>glm()</code> . Note that a weight

	above one on a binary response repeats that observation's outcome at its own covariate values, which separates the response there and can leave the leaf scale unidentified; see <code>update_sigma_mu</code> in <code>bartisan_control()</code> .
<code>offset</code>	optional; a known component of the additive predictor, on the link scale. One value per observation, or a matrix with one column per additive predictor to give each its own; see Details.
<code>subset</code>	optional; a vector specifying the subset of rows to use.
<code>na.action</code>	how missing values are handled. Default is <code>stats::na.pass</code> , which keeps rows whose <i>predictors</i> are missing and lets the splitting rules decide where they go, which is something the trees can do and <code>lm()</code> and <code>glm()</code> cannot; see Details. Pass <code>stats::na.omit</code> to drop any row with a missing value anywhere instead. Note that rows with a missing response, weight, or offset are dropped either way, with a warning, since there is nothing to fit them to.
<code>control</code>	a <code><bartisan_control></code> object; the output of a call to <code>bartisan_control()</code> , containing the sampler and prior settings.
<code>prior_only</code>	logical; whether to draw from the prior rather than the posterior, as a prior predictive check requires. Default is <code>FALSE</code> . Not available for every family; see Details.
<code>...</code>	further arguments to <code>bartisan_control()</code> , which are merged into <code>control</code> and override any value given there, so that <code>bartisan(..., num_trees = 20)</code> and <code>bartisan(..., control = bartisan_control(num_trees = 20))</code> are the same call. A name that is not an argument of <code>bartisan_control()</code> is an error rather than being silently ignored.

Details

The Model Formula:

The right-hand side of formula lists candidate predictors; the model finds interactions and non-linearity on its own, so $y \sim x1 + x2 + x3$ is usually the right specification. For a family with more than one additive predictor this may be a list of formulas, one per forest, to give each one its own predictors. The first is the model for the main parameter and carries the response; the rest need no response, and follow the order in [bartisan-families](#), under "Several additive predictors", which also gives the name of each forest so the list can be named instead of ordered:

```
bartisan(list(y ~ x1 + x2,
             ~ x2 + x3), data = d,
         family = gaussian_ls())

bartisan(list(mean = y ~ x1 + x2,
             log_sd = ~ x2), data = d,
         family = gaussian_ls())
```

When a single formula is given, it applies to every forest, which is the ordinary case.

A formula naming no predictor at all makes that parameter a constant. ~ 1 leaves its forest nothing to split on, so every tree in it is a stump and the forest is a single drawn scalar rather than a function of the predictors.

`vc()` terms for varying coefficient models are read out of each formula in turn, so a parameter has the varying coefficients its own formula asks for and no others, which makes the forests two-dimensional (one axis the parameter, the other the coefficient). `vc()` documents how they are then named and keyed.

The Sampler:

At each candidate move it builds a Gaussian approximation to the conditional posterior of the affected leaf parameters, by Fisher scoring, and uses that approximation as the proposal in a reversible-jump Metropolis step. The approximation only has to be good enough to be accepted often; the stationary distribution is the exact posterior either way.

What a new family therefore has to supply is only the log density of one observation and its first two derivatives with respect to the additive predictor. Families whose response has more than one unconstrained parameter, such as `multinomial()` and `gaussian_ls()`, carry one forest per parameter. Because that is the whole interface, it can be reached from R: `custom_family()` takes the log density as an R function and differences it for the derivatives.

Inferring the Family:

The family argument may be left alone, in which case it is read off the response:

Response	Family
<code>survival::Surv()</code> object, or a two-column matrix of times and events	<code>dpm_aft()</code>
ordered factor	<code>ordinal()</code>
logical, or two levels, or numeric zeros and ones	<code>binomial()</code>
factor or character with more than two levels	<code>multinomial()</code>
two-column matrix of successes and failures	<code>binomial()</code>
anything else	<code>dpm()</code>

A message reports the choice, and naming family silences it and overrides the choice.

Some scenarios are worth noting. A count is read as a numeric variable and therefore has `dpm()` as its default. And a numeric response with exactly two values other than zero and one (e.g., `c(1, 2)`) is also given `dpm()` as its default rather than `binomial()`, which of the two counts as the success not being something to guess at.

Soft Decision Rules:

By default a decision rule is a smooth gate rather than a step, so an observation reaches every leaf with some weight and the fitted function is smooth. `gate` in `bartisan_control()` chooses both whether the rules are soft and, if they are, the gate's shape; the default is the bounded "smoothstep", and "logistic" is Linero and Yang's (2018) original. Soft rules cost more per iteration, since a leaf now touches every observation rather than only the ones inside its cell, and they make the leaf parameters of a tree dependent on one another. Combining them with a non-conjugate likelihood is an extension of Linero (2025), which leaves it as an open problem; it is handled here by giving the reversible-jump move a bivariate Laplace proposal for the pair of child leaves, which reduces to Linero's independent pair exactly when the rules are hard. Set `gate = "hard"` in `bartisan_control()` for the faster hard-rule sampler.

Random Intercepts:

A `(1 | group)` term in the formula adds an intercept per level of group, drawn from a common mean-zero normal whose standard deviation is itself drawn under the same half-Cauchy prior the

leaf scale uses. Several grouping factors are allowed, and $(1 \mid a/b)$ expands to nesting as it does in **lme4**:

```
bartisan(y ~ x1 + x2 + (1 | school), data = d)
bartisan(y ~ x1 + (1 | school) + (1 | year), data = d)
```

The intercepts are in `fit$ranef` and their standard deviations in `fit$tau`, one matrix per additive predictor. A family with several predictors gets a separate set for each (i.e., a zero-inflated count model has a group effect on the count part and another on the inflation part), and they are independent of one another.

Only random intercepts are supported; a random slope is refused rather than ignored. The reason is that a random intercept is a scalar entering the predictor with weight one for the observations in its level, exactly as a leaf is once its gate is removed, so the sampler's leaf machinery handles it exactly; a slope is a different shape of parameter. A variable whose effect varies by group belongs in the fixed part of the formula, where a tree can split on the group and on the variable together and get an interaction of any shape.

A grouping factor can also go in the fixed part, where a tree splits on it like anything else, and with few large groups that is the better choice: the group means are well determined without pooling and a split can interact the group with the covariates. The random intercept wins where there are many small groups, which is where partial pooling earns its keep.

A level of group that was not present at fitting time is given the prior mean of zero when predicting, with a warning.

Offsets:

An offset is a known part of the additive predictor, supplied on the link scale and not estimated: the log of an exposure for a count, say. It is given as one value per observation, through `offset` or as an `offset()` term in the formula.

A family with several additive predictors takes the same offset on each of them, unless it is given a matrix with one column per predictor, which offsets each separately: a category-specific exposure for `multinomial()`, say, or an offset on the count part of `zi_poisson()` and not on its zero-inflation part. `?bartisan-families` lists each family's forests in order, which is the column order. `predict()` takes the same two forms.

An offset is not a function of the predictors, so it cannot be rebuilt for rows the fit has not seen: a model fitted with one requires offset at `predict()` time.

Missing Predictor Values:

A missing predictor is not imputed and its row is not dropped, which is the default here because a tree can do something better with a missing value than either. Instead each splitting rule carries the answer for itself. A rule on a variable that has missing values is drawn as one of three, with equal probability:

- $x < c$, or missing, goes left;
- $x < c$ goes left, missing goes right;
- missing goes left, present goes right.

This is missingness incorporated in attributes, and the third rule lets the model split on missingness itself, so a variable whose absence carries the signal is usable even where its observed values say nothing.

`predict()` accepts missing values in a column that had them at fitting time, those being the columns whose rules carry an answer. And what the model estimates is the mean of the response

given the predictors and the pattern of missingness, which is the quantity prediction calls for; where the estimand is a regression or causal effect defined on complete data, multiple imputation may be a better tool.

Preprocessing:

Predictors are mapped to the unit interval because the cutpoint prior is uniform on a node's live range and the soft-rule bandwidth is measured on the predictor scale. Factors share a single weight in the sparsity prior, so that a factor is selected or not as a whole rather than one level at a time. The additive predictor starts from an intercept-only fit, so the leaf prior describes departures from that fit rather than the absolute level of the response. That starting value is the exact null-model estimate for most families; for the accelerated failure time families, where censoring makes the sample mean of the log times biased, and for the zero-inflated and ordered beta families, it is a moment approximation, which the sampler then moves away from.

Drawing From the Prior (`prior_only`):

Setting `prior_only = TRUE` fits the same model to no data. Every observation is given a weight of zero, and since the weight multiplies that observation's log density, its gradient, and its curvature, the likelihood is flat: each tree move is accepted or rejected on the prior alone and each leaf is drawn from its prior. A family that draws an auxiliary parameter from the response directly rather than through the weighted density (the mixture atoms under `dpm()`, the latent utilities under `multinomial(link = "probit")`) is told separately that a weightless observation carries no information, and draws that parameter from its own prior instead. The sampler is otherwise untouched, so what comes back is an ordinary fit whose draws are prior draws, and `rstantools::posterior_predict()` on it gives the prior predictive distribution.

It answers a question the priors themselves cannot. `k`, `gamma` and `beta` are statements about trees and leaves, and what they imply about an outcome is opaque. The replicates put that on the response's own scale, where it can be judged: a prior predictive that puts its mass where the outcome cannot go, or spread over an implausible range, is a prior worth changing before the data are seen and before any of their information is spent.

The additive predictor is anchored at an intercept-only fit on the link scale and the leaf scale is calibrated from the response, which is how a BART prior is specified. The replicates therefore take their location and scale from the response and everything else from the prior: which predictors are split on, how deep, how far the fitted function departs from that anchor. Read them for shape and spread rather than for level, and note that the wider a prior is the wider its replicates, so `gaussian_ls()` and `Gamma_ls()`, which put a log scale in a second forest, run far wider than the response ever does.

`loo()`, `waic()` and `kfold()` score a fit against data, so they refuse a prior-only fit, and `performance::model_performance` leaves out the three columns built on them.

Value

A `<bartisan_fit>` object, a list with the following components among others.

<code>eta</code>	a list with one matrix per additive predictor, each of posterior draws by observation, on the link scale.
<code>fitted</code>	fitted values on the response scale, averaged over draws.
<code>counts</code>	a list with one matrix per additive predictor, of the number of splitting rules using each predictor group in each draw. Useful for variable selection.

aux	draws of the nuisance parameters, such as the residual standard deviation or the ordinal cutpoints, when the family has any.
has_na	which predictor columns contained a missing value, which determines where <code>predict()</code> will accept one.
sigma_mu, bandwidth	draws of the leaf standard deviation and, for soft rules, the per-tree gate bandwidths.
loglik	the log likelihood at each draw.
control	the <code><bartisan_control></code> object the fit used, with any settings given in ... merged in.

References

Linero, A. R. (2025). Generalized Bayesian additive regression trees models: beyond conditional conjugacy. *Journal of the American Statistical Association*, 120(549), 356–369. doi:10.1080/01621459.2024.2337156

Linero, A. R., & Yang, Y. (2018). Bayesian regression tree ensembles that adapt to smoothness and sparsity. *Journal of the Royal Statistical Society Series B*, 80(5), 1087–1110. doi:10.1111/rssb.12293

See Also

`bartisan_control()` for the sampler and prior settings; `predict.bartisan_fit()` for prediction; `bartisan-families` for the likelihoods, and `vignette("families")` for a family-by-family guide; `bartisan-marginaleffects` for reading effects off a fit

Examples

```
data("rhc")
set.seed(123)

# Whether a patient died, with the treatment and every
# covariate recorded on admission as candidate
# predictors, and the family read off the response
model <- death ~ rhc + age + sex + race + edu + aps +
  meanbp + resp + hema + pafi + paco2 + crea + surv2m +
  card

fit <- bartisan(model, data = rhc, num_trees = 10,
  num_burn = 50, num_draws = 50,
  verbose = FALSE)

fit

# Fitted probabilities
head(predict(fit, type = "response"))

# The forest has no coefficients, so an effect is a
# contrast of predictions, here of catheterization on
# the probability of death
if (rlang::is_installed(c("marginaleffects", "collapse"))) {
```

```

    marginaleffects::avg_comparisons(fit, variables = "rhc")
}

```

bartisan-families

Response families for generalized BART

Description

`bartisan()` accepts the `stats::family` objects used by `stats::glm()`, so `gaussian()`, `binomial("probit")`, `poisson()` and `stats::Gamma()` all work unchanged. The functions documented here supply the additional families that have no `glm()` counterpart, in the same style, so that they can be passed to the family argument the same way.

One caveat carries over from base R: `stats::Gamma()` defaults to `link = "inverse"`, which is the worst link for this sampler, so write `Gamma("log")`, or name the family as the string `family = "Gamma"`, which is this package's own spelling and resolves to the log link; see Details.

Usage

```
negbin(link = "log", theta = NULL)
```

```
ordinal(link = "logit", cut_alpha = 1)
```

```
multinomial(link = "logit", reference = NULL, replicates = 200L)
```

```

dpm_aft(
  nu = 10,
  q = 0.95,
  k_s = 10,
  alpha = NULL,
  max_clusters = NULL,
  psi = 0.5
)

```

```
dpm(nu = 10, q = 0.95, k_s = 10, alpha = NULL, max_clusters = NULL, psi = 0.5)
```

```
weibull_aft()
```

```
loglogistic_aft()
```

```
lognormal_aft()
```

```
ph(num_bins = NULL, lambda_shape = 1, update_lambda = TRUE)
```

```
gaussian_ls(link = "identity")
```

```
Gamma_ls(link = "log")
```

```

zi_poisson(link = "log")

zi_negbin(link = "log", theta = NULL)

Beta(link = "logit", phi = NULL)

ordbeta(link = "logit", phi = NULL, cut_alpha = 1)

tweedie(link = "log", power = 1.5, phi = NULL)

```

Arguments

link	string; the link function. Allowable options include "logit" (the default), "probit", and "cloglog" for ordinal() and Beta(), and "logit" (the default) and "probit" for multinomial(). The remaining families take one link each, which is therefore the default: "log" for negbin(), zi_poisson(), zi_negbin(), and Gamma_ls(), "logit" for ordbeta(), and "identity" for gaussian_ls(). Each family compiles the links for which the additive predictor is the natural unconstrained scale; any other link is applied from R, for the families where that is well defined. See Details.
theta	numeric; for negbin() and zi_negbin(), a fixed value for the dispersion parameter, which must be positive. Default is NULL to draw it along with everything else.
cut_alpha	numeric; for ordinal() and ordbeta(), the concentration of the induced-Dirichlet prior on the cutpoints. Default is 1, which says the category probabilities they imply are uniform over the simplex; larger values pull them toward equal shares. See Details.
reference	for multinomial(), the response category to hold as the reference, given as a single value naming one of the response's levels. Default is NULL, which with the logit link fits one forest per category instead and leaves the model unidentified, which makes the prior symmetric in the categories; see Details. The probit link is always written as contrasts against a reference, so there the default is the first level.
replicates	numeric; for multinomial("probit"), how many simulation draws to use for the category probabilities, which have no closed form. Default is 200. More is always better but resulting calculations will take longer.
nu, q	numeric; for dpm() and dpm_aft(), the degrees of freedom of the baseline's inverse-chi-square prior on a component's variance and the quantile of that prior placed at a rough estimate of the residual standard deviation. Defaults are 10 and .95, which are tighter than BART's own 3 and .90 because the mixture covers small errors with extra components rather than with one component's left tail.
k_s	numeric; for dpm() and dpm_aft(), how many units of the baseline's own scale the component means are allowed to reach out to. Must be positive. Default is 10, which places the marginal of a component mean so that it reaches the largest residual of a linear fit.

alpha	numeric; for dpm() and dpm_aft(), a fixed Dirichlet process concentration, which must be positive. Default is NULL to draw it.
max_clusters, psi	numeric; for dpm() and dpm_aft(), the largest number of mixture components thought plausible (2 or greater) and the shape of the taper toward it, which together set the prior on alpha. Defaults are NULL to use a tenth of the sample size, and .5.
num_bins	<i>Advanced.</i> numeric; for ph(), how many pieces the baseline hazard has, with the edges at evenly spaced quantiles of the observed times. Must be 2 or greater. Default is NULL to use about the cube root of the sample size, which is the order the Freedman-Diaconis rule gives for a histogram. This should not need to be set: in simulations the estimates were essentially unchanged from 4 bins to 100, and it is here for checking that rather than for tuning. See Details.
lambda_shape	numeric; for ph(), the shape of the gamma prior on each bin's baseline hazard, which must be positive. Its rate is drawn. Default is 1.
update_lambda	logical; for ph(), whether to draw the baseline hazards. Default is TRUE; FALSE holds them at their prior mean, which is for diagnosis rather than analysis.
phi	numeric; for Beta() and ordbeta(), a fixed value for the beta precision, and for tweedie() a fixed value for the dispersion. Must be positive. Default is NULL to draw it.
power	numeric; for tweedie(), the variance power p in $\text{Var}(y) = \phi\mu^p$, strictly between 1 and 2. Default is 1.5, and the value is held fixed there rather than drawn, because the power is weakly identified from data of the sizes this package is used on and a badly determined power drags the dispersion around with it. Pass NULL to draw it, which is worth doing only with a large sample and a real interest in the shape rather than the mean.

Details

Every family reduces to a scalar additive predictor, or to several of them, together with the first two derivatives of the log density with respect to each. That is the whole interface the sampler needs, which is why the set of available families is not restricted to the conditionally conjugate ones.

vignette("families", package = "bartisan") covers all of this at length: how to choose a family, how to choose among the links a family offers, and what each family is and is not for. What follows is the short version, and the points that could lead to output being misread.

The supported families and links are:

Family	Links	Additive predictors
gaussian()	identity	1
binomial()	logit, probit, cloglog	1
poisson()	log	1
negbin()	log	1
Gamma("log")	log; any other link is ignored	1
ordinal()	logit, probit, cloglog	1
multinomial()	logit, probit	one per category, or per non-ref
weibull_aft(), loglogistic_aft(), lognormal_aft()	none	1

<code>ph()</code>	<code>none</code>	1
<code>dpm_aft()</code>	<code>none</code>	1
<code>gaussian_ls()</code>	<code>identity</code>	2
<code>Gamma_ls()</code>	<code>log</code>	2
<code>zi_poisson()</code>	<code>log</code>	2
<code>zi_negbin()</code>	<code>log</code>	2
<code>Beta()</code>	<code>logit, probit, cloglog</code>	1
<code>ordbeta()</code>	<code>logit</code>	1
<code>tweedie()</code>	<code>log</code>	1
<code>dpm()</code>	<code>identity</code>	1

A family with more than one additive predictor fits one forest per predictor. Nuisance parameters are drawn alongside the trees and reported in `fit$aux`.

Setting link:

The links listed above are the ones the sampler evaluates in compiled code. Any other link is accepted for `gaussian()`, `binomial()`, `poisson()` and `Beta()`, and applied from R by composing the caller's inverse link with the family's own, with the chain rule carrying the derivatives back. So `binomial("cauchit")` works, as does any link object of the kind `stats::make.link()` returns. It costs a call into R for every leaf the sampler visits, and the leaf prior scale is calibrated for the compiled link.

Some single-predictor families are exceptions. `negbin()` takes "log" alone, so a link given to it is an error rather than a composition. `stats::Gamma()` accepts any link and fits none of them but "log": every other link is dropped with a message, because the ones base R offers have inverses that go non-positive over part of the line and the additive predictor is unconstrained. `custom_family()` is the route to a gamma response on another link.

A composed link whose inverse has a restricted range (`poisson("identity")`, `poisson("sqrt")`) will give non-finite densities for some predictors. Those proposals are rejected rather than breaking the chain, but they are wasted work and the fit is worse for it, so `bartisan()` says so when it starts. Prefer links whose inverse is defined on the whole line.

The families with more than one additive predictor, or whose link enters somewhere other than a single mean (`ordinal()`, `multinomial()`, the accelerated failure time families, `gaussian_ls()`, the zero-inflated families and `ordbeta()`), take only their listed links. `custom_family()` is the way to reach anything else.

Omitting family:

The family argument may be omitted, in which case it is read off the response's type and the choice reported with a message; `bartisan()` tabulates the lookup, since it is the function that takes the family argument. A count is read as `gaussian()` rather than `poisson()`, and a numeric response taking two values other than 0 and 1 as `gaussian()` rather than `binomial()`, both being modeling decisions rather than readings of the response's type.

Reading the Output:

`ordinal()` accepts a numeric response as well as an ordered factor, taking its sorted unique values as the categories, and that is a method rather than a fallback: the cutpoints absorb the marginal distribution of the response and the forest explains only the ordering, so nothing is assumed about the error distribution and the model for $P(Y \leq y \mid x)$ is invariant to any monotone transformation

of the response. Predict with `type = "mean"`. Bin the response onto twenty-odd quantiles first, since one cutpoint per distinct value is both far slower and slightly less accurate than twenty-five bins; `vignette("families")` works it through.

The cutpoints of `ordinal()` and `ordbeta()` carry an induced-Dirichlet prior, which puts the prior on the category probabilities the cutpoints imply rather than on the cutpoints themselves. The `cut_alpha` argument is its concentration, and it regularizes a thinly observed category without disturbing a well observed one.

A level of an ordered factor that nobody selected is kept rather than dropped, so a rating scale with an unused point is fitted on all of its points. A numeric response is read as the distinct values it takes, so such a scale has to be given as an ordered factor for the unused point to be modeled, and a warning says so. `vignette("families")` derives the prior and works both through.

`ordinal()` uses the cumulative-link parameterization of `MASS::polr()`, in which $P(Y \leq k) = F(c_k - \eta)$, so larger values of the additive predictor shift mass toward higher categories. Only the differences $c_k - \eta_i$ are identified, so one location has to be pinned: with three or more categories the draws are reported in the chart where the additive predictor has mean zero over the fitted sample and every cutpoint is free, which is the chart `polr()` reports in when its predictors are centered. With exactly two categories the single boundary is folded into the intercept instead, so a two-category response is exactly binary regression with the matching link and on the same scale. `cut1` is therefore a free parameter rather than a constant zero, which is a change from earlier versions.

`multinomial()` by default fits one forest per category and leaves the model unidentified, since adding any function of the predictors to every category's forest leaves the probabilities alone. The point of that parameterization is that the prior is then symmetric in the categories, and every identified quantity is recovered from the draws. Passing `reference` instead pins that category at zero and fits one fewer forest, giving log odds against it.

`multinomial("probit")` lets the latent utilities correlate, which a multinomial logit cannot express at all. Σ is normalized by the trace constraint $\text{tr}(\Sigma) = C$, and its lower triangle appears in `fit$aux` as `sigma11`, `sigma21` and so on. Read the fitted probabilities rather than the covariance: the correlations enter the likelihood only through orthant probabilities of a distribution whose location is a sum of trees, so a flexible mean absorbs much of the dependence they are meant to measure and they are weakly identified until the sample is large. The likelihood has no closed form, so it and every category probability are simulated with `replicates` draws, and the `augment` argument does not apply, the latent variables being the model rather than a rewriting of it.

`dpm()` is not a distribution but a way of not choosing one. It is DPMBART: a numeric response with the sum of trees for its mean, as `gaussian()` has, and a Dirichlet process mixture of normals for its errors instead of a single normal, so the error distribution comes out as whatever mixture the data ask for. `error_density()` gives that density, which is the object the method exists to produce.

It is the family to reach for by default on a numeric response, because it does not pay for its flexibility: on normal errors, where `gaussian()` is exactly right, it came out slightly ahead on both held-out error and log score at the same time to one decimal place, and on heavy-tailed, skewed and bimodal errors it was ahead by a great deal (on bimodal errors at a thousand observations, 0.050 against 0.154 in held-out RMSE, a factor of three, at the same time to a tenth of a second). So it is the family a numeric response gets when none is named. The reasons to prefer `gaussian()` are not statistical: it takes prior weights, which `dpm()` refuses, and it reports one interpretable sigma where `dpm()` has a mixture. It is also faster, by 1.4 times at a thousand observations. The vignette has the comparison.

Some properties of `dpm()` itself are worth knowing. It does not buy heteroskedasticity: the error distribution is flexible but it is the same distribution at every x , and `gaussian_ls()` is the family for a spread that depends on the predictors. And the additive predictor is the conditional mean, as it is for `gaussian()`: nothing in the model forces the mixture to be centered, so the sampler works in a chart where only the sum of the predictor and the error mean is identified, but reporting is done in the chart where the mixture has mean zero and the whole conditional mean sits on the predictor. `type = "link"` and `type = "response"` therefore agree exactly, and `fit$aux` reports the shift that was taken out as center rather than an error mean, which is zero by construction. Prior weights are refused, since a weight would have to be a multiplicity in the Dirichlet process. The gamma family puts the forest on the log mean and draws the shape, which acts as the inverse dispersion; it does *not* regress the shape on the predictors. `negbin()` and `ordbeta()` take `theta` and `phi` to fix their equivalents, and the gamma shape has no such argument because a caller who knows it is rare. The link is where the care is needed. Only `log` is compiled, and the base R default of `inverse` is the worst case for this sampler: its inverse maps a negative predictor to a negative mean, whose log is not a number, so the proposal is rejected, dozens of times per fit, and the fit is both slower and slightly worse for it. So write the link, or name the family as the string "Gamma", which resolves to the log link; base R's own function is left as base R defines it, so that attaching **bartisan** cannot change what `glm()` does. Any composed link whose inverse has a restricted range is reported when the fit starts.

The accelerated failure time families expect a right-censored response, supplied either as a `survival::Surv()` object or as a two-column matrix of times and event indicators. They model $\log T = \eta + \sigma\epsilon$ with ϵ a standard smallest extreme value, logistic or normal variate respectively, giving Weibull, log-logistic and log-normal survival times.

A contrast in the predictor is a log time ratio in all of them, and in `dpm_aft()` too, because with ϵ independent of x every quantile and both means of T scale by $e^{\Delta\eta}$ whatever shape the error has. The families differ in what e^η is on its own, each family pinning its error's location differently: the median of T for `loglogistic_aft()` and `lognormal_aft()`, the geometric mean for `dpm_aft()`, and the Weibull scale for `weibull_aft()`. Contrasts are unaffected by any of that; `vignette("survival")` tabulates the levels.

`weibull_aft()` is also the one family whose predictor carries a log *hazard* ratio, of $-\Delta\eta/\sigma$, alongside its log time ratio. That is a property of the smallest extreme value error rather than of the structural part it shares with the other three: it is the only error making an accelerated failure time model a proportional hazards model as well.

`ph()` is the proportional hazards alternative, with a piecewise-constant baseline: $\lambda(t \mid x) = \lambda_0(t) \exp(r(x))$, so its predictor is a log *hazard* ratio and the baseline is free to take any shape rather than the monotone one a Weibull imposes. The `num_bins` argument sets how many pieces, with the edges at evenly spaced quantiles of the observed times; the default is about the cube root of the sample size. The bin hazards are drawn from their exact gamma conditionals and reported as `lambda1`, `lambda2`, ... in `fit$aux`, together with the rate of their own prior. The predictor and the baseline are identified only jointly, so each draw is reported with the predictor centered over the fitted sample and the hazards scaled to match: a single value of the predictor is a log hazard ratio against a unit whose predictor sits at that average, and `lambda1`, `lambda2`, ... are that unit's hazards. Setting `update_lambda = FALSE` holds the baseline fixed, in which case the draws are reported uncentered and the hazards at the values they were fixed at.

Cox's *partial* likelihood cannot be used here: it couples observations through risk sets and so does not decompose into a sum over the observations reaching a leaf. The full likelihood of the piecewise-exponential model does decompose, and it approaches the partial likelihood as the bins shrink, which is how `ph()` reaches proportional hazards without it.

The `num_bins` argument should be left at its default. The estimates are insensitive to it, and what it does change is the effective number of parameters, which grows with the bin count and so matters for `loo()` and `waic()`. In the simulations vignette("survival") summarizes, the fitted survival curves were essentially unchanged from 4 bins to 100.

The three differ in cost, though not enough to decide a model on. `lognormal_aft()` and `loglogistic_aft()` impute each censored failure time above its censoring time, which makes their targets quadratic and is worth a large multiple of the speed; `weibull_aft()` needs no imputation because its likelihood already has a form the sampler can collapse to a single pass, but only under hard rules, which makes it the slowest of the three at the default gate.

`dpm_aft()` is the accelerated failure time model with the error distribution estimated rather than assumed: $\log T = m(x) + W$ with W a Dirichlet process mixture of normals constrained to mean zero, and censored log-times imputed. It is `dpm()`'s error model with censoring, so its predictor is the conditional mean of $\log T$ and a contrast in it is a log time ratio exactly as for the three parametric families above, `error_density()` reports the fitted error density, and prior weights are refused for the same reason `dpm()` refuses them.

Reach for it when the shape of the error is in doubt and there is no reason to assert one. It gains substantially where a fixed-error family would have been wrong and costs next to nothing where one would have been right, which is the property `dpm()` has against `gaussian()`; vignette("survival") summarizes the simulations behind that.

`gaussian_ls()` regresses the mean and the log standard deviation of a normal response on separate forests, so the variance is an unrestricted function of the predictors.

`Gamma_ls()` does the same for a gamma response: the first forest is the log mean, exactly as `Gamma("log")`'s is, and the second is the log dispersion, so the shape is $\exp(-\log_dispersion)$ at each observation rather than one value drawn for the whole sample. It relaxes the assumption that the coefficient of variation is constant, which a gamma with a single shape asserts, and giving its second forest an intercept-only formula puts that assumption back; see "Several additive predictors" above.

Note that the mean forest is quadratic in neither family, but it takes the same cheap exponential form `Gamma("log")`'s does, while the dispersion forest takes the general path, so the cost sits in the second forest and it is worth giving that one fewer trees.

`zi_poisson()` and `zi_negbin()` are zero-inflated counts. Both parts get their own forest: the first predictor is the log mean of the count component and the second the log odds that an observation is a structural zero, so the excess-zero mechanism is free to depend on the predictors. The two are reported as the count and zero predictors.

`Beta()` is beta regression for a response strictly inside the unit interval: a forest on the link of the mean, and a precision drawn alongside it. A response *at* either endpoint has no beta density, so it is an error rather than something to nudge inward.

`ordbeta()` is ordered beta regression, for a response on the closed unit interval with point masses at zero and one. One predictor drives both the probability of landing on an endpoint, through a pair of cutpoints as in an ordinal model, and the mean of the beta density in between. Because the predictor also enters the beta mean it is identified, so unlike `ordinal()` both cutpoints are drawn. Choose between them on whether the response can reach a boundary, not on whether it happens to in the sample: the two ask different questions, and `ordbeta()` fitted to a response with no boundary observations leaves its cutpoints with nothing to identify them.

`tweedie()` is the compound Poisson-gamma, for a non-negative response with a point mass at zero and a continuous positive part, which is the shape of spending, rainfall, insurance claims and earnings. It is the analogue of `ordbeta()` at the other end: one predictor again drives both

parts, but through the mean rather than through a cutpoint, since $\mu = \exp(\eta)$ and $\text{Var}(y) = \phi\mu^p$ together fix the probability of a zero at $\exp(-\mu^{2-p}/(\phi(2-p)))$. That makes it a single process and is also its restriction: the share of zeros has no level of its own, so a response whose zeros are more or less common than its mean implies wants a two-part model instead, which `zi_poisson()` and `zi_negbin()` are for counts and which `custom_family()` can supply for anything else.

Because the mean is exactly $\exp(\eta)$, a counterfactual mean through **marginalEffects** needs nothing beyond the forest, unlike a two-part model where it has to be recombined across predictors; and the fit is comparable with a `poisson()` or `Gamma("log")` fit of the same response, since all three put the same quantity on the same scale.

Several Additive Predictors:

Most families model one parameter with one forest. Some model several, and then every argument that could mean something different for each of them may be given per forest, keyed by the names below or positionally; `bartisan_control()` states the recycling rule and lists which arguments it covers, and the formula argument is among them, so a forest can have predictors of its own.

The first forest is always the main parameter, the one a single-forest family would have on its own. This table is the canonical list of the names, which `vignette("families")` reproduces:

Family	Forests, in order
<code>gaussian_ls()</code>	mean, log_sd
<code>Gamma_ls()</code>	mean, log_dispersion
<code>zi_poisson()</code> , <code>zi_negbin()</code>	count, zero
<code>custom_family(num_predictors = k)</code>	eta1 ... etak
<code>multinomial("logit")</code>	one per level, or per non-reference level
<code>multinomial("probit")</code>	one per non-reference level, named for its contrast
everything else	eta

mean is the mean and log_sd is the logarithm of the standard deviation, which is the scale the forest works on. `Gamma_ls()`'s mean is the log mean, as `Gamma("log")`'s single forest is, and its log_dispersion is the logarithm of the dispersion, so that in both location-scale families a larger second predictor means more spread. count is the linear predictor of the count component and zero that of the inflation component. A custom family's nuisance parameters are not on this list: they are carried as trailing forests pinned to a single leaf, and nothing about them is set per forest.

A forest whose formula names no predictor is a constant. `~ 1` leaves that forest nothing to split on, so every tree in it is a stump and the parameter is one drawn scalar. Every family here that takes more than one formula accepts that, which makes the distinction between a nuisance parameter and an empty forest a thin one: `gaussian_ls()` with `~ 1` on its scale is `gaussian()`, `Gamma_ls()` with `~ 1` is `Gamma("log")`, and `zi_poisson()` with `~ 1` on its inflation part is the zero-inflated Poisson with a single structural-zero probability. Note that the scalar is drawn under the leaf prior rather than under the prior the corresponding built-in family puts on its nuisance parameter, so the two agree closely rather than exactly. The multinomial families are the exception, for the reason given below.

So, for a location-scale model with a smaller scale forest and a restricted set of predictors for it:

```
bartisan(list(y ~ x1 + x2 + x3, log_sd = ~ x1), data = d,
         family = gaussian_ls(), num_trees = c(mean = 50, log_sd = 10))
```

The multinomial families are the exception. Their forests are the levels of one categorical parameter and act together rather than describing separate components of the response distribution, so there is nothing a caller could mean by giving one level a different prior or a different set of predictors from another. Every argument applies to all of their forests at once, and more than one value is an error rather than a silent recycling.

Supplying a Likelihood:

When none of the families above is the right one, `custom_family()` takes the log density itself, as an R function, and fits the model that goes with it. Its page has the details, including what a log density cannot supply: a posterior predictive distribution and a fitted mean.

Value

A `<bartisan_family>` object, which is a list containing at least the elements `family` and `link` and which inherits from `family`, so that `bartisan()` recognizes it wherever it recognizes an ordinary `stats::family` object.

See Also

`bartisan()` for fitting a model with one of these families; `custom_family()` for a likelihood none of them covers; `error_density()` for the error distribution a `dpm()` or `dpm_aft()` fit estimates; `vignette("families", package = "bartisan")` for the long form

Examples

```
data("rhc")
set.seed(123)

# A right-censored response, given as the time and the
# event indicator, with the error distribution estimated
# rather than assumed
surv_model <- cbind(days, death) ~ rhc + age + sex +
  race + edu + aps + meanbp + resp + hema + pafi +
  paco2 + crea + surv2m + card

fit <- bartisan(surv_model, data = rhc,
               family = dpm_aft(), num_trees = 10,
               num_burn = 50, num_draws = 50)

# The shape the errors came out
head(error_density(fit))

# The same response under proportional hazards, whose
# predictor is a log hazard ratio and whose baseline is
# free to take any shape
bartisan(surv_model, data = rhc, family = ph(),
         num_trees = 10, num_burn = 50, num_draws = 50)

# An unordered response, with one forest per category
# and a prior that is symmetric in them
bartisan(race ~ rhc + age + sex + edu + aps + meanbp +
```

```

      resp + hema + pafi + paco2 + crea + surv2m +
      card,
    data = rhc, family = multinomial(),
    num_trees = 10, num_burn = 50, num_draws = 50)

# A link the engine does not compile, applied from R
bartisan(death ~ rhc + age + sex + race + edu + aps +
  meanbp + resp + hema + pafi + paco2 + crea +
  surv2m + card,
  data = rhc, family = binomial("cauchit"),
  num_trees = 10, num_burn = 50, num_draws = 50)

```

bartisan-interop	<i>Interfaces to other packages</i>
------------------	-------------------------------------

Description

Methods that let a <bartisan_fit> object be used by the packages that assess model fit. There is nothing to set up: load the other package and call its function on the fit.

Usage

```

## S3 method for class 'bartisan_fit'
posterior_predict(
  object,
  newdata = NULL,
  iterations = NULL,
  offset = NULL,
  weights = NULL,
  ...
)

## S3 method for class 'bartisan_fit'
posterior_epred(object, newdata = NULL, ...)

## S3 method for class 'bartisan_fit'
posterior_linpred(object, transform = FALSE, newdata = NULL, ...)

## S3 method for class 'bartisan_fit'
log_lik(object, newdata = NULL, ...)

## S3 method for class 'bartisan_fit'
simulate(object, nsim = 1, seed = NULL, ...)

## S3 method for class 'bartisan_fit'
fitted(object, type = "response", ...)

```

```

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

## S3 method for class 'bartisan_fit'
weights(object, ...)

## S3 method for class 'bartisan_fit'
sigma(object, ...)

## S3 method for class 'bartisan_fit'
prior_summary(object, ...)

## S3 method for class 'bartisan_prior_summary'
print(x, digits = 3L, ...)

## S3 method for class 'bartisan_fit'
loo(x, scale = NULL, ...)

## S3 method for class 'bartisan_fit'
waic(x, scale = NULL, ...)

## S3 method for class 'bartisan_fit'
kfold(x, K = 10, folds = NULL, scale = NULL, save_fits = FALSE, ...)

## S3 method for class 'bartisan_fit'
pp_check(object, type = "dens_overlay", ndraws = 10, ...)

## S3 method for class 'bartisan_fit'
as_draws(x, eta = TRUE, ...)

## S3 method for class 'bartisan_fit'
r2_posterior(model, verbose = TRUE, ...)

## S3 method for class 'bartisan_fit'
r2(model, ...)

## S3 method for class 'bartisan_fit'
model_performance(model, metrics = "all", verbose = TRUE, ...)

```

Arguments

<code>object, model, x</code>	a <bartisan_fit> object; the output of a call to bartisan() .
<code>newdata</code>	optional; a data frame at which to predict. Default is NULL to use the data the model was fit to. Missing predictor values are allowed in the columns that had them when the model was fit, since only those columns' splitting rules carry an answer for one; a missing value anywhere else is an error. See section <i>Missing Predictor Values</i> in the Details of bartisan() .
<code>iterations</code>	numeric; optional indices of the stored draws to use, between 1 and the number

	of draws the fit retains. Default is NULL to use all of them.
offset, weights	an offset and prior weights for newdata, as in <code>predict.bartisan_fit()</code> . Defaults are NULL to use those the model was fit with. For a binomial response the weights are the numbers of trials, and so give the denominator of a replicate outcome; they must be given alongside newdata when the model was fit with more than one trial, since the number of trials is not a function of the predictors and cannot be reconstructed.
...	further arguments, passed to whatever the method calls (the bayesplot check from <code>pp_check()</code> , <code>loo::loo()</code> and <code>loo::waic()</code> from <code>loo()</code> and <code>waic()</code> , and <code>predict.bartisan_fit()</code> from the rest) and ignored where there is nowhere to pass them.
transform	logical; for <code>posterior_linpred()</code> , whether to map the predictor through the inverse link, as <code>posterior_epred()</code> does. Default is FALSE.
nsim, ndraws	numeric; the number of posterior draws to use, chosen at random from the retained ones. Defaults are 1 for <code>simulate()</code> and 10 for <code>pp_check()</code> . A <code>ppc_loo_*</code> check uses every retained draw whatever this is set to, and says so; see Details.
seed	optional seed, set with <code>set.seed()</code> before drawing and restored afterwards, following the <code>stats::simulate()</code> convention. Default is NULL to leave the stream alone.
type	string; for <code>fitted()</code> , the prediction scale, passed to <code>predict.bartisan_fit()</code> ; default is "response". For <code>pp_check()</code> , the name of the bayesplot check to run without its <code>ppc_</code> prefix, so that "dens_overlay" (the default) calls <code>bayesplot::ppc_dens_overlay()</code> ; <code>bayesplot::available_ppc()</code> lists them.
digits	integer; for <code>print()</code> on the output of <code>prior_summary()</code> , how many digits to round the prior's scales to. Default is 3.
scale	string; for <code>loo()</code> , <code>waic()</code> and <code>kfold()</code> on a survival fit, the measure to report the pointwise densities with respect to: "time" for the density of T and "log_time" for the density of $\log T$. Default is NULL to use the family's own, which is $\log T$ for the accelerated failure time families and T for <code>ph()</code> . A fit already on the scale named is left alone, so naming one scale for every model in a comparison is enough. See Details.
K	numeric; for <code>kfold()</code> , how many folds to split the sample into. Default is 10. Ignored when folds is given.
folds	optional; for <code>kfold()</code> , an integer vector of one fold number per observation, as <code>loo::kfold_split_random()</code> and its relatives return. Default is NULL to draw them at random. Supply them to stratify, to group, or to score two models on the same split.
save_fits	logical; for <code>kfold()</code> , whether to keep the K refits in the result's <code>fits</code> element. Default is FALSE, since each is a whole fit.
eta	for <code>as_draws()</code> , which columns of the additive predictor to carry into the draws array alongside the scalar parameters, given as either a logical value or a numeric vector. Default is TRUE, which takes a representative ten spread across the range of the fitted function; FALSE takes none, and a numeric vector takes those observations. The default takes a handful rather than all of them because

	there is one column per observation, and an array with thousands of them is not something <code>posterior::summarise_draws()</code> or a trace plot can be pointed at. The predictor is the quantity whose convergence usually matters, and the one <code>diagnose()</code> reports on, so it is included by default.
verbose	logical; whether to report problems that do not stop the computation, such as a family that has no mean and so no Bayesian R^2 . Default is TRUE.
metrics	character; for <code>model_performance()</code> , which fit statistics to report. Allowable options include "all" (the default), "ELPD", "LOOIC", "WAIC", "R2", "RMSE", and "SIGMA", and a vector of them selects several.

Details

Available Methods:

`rstantools::posterior_predict()` draws replicate outcomes from the fitted model, `rstantools::posterior_epred()` gives their mean and `rstantools::posterior_linpred()` the additive predictor, following the **rstantools** conventions that **brms** and **rstanarm** follow, and `stats::simulate()` is the same thing in the shape base R expects. `rstantools::log_lik()` returns the draws-by-observations matrix of log-likelihood contributions, which `loo::loo()` and `loo::waic()` need. `pp_check()` runs any of the **bayesplot** posterior-predictive checks on the fit, `performance::model_performance()` collects the fit statistics in one table, `performance::r2()` gives the Bayesian R^2 , and `posterior::as_draws()` hands the scalar parameters to `posterior::summarise_draws()` or to the **bayesplot** MCMC diagnostics. `rstantools::prior_summary()` writes out every prior the fit was given, on the scale it was given on, which is the companion to `prior_only = TRUE` in `bartisan()`: one says what the prior is and the other what it implies about the outcome. `stats::fitted()`, `stats::residuals()`, `stats::weights()` and `stats::sigma()` do what they do for a `glm`, which is also most of what **insight** needs to make the fit legible to the **easystats** packages.

Accuracy of the Leave-One-Out Approximation:

`loo::loo()` estimates the leave-one-out predictive density by importance sampling from the full-data posterior, and the estimate is trustworthy only when the importance weights have a finite variance, which the Pareto k diagnostic reports on. A forest is a flexible function of the predictors, so the worry is that one observation carries enough influence over the leaves it lands in that dropping it cannot be approximated from the fit in hand. In practice it rarely does: the leaf prior shrinks every leaf toward zero and the fit is a sum over many trees, so no single observation dominates.

The warning **loo** prints there is therefore worth reading rather than expecting, and the exceptions that do turn up are usually about the likelihood rather than the trees. When it names a handful of observations, those are the influential ones, and refitting without them shows how badly they are predicted. A log score on data the model has not seen is available directly:

```
predict(fit, newdata = held_out, type = "density", log = TRUE)
```

Cross-Validation (`kfold()`):

Where `loo()` estimates the leave-one-out density from one fit, `kfold()` splits the sample, refits K times, and scores each part under a fit that never saw it. That costs K fits and owes nothing to an approximation, which makes it the thing to reach for when the Pareto diagnostics say the weights cannot be trusted.

It returns a `<kfold>` object that `loo::loo_compare()` accepts beside a `<loo>` one, so two models can be compared on one split by passing the folds from the first to the second:

```

folds <- loo::kfold_split_random(K = 10, N = nobs(fit))

loo_compare(list(full = kfold(fit, folds = folds),
                             small = kfold(other, folds = folds)))

```

The refits run under a future plan when one is set, and one `set.seed()` reproduces them either way. Each is refitted from the original call, so the data argument has to still name the data the fit was made from. Prior weights, an offset, and a propensity score supplied to `bcf()` are carried into both the refits and the held-out scores. `p_kfold` is the gap between what the model predicts for an observation it was fitted to and what it predicts for the same one held out, which is the price of having used it. `vignette("comparison")` reads an example.

Setting scale for Survival Families:

The accelerated failure time families report the density of $\log T$ and `ph()` the density of T . Both are correct for the model that produced them, and they differ by the Jacobian of the change of variable, so a log score taken across that boundary is off by $\sum \log t$ over the events, which can reverse which family looks better.

The scale argument puts them on one measure, and reads the same from either side, since a fit already on the scale named is returned untouched:

```

loo_compare(list(aft = loo(aft_fit, scale = "time"),
                 ph = loo(ph_fit, scale = "time")))

```

It is left to the caller rather than applied automatically so that `loo()` keeps reporting the model's own predictive density and keeps agreeing with `log_lik()`. Censored observations are not adjusted, a survival probability being a probability on either scale. `vignette("comparison")` works through the comparison and `vignette("survival")` through the families.

Posterior Predictive Checks:

The seven `ppc_loo_*` checks reweight the replicates towards the leave-one-out predictive, so they need those weights. `pp_check()` computes them from the fit's own pointwise log likelihood and passes them on, and the `ndraws` argument does not apply to those checks, because the weights and the replicates have to line up draw for draw; supplying `lw` or `psis_object` takes over from it.

The two calibration checks are the ones to reach for when the response is binary, since the default check compares two distributions that can only take two values. Setting `type = "loo_calibration"` gives the honest one, holding each observation out of the probability it is judged against, where `type = "calibration"` is its in-sample counterpart and reads optimistically. Those two and a binned residual plot (`type = "error_binned"`) are about the predicted probabilities rather than replicate outcomes, so they are passed the mean of the predictive distribution instead of a draw from it.

The Scale of a Posterior Predictive Draw:

Replicate outcomes are on the scale the likelihood was written on, which is the scale `bartisan()` stored the response on. A binomial response is a proportion, so binary data come back as 0 and 1, and data given as two columns or with prior weights come back as a fraction of the trials. A response with categories comes back as an integer category index, since a matrix cannot hold a factor; `fit$levels` names them, and `stats::simulate()` returns factors instead, its result being a data frame. An accelerated failure time response comes back as an event time rather than a log time, and the predictive distribution of the outcome knows nothing of the censoring that may have

hidden it, so `pp_check()` says so when replicates are compared against censored observations. A `custom_family()` fit supplies a log density and no way to draw from it, so these methods error on one.

AIC, BIC and Normality Checks:

`stats::AIC()` and `stats::BIC()` need a count of parameters, which a forest does not have, since the number of leaves is itself drawn from the posterior; there is accordingly no `logLik()` method. `loo::loo()` and `loo::waic()` are the corresponding quantities for a model like this one, computed from the posterior rather than from a parameter count. For the same reason `performance::check_normality()` and `performance::check_outliers()`, which ask for a likelihood-ratio test and for Cook's distance, are unavailable, where `performance::check_predictions()` works through `stats::simulate()`.

The Bayesian R-Squared:

`performance::r2()` returns, per draw, the variance of the fitted means across observations divided by that variance plus the variance of the residuals. Being a per-draw quantity it has a posterior, which is why it is reported with an interval and why it can fall as the model is made more flexible. It needs a mean, so it is available for every family except `ordinal()` and `multinomial()`.

Value

`kfold()` returns a `<kfold>` object, a list whose `estimates` holds `elpd_kfold`, `p_kfold` and `kfoldic` with their standard errors, whose `pointwise` holds the same three per observation, and whose `folds` records the split; setting `save_fits = TRUE` adds the K refits in `fits`.

`posterior_predict()`, `posterior_epred()`, `posterior_linpred()` and `log_lik()` return a matrix of draws by observations. `simulate()` returns a data frame of one column per replicate. `loo()` and `waic()` return the `<loo>` and `<waic>` objects those functions produce, and `model_performance()` a one-row data frame of class `<performance_model>`. `as_draws()` returns a `<draws_array>` of iterations by chains by parameters.

`prior_summary()` returns a `<bartisan_prior_summary>` object, a list whose `forests` is a data frame of one row per additive predictor and one column per setting the prior is made of, whose `estimated` says in the same shape which of them were drawn rather than held, and whose `family` holds the family's own parameters with the prior each was given. `random` and `response` record the group-intercept scale and what was read off the response.

The accessors return what their names suggest.

See Also

`predict.bartisan_fit()` for the predictions these methods are built on; `diagnose()` for the convergence and mixing diagnostics; `bartisan-marginaleffects` for reading effects off a fit; `vignette("diagnostics")` for the fuller treatment

Examples

```
data("rhc")
set.seed(123)

model <- death ~ rhc + age + sex + race + edu + aps +
  meanbp + resp + hema + pafi + paco2 + crea + surv2m +
```

```

card

fit <- bartisan(model, data = rhc, num_trees = 10,
               num_burn = 50, num_draws = 50, chains = 2,
               verbose = FALSE)

# Replicate outcomes, one per draw per observation,
# whose mean `fitted()` reports
yrep <- rstantools::posterior_predict(fit)
epred <- rstantools::posterior_epred(fit)
range(colMeans(epred) - fitted(fit))

# Pointwise log likelihood, and the fit statistics
# built on it
loo::waic(rstantools::log_lik(fit))

# Every prior the fit was given, on the scale it was
# given on
rstantools::prior_summary(fit)

# Whether replicate outcomes look like the observed ones
if (rlang::is_installed("bayesplot")) {
  bayesplot::pp_check(fit, type = "bars")
}

# The scalar parameters and a spread of the predictor,
# as a draws array
if (rlang::is_installed("posterior")) {
  posterior::summarise_draws(posterior::as_draws(fit))
}

```

bartisan-marginaleffects

Counterfactual estimands with marginaleffects

Description

Methods that let a `<bartisan_fit>` object be used by the **marginaleffects** package, which computes predictions, comparisons and slopes (and hypothesis tests on any of them) from the posterior draws. There is nothing to set up: load **marginaleffects** and call its functions on the fit.

Usage

```

## S3 method for class 'bartisan_fit'
formula(x, ...)

## S3 method for class 'bartisan_fit'
terms(x, ...)

```

```
## S3 method for class 'bartisan_fit'
model.frame(formula, ...)

## S3 method for class 'bartisan_fit'
nobs(object, ...)

## S3 method for class 'bartisan_fit'
family(object, ...)

## S3 method for class 'bartisan_fit'
get_predict(model, newdata = NULL, type = NULL, ...)

## S3 method for class 'bartisan_fit'
get_group_names(model, ...)

## S3 method for class 'bartisan_fit'
get_coef(model, ...)

## S3 method for class 'bartisan_fit'
set_coef(model, coefs, ...)

## S3 method for class 'bartisan_fit'
get_vcov(model, ...)

## S3 method for class 'bartisan_fit'
get_data(x, ...)
```

Arguments

...	further arguments. values, iterations, offset, weights, log and times are passed on to <code>predict.bartisan_fit()</code> ; anything else is ignored, since marginaleffects puts arguments of its own here too. Note that marginaleffects warns that it does not recognize values, which is expected (it is this package's argument, not one of its own), and the value is used regardless.
model, x, object, formula	a <bartisan_fit> object; the output of a call to <code>bartisan()</code> . The last is named formula only because <code>stats::model.frame()</code> names its first argument that way.
newdata	optional; a data frame at which to evaluate the fit. Default is NULL to use the data the model was fitted to, which is retained in the fit for this purpose.
type	string; the scale to work on. Allowable options include "response" (the default), "link", "prob", "mean", "stdlv", "survival", and "hazard". "response" is the fitted mean, "link" the additive predictor, "prob" the per-category probabilities of a categorical family, "mean" the mean of a categorical response with its labels read as numbers, "stdlv" the standardized latent variable of an ordinal fit, and "survival" and "hazard" the survival function and the hazard at the times given in times; all but the first two are described under

`predict.bartisan_fit()`. An ordinal() or multinomial() response has no single mean, so "response" gives "prob" for those families, while a binomial fit reports the probability of its second level, as `stats::glm()` does, and "prob" asks for both of its columns. "probs", "lp", "lv" and "surv" are accepted as aliases for "prob", "link", "link" and "survival", since those are the names the same quantities go by for other ordinal fits in **marginaleffects**.

coefs ignored; a forest has no coefficient vector.

Details

Uncertainty:

A forest has no coefficient vector and no variance-covariance matrix, so the delta method **marginal-effects** uses for a frequentist model has nothing to work with. It has the posterior draws instead: every estimand is computed by pushing all of the draws through the same transformation and summarizing at the end, so an interval is a posterior quantile rather than a normal approximation, and a nonlinear estimand needs no approximation at all. This is the same path **marginaleffects** takes for **brms** and **rstanarm** fits.

One consequence is worth knowing: **marginaleffects** centers a posterior at its median, where `predict.bartisan_fit()` reports its mean. The two summarize the same draws, so a difference between them is the skewness of the posterior rather than a disagreement.

Contrasts of Exactly Zero:

`avg_comparisons()` reporting an estimate of exactly 0 is the most common surprise here, and it comes from the prior and the posterior summary acting together.

The posterior of a contrast has an atom at exactly zero. In any draw where no tree splits on the variable being contrasted, the fit does not depend on that variable at all, so the two counterfactual predictions are identical to the last bit and their difference is exactly zero. The Dirichlet sparsity prior, set by the sparsity argument of `bartisan_control()`, makes those draws common, since dropping a weak predictor from every tree is the purpose of a variable selection prior. And **marginaleffects** centers a posterior at its median, so once the atom holds more than half the mass the reported estimate is exactly zero however large the rest of the posterior is: the median can land on the atom while the posterior mean and the upper limit of the interval are both far from zero.

There are a few things worth doing about it, in the order given. Look at `prop_used` in `summary()`, the posterior probability that the predictor appears anywhere in the forest, the quantity the zero reports. Ask for the mean instead, with `options(marginaleffects_posterior_center = mean)`, which is the summary `predict.bartisan_fit()` reports and the one that behaves sensibly against an atom. Reconsider the sparsity prior, since setting `sparsity = FALSE` removes the atom almost entirely where a larger `num_trees` leaves it in place; that is a modeling choice rather than a fix, and it is the right one when a contrast on a particular predictor is the estimand. And run several chains and compare them, because the variable selection state mixes slowly and a single chain can look much more settled than the posterior is. `vignette("effects")` works the choice through.

Slopes and the Predictor Transform:

A slope is a numerical derivative, and taking one requires the fitted function to be differentiable in the predictor as the caller supplies it. Whether it is depends on `x_transform` in `bartisan_control()`, because the fit is a smooth function of the transformed predictor rather than of the original one.

Setting `x_transform = "range"` suits slopes, and refitting with it is worthwhile when a derivative is the quantity being reported rather than a prediction. It is the only affine map of the three, so it

is the only one through which a slope of the fit is a slope of the original predictor. The default, "smoothcdf", carries an estimated density along with it and lands close; "quantile" maps each predictor through a step function and so has no derivative to take, and the number returned there is a property of the step size rather than of the fit. `vignette("effects")` shows what each does as the step shrinks.

None of this affects `marginaleffects::predictions()` or `marginaleffects::comparisons()`, which evaluate the fit at two points a substantive distance apart rather than dividing by a vanishing one.

The Usual Survival Estimand:

For a survival family, the estimand is usually a contrast in survival at a horizon rather than in the predictor. `type = "survival"` with `times` gives it:

```
# The difference in one-year survival between treated and untreated
avg_comparisons(fit, variables = "trt", type = "survival", times = 1)
```

Several times can be given in one call, in which case each becomes its own group in the output, labeled by the time as text, so `as.numeric()` on the group column recovers the times for drawing a curve. **marginaleffects** checks the dots against a whitelist of its own, hardcoded per model class, so it warns that it does not recognize times while passing it through, as the warning says. There is no hook for registering an argument with it, so the warning is expected and the result is correct.

Setting `type = "hazard"` with the same times contrasts the hazard at those times instead, so `avg_comparisons()` averages each unit's hazard difference at a horizon, and `comparison = "ratio"` gives the ratio of the averaged hazards. Note that an average of hazards is not the hazard of the average survival curve: that one weights each unit by its probability of having survived to the time, and it comes from `type = "survival"` rather than from here.

Setting `type = "link"` contrasts the additive predictor instead. For `ph()` the contrast at each unit is the log hazard ratio at that unit's covariates, and for the accelerated failure time families it is the log time ratio, so `avg_comparisons()` averages conditional ratios over the units; the average of conditional log hazard ratios is not the marginal one, since a hazard ratio is noncollapsible.

Prediction Types:

Setting `type = "link"` needs a family with a single additive predictor, since there is no one link to be talking about in `gaussian_ls()`, the zero-inflated families or `multinomial()`. Those families work on the response scale, which is one number per observation whatever the family, and `multinomial()` and `ordinal()` work on the probability scale, which gives one group per category. `predict.bartisan_fit()` reaches a particular predictor of a multi-predictor family, and computes `type = "class"` and `type = "density"`, which are not one number per observation that an average could be taken of.

Extrapolation is worth keeping in mind for `comparisons()`: a forest is constant outside the range of the predictor it was fitted on, so a contrast that steps a predictor beyond that range reports the boundary value rather than an extrapolated one.

Value

`get_predict()` returns a data frame with columns `rowid`, `group` and `estimate`, carrying the draws in a "posterior_draws" attribute of observations by draws, which is the interface **marginaleffects** documents. The other methods exist to satisfy the generic and return what their names suggest.

See Also

`predict.bartisan_fit()` for the prediction scales these estimands are computed on; `bartisan_control()` for sparsity and `x_transform`; `bartisan-interop` for the methods that let other packages assess the fit

Examples

```
data("rhc")
set.seed(123)

# Whether a patient died, with the treatment and every
# covariate recorded on admission as candidate
# predictors
model <- death ~ rhc + age + sex + race + edu + aps +
  meanbp + resp + hema + pafi + paco2 + crea + surv2m +
  card

fit <- bartisan(model, data = rhc, num_trees = 10,
  num_burn = 50, num_draws = 50)

# The effect of catheterization on the probability of
# death, as an average contrast of counterfactual
# predictions
marginaleffects::avg_comparisons(fit, variables = "rhc")

# The same contrast within each sex, and a test that the
# two are equal
marginaleffects::avg_comparisons(fit, variables = "rhc",
  by = "sex",
  hypothesis = ~pairwise)

# Centering each posterior at its mean rather than its
# median, which is the summary `predict()` reports
op <- options(marginaleffects_posterior_center = mean)
marginaleffects::avg_comparisons(fit, variables = "rhc")
options(op)
```

bartisan_control

Sampler and prior settings for bartisan()

Description

Collects the tuning constants of the sampler and the hyperparameters of the tree prior, for passing to the control argument of `bartisan()`. The defaults follow Linero (2025) and, for the soft decision rules, Linero and Yang (2018), and are meant to be usable without adjustment.

Usage

```

bartisan_control(
  chains = 1L,
  num_burn = 200L,
  num_draws = 800L,
  num_thin = 1L,
  num_trees = 50L,
  gate = "smoothstep",
  sparsity = FALSE,
  split_prior = NULL,
  x_transform = "smoothcdf",
  k = 2,
  bandwidth = 0.1,
  share_sparsity = FALSE,
  categorical = "subset",
  augment = TRUE,
  gamma = 0.95,
  beta = 2,
  sigma_mu = NULL,
  update_sigma_mu = TRUE,
  sigma_mu_ramp = 0.25,
  update_tau = TRUE,
  update_bandwidth = TRUE,
  bandwidth_every = 1L,
  alpha = NULL,
  alpha_scale = NULL,
  alpha_shape_1 = NULL,
  alpha_shape_2 = NULL,
  update_s = NULL,
  update_alpha = NULL,
  verbose = FALSE,
  num_print = 100L,
  block_eval = FALSE,
  exact_quadratic = TRUE,
  generic_accumulate = FALSE
)

```

Arguments

chains	numeric; how many independent chains to run. Default is 1. The draws are pooled, and <code>diagnose()</code> computes split-R-hat from them on request rather than the fit carrying it. With the future.apply package installed the chains run in parallel under whatever backend <code>future::plan()</code> has set. One <code>set.seed()</code> before the call reproduces the run either way, since each chain gets its own L'Ecuyer stream. See <code>vignette("diagnostics")</code> .
num_burn	numeric; the number of warmup iterations to discard. Default is 200, which measurement finds ample for the samplers here. Warmup is where the trees grow into the data and the hyperparameters find their scale, so raising it buys

	convergence rather than precision: increase it when <code>diagnose()</code> 's <code>rhat</code> says the chains have not agreed, and for a <code>dpm()</code> fit, whose component-count state settles more slowly than a forest does. See Details.
<code>num_draws</code>	numeric; the number of draws to keep. Default is 800. Every estimate and interval is computed from these, so raising it narrows Monte Carlo error; increase it when <code>ess_bulk</code> or <code>ess_tail</code> is small relative to what the reported quantity needs.
<code>num_thin</code>	numeric; keep one draw in every <code>num_thin</code> after warmup. Default is 1 to keep every draw. Thinning discards draws to make the kept ones less correlated, which costs information and is worth it only to hold down the memory a long chain would otherwise take: for a given amount of computing, more draws beat fewer less-correlated ones, so it is best to avoid setting <code>num_thin</code> .
<code>num_trees</code>	numeric; the number of trees, as one number for every forest or one per additive predictor. Default is 50. A family with more than one additive predictor takes a vector, which can be worth using because the forests are neither equally expensive nor in need of equal capacity. See Details.
<code>gate</code>	string; the shape of the decision rule at each node. Allowable options include "smoothstep" (the default), "smootherstep", "logistic", and "hard" (or equivalently "step"). "hard" gives the step functions of standard BART; the other three give soft rules, in which every observation reaches every leaf with some weight and the fit is smooth. Soft rules are slower but improve predictive performance and yield smooth fits. Which soft gate is chosen matters much less than that one is. See Details.
<code>sparsity</code>	logical or string; the prior on which predictors are split on. FALSE (the default) gives every predictor the same splitting probability, which is classic BART, and TRUE is the Dirichlet sparsity prior of Linero (2018) (i.e., DART), which can drop a predictor from the forest entirely. The strings "none", "weak", "moderate", and "strong" name four strengths, "none" equal to FALSE and "moderate" to TRUE. Setting <code>sparsity = TRUE</code> is for when there are many predictors, when only a few of them are expected to matter, or when the goal is to identify the predictors the model needs; it is not recommended when an effect is being estimated. See Details. Note that this sets <code>update_s</code> , <code>update_alpha</code> , <code>alpha_shape_1</code> and <code>alpha_shape_2</code> together, and that supplying any of those overrides it.
<code>split_prior</code>	numeric; a named vector of relative prior weights on the predictors, keyed by their names in the formula. A predictor not named gets a weight of 1, and the probability of splitting on one is its weight over the total. Weights must be finite and nonnegative; zero keeps the predictor out of every tree, and naming a predictor the model does not have is an error. Default is NULL to weight every predictor equally. Note that setting this overrides <code>sparsity</code> ; see Details.
<code>x_transform</code>	string; how numeric predictors are mapped to $[0, 1]$ before any rule sees them. Allowable options include "smoothcdf" (the default), "quantile", and "range". The map decides where a cutpoint can fall, the cutpoint prior being uniform on a node's live range in the transformed coordinate, and for soft rules it is also the scale bandwidth is measured on. "smoothcdf" smooths the empirical distribution function with an Epanechnikov kernel, "quantile" uses the empirical distribution function itself, and "range" rescales linearly. Note that

	"range" is the one to use when a slope is the quantity of interest, being the only affine map of the three; see vignette("effects"). Where the other two fail, and why the default is the default, is in vignette("implementation").
k	<i>Advanced.</i> numeric; controls the leaf prior. The prior standard deviation of a forest is $3 / k$ times the natural scale of its additive predictor, so larger values shrink the fit harder toward the intercept-only model. Default is 2.
bandwidth	<i>Advanced.</i> numeric; the prior mean of the gate bandwidth of a soft rule, on the scale of the transformed predictors, which lie in $[0, 1]$. Smaller values approach hard rules. Default is .1. Ignored when gate = "hard". See also update_bandwidth and bandwidth_every.
share_sparsity	<i>Advanced.</i> logical; for a family with more than one additive predictor, whether the forests draw their splitting proportions from one pooled Dirichlet rather than one each. Default is FALSE. TRUE is an assumption about the data rather than a free improvement, and requires that the proportions be drawn at all and over the same predictors, so the forests that are to share must have sparsity = TRUE, which is not the default. Ignored for a family with a single forest. See Details.
categorical	<i>Advanced.</i> string; how a splitting rule divides the levels of a factor. Allowable options include "subset" (the default), which draws a subset of the levels still available at the node and sends those left, and "onehot", which follows most BART implementations: it splits on one indicator column, peeling a single level off the rest.
augment	<i>Advanced.</i> logical; whether to rewrite the likelihood as the margin of a Gaussian or a Poisson one, which makes the target a shape the sampler can exploit. The posterior is the same either way, so this is a sampling setting rather than a modeling one. Default is TRUE, which rewrites wherever a rewriting exists: the binomial, ordinal, multinomial, negative binomial, zero-inflated, and survival families. A rewriting is faster and has not been measured to mix worse per second, so the default is rarely worth changing; see vignette("implementation").
gamma, beta	<i>Advanced.</i> numeric; the branching probability at depth d is $\gamma * (1 + d)^{-\beta}$. Defaults are .95 and 2.
sigma_mu	<i>Advanced.</i> numeric; the prior median of the leaf standard deviation, one value per additive predictor. Default is NULL to derive it from k and that forest's own tree count.
update_sigma_mu	<i>Advanced.</i> logical; whether to draw the leaf standard deviation under a half-Cauchy prior rather than fixing it. Default is TRUE. FALSE is worth reaching for when a binary fit mixes badly: where the predictors separate the response well the leaf scale is barely identified, and it wanders rather than settling, which drags the effective sample size of everything built on it down with it. Fixing it can raise the effective sample size of the additive predictor and the log likelihood several times over at little cost in fit; see vignette("implementation").
sigma_mu_ramp	<i>Advanced.</i> numeric; the fraction of warmup over which the leaf standard deviation is raised from near zero to its target. Default is .25; set to 0 to disable. Linero (2025) describes this as essential: started at its full value, the sampler can settle early into a poor configuration and fail to move.

update_tau	<i>Advanced.</i> logical; whether to draw the standard deviation of each random-effect term under the same half-Cauchy prior the leaf scale uses, rather than fixing it at that prior's median. Default is TRUE. Only relevant when the formula has a (1 group) term.
update_bandwidth	<i>Advanced.</i> logical; whether to draw the bandwidth of each tree rather than holding it at bandwidth. Default is TRUE.
bandwidth_every	<i>Advanced.</i> numeric; how many sweeps between bandwidth draws for a given tree. Default is 1. The bandwidth is one scalar per tree, drawn by an adaptive random walk, and every attempt costs a full rebuild of the tree's memberships, which is the largest single item in a soft-rule fit, so drawing it less often trades mixing in that one parameter for speed. Raising it is reasonable when the mean function is known to be smooth and the fit is compute-bound; see Details.
alpha	<i>Advanced.</i> numeric; the concentration of the Dirichlet prior on the splitting proportions, where smaller values concentrate splits on fewer predictors. Default is NULL, which uses 1 as the starting value for a parameter that is then drawn.
alpha_scale, alpha_shape_1, alpha_shape_2	<i>Advanced.</i> numeric; the prior on alpha, in which $\alpha / (\alpha + \alpha_scale)$ is $\text{Beta}(\alpha_shape_1, \alpha_shape_2)$. Default for alpha_scale is NULL to use the number of predictor groups; the two shapes default to whatever sparsity implies.
update_s, update_alpha	<i>Advanced.</i> logical; whether to draw the splitting proportions and their concentration. Defaults are NULL to follow sparsity. Turning both off recovers a uniform prior over predictors, as setting sparsity = FALSE does.
verbose	<i>Advanced.</i> logical; whether to print progress to the console while sampling. Default is FALSE. For a progress bar instead, see the Progress section below, which needs no argument here.
num_print	<i>Advanced.</i> numeric; how many iterations between the reports verbose prints. Default is 100.
block_eval	<i>Validation.</i> logical; whether to evaluate the likelihood one leaf at a time rather than one observation at a time. Default is FALSE. A family built by <code>custom_family()</code> does this regardless, since it must call back into R; setting it for a compiled family produces the same draws from the same seed at somewhat greater cost.
exact_quadratic	<i>Validation.</i> logical; whether to use the closed forms that a target quadratic in the additive predictor allows, in which one pass over a node determines the log target everywhere, so that the Laplace approximation is the conditional posterior rather than an approximation to it. Default is TRUE, which makes a Gaussian response, or any of the rewritings in augment, cheap. FALSE falls back on the general path; the two agree, at greater cost.
generic_accumulate	<i>Validation.</i> logical; whether to accumulate a leaf's sums through the family's virtual interface rather than through its own statically dispatched loop. Default

is FALSE. The two compute the same thing; the second lets the compiler inline the family's arithmetic, which is most of the remaining per-observation cost.

Details

The arguments are roughly ordered by how often they are worth changing. Those from chains to `x_transform` are modeling decisions, in that changing one changes what is fitted or how long it is fitted for, and the chain lengths and `num_trees` come first because they are adjusted most. Those marked *Advanced*. are hyperparameters of a prior that one of the first group summarizes, or switches whose default is almost always right. The three marked *Validation*. compute the same posterior more slowly, and are documented so that the checks using them can be read.

Setting `num_burn`:

The default is comfortably longer than the transient in every family measured, so raising `num_burn` buys convergence rather than precision: raise it when `diagnose()` says the chains have not agreed. `dpm()` is the exception, as its mixture carries a component-count state that settles more slowly than a forest does, so a longer warmup is worth having there and especially when the error distribution itself is the object of interest.

Setting `num_draws`:

The defaults were chosen on out-of-sample error and effective sample size per second. Those are the right criteria for a posterior mean and they are not the binding ones for an interval, because an interval endpoint is a quantile of the draws and a short run places it too far out. The effect is Monte Carlo error and it shrinks at the usual rate, so it is a question of how much rather than whether.

Measured on one design, a Friedman function at $n = 4000$ with 25 of 30 predictors irrelevant, comparing the default against a run 64 times longer: pointwise 95% intervals for the regression function came out about 5% wider at the default and covered 0.938 against 0.958. Halving the excess width took about four times the sweeps, as an error decaying as the square root of the run predicts.

This suggests some habits rather than a number to apply. Raise `num_draws` when an interval, a tail quantile or a posterior probability is the deliverable and the cost is affordable, and leave it alone when a posterior mean or a held-out prediction is. And read the width rather than the coverage when checking whether a run was long enough, since a short run errs by being too wide and that is visible without knowing the truth. The magnitude above is one design and one family; the direction is general.

Setting `num_trees`:

The performance of the BART model is relatively insensitive to the number of trees. More trees create a more complicated model, which can mix poorly. A model with soft trees needs fewer trees than one with hard trees, but the default is the same for both. Generally, you don't need to change the number of trees for a model with a single forest; the defaults tend to do well, which is a strength of BART. The leaf prior scale divides by the square root of a forest's own tree count, so shrinking one forest does not change the prior on the sum.

The `num_trees` argument can be supplied as a vector when the family has more than one forest, where each value corresponds to the number of trees for that forest. The scale forest of location-scale models (e.g., `family = gaussian_ls()`) dominates the run time, its target not being quadratic, and a variance surface carries much less information than a mean surface, so giving it fewer trees runs substantially faster at the same accuracy. It is not the default because how much

resolution a variance surface needs depends on the surface, and quietly under-parameterizing it would show up as intervals that are wrong.

Choosing a gate type with gate:

A gate determines how much of each branch of a tree an observation contributes to, based on its value of the numeric predictor the rule splits on. For a predictor x and a threshold c , the contribution of the observation to the right-hand branch is $w_R(x; c)$, and the contribution to the left branch is $w_L(x; c) = 1 - w_R(x; c)$.

Hard Gates:

For a hard gate (gate = "hard"), which is used in traditional BART and most other tree-based models,

$$w_R(x; c) = \mathbb{I}(x > c)$$

where $\mathbb{I}(\cdot)$ is the indicator function. An observation contributes only to one branch. This is the fastest to compute but cannot capture smooth functions well, yielding jagged fits.

Soft Gates:

For a soft gate, used in the SoftBart model of Linero and Yang (2018),

$$w_R(x; c) = F\left(\frac{x - c}{h}\right)$$

for some S-shaped function $F(\cdot)$ with a bandwidth h .

The options for $F(\cdot)$ are the different values gate can take. Computing the contribution weights for soft gates is slower than for hard gates, and the bandwidth is itself a parameter with a Metropolis step per tree per sweep, each rebuilding every membership weight in the tree, which is the single largest item in a soft-rule fit. These features make models with soft gates slower to compute than models with hard gates. The difference among the soft gates ("smoothstep", "smootherstep", and "logistic") is negligible in predictive performance. "smoothstep" is computationally fastest and lets some observations have weights of exactly 0 or 1, which is why it is the default; "logistic" is the one Linero and Yang (2018) originally describe.

The `bandwidth_every` argument controls how often the bandwidth is updated from its starting value specified by `bandwidth` for a soft gate when `update_bandwidth = TRUE` (the default). Drawing the bandwidth lets a rule sharpen toward a step, so setting `update_bandwidth = FALSE` is faster and tends to be more accurate on smooth functions, but is much worse on nonsmooth ones. Raising `bandwidth_every` is the middle course, recovering some speed while keeping soft rules, at a real cost in mixing and a small one in accuracy where the mean function jumps.

Controlling Sparsity:

By default (`sparsity = FALSE`), every predictor has the same prior probability of being chosen for a splitting rule, as in classic BART, so no predictor can be left out of the forest. Setting `sparsity = TRUE` places the Dirichlet prior of Linero (2018) on the splitting proportions instead, which is a genuine variable-selection prior: it can and does drop a predictor from every tree at once, which is the point of it in the high-dimensional problems it was built for.

The sparsity prior is worth turning on when there are many predictors, when only a few of them are expected to matter, or when the goal is to identify the subset of predictors the model needs. Under it, the `prop_used` column of `variable_importance()` is readable as a posterior probability that a predictor belongs in the model, which it is not under the default; `vignette("importance")` covers that use. It also tends to improve prediction when many of the predictors are irrelevant.

It is not recommended when a contrast, a partial effect, or a treatment effect is being estimated. A contrast on a predictor the prior has dropped is exactly zero because in that draw the fit does not depend on that predictor at all, so the posterior of a contrast has an atom at zero whose mass is one minus the predictor's inclusion probability; any summary reporting a median lands on it once it holds half the mass. On a weak signal the estimate is attenuated substantially and its interval covers below its nominal rate, while a strong effect is untouched. More trees does not fix it, because a predictor whose splitting proportion has gone small is rarely proposed and so is hard to get back in. `vignette("effects")` works this through and `bartisan-marginaleffects` covers the atom. When an effect is the estimand but there are too many predictors to weight alike, `split_prior` is the middle course, since fixed weights cannot drop anything.

For a varying-coefficient model, the sparsity choice can differ by forest (see more details below). What the prior can drop is a predictor a forest splits on, and in a `vc()` model the treatment is the coefficient rather than one of those, carried by a forest of its own, so no splitting proportion can drop it; the prior on that forest selects among the moderators instead. So `sparsity = c(FALSE, TRUE)`, which keeps the default for the control function forest and enables the sparsity prior for the varying coefficient forest, is coherent, and in `bcf()` it is the setting worth considering when there are many candidate moderators.

Sparsity can also be controlled by choosing the sparsity parameter `alpha` or drawing it from a prior determined by the parameters `alpha_scale`, `alpha_shape_1`, and `alpha_shape_2`. The `sparsity` argument is a shortcut to setting these, since the relationship between them and the sparsity they induce is a bit opaque.

Telling the Prior What Is Already Known:

`sparsity` and `split_prior` answer different questions and cannot both be in force, so giving `split_prior` turns `sparsity` off. The `sparsity` argument is for when which predictors matter is unknown and the prior is to work it out from the data, and a predictor can be dropped entirely; the `split_prior` argument is for when something is known and is to be honored, with the proportions held at the supplied values.

A weight is a statement about relative attention, not about effect size. It changes how often a split on a predictor is proposed, which is a prior, so the data can still overrule it in either direction. Because the weights are fixed, `split_prior` does not accumulate the atom-at-zero mass above, which makes it a reasonable middle course when a particular contrast is the estimand but there are too many predictors to treat alike. One weight per term in the formula, not per column of the design matrix, so a factor is named once and its levels share the weight.

Arguments That Vary by Forest:

A family with several additive predictors has one forest per predictor, each with its own prior. Every argument that could mean something different for one of them may be given once, to apply to all, or once per forest, either positionally or keyed by the forest names listed in `bartisan-families`. A forest a named argument does not mention keeps that argument's default rather than borrowing another forest's value. That covers `num_trees`, `k`, `sigma_mu`, `sparsity`, `split_prior`, `bandwidth`, `gamma`, `beta`, the four `alpha` arguments, and the three `update_flags`; the formula argument works the same way, as `bartisan()` describes.

```
bartisan_control(num_trees = c(mean = 50, log_sd = 10),
                 sparsity = c(mean = TRUE, log_sd = FALSE))
```

The multinomial families are the exception, for the reason given in `bartisan-families`: their forests act as one, so these arguments take a single value.

Progress:

Setting `verbose = TRUE` prints a line every `num_print` iterations, which is the whole of what this package decides about progress. A progress bar is **progressr**'s business, and the sampler reports to it unconditionally: nothing is shown unless a handler is active, so there is no argument to switch on. To request a progress bar, use one of the calls below:

```
progressr::with_progress(  
  bartisan(y ~ ., data = d, family = gaussian())  
)
```

```
# or once, for the session  
progressr::handlers(global = TRUE)
```

The bar is sized for the whole fit, so setting `chains = 4` fills one bar once rather than four in sequence, and chains running in parallel under **future** relay their progress back as it arrives.

Parallelization with future:

To request parallel processing, specify a future plan, e.g., using

```
future::plan("multisession")
```

Parallelization only works with multiple chains (i.e., `chains` set to greater than 1), and each chain is run in parallel. With large fits (many draws or a large sample size), an error might be thrown by `future::plan()`, which refuses a single export above `future.globals.maxSize`, 500 MB by default. The error names the option; raising it is the fix, and running sequentially is the alternative. Parallelization can be used with a **progressr** progress bar.

Value

A `<bartisan_control>` object, a list containing the supplied settings and the defaults for those not supplied, for passing to the `control` argument of `bartisan()`.

References

- Linero, A. R. (2018). Bayesian regression trees for high-dimensional prediction and variable selection. *Journal of the American Statistical Association*, 113(522), 626–636. doi:10.1080/01621459.2016.1264957
- Linero, A. R. (2025). Generalized Bayesian additive regression trees models: beyond conditional conjugacy. *Journal of the American Statistical Association*, 120(549), 356–369. doi:10.1080/01621459.2024.2337156
- Linero, A. R., Sinha, D., & Lipsitz, S. R. (2020). Semiparametric mixed-scale models using shared Bayesian forests. *Biometrics*, 76(1), 131–144. doi:10.1111/biom.13107
- Linero, A. R., & Yang, Y. (2018). Bayesian regression tree ensembles that adapt to smoothness and sparsity. *Journal of the Royal Statistical Society Series B*, 80(5), 1087–1110. doi:10.1111/rssb.12293

See Also

`bartisan()`, which takes the result as its `control` argument; `bartisan-families` for the forest names the per-forest arguments are keyed by
`vignette("implementation")` for more details on the BART implementation.

Examples

```
data("rhc")
set.seed(123)

# Settings can be built up once and reused across fits
ctrl <- bartisan_control(num_trees = 20,
                        gate = "hard",
                        num_burn = 50,
                        num_draws = 50)

model <- death ~ rhc + age + sex + race + edu + aps +
  meanbp + resp + hema + pafi + paco2 + crea + surv2m +
  card

fit <- bartisan(model, data = rhc, control = ctrl)

# The same call, with the settings passed through `...`
# instead
fit2 <- bartisan(model, data = rhc,
                 num_trees = 20, gate = "hard",
                 num_burn = 50, num_draws = 50)

# A setting given once applies to every forest, and a
# vector gives each forest its own value. A variance
# surface needs less capacity than a mean surface
bartisan_control(num_trees = c(mean = 50, log_sd = 10))

# Weighting the splitting prior toward the treatment, for
# a model with more predictors than are worth weighting
# alike
bartisan_control(split_prior = c(rhc = 10))
```

bcf

Bayesian causal forests

Description

Fits a varying-coefficient model set up for estimating a treatment effect: a control function for the outcome under no treatment and a separate forest for the effect, with the prior on the effect regularized more heavily than the prior on the control function. This is Hahn, Murray and Carvalho (2020) for a binary treatment and Woody, Carvalho, Hahn and Murray (2020) for a continuous one.

Usage

```
bcf(
  formula,
  treat,
  data,
```

```

    family = NULL,
    moderators = NULL,
    propensity = TRUE,
    propensity_args = list(),
    ...
)

```

Arguments

formula	a model formula. The right-hand side lists the covariates; the treatment is named in <code>treat</code> rather than here, and is removed from the covariates if it appears among them.
treat	a one-sided formula naming the treatment, as in <code>~ z</code> . The treatment may be binary, categorical, or continuous, and which it is decides what the propensity score is and how it is modeled; see Details.
data	a data frame containing the variables named in <code>formula</code> .
family	the response distribution, given as a <code>stats::family</code> object, as one of the families in <code>bartisan-families</code> , or as the name of either. A family object is accepted when the distribution it names is one this package implements, since the likelihood is the package's rather than the object's: a family object carries a link and a variance function and not a density, so one naming anything else (e.g., <code>stats::inverse.gaussian</code> , or a Tweedie from another package) is an error rather than something a likelihood can be built from, and <code>custom_family()</code> is the route for those. Links are used as supplied, and a link the package does not compile is composed onto the scale its family works on. Default is <code>NULL</code> , in which case the family is read off the response and a message reports the choice; see Details for the rules.
moderators	a one-sided formula naming the covariates the treatment effect may vary with, in which <code>.</code> stands for the covariates in <code>formula</code> , so that <code>~ . - x</code> lets it vary with every covariate but <code>x</code> . Default is <code>NULL</code> to let the effect vary with every covariate.
propensity	what to do about the propensity score, given as either a logical value, a numeric vector or matrix, or a one-sided formula. Default is <code>TRUE</code> to fit a model for it and add the fitted values to the control function. <code>FALSE</code> fits nothing, a numeric vector or matrix is used as given, and a one-sided formula fits it with the predictors that <code>formula</code> names, in which <code>.</code> stands for the covariates in <code>formula</code> rather than for every column of data. For a continuous treatment the fitted values are the treatment's conditional mean given the covariates; see Details.
propensity_args	a list of arguments for the propensity model: <code>bartisan_control()</code> settings, and <code>family</code> to replace the family chosen from the treatment's type (see Details). Default is <code>list()</code> to leave every setting at its own default.
...	passed to <code>bartisan()</code> , including <code>bartisan_control()</code> settings.

Details

The Model:

`bcf()` writes several settings into a `bartisan()` call, all of which can be written out there directly. The treatment gets a `vc()` term, so the effect is a forest of its own with its own prior rather than whatever difference a single forest with the treatment among its predictors happens to produce. The propensity score goes in the control function and not in the effect forest, so that the control function absorbs the selection while the effect stays free of it. The effect forest gets fewer trees than the control function, since patterns of effect heterogeneity are usually simpler than prognostic surfaces. A binary treatment has its coding drawn rather than fixed, which is `center = "estimate"` in `vc()` and makes the answer the same whichever level was written as 1; a treatment with more levels keeps the symmetric per-level coding. The `sparsity` argument keeps its default of `FALSE` for both forests.

The propensity score that enters the control function is the posterior mean of a separate model for the treatment, fit before the outcome model and then held fixed. Its uncertainty is therefore not carried into the interval for the effect, and the outcome has no say in the score, as it would in a joint model of the two. See `vignette("causal")` for more on this choice. Including the score at all matters because, with flexible priors on the outcome that are independent of the model for the treatment, the implied prior on the amount of confounding bias concentrates near zero (Linero, 2024).

Setting sparsity:

The default splitting prior weights the predictors equally and cannot drop one. That is the right setting when a treatment is one predictor among many in a single forest, since a variable-selection prior can drop it from the forest entirely and put a point mass at exactly zero in the posterior of a contrast on it. Here the treatment is the coefficient rather than a predictor the forest splits on, so no splitting proportion can drop it, and the sparsity prior can be turned on without that risk. What it selects among on the effect forest is the moderators, and dropping all of them leaves an effect that does not vary rather than one that is zero, which is the shrinkage a heterogeneity model wants.

Setting `sparsity = c(FALSE, TRUE)` leaves every predictor in the control function and asks only the effect forest to select among them, which is worth considering when there are many candidate moderators and a few of them are expected to matter. It is not the default because it costs accuracy when the effect does not vary: with nothing real to select, the prior concentrates the effect forest's splits on one or two of the nuisance moderators and produces heterogeneity that is not there, where the uniform prior spreads them thinly and averages it away. In simulations with twenty candidate moderators, turning it on lowered the error of the conditional effect by about 14% when two moderators were real and raised it by about 18% when none was, and left the average effect unchanged either way.

Setting treat:

The treatment decides what the propensity score is and, by default, the family of the model that estimates it. A different family can be named with `propensity_args = list(family = ...)`.

treatment	propensity score	model
binary	one column, the probability of treatment	<code>binomial()</code>
K categories	the whole vector of assignment probabilities	<code>multinomial()</code>
continuous	the conditional mean of the treatment	<code>gaussian()</code>

For a treatment with more than two categories the balancing score is the whole vector of assignment probabilities rather than any one of them, so all of them go into the control function. They

sum to one and are therefore collinear, which costs a tree ensemble nothing.

Continuous Treatments:

For a continuous treatment, the score added to the control function is the treatment's conditional mean given the covariates, fit with a Gaussian model. The balancing score proper would be the conditional density of the treatment at the observed dose, but the purpose of the score here is to let the control function absorb the confounding, and Linero (2024) shows for linear models that it is the conditional mean of the treatment whose absence leaves the prior on the confounding bias concentrated near zero. A score supplied as a number is used as given.

The default family is `gaussian()` rather than `dpm()`, which `bartisan()` would choose for a numeric response. Only the ordering of the fitted means matters to the trees that use them, and in simulations `dpm()` ordered them slightly better when the treatment's errors were heavy-tailed or skewed but far worse when the treatment had a point mass at zero, which its error mixture absorbs. For a treatment with heavy tails, `propensity_args = list(family = dpm())` may do better, and any family whose fitted values are the conditional mean can be named the same way.

A continuous treatment also carries an assumption. This fits $f_0(x) + z * f_1(x)$, a dose response that is linear in the dose with a slope that varies with the covariates, which for a binary treatment is no assumption at all and for a continuous one is a real one. Where the dose response itself might be curved, either put the treatment in as an ordinary predictor or give `f1` the treatment among its moderators by writing the term as `vc(z, ~ z + ...)` in `bartisan()` directly, which lets the effect vary across the dose; see `vc()`.

Predicting for New Data:

The propensity score is a predictor of the control function and one the caller never named, so a `newdata` data frame taken from their own frame does not carry it. `predict()` rebuilds it from the model the fit kept, which makes that work. Supplying propensity as a number rather than fitting it removes the reconstruction, and the data given as `newdata` then has to carry the column itself.

Value

A `<bcf_fit>` object, which is the `<bartisan_fit>` `bartisan()` returns with a class in front of it and the treatment's coefficient forest named for the treatment. Everything that works on a `<bartisan_fit>` works here unchanged; the class exists so that methods needing a named treatment have something to dispatch on. For a binary or categorical treatment, `estimate_effect()` can be used on the output to return interpretable marginal and conditional effects; for a continuous treatment, see `bartisan-marginaleffects`. `coef()` gives the conditional effect on the link scale for each observation.

References

- Hahn, P. R., Murray, J. S., & Carvalho, C. M. (2020). Bayesian regression tree models for causal inference: regularization, confounding, and heterogeneous effects. *Bayesian Analysis*, 15(3), 965–1056. doi:10.1214/19BA1195
- Linero, A. R. (2024). In nonparametric and high-dimensional models, Bayesian ignorability is an informative prior. *Journal of the American Statistical Association*, 119(548), 2785–2798. doi:10.1080/01621459.2023.2278202
- Woody, S., Carvalho, C. M., Hahn, P. R., & Murray, J. S. (2020). Estimating heterogeneous effects of continuous exposures using Bayesian tree ensembles. doi:10.48550/arXiv.2007.09845

See Also

`estimate_effect()` for the average or conditional effect; `bartisan()` and `vc()` for the general interface this is written in terms of; `vignette("varying")` for the varying-coefficient model this is a case of and what it writes into the call; and `vignette("causal")` for the assumptions under which the effect is causal.

Examples

```
data("rhc")

set.seed(123)

# The effect of right heart catheterization on death,
# free to vary with every covariate, with the propensity
# score entering the control function alone
model <- death ~ age + sex + race + edu + aps + meanbp +
  resp + hema + pafi + paco2 + crea + surv2m + card

# Note: we use small sampler settings that would not be
# appropriate for analysis; this is just for the
# examples
fit <- bcf(model, treat = ~ rhc, data = rhc,
  family = binomial(), num_trees = c(10, 5),
  num_burn = 50, num_draws = 50,
  propensity_args = list(num_trees = 10,
    num_burn = 50,
    num_draws = 50))

# One conditional effect per patient: the effect forest
# evaluated at each observation
head(coef(fit))

# The average effect over the sample, on the response
# scale, which for a binomial fit makes it a risk
# difference rather than a log odds ratio
estimate_effect(fit)

# Or the whole picture at once
summary(fit)
```

coef.bartisan_fit

*Varying coefficients***Description**

The coefficient functions of a model fitted with `vc()` terms, evaluated at each observation. A forest has no coefficient vector, so a fit with no `vc()` term has no coefficient to report and this errors rather than returning anything.

Usage

```
## S3 method for class 'bartisan_fit'
coef(object, newdata = NULL, draws = FALSE, ...)
```

Arguments

object	a <bartisan_fit> object; the output of a call to <code>bartisan()</code> , fitted with at least one <code>vc()</code> term.
newdata	optional; a data frame at which to evaluate the coefficients. Default is NULL to use the data the model was fitted to.
draws	logical; whether to return every posterior draw of each coefficient rather than its posterior mean at each observation. Default is FALSE to return the posterior means.
...	not used.

Details

The coefficients are the varying ones alone. The control function is the surface at the value each covariate was centered on, which is a prediction rather than a coefficient, and `predict(object)` reports it.

For a factor the coefficients are recentered to sum to zero across its levels, which makes them the deviations they are reported as. The symmetric coding carries one spare function-valued dimension, so this is exact rather than an approximation, and it is the reason a factor's reference level is a choice made here rather than at fitting time.

Value

With `draws = FALSE`, a matrix with one row per observation and one column per coefficient. With `draws = TRUE`, a named list of draws-by-observations matrices, one per coefficient.

See Also

`vc()` for declaring a varying coefficient; `variable_importance()` for which predictors a forest uses at all

Examples

```
data("rhc")
set.seed(123)

# The effect of catheterization is allowed to vary with
# the other predictors, so its coefficient is a function
# rather than a number
fit <- bartisan(death ~ age + aps + surv2m + vc(rhc),
               data = rhc, num_trees = 10, num_burn = 50,
               num_draws = 50, verbose = FALSE)

# One coefficient per patient, on the link scale
head(coef(fit))
```

```
# How much it varies across patients
quantile(coef(fit)[, "rhc"])
```

custom_family

Fit a model to a likelihood written in R

Description

custom_family() takes the log density itself, as an R function, and returns a family object [bartisan\(\)](#) fits the model for. It is the route to a response distribution none of the compiled families in [bartisan-families](#) cover.

Usage

```
custom_family(
  logdens,
  num_predictors = 1L,
  start = 0,
  derivatives = NULL,
  aux_names = NULL,
  aux_start = NULL,
  name = "custom"
)
```

Arguments

- | | |
|----------------|--|
| logdens | the log density, given as a function of the response and the additive predictors, function(y, eta), where y holds the response of n observations and eta is an n by num_predictors matrix of their additive predictors, returning a numeric vector of length n. y is the response as it appears in the model formula, with only those observations kept: a vector or factor of length n, or a matrix with n rows, such as one made with cbind() or survival::Surv() . See Details. With nuisance parameters it must take a third argument, function(y, eta, aux), where aux is a numeric vector of their current values. It is the log density of one unit of prior weight, so that weights behave as they do elsewhere, and terms free of eta may be dropped. |
| num_predictors | numeric; how many additive predictors the density has (i.e., how many forests to fit). Default is 1. |
| start | numeric; the value each additive predictor starts at, in place of the intercept-only fit the compiled families use. One value or one per predictor. Default is 0. |
| derivatives | optional; a function(y, eta, h) returning a list with elements score and info, the first derivative of logdens with respect to the hth predictor and minus its second derivative, each a vector of length n. With nuisance parameters it |

	takes a fourth argument, <code>function(y, eta, h, aux)</code> , which receives their current values as <code>logdens</code> does. Default is <code>NULL</code> to take central differences of <code>logdens</code> . It covers the additive predictors only: a nuisance parameter is always differenced, which costs three calls per sweep rather than three per leaf.
<code>aux_names</code>	optional character; the names of the nuisance parameters to draw, if any. Naming them declares them, because the names label the columns of <code>fit\$aux</code> and are the labels <code>summary()</code> and <code>diagnose()</code> report them under. They must be distinct and non-empty. Default is <code>NULL</code> for none, unless <code>aux_start</code> is given, in which case the parameters are named by <code>names(aux_start)</code> when it carries names and positionally (" <code>aux1</code> ", " <code>aux2</code> ", and so on) when it does not.
<code>aux_start</code>	optional numeric; the value each nuisance parameter starts at, given as one value or one per parameter. Default is <code>NULL</code> , which is 0 for each. The sampler will walk to the posterior from a poor start, so this need only be the right order of magnitude. Supplying it is a second way to declare the parameters, so setting <code>aux_start = c(shape = 1)</code> both names one and starts it at 1.
<code>name</code>	string; a label used when printing the fit. Default is " <code>custom</code> ".

Details

Nothing else about the sampler changes: the leaf-level Laplace proposal needs the first two derivatives of the log density with respect to each additive predictor and nothing more, and central differences of the supplied function produce both.

```
# A Poisson model written out by hand. Terms free of eta may be dropped;
# they cancel from every acceptance ratio.
pois <- custom_family(function(y, eta) y * eta[, 1] - exp(eta[, 1]),
                      start = log(mean(d$y)))

# Two predictors: a mean and a log standard deviation.
ls <- custom_family(function(y, eta) dnorm(y, eta[, 1], exp(eta[, 2])),
                    log = TRUE),
                    num_predictors = 2, start = c(0, 0))
```

The function is called once per leaf per Fisher-scoring step with the observations reaching that leaf, so it must be vectorized over `y` and the rows of `eta`; it must not be vectorized *within* an observation, and it must return exactly one value per row. Supplying derivatives cuts three calls to one and removes the differencing error.

Because it sees a subset rather than the whole sample, a per-observation vector captured from the enclosing environment will not line up with the rows it is handed, and nothing can detect that for you: the fit runs and is wrong. Anything the density needs that varies across observations should instead be part of the response, which is subset along with `eta`. A varying number of trials, for example, can be supplied as a second column, as in `cbind(hits, size) ~ x`, and read inside `logdens` as `y[, 2]`. An offset belongs in the formula, as it does for the other families.

Nuisance Parameters:

These are drawn alongside the trees when the `aux_names` argument names them, and the function supplied as `logdens` then takes a third argument holding their current values:

```
# A Gaussian written out by hand, with its scale drawn rather than fixed.
by_hand <- custom_family(
  logdens = function(y, eta, aux) dnorm(y, eta[, 1], exp(aux[1]), log = TRUE),
  aux_names = "log_sigma", aux_start = 0)
```

They are reported in `fit$aux` under those names, and covered by `summary()` and `diagnose()` like any other family's. There is no prior argument and no bounds argument, because a nuisance parameter here is carried as an additive predictor whose forest is pinned at depth zero (one tree that can never split, so the forest is a single scalar), and it is drawn by the same Laplace-plus-Metropolis step as any leaf, under that step's Gaussian leaf prior. So a parameter with a restricted range is handled the way it would be for a real predictor, by writing the transform into `logdens`: the `exp()` above keeps the scale positive.

Responses Other Than a Numeric Vector:

The response reaches `logdens` as it appears in the model formula, so it can be a factor, a matrix with a column per outcome, or a `survival::Surv()` object, which keeps its time and status columns. A factor arrives with the levels it had when the model was fit, in `predict()` with `newdata` as well, so a density can read as `integer(y)` as a category code. A matrix response makes it possible to fit a multivariate BART model, with one additive predictor per outcome and the outcomes related through the error distribution:

```
# A bivariate normal, with the two log standard deviations and the Fisher
# transform of the correlation drawn as nuisance parameters.
bvn <- custom_family(
  logdens = function(y, eta, aux) {
    s <- exp(aux[1:2])
    rho <- tanh(aux[3])
    z1 <- (y[, 1] - eta[, 1]) / s[1]
    z2 <- (y[, 2] - eta[, 2]) / s[2]
    -sum(log(s)) - log1p(-rho^2) / 2 -
      (z1^2 - 2 * rho * z1 * z2 + z2^2) / (2 * (1 - rho^2))
  },
  num_predictors = 2, start = c(mean(d$y1), mean(d$y2)),
  aux_names = c("log_sd1", "log_sd2", "atanh_rho"))
```

```
fit <- bartisan(cbind(y1, y2) ~ x1 + x2, data = d, family = bvn)
```

Each forest's prior is scaled for an additive predictor that varies by a few units around its start, so outcomes measured on a large scale are best standardized first. A row with any missing value in the response is dropped, as it is for the other families.

Limits of a Log Density:

A density corresponds to how likely an observed value is, not how to draw a new one, so a `custom_family()` fit has no posterior predictive distribution, which `simulate()`, `pp_check()` and `r2()` rely on. Each of them errors on such a fit rather than returning something it cannot support. `loo()` and `waic()` are unaffected, since both read the pointwise log likelihood the family already computes. Note that terms dropped from `logdens` because they are free of `eta` are missing from that log likelihood too, so its values can be compared across fits only when the same terms are dropped from each (e.g., not against a compiled family, whose density is complete).

There is a smaller limit as well. Since the package cannot know what the mean of the density is, `predict(type = "response")` returns the additive predictors rather than a fitted mean.

Value

A `<bartisan_family>` object, which is a list containing at least the elements `family` and `link` and which inherits from `family`, so that `bartisan()` recognizes it wherever it recognizes an ordinary `stats::family` object.

See Also

`bartisan-families` for the compiled families, one of which is usually the better answer; `bartisan()` for fitting a model with the result; `vignette("families", package = "bartisan")` for the long form

Examples

```
set.seed(123)

d <- data.frame(x1 = runif(300), x2 = runif(300))
d$y <- rpois(300, exp(1 + sin(pi * d$x1)))

# A Poisson likelihood written out by hand, with its
# derivatives. Terms free of eta may be dropped, since
# they cancel from every acceptance ratio, and the same
# terms are absent from the score.
pois <- custom_family(
  function(y, eta) y * eta[, 1] - exp(eta[, 1]),
  derivatives = function(y, eta, h) {
    list(score = y - exp(eta[, 1]),
         info = exp(eta[, 1]))
  },
  start = log(mean(d$y)))

fit <- bartisan(y ~ x1 + x2,
               data = d, family = pois,
               num_trees = 10, num_burn = 50,
               num_draws = 50)

fit

# A beta-binomial, which no built-in family covers:
# counts out of a known number of trials, overdispersed
# relative to a binomial. The number of trials varies,
# so it is the response's second column, which reaches
# `logdens` row for row with `eta`. `phi` is drawn
# alongside the trees.
d$size <- sample(10:40, 300, replace = TRUE)
p_true <- plogis(d$x1)
d$hits <- rbinom(300, d$size,
               rbeta(300, p_true * 6, (1 - p_true) * 6))

bb <- custom_family(
  logdens = function(y, eta, aux) {
    hits <- y[, 1]
```

```

size <- y[, 2]
p <- plogis(eta[, 1])
phi <- exp(aux[1])
lbeta(hits + p * phi, size - hits + (1 - p) * phi) -
  lbeta(p * phi, (1 - p) * phi)
},
aux_names = "log_phi", aux_start = log(5),
name = "beta-binomial")

fit_bb <- bartisan(cbind(hits, size) ~ x1 + x2,
  data = d, family = bb,
  num_trees = 10, num_burn = 50,
  num_draws = 50)

# The drawn precision, on the scale it was written on.
exp(mean(fit_bb$aux[, "log_phi"]))

```

diagnose

*Check whether a fit converged and mixed***Description**

Reports the convergence and mixing diagnostics worth looking at before anything is read off a fit, and says what to do about whichever of them fall short. Everything is computed from the stored draws, so no other package is needed.

Usage

```

diagnose(object, rhat_max = 1.01, ess_min = 400, ...)

## Default S3 method:
diagnose(object, rhat_max = 1.01, ess_min = 400, ...)

## S3 method for class 'bartisan_fit'
diagnose(object, rhat_max = 1.01, ess_min = 400, ...)

## S3 method for class 'bartisan_effect'
diagnose(object, rhat_max = 1.01, ess_min = 400, ...)

```

Arguments

object	a <bartisan_fit> object, the output of a call to bartisan() , or a <bartisan_effect> object, the output of a call to estimate_effect() .
rhat_max	numeric; the largest R-hat treated as acceptable. Default is 1.01, the threshold of Vehtari et al. (2021); 1.1 was the older convention and is now considered too permissive.

<code>ess_min</code>	numeric; the smallest effective sample size treated as acceptable, for the bulk and the tail alike. Default is 400, Vehtari et al.'s recommendation of 100 per chain at four chains, which is about what it takes for the Monte Carlo error of an interval endpoint to be small next to the posterior's own width.
<code>...</code>	ignored.

Details

This is where the diagnostics are computed, and the only place: `bartisan()` does not carry a table of its own, because the per-observation statistics cost more than the sampling and would then be recomputed here. With a **future** plan in place the pass is spread over the workers, and it reports progress through **progressr** the way the sampler does.

The Rows of the Table:

One row per scalar the sampler draws (the log likelihood, the nuisance parameters of the family, and the scale of each random-effect term), plus two rows for the additive predictor and two for each set of group intercepts: one summarizing the worst 5% of observations or levels, and one for their average. A fit with soft decision rules gets the same pair for the gate bandwidth, summarized over trees rather than over observations, since one bandwidth is drawn per tree.

Both are reported because they routinely disagree, and which one binds depends on what is being reported. A forest settles the level of the fitted function within a sweep or two and takes much longer to settle which observation gets which share of it, so an average or a contrast of averages is governed by the average row and a prediction for one observation by the worst 5%.

An `ordinal()` fit with three or more categories is the exception, and which of the two rows to read is different there. Only the differences between the cutpoints and the additive predictor are identified, so the draws are recorded in the chart where the predictor has mean zero over the fitted sample and every cutpoint is free (see `bartisan-families`). The average over observations is then zero in every draw, which leaves it nothing to diagnose, and it is reported as NA. The level of the fitted function has not gone anywhere: the sampler pins the first threshold, so `aux.cut1` is that level rather than a cutpoint, and it is the row to read wherever the level matters, as it is for a probability in the lowest categories. It is usually the slowest row in such a fit, and the most pessimistic one, since it carries the level on its own where every quantity computed from the draws mixes the level with faster-moving ones.

A `ph()` fit is recorded the same way, for the same reason. Multiplying every baseline hazard by a constant and subtracting its log from the additive predictor leaves the likelihood unchanged, so each draw is recorded with the predictor centered over the fitted sample and the hazards scaled to match. The average row is again reported as NA, and the level of the fitted function is in the `aux.lambda` rows, which are the hazards of a unit whose predictor sits at that average. Recorded in the sampler's own chart, the hazards and the level would drift together and read as badly mixed while the survival probabilities computed from them mixed well. A fit with `update_lambda = FALSE` holds the baseline fixed, which leaves nothing to trade off, so its draws are recorded as they were sampled.

`rhat` is split-R-hat, so drift inside a chain counts as disagreement rather than hiding inside a chain mean. `rhat_late` is that same statistic on the second half of the retained draws alone, which separates the two reasons chains disagree: if R-hat is high overall and acceptable late, warmup ended too early, and if it stays high late, the chains have each settled somewhere different. `ess_bulk` and `ess_tail` are rank-normalized effective sample sizes, reported separately because a chain can be ample for a posterior mean and nowhere near enough for an interval endpoint. The

forest itself is checked through the total number of splitting rules at each draw, since chains that disagree about how large the forest is are exploring different tree structures.

That last row is graded apart from the others, and the distinction is worth understanding before acting on either. A sum of trees represents one function through many different partitions, so the number of splitting rules is not pinned down by the fit the way a fitted value is: two chains can agree to three figures about every value of the additive predictor while using forests of different sizes. The quantities a fit reports are integrals over the tree structure, so their convergence is a separate question from its convergence. The checks on R-hat and effective sample size therefore read the reported quantities, and the splitting rules get a check and a remedy of their own. It is separated rather than suppressed, because it does bind on anything computed from the split counts themselves, which is `variable_importance()` and `vignette("importance")`.

The gate bandwidth is graded apart for the same reason and is worth reading for a different one. A tree's rules can be made wider or narrower while the function the tree encodes stays where it was, so the bandwidth is no more pinned down by the fit than the number of splitting rules is; but it can be the slowest quantity in the model, and far slower than anything the fit reports. Under a Poisson likelihood a tree's bandwidth carries a median of 68 effective draws per 1000 against a binomial's 190, and the worst tree in a fit 9 against 114. Note that this is a property of the family and not of the data, so it is worth looking at on any `poisson()`, `negbin()` or other count fit left at the default soft rules. Setting `update_bandwidth = FALSE` in `artisan_control()` removes the draw entirely at the cost of fixing the amount of smoothing, and raising `num_draws` is the alternative.

The grading rests on how the model is parameterized and not on that row being the worst one, which it usually is not. Measured over 144 fits spanning three families, hard and soft rules and sample sizes from 500 to 8000, the splitting rules carried the highest R-hat in 8 of them; a reported quantity carried it in the other 136. Its R-hat runs above the averaged predictor's by 0.13 on a Gaussian fit, 0.05 on a probit one and 0.03 on an ordinal one, which is real and small. **The wide gap in a fit is not between the reported rows and this one, but within the reported rows:** the predictor averaged over observations carried a median of 93 times the effective sample size of its own worst 5%, 7815 against 52. Which of those two governs a given summary is the question the table is for, and the note about individual observations against their average is the line to read.

The leaf scale `sigma_mu` is left out of the table, and is in `fit$sigma_mu` and `as_draws()` for anyone who wants to look.

Setting `rhat_max` and `ess_min`:

R-hat is a ratio of two variance estimates taken from the same draws, so with few effective draws it sits above 1 whether or not anything is wrong, and how far above depends on how many chains are being compared. The `rhat_max` argument is therefore not a threshold a quantity can be held to at any effective sample size: four chains need about 400 effective draws before 1.01 is even the average of R-hat's null, which is where the pairing of the two defaults comes from, and sixteen chains need about 1600 for the same 1.01. Where a quantity fails R-hat while carrying fewer than that, the checks report that the number cannot be read yet and send the reader to the effective sample size instead.

Remedies for Poor Mixing:

The advice the print method gives follows from which statistic failed, and the order matters because the fixes are not interchangeable.

One chain comes first, since nothing else can be diagnosed properly until there are several; setting `chains = 4` is the change to make, and with **future** installed and a parallel backend in use it

usually costs little wall clock. R-hat elevated but acceptable on the late draws says warmup ended too early, so the `num_burn` argument is the one to raise. R-hat elevated on the late draws too says the chains have each settled somewhere different: raise `num_burn` and `num_draws` together, and failing that reduce `num_trees` and check the family, since a likelihood that fits badly can produce a posterior with no single place to be.

Effective sample size depends on the total number of draws rather than on how they are divided between chains, where R-hat compares chains against each other, so a fit failing on R-hat wants longer chains rather than more of them and a fit failing only on effective sample size can have either. A low effective sample size with R-hat fine is the benign case and wants only more draws; a low tail effective sample size with the bulk fine says the posterior mean is sound and the interval endpoints are not.

`vignette("diagnostics")` works all of this through on a fit.

Value

A `<bartisan_diagnosis>` object, a list with a `print()` method that shows the table, the checks and the advice. Its components are

`table` a data frame with one row per quantity: `rhat`, `rhat_late` (the same statistic on the second half of the draws alone), `ess_bulk`, `ess_tail`, and `ess_frac`, the bulk effective sample size as a fraction of the draws kept.

`checks` a data frame of check, status ("ok", "warn" or "note") and detail.

`advice` a character vector, most important first, empty when everything passed.

`chains,draws` how many chains, and how many draws were kept in total.

Diagnosing an Estimand:

Called on the output of `estimate_effect()`, the table has one row per reported quantity rather than one per sampled parameter, and everything else reads the same way.

It is worth doing rather than inferred from the fit's own table, because the two can disagree. An estimand is a contrast, and a contrast can mix badly where the function it is a contrast of mixes well: under the default splitting prior a draw that gives the treatment no rule puts the contrast at exactly zero, and the sampler can stay there for a long run while the other predictors keep the fitted function moving. The check reports the share of draws sitting at the atom when there is one. Chains that have settled on different fitted functions can still agree about an average over them, so this complements the fit's own diagnosis rather than replacing it.

See Also

`bartisan_control()` for the settings the advice names; `as_draws()` for handing the draws to **bayesplot** or **posterior**; `vignette("diagnostics")` for the fuller treatment, including posterior predictive checks

Examples

```
data("rhc")
set.seed(123)

model <- death ~ rhc + age + sex + race + edu + aps +
  meanbp + resp + hema + pafi + paco2 + crea + surv2m +
```

```

card

# Two chains, both deliberately short, so that there is
# something to report
fit <- bartisan(model, data = rhc, num_trees = 10,
               chains = 2, num_burn = 50, num_draws = 50,
               verbose = FALSE)

# The table, the checks, and what to do about whichever
# of them failed
diagnose(fit)

# A stricter effective sample size, which an interval
# endpoint needs and a posterior mean does not
diagnose(fit, ess_min = 1000)

```

error_density

*Error distribution of a Dirichlet process mixture fit***Description**

Estimates the density of the errors of a `dpm()` or `dpm_aft()` fit, as the posterior of the density of a new error evaluated on a grid. Where BART commits to one normal, this says what shape the errors actually have.

Usage

```
error_density(object, at = NULL, level = 0.95, iterations = NULL)
```

```
## S3 method for class 'bartisan_error_density'
plot(x, y, ...)
```

Arguments

<code>object</code>	a <code><bartisan_fit></code> object fit with <code>family = dpm()</code> or <code>family = dpm_aft()</code> ; every other family fixes the error distribution, so its density is a closed form rather than something to estimate.
<code>at</code>	numeric; the grid to evaluate the density on. Default is NULL for 201 points spanning four posterior-mean error standard deviations either side of zero.
<code>level</code>	numeric; the width of the pointwise interval. Default is .95 for 95% intervals.
<code>iterations</code>	numeric; optional indices of the stored draws to use, between 1 and the number of draws the fit retains. Default is NULL to use all of them.
<code>x</code>	a <code><bartisan_error_density></code> object; the output of a call to <code>error_density()</code> .
<code>y</code>	not used.
<code>...</code>	ignored; present for compatibility with the generic.

Value

A `<bartisan_error_density>` object, which is a data frame with one row per grid point and its own `plot()` method, with columns `at`, `mean`, `lower`, and `upper`, giving the posterior mean density and a pointwise interval.

`plot()` on the result draws the posterior mean density with that interval as a ribbon and returns a **ggplot2** object, so it can be added to in the usual way. The values are the thing to reach for when the density is to be drawn against something else, such as the normal a `lognormal_aft()` fit would have assumed.

See Also

`dpm()` and `dpm_aft()` for the families with an estimated error distribution; `bartisan()`

Examples

```
data("rhc")
set.seed(123)

# How long a patient survived, among those who died, so
# that the outcome is a complete rather than a censored
# time
died <- rhc[rhc$death == 1, ]

fit <- bartisan(log(days) ~ rhc + age + sex + race +
  edu + aps + meanbp + resp + hema +
  pafi + paco2 + crea + surv2m + card,
  data = died, family = dpm(),
  num_trees = 10, num_burn = 50,
  num_draws = 50)

# The shape of the errors, which a Gaussian fit would
# have assumed to be normal
head(error_density(fit, at = c(-2, 0, 2)))

# The same thing drawn, with the pointwise interval as a
# ribbon
plot(error_density(fit))
```

estimate_effect

Causal effects from a fitted model

Description

Estimates the average or conditional effect of a treatment by g-computation over the posterior draws. `estimate_effect()` predicts every unit under every level of the treatment, contrasts those potential outcomes within each draw, and averages over the units the estimand asks for.

Usage

```
estimate_effect(
  object,
  treat = NULL,
  estimand = "ATE",
  comparison = "difference",
  by = NULL,
  newdata = NULL,
  level = 0.95,
  interval = "eti",
  focal = NULL,
  type = "response"
)

## S3 method for class 'bartisan_effect'
print(x, digits = 3L, contrasts = NULL, potential_outcomes = TRUE, ...)

## S3 method for class 'bartisan_effect'
plot(x, marginal = TRUE, ...)
```

Arguments

object	a <bartisan_fit> object; the output of a call to bcf() or bartisan() .
treat	string; the name of the treatment variable. A fit from bcf() carries its own and needs none, so this is for a fit from bartisan() , where nothing marks one predictor as the treatment.
estimand	string; which units the effect is averaged over. "ATE" (the default) uses all of them, and "ATT" and "ATC" those in the focal group, which by default is the treatment's second level for the former and its first for the latter. With a binary treatment, that makes them the treated and the untreated without focal being named. "CATE" does not average at all and returns the conditional effect at each unit's covariates, which is not that unit's own individual effect; see Details. Abbreviations and lowercase spellings are allowed.
comparison	string; how the two potential outcomes are contrasted. Allowable options include "difference" (the default), "ratio", "lnratio", "or", and "lnor". The last two are available only when the response is a probability.
by	optional; a one-sided formula or a variable name naming a grouping variable, in which case the effect is averaged within each of its levels rather than over the whole sample. A formula is evaluated with stats::model.frame() rather than read for the names it mentions, so it can define a grouping the data has no column for, as in <code>by = ~ age > 50</code> or <code>by = ~ interaction(sex, region)</code> . It must give exactly one grouping variable, and the term as written names the column it occupies in the output.
newdata	optional; a data frame of units to average over. Default is the data the model was fit to, so that the average runs over the covariates of the fitted sample; see Details for what that average is.
level	numeric; the level of the credible interval. Default is .95.

interval	string; "eti" (the default) for an equal-tailed interval from the quantiles of the draws, or "hpdi" for the highest posterior density interval, which is the shortest interval containing level of the posterior mass.
focal	optional; the treatment level whose units the effect is averaged over for "ATT" and "ATC". With a binary treatment it defaults to the second level for "ATT" and the first for "ATC", which makes them the effect among the treated and among the untreated. With a treatment of more than two levels it has no default and must be supplied, and then "ATT" and "ATC" differ only in the level named.
type	string; the prediction scale the effect is computed on, passed to <code>predict.bartisan_fit()</code> . Default is "response", which is the scale on which an average of unit-level differences is the marginal effect. See Details before changing it.
x	a <code><bartisan_effect></code> object; the output of a call to <code>estimate_effect()</code> .
digits	integer; the number of significant digits to print.
contrasts	string; which contrasts to display when the treatment has more than two levels. "focal" (the default when a focal group is known) shows only the contrasts involving it, "all" shows every pairwise contrast, and a character vector of contrast labels shows those. All of them are computed either way; this only decides what is printed.
potential_outcomes	logical; whether to print the average response under each treatment level below the contrasts, those being what the contrasts were computed from. Default is TRUE. They are in the result's "potential_outcomes" attribute either way.
...	ignored.
marginal	logical; for <code>plot()</code> on conditional effects (i.e., <code>estimand = "CATE"</code>), whether to draw the marginal effect beside the units. Default is TRUE. Ignored by the other plots, which draw no marginal effect beside their estimates.

Details

The Estimand:

The ATE, ATT and ATC are averages of the conditional effect over the covariates of the units averaged over, computed within each posterior draw. This is sometimes called the mixed average treatment effect. The covariates are treated as fixed, so the interval reflects uncertainty about the outcome model but not the further variation a population average would carry, and the units' observed outcomes are not conditioned on, as they would be for a sample average of individual effects. Likewise, `estimand = "CATE"` reports the expected effect at each unit's covariates rather than the unit's own effect, which depends on how its two potential outcomes are associated and is not identified. See `vignette("causal")` for the distinction between these estimands.

Setting type:

A varying coefficient is a contrast on the link scale: on a `binomial("logit")` fit, `coef(object)` is a per-unit difference in log odds, and the average of those is the average conditional log odds ratio rather than the marginal one a treatment question usually asks for. The default is therefore `type = "response"`, where every unit's contrast is on the scale the response is measured on and averaging them gives the marginal effect.

Setting `type = "link"` is the right choice for looking at how the effect varies, since that is the scale the forest models it on, and the wrong one for reporting an average.

Setting comparison:

For "ratio", "lnratio", "or" and "lnor", the potential outcomes are averaged over units first and contrasted afterward, which gives the marginal ratio. With estimand = "CATE" there is no averaging to do, so a ratio reported there is a conditional ratio and the `print()` method says so.

For the same reason, "or" and "lnor" are not each other's `exp()` and `log()`. Each summarizes the posterior of the quantity it names, and a posterior mean does not survive a nonlinear transformation: "or" reports the mean of the odds ratio and "lnor" the mean of its logarithm, which exponentiates to something smaller. Report whichever scale the interval should be symmetric on, which for a ratio is usually the log. The same holds of "ratio" against "lnratio".

Setting focal:

With a binary treatment and no focal, which level is the treated one is worked out from the treatment's values by the rules `WeightIt::weightit()` uses, so that the same variable is read the same way by both packages. In order: a logical treatment is treated at TRUE; a numeric one is untreated at 0; a character or factor one whose levels parse as numbers is untreated at 0, or else treated at the largest; and otherwise the conventional names are matched, "t", "tr", "treat", "treated" and "exposed" against "c", "co", "ctrl", "control" and "unexposed". When none of those applies the second level is taken as the treated one and a message says so, that being the case where the guess can be wrong. "ATC" takes the other level as its focal group, so both estimands average over the group focal names and only the default differs.

Multi-category Treatments:

There is nothing to work out from the values, so the focal argument is required for "ATT" and "ATC", and those two then name the same estimand: the effect among the units in the level named.

Every pairwise contrast is computed. Which ones are shown is a display choice, made by contrasts in the `print()` method: with a focal group the default is to show only the contrasts involving it, matching how `WeightIt::weightit()` and `cobalt::bal.tab()` use focal.

The effect of a continuous treatment is a slope or a dose-response curve rather than a contrast of levels, and the **adrf** package has tools for summarizing and visualizing one.

Value

A `<bartisan_effect>` object, a data frame with one row per reported effect and columns contrast, estimate, lower, and upper, plus unit when estimand = "CATE" and the by variable's name when by is used. The posterior draws of every reported quantity are kept in the "draws" attribute, and the marginal mean of each potential outcome in "potential_outcomes", so a caller can re-contrast or re-summarize without refitting.

`print()` returns its input invisibly, and prints the potential outcomes below the contrasts unless `potential_outcomes = FALSE`. `plot()` returns a **ggplot2** object.

See Also

`bcf()`, which fits the model this is usually called on; `print.bcf_fit()` for its other methods; `bartisan-marginaleffects` for the same estimands through **marginaleffects**, which also covers the ones not offered here

Examples

```

data("rhc")
set.seed(123)

# Note: we use small sampler settings that would not be
# appropriate for analysis; this is just for the
# examples
fit <- bcf(death ~ age + sex + meanbp + aps,
           treat = ~ rhc, data = rhc, num_trees = 10,
           num_burn = 50, num_draws = 50,
           propensity_args = list(num_trees = 10,
                                  num_burn = 50,
                                  num_draws = 50),
           verbose = FALSE)

# The risk difference, averaged over everyone
estimate_effect(fit)

# Among the treated, and as a risk ratio rather than a
# difference
estimate_effect(fit, estimand = "ATT",
                comparison = "ratio")

# The effect at each unit's covariates, ordered, with
# the marginal effect beside them
cate <- estimate_effect(fit, estimand = "CATE")
plot(cate)

# By a subgroup, which is where a forest plot earns its keep
estimate_effect(fit, by = ~ sex)

```

partial_dependence	<i>Partial dependence on one or two predictors</i>
--------------------	--

Description

Averages the fitted surface over the sample at each value of one or two predictors, so that what is left is how the prediction moves with them. `partial_dependence()` returns the values and `plot()` draws them.

Usage

```

partial_dependence(
  object,
  variables,
  newdata = NULL,
  grid = 26L,
  values = NULL,

```

```

    level = 0.95,
    interval = "eti",
    type = "response",
    ...
)

## S3 method for class 'bartisan_partial'
print(x, digits = 3L, n_print = 10L, ...)

## S3 method for class 'bartisan_partial'
plot(x, ...)

## S3 method for class 'bartisan_fit'
plot(x, y, ...)

```

Arguments

object	a <bartisan_fit> object; the output of a call to bartisan() .
variables	a one-sided formula naming one or two predictors, as in <code>~ age</code> or <code>~ age + sex</code> , or a character vector of their names.
newdata	optional; a data frame to average over. Default is the data the model was fit to.
grid	integer; how many values of a numeric predictor to evaluate. Default is 26. A factor is evaluated at each of its levels whatever this is, and a numeric predictor named second at three values rather than grid of them; see Details.
values	optional; a named list giving the values to evaluate a predictor at, which overrides grid for the predictors it names. An entry may be a function of the predictor rather than the values themselves, so that <code>values = list(age = unique)</code> evaluates age at every value it takes; NA is dropped from what such a function returns, while a vector written out by hand is used exactly as written. A numeric second predictor this does not name is held at three of its values near its quartiles, and a message reports which.
level	numeric; the level of the credible interval. Default is .95.
interval	string; "eti" (the default) for an equal-tailed interval from the quantiles of the draws, or "hpdi" for the highest posterior density interval, which is the shortest interval containing level of the posterior mass.
type	string; the prediction scale, passed to predict.bartisan_fit() . Default is "response".
...	for plot.bartisan_fit() , further arguments passed to partial_dependence() ; for partial_dependence() , further arguments passed to predict.bartisan_fit() .
x	a <bartisan_partial> object; the output of a call to partial_dependence() . For plot.bartisan_fit() , a <bartisan_fit>.
digits	integer; for print() , the number of significant digits to print the estimates and their interval to. Default is 3.
n_print	integer; for print() , the total number of rows to show, taken half from the top of the grid and half from the bottom, with the odd row going to the top. A line

between the two halves counts what was left out. Default is 10, and Inf shows every row.

`y` for `plot.bartisan_fit()`, the predictors to plot, as variables above.

Details

At each grid value every unit is assigned that value, the prediction is taken for all of them, and the average over units is taken *within each posterior draw*. The interval is then computed from those averages, so it is an interval on the average prediction and not on any one unit's.

The second predictor groups the curves rather than adding an axis, which is readable for a factor and for a numeric predictor with a few values, and not for a continuous one: grid values of it would give grid curves, each with a ribbon of its own. A numeric second predictor with more than three distinct values is therefore held at three of them and a message says which, with the `values` argument there to choose others and `values = list(z = unique)` the short way to ask for all of them, which usually suits a predictor with four or five values. The three are the values nearest its quartiles, and they are distinct even when the quartiles are not, since a predictor with a large mass at one value takes that value for two or three of them; each quartile in turn takes the nearest value the predictor has that an earlier one did not take. They are values the predictor takes rather than points on an even grid, which also keeps the legend readable.

The usual caveat on a partial dependence plot applies. Averaging over the other predictors evaluates the model at covariate combinations that may not occur, so a curve over a region where the predictor's values are sparse says more about the prior than about the data, and the result summarizes the fitted function rather than supporting a causal claim; `estimate_effect()` is for a contrast that is meant causally.

Value

A `<bartisan_partial>` object, a data frame with one row per grid point, columns for the predictors, and estimate, lower and upper. `plot()` returns a **ggplot2** object.

See Also

`estimate_effect()` for treatment effects rather than fitted surfaces; `variable_importance()` for which predictors the forest uses; `bartisan-marginaleffects`, since `marginaleffects::plot_predictions()` draws the same thing with more control over the grid

Examples

```
data("rhc")
set.seed(123)

# Note: we use small sampler settings that would not be
# appropriate for analysis; this is just for the
# examples
fit <- bartisan(death ~ age + sex + meanbp + aps,
               data = rhc, num_trees = 10, num_burn = 50,
               num_draws = 50, verbose = FALSE)

# How the fitted risk moves with mean blood pressure
```

```

pd <- partial_dependence(fit, ~ meanbp)
pd

plot(pd)

# The same thing from the fit, using the `plot()` method
plot(fit, ~ meanbp)

# Two predictors, one of them a factor, which gives a
# curve per level
plot(fit, ~ meanbp + sex)

# Two numeric predictors, where the second is held at
# three values and a message says which
plot(fit, ~ meanbp + aps)

# An entry of `values` may be a function of the
# predictor, which is how to ask for values of your own
# without naming them
three_values <- function(x) quantile(x, c(.1, .5, .9))
plot(fit, ~ meanbp + aps, values = list(aps = three_values))

```

predict.bartisan_fit *Predictions from a generalized BART model*

Description

Computes predictions from a `bartisan()` fit, at the data the model was fit to or at new data, on any of several scales. Because every posterior draw of every tree is retained, a prediction can be returned as its posterior mean or as the draws themselves.

Usage

```

## S3 method for class 'bartisan_fit'
predict(
  object,
  newdata = NULL,
  type = "response",
  draws = FALSE,
  iterations = NULL,
  offset = NULL,
  weights = NULL,
  values = NULL,
  log = FALSE,
  times = NULL,
  ...
)

```

Arguments

object	a <bartisan_fit> object; the output of a call to bartisan() .
newdata	optional; a data frame at which to predict. Default is NULL to use the data the model was fit to. Missing predictor values are allowed in the columns that had them when the model was fit, since only those columns' splitting rules carry an answer for one; a missing value anywhere else is an error. See section <i>Missing Predictor Values</i> in the Details of bartisan() .
type	string; the scale of the prediction. Allowable options include "link", "response" (the default), "prob", "class", "mean", "stdlv", "density", "survival", and "hazard". "link" the additive predictor, one column per predictor for families that have more than one. For the accelerated failure time families it is on the scale of log time, so a difference between two units is a log time ratio. For ph() it is reported centered over the fitted sample, so a single value is a log hazard ratio against a unit whose predictor sits at that average and a difference between two units is the log hazard ratio between them. "response" the mean of the response; the median survival time for every survival family, ph() included; and, for a response with categories, the category probabilities, there being no single mean to report. "prob" category probabilities, for the binomial, ordinal and multinomial families. "class" the most probable category, as a factor, for the same families. "mean" the mean of the response with the category labels read as numbers, for the same families, so that "4" counts as four. The values argument says what the categories are worth where the labels are not the numbers intended. "stdlv" the additive predictor on the scale of the latent variable it indexes, for the ordinal and binomial families under the probit, logit and complementary log-log links. This puts fits with different links, or different amounts of signal, on one scale. See Details. "density" the conditional density of the outcome given the predictors, evaluated at the observed outcome, so the newdata argument must carry the outcome and leaving it empty uses the fitted data. This is the observation's likelihood contribution, so it is a probability for a discrete response and a survival probability for a censored time. See Details. "survival" the survival function $S(t x)$ at the times given in times, for the accelerated failure time families and ph(). Returns one column per time, or a draws by rows by times array when draws = TRUE. This makes the usual survival estimand reachable through marginaleffects; see bartisan-marginaleffects. "hazard" the hazard $h(t x) = f(t x) / S(t x)$ at the times given in times, the rate at which the event occurs at t among those who have survived to it, for the same families and in the same shape as "survival". The cumulative hazard is $-\log S(t x)$ and so comes from "survival".
draws	logical; whether to return every posterior draw rather than the posterior mean. Default is FALSE to return the mean. If TRUE, the result gains a leading dimension indexing the draws.

iterations	numeric; optional indices of the stored draws to use, between 1 and the number of draws the fit retains. Default is NULL to use all of them.
offset	numeric; an offset for the rows of newdata, on the link scale, given either as one value per row or as a matrix with one column per additive predictor. Used only when newdata is supplied, and required there when the model was fit with an observation-level offset, since an offset is not a function of the predictors and so cannot be reconstructed. Default is NULL to add nothing.
weights	numeric; prior weights for the rows of newdata, used only by type = "density" and ignored otherwise. For a binomial response given as proportions these are the numbers of trials, as in <code>stats::glm()</code> . Default is NULL, which uses the weights the model was fit with when newdata is omitted and a weight of 1 for every row when it is supplied.
values	numeric; for type = "mean", what each response category is worth, given as a vector named for every level of the response. Default is NULL to read the level labels as numbers, which is an error rather than a guess when they cannot be read that way. Ignored with a warning for every other type.
log	logical; for type = "density", whether to return the log of the value. Default is FALSE. Summing the log density across observations gives a log score. Note that with draws = FALSE the density is averaged over the draws before the log is taken, so the result is the pointwise predictive density rather than the average log density.
times	numeric; for type = "survival" and type = "hazard", the times at which to report the survival function or the hazard, which must be finite and strictly positive. It has no default, because the horizon is a choice rather than a property of the fit. Ignored with a warning for every other type.
...	ignored; present for compatibility with the generic.

Details

Simulated Multinomial Probit Probabilities:

The likelihood of a `multinomial("probit")` fit has no closed form: the probability of a category is the chance that the largest of several correlated Gaussian variables is the one belonging to it, which is a multivariate orthant probability. So "prob", "class", "response" and "density" are all simulated, using the number of replicates the family was given. Fresh draws are taken on every call, so two calls differ by Monte Carlo error; that error is per posterior draw and averages down over them, which makes draws = FALSE much more accurate than any single row of draws = TRUE.

type = "stdlv":

This reports $(\eta - E[e]) / \text{sd}(y^*)$ for the latent $y^* = \eta + e$, following `WeightIt::predict.ordinal_weightit()`, and makes an ordinal or binary fit comparable across links.

The scale is $\text{sd}(y^*) = \sqrt{\text{var}(\eta) + \text{var}(e)}$, where $\text{var}(\eta)$ is taken over the sample the model was fitted to, per draw, so it is a property of the model rather than of whatever is being predicted and the same divisor is used for new data. $\text{var}(e)$ is whatever the link implies: 1 for the probit link, $\pi^2/3$ for the logit, $\pi^2/6$ for the complementary log-log. The location subtracts the latent error's mean, which is invisible for the logit and probit links, whose errors are already centered, and is the whole of the difference for the complementary log-log.

Such a model is identified only up to a common shift of its thresholds and its predictor, so the location here is a convention, and the one used is the same the cutpoints use: a predictor centered over the fitted sample. Differences on this scale, which a standardized quantity exists to support, are unaffected by that choice.

`type = "density":`

The measure differs across the survival families: the accelerated failure time families, `dpm_aft()` included, report the density of $\log T$, where `ph()` reports the density of T . The two differ by $\sum \log t$, so log scores are comparable within each group rather than across them, and the `type = "survival"` prediction is comparable throughout. `loo()` takes a scale argument that puts them on one measure.

Setting `type = "density"` returns `NaN` for an observation whose density is undefined at any of the draws, which happens when a link the package does not compile has been composed onto the family's own scale and its inverse does not cover the whole additive predictor (e.g., `poisson("identity")`), which gives a positive mean only where the predictor is positive). A warning reports how many draw-by-observation values were undefined and how many returned values that made `NaN`. The draws are averaged before the log is taken, so one undefined draw is enough to make an observation `NaN`.

Value

With `draws = FALSE`, a vector with one element per observation for `"response"`, `"mean"` and `"density"`, and for `"link"` and `"stdlv"` when the family has a single additive predictor; a matrix of observations by additive predictors for `"link"` and `"stdlv"` when it has more than one; a matrix of observations by categories for `"prob"`, and for `"response"` with an ordinal or multinomial family; a matrix of observations by times for `"survival"` and `"hazard"`; and a factor for `"class"`, ordered when the family is ordinal().

With `draws = TRUE` each of these gains a leading dimension indexing the posterior draws, so that a vector becomes a matrix of draws by observations and a matrix becomes an array of draws by observations by categories or by times. On the `"link"` and `"stdlv"` scales a family with more than one additive predictor returns a list with one draws-by-observations matrix per predictor. `"class"` is a factor either way.

See Also

`bartisan()` for fitting the model; `bartisan-marginaleffects` for averages and contrasts of these predictions; `bartisan-interop` for the methods that let other packages assess the fit

Examples

```
data("rhc")
set.seed(123)

# Whether a patient died, with the treatment and every
# covariate recorded on admission as candidate
# predictors
model <- death ~ rhc + age + sex + race + edu + aps +
  meanbp + resp + hema + pafi + paco2 + crea + surv2m +
  card
```

```

fit <- bartisan(model, data = rhc, num_trees = 10,
               num_burn = 50, num_draws = 50)

# Fitted probabilities of death, averaged over the draws
head(predict(fit, type = "response"))

# The whole posterior for the first five patients rather
# than its mean
post <- predict(fit, newdata = rhc[1:5, ], draws = TRUE)
apply(post, 2, quantile, c(.025, .5, .975))

# A held-out log score, which needs the outcome, so the
# `newdata` argument carries it
dens <- predict(fit, newdata = rhc[1:100, ],
               type = "density")
sum(log(dens))

```

print.bartisan_fit	<i>Summarize a generalized BART model</i>
--------------------	---

Description

print() reports what was fit and how long the chain is. summary() reports what print() does not: posterior summaries of the nuisance parameters and the random-effect scales, a quick check of convergence, and the most used predictors, with pointers to the functions that examine each of these in full.

Usage

```

## S3 method for class 'bartisan_fit'
print(x, digits = 3L, ...)

## S3 method for class 'bartisan_fit'
summary(object, level = 0.95, ...)

## S3 method for class 'summary.bartisan_fit'
print(x, digits = 3, ...)

```

Arguments

x, object	a <bartisan_fit> object; the output of a call to bartisan() . For print.summary.bartisan_fit(), x is instead the <summary.bartisan_fit> object summary() returned.
digits	numeric; the number of significant digits to print. Default is 3.
...	not used.
level	numeric; the width of the posterior intervals summary() reports. Default is .95 for 95% intervals.

Details

The printed summary is a starting point for the functions that examine a fit in full, and it computes only what is cheap. Its convergence line gives R-hat and the bulk and tail effective sample sizes of the log likelihood, which take milliseconds because the log likelihood is a single series. It can look fine while individual fitted values mix badly, which is what `diagnose()` checks, at a cost that grows with the number of observations. The variable importance table shows at most the five most used predictors of the first forest; `variable_importance()` gives every predictor in every forest. The summary also points to `loo()` and `kfold()` for comparing fits and to `partial_dependence()` for how the predictions depend on a predictor, none of which it runs, since each can take seconds or more on a large fit.

Value

`print()` returns its argument invisibly. `summary()` returns a `<summary.bartisan_fit>` object, a list of the posterior summaries its own `print()` method displays: the nuisance parameters when the family has any, the scale of each random-effect term, the splitting counts of each predictor group, the output of `variable_importance()`, and R-hat and the effective sample sizes of the log likelihood.

See Also

`bartisan()`; `variable_importance()` for the splitting counts as a data frame rather than printed; `diagnose()` for whether the chains the summaries are computed from have converged

Examples

```
data("rhc")
set.seed(123)

# Whether a patient died, with catheterization among the
# predictors
fit <- bartisan(death ~ rhc + age + sex + race + edu +
               aps + meanbp + resp + hema + pafi +
               paco2 + crea + surv2m + card,
               data = rhc, num_trees = 10, num_burn = 50,
               num_draws = 50, chains = 2, verbose = FALSE)

# What was fit, and how many draws it rests on
fit

# The splitting counts, which say which predictors the
# forest reaches for
summary(fit)

# A family with a nuisance parameter reports its
# posterior too, here the residual standard deviation of
# the log survival time
fit2 <- bartisan(log(days) ~ rhc + age + sex + race +
                 edu + aps + meanbp + resp + hema +
                 pafi + paco2 + crea + surv2m + card,
                 data = rhc, family = gaussian(),
```

```

num_trees = 10, num_burn = 50,
num_draws = 50, verbose = FALSE)

summary(fit2, level = .8)

```

print.bcf_fit

Methods for Bayesian causal forest fits

Description

A fit from `bcf()` is a `<bartisan_fit>` with a treatment named, so it takes every method that fit takes. `print()` adds the treatment and says where the effect comes from, and `plot()` draws the conditional effects.

Usage

```

## S3 method for class 'bcf_fit'
print(x, digits = 3L, ...)

## S3 method for class 'bcf_fit'
plot(x, level = 0.95, comparison = "difference", marginal = TRUE, ...)

```

Arguments

<code>x</code>	a <code><bcf_fit></code> object; the output of a call to <code>bcf()</code> .
<code>digits</code>	integer; the number of significant digits to print.
<code>...</code>	for <code>plot()</code> , further arguments passed to <code>estimate_effect()</code> ; otherwise ignored.
<code>level</code>	numeric; the level of the credible interval. Default is .95.
<code>comparison</code>	passed to <code>estimate_effect()</code> , which computes the numbers.
<code>marginal</code>	logical; whether to draw the marginal effect beside the conditional ones. Default is TRUE.

Details

The effect itself is `estimate_effect()`'s to report, and `summary()` on a `<bcf_fit>` is the same summary of the forests it is on any other fit, with a line at the end saying so. That way the same call prints the same thing whether the model came from `bcf()` or from `bartisan()` with a `vc()` term.

Value

`plot()` returns a **ggplot2** object. `print()` returns its input invisibly.

See Also

`estimate_effect()` for the effect and its estimands; `bcf()` for the model

Examples

```
data("rhc")
set.seed(123)

# Note: we use small sampler settings that would not be
# appropriate for analysis; this is just for the
# examples
fit <- bcf(death ~ age + sex + meanbp + aps,
           treat = ~ rhc, data = rhc, num_trees = 10,
           num_burn = 50, num_draws = 50,
           propensity_args = list(num_trees = 10,
                                  num_burn = 50,
                                  num_draws = 50),
           verbose = FALSE)

fit

# The effect, with the potential outcomes it is a
# difference of
estimate_effect(fit)

# The conditional effects, ordered, with the marginal
# effect beside them
plot(fit)
```

ranef.bartisan_fit	<i>Group intercepts from a random-effect term</i>
--------------------	---

Description

Extracts the intercept each level of a grouping factor was given by a (1 | group) term in the formula, as a posterior mean or as every draw.

Usage

```
## S3 method for class 'bartisan_fit'
ranef(object, draws = FALSE, ...)
```

Arguments

object	a <bartisan_fit> object; the output of a call to bartisan() , fitted with at least one (1 group) term.
draws	logical; whether to return every posterior draw of each intercept rather than its posterior mean. Default is FALSE.
...	not used.

Details

The column is named (Intercept), as in **lme4**, when the family has a single additive predictor. A family with more than one gets a group intercept on each, independent of the others, and the columns are named for the predictors instead; `vignette("families")` lists them per family.

Only the intercepts are returned. The standard deviation each grouping factor was drawn under is in `object$tau`, one column per factor and one matrix per additive predictor, and `prior_summary()` reports the prior it was drawn from.

A posterior mean is the wrong summary for a level with few observations, which is the case a group intercept exists for. Setting `draws = TRUE` gives an interval, and a level whose interval covers zero is one the data had little to say about.

The intercepts are shrunk towards zero by their prior and are deviations from the additive predictor, so they come out approximately centered without being constrained to sum to zero exactly. Nothing is lost by that: the level of the fitted function is the additive predictor's, and a shift common to every intercept is one the forest did not take.

Value

With `draws = FALSE`, a named list with one entry per grouping factor, each a data frame of one row per level of that factor and one column per additive predictor that has a group intercept, with the levels as row names. This is the shape `lme4::ranef()` returns, so anything that reads that reads this.

With `draws = TRUE`, a named list with one entry per grouping factor, each itself a named list of draws-by-levels matrices, one per additive predictor.

See Also

`bartisan()` for the `(1 | group)` syntax and what it fits; `prior_summary()` for the prior on these

Examples

```
set.seed(123)
sites <- factor(sample(letters[1:5], 200, TRUE))
d <- data.frame(x = runif(200), site = sites)
d$y <- rnorm(200, d$x + as.numeric(d$site) / 3)

fit <- bartisan(y ~ x + (1 | site), data = d,
               num_trees = 10, num_burn = 50,
               num_draws = 50, verbose = FALSE)

# One intercept per site, as posterior means. The
# generic is \pkg{nlme}'s, which \pkg{lme4} re-exports,
# so either qualification reaches this.
nlme::ranef(fit)

# With the draws, so the intercepts come with intervals
re <- nlme::ranef(fit, draws = TRUE)
apply(re$site[["(Intercept)"]], 2L, quantile,
      c(.025, .975))
```

rhc

*Right heart catheterization in critically ill patients***Description**

Data from the SUPPORT study on whether right heart catheterization within 24 hours of admission to an intensive care unit affects survival (Connors et al., 1996). Catheterization was not randomized, so the comparison is confounded by how sick each patient was on admission, which the physiological covariates are there to adjust for.

Usage

rhc

Format

A data frame with 1500 rows and 16 columns.

rhc whether the patient received right heart catheterization, 1 or 0. This is the treatment.

death whether the patient died during follow-up, 1 or 0.

days days from admission to death, or to last contact for a patient who did not die. Together with death this is the survival outcome.

age age in years.

sex "female" or "male".

race "white", "black" or "other".

edu years of education.

aps APACHE III score on day 1, ignoring coma. Higher is sicker.

meanbp mean blood pressure on day 1.

resp respiratory rate on day 1.

hema hematocrit on day 1.

pafi ratio of arterial oxygen to inspired oxygen on day 1.

paco2 arterial carbon dioxide on day 1.

crea serum creatinine on day 1.

surv2m the study's own model-based estimate, made on day 1, of the probability of surviving two months.

card whether cardiovascular disease was a diagnosis, "no" or "yes".

Details

The outcome appears in two forms. death is whether the patient died during follow-up, and days is how long that took, so the same event supports a binary analysis that ignores timing and a right-censored survival analysis that does not. Note that a patient who did not die is censored at their last contact.

These are a random 1500 of the 5735 patients in the original file. The covariates are the thirteen used in the worked example at <https://iqss.github.io/dss-ps/example.html>, all recorded before catheterization; the full study collected many more.

The treatment and the outcome are coded 0 and 1 rather than as factors, so that a contrast between them is a single number rather than one per level.

Nothing here makes the causal assumptions hold. Whether the effect of rhc on death can be read causally depends on whether these covariates account for how patients were selected for catheterization, which is a question about the study rather than about any model. See vignette("causal").

Source

Assembled from <https://hbiostat.org/data/repo/rhc.csv> by data-raw/rhc.R.

References

Connors, A. F., Speroff, T., Dawson, N. V., et al. (1996). The effectiveness of right heart catheterization in the initial care of critically ill patients. *JAMA*, 276(11), 889–897. doi:10.1001/jama.1996.03540110043030

Examples

```
data("rhc")

# The treatment against the binary outcome
table(rhc$rhc, rhc$death)

# The covariates the confounding runs through, all
# recorded on admission
summary(rhc[c("age", "aps", "meanbp", "surv2m")])

# The same event as a survival outcome, a patient who
# did not die being censored at their last contact
if (rlang::is_installed("survival")) {
  with(rhc, summary(survival::Surv(days, death)))
}
```

variable_importance	<i>How often each predictor is used</i>
---------------------	---

Description

Reports, for every predictor, how many splitting rules the forest spends on it and how often it is used at all, which is the meaning of "variable importance" for a BART model. This is the table `summary.bartisan_fit()` prints, returned as a data frame rather than displayed (i.e., ready to sort, filter, or plot).

Usage

```
variable_importance(object, level = 0.95, draws = FALSE)
```

```
## S3 method for class 'bartisan_importance'
plot(x, y, ...)
```

Arguments

object	a <bartisan_fit> object; the output of a call to bartisan() .
level	numeric; the width of the interval reported for splits. Default is .95 for a 95% interval.
draws	logical; whether to return the splitting counts of every posterior draw rather than a summary of them, for a comparison the summary does not offer. Default is FALSE.
x	a <bartisan_importance> object; the output of a call to <code>variable_importance()</code> .
y	not used.
...	not used.

Details**Reading the Columns:**

What `prop_used` means depends on the splitting prior. Under the default (`sparsity = FALSE` in [bartisan_control\(\)](#)), every predictor keeps a share of the rules and `prop_used` sits near 1 throughout, which the `print()` method notes; the ranking is then carried by the other two columns. With `sparsity = TRUE`, it behaves like a posterior probability that the predictor belongs in the model, separates signal from noise sharply, and is the column to read first.

`prop_splits` is the one to reach for when two fits are being compared. The share is computed within each draw before averaging, so the shares add to one whatever the forest size, where `splits` counts rules and so scales with `num_trees`. `splits` itself, the mean number of rules per draw, says how much of the forest's structure a predictor accounts for, and is the easier of the three to over-read.

Limits of a Usage Ranking:

Usage is not effect size: a predictor can be split on constantly and move the prediction very little.

Where the question is how much a predictor moves the outcome, [marginaleffects::avg_comparisons\(\)](#) on the fitted model answers it; see [bartisan-marginaleffects](#). Where two predictors carry the same information the trees split on whichever is convenient and the usage distributes between them arbitrarily, so a group of correlated predictors is best read as a group. And a ranking by usage describes this fitted function rather than what would happen if a predictor were changed.

Reading It as Variable Selection:

When the question is which predictors the model needs, the fit should be made with `sparsity = TRUE`. `prop_used` is then usable as a selection rule: the predictors the forest genuinely needs sit near 1 and the rest fall near 0, usually with a wide gap rather than a continuum, and cutting that gap at .5 gives the median probability model. No threshold is correct in general, so the gap is the thing to look at, and a conclusion worth reporting will not depend on where in it the cut is made. A forest asked to fit noise still puts its rules somewhere, so the cut chooses a predictive

submodel rather than testing one. `vignette("importance")` calibrates it against noise predictors and works through correlated ones.

Value

A `<bartisan_importance>` object, which is a data frame with one row per predictor and its own `print()` and `plot()` methods, sorted by `prop_used` and then `splits`, both decreasing, and with the following columns:

- `variable`: the predictor, under the name the formula gave it
- `prop_used`: the proportion of draws in which it received at least one rule
- `prop_splits`: its share of all the splitting rules in the forest, averaged over draws
- `splits`: the mean number of splitting rules per draw that use it
- `splits_lower` and `splits_upper`: the endpoints of the level interval on that number, which the `print()` method leaves out of the displayed table

A family with more than one additive predictor has a forest for each, and the data frame then gains a leading predictor column naming which.

With `draws = TRUE`, the draws-by-predictors matrix of counts instead, or a named list of them when there is more than one forest.

`plot()` on the result draws `splits` and its interval for each predictor and returns a **ggplot2** object. The result can be subset first, so `plot(vi[vi$prop_used > .5, ])` draws only the predictors that are used.

See Also

`summary.bartisan_fit()`, which prints the same table; `bartisan_control()` for sparsity; `bartisan-marginaleffects` for effects rather than usage

Examples

```
data("rhc")
set.seed(123)

model <- death ~ rhc + age + sex + race + edu + aps +
  meanbp + resp + hema + pafi + paco2 + crea + surv2m +
  card

# The sparsity prior concentrates the splitting rules on
# the predictors that earn them, which makes `prop_used`
# readable as a selection rule
fit <- bartisan(model, data = rhc, num_trees = 10,
               num_burn = 50, num_draws = 50,
               sparsity = TRUE, verbose = FALSE)

imp <- variable_importance(fit)
imp

# The predictors the forest reaches for in nearly every
# draw
```

```

subset(imp, prop_used > .9)

# `prop_splits` is the column that survives a change of
# forest size, since the shares add to one however many
# rules there are to share
big <- bartisan(model, data = rhc, num_trees = 40,
               num_burn = 50, num_draws = 50,
               sparsity = TRUE, verbose = FALSE)

shares <- c("variable", "prop_splits")
merge(variable_importance(fit)[shares],
       variable_importance(big)[shares],
       by = "variable", suffixes = c("_10", "_40"))

# The counts themselves, for a comparison the summary
# does not make
counts <- variable_importance(fit, draws = TRUE)
mean(counts[, "aps"] > counts[, "meanbp"])

# The ranking, drawn. Subsetting first keeps a wide
# model readable.
plot(head(imp, 8))

```

vc

Give a predictor a varying coefficient

Description

Marks a predictor inside a `bartisan()` formula as one whose coefficient is a function of the other predictors rather than a constant, so that the coefficient gets a forest and a prior of its own. `vc()` is a marker rather than a function: it is read out of the formula it appears in and throws an error if it is called on its own.

Usage

```
vc(x, modifiers = NULL, center = "auto")
```

Arguments

<code>x</code>	the predictor whose coefficient varies, named as a bare variable rather than as an expression. A numeric variable gets one forest; a factor gets one per level, coded symmetrically (i.e., with no level held out as a reference) as <code>multinomial()</code> codes its predictors.
<code>modifiers</code>	a one-sided formula naming the variables this coefficient's forest may split on. Default is <code>NULL</code> to allow every predictor in the model except <code>x</code> itself. A <code>.</code> in this formula stands for those same predictors, so <code>~ .</code> is the default written out and <code>~ . + w</code> or <code>~ . - z</code> adds to or removes from it; see Details. A variable the model formula's fixed part does not carry may be named here, and it then modifies the

	coefficient without entering the control function. Naming something that is not a column of data at all is an error rather than a silent restriction.
center	the value of x at which the control function is read, given as either a string or a number. Allowable options include "auto" (the default), "mean", "zero", "mid", and "estimate". "mean" centers x , "zero" leaves it alone, "mid" uses the midpoint of its range, a number uses that number, and "auto" picks between "zero" and "mean" by the covariate. For a factor, center is "mean" or the name of a level to report against. "estimate" draws the coding rather than fixing it, and needs a covariate with between two and twenty distinct values. See Details.

Details

The model is

$$g(\mu_i) = f_0(Z_i) + \sum_j (X_{ij} - c_j) f_j(Z_i)$$

with a forest for the control function f_0 and one for each varying coefficient f_j . Every forest is fitted at once, so the coefficient has a prior of its own rather than being whatever difference a single forest with x among its predictors happens to produce.

Setting modifiers:

By default a coefficient may vary with every predictor in the model except x itself, and the modifiers argument narrows that.

Within modifiers, `.` stands for the predictors of the control function rather than, as in a model formula, for every column of the data. That makes `vc(z, ~ .)` the same model as `vc(z)`, and `vc(z, ~ . - x1)` the default with one modifier taken away, which is the short way to leave a coefficient free to vary with everything but one thing. `vc(z, ~ . + w)` adds one, and is how a variable the fixed part leaves out (below) is given to a coefficient without listing the rest. This is the sense `.` has in `update.formula()`, not the one it has in `lm()`.

It can also widen it. A variable named here that the fixed part leaves out is given a column of its own and reaches this coefficient's forest and no other, so the coefficient varies with it while the control function stays blind to it. That asymmetry is the point rather than a side effect: it is how an effect is allowed to depend on something a prognostic function must not see. The case that needs it is a modifier from which the treatment could be reconstructed: a control function allowed to see such a variable can represent the treatment itself, so the effect is not identified, even though the effect may legitimately vary with it. Naming it as a modifier alone keeps it out of the control function. Only an explicit modifiers formula reaches such a variable: a bare `vc(z)` means the predictors of the model, which these are not.

A covariate with a varying coefficient is kept out of the control function, since $f_0(Z) + z f_1(Z)$ is not identified when z is among f_0 's predictors: any function of z moves between the two. Reached through `.` it is dropped quietly, because `.` did not name it; named outright, the model is fitted as asked and a warning says why that is a choice.

A numeric covariate may modify its own coefficient, and that is how the effect stops being linear in it. `vc(z, ~ z + x1)` fits $z f_1(z, x_1)$, so the slope moves across z 's range and the dose response is a curve rather than a line, which is worth reaching for whenever the effect of a continuous predictor might not be proportional to it. A categorical covariate is removed from its own forests instead: a level's indicator is nonzero only on the rows where that level holds, and the variable

is constant on exactly those rows, so such a split separates rows that contribute from rows that contribute nothing.

At the other extreme, ~ 1 names nothing at all, which leaves the coefficient's forest no predictor to split on: every tree in it is a stump, so the coefficient is one drawn number and x enters as a linear term while the rest of the model stays nonparametric. It is drawn under the leaf prior rather than a prior written for a regression coefficient, so it is shrunk toward zero. Comparing $vc(z, \sim 1)$ against $vc(z)$ with `loo()` is a test of whether the effect of z varies at all; `vignette("varying")` works one through.

Setting center:

Centering is a reparameterization of f_0 alone: every coefficient and every estimand is identical under any choice, and the choice changes only what the control function means. "auto" picks by the covariate. For a 0/1 covariate it uses zero, so f_0 is the surface among the untreated, which is a quantity with its own meaning. For any other numeric covariate it uses the mean, because zero may be nowhere near the data and the control function there would be an extrapolation.

A factor is always fitted mean-centered and gets one forest per level, coded symmetrically rather than as contrasts against a level that happened to sort first. That coding leaves the reference a reporting choice: the center argument names the level `coef()` reports against, and no refit is needed to change it.

Drawing the Coding (center = "estimate"):

This is different in kind from the choices above. Rather than subtract a number from x , it gives each of x 's values a coefficient of its own and draws it, so that every contrast carries the same prior whatever the number of values and no value is a reference. `coef()` returns the identified contrasts.

At two values it restricts nothing, and `bcbf()` uses it: with it, the answer stops depending on which level was written as 1. Above two values every contrast becomes one shared shape times a scalar, where the symmetric coding gives each level its own, so it is the parsimonious model against a general one. Reach for it when the levels plausibly differ in degree rather than in kind.

Families with Several Additive Predictors:

`gaussian_ls()`, `zi_poisson()` and the rest fit one forest per distributional parameter, and each parameter's formula carries its own `vc()` terms. The forests are then two-dimensional (a control function and its coefficients, for each parameter) and named accordingly, mean, mean:z, log_sd and log_sd:z for $y \sim x_1 + x_2 + vc(z)$ under `gaussian_ls()`, the names per-forest settings are keyed by. One formula reaches every parameter, and a list of formulas gives each its own; `vignette("varying")` shows both.

So the same covariate may have a coefficient on more than one parameter: z shifting the mean and widening the spread are different questions, and both are answered at once. `coef()` returns one column per coefficient, named for its forest.

Some consequences follow from the parameters being different. A group intercept from $(1 | g)$ reaches every control function and no coefficient, since a group-varying coefficient is a random slope. And center = "estimate" is judged per parameter: the drawn coding needs a leaf target that is quadratic in the predictor it feeds, which `gaussian_ls()` is in the mean and is not in the log standard deviation, so the same request is accepted on one and refused on the other.

The two multinomial families are the exception and refuse `vc()`. Their forests are the levels of one parameter rather than separate parameters, identified only up to a function they all share, and

reporting removes it; a coefficient forest per level would add one such direction per coefficient and the reporting does not carry them.

`vignette("varying")` is the worked account of all of this: each argument on the `rhc` data, the constant-against-varying comparison, reading the fit with `coef()` and `estimate_effect()`, and how `bcf()` writes its model in these terms.

Value

Nothing; `vc()` is never evaluated, and calling it directly is an error.

See Also

- `vignette("varying")` for the guide
- `bartisan()` for the formula interface
- `bcf()` for the causal case, which is this term with the priors and the propensity score set up for it
- `coef.bartisan_fit()` for reading the coefficients out
- `bartisan-families` for the order the forests come in

Examples

```
data("rhc")

set.seed(123)

# The effect of right heart catheterization on death,
# free to vary with every other covariate. `rhc` is left
# out of the control function, which keeps the two
# forests identified
model <- death ~ age + sex + race + edu + aps + meanbp +
  resp + hema + pafi + paco2 + crea + surv2m + card +
  vc(rhc)

fit <- bartisan(model, data = rhc, family = binomial(),
  num_trees = 10, num_burn = 50,
  num_draws = 50)

# One coefficient per patient: the coefficient function
# at each observation
head(coef(fit))

# The same effect, free to vary with severity of illness
# alone
model2 <- death ~ age + sex + race + edu + aps + meanbp +
  resp + hema + pafi + paco2 + crea + surv2m + card +
  vc(rhc, ~ aps)

fit2 <- bartisan(model2, data = rhc, family = binomial(),
  num_trees = 10, num_burn = 50,
  num_draws = 50)
```

```
head(coef(fit2))
```

Index

* **datasets**
 rhc, 70

as_draws(), 3, 51, 52
as_draws.bartisan_fit
 (bartisan-interop), 20

bartisan, 5
bartisan(), 3, 4, 11, 14, 19, 21, 23, 24, 27,
 30, 37, 38, 40–45, 48–50, 54, 55, 59,
 61, 62, 64–69, 72, 74, 77
bartisan-families, 3, 11
bartisan-interop, 20
bartisan-marginaleffects, 26
bartisan-package, 3
bartisan_control, 30
bartisan_control(), 3, 4, 6, 7, 10, 18, 28,
 30, 40, 51, 52, 72, 73
bayesplot::available_ppc(), 22
bayesplot::ppc_dens_overlay(), 22
bcf, 39
bcf(), 3, 24, 37, 55, 57, 67, 76, 77
Beta(bartisan-families), 11
binomial(), 41

cbind(), 45
cobalt::bal.tab(), 57
coef(), 42, 76, 77
coef.bartisan_fit, 43
coef.bartisan_fit(), 77
custom_family, 45
custom_family(), 4, 5, 7, 19, 25, 34, 40

diagnose, 49
diagnose(), 3, 23, 25, 31, 32, 35, 46, 47, 66
dpm(bartisan-families), 11
dpm(), 42, 53, 54
dpm_aft(bartisan-families), 11
dpm_aft(), 4, 53, 54

error_density, 53
error_density(), 15, 19
estimate_effect, 54
estimate_effect(), 3, 42, 43, 49, 52, 60, 67,
 77

family, 4
family.bartisan_fit
 (bartisan-marginaleffects), 26
fitted.bartisan_fit(bartisan-interop),
 20
formula.bartisan_fit
 (bartisan-marginaleffects), 26
future::plan(), 31, 38

Gamma_ls(bartisan-families), 11
gaussian(), 41, 42
gaussian_ls(bartisan-families), 11
get_coef.bartisan_fit
 (bartisan-marginaleffects), 26
get_data.bartisan_fit
 (bartisan-marginaleffects), 26
get_group_names.bartisan_fit
 (bartisan-marginaleffects), 26
get_predict.bartisan_fit
 (bartisan-marginaleffects), 26
get_vcov.bartisan_fit
 (bartisan-marginaleffects), 26

kfold(), 66
kfold.bartisan_fit(bartisan-interop),
 20

lm(), 75
lme4::ranef(), 69
log_lik.bartisan_fit
 (bartisan-interop), 20
loglogistic_aft(bartisan-families), 11
lognormal_aft(bartisan-families), 11
loo(), 4, 17, 47, 64, 66, 76
loo.bartisan_fit(bartisan-interop), 20

`loo::kfold_split_random()`, 22
`loo::loo()`, 22, 23, 25
`loo::loo_compare()`, 23
`loo::waic()`, 22, 23, 25

`marginalEffects::avg_comparisons()`, 3, 72
`marginalEffects::comparisons()`, 29
`marginalEffects::plot_predictions()`, 60
`marginalEffects::predictions()`, 3, 29
`MASS::polr()`, 15
`model.frame.bartisan_fit`
 (`bartisan-marginalEffects`), 26
`model_performance.bartisan_fit`
 (`bartisan-interop`), 20
`multinomial (bartisan-families)`, 11
`multinomial()`, 8, 41, 74

`negbin (bartisan-families)`, 11
`nobs.bartisan_fit`
 (`bartisan-marginalEffects`), 26

`ordbeta (bartisan-families)`, 11
`ordinal (bartisan-families)`, 11

`partial_dependence`, 58
`partial_dependence()`, 3, 66
`performance::check_normality()`, 25
`performance::check_outliers()`, 25
`performance::check_predictions()`, 25
`performance::model_performance()`, 9, 23
`performance::r2()`, 23, 25
`ph (bartisan-families)`, 11
`ph()`, 4, 22, 24, 29, 50, 62, 64
`plot.bartisan_effect (estimate_effect)`, 54
`plot.bartisan_error_density`
 (`error_density`), 53
`plot.bartisan_fit (partial_dependence)`, 58
`plot.bartisan_importance`
 (`variable_importance`), 71
`plot.bartisan_partial`
 (`partial_dependence`), 58
`plot.bcf_fit (print.bcf_fit)`, 67
`posterior::as_draws()`, 23
`posterior::summarise_draws()`, 23
`posterior_epred.bartisan_fit`
 (`bartisan-interop`), 20

`posterior_linpred.bartisan_fit`
 (`bartisan-interop`), 20
`posterior_predict()`, 3
`posterior_predict.bartisan_fit`
 (`bartisan-interop`), 20
`pp_check()`, 4, 47
`pp_check.bartisan_fit`
 (`bartisan-interop`), 20
`predict()`, 47
`predict.bartisan_fit`, 61
`predict.bartisan_fit()`, 3, 10, 22, 25, 27–30, 56, 59
`print.bartisan_effect`
 (`estimate_effect`), 54
`print.bartisan_fit`, 65
`print.bartisan_partial`
 (`partial_dependence`), 58
`print.bartisan_prior_summary`
 (`bartisan-interop`), 20
`print.bcf_fit`, 67
`print.bcf_fit()`, 57
`print.summary.bartisan_fit`
 (`print.bartisan_fit`), 65
`prior_summary()`, 3, 69
`prior_summary.bartisan_fit`
 (`bartisan-interop`), 20

`r2()`, 47
`r2.bartisan_fit (bartisan-interop)`, 20
`r2_posterior.bartisan_fit`
 (`bartisan-interop`), 20
`ranef.bartisan_fit`, 68
`residuals()`, 4
`residuals.bartisan_fit`
 (`bartisan-interop`), 20

`rhc`, 70
`rstantools::log_lik()`, 23
`rstantools::posterior_epred()`, 23
`rstantools::posterior_linpred()`, 23
`rstantools::posterior_predict()`, 9, 23
`rstantools::prior_summary()`, 23

`set.seed()`, 22
`set_coef.bartisan_fit`
 (`bartisan-marginalEffects`), 26
`sigma.bartisan_fit (bartisan-interop)`, 20
`simulate()`, 47

`simulate.bartisan_fit`
 (`bartisan-interop`), 20
`stats::AIC()`, 25
`stats::BIC()`, 25
`stats::family`, 5, 11, 19, 40, 48
`stats::fitted()`, 23
`stats::Gamma()`, 11, 14
`stats::glm()`, 3, 5, 11, 28, 63
`stats::inverse.gaussian`, 5, 40
`stats::make.link()`, 14
`stats::model.frame()`, 27, 55
`stats::na.omit`, 6
`stats::na.pass`, 6
`stats::residuals()`, 23
`stats::sigma()`, 23
`stats::simulate()`, 22–25
`stats::weights()`, 23
`summary.bartisan_fit`
 (`print.bartisan_fit`), 65
`summary.bartisan_fit()`, 71, 73
`survival::Surv()`, 5, 7, 16, 45, 47

`terms.bartisan_fit`
 (`bartisan-marginaleffects`), 26
`tweedie` (`bartisan-families`), 11

`update.formula()`, 75

`variable_importance`, 71
`variable_importance()`, 3, 36, 44, 51, 60, 66
`vc`, 74
`vc()`, 3, 7, 37, 41–44, 67

`waic()`, 47
`waic.bartisan_fit` (`bartisan-interop`), 20
`weibull_aft` (`bartisan-families`), 11
`WeightIt::predict.ordinal_weightit()`,
 63
`WeightIt::weightit()`, 57
`weights.bartisan_fit`
 (`bartisan-interop`), 20

`zi_negbin` (`bartisan-families`), 11
`zi_poisson` (`bartisan-families`), 11
`zi_poisson()`, 8