Package: stochtree (via r-universe)

October 28, 2026

Title Stochastic Tree Ensembles (XBART and BART) for Supervised Learning and Causal Inference

Version 0.4.5

Copyright Copyright details for stochtree's C++ dependencies, which are vendored along with the core stochtree source code, are detailed in inst/COPYRIGHTS

Description Flexible stochastic tree ensemble software. Robust implementations of Bayesian Additive Regression Trees (BART) (Chipman, George, McCulloch (2010) <doi:10.1214/09-AOAS285>) for supervised learning and Bayesian Causal Forests (BCF) (Hahn, Murray, Carvalho (2020) <doi:10.1214/19-BA1195>) for causal inference. Enables model serialization and parallel sampling and provides a low-level interface for custom stochastic forest samplers. Includes the grow-from-root algorithm for accelerated forest sampling (He and Hahn (2021) <doi:10.1080/01621459.2021.1942012>), a log-linear leaf model for forest-based heteroskedasticity (Murray (2020) <doi:10.1080/01621459.2020.1813587>), and the cloglog BART model of Alam and Linero (2025) <doi:10.48550/arXiv.2502.00606> for ordinal outcomes.

License MIT + file LICENSE

Encoding UTF-8

RoxygenNote 7.3.3

LinkingTo cpp11, BH

Suggests testthat (>= 3.0.0),

SystemRequirements C++17

Imports R6, stats

URL https://stochtree.ai/, https://github.com/StochasticTree/stochtree

BugReports https://github.com/StochasticTree/stochtree/issues

Config/testthat/edition 3

NeedsCompilation yes

Author Drew Herren [aut, cre] (ORCID:
<https://orcid.org/0000-0003-4109-6611>), Richard Hahn [aut],
Jared Murray [aut], Carlos Carvalho [aut], Jingyu He [aut],
Pedro Lima [ctb], Entejar Alam [ctb], stochtree contributors
[cph], Eigen contributors [cph] (C++ source uses the Eigen
library for matrix operations, see inst/COPYRIGHTS), xgboost
contributors [cph] (C++ tree code and related operations
include or are inspired by code from the xgboost library, see
inst/COPYRIGHTS), treelite contributors [cph] (C++ tree code
and related operations include or are inspired by code from the
treelite library, see inst/COPYRIGHTS), Microsoft Corporation
[cph] (C++ I/O and various project structure code include or
are inspired by code from the LightGBM library, which is a
copyright of Microsoft, see inst/COPYRIGHTS), Niels Lohmann
[cph] (C++ source uses the JSON for Modern C++ library for JSON
operations, see inst/COPYRIGHTS), Daniel Lemire [cph] (C++
source uses the fast_double_parser library internally, see
inst/COPYRIGHTS), Victor Zverovich [cph] (C++ source uses the
fmt library internally, see inst/COPYRIGHTS)

Maintainer Drew Herren <drewherrenopensource@gmail.com>

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

Date/Publication 2026-06-30 16:40:02 UTC

RemoteUrl https://github.com/cran/stochtree

RemoteRef HEAD

RemoteSha ea813ce2c84a84f034184b999aff800f24cd512a

Contents

stochtree-package 4
bart 5
BARTSerialization 11
bcf 13
BCFSerialization 21
calibrateInverseGammaErrorVariance 23
computeBARTPosteriorInterval 25
computeBCFPosteriorInterval 26
computeContrastBARTModel 28
computeContrastBCFModel 30
CppJson 33
CppRNG 40
createCppRNG 40
createForest 41
createForestDataset 42
createForestModel 42
createForestModelConfig 43
createForestSamples 45

createGlobalModelConfig . . . 46
createOutcome . . . 47
createRandomEffectSamples . . . 47
createRandomEffectsDataset . . . 48
createRandomEffectsModel . . . 49
createRandomEffectsTracker . . . 50
DataPreprocessing . . . 50
extractParameter . . . 52
extractParameter.bartmodel . . . 53
extractParameter.bcfmodel . . . 54
Forest . . . 56
ForestDataset . . . 61
ForestKernelComputation . . . 65
ForestModel . . . 67
ForestModelConfig . . . 71
ForestSamples . . . 78
ForestSamplesSerialization . . . 92
ForestStateManagement . . . 93
getRandomEffectSamples . . . 95
getRandomEffectSamples.bartmodel . . . 96
getRandomEffectSamples.bcfmodel . . . 97
GlobalModelConfig . . . 99
JSONCreation . . . 100
loadScalarJson . . . 101
loadVectorJson . . . 102
Outcome . . . 102
OutcomeModel . . . 104
plot.bartmodel . . . 105
plot.bcfmodel . . . 105
predict.bartmodel . . . 106
predict.bcfmodel . . . 107
print.bartmodel . . . 110
print.bcfmodel . . . 110
print.ForestSamples . . . 111
print.RandomEffectSamples . . . 111
RandomEffectSamples . . . 112
RandomEffectSamplesSerialization . . . 116
RandomEffectsDataset . . . 117
RandomEffectsModel . . . 120
RandomEffectStateManagement . . . 123
RandomEffectsTracker . . . 125
sample_without_replacement . . . 126
sampleBARTPosteriorPredictive . . . 127
sampleBCFPosteriorPredictive . . . 128
sampleGlobalErrorVarianceOneIteration . . . 129
sampleLeafVarianceOneIteration . . . 130
summary.bartmodel . . . 131
summary.bcfmodel . . . 131

Index **132**

`stochtree-package`	*stochtree: Stochastic Tree Ensembles (XBART and BART) for Supervised Learning and Causal Inference*

Description

Flexible stochastic tree ensemble software. Robust implementations of Bayesian Additive Regression Trees (BART) (Chipman, George, McCulloch (2010) doi:10.1214/09AOAS285) for supervised learning and Bayesian Causal Forests (BCF) (Hahn, Murray, Carvalho (2020) doi:10.1214/19BA1195) for causal inference. Enables model serialization and parallel sampling and provides a low-level interface for custom stochastic forest samplers. Includes the grow-from-root algorithm for accelerated forest sampling (He and Hahn (2021) doi:10.1080/01621459.2021.1942012), a log-linear leaf model for forest-based heteroskedasticity (Murray (2020) doi:10.1080/01621459.2020.1813587), and the cloglog BART model of Alam and Linero (2025) doi:10.48550/arXiv.2502.00606 for ordinal outcomes.

Author(s)

Maintainer: Drew Herren <drewherrenopensource@gmail.com> (ORCID)

Authors:

- Richard Hahn
- Jared Murray
- Carlos Carvalho
- Jingyu He

Other contributors:

- Pedro Lima [contributor]
- Entejar Alam [contributor]
- stochtree contributors [copyright holder]
- Eigen contributors (C++ source uses the Eigen library for matrix operations, see inst/COPYRIGHTS) [copyright holder]
- xgboost contributors (C++ tree code and related operations include or are inspired by code from the xgboost library, see inst/COPYRIGHTS) [copyright holder]
- treelite contributors (C++ tree code and related operations include or are inspired by code from the treelite library, see inst/COPYRIGHTS) [copyright holder]
- Microsoft Corporation (C++ I/O and various project structure code include or are inspired by code from the LightGBM library, which is a copyright of Microsoft, see inst/COPYRIGHTS) [copyright holder]
- Niels Lohmann (C++ source uses the JSON for Modern C++ library for JSON operations, see inst/COPYRIGHTS) [copyright holder]
- Daniel Lemire (C++ source uses the fast_double_parser library internally, see inst/COPYRIGHTS) [copyright holder]
- Victor Zverovich (C++ source uses the fmt library internally, see inst/COPYRIGHTS) [copyright holder]

See Also

Useful links:

- https://stochtree.ai/
- https://github.com/StochasticTree/stochtree
- Report bugs at https://github.com/StochasticTree/stochtree/issues

---

`bart` *Run BART for Supervised Learning*

Description

Run the BART algorithm for supervised learning.

Usage

```
bart(
  X_train,
  y_train,
  leaf_basis_train = NULL,
  rfx_group_ids_train = NULL,
  rfx_basis_train = NULL,
  X_test = NULL,
  leaf_basis_test = NULL,
  rfx_group_ids_test = NULL,
  rfx_basis_test = NULL,
  observation_weights = NULL,
  num_gfr = 5,
  num_burnin = 0,
  num_mcmc = 100,
  previous_model_json = NULL,
  previous_model_warmstart_sample_num = NULL,
  general_params = list(),
  mean_forest_params = list(),
  variance_forest_params = list(),
  random_effects_params = list()
)
```

Arguments

`X_train`	Covariates used to split trees in the ensemble. May be provided either as a dataframe or a matrix. Matrix covariates will be assumed to be all numeric. Covariates passed as a dataframe will be preprocessed based on the variable types (e.g. categorical columns stored as unordered factors will be one-hot encoded, categorical columns stored as ordered factors will passed as integers to the core algorithm, along with the metadata that the column is ordered categorical).

y_train Outcome to be modeled by the ensemble.

leaf_basis_train
(Optional) Bases used to define a regression model y ~ W in each leaf of each regression tree. By default, BART assumes constant leaf node parameters, implicitly regressing on a constant basis of ones (i.e. y ~ 1).

rfx_group_ids_train
(Optional) Group labels used for an additive random effects model.

rfx_basis_train
(Optional) Basis for "random-slope" regression in an additive random effects model. If rfx_group_ids_train is provided with a regression basis, an intercept-only random effects model will be estimated.

X_test (Optional) Test set of covariates used to define "out of sample" evaluation data. May be provided either as a dataframe or a matrix, but the format of X_test must be consistent with that of X_train.

leaf_basis_test
(Optional) Test set of bases used to define "out of sample" evaluation data. While a test set is optional, the structure of any provided test set must match that of the training set (i.e. if both X_train and leaf_basis_train are provided, then a test set must consist of X_test and leaf_basis_test with the same number of columns).

rfx_group_ids_test
(Optional) Test set group labels used for an additive random effects model. We do not currently support (but plan to in the near future), test set evaluation for group labels that were not in the training set.

rfx_basis_test (Optional) Test set basis for "random-slope" regression in additive random effects model.

observation_weights
(Optional) Numeric vector of observation weights of length nrow(X_train). Weights are applied as y_i | - ~ N(mu(X_i), sigma^2 / w_i), so larger weights increase an observation's influence on the fit. All weights must be non-negative. Default: NULL (all observations equally weighted). Compatible with Gaussian (continuous/identity) and probit outcome models; not compatible with cloglog link functions. Note: these are referred to internally in the C++ layer as "variance weights" (var_weights), since they scale the residual variance.

num_gfr Number of "warm-start" iterations run using the grow-from-root algorithm (He and Hahn, 2021). Default: 5.

num_burnin Number of "burn-in" iterations of the MCMC sampler. Default: 0.

num_mcmc Number of "retained" iterations of the MCMC sampler. Default: 100.

previous_model_json
(Optional) JSON string containing a previous BART model. This can be used to "continue" a sampler interactively after inspecting the samples or to run parallel chains "warm-started" from existing forest samples. Default: NULL.

previous_model_warmstart_sample_num
(Optional) Sample number from previous_model_json that will be used to warmstart this BART sampler. One-indexed (so that the first sample is used

for warm-start by setting previous_model_warmstart_sample_num = 1). Default: NULL. If num_chains in the general_params list is > 1, then each successive chain will be initialized from a different sample, counting backwards from previous_model_warmstart_sample_num. That is, if previous_model_warmstart_sample_num = 10 and num_chains = 4, then chain 1 will be initialized from sample 10, chain 2 from sample 9, chain 3 from sample 8, and chain 4 from sample 7. If previous_model_json is provided but previous_model_warmstart_sample_num is NULL, the last sample in the previous model will be used to initialize the first chain, counting backwards as noted before. If more chains are requested than there are samples in previous_model_json, a warning will be raised and only the last sample will be used.

general_params (Optional) A list of general (non-forest-specific) model parameters, each of which has a default value processed internally, so this argument list is optional.

- cutpoint_grid_size Maximum size of the "grid" of potential cutpoints to consider in the GFR algorithm. Default: 100.
- standardize Whether or not to standardize the outcome (and store the offset / scale in the model object). Default: TRUE.
- sample_sigma2_global Whether or not to update the sigma^2 global error variance parameter based on IG(sigma2_global_shape, sigma2_global_scale). Default: TRUE.
- sigma2_global_init Starting value of global error variance parameter. Calibrated internally as 1.0*var(y_train), where y_train is the possibly standardized outcome, if not set.
- sigma2_global_shape Shape parameter in the IG(sigma2_global_shape, sigma2_global_scale) global error variance model. Default: 0.
- sigma2_global_scale Scale parameter in the IG(sigma2_global_shape, sigma2_global_scale) global error variance model. Default: 0.
- variable_weights Numeric weights reflecting the relative probability of splitting on each variable. Does not need to sum to 1 but cannot be negative. Defaults to rep(1/ncol(X_train), ncol(X_train)) if not set here. Note that if the propensity score is included as a covariate in either forest, its weight will default to 1/ncol(X_train).
- random_seed Integer parameterizing the C++ random number generator. If not specified, the C++ random number generator is seeded according to std::random_device.
- keep_burnin Whether or not "burnin" samples should be included in the stored samples of forests and other parameters. Default FALSE. Ignored if num_mcmc = 0.
- keep_gfr Whether or not "grow-from-root" samples should be included in the stored samples of forests and other parameters. Default FALSE. Ignored if num_mcmc = 0.
- keep_every How many iterations of the burned-in MCMC sampler should be run before forests and parameters are retained. Default 1. Setting keep_every <- k for some k > 1 will "thin" the MCMC samples by retaining every k-th sample, rather than simply every sample. This can reduce the autocorrelation of the MCMC samples.

- num_chains How many independent MCMC chains should be sampled. If num_mcmc = 0, this is ignored. If num_gfr = 0, then each chain is run from root for num_mcmc * keep_every + num_burnin iterations, with num_mcmc samples retained. If num_gfr > 0, each MCMC chain will be initialized from a separate GFR ensemble, with the requirement that num_gfr >= num_chains. Default: 1. Note that if num_chains > 1, the returned model object will contain samples from all chains, stored consecutively. That is, if there are 4 chains with 100 samples each, the first 100 samples will be from chain 1, the next 100 samples will be from chain 2, etc... For more detail on working with multi-chain BART models, see the multi chain vignette.
- verbose Whether or not to print progress during the sampling loops. Default: FALSE.
- outcome_model A structured OutcomeModel object that specifies the outcome type and desired link function. This argument pre-empts the legacy (deprecated) probit_outcome_model option. Default: OutcomeModel(outcome='continuous', link='identity').
- probit_outcome_model Deprecated in favor of outcome_model. Whether or not the outcome should be modeled as explicitly binary via a probit link. If TRUE, y must only contain the values 0 and 1. Default: FALSE.
- num_threads Number of threads to use in the GFR and MCMC algorithms, as well as prediction. If OpenMP is not available on a user's setup, this will default to 1, otherwise to the maximum number of available threads.

mean_forest_params

(Optional) A list of mean forest model parameters, each of which has a default value processed internally, so this argument list is optional.

- num_trees Number of trees in the ensemble for the conditional mean model. Default: 200. If num_trees = 0, the conditional mean will not be modeled using a forest, and the function will only proceed if num_trees > 0 for the variance forest.
- alpha Prior probability of splitting for a tree of depth 0 in the mean model. Tree split prior combines alpha and beta via alpha*(1+node_depth)^-beta. Default: 0.95.
- beta Exponent that decreases split probabilities for nodes of depth > 0 in the mean model. Tree split prior combines alpha and beta via alpha*(1+node_depth)^-beta. Default: 2.
- min_samples_leaf Minimum allowable size of a leaf, in terms of training samples, in the mean model. Default: 5.
- max_depth Maximum depth of any tree in the ensemble in the mean model. Default: 10. Can be overridden with -1 which does not enforce any depth limits on trees.
- sample_sigma2_leaf Whether or not to update the leaf scale variance parameter based on IG(sigma2_leaf_shape, sigma2_leaf_scale). Cannot (currently) be set to true if ncol(leaf_basis_train)>1. Default: FALSE.
- sigma2_leaf_init Starting value of leaf node scale parameter. Calibrated internally as 1/num_trees if not set here.

- sigma2_leaf_shape Shape parameter in the IG(sigma2_leaf_shape, sigma2_leaf_scale) leaf node parameter variance model. Default: 3.
- sigma2_leaf_scale Scale parameter in the IG(sigma2_leaf_shape, sigma2_leaf_scale) leaf node parameter variance model. Calibrated internally as 0.5/num_trees if not set here.
- keep_vars Vector of variable names or column indices denoting variables that should be included in the forest. Default: NULL.
- drop_vars Vector of variable names or column indices denoting variables that should be excluded from the forest. Default: NULL. If both drop_vars and keep_vars are set, drop_vars will be ignored.
- num_features_subsample How many features to subsample when growing each tree for the GFR algorithm. Defaults to the number of features in the training dataset.

variance_forest_params

(Optional) A list of variance forest model parameters, each of which has a default value processed internally, so this argument list is optional.

- num_trees Number of trees in the ensemble for the conditional variance model. Default: 0. Variance is only modeled using a tree / forest if num_trees > 0.
- alpha Prior probability of splitting for a tree of depth 0 in the variance model. Tree split prior combines alpha and beta via alpha*(1+node_depth)^-beta. Default: 0.95.
- beta Exponent that decreases split probabilities for nodes of depth > 0 in the variance model. Tree split prior combines alpha and beta via alpha*(1+node_depth)^-beta. Default: 2.
- min_samples_leaf Minimum allowable size of a leaf, in terms of training samples, in the variance model. Default: 5.
- max_depth Maximum depth of any tree in the ensemble in the variance model. Default: 10. Can be overridden with -1 which does not enforce any depth limits on trees.
- leaf_prior_calibration_param Hyperparameter used to calibrate the IG(var_forest_prior_shape, var_forest_prior_scale) conditional error variance model. If var_forest_prior_shape and var_forest_prior_scale are not set below, this calibration parameter is used to set these values to num_trees / leaf_prior_calibration_param^2 + 0.5 and num_trees / leaf_prior_calibration_param^2, respectively. Default: 1.5.
- var_forest_leaf_init Starting value of root forest prediction in conditional (heteroskedastic) error variance model. Calibrated internally as log(0.6*var(y_train))/num_trees, where y_train is the possibly standardized outcome, if not set.
- var_forest_prior_shape Shape parameter in the IG(var_forest_prior_shape, var_forest_prior_scale) conditional error variance model (which is only sampled if num_trees > 0). Calibrated internally as num_trees / leaf_prior_calibration_param + 0.5 if not set.
- var_forest_prior_scale Scale parameter in the IG(var_forest_prior_shape, var_forest_prior_scale) conditional error variance model (which is only

sampled if num_trees > 0). Calibrated internally as num_trees / leaf_prior_calibration_param if not set.

- keep_vars Vector of variable names or column indices denoting variables that should be included in the forest. Default: NULL.
- drop_vars Vector of variable names or column indices denoting variables that should be excluded from the forest. Default: NULL. If both drop_vars and keep_vars are set, drop_vars will be ignored.
- num_features_subsample How many features to subsample when growing each tree for the GFR algorithm. Defaults to the number of features in the training dataset.

random_effects_params

(Optional) A list of random effects model parameters, each of which has a default value processed internally, so this argument list is optional.

- model_spec Specification of the random effects model. Options are "custom" and "intercept_only". If "custom" is specified, then a user-provided basis must be passed through rfx_basis_train. If "intercept_only" is specified, a random effects basis of all ones will be dispatched internally at sampling and prediction time. If "intercept_plus_treatment" is specified, a random effects basis that combines an "intercept" basis of all ones with the treatment variable (Z_train) will be dispatched internally at sampling and prediction time. Default: "custom". If "intercept_only" is specified, rfx_basis_train and rfx_basis_test (if provided) will be ignored.
- working_parameter_prior_mean Prior mean for the random effects "working parameter". Default: NULL. Must be a vector whose dimension matches the number of random effects bases, or a scalar value that will be expanded to a vector.
- group_parameters_prior_mean Prior mean for the random effects "group parameters." Default: NULL. Must be a vector whose dimension matches the number of random effects bases, or a scalar value that will be expanded to a vector.
- working_parameter_prior_cov Prior covariance matrix for the random effects "working parameter." Default: NULL. Must be a square matrix whose dimension matches the number of random effects bases, or a scalar value that will be expanded to a diagonal matrix.
- group_parameter_prior_cov Prior covariance matrix for the random effects "group parameters." Default: NULL. Must be a square matrix whose dimension matches the number of random effects bases, or a scalar value that will be expanded to a diagonal matrix.
- variance_prior_shape Shape parameter for the inverse gamma prior on the variance of the random effects "group parameter." Default: 1.
- variance_prior_scale Scale parameter for the inverse gamma prior on the variance of the random effects "group parameter." Default: 1.

Value

List of sampling outputs and a wrapper around the sampled forests (which can be used for in-memory prediction on new data, or serialized to JSON on disk).

Examples

```
n <- 100
p <- 5
X <- matrix(runif(n*p), ncol = p)
f_XW <- (
    ((0 <= X[,1]) & (0.25 > X[,1])) * (-7.5) +
    ((0.25 <= X[,1]) & (0.5 > X[,1])) * (-2.5) +
    ((0.5 <= X[,1]) & (0.75 > X[,1])) * (2.5) +
    ((0.75 <= X[,1]) & (1 > X[,1])) * (7.5)
)
noise_sd <- 1
y <- f_XW + rnorm(n, 0, noise_sd)
test_set_pct <- 0.2
n_test <- round(test_set_pct*n)
n_train <- n - n_test
test_inds <- sort(sample(1:n, n_test, replace = FALSE))
train_inds <- (1:n)[!((1:n) %in% test_inds)]
X_test <- X[test_inds,]
X_train <- X[train_inds,]
y_test <- y[test_inds]
y_train <- y[train_inds]

bart_model <- bart(X_train = X_train, y_train = y_train, X_test = X_test,
                   num_gfr = 10, num_burnin = 0, num_mcmc = 10)
```

`BARTSerialization` *BART Serialization Routines*

Description

BART models contains external pointers to C++ objects, which means they cannot be correctly serialized to `.Rds` from an R session in their default state. This group of serialization functions allow us to convert between C++ data structures and a persistent JSON representation. The `CppJson` class wraps a performant C++ JSON API, and the functions `saveBARTModelToJson` and `createBARTModelFromJson` save to and load from this format. This representation, of course, also relies on external C++ pointers, so in order to save and reload BART models across sessions, we provide two other interfaces.

`saveBARTModelToJsonString` converts a BART model to an in-memory string containing the model's JSON representation and `createBARTModelFromJsonString` converts this representation back to a BART model object.

`saveBARTModelToJsonFile` and `createBARTModelFromJsonFile` save or reload a BART model directly to / from a `.json` file.

Finally, for cases in which multiple BART models have been sampled (for instance, multiple processes run via `doParallel`), we offer `createBARTModelFromCombinedJson` and `createBARTModelFromCombinedJsonStri` for loading a new combined BART model from a list of BART JSON objects / strings.

Usage

```
saveBARTModelToJson(object)

saveBARTModelToJsonFile(object, filename)

saveBARTModelToJsonString(object)

createBARTModelFromJson(json_object)

createBARTModelFromJsonFile(json_filename)

createBARTModelFromJsonString(json_string)

createBARTModelFromCombinedJson(json_object_list)

createBARTModelFromCombinedJsonString(json_string_list)
```

Arguments

`object`	Object of type `bartmodel` containing draws of a BART model and associated sampling outputs.
`filename`	String of filepath, must end in ".json"
`json_object`	Object of type `CppJson` containing Json representation of a BART model
`json_filename`	String of filepath, must end in ".json"
`json_string`	JSON string dump
`json_object_list`	List of objects of type `CppJson` containing Json representation of a BART model
`json_string_list`	List of JSON strings which can be parsed to objects of type `CppJson` containing Json representation of a BART model

Value

`saveBARTModelToJson` return an object of type `CppJson`. `saveBARTModelToJsonString` returns a string dump of the BART model's JSON representation. `saveBARTModelToJsonFile` returns nothing, but writes to the provided filename.

`createBARTModelFromJson`, `createBARTModelFromJsonFile`, `createBARTModelFromJsonString`, `createBARTModelFromCombinedJson`, and `createBARTModelFromCombinedJsonString` all return objects of type `bartmodel`.

Examples

```
# Generate data
n <- 100
p <- 5
X <- matrix(runif(n*p), ncol = p)
y <- X[,1] + rnorm(n, 0, 1)
```

```
# Sample BART model
bart_model <- bart(X_train = X, y_train = y, num_gfr = 0,
                   num_burnin = 0, num_mcmc = 10)

# Save to in-memory JSON
bart_json <- saveBARTModelToJson(bart_model)
# Save to JSON string
bart_json_string <- saveBARTModelToJsonString(bart_model)
# Save to JSON file
tmpjson <- tempfile(fileext = ".json")
saveBARTModelToJsonFile(bart_model, file.path(tmpjson))

# Reload BART model from in-memory JSON object
bart_model_roundtrip <- createBARTModelFromJson(bart_json)
# Reload BART model from JSON string
bart_model_roundtrip <- createBARTModelFromJsonString(bart_json_string)
# Reload BART model from JSON file
bart_model_roundtrip <- createBARTModelFromJsonFile(file.path(tmpjson))
unlink(tmpjson)
# Reload BART model from list of JSON objects
bart_model_roundtrip <- createBARTModelFromCombinedJson(list(bart_json))
# Reload BART model from list of JSON strings
bart_model_roundtrip <- createBARTModelFromCombinedJsonString(list(bart_json_string))
```

---

bcf *Run BCF for Causal Effect Estimation*

Description

Run the Bayesian Causal Forest (BCF) algorithm for regularized causal effect estimation.

Usage

```
bcf(
  X_train,
  Z_train,
  y_train,
  propensity_train = NULL,
  rfx_group_ids_train = NULL,
  rfx_basis_train = NULL,
  X_test = NULL,
  Z_test = NULL,
  propensity_test = NULL,
  rfx_group_ids_test = NULL,
  rfx_basis_test = NULL,
  observation_weights = NULL,
  num_gfr = 5,
```

```
  num_burnin = 0,
  num_mcmc = 100,
  previous_model_json = NULL,
  previous_model_warmstart_sample_num = NULL,
  general_params = list(),
  prognostic_forest_params = list(),
  treatment_effect_forest_params = list(),
  variance_forest_params = list(),
  random_effects_params = list()
)
```

Arguments

X_train	Covariates used to split trees in the ensemble. May be provided either as a dataframe or a matrix. Matrix covariates will be assumed to be all numeric. Covariates passed as a dataframe will be preprocessed based on the variable types (e.g. categorical columns stored as unordered factors will be one-hot encoded, categorical columns stored as ordered factors will passed as integers to the core algorithm, along with the metadata that the column is ordered categorical).
Z_train	Vector of (continuous or binary) treatment assignments.
y_train	Outcome to be modeled by the ensemble.
propensity_train	(Optional) Vector of propensity scores. If not provided, this will be estimated from the data. If NULL and previous_model_json is provided with an internally estimated propensity model, that model's propensity estimates are re-used rather than re-fitted.
rfx_group_ids_train	(Optional) Group labels used for an additive random effects model.
rfx_basis_train	(Optional) Basis for "random-slope" regression in an additive random effects model. If rfx_group_ids_train is provided with a regression basis, an intercept-only random effects model will be estimated.
X_test	(Optional) Test set of covariates used to define "out of sample" evaluation data. May be provided either as a dataframe or a matrix, but the format of X_test must be consistent with that of X_train.
Z_test	(Optional) Test set of (continuous or binary) treatment assignments.
propensity_test	(Optional) Vector of propensity scores. If not provided, this will be estimated from the data.
rfx_group_ids_test	(Optional) Test set group labels used for an additive random effects model. We do not currently support (but plan to in the near future), test set evaluation for group labels that were not in the training set.
rfx_basis_test	(Optional) Test set basis for "random-slope" regression in additive random effects model.

observation_weights
: (Optional) Numeric vector of observation weights of length nrow(X_train). Weights are applied as y_i | - ~ N(mu(X_i), sigma^2 / w_i), so larger weights increase an observation's influence on the fit. All weights must be non-negative. Default: NULL (all observations equally weighted). Applied to both the prognostic and treatment effect forests. Compatible with Gaussian (continuous/identity) and probit outcome models; not compatible with cloglog link functions.

num_gfr
: Number of "warm-start" iterations run using the grow-from-root algorithm (He and Hahn, 2021). Default: 5.

num_burnin
: Number of "burn-in" iterations of the MCMC sampler. Default: 0.

num_mcmc
: Number of "retained" iterations of the MCMC sampler. Default: 100.

previous_model_json
: (Optional) JSON string containing a previous BCF model. This can be used to "continue" a sampler interactively after inspecting the samples or to run parallel chains "warm-started" from existing forest samples. If the previous model used an internally estimated propensity score (i.e. propensity_train was not supplied to that run), the fitted propensity model is carried forward and re-used rather than being re-estimated. This ensures that multi-chain warm-starts remain consistent with the propensity scores used in the initial run. Default: NULL.

previous_model_warmstart_sample_num
: (Optional) Sample number from previous_model_json that will be used to warmstart this BCF sampler. One-indexed (so that the first sample is used for warm-start by setting previous_model_warmstart_sample_num = 1). Default: NULL. If num_chains in the general_params list is > 1, then each successive chain will be initialized from a different sample, counting backwards from previous_model_warmstart_sample_num. That is, if previous_model_warmstart_sample_num = 10 and num_chains = 4, then chain 1 will be initialized from sample 10, chain 2 from sample 9, chain 3 from sample 8, and chain 4 from sample 7. If previous_model_json is provided but previous_model_warmstart_sample_num is NULL, the last sample in the previous model will be used to initialize the first chain, counting backwards as noted before. If more chains are requested than there are samples in previous_model_json, a warning will be raised and only the last sample will be used.

general_params
: (Optional) A list of general (non-forest-specific) model parameters, each of which has a default value processed internally, so this argument list is optional.

  - cutpoint_grid_size Maximum size of the "grid" of potential cutpoints to consider in the GFR algorithm. Default: 100.
  - standardize Whether or not to standardize the outcome (and store the offset / scale in the model object). Default: TRUE.
  - sample_sigma2_global Whether or not to update the sigma^2 global error variance parameter based on IG(sigma2_global_shape, sigma2_global_scale). Default: TRUE.
  - sigma2_global_init Starting value of global error variance parameter. Calibrated internally as 1.0*var((y_train-mean(y_train))/sd(y_train)) if not set.
  - sigma2_global_shape Shape parameter in the IG(sigma2_global_shape, sigma2_global_scale) global error variance model. Default: 0.

- sigma2_global_scale Scale parameter in the IG(sigma2_global_shape, sigma2_global_scale) global error variance model. Default: 0.
- variable_weights Numeric weights reflecting the relative probability of splitting on each variable. Does not need to sum to 1 but cannot be negative. Defaults to rep(1/ncol(X_train), ncol(X_train)) if not set here. Note that if the propensity score is included as a covariate in either forest, its weight will default to 1/ncol(X_train). A workaround if you wish to provide a custom weight for the propensity score is to include it as a column in X_train and then set propensity_covariate to 'none' adjust keep_vars accordingly for the prognostic or treatment_effect forests.
- propensity_covariate Whether to include the propensity score as a covariate in either or both of the forests. Enter "none" for neither, "prognostic" for the prognostic forest, "treatment_effect" for the treatment forest, and "both" for both forests. If this is not "none" and a propensity score is not provided, it will be estimated from (X_train, Z_train) using stochtree::bart(). Default: "mu".
- adaptive_coding Whether or not to use an "adaptive coding" scheme in which a binary treatment variable is not coded manually as (0,1) or (-1,1) but learned via parameters b_0 and b_1 that attach to the outcome model [b_0 (1-Z) + b_1 Z] tau(X). This is ignored when Z is not binary. Default: FALSE.
- control_coding_init Initial value of the "control" group coding parameter. This is ignored when Z is not binary. Default: -0.5.
- treated_coding_init Initial value of the "treatment" group coding parameter. This is ignored when Z is not binary. Default: 0.5.
- rfx_prior_var Prior on the (diagonals of the) covariance of the additive group-level random regression coefficients. Must be a vector of length ncol(rfx_basis_train). Default: rep(1, ncol(rfx_basis_train))
- random_seed Integer parameterizing the C++ random number generator. If not specified, the C++ random number generator is seeded according to std::random_device.
- keep_burnin Whether or not "burnin" samples should be included in the stored samples of forests and other parameters. Default FALSE. Ignored if num_mcmc = 0.
- keep_gfr Whether or not "grow-from-root" samples should be included in the stored samples of forests and other parameters. Default FALSE. Ignored if num_mcmc = 0.
- keep_every How many iterations of the burned-in MCMC sampler should be run before forests and parameters are retained. Default 1. Setting keep_every <- k for some k > 1 will "thin" the MCMC samples by retaining every k-th sample, rather than simply every sample. This can reduce the autocorrelation of the MCMC samples.
- num_chains How many independent MCMC chains should be sampled. If num_mcmc = 0, this is ignored. If num_gfr = 0, then each chain is run from root for num_mcmc * keep_every + num_burnin iterations, with num_mcmc samples retained. If num_gfr > 0, each MCMC chain will be initialized from a separate GFR ensemble, with the requirement that num_gfr >= num_chains.

Default: 1. Note that if num_chains > 1, the returned model object will contain samples from all chains, stored consecutively. That is, if there are 4 chains with 100 samples each, the first 100 samples will be from chain 1, the next 100 samples will be from chain 2, etc... For more detail on working with multi-chain BCF models, see the multi chain vignette.

- verbose Whether or not to print progress during the sampling loops. Default: FALSE.
- outcome_model A structured OutcomeModel object that specifies the outcome type and desired link function. This argument pre-empts the legacy (deprecated) probit_outcome_model option. Default: OutcomeModel(outcome='continuous', link='identity').
- probit_outcome_model Deprecated in favor of outcome_model. Whether or not the outcome should be modeled as explicitly binary via a probit link. If TRUE, y must only contain the values 0 and 1. Default: FALSE.
- num_threads Number of threads to use in the GFR and MCMC algorithms, as well as prediction. If OpenMP is not available on a user's setup, this will default to 1, otherwise to the maximum number of available threads.

prognostic_forest_params

(Optional) A list of prognostic forest model parameters, each of which has a default value processed internally, so this argument list is optional.

- num_trees Number of trees in the ensemble for the prognostic forest. Default: 250. Must be a positive integer.
- alpha Prior probability of splitting for a tree of depth 0 in the prognostic forest. Tree split prior combines alpha and beta via alpha*(1+node_depth)^-beta. Default: 0.95.
- beta Exponent that decreases split probabilities for nodes of depth > 0 in the prognostic forest. Tree split prior combines alpha and beta via alpha*(1+node_depth)^-beta. Default: 2.
- min_samples_leaf Minimum allowable size of a leaf, in terms of training samples, in the prognostic forest. Default: 5.
- max_depth Maximum depth of any tree in the ensemble in the prognostic forest. Default: 10. Can be overridden with -1 which does not enforce any depth limits on trees.
- variable_weights Numeric weights reflecting the relative probability of splitting on each variable in the prognostic forest. Does not need to sum to 1 but cannot be negative. Defaults to rep(1/ncol(X_train), ncol(X_train)) if not set here.
- sample_sigma2_leaf Whether or not to update the leaf scale variance parameter based on IG(sigma2_leaf_shape, sigma2_leaf_scale).
- sigma2_leaf_init Starting value of leaf node scale parameter. Calibrated internally as 1/num_trees if not set here.
- sigma2_leaf_shape Shape parameter in the IG(sigma2_leaf_shape, sigma2_leaf_scale) leaf node parameter variance model. Default: 3.
- sigma2_leaf_scale Scale parameter in the IG(sigma2_leaf_shape, sigma2_leaf_scale) leaf node parameter variance model. Calibrated internally as 0.5/num_trees if not set here.

- keep_vars Vector of variable names or column indices denoting variables that should be included in the forest. Default: NULL.
- drop_vars Vector of variable names or column indices denoting variables that should be excluded from the forest. Default: NULL. If both drop_vars and keep_vars are set, drop_vars will be ignored.
- num_features_subsample How many features to subsample when growing each tree for the GFR algorithm. Defaults to the number of features in the training dataset.

treatment_effect_forest_params

(Optional) A list of treatment effect forest model parameters, each of which has a default value processed internally, so this argument list is optional.

- num_trees Number of trees in the ensemble for the treatment effect forest. Default: 100. Must be a positive integer.
- alpha Prior probability of splitting for a tree of depth 0 in the treatment effect forest. Tree split prior combines alpha and beta via alpha*(1+node_depth)^-beta. Default: 0.25.
- beta Exponent that decreases split probabilities for nodes of depth > 0 in the treatment effect forest. Tree split prior combines alpha and beta via alpha*(1+node_depth)^-beta. Default: 3.
- min_samples_leaf Minimum allowable size of a leaf, in terms of training samples, in the treatment effect forest. Default: 5.
- max_depth Maximum depth of any tree in the ensemble in the treatment effect forest. Default: 5. Can be overridden with -1 which does not enforce any depth limits on trees.
- variable_weights Numeric weights reflecting the relative probability of splitting on each variable in the treatment effect forest. Does not need to sum to 1 but cannot be negative. Defaults to rep(1/ncol(X_train), ncol(X_train)) if not set here.
- sample_sigma2_leaf Whether or not to update the leaf scale variance parameter based on IG(sigma2_leaf_shape, sigma2_leaf_scale). Cannot (currently) be set to true if ncol(Z_train)>1. Default: FALSE.
- sigma2_leaf_init Starting value of leaf node scale parameter. Calibrated internally as 0.5 * var(y)/num_trees if not set here (0.5 / num_trees if y is continuous and standardize = TRUE in the general_params list).
- sigma2_leaf_shape Shape parameter in the IG(sigma2_leaf_shape, sigma2_leaf_scale) leaf node parameter variance model. Default: 3.
- sigma2_leaf_scale Scale parameter in the IG(sigma2_leaf_shape, sigma2_leaf_scale) leaf node parameter variance model. Calibrated internally as 0.5/num_trees if not set here.
- delta_max Maximum plausible conditional distributional treatment effect (i.e. P(Y(1) = 1 | X) - P(Y(0) = 1 | X)) when the outcome is binary. Only used when the outcome is specified as a probit model in general_params. Must be > 0 and < 1. Default: 0.9. Ignored if sigma2_leaf_init is set directly, as this parameter is used to calibrate sigma2_leaf_init.
- keep_vars Vector of variable names or column indices denoting variables that should be included in the forest. Default: NULL.

- drop_vars Vector of variable names or column indices denoting variables that should be excluded from the forest. Default: NULL. If both drop_vars and keep_vars are set, drop_vars will be ignored.
- num_features_subsample How many features to subsample when growing each tree for the GFR algorithm. Defaults to the number of features in the training dataset.
- sample_intercept Whether to sample a global treatment effect intercept tau_0 so the full CATE is tau_0 + tau(X). Default: TRUE. Compatible with adaptive_coding = TRUE, in which case the recoded treatment basis is used.
- tau_0_prior_var Variance of the normal prior on tau_0 (a scalar applied to each treatment dimension independently). Auto-calibrated to outcome variance when NULL and outcome is continuous. Only used when sample_intercept = TRUE.

variance_forest_params

(Optional) A list of variance forest model parameters, each of which has a default value processed internally, so this argument list is optional.

- num_trees Number of trees in the ensemble for the conditional variance model. Default: 0. Variance is only modeled using a tree / forest if num_trees > 0.
- alpha Prior probability of splitting for a tree of depth 0 in the variance model. Tree split prior combines alpha and beta via alpha*(1+node_depth)^-beta. Default: 0.95.
- beta Exponent that decreases split probabilities for nodes of depth > 0 in the variance model. Tree split prior combines alpha and beta via alpha*(1+node_depth)^-beta. Default: 2.
- min_samples_leaf Minimum allowable size of a leaf, in terms of training samples, in the variance model. Default: 5.
- max_depth Maximum depth of any tree in the ensemble in the variance model. Default: 10. Can be overridden with -1 which does not enforce any depth limits on trees.
- leaf_prior_calibration_param Hyperparameter used to calibrate the IG(var_forest_prior_shape, var_forest_prior_scale) conditional error variance model. If var_forest_prior_shape and var_forest_prior_scale are not set below, this calibration parameter is used to set these values to num_trees / leaf_prior_calibration_param^2 + 0.5 and num_trees / leaf_prior_calibration_param^2, respectively. Default: 1.5.
- variance_forest_init Starting value of root forest prediction in conditional (heteroskedastic) error variance model. Calibrated internally as log(0.6*var((y_train-mean(y_train))/sd(y_train)))/num_trees if not set.
- var_forest_prior_shape Shape parameter in the IG(var_forest_prior_shape, var_forest_prior_scale) conditional error variance model (which is only sampled if num_trees > 0). Calibrated internally as num_trees / 1.5^2 + 0.5 if not set.
- var_forest_prior_scale Scale parameter in the IG(var_forest_prior_shape, var_forest_prior_scale) conditional error variance model (which is only

sampled if num_trees > 0). Calibrated internally as num_trees / 1.5^2 if not set.

- keep_vars Vector of variable names or column indices denoting variables that should be included in the forest. Default: NULL.
- drop_vars Vector of variable names or column indices denoting variables that should be excluded from the forest. Default: NULL. If both drop_vars and keep_vars are set, drop_vars will be ignored.
- num_features_subsample How many features to subsample when growing each tree for the GFR algorithm. Defaults to the number of features in the training dataset.

random_effects_params

(Optional) A list of random effects model parameters, each of which has a default value processed internally, so this argument list is optional.

- model_spec Specification of the random effects model. Options are "custom", "intercept_only", and "intercept_plus_treatment". If "custom" is specified, then a user-provided basis must be passed through rfx_basis_train. If "intercept_only" is specified, a random effects basis of all ones will be dispatched internally at sampling and prediction time. If "intercept_plus_treatment" is specified, a random effects basis that combines an "intercept" basis of all ones with the treatment variable (Z_train) will be dispatched internally at sampling and prediction time. Default: "custom". If either "intercept_only" or "intercept_plus_treatment" is specified, rfx_basis_train and rfx_basis_test (if provided) will be ignored.
- working_parameter_prior_mean Prior mean for the random effects "working parameter". Default: NULL. Must be a vector whose dimension matches the number of random effects bases, or a scalar value that will be expanded to a vector.
- group_parameters_prior_mean Prior mean for the random effects "group parameters." Default: NULL. Must be a vector whose dimension matches the number of random effects bases, or a scalar value that will be expanded to a vector.
- working_parameter_prior_cov Prior covariance matrix for the random effects "working parameter." Default: NULL. Must be a square matrix whose dimension matches the number of random effects bases, or a scalar value that will be expanded to a diagonal matrix.
- group_parameter_prior_cov Prior covariance matrix for the random effects "group parameters." Default: NULL. Must be a square matrix whose dimension matches the number of random effects bases, or a scalar value that will be expanded to a diagonal matrix.
- variance_prior_shape Shape parameter for the inverse gamma prior on the variance of the random effects "group parameter." Default: 1.
- variance_prior_scale Scale parameter for the inverse gamma prior on the variance of the random effects "group parameter." Default: 1.

Value

List of sampling outputs and a wrapper around the sampled forests (which can be used for in-memory prediction on new data, or serialized to JSON on disk).

Examples

```
n <- 500
p <- 5
X <- matrix(runif(n*p), ncol = p)
mu_x <- (
    ((0 <= X[,1]) & (0.25 > X[,1])) * (-7.5) +
    ((0.25 <= X[,1]) & (0.5 > X[,1])) * (-2.5) +
    ((0.5 <= X[,1]) & (0.75 > X[,1])) * (2.5) +
    ((0.75 <= X[,1]) & (1 > X[,1])) * (7.5)
)
pi_x <- (
    ((0 <= X[,1]) & (0.25 > X[,1])) * (0.2) +
    ((0.25 <= X[,1]) & (0.5 > X[,1])) * (0.4) +
    ((0.5 <= X[,1]) & (0.75 > X[,1])) * (0.6) +
    ((0.75 <= X[,1]) & (1 > X[,1])) * (0.8)
)
tau_x <- (
    ((0 <= X[,2]) & (0.25 > X[,2])) * (0.5) +
    ((0.25 <= X[,2]) & (0.5 > X[,2])) * (1.0) +
    ((0.5 <= X[,2]) & (0.75 > X[,2])) * (1.5) +
    ((0.75 <= X[,2]) & (1 > X[,2])) * (2.0)
)
Z <- rbinom(n, 1, pi_x)
noise_sd <- 1
y <- mu_x + tau_x*Z + rnorm(n, 0, noise_sd)
test_set_pct <- 0.2
n_test <- round(test_set_pct*n)
n_train <- n - n_test
test_inds <- sort(sample(1:n, n_test, replace = FALSE))
train_inds <- (1:n)[!((1:n) %in% test_inds)]
X_test <- X[test_inds,]
X_train <- X[train_inds,]
pi_test <- pi_x[test_inds]
pi_train <- pi_x[train_inds]
Z_test <- Z[test_inds]
Z_train <- Z[train_inds]
y_test <- y[test_inds]
y_train <- y[train_inds]
mu_test <- mu_x[test_inds]
mu_train <- mu_x[train_inds]
tau_test <- tau_x[test_inds]
tau_train <- tau_x[train_inds]
bcf_model <- bcf(X_train = X_train, Z_train = Z_train, y_train = y_train,
                 propensity_train = pi_train, X_test = X_test, Z_test = Z_test,
                 propensity_test = pi_test, num_gfr = 10,
                 num_burnin = 0, num_mcmc = 10)
```

BCFSerialization *BCF Serialization Routines*

Description

BCF models contains external pointers to C++ objects, which means they cannot be correctly serialized to `.Rds` from an R session in their default state. These functions allow us to convert between C++ data structures and a persistent JSON representation. The `CppJson` class wraps a performant C++ JSON API, and the functions `saveBCFModelToJson` and `createBCFModelFromJson` save to and load from this format. This representation, of course, also relies on external C++ pointers, so in order to save and reload BCF models across sessions, we provide two other interfaces.

`saveBCFModelToJsonFile` and `createBCFModelFromJsonFile` save or reload a BCF model's JSON representation directly to / from a `.json` file.

`saveBCFModelToJsonString` and `createBCFModelFromJsonString` handle in-memory strings containing JSON data, which can be written to disk or passed between processes.

Finally, for cases in which multiple BCF models have been sampled (for instance, sampled in multiple processes via `doParallel`), we offer `createBCFModelFromCombinedJson` and `createBCFModelFromCombinedJsonStr` for loading a new combined BCF model from a list of BCF JSON objects or strings.

Usage

```
saveBCFModelToJson(object)

saveBCFModelToJsonFile(object, filename)

saveBCFModelToJsonString(object)

createBCFModelFromJson(json_object)

createBCFModelFromJsonFile(json_filename)

createBCFModelFromJsonString(json_string)

createBCFModelFromCombinedJson(json_object_list)

createBCFModelFromCombinedJsonString(json_string_list)
```

Arguments

`object`	Object of type `bcfmodel` containing draws of a Bayesian causal forest model and associated sampling outputs.
`filename`	String of filepath, must end in ".json"
`json_object`	Object of type `CppJson` containing Json representation of a BCF model
`json_filename`	String of filepath, must end in ".json"
`json_string`	JSON string dump
`json_object_list`	List of objects of type `CppJson` containing Json representation of a BCF model
`json_string_list`	List of JSON strings which can be parsed to objects of type `CppJson` containing Json representation of a BCF model

Value

saveBCFModelToJson return an object of type CppJson. saveBCFModelToJsonFile returns nothing, but writes to the provided filename. saveBCFModelToJsonString returns a string dump of the BCF model's JSON representation.

createBCFModelFromJson, createBCFModelFromJsonFile, createBCFModelFromJsonString, createBCFModelFromCombinedJson, and createBCFModelFromCombinedJsonString all return objects of type bcfmodel.

Examples

```
# Generate data
n <- 100
p <- 5
X <- matrix(runif(n*p), ncol = p)
pi_X <- runif(n, 0.3, 0.7)
Z <- rbinom(n, p = pi_X, size = 1)
y <- X[,1] + Z + rnorm(n, 0, 1)

# Sample BCF model
bcf_model <- bcf(X_train = X, Z_train = Z, propensity_train = pi_X, y_train = y,
                 num_gfr = 0, num_burnin = 0, num_mcmc = 10)

# Save to in-memory JSON
bcf_json <- saveBCFModelToJson(bcf_model)
# Save to JSON string
bcf_json_string <- saveBCFModelToJsonString(bcf_model)
# Save to JSON file
tmpjson <- tempfile(fileext = ".json")
saveBCFModelToJsonFile(bcf_model, file.path(tmpjson))

# Reload BCF model from in-memory JSON object
bcf_model_roundtrip <- createBCFModelFromJson(bcf_json)
# Reload BCF model from JSON string
bcf_model_roundtrip <- createBCFModelFromJsonString(bcf_json_string)
# Reload BCF model from JSON file
bcf_model_roundtrip <- createBCFModelFromJsonFile(file.path(tmpjson))
unlink(tmpjson)
# Reload BCF model from list of JSON objects
bcf_model_roundtrip <- createBCFModelFromCombinedJson(list(bcf_json))
# Reload BCF model from list of JSON strings
bcf_model_roundtrip <- createBCFModelFromCombinedJsonString(list(bcf_json_string))
```

---

calibrateInverseGammaErrorVariance

Calibrate Inverse Gamma Prior

Description

Calibrate the scale parameter on an inverse gamma prior for the global error variance as in Chipman et al (2022)

This function is intended for advanced use cases in which users require detailed control of sampling algorithms and data structures. Minimal input validation and error checks are performed – users are responsible for providing the correct inputs. For tutorials on the "proper" usage of the stochtree's advanced workflow, we provide several vignettes at https://stochtree.ai/

Chipman, H., George, E., Hahn, R., McCulloch, R., Pratola, M. and Sparapani, R. (2022). Bayesian Additive Regression Trees, Computational Approaches. In Wiley StatsRef: Statistics Reference Online (eds N. Balakrishnan, T. Colton, B. Everitt, W. Piegorsch, F. Ruggeri and J.L. Teugels). doi:10.1002/9781118445112.stat08288

Usage

```
calibrateInverseGammaErrorVariance(
  y,
  X,
  W = NULL,
  nu = 3,
  quant = 0.9,
  standardize = TRUE
)
```

Arguments

y	Outcome to be modeled using BART, BCF or another nonparametric ensemble method.
X	Covariates to be used to partition trees in an ensemble or series of ensemble.
W	(Optional) Basis used to define a "leaf regression" model for each decision tree. The "classic" BART model assumes a constant leaf parameter, which is equivalent to a "leaf regression" on a basis of all ones, though it is not necessary to pass a vector of ones, here or to the BART function. Default: NULL.
nu	The shape parameter for the global error variance's IG prior. The scale parameter in the Sparapani et al (2021) parameterization is defined as nu*lambda where lambda is the output of this function. Default: 3.
quant	(Optional) Quantile of the inverse gamma prior distribution represented by a linear-regression-based overestimate of sigma^2. Default: 0.9.
standardize	(Optional) Whether or not outcome should be standardized ((y-mean(y))/sd(y)) before calibration of lambda. Default: TRUE.

Value

Value of lambda which determines the scale parameter of the global error variance prior (sigma^2 ~ IG(nu,nu*lambda))

Examples

```
n <- 100
p <- 5
X <- matrix(runif(n*p), ncol = p)
y <- 10*X[,1] - 20*X[,2] + rnorm(n)
nu <- 3
lambda <- calibrateInverseGammaErrorVariance(y, X, nu = nu)
sigma2hat <- mean(resid(lm(y~X))^2)
mean(var(y)/rgamma(100000, nu, rate = nu*lambda) < sigma2hat)
```

`computeBARTPosteriorInterval`

Compute BART Posterior Credible Intervals

Description

Compute posterior credible intervals for specified terms from a fitted BART model. Supports intervals for mean functions, variance functions, random effects, and overall outcome predictions.

Usage

```
computeBARTPosteriorInterval(
  model_object,
  terms,
  level = 0.95,
  scale = "linear",
  X = NULL,
  leaf_basis = NULL,
  rfx_group_ids = NULL,
  rfx_basis = NULL
)
```

Arguments

`model_object`	A fitted BART or BCF model object of class `bartmodel`.
`terms`	A character string specifying the model term(s) for which to compute intervals. Options for BART models are `"mean_forest"`, `"variance_forest"`, `"rfx"`, or `"y_hat"`.
`level`	A numeric value between 0 and 1 specifying the credible interval level (default is 0.95 for a 95% credible interval).
`scale`	(Optional) Scale of mean function predictions. Options are "linear", which returns predictions on the original scale of the mean forest / RFX terms, and "probability". `scale = "probability"` is only valid for models fit with a probit / cloglog links on binary or ordinal outcomes. For binary outcome models, `scale = "probability"` will return an interval over the probability that `y == 1`. For ordinal outcome models, `scale = "probability"` will return intervals over the "survival function" `P(y > k)` for `k = 1, 2, ..., K-1` where `K` is the total number of categories. Default: "linear".

X	A matrix or data frame of covariates at which to compute the intervals. Required if the requested term depends on covariates (e.g., mean forest, variance forest, or overall predictions).
leaf_basis	An optional matrix of basis function evaluations for mean forest models with regression defined in the leaves. Required for "leaf regression" models.
rfx_group_ids	An optional vector of group IDs for random effects. Required if the requested term includes random effects.
rfx_basis	An optional matrix of basis function evaluations for random effects. Required if the requested term includes random effects.

Value

A list containing the lower and upper bounds of the credible interval for the specified term. If multiple terms are requested, a named list with intervals for each term is returned.

Examples

```
n <- 100
p <- 5
X <- matrix(rnorm(n * p), nrow = n, ncol = p)
y <- 2 * X[,1] + rnorm(n)
bart_model <- bart(y_train = y, X_train = X)
intervals <- computeBARTPosteriorInterval(
 model_object = bart_model,
 terms = c("mean_forest", "y_hat"),
 X = X,
 level = 0.90
)
```

---

computeBCFPosteriorInterval

Compute BCF Posterior Credible Intervals

Description

Compute posterior credible intervals for specified terms from a fitted BCF model. Supports intervals for prognostic forests, CATE forests, variance forests, random effects, and overall mean outcome predictions.

Usage

```
computeBCFPosteriorInterval(
  model_object,
  terms,
  level = 0.95,
  scale = "linear",
  X = NULL,
```

```
  Z = NULL,
  propensity = NULL,
  rfx_group_ids = NULL,
  rfx_basis = NULL
)
```

Arguments

model_object	A fitted BCF model object of class bcfmodel.
terms	A character string specifying the model term(s) for which to compute intervals. Options are "prognostic_function", "mu", "cate", "tau", "variance_forest", "rfx", or "y_hat". The treatment effect terms follow a three-level hierarchy: • "tau" returns tau_0 + tau(X): the parametric treatment intercept (if sampled) plus the treatment forest. This matches model$tau_hat_train / model$tau_hat_test. • "cate" additionally folds in the random slope on treatment when random effects are fit with rfx_model_spec = "intercept_plus_treatment"; otherwise it is identical to "tau". • The raw forest-only component (without tau_0) is not directly returned by this method; use model$forests_tau to access it. Similarly for the prognostic term: "mu" returns the prognostic forest only, while "prognostic_function" additionally folds in the random intercept when rfx_model_spec is "intercept_only" or "intercept_plus_treatment"; otherwise the two are identical.
level	A numeric value between 0 and 1 specifying the credible interval level (default is 0.95 for a 95% credible interval).
scale	(Optional) Scale of mean function predictions. Options are "linear", which returns predictions on the original scale of the mean forest / RFX terms, and "probability", which transforms predictions into a probability of observing y == 1. "probability" is only valid for models fit with a probit outcome model. Default: "linear".
X	(Optional) A matrix or data frame of covariates at which to compute the intervals. Required if the requested term depends on covariates (e.g., prognostic forest, CATE forest, variance forest, or overall predictions).
Z	(Optional) A vector or matrix of treatment assignments. Required if the requested term is "y_hat" (overall predictions).
propensity	(Optional) A vector or matrix of propensity scores. Required if the underlying model depends on user-provided propensities.
rfx_group_ids	An optional vector of group IDs for random effects. Required if the requested term includes random effects.
rfx_basis	An optional matrix of basis function evaluations for random effects. Required if the requested term includes random effects.

Value

A list containing the lower and upper bounds of the credible interval for the specified term. If multiple terms are requested, a named list with intervals for each term is returned.

Examples

```
n <- 100
p <- 5
X <- matrix(rnorm(n * p), nrow = n, ncol = p)
pi_X <- pnorm(0.5 * X[,1])
Z <- rbinom(n, 1, pi_X)
mu_X <- X[,1]
tau_X <- 0.25 * X[,2]
y <- mu_X + tau_X * Z + rnorm(n)
bcf_model <- bcf(X_train = X, Z_train = Z, y_train = y,
                 propensity_train = pi_X)
intervals <- computeBCFPosteriorInterval(
 model_object = bcf_model,
 terms = c("prognostic_function", "cate"),
 X = X,
 Z = Z,
 propensity = pi_X,
 level = 0.90
)
```

---

computeContrastBARTModel

Compute Contrast for BART Model

Description

Compute a contrast using a BART model by making two sets of outcome predictions and taking their difference. This function provides the flexibility to compute any contrast of interest by specifying covariates, leaf basis, and random effects bases / IDs for both sides of a two term contrast. For simplicity, we refer to the subtrahend of the contrast as the "control" or `Y0` term and the minuend of the contrast as the `Y1` term, though the requested contrast need not match the "control vs treatment" terminology of a classic two-treatment causal inference problem. We mirror the function calls and terminology of the `predict.bartmodel` function, labeling each prediction data term with a `1` to denote its contribution to the treatment prediction of a contrast and `0` to denote inclusion in the control prediction.

Only valid when there is either a mean forest or a random effects term in the BART model.

Usage

```
computeContrastBARTModel(
  object,
  X_0,
  X_1,
```

```
  leaf_basis_0 = NULL,
  leaf_basis_1 = NULL,
  rfx_group_ids_0 = NULL,
  rfx_group_ids_1 = NULL,
  rfx_basis_0 = NULL,
  rfx_basis_1 = NULL,
  type = "posterior",
  scale = "linear"
)
```

Arguments

`object`	Object of type `bart` containing draws of a regression forest and associated sampling outputs.
`X_0`	Covariates used for prediction in the "control" case. Must be a matrix or dataframe.
`X_1`	Covariates used for prediction in the "treatment" case. Must be a matrix or dataframe.
`leaf_basis_0`	(Optional) Bases used for prediction in the "control" case (by e.g. dot product with leaf values). Default: `NULL`.
`leaf_basis_1`	(Optional) Bases used for prediction in the "treatment" case (by e.g. dot product with leaf values). Default: `NULL`.
`rfx_group_ids_0`	(Optional) Test set group labels used for prediction from an additive random effects model in the "control" case. We do not currently support (but plan to in the near future), test set evaluation for group labels that were not in the training set. Must be a vector.
`rfx_group_ids_1`	(Optional) Test set group labels used for prediction from an additive random effects model in the "treatment" case. We do not currently support (but plan to in the near future), test set evaluation for group labels that were not in the training set. Must be a vector.
`rfx_basis_0`	(Optional) Test set basis for used for prediction from an additive random effects model in the "control" case. Must be a matrix or vector.
`rfx_basis_1`	(Optional) Test set basis for used for prediction from an additive random effects model in the "treatment" case. Must be a matrix or vector.
`type`	(Optional) Aggregation level of the contrast. Options are "mean", which averages the contrast evaluations over every draw of a BART model, and "posterior", which returns the entire matrix of posterior contrast estimates. Default: "posterior".
`scale`	(Optional) Scale of the contrast. Options are "linear", which returns contrast of predictions on the original scale of the mean forest / RFX terms, and "probability". `scale = "probability"` is only valid for models fit with a probit / cloglog links on binary or ordinal outcomes. For binary outcome models, `scale = "probability"` will return contrasts of the probability that `y == 1`. For ordinal outcome models, `scale = "probability"` will return contrasts over the "survival function" `P(y > k)` for `k = 1, 2, ..., K-1` where `K` is the total number of categories. Default: "linear".

Value

Contrast matrix or vector, depending on whether type = "mean" or "posterior".

Examples

```
n <- 100
p <- 5
X <- matrix(runif(n*p), ncol = p)
W <- matrix(runif(n*1), ncol = 1)
f_XW <- (
    ((0 <= X[,1]) & (0.25 > X[,1])) * (-7.5*W[,1]) +
    ((0.25 <= X[,1]) & (0.5 > X[,1])) * (-2.5*W[,1]) +
    ((0.5 <= X[,1]) & (0.75 > X[,1])) * (2.5*W[,1]) +
    ((0.75 <= X[,1]) & (1 > X[,1])) * (7.5*W[,1])
)
noise_sd <- 1
y <- f_XW + rnorm(n, 0, noise_sd)
test_set_pct <- 0.2
n_test <- round(test_set_pct*n)
n_train <- n - n_test
test_inds <- sort(sample(1:n, n_test, replace = FALSE))
train_inds <- (1:n)[!((1:n) %in% test_inds)]
X_test <- X[test_inds,]
X_train <- X[train_inds,]
W_test <- W[test_inds,]
W_train <- W[train_inds,]
y_test <- y[test_inds]
y_train <- y[train_inds]
bart_model <- bart(X_train = X_train, leaf_basis_train = W_train, y_train = y_train,
                   num_gfr = 10, num_burnin = 0, num_mcmc = 10)
contrast_test <- computeContrastBARTModel(
    bart_model,
    X_0 = X_test,
    X_1 = X_test,
    leaf_basis_0 = matrix(0, nrow = n_test, ncol = 1),
    leaf_basis_1 = matrix(1, nrow = n_test, ncol = 1),
    type = "posterior",
    scale = "linear"
)
```

`computeContrastBCFModel` *Compute Contrast for BCF Model*

Description

Compute a contrast using a BCF model by making two sets of outcome predictions and taking their difference. For simple BCF models with binary treatment, this will yield the same prediction as requesting `terms = "cate"` in the `predict.bcfmodel` function. For more general models, such as

models with continuous / multivariate treatments or an additive random effects term with a coefficient on the treatment, this function provides the flexibility to compute a any contrast of interest by specifying covariates, treatment, and random effects bases and IDs for both sides of a two term contrast. For simplicity, we refer to the subtrahend of the contrast as the "control" or `Y0` term and the minuend of the contrast as the `Y1` term, though the requested contrast need not match the "control vs treatment" terminology of a classic two-arm experiment. We mirror the function calls and terminology of the `predict.bcfmodel` function, labeling each prediction data term with a `1` to denote its contribution to the treatment prediction of a contrast and `0` to denote inclusion in the control prediction.

Usage

```
computeContrastBCFModel(
  object,
  X_0,
  X_1,
  Z_0,
  Z_1,
  propensity_0 = NULL,
  propensity_1 = NULL,
  rfx_group_ids_0 = NULL,
  rfx_group_ids_1 = NULL,
  rfx_basis_0 = NULL,
  rfx_basis_1 = NULL,
  type = "posterior",
  scale = "linear"
)
```

Arguments

`object`	Object of type `bcfmodel` containing draws of a Bayesian causal forest model and associated sampling outputs.
`X_0`	Covariates used for prediction in the "control" case. Must be a matrix or dataframe.
`X_1`	Covariates used for prediction in the "treatment" case. Must be a matrix or dataframe.
`Z_0`	Treatments used for prediction in the "control" case. Must be a matrix or vector.
`Z_1`	Treatments used for prediction in the "treatment" case. Must be a matrix or vector.
`propensity_0`	(Optional) Propensities used for prediction in the "control" case. Must be a matrix or vector.
`propensity_1`	(Optional) Propensities used for prediction in the "treatment" case. Must be a matrix or vector.
`rfx_group_ids_0`	(Optional) Test set group labels used for prediction from an additive random effects model in the "control" case. We do not currently support (but plan to in the near future), test set evaluation for group labels that were not in the training set. Must be a vector.

`rfx_group_ids_1`
: (Optional) Test set group labels used for prediction from an additive random effects model in the "treatment" case. We do not currently support (but plan to in the near future), test set evaluation for group labels that were not in the training set. Must be a vector.

`rfx_basis_0`
: (Optional) Test set basis for used for prediction from an additive random effects model in the "control" case. Must be a matrix or vector.

`rfx_basis_1`
: (Optional) Test set basis for used for prediction from an additive random effects model in the "treatment" case. Must be a matrix or vector.

`type`
: (Optional) Aggregation level of the contrast. Options are "mean", which averages the contrast evaluations over every draw of a BCF model, and "posterior", which returns the entire matrix of posterior contrast estimates. Default: "posterior".

`scale`
: (Optional) Scale of the contrast. Options are "linear", which returns a contrast on the original scale of the mean forest / RFX terms, and "probability", which transforms each contrast term into a probability of observing `y == 1` before taking their difference. "probability" is only valid for models fit with a probit outcome model. Default: "linear".

Value

List of prediction matrices or single prediction matrix / vector, depending on the terms requested.

Examples

```
n <- 500
p <- 5
X <- matrix(runif(n*p), ncol = p)
mu_x <- (
    ((0 <= X[,1]) & (0.25 > X[,1])) * (-7.5) +
    ((0.25 <= X[,1]) & (0.5 > X[,1])) * (-2.5) +
    ((0.5 <= X[,1]) & (0.75 > X[,1])) * (2.5) +
    ((0.75 <= X[,1]) & (1 > X[,1])) * (7.5)
)
pi_x <- (
    ((0 <= X[,1]) & (0.25 > X[,1])) * (0.2) +
    ((0.25 <= X[,1]) & (0.5 > X[,1])) * (0.4) +
    ((0.5 <= X[,1]) & (0.75 > X[,1])) * (0.6) +
    ((0.75 <= X[,1]) & (1 > X[,1])) * (0.8)
)
tau_x <- (
    ((0 <= X[,2]) & (0.25 > X[,2])) * (0.5) +
    ((0.25 <= X[,2]) & (0.5 > X[,2])) * (1.0) +
    ((0.5 <= X[,2]) & (0.75 > X[,2])) * (1.5) +
    ((0.75 <= X[,2]) & (1 > X[,2])) * (2.0)
)
Z <- rbinom(n, 1, pi_x)
noise_sd <- 1
y <- mu_x + tau_x*Z + rnorm(n, 0, noise_sd)
test_set_pct <- 0.2
```

```
n_test <- round(test_set_pct*n)
n_train <- n - n_test
test_inds <- sort(sample(1:n, n_test, replace = FALSE))
train_inds <- (1:n)[!((1:n) %in% test_inds)]
X_test <- X[test_inds,]
X_train <- X[train_inds,]
pi_test <- pi_x[test_inds]
pi_train <- pi_x[train_inds]
Z_test <- Z[test_inds]
Z_train <- Z[train_inds]
y_test <- y[test_inds]
y_train <- y[train_inds]
mu_test <- mu_x[test_inds]
mu_train <- mu_x[train_inds]
tau_test <- tau_x[test_inds]
tau_train <- tau_x[train_inds]
bcf_model <- bcf(X_train = X_train, Z_train = Z_train, y_train = y_train,
                 propensity_train = pi_train, num_gfr = 10,
                 num_burnin = 0, num_mcmc = 10)
tau_hat_test <- computeContrastBCFModel(
    bcf_model, X_0=X_test, X_1=X_test, Z_0=rep(0, n_test), Z_1=rep(1, n_test),
    propensity_0 = pi_test, propensity_1 = pi_test
)
```

`CppJson` *JSON C++ Object Wrapper*

Description

Wrapper around a C++ `nlohmann::json` object

This class is intended for advanced use cases in which users require detailed control of sampling algorithms and data structures. Minimal input validation and error checks are performed – users are responsible for providing the correct inputs. For tutorials on the "proper" usage of the stochtree's advanced workflow, we provide several vignettes at https://stochtree.ai/

Public fields

`json_ptr` External pointer to a C++ nlohmann::json object

`num_forests` Number of forests in the nlohmann::json object

`forest_labels` Names of forest objects in the overall nlohmann::json object

`num_rfx` Number of random effects terms in the nlohman::json object

`rfx_container_labels` Names of rfx container objects in the overall nlohmann::json object

`rfx_mapper_labels` Names of rfx label mapper objects in the overall nlohmann::json object

`rfx_groupid_labels` Names of rfx group id objects in the overall nlohmann::json object

Methods

Public methods:

- CppJson$new()
- CppJson$add_forest()
- CppJson$add_random_effects()
- CppJson$add_scalar()
- CppJson$add_integer()
- CppJson$add_boolean()
- CppJson$add_string()
- CppJson$add_vector()
- CppJson$add_integer_vector()
- CppJson$add_string_vector()
- CppJson$add_list()
- CppJson$add_string_list()
- CppJson$get_scalar()
- CppJson$get_integer()
- CppJson$get_boolean()
- CppJson$get_string()
- CppJson$get_vector()
- CppJson$get_integer_vector()
- CppJson$get_string_vector()
- CppJson$get_numeric_list()
- CppJson$get_string_list()
- CppJson$return_json_string()
- CppJson$save_file()
- CppJson$load_from_file()
- CppJson$load_from_string()

Method `new()`**:** Create a new CppJson object.

Usage:

```
CppJson$new()
```

Returns: A new `CppJson` object.

Method `add_forest()`**:** Convert a forest container to json and add to the current `CppJson` object

Usage:

```
CppJson$add_forest(forest_samples)
```

Arguments:

`forest_samples` `ForestSamples` R class

Returns: None

Method `add_random_effects()`**:** Convert a random effects container to json and add to the current `CppJson` object

Usage:

```
CppJson$add_random_effects(rfx_samples)
```

Arguments:

`rfx_samples` `RandomEffectSamples` R class

Returns: None

Method `add_scalar()`**:** Add a scalar to the json object under the name "field_name" (with optional subfolder "subfolder_name")

Usage:

```
CppJson$add_scalar(field_name, field_value, subfolder_name = NULL)
```

Arguments:

`field_name` The name of the field to be added to json

`field_value` Numeric value of the field to be added to json

`subfolder_name` (Optional) Name of the subfolder / hierarchy under which to place the value

Returns: None

Method `add_integer()`**:** Add a scalar to the json object under the name "field_name" (with optional subfolder "subfolder_name")

Usage:

```
CppJson$add_integer(field_name, field_value, subfolder_name = NULL)
```

Arguments:

`field_name` The name of the field to be added to json

`field_value` Integer value of the field to be added to json

`subfolder_name` (Optional) Name of the subfolder / hierarchy under which to place the value

Returns: None

Method `add_boolean()`**:** Add a boolean value to the json object under the name "field_name" (with optional subfolder "subfolder_name")

Usage:

```
CppJson$add_boolean(field_name, field_value, subfolder_name = NULL)
```

Arguments:

`field_name` The name of the field to be added to json

`field_value` Numeric value of the field to be added to json

`subfolder_name` (Optional) Name of the subfolder / hierarchy under which to place the value

Returns: None

Method `add_string()`**:** Add a string value to the json object under the name "field_name" (with optional subfolder "subfolder_name")

Usage:

```
CppJson$add_string(field_name, field_value, subfolder_name = NULL)
```

Arguments:

`field_name` The name of the field to be added to json

`field_value` Numeric value of the field to be added to json

`subfolder_name` (Optional) Name of the subfolder / hierarchy under which to place the value

Returns: None

Method `add_vector()`**:** Add a vector to the json object under the name "field_name" (with optional subfolder "subfolder_name")

Usage:

```
CppJson$add_vector(field_name, field_vector, subfolder_name = NULL)
```

Arguments:

`field_name` The name of the field to be added to json

`field_vector` Vector to be stored in json

`subfolder_name` (Optional) Name of the subfolder / hierarchy under which to place the value

Returns: None

Method `add_integer_vector()`**:** Add an integer vector to the json object under the name "field_name" (with optional subfolder "subfolder_name")

Usage:

```
CppJson$add_integer_vector(field_name, field_vector, subfolder_name = NULL)
```

Arguments:

`field_name` The name of the field to be added to json

`field_vector` Vector to be stored in json

`subfolder_name` (Optional) Name of the subfolder / hierarchy under which to place the value

Returns: None

Method `add_string_vector()`**:** Add an array to the json object under the name "field_name" (with optional subfolder "subfolder_name")

Usage:

```
CppJson$add_string_vector(field_name, field_vector, subfolder_name = NULL)
```

Arguments:

`field_name` The name of the field to be added to json

`field_vector` Character vector to be stored in json

`subfolder_name` (Optional) Name of the subfolder / hierarchy under which to place the value

Returns: None

Method `add_list()`**:** Add a list of vectors (as an object map of arrays) to the json object under the name "field_name"

Usage:

```
CppJson$add_list(field_name, field_list)
```

Arguments:

`field_name` The name of the field to be added to json

`field_list` List to be stored in json

Returns: None

Method `add_string_list()`: Add a list of vectors (as an object map of arrays) to the json object under the name "field_name"

Usage:

```
CppJson$add_string_list(field_name, field_list)
```

Arguments:

`field_name` The name of the field to be added to json

`field_list` List to be stored in json

Returns: None

Method `get_scalar()`: Retrieve a scalar value from the json object under the name "field_name" (with optional subfolder "subfolder_name")

Usage:

```
CppJson$get_scalar(field_name, subfolder_name = NULL)
```

Arguments:

`field_name` The name of the field to be accessed from json

`subfolder_name` (Optional) Name of the subfolder / hierarchy under which the field is stored

Returns: None

Method `get_integer()`: Retrieve a integer value from the json object under the name "field_name" (with optional subfolder "subfolder_name")

Usage:

```
CppJson$get_integer(field_name, subfolder_name = NULL)
```

Arguments:

`field_name` The name of the field to be accessed from json

`subfolder_name` (Optional) Name of the subfolder / hierarchy under which the field is stored

Returns: None

Method `get_boolean()`: Retrieve a boolean value from the json object under the name "field_name" (with optional subfolder "subfolder_name")

Usage:

```
CppJson$get_boolean(field_name, subfolder_name = NULL)
```

Arguments:

`field_name` The name of the field to be accessed from json

`subfolder_name` (Optional) Name of the subfolder / hierarchy under which the field is stored

Returns: None

Method `get_string()`: Retrieve a string value from the json object under the name "field_name" (with optional subfolder "subfolder_name")

Usage:

```
CppJson$get_string(field_name, subfolder_name = NULL)
```

Arguments:

`field_name` The name of the field to be accessed from json

`subfolder_name` (Optional) Name of the subfolder / hierarchy under which the field is stored

Returns: None

Method `get_vector()`**:** Retrieve a vector from the json object under the name "field_name" (with optional subfolder "subfolder_name")

Usage:

```
CppJson$get_vector(field_name, subfolder_name = NULL)
```

Arguments:

`field_name` The name of the field to be accessed from json

`subfolder_name` (Optional) Name of the subfolder / hierarchy under which the field is stored

Returns: None

Method `get_integer_vector()`**:** Retrieve an integer vector from the json object under the name "field_name" (with optional subfolder "subfolder_name")

Usage:

```
CppJson$get_integer_vector(field_name, subfolder_name = NULL)
```

Arguments:

`field_name` The name of the field to be accessed from json

`subfolder_name` (Optional) Name of the subfolder / hierarchy under which the field is stored

Returns: None

Method `get_string_vector()`**:** Retrieve a character vector from the json object under the name "field_name" (with optional subfolder "subfolder_name")

Usage:

```
CppJson$get_string_vector(field_name, subfolder_name = NULL)
```

Arguments:

`field_name` The name of the field to be accessed from json

`subfolder_name` (Optional) Name of the subfolder / hierarchy under which the field is stored

Returns: None

Method `get_numeric_list()`**:** Reconstruct a list of numeric vectors from the json object stored under "field_name"

Usage:

```
CppJson$get_numeric_list(field_name, key_names)
```

Arguments:

`field_name` The name of the field to be added to json

`key_names` Vector of names of list elements (each of which is a vector)

Returns: None

Method `get_string_list()`**:** Reconstruct a list of string vectors from the json object stored under "field_name"

Usage:

```
CppJson$get_string_list(field_name, key_names)
```

Arguments:

`field_name` The name of the field to be added to json

`key_names` Vector of names of list elements (each of which is a vector)

Returns: None

Method `return_json_string()`**:** Convert a JSON object to in-memory string

Usage:

```
CppJson$return_json_string()
```

Returns: JSON string

Method `save_file()`**:** Save a json object to file

Usage:

```
CppJson$save_file(filename)
```

Arguments:

`filename` String of filepath, must end in ".json"

Returns: None

Method `load_from_file()`**:** Load a json object from file

Usage:

```
CppJson$load_from_file(filename)
```

Arguments:

`filename` String of filepath, must end in ".json"

Returns: None

Method `load_from_string()`**:** Load a json object from string

Usage:

```
CppJson$load_from_string(json_string)
```

Arguments:

`json_string` JSON string dump

Returns: None

CppRNG *Random Number Generator C++ Wrapper*

Description

Wrapper around a C++ random number generator object (for reproducibility). The class persists a C++ random number generator throughout an R session to ensure a given seed generates the same outputs (on the same OS). If no seed is provided, the C++ random number generator is initialized using `std::random_device`.

Public fields

`rng_ptr` External pointer to a C++ std::mt19937 class

Methods

Public methods:

- `CppRNG$new()`

Method `new()`**:** Create a new CppRNG object.

Usage:

```
CppRNG$new(random_seed = -1)
```

Arguments:

`random_seed` (Optional) random seed for sampling

Returns: A new `CppRNG` object.

createCppRNG *Create CppRNG Object*

Description

Create an R class that wraps a C++ random number generator

Usage

```
createCppRNG(random_seed = -1)
```

Arguments

`random_seed`	(Optional) random seed for sampling

Value

CppRng object

Examples

```
rng <- createCppRNG(1234)
rng <- createCppRNG()
```

`createForest` *Create Forest Object*

Description

Create a forest

This function is intended for advanced use cases in which users require detailed control of sampling algorithms and data structures. Minimal input validation and error checks are performed – users are responsible for providing the correct inputs. For tutorials on the "proper" usage of the stochtree's advanced workflow, we provide several vignettes at https://stochtree.ai/

Usage

```
createForest(
  num_trees,
  leaf_dimension = 1,
  is_leaf_constant = FALSE,
  is_exponentiated = FALSE
)
```

Arguments

`num_trees`	Number of trees in the forest
`leaf_dimension`	Dimensionality of the outcome model
`is_leaf_constant`	Whether leaf is constant
`is_exponentiated`	Whether forest predictions should be exponentiated before being returned

Value

`Forest` object

Examples

```
num_trees <- 100
leaf_dimension <- 2
is_leaf_constant <- FALSE
is_exponentiated <- FALSE
forest <- createForest(num_trees, leaf_dimension, is_leaf_constant, is_exponentiated)
```

createForestDataset *Create ForestDataset Object*

Description

Create a forest dataset object

This function is intended for advanced use cases in which users require detailed control of sampling algorithms and data structures. Minimal input validation and error checks are performed – users are responsible for providing the correct inputs. For tutorials on the "proper" usage of the stochtree's advanced workflow, we provide several vignettes at https://stochtree.ai/

Usage

```
createForestDataset(covariates, basis = NULL, variance_weights = NULL)
```

Arguments

covariates	Matrix of covariates
basis	(Optional) Matrix of bases used to define a leaf regression
variance_weights	(Optional) Vector of observation-specific variance weights

Value

ForestDataset object

Examples

```
covariate_matrix <- matrix(runif(10*100), ncol = 10)
basis_matrix <- matrix(rnorm(3*100), ncol = 3)
weight_vector <- rnorm(100)
forest_dataset <- createForestDataset(covariate_matrix)
forest_dataset <- createForestDataset(covariate_matrix, basis_matrix)
forest_dataset <- createForestDataset(covariate_matrix, basis_matrix, weight_vector)
```

createForestModel *Create ForestModel Object*

Description

Create a forest model object

This function is intended for advanced use cases in which users require detailed control of sampling algorithms and data structures. Minimal input validation and error checks are performed – users are responsible for providing the correct inputs. For tutorials on the "proper" usage of the stochtree's advanced workflow, we provide several vignettes at https://stochtree.ai/

Usage

```
createForestModel(forest_dataset, forest_model_config, global_model_config)
```

Arguments

forest_dataset	ForestDataset object, used to initialize forest sampling data structures
forest_model_config	ForestModelConfig object containing forest model parameters and settings
global_model_config	GlobalModelConfig object containing global model parameters and settings

Value

ForestModel object

Examples

```
num_trees <- 100
n <- 100
p <- 10
alpha <- 0.95
beta <- 2.0
min_samples_leaf <- 2
max_depth <- 10
feature_types <- as.integer(rep(0, p))
X <- matrix(runif(n*p), ncol = p)
forest_dataset <- createForestDataset(X)
forest_model_config <- createForestModelConfig(feature_types=feature_types,
                                               num_trees=num_trees, num_features=p,
                                              num_observations=n, alpha=alpha, beta=beta,
                                               min_samples_leaf=min_samples_leaf,
                                               max_depth=max_depth, leaf_model_type=1)
global_model_config <- createGlobalModelConfig(global_error_variance=1.0)
forest_model <- createForestModel(forest_dataset, forest_model_config, global_model_config)
```

---

createForestModelConfig

Create ForestModelConfig Object

Description

Create a forest model config object

This function is intended for advanced use cases in which users require detailed control of sampling algorithms and data structures. Minimal input validation and error checks are performed – users are responsible for providing the correct inputs. For tutorials on the "proper" usage of the stochtree's advanced workflow, we provide several vignettes at https://stochtree.ai/

Usage

```
createForestModelConfig(
  feature_types = NULL,
  sweep_update_indices = NULL,
  num_trees = NULL,
  num_features = NULL,
  num_observations = NULL,
  variable_weights = NULL,
  leaf_dimension = 1,
  alpha = 0.95,
  beta = 2,
  min_samples_leaf = 5,
  max_depth = -1,
  leaf_model_type = 1,
  leaf_model_scale = NULL,
  variance_forest_shape = 1,
  variance_forest_scale = 1,
  cloglog_forest_shape = 2,
  cloglog_forest_rate = 2,
  cutpoint_grid_size = 100,
  num_features_subsample = NULL
)
```

Arguments

`feature_types`	Vector of integer-coded feature types (integers where 0 = numeric, 1 = ordered categorical, 2 = unordered categorical)
`sweep_update_indices`	Vector of (0-indexed) indices of trees to update in a sweep
`num_trees`	Number of trees in the forest being sampled
`num_features`	Number of features in training dataset
`num_observations`	Number of observations in training dataset
`variable_weights`	Vector specifying sampling probability for all p covariates in ForestDataset
`leaf_dimension`	Dimension of the leaf model (default: `1`)
`alpha`	Root node split probability in tree prior (default: `0.95`)
`beta`	Depth prior penalty in tree prior (default: `2.0`)
`min_samples_leaf`	Minimum number of samples in a tree leaf (default: `5`)
`max_depth`	Maximum depth of any tree in the ensemble in the model. Setting to `-1` does not enforce any depth limits on trees. Default: `-1`.
`leaf_model_type`	Integer specifying the leaf model type (0 = constant leaf, 1 = univariate leaf regression, 2 = multivariate leaf regression). Default: `0`.

`leaf_model_scale`
: Scale parameter used in Gaussian leaf models (can either be a scalar or a q x q matrix, where q is the dimensionality of the basis and is only >1 when `leaf_model_int = 2`). Calibrated internally as `1/num_trees`, propagated along diagonal if needed for multivariate leaf models.

`variance_forest_shape`
: Shape parameter for IG leaf models (applicable when `leaf_model_type = 3`). Default: `1`.

`variance_forest_scale`
: Scale parameter for IG leaf models (applicable when `leaf_model_type = 3`). Default: `1`.

`cloglog_forest_shape`
: Shape parameter for conditional gamma component of cloglog leaf models (applicable when `leaf_model_type = 4`). Default: `1`.

`cloglog_forest_rate`
: Rate parameter for conditional gamma component of cloglog leaf models (applicable when `leaf_model_type = 4`). Default: `1`.

`cutpoint_grid_size`
: Number of unique cutpoints to consider (default: `100`)

`num_features_subsample`
: Number of features to subsample for the GFR algorithm

Value

ForestModelConfig object

Examples

```
config <- createForestModelConfig(num_trees = 10, num_features = 5, num_observations = 100)
```

---

createForestSamples *Create ForestSamples Object*

Description

Create a container of forest samples

This function is intended for advanced use cases in which users require detailed control of sampling algorithms and data structures. Minimal input validation and error checks are performed – users are responsible for providing the correct inputs. For tutorials on the "proper" usage of the stochtree's advanced workflow, we provide several vignettes at https://stochtree.ai/

Usage

```
createForestSamples(
  num_trees,
  leaf_dimension = 1,
  is_leaf_constant = FALSE,
  is_exponentiated = FALSE
)
```

Arguments

num_trees	Number of trees
leaf_dimension	Dimensionality of the outcome model
is_leaf_constant	Whether leaf is constant
is_exponentiated	Whether forest predictions should be exponentiated before being returned

Value

ForestSamples object

Examples

```
num_trees <- 100
leaf_dimension <- 2
is_leaf_constant <- FALSE
is_exponentiated <- FALSE
forest_samples <- createForestSamples(num_trees, leaf_dimension, is_leaf_constant, is_exponentiated)
```

---

createGlobalModelConfig

Create GlobalModelConfig Object

Description

Create a global model config object

This function is intended for advanced use cases in which users require detailed control of sampling algorithms and data structures. Minimal input validation and error checks are performed – users are responsible for providing the correct inputs. For tutorials on the "proper" usage of the stochtree's advanced workflow, we provide several vignettes at https://stochtree.ai/

Usage

```
createGlobalModelConfig(global_error_variance = 1)
```

Arguments

global_error_variance	Global error variance parameter (default: 1.0)

Value

GlobalModelConfig object

Examples

```
config <- createGlobalModelConfig(global_error_variance = 100)
```

createOutcome *Create Outcome Object*

Description

Create an outcome object

This function is intended for advanced use cases in which users require detailed control of sampling algorithms and data structures. Minimal input validation and error checks are performed – users are responsible for providing the correct inputs. For tutorials on the "proper" usage of the stochtree's advanced workflow, we provide several vignettes at https://stochtree.ai/

Usage

```
createOutcome(outcome)
```

Arguments

outcome	Vector of outcome values

Value

Outcome object

Examples

```
X <- matrix(runif(10*100), ncol = 10)
y <- -5 + 10*(X[,1] > 0.5) + rnorm(100)
outcome <- createOutcome(y)
```

createRandomEffectSamples *Create RandomEffectSamples Object*

Description

Create a RandomEffectSamples object

This function is intended for advanced use cases in which users require detailed control of sampling algorithms and data structures. Minimal input validation and error checks are performed – users are responsible for providing the correct inputs. For tutorials on the "proper" usage of the stochtree's advanced workflow, we provide several vignettes at https://stochtree.ai/

Usage

```
createRandomEffectSamples(num_components, num_groups, random_effects_tracker)
```

Arguments

num_components	Number of "components" or bases defining the random effects regression
num_groups	Number of random effects groups
random_effects_tracker	Object of type RandomEffectsTracker

Value

RandomEffectSamples object

Examples

```
n <- 100
rfx_group_ids <- sample(1:2, size = n, replace = TRUE)
rfx_basis <- matrix(rep(1.0, n), ncol=1)
num_groups <- length(unique(rfx_group_ids))
num_components <- ncol(rfx_basis)
rfx_tracker <- createRandomEffectsTracker(rfx_group_ids)
rfx_samples <- createRandomEffectSamples(num_components, num_groups, rfx_tracker)
```

---

createRandomEffectsDataset

Create RandomEffectsDataset Object

Description

Create a random effects dataset object

This function is intended for advanced use cases in which users require detailed control of sampling algorithms and data structures. Minimal input validation and error checks are performed – users are responsible for providing the correct inputs. For tutorials on the "proper" usage of the stochtree's advanced workflow, we provide several vignettes at https://stochtree.ai/

Usage

```
createRandomEffectsDataset(group_labels, basis, variance_weights = NULL)
```

Arguments

group_labels	Vector of group labels
basis	Matrix of bases used to define the random effects regression (for an intercept-only model, pass an array of ones)
variance_weights	(Optional) Vector of observation-specific variance weights

Value

RandomEffectsDataset object

Examples

```
rfx_group_ids <- sample(1:2, size = 100, replace = TRUE)
rfx_basis <- matrix(rnorm(3*100), ncol = 3)
weight_vector <- rnorm(100)
rfx_dataset <- createRandomEffectsDataset(rfx_group_ids, rfx_basis)
rfx_dataset <- createRandomEffectsDataset(rfx_group_ids, rfx_basis, weight_vector)
```

---

createRandomEffectsModel

Create RandomEffectsModel Object

Description

Create a `RandomEffectsModel` object

This function is intended for advanced use cases in which users require detailed control of sampling algorithms and data structures. Minimal input validation and error checks are performed – users are responsible for providing the correct inputs. For tutorials on the "proper" usage of the stochtree's advanced workflow, we provide several vignettes at https://stochtree.ai/

Usage

```
createRandomEffectsModel(num_components, num_groups)
```

Arguments

`num_components`	Number of "components" or bases defining the random effects regression
`num_groups`	Number of random effects groups

Value

`RandomEffectsModel` object

Examples

```
n <- 100
rfx_group_ids <- sample(1:2, size = n, replace = TRUE)
rfx_basis <- matrix(rep(1.0, n), ncol=1)
num_groups <- length(unique(rfx_group_ids))
num_components <- ncol(rfx_basis)
rfx_model <- createRandomEffectsModel(num_components, num_groups)
```

createRandomEffectsTracker *Create RandomEffectsTracker Object*

Description

Create a RandomEffectsTracker object

This function is intended for advanced use cases in which users require detailed control of sampling algorithms and data structures. Minimal input validation and error checks are performed – users are responsible for providing the correct inputs. For tutorials on the "proper" usage of the stochtree's advanced workflow, we provide several vignettes at https://stochtree.ai/

Usage

```
createRandomEffectsTracker(rfx_group_indices)
```

Arguments

rfx_group_indices
: Integer indices indicating groups used to define random effects

Value

RandomEffectsTracker object

Examples

```
n <- 100
rfx_group_ids <- sample(1:2, size = n, replace = TRUE)
rfx_basis <- matrix(rep(1.0, n), ncol=1)
num_groups <- length(unique(rfx_group_ids))
num_components <- ncol(rfx_basis)
rfx_tracker <- createRandomEffectsTracker(rfx_group_ids)
```

DataPreprocessing *Data Preprocessing Routines*

Description

The functions in this group are designed to handle data preprocessing for stochastic forest models. For example, factor-valued columns in data frames are either one-hot encoded or converted to integer indices before the dataframe is converted to a standard matrix format for sampling. This preprocessing routine defines a set of "steps" that must be repeated on out-of-sample datasets before predictions can be obtained from a sampling model.

preprocessTrainData preprocesses covariates for the forest sampler routines, depending on the input type. DataFrames will be preprocessed based on their column types (numeric columns are not

modified, ordered factors are integer coded, and unordered factors are one-hot encoded). Matrices are unmodified (assuming all columns are numeric). This function also records and returns a "metadata" list with preprocessing details to ensure that other datasets can be preprocessing identically.

`preprocessPredictionData` preprocesses covariates for the forest sampler routines, based on the steps outlined in a metadata list produced by `preprocessTrainData`.

These procedures are handled internally in the `bart()` and `bcf()` functions, but they are provided in `stochtree` as convenience functions for users writing custom samplers. Furthermore, while R lists can be serialized to `RDS` format, we offer a number of JSON serialization routines for the metadata list produced by `preprocessTrainData` for consistency with the broader serialization approach of stochtree (see BARTSerialization and BCFSerialization).

Following the API for serializing `bartmodel` and `bcfmodel` objects, we can convert metadata to JSON or JSON strings via `savePreprocessorToJson` and `savePreprocessorToJsonString`. Similarly, we can reload a metadata list from JSON or JSON strings via `createPreprocessorFromJson` and `createPreprocessorFromJsonString`.

Usage

```
preprocessTrainData(input_data)

preprocessPredictionData(input_data, metadata)

savePreprocessorToJson(object)

savePreprocessorToJsonString(object)

createPreprocessorFromJson(json_object)

createPreprocessorFromJsonString(json_string)
```

Arguments

`input_data`	Covariates, provided as either a dataframe or a matrix
`metadata`	List containing information on variables, including train set categories for categorical variables
`object`	List containing information on variables, including train set categories for categorical variables
`json_object`	in-memory wrapper around JSON C++ object containing covariate preprocessor metadata
`json_string`	in-memory JSON string containing covariate preprocessor metadata

Value

`preprocessTrainData` returns a list with transformed matrix data and a "metadata" list with details on the preprocessing procedures applied. `preprocessPredictionData` returns a matrix reflecting the data transformations specified in the provided metadata list.

`savePreprocessorToJson` return an object of type `CppJson`. `savePreprocessorToJsonString` returns a string dump of the preprocessor's JSON representation.

createPreprocessorFromJson and createPreprocessorFromJsonString both return metadata lists.

Examples

```
# Check that running the same data through `preprocessTrainData`
# and `preprocessPredictionData` yields the same result
n <- 100
x1 <- rnorm(n)
x2 <- factor(sample(1:3, n, replace = TRUE), ordered = TRUE)
x3 <- factor(sample(1:3, n, replace = TRUE), ordered = FALSE)
df1 <- data.frame(x1 = x1, x2 = x2, x3 = x3)
df2 <- data.frame(x1 = x1, x2 = x2, x3 = x3)
preprocess_train_list <- preprocessTrainData(df1)
df1_process <- preprocess_train_list$data
df1_metadata <- preprocess_train_list$metadata
df2_process <- preprocessPredictionData(df2, df1_metadata)
all.equal(df1_process, df2_process)

# Save to in-memory JSON
metadata_json <- savePreprocessorToJson(df1_metadata)
# Save to JSON string
metadata_json_string <- savePreprocessorToJsonString(df1_metadata)

# Reload metadata list from in-memory JSON object
metadata_roundtrip <- createPreprocessorFromJson(metadata_json)
# Reload metadata list from JSON string
metadata_roundtrip <- createPreprocessorFromJsonString(metadata_json_string)

cov_df <- data.frame(x1 = 1:5, x2 = 5:1, x3 = 6:10)
metadata <- list(num_ordered_cat_vars = 0, num_unordered_cat_vars = 0,
                 num_numeric_vars = 3, numeric_vars = c("x1", "x2", "x3"))
X_preprocessed <- preprocessPredictionData(cov_df, metadata)
```

---

extractParameter *Extract Parameter Samples Generic Function*

Description

Generic function for extracting parameter samples from a model object (BCF, BART, etc...)

Usage

```
extractParameter(object, term)
```

Arguments

object	Fitted model object from which to extract parameter samples
term	Name of the parameter to extract (e.g., "sigma2", "y_hat_train", etc.)

Value

Parameter sample array

Examples

```
n <- 100
p <- 10
X <- matrix(runif(n*p), ncol = p)
rfx_group_ids <- sample(1:2, size = n, replace = TRUE)
rfx_basis <- rep(1.0, n)
y <- (-5 + 10*(X[,1] > 0.5)) + (-2*(rfx_group_ids==1)+2*(rfx_group_ids==2)) + rnorm(n)
bart_model <- bart(X_train=X, y_train=y, rfx_group_ids_train=rfx_group_ids,
                   rfx_basis_train = rfx_basis, num_gfr=0, num_mcmc=10)
sigma2_samples <- extractParameter(bart_model, "sigma2")
```

---

extractParameter.bartmodel

Extract BART Parameter Samples

Description

Extract a vector, matrix or array of parameter samples from a BART model by name. Random effects are handled by a separate `getRandomEffectSamples` function due to the complexity of the random effects parameters. If the requested model term is not found, an error is thrown. The following conventions are used for parameter names:

- Global error variance: `"sigma2"`, `"global_error_scale"`, `"sigma2_global"`
- Leaf scale: `"sigma2_leaf"`, `"leaf_scale"`
- In-sample mean function predictions: `"y_hat_train"`
- Test set mean function predictions: `"y_hat_test"`
- In-sample variance forest predictions: `"sigma2_x_train"`, `"var_x_train"`
- Test set variance forest predictions: `"sigma2_x_test"`, `"var_x_test"`
- Ordinal model cutpoints (valid only for ordinal cloglog models): `"cloglog_cutpoints"`, `"cutpoints"`

Usage

```
## S3 method for class 'bartmodel'
extractParameter(object, term)
```

Arguments

`object`	Object of type `bartmodel` containing draws of a BART model and associated sampling outputs.
`term`	Name of the parameter to extract (e.g., `"sigma2"`, `"y_hat_train"`, etc.)

Value

Array of parameter samples. If the underlying parameter is a scalar, this will be a vector of length `num_samples`. If the underlying parameter is vector-valued, this will be (`parameter_dimension` x `num_samples`) matrix, and if the underlying parameter is multidimensional, this will be an array of dimension (`parameter_dimension_1` x `parameter_dimension_2` x ... x `num_samples`).

Examples

```
n <- 100
p <- 5
X <- matrix(runif(n*p), ncol = p)
f_XW <- (
    ((0 <= X[,1]) & (0.25 > X[,1])) * (-7.5) +
    ((0.25 <= X[,1]) & (0.5 > X[,1])) * (-2.5) +
    ((0.5 <= X[,1]) & (0.75 > X[,1])) * (2.5) +
    ((0.75 <= X[,1]) & (1 > X[,1])) * (7.5)
)
snr <- 3
group_ids <- rep(c(1,2), n %/% 2)
rfx_coefs <- matrix(c(-1, -1, 1, 1), nrow=2, byrow=TRUE)
rfx_basis <- cbind(1, runif(n, -1, 1))
rfx_term <- rowSums(rfx_coefs[group_ids,] * rfx_basis)
E_y <- f_XW + rfx_term
y <- E_y + rnorm(n, 0, 1)*(sd(E_y)/snr)
test_set_pct <- 0.2
n_test <- round(test_set_pct*n)
n_train <- n - n_test
test_inds <- sort(sample(1:n, n_test, replace = FALSE))
train_inds <- (1:n)[!((1:n) %in% test_inds)]
X_test <- X[test_inds,]
X_train <- X[train_inds,]
y_test <- y[test_inds]
y_train <- y[train_inds]
rfx_group_ids_test <- group_ids[test_inds]
rfx_group_ids_train <- group_ids[train_inds]
rfx_basis_test <- rfx_basis[test_inds,]
rfx_basis_train <- rfx_basis[train_inds,]
rfx_term_test <- rfx_term[test_inds]
rfx_term_train <- rfx_term[train_inds]
bart_model <- bart(X_train = X_train, y_train = y_train, X_test = X_test,
                   rfx_group_ids_train = rfx_group_ids_train,
                   rfx_group_ids_test = rfx_group_ids_test,
                   rfx_basis_train = rfx_basis_train,
                   rfx_basis_test = rfx_basis_test,
                   num_gfr = 10, num_burnin = 0, num_mcmc = 10)
sigma2_samples <- extractParameter(bart_model, "sigma2")
```

`extractParameter.bcfmodel`

Extract BCF Parameter Samples

Description

Extract a vector, matrix or array of parameter samples from a BCF model by name. Random effects are handled by a separate `getRandomEffectSamples` function due to the complexity of the random effects parameters. If the requested model term is not found, an error is thrown. The following conventions are used for parameter names:

- Global error variance: `"sigma2"`, `"global_error_scale"`, `"sigma2_global"`
- Prognostic forest leaf scale: `"sigma2_leaf_mu"`, `"leaf_scale_mu"`, `"mu_leaf_scale"`
- Treatment effect forest leaf scale: `"sigma2_leaf_tau"`, `"leaf_scale_tau"`, `"tau_leaf_scale"`
- Adaptive coding parameters: `"adaptive_coding"` (returns both the control and treated parameters jointly, with control in the first row and treated in the second row)
- In-sample mean function predictions: `"y_hat_train"`
- Test set mean function predictions: `"y_hat_test"`
- In-sample treatment effect forest predictions: `"tau_hat_train"`
- Test set treatment effect forest predictions: `"tau_hat_test"`
- Treatment effect intercept: `"tau_0"`, `"treatment_intercept"`, `"tau_intercept"`
- In-sample variance forest predictions: `"sigma2_x_train"`, `"var_x_train"`
- Test set variance forest predictions: `"sigma2_x_test"`, `"var_x_test"`

Usage

```
## S3 method for class 'bcfmodel'
extractParameter(object, term)
```

Arguments

`object`	Object of type `bcfmodel` containing draws of a BCF model and associated sampling outputs.
`term`	Name of the parameter to extract (e.g., `"sigma2"`, `"y_hat_train"`, etc.)

Value

Array of parameter samples. If the underlying parameter is a scalar, this will be a vector of length `num_samples`. If the underlying parameter is vector-valued, this will be (`parameter_dimension` x `num_samples`) matrix, and if the underlying parameter is multidimensional, this will be an array of dimension (`parameter_dimension_1` x `parameter_dimension_2` x ... x `num_samples`).

Examples

```
n <- 500
p <- 5
X <- matrix(runif(n*p), ncol = p)
mu_x <- (
    ((0 <= X[,1]) & (0.25 > X[,1])) * (-7.5) +
    ((0.25 <= X[,1]) & (0.5 > X[,1])) * (-2.5) +
    ((0.5 <= X[,1]) & (0.75 > X[,1])) * (2.5) +
```

```
    ((0.75 <= X[,1]) & (1 > X[,1])) * (7.5)
)
pi_x <- (
    ((0 <= X[,1]) & (0.25 > X[,1])) * (0.2) +
    ((0.25 <= X[,1]) & (0.5 > X[,1])) * (0.4) +
    ((0.5 <= X[,1]) & (0.75 > X[,1])) * (0.6) +
    ((0.75 <= X[,1]) & (1 > X[,1])) * (0.8)
)
tau_x <- (
    ((0 <= X[,2]) & (0.25 > X[,2])) * (0.5) +
    ((0.25 <= X[,2]) & (0.5 > X[,2])) * (1.0) +
    ((0.5 <= X[,2]) & (0.75 > X[,2])) * (1.5) +
    ((0.75 <= X[,2]) & (1 > X[,2])) * (2.0)
)
Z <- rbinom(n, 1, pi_x)
E_XZ <- mu_x + Z*tau_x
snr <- 3
rfx_group_ids <- rep(c(1,2), n %/% 2)
rfx_coefs <- matrix(c(-1, -1, 1, 1), nrow=2, byrow=TRUE)
rfx_basis <- cbind(1, runif(n, -1, 1))
rfx_term <- rowSums(rfx_coefs[rfx_group_ids,] * rfx_basis)
y <- E_XZ + rfx_term + rnorm(n, 0, 1)*(sd(E_XZ)/snr)
bcf_model <- bcf(X_train = X, y_train = y, Z_train = Z,
                 rfx_group_ids_train = rfx_group_ids,
                 rfx_basis_train = rfx_basis,
                 num_gfr = 10, num_burnin = 0, num_mcmc = 10)
sigma2_samples <- extractParameter(bcf_model, "sigma2")
```

`Forest` *Forest C++ Wrapper*

Description

Wrapper around a C++ class that stores a single ensemble of decision trees (often treated as the "active forest" / current state of a forest term in a sampling loop in R)

This class is intended for advanced use cases in which users require detailed control of sampling algorithms and data structures. Minimal input validation and error checks are performed – users are responsible for providing the correct inputs. For tutorials on the "proper" usage of the stochtree's advanced workflow, we provide several vignettes at https://stochtree.ai/

Public fields

`forest_ptr` External pointer to a C++ TreeEnsemble class

`internal_forest_is_empty` Whether the forest has not yet been "initialized" such that its `predict` function can be called.

Methods

Public methods:

- Forest$new()
- Forest$merge_forest()
- Forest$add_constant()
- Forest$multiply_constant()
- Forest$predict()
- Forest$predict_raw()
- Forest$set_root_leaves()
- Forest$prepare_for_sampler()
- Forest$adjust_residual()
- Forest$num_trees()
- Forest$leaf_dimension()
- Forest$is_leaf_constant()
- Forest$is_exponentiated()
- Forest$add_numeric_split_tree()
- Forest$get_tree_leaves()
- Forest$get_tree_split_counts()
- Forest$get_forest_split_counts()
- Forest$tree_max_depth()
- Forest$average_max_depth()
- Forest$is_empty()

Method `new()`**:** Create a new Forest object.

Usage:

```
Forest$new(
  num_trees,
  leaf_dimension = 1,
  is_leaf_constant = FALSE,
  is_exponentiated = FALSE
)
```

Arguments:

`num_trees` Number of trees in the forest

`leaf_dimension` Dimensionality of the outcome model

`is_leaf_constant` Whether leaf is constant

`is_exponentiated` Whether forest predictions should be exponentiated before being returned

Returns: A new `Forest` object.

Method `merge_forest()`**:** Create a larger forest by merging the trees of this forest with those of another forest

Usage:

```
Forest$merge_forest(forest)
```

Arguments:

`forest` Forest to be merged into this forest

Method `add_constant()`**:** Add a constant value to every leaf of every tree in an ensemble. If leaves are multi-dimensional, `constant_value` will be added to every dimension of the leaves.

Usage:

```
Forest$add_constant(constant_value)
```

Arguments:

`constant_value` Value that will be added to every leaf of every tree

Method `multiply_constant()`**:** Multiply every leaf of every tree by a constant value. If leaves are multi-dimensional, `constant_multiple` will be multiplied through every dimension of the leaves.

Usage:

```
Forest$multiply_constant(constant_multiple)
```

Arguments:

`constant_multiple` Value that will be multiplied by every leaf of every tree

Method `predict()`**:** Predict forest on every sample in `forest_dataset`

Usage:

```
Forest$predict(forest_dataset)
```

Arguments:

`forest_dataset` `ForestDataset` R class

Returns: vector of predictions with as many rows as in `forest_dataset`

Method `predict_raw()`**:** Predict "raw" leaf values (without being multiplied by basis) for every sample in `forest_dataset`

Usage:

```
Forest$predict_raw(forest_dataset)
```

Arguments:

`forest_dataset` `ForestDataset` R class

Returns: Array of predictions for each observation in `forest_dataset` and each sample in the `ForestSamples` class with each prediction having the dimensionality of the forests' leaf model. In the case of a constant leaf model or univariate leaf regression, this array is a vector (length is the number of observations). In the case of a multivariate leaf regression, this array is a matrix (number of observations by leaf model dimension, number of samples).

Method `set_root_leaves()`**:** Set a constant predicted value for every tree in the ensemble. Stops program if any tree is more than a root node.

Usage:

```
Forest$set_root_leaves(leaf_value)
```

Arguments:

`leaf_value` Constant leaf value(s) to be fixed for each tree in the ensemble indexed by `forest_num`. Can be either a single number or a vector, depending on the forest's leaf dimension.

Method `prepare_for_sampler()`**:** Set a constant predicted value for every tree in the ensemble. Stops program if any tree is more than a root node.

Usage:

```
Forest$prepare_for_sampler(
  dataset,
  outcome,
  forest_model,
  leaf_model_int,
  leaf_value
)
```

Arguments:

`dataset` `ForestDataset` Dataset class (covariates, basis, etc...)

`outcome` `Outcome` Outcome class (residual / partial residual)

`forest_model` `ForestModel` object storing tracking structures used in training / sampling

`leaf_model_int` Integer value encoding the leaf model type (0 = constant gaussian, 1 = univariate gaussian, 2 = multivariate gaussian, 3 = log linear variance).

`leaf_value` Constant leaf value(s) to be fixed for each tree in the ensemble indexed by `forest_num`. Can be either a single number or a vector, depending on the forest's leaf dimension.

Method `adjust_residual()`**:** Adjusts residual based on the predictions of a forest

This is typically run just once at the beginning of a forest sampling algorithm. After trees are initialized with constant root node predictions, their root predictions are subtracted out of the residual.

Usage:

```
Forest$adjust_residual(dataset, outcome, forest_model, requires_basis, add)
```

Arguments:

`dataset` `ForestDataset` object storing the covariates and bases for a given forest

`outcome` `Outcome` object storing the residuals to be updated based on forest predictions

`forest_model` `ForestModel` object storing tracking structures used in training / sampling

`requires_basis` Whether or not a forest requires a basis for prediction

`add` Whether forest predictions should be added to or subtracted from residuals

Method `num_trees()`**:** Return number of trees in each ensemble of a `Forest` object

Usage:

```
Forest$num_trees()
```

Returns: Tree count

Method `leaf_dimension()`**:** Return output dimension of trees in a `Forest` object

Usage:

```
Forest$leaf_dimension()
```

Returns: Leaf node parameter size

Method `is_leaf_constant()`**:** Return constant leaf status of trees in a `Forest` object

Usage:

```
Forest$is_leaf_constant()
```

Returns: TRUE if leaves are constant, FALSE otherwise

Method `is_exponentiated()`**:** Return exponentiation status of trees in a `Forest` object

Usage:

```
Forest$is_exponentiated()
```

Returns: TRUE if leaf predictions must be exponentiated, FALSE otherwise

Method `add_numeric_split_tree()`**:** Add a numeric (i.e. `X[,i] <= c`) split to a given tree in the ensemble

Usage:

```
Forest$add_numeric_split_tree(
  tree_num,
  leaf_num,
  feature_num,
  split_threshold,
  left_leaf_value,
  right_leaf_value
)
```

Arguments:

`tree_num` Index of the tree to be split

`leaf_num` Leaf to be split

`feature_num` Feature that defines the new split

`split_threshold` Value that defines the cutoff of the new split

`left_leaf_value` Value (or vector of values) to assign to the newly created left node

`right_leaf_value` Value (or vector of values) to assign to the newly created right node

Method `get_tree_leaves()`**:** Retrieve a vector of indices of leaf nodes for a given tree in a given forest

Usage:

```
Forest$get_tree_leaves(tree_num)
```

Arguments:

`tree_num` Index of the tree for which leaf indices will be retrieved

Method `get_tree_split_counts()`**:** Retrieve a vector of split counts for every training set variable in a given tree in the forest

Usage:

```
Forest$get_tree_split_counts(tree_num, num_features)
```

Arguments:

`tree_num` Index of the tree for which split counts will be retrieved

`num_features` Total number of features in the training set

Method `get_forest_split_counts()`**:** Retrieve a vector of split counts for every training set variable in the forest

Usage:

```
Forest$get_forest_split_counts(num_features)
```

Arguments:

`num_features` Total number of features in the training set

Method `tree_max_depth()`**:** Maximum depth of a specific tree in the forest

Usage:

```
Forest$tree_max_depth(tree_num)
```

Arguments:

`tree_num` Tree index within forest

Returns: Maximum leaf depth

Method `average_max_depth()`**:** Average the maximum depth of each tree in the forest

Usage:

```
Forest$average_max_depth()
```

Returns: Average maximum depth

Method `is_empty()`**:** When a forest object is created, it is "empty" in the sense that none of its component trees have leaves with values. There are two ways to "initialize" a Forest object. First, the `set_root_leaves()` method simply initializes every tree in the forest to a single node carrying the same (user-specified) leaf value. Second, the `prepare_for_sampler()` method initializes every tree in the forest to a single node with the same value and also propagates this information through to a ForestModel object, which must be synchronized with a Forest during a forest sampler loop.

Usage:

```
Forest$is_empty()
```

Returns: `TRUE` if a Forest has not yet been initialized with a constant root value, `FALSE` otherwise if the forest has already been initialized / grown.

`ForestDataset` *Forest Dataset C++ Wrapper*

Description

Wrapper around a C++ dataset class used to sample a forest. A dataset consists of three matrices / vectors: covariates, bases, and variance weights. Both the basis vector and variance weights are optional.

This class is intended for advanced use cases in which users require detailed control of sampling algorithms and data structures. Minimal input validation and error checks are performed – users are responsible for providing the correct inputs. For tutorials on the "proper" usage of the stochtree's advanced workflow, we provide several vignettes at https://stochtree.ai/

Public fields

`data_ptr` External pointer to a C++ ForestDataset class

Methods

Public methods:

- `ForestDataset$new()`
- `ForestDataset$update_basis()`
- `ForestDataset$update_variance_weights()`
- `ForestDataset$num_observations()`
- `ForestDataset$num_covariates()`
- `ForestDataset$num_basis()`
- `ForestDataset$get_covariates()`
- `ForestDataset$get_basis()`
- `ForestDataset$get_variance_weights()`
- `ForestDataset$has_basis()`
- `ForestDataset$has_variance_weights()`
- `ForestDataset$has_auxiliary_dimension()`
- `ForestDataset$add_auxiliary_dimension()`
- `ForestDataset$get_auxiliary_data_value()`
- `ForestDataset$set_auxiliary_data_value()`
- `ForestDataset$get_auxiliary_data_vector()`

Method `new()`**:** Create a new ForestDataset object.

Usage:

```
ForestDataset$new(covariates, basis = NULL, variance_weights = NULL)
```

Arguments:

`covariates` Matrix of covariates

`basis` (Optional) Matrix of bases used to define a leaf regression

`variance_weights` (Optional) Vector of observation-specific variance weights

Returns: A new `ForestDataset` object.

Method `update_basis()`**:** Update basis matrix in a dataset

Usage:

```
ForestDataset$update_basis(basis)
```

Arguments:

`basis` Updated matrix of bases used to define a leaf regression

Method `update_variance_weights()`**:** Update variance_weights in a dataset

Usage:

```
ForestDataset$update_variance_weights(variance_weights, exponentiate = F)
```

Arguments:

`variance_weights` Updated vector of variance weights used to define individual variance / case weights

`exponentiate` Whether or not input vector should be exponentiated before being written to the Dataset's variance weights. Default: F.

Method `num_observations()`**:** Return number of observations in a `ForestDataset` object

Usage:

```
ForestDataset$num_observations()
```

Returns: Observation count

Method `num_covariates()`**:** Return number of covariates in a `ForestDataset` object

Usage:

```
ForestDataset$num_covariates()
```

Returns: Covariate count

Method `num_basis()`**:** Return number of bases in a `ForestDataset` object

Usage:

```
ForestDataset$num_basis()
```

Returns: Basis count

Method `get_covariates()`**:** Return covariates as an R matrix

Usage:

```
ForestDataset$get_covariates()
```

Returns: Covariate data

Method `get_basis()`**:** Return bases as an R matrix

Usage:

```
ForestDataset$get_basis()
```

Returns: Basis data

Method `get_variance_weights()`**:** Return variance weights as an R vector

Usage:

```
ForestDataset$get_variance_weights()
```

Returns: Variance weight data

Method `has_basis()`**:** Whether or not a dataset has a basis matrix

Usage:

```
ForestDataset$has_basis()
```

Returns: True if basis matrix is loaded, false otherwise

Method `has_variance_weights()`**:** Whether or not a dataset has variance weights

Usage:

```
ForestDataset$has_variance_weights()
```

Returns: True if variance weights are loaded, false otherwise

Method `has_auxiliary_dimension()`**:** Whether or not a dataset has auxiliary data stored at the dimension indicated

Usage:

```
ForestDataset$has_auxiliary_dimension(dim_idx)
```

Arguments:

`dim_idx` Dimension of auxiliary data

Returns: True if auxiliary data has been allocated for `dim_idx` False otherwise

Method `add_auxiliary_dimension()`**:** Initialize a new dimension / lane of auxiliary data and allocate data in its place

Usage:

```
ForestDataset$add_auxiliary_dimension(dim_size)
```

Arguments:

`dim_size` Size of the new vector of data to allocate

Returns: None

Method `get_auxiliary_data_value()`**:** Retrieve auxiliary data value

Usage:

```
ForestDataset$get_auxiliary_data_value(dim_idx, element_idx)
```

Arguments:

`dim_idx` Dimension from which data value to be retrieved

`element_idx` Element to retrieve from dimension `dim_idx`

Returns: Floating point value stored in the requested auxiliary data space

Method `set_auxiliary_data_value()`**:** Set auxiliary data value

Usage:

```
ForestDataset$set_auxiliary_data_value(dim_idx, element_idx, value)
```

Arguments:

`dim_idx` Dimension in which data value to be set

`element_idx` Element to set within dimension `dim_idx`

`value` Data value to set at auxiliary data dimension `dim_idx` and element `element_idx`

Returns: None

Method `get_auxiliary_data_vector()`**:** Retrieve entire auxiliary data vector

Usage:

```
ForestDataset$get_auxiliary_data_vector(dim_idx)
```

Arguments:

`dim_idx` Dimension to retrieve

Returns: Vector of all of the auxiliary data stored at dimension `dim_idx`

ForestKernelComputation

Forest Kernel Computation Routines

Description

Decision tree ensembles can be represented in part by a "kernel" function whose distance metric is based on the extent to which two observations are mapped to the same leaf nodes. This function group offers utilities for evaluating this kernel.

`computeForestLeafIndices` computes and return a vector representation of a forest's leaf predictions for every observation in a dataset. The resulting vector has a "tree-major" format that can be easily re-represented as as a CSR sparse matrix: elements are organized so that the first `n` elements correspond to leaf predictions for all `n` observations in a dataset for the first tree in an ensemble, the next `n` elements correspond to predictions for the second tree and so on. The "data" for each element corresponds to a uniquely mapped column index that corresponds to a single leaf of a single tree (i.e. if tree 1 has 3 leaves, its column indices range from 0 to 2, and then tree 2's leaf indices begin at 3, etc...).

`computeForestLeafVariances` returns each forest's leaf node scale parameters. If leaf scale is not sampled for the forest in question, the function throws an error that the leaf model does not have a stochastic scale parameter.

`computeForestMaxLeafIndex` computes and returns the largest possible leaf index computable by `computeForestLeafIndices` for the forests in a designated forest sample container.

These functions are intended for advanced use cases in which users require detailed control of sampling algorithms and data structures. Minimal input validation and error checks are performed – users are responsible for providing the correct inputs. For tutorials on the "proper" usage of the stochtree's advanced workflow, we provide several vignettes at https://stochtree.ai/

Usage

```
computeForestLeafIndices(
  model_object,
  covariates,
  forest_type = NULL,
  propensity = NULL,
  forest_inds = NULL
)

computeForestLeafVariances(model_object, forest_type, forest_inds = NULL)

computeForestMaxLeafIndex(model_object, forest_type = NULL, forest_inds = NULL)
```

Arguments

`model_object`	Object of type `bartmodel`, `bcfmodel`, or `ForestSamples` corresponding to a BART / BCF model with at least one forest sample, or a low-level `ForestSamples` object.

`covariates` Covariates to use for prediction. Must have the same dimensions / column types as the data used to train a forest.

`forest_type` Which forest to use from `model_object`. Valid inputs depend on the model type, and whether or not a

1. BART

- `'mean'`: Extracts leaf indices for the mean forest
- `'variance'`: Extracts leaf indices for the variance forest

2. BCF

- `'prognostic'`: Extracts leaf indices for the prognostic forest
- `'treatment'`: Extracts leaf indices for the treatment effect forest
- `'variance'`: Extracts leaf indices for the variance forest

3. ForestSamples

- `NULL`: It is not necessary to disambiguate when this function is called directly on a `ForestSamples` object. This is the default value of this

`propensity` (Optional) Propensities used for prediction (BCF-only).

`forest_inds` (Optional) Indices of the forest sample(s) for which to compute max leaf indices. If not provided, this function will return max leaf indices for every sample of a forest. This function uses 0-indexing, so the first forest sample corresponds to `forest_num = 0`, and so on.

Value

`computeForestLeafIndices` returns a vector of size `num_obs * num_trees`, where `num_obs = nrow(covariates)` and `num_trees` is the number of trees in the relevant forest of `model_object`.

`computeForestLeafVariances` returns a vector of size `length(forest_inds)` with the leaf scale parameter for each requested forest.

`computeForestMaxLeafIndex` returns a vector containing the largest possible leaf index computable by `computeForestLeafIndices` for the forests in a designated forest sample container.

Examples

```
X <- matrix(runif(10*100), ncol = 10)
y <- -5 + 10*(X[,1] > 0.5) + rnorm(100)
bart_model <- bart(X, y, num_gfr=0, num_mcmc=10)
leaf_indices <- computeForestLeafIndices(bart_model, X, "mean")
leaf_indices <- computeForestLeafIndices(bart_model, X, "mean", 0)
leaf_indices <- computeForestLeafIndices(bart_model, X, "mean", c(1,3,9))
leaf_variances <- computeForestLeafVariances(bart_model, "mean")
leaf_variances <- computeForestLeafVariances(bart_model, "mean", 0)
leaf_variances <- computeForestLeafVariances(bart_model, "mean", c(1,3,5))
max_leaf_index <- computeForestMaxLeafIndex(bart_model, "mean")
max_leaf_index <- computeForestMaxLeafIndex(bart_model, "mean", 0)
max_leaf_index <- computeForestMaxLeafIndex(bart_model, "mean", c(1,3,9))
```

ForestModel *Forest Model C++ Wrapper*

Description

Wraps the C++ data structures needed to sample an ensemble of decision trees and exposes functionality to run a forest sampler (using either MCMC or the grow-from-root algorithm).

This class is intended for advanced use cases in which users require detailed control of sampling algorithms and data structures. Minimal input validation and error checks are performed – users are responsible for providing the correct inputs. For tutorials on the "proper" usage of the stochtree's advanced workflow, we provide several vignettes at https://stochtree.ai/

Public fields

`tracker_ptr` External pointer to a C++ ForestTracker class

`tree_prior_ptr` External pointer to a C++ TreePrior class

Methods

Public methods:

- `ForestModel$new()`
- `ForestModel$sample_one_iteration()`
- `ForestModel$get_cached_forest_predictions()`
- `ForestModel$propagate_basis_update()`
- `ForestModel$propagate_residual_update()`
- `ForestModel$update_alpha()`
- `ForestModel$update_beta()`
- `ForestModel$update_min_samples_leaf()`
- `ForestModel$update_max_depth()`
- `ForestModel$get_alpha()`
- `ForestModel$get_beta()`
- `ForestModel$get_min_samples_leaf()`
- `ForestModel$get_max_depth()`

Method `new()`**:** Create a new ForestModel object.

Usage:

```
ForestModel$new(
  forest_dataset,
  feature_types,
  num_trees,
  n,
  alpha,
  beta,
  min_samples_leaf,
  max_depth = -1
)
```

Arguments:

`forest_dataset` `ForestDataset` object, used to initialize forest sampling data structures

`feature_types` Feature types (integers where 0 = numeric, 1 = ordered categorical, 2 = unordered categorical)

`num_trees` Number of trees in the forest being sampled

`n` Number of observations in `forest_dataset`

`alpha` Root node split probability in tree prior

`beta` Depth prior penalty in tree prior

`min_samples_leaf` Minimum number of samples in a tree leaf

`max_depth` Maximum depth that any tree can reach

Returns: A new `ForestModel` object.

Method `sample_one_iteration()`**:** Run a single iteration of the forest sampling algorithm (MCMC or GFR)

Usage:

```
ForestModel$sample_one_iteration(
  forest_dataset,
  residual,
  forest_samples,
  active_forest,
  rng,
  forest_model_config,
  global_model_config,
  num_threads = -1,
  keep_forest = TRUE,
  gfr = TRUE
)
```

Arguments:

`forest_dataset` Dataset used to sample the forest

`residual` Outcome used to sample the forest

`forest_samples` Container of forest samples

`active_forest` "Active" forest updated by the sampler in each iteration

`rng` Wrapper around C++ random number generator

`forest_model_config` ForestModelConfig object containing forest model parameters and settings

`global_model_config` GlobalModelConfig object containing global model parameters and settings

`num_threads` Number of threads to use in the GFR and MCMC algorithms, as well as prediction. If OpenMP is not available on a user's system, this will default to 1, otherwise to the maximum number of available threads.

`keep_forest` (Optional) Whether the updated forest sample should be saved to `forest_samples`. Default: `TRUE`.

`gfr` (Optional) Whether or not the forest should be sampled using the "grow-from-root" (GFR) algorithm. Default: `TRUE`.

Method `get_cached_forest_predictions()`: Extract an internally-cached prediction of a forest on the training dataset in a sampler.

Usage:

```
ForestModel$get_cached_forest_predictions()
```

Returns: Vector with as many elements as observations in the training dataset

Method `propagate_basis_update()`: Propagates basis update through to the (full/partial) residual by iteratively (a) adding back in the previous prediction of each tree, (b) recomputing predictions for each tree (caching on the C++ side), (c) subtracting the new predictions from the residual.

This is useful in cases where a basis (for e.g. leaf regression) is updated outside of a tree sampler (as with e.g. adaptive coding for binary treatment BCF). Once a basis has been updated, the overall "function" represented by a tree model has changed and this should be reflected through to the residual before the next sampling loop is run.

Usage:

```
ForestModel$propagate_basis_update(dataset, outcome, active_forest)
```

Arguments:

`dataset` `ForestDataset` object storing the covariates and bases for a given forest

`outcome` `Outcome` object storing the residuals to be updated based on forest predictions

`active_forest` "Active" forest updated by the sampler in each iteration

Method `propagate_residual_update()`: Update the current state of the outcome (i.e. partial residual) data by subtracting the current predictions of each tree. This function is run after the `Outcome` class's `update_data` method, which overwrites the partial residual with an entirely new stream of outcome data.

Usage:

```
ForestModel$propagate_residual_update(residual)
```

Arguments:

`residual` Outcome used to sample the forest

Returns: None

Method `update_alpha()`: Update alpha in the tree prior

Usage:

```
ForestModel$update_alpha(alpha)
```

Arguments:

`alpha` New value of alpha to be used

Returns: None

Method `update_beta()`: Update beta in the tree prior

Usage:

```
ForestModel$update_beta(beta)
```

Arguments:

`beta` New value of beta to be used

Returns: None

Method `update_min_samples_leaf()`**:** Update min_samples_leaf in the tree prior

Usage:

```
ForestModel$update_min_samples_leaf(min_samples_leaf)
```

Arguments:

`min_samples_leaf` New value of min_samples_leaf to be used

Returns: None

Method `update_max_depth()`**:** Update max_depth in the tree prior

Usage:

```
ForestModel$update_max_depth(max_depth)
```

Arguments:

`max_depth` New value of max_depth to be used

Returns: None

Method `get_alpha()`**:** Update alpha in the tree prior

Usage:

```
ForestModel$get_alpha()
```

Returns: Value of alpha in the tree prior

Method `get_beta()`**:** Update beta in the tree prior

Usage:

```
ForestModel$get_beta()
```

Returns: Value of beta in the tree prior

Method `get_min_samples_leaf()`**:** Query min_samples_leaf in the tree prior

Usage:

```
ForestModel$get_min_samples_leaf()
```

Returns: Value of min_samples_leaf in the tree prior

Method `get_max_depth()`**:** Query max_depth in the tree prior

Usage:

```
ForestModel$get_max_depth()
```

Returns: Value of max_depth in the tree prior

`ForestModelConfig` *Forest Model Configuration Object*

Description

Object used to get / set parameters and other model configuration options for a forest model in the "low-level" stochtree interface. The "low-level" stochtree interface enables a high degreee of sampler customization, in which users employ R wrappers around C++ objects like ForestDataset, Outcome, CppRng, and ForestModel to run the Gibbs sampler of a BART model with custom modifications. ForestModelConfig allows users to specify / query the parameters of a forest model they wish to run.

This class is intended for advanced use cases in which users require detailed control of sampling algorithms and data structures. Minimal input validation and error checks are performed – users are responsible for providing the correct inputs. For tutorials on the "proper" usage of the stochtree's advanced workflow, we provide several vignettes at https://stochtree.ai/

Value

Vector of integer-coded feature types (integers where 0 = numeric, 1 = ordered categorical, 2 = unordered categorical)

Vector of (0-indexed) indices of trees to update in a sweep

Vector specifying sampling probability for all p covariates in ForestDataset

Number of trees in a forest

Number of features in a forest model training set

Number of observations in a forest model training set

Root node split probability in tree prior

Depth prior penalty in tree prior

Minimum number of samples in a tree leaf

Maximum depth of any tree in the ensemble in the model

Integer coded leaf model type

Scale parameter used in Gaussian leaf models

Shape parameter for IG leaf models

Scale parameter for IG leaf models

Shape parameter for conditional gamma component of cloglog leaf models

Rate parameter for conditional gamma component of cloglog leaf models

Number of unique cutpoints to consider

Number of features to subsample for the GFR algorithm

Public fields

`feature_types` Vector of integer-coded feature types (integers where 0 = numeric, 1 = ordered categorical, 2 = unordered categorical)

`sweep_update_indices` Vector of trees to update in a sweep

`num_trees` Number of trees in the forest being sampled

`num_features` Number of features in training dataset

`num_observations` Number of observations in training dataset

`leaf_dimension` Dimension of the leaf model

`alpha` Root node split probability in tree prior

`beta` Depth prior penalty in tree prior

`min_samples_leaf` Minimum number of samples in a tree leaf

`max_depth` Maximum depth of any tree in the ensemble in the model. Setting to `-1` does not enforce any depth limits on trees.

`leaf_model_type` Integer specifying the leaf model type (0 = constant leaf, 1 = univariate leaf regression, 2 = multivariate leaf regression)

`leaf_model_scale` Scale parameter used in Gaussian leaf models

`variable_weights` Vector specifying sampling probability for all p covariates in ForestDataset

`variance_forest_shape` Shape parameter for IG leaf models (applicable when `leaf_model_type = 3`)

`variance_forest_scale` Scale parameter for IG leaf models (applicable when `leaf_model_type = 3`)

`cloglog_forest_shape` Shape parameter for conditional gamma component of cloglog leaf models (applicable when `leaf_model_type = 4`)

`cloglog_forest_rate` Rate parameter for conditional gamma component of cloglog leaf models (applicable when `leaf_model_type = 4`)

`cutpoint_grid_size` Number of unique cutpoints to consider

`num_features_subsample` Number of features to subsample for the GFR algorithm Create a new ForestModelConfig object.

Methods

Public methods:

- `ForestModelConfig$new()`
- `ForestModelConfig$update_feature_types()`
- `ForestModelConfig$update_sweep_indices()`
- `ForestModelConfig$update_variable_weights()`
- `ForestModelConfig$update_alpha()`
- `ForestModelConfig$update_beta()`
- `ForestModelConfig$update_min_samples_leaf()`
- `ForestModelConfig$update_max_depth()`
- `ForestModelConfig$update_leaf_model_scale()`

- ForestModelConfig$update_variance_forest_shape()
- ForestModelConfig$update_variance_forest_scale()
- ForestModelConfig$update_cloglog_forest_shape()
- ForestModelConfig$update_cloglog_forest_rate()
- ForestModelConfig$update_cutpoint_grid_size()
- ForestModelConfig$update_num_features_subsample()
- ForestModelConfig$get_feature_types()
- ForestModelConfig$get_sweep_indices()
- ForestModelConfig$get_variable_weights()
- ForestModelConfig$get_num_trees()
- ForestModelConfig$get_num_features()
- ForestModelConfig$get_num_observations()
- ForestModelConfig$get_alpha()
- ForestModelConfig$get_beta()
- ForestModelConfig$get_min_samples_leaf()
- ForestModelConfig$get_max_depth()
- ForestModelConfig$get_leaf_model_type()
- ForestModelConfig$get_leaf_model_scale()
- ForestModelConfig$get_variance_forest_shape()
- ForestModelConfig$get_variance_forest_scale()
- ForestModelConfig$get_cloglog_forest_shape()
- ForestModelConfig$get_cloglog_forest_rate()
- ForestModelConfig$get_cutpoint_grid_size()
- ForestModelConfig$get_num_features_subsample()

Method `new()`**:**

Usage:

```
ForestModelConfig$new(
  feature_types = NULL,
  sweep_update_indices = NULL,
  num_trees = NULL,
  num_features = NULL,
  num_observations = NULL,
  variable_weights = NULL,
  leaf_dimension = 1,
  alpha = 0.95,
  beta = 2,
  min_samples_leaf = 5,
  max_depth = -1,
  leaf_model_type = 1,
  leaf_model_scale = NULL,
  variance_forest_shape = 1,
  variance_forest_scale = 1,
  cloglog_forest_shape = 2,
  cloglog_forest_rate = 2,
```

```
  cutpoint_grid_size = 100,
  num_features_subsample = NULL
)
```

Arguments:

`feature_types` Vector of integer-coded feature types (where 0 = numeric, 1 = ordered categorical, 2 = unordered categorical)

`sweep_update_indices` Vector of (0-indexed) indices of trees to update in a sweep

`num_trees` Number of trees in the forest being sampled

`num_features` Number of features in training dataset

`num_observations` Number of observations in training dataset

`variable_weights` Vector specifying sampling probability for all p covariates in ForestDataset

`leaf_dimension` Dimension of the leaf model (default: `1`)

`alpha` Root node split probability in tree prior (default: `0.95`)

`beta` Depth prior penalty in tree prior (default: `2.0`)

`min_samples_leaf` Minimum number of samples in a tree leaf (default: `5`)

`max_depth` Maximum depth of any tree in the ensemble in the model. Setting to `-1` does not enforce any depth limits on trees. Default: `-1`.

`leaf_model_type` Integer specifying the leaf model type (0 = constant leaf, 1 = univariate leaf regression, 2 = multivariate leaf regression). Default: `0`.

`leaf_model_scale` Scale parameter used in Gaussian leaf models (can either be a scalar or a q x q matrix, where q is the dimensionality of the basis and is only >1 when `leaf_model_int = 2`). Calibrated internally as `1/num_trees`, propagated along diagonal if needed for multivariate leaf models.

`variance_forest_shape` Shape parameter for IG leaf models (applicable when `leaf_model_type = 3`). Default: `1`.

`variance_forest_scale` Scale parameter for IG leaf models (applicable when `leaf_model_type = 3`). Default: `1`.

`cloglog_forest_shape` Shape parameter for conditional gamma component of cloglog leaf models (applicable when `leaf_model_type = 4`). Default: `1`.

`cloglog_forest_rate` Rate parameter for conditional gamma component of cloglog leaf models (applicable when `leaf_model_type = 4`). Default: `1`.

`cutpoint_grid_size` Number of unique cutpoints to consider (default: `100`)

`num_features_subsample` Number of features to subsample for the GFR algorithm

Returns: A new ForestModelConfig object.

Method `update_feature_types()`**:** Update feature types

Usage:

```
ForestModelConfig$update_feature_types(feature_types)
```

Arguments:

`feature_types` Vector of integer-coded feature types (integers where 0 = numeric, 1 = ordered categorical, 2 = unordered categorical)

Method `update_sweep_indices()`**:** Update sweep update indices

Usage:

```
ForestModelConfig$update_sweep_indices(sweep_update_indices)
```

Arguments:

`sweep_update_indices` Vector of (0-indexed) indices of trees to update in a sweep

Method `update_variable_weights()`**:** Update variable weights

Usage:

```
ForestModelConfig$update_variable_weights(variable_weights)
```

Arguments:

`variable_weights` Vector specifying sampling probability for all p covariates in ForestDataset

Method `update_alpha()`**:** Update root node split probability in tree prior

Usage:

```
ForestModelConfig$update_alpha(alpha)
```

Arguments:

`alpha` Root node split probability in tree prior

Method `update_beta()`**:** Update depth prior penalty in tree prior

Usage:

```
ForestModelConfig$update_beta(beta)
```

Arguments:

`beta` Depth prior penalty in tree prior

Method `update_min_samples_leaf()`**:** Update minimum number of samples per leaf node in the tree prior

Usage:

```
ForestModelConfig$update_min_samples_leaf(min_samples_leaf)
```

Arguments:

`min_samples_leaf` Minimum number of samples in a tree leaf

Method `update_max_depth()`**:** Update max depth in the tree prior

Usage:

```
ForestModelConfig$update_max_depth(max_depth)
```

Arguments:

`max_depth` Maximum depth of any tree in the ensemble in the model

Method `update_leaf_model_scale()`**:** Update scale parameter used in Gaussian leaf models

Usage:

```
ForestModelConfig$update_leaf_model_scale(leaf_model_scale)
```

Arguments:

`leaf_model_scale` Scale parameter used in Gaussian leaf models

Method `update_variance_forest_shape()`**:** Update shape parameter for IG leaf models

Usage:

```
ForestModelConfig$update_variance_forest_shape(variance_forest_shape)
```

Arguments:

`variance_forest_shape` Shape parameter for IG leaf models

Method `update_variance_forest_scale()`**:** Update scale parameter for IG leaf models

Usage:

```
ForestModelConfig$update_variance_forest_scale(variance_forest_scale)
```

Arguments:

`variance_forest_scale` Scale parameter for IG leaf models

Method `update_cloglog_forest_shape()`**:** Update shape parameter for conditional gamma component of cloglog leaf models

Usage:

```
ForestModelConfig$update_cloglog_forest_shape(cloglog_forest_shape)
```

Arguments:

`cloglog_forest_shape` Shape parameter for conditional gamma component of cloglog leaf models

Method `update_cloglog_forest_rate()`**:** Update rate parameter for conditional gamma component of cloglog leaf models

Usage:

```
ForestModelConfig$update_cloglog_forest_rate(cloglog_forest_rate)
```

Arguments:

`cloglog_forest_rate` Rate parameter for conditional gamma component of cloglog leaf models

Method `update_cutpoint_grid_size()`**:** Update number of unique cutpoints to consider

Usage:

```
ForestModelConfig$update_cutpoint_grid_size(cutpoint_grid_size)
```

Arguments:

`cutpoint_grid_size` Number of unique cutpoints to consider

Method `update_num_features_subsample()`**:** Update number of features to subsample for the GFR algorithm

Usage:

```
ForestModelConfig$update_num_features_subsample(num_features_subsample)
```

Arguments:

`num_features_subsample` Number of features to subsample for the GFR algorithm

Method `get_feature_types()`**:** Query feature types for this ForestModelConfig object

Usage:

```
ForestModelConfig$get_feature_types()
```

Method `get_sweep_indices()`**:** Query sweep update indices for this ForestModelConfig object

Usage:

```
ForestModelConfig$get_sweep_indices()
```

Method `get_variable_weights()`**:** Query variable weights for this ForestModelConfig object

Usage:

```
ForestModelConfig$get_variable_weights()
```

Method `get_num_trees()`**:** Query number of trees

Usage:

```
ForestModelConfig$get_num_trees()
```

Method `get_num_features()`**:** Query number of features

Usage:

```
ForestModelConfig$get_num_features()
```

Method `get_num_observations()`**:** Query number of observations

Usage:

```
ForestModelConfig$get_num_observations()
```

Method `get_alpha()`**:** Query root node split probability in tree prior for this ForestModelConfig object

Usage:

```
ForestModelConfig$get_alpha()
```

Method `get_beta()`**:** Query depth prior penalty in tree prior for this ForestModelConfig object

Usage:

```
ForestModelConfig$get_beta()
```

Method `get_min_samples_leaf()`**:** Query root node split probability in tree prior for this ForestModelConfig object

Usage:

```
ForestModelConfig$get_min_samples_leaf()
```

Method `get_max_depth()`**:** Query root node split probability in tree prior for this ForestModelConfig object

Usage:

```
ForestModelConfig$get_max_depth()
```

Method `get_leaf_model_type()`**:** Query (integer-coded) type of leaf model

Usage:

```
ForestModelConfig$get_leaf_model_type()
```

Method `get_leaf_model_scale()`**:** Query scale parameter used in Gaussian leaf models for this ForestModelConfig object

Usage:

```
ForestModelConfig$get_leaf_model_scale()
```

Method `get_variance_forest_shape()`: Query shape parameter for IG leaf models for this ForestModelConfig object

Usage:

```
ForestModelConfig$get_variance_forest_shape()
```

Method `get_variance_forest_scale()`: Query scale parameter for IG leaf models for this ForestModelConfig object

Usage:

```
ForestModelConfig$get_variance_forest_scale()
```

Method `get_cloglog_forest_shape()`: Query shape parameter for conditional gamma component of cloglog leaf models for this ForestModelConfig object

Usage:

```
ForestModelConfig$get_cloglog_forest_shape()
```

Method `get_cloglog_forest_rate()`: Query rate parameter for conditional gamma component of cloglog leaf models for this ForestModelConfig object

Usage:

```
ForestModelConfig$get_cloglog_forest_rate()
```

Method `get_cutpoint_grid_size()`: Query number of unique cutpoints to consider for this ForestModelConfig object

Usage:

```
ForestModelConfig$get_cutpoint_grid_size()
```

Method `get_num_features_subsample()`: Query number of features to subsample for the GFR algorithm

Usage:

```
ForestModelConfig$get_num_features_subsample()
```

`ForestSamples` *Forest Container C++ Wrapper*

Description

Wrapper around a C++ class that stores draws from an random ensemble of decision trees.

This class is intended for advanced use cases in which users require detailed control of sampling algorithms and data structures. Minimal input validation and error checks are performed – users are responsible for providing the correct inputs. For tutorials on the "proper" usage of the stochtree's advanced workflow, we provide several vignettes at https://stochtree.ai/

Public fields

`forest_container_ptr` External pointer to a C++ ForestContainer class

Methods

Public methods:

- ForestSamples$new()
- ForestSamples$collapse()
- ForestSamples$combine_forests()
- ForestSamples$add_to_forest()
- ForestSamples$multiply_forest()
- ForestSamples$load_from_json()
- ForestSamples$append_from_json()
- ForestSamples$load_from_json_string()
- ForestSamples$append_from_json_string()
- ForestSamples$predict()
- ForestSamples$predict_raw()
- ForestSamples$predict_raw_single_forest()
- ForestSamples$predict_raw_single_tree()
- ForestSamples$set_root_leaves()
- ForestSamples$prepare_for_sampler()
- ForestSamples$adjust_residual()
- ForestSamples$save_json()
- ForestSamples$load_json()
- ForestSamples$num_samples()
- ForestSamples$num_trees()
- ForestSamples$leaf_dimension()
- ForestSamples$is_leaf_constant()
- ForestSamples$is_exponentiated()
- ForestSamples$add_forest_with_constant_leaves()
- ForestSamples$add_numeric_split_tree()
- ForestSamples$get_tree_leaves()
- ForestSamples$get_tree_split_counts()
- ForestSamples$get_forest_split_counts()
- ForestSamples$get_aggregate_split_counts()
- ForestSamples$get_granular_split_counts()
- ForestSamples$ensemble_tree_max_depth()
- ForestSamples$average_ensemble_max_depth()
- ForestSamples$average_max_depth()
- ForestSamples$num_forest_leaves()
- ForestSamples$sum_leaves_squared()
- ForestSamples$is_leaf_node()
- ForestSamples$is_numeric_split_node()
- ForestSamples$is_categorical_split_node()
- ForestSamples$parent_node()
- ForestSamples$left_child_node()

- ForestSamples$right_child_node()
- ForestSamples$node_depth()
- ForestSamples$node_split_index()
- ForestSamples$node_split_threshold()
- ForestSamples$node_split_categories()
- ForestSamples$node_leaf_values()
- ForestSamples$num_nodes()
- ForestSamples$num_leaves()
- ForestSamples$num_leaf_parents()
- ForestSamples$num_split_nodes()
- ForestSamples$nodes()
- ForestSamples$leaves()
- ForestSamples$delete_sample()

Method `new()`**:** Create a new ForestContainer object.

Usage:

```
ForestSamples$new(
  num_trees,
  leaf_dimension = 1,
  is_leaf_constant = FALSE,
  is_exponentiated = FALSE
)
```

Arguments:

`num_trees` Number of trees

`leaf_dimension` Dimensionality of the outcome model

`is_leaf_constant` Whether leaf is constant

`is_exponentiated` Whether forest predictions should be exponentiated before being returned

Returns: A new `ForestContainer` object.

Method `collapse()`**:** Collapse forests in this container by a pre-specified batch size. For example, if we have a container of twenty 10-tree forests, and we specify a `batch_size` of 5, then this method will yield four 50-tree forests. "Excess" forests remaining after the size of a forest container is divided by `batch_size` will be pruned from the beginning of the container (i.e. earlier sampled forests will be deleted). This method has no effect if `batch_size` is larger than the number of forests in a container.

Usage:

```
ForestSamples$collapse(batch_size)
```

Arguments:

`batch_size` Number of forests to be collapsed into a single forest

Method `combine_forests()`**:** Merge specified forests into a single forest

Usage:

```
ForestSamples$combine_forests(forest_inds)
```

Arguments:

`forest_inds` Indices of forests to be combined (0-indexed)

Method `add_to_forest()`**:** Add a constant value to every leaf of every tree of a given forest

Usage:

```
ForestSamples$add_to_forest(forest_index, constant_value)
```

Arguments:

`forest_index` Index of forest whose leaves will be modified (0-indexed)

`constant_value` Value to add to every leaf of every tree of the forest at `forest_index`

Method `multiply_forest()`**:** Multiply every leaf of every tree of a given forest by constant value

Usage:

```
ForestSamples$multiply_forest(forest_index, constant_multiple)
```

Arguments:

`forest_index` Index of forest whose leaves will be modified (0-indexed)

`constant_multiple` Value to multiply through by every leaf of every tree of the forest at `forest_index`

Method `load_from_json()`**:** Create a new `ForestContainer` object from a json object

Usage:

```
ForestSamples$load_from_json(json_object, json_forest_label)
```

Arguments:

`json_object` Object of class `CppJson`

`json_forest_label` Label referring to a particular forest (i.e. "forest_0") in the overall json hierarchy

Returns: A new `ForestContainer` object.

Method `append_from_json()`**:** Append to a `ForestContainer` object from a json object

Usage:

```
ForestSamples$append_from_json(json_object, json_forest_label)
```

Arguments:

`json_object` Object of class `CppJson`

`json_forest_label` Label referring to a particular forest (i.e. "forest_0") in the overall json hierarchy

Returns: None

Method `load_from_json_string()`**:** Create a new `ForestContainer` object from a json object

Usage:

```
ForestSamples$load_from_json_string(json_string, json_forest_label)
```

Arguments:

`json_string` JSON string which parses into object of class `CppJson`

`json_forest_label` Label referring to a particular forest (i.e. "forest_0") in the overall json hierarchy

Returns: A new `ForestContainer` object.

Method `append_from_json_string()`**:** Append to a `ForestContainer` object from a json object

Usage:

```
ForestSamples$append_from_json_string(json_string, json_forest_label)
```

Arguments:

`json_string` JSON string which parses into object of class `CppJson`

`json_forest_label` Label referring to a particular forest (i.e. "forest_0") in the overall json hierarchy

Returns: None

Method `predict()`**:** Predict every tree ensemble on every sample in `forest_dataset`

Usage:

```
ForestSamples$predict(forest_dataset)
```

Arguments:

`forest_dataset` `ForestDataset` R class

Returns: matrix of predictions with as many rows as in forest_dataset and as many columns as samples in the `ForestContainer`

Method `predict_raw()`**:** Predict "raw" leaf values (without being multiplied by basis) for every tree ensemble on every sample in `forest_dataset`

Usage:

```
ForestSamples$predict_raw(forest_dataset)
```

Arguments:

`forest_dataset` `ForestDataset` R class

Returns: Array of predictions for each observation in `forest_dataset` and each sample in the `ForestSamples` class with each prediction having the dimensionality of the forests' leaf model. In the case of a constant leaf model or univariate leaf regression, this array is two-dimensional (number of observations, number of forest samples). In the case of a multivariate leaf regression, this array is three-dimension (number of observations, leaf model dimension, number of samples).

Method `predict_raw_single_forest()`**:** Predict "raw" leaf values (without being multiplied by basis) for a specific forest on every sample in `forest_dataset`

Usage:

```
ForestSamples$predict_raw_single_forest(forest_dataset, forest_num)
```

Arguments:

`forest_dataset` `ForestDataset` R class

`forest_num` Index of the forest sample within the container

Returns: matrix of predictions with as many rows as in forest_dataset and as many columns as dimensions in the leaves of trees in `ForestContainer`

Method `predict_raw_single_tree()`**:** Predict "raw" leaf values (without being multiplied by basis) for a specific tree in a specific forest on every observation in `forest_dataset`

Usage:

```
ForestSamples$predict_raw_single_tree(forest_dataset, forest_num, tree_num)
```

Arguments:

`forest_dataset` `ForestDataset` R class

`forest_num` Index of the forest sample within the container

`tree_num` Index of the tree to be queried

Returns: matrix of predictions with as many rows as in `forest_dataset` and as many columns as dimensions in the leaves of trees in `ForestContainer`

Method `set_root_leaves()`**:** Set a constant predicted value for every tree in the ensemble. Stops program if any tree is more than a root node.

Usage:

```
ForestSamples$set_root_leaves(forest_num, leaf_value)
```

Arguments:

`forest_num` Index of the forest sample within the container.

`leaf_value` Constant leaf value(s) to be fixed for each tree in the ensemble indexed by `forest_num`. Can be either a single number or a vector, depending on the forest's leaf dimension.

Method `prepare_for_sampler()`**:** Set a constant predicted value for every tree in the ensemble. Stops program if any tree is more than a root node.

Usage:

```
ForestSamples$prepare_for_sampler(
  dataset,
  outcome,
  forest_model,
  leaf_model_int,
  leaf_value
)
```

Arguments:

`dataset` `ForestDataset` Dataset class (covariates, basis, etc...)

`outcome` `Outcome` Outcome class (residual / partial residual)

`forest_model` `ForestModel` object storing tracking structures used in training / sampling

`leaf_model_int` Integer value encoding the leaf model type (0 = constant gaussian, 1 = univariate gaussian, 2 = multivariate gaussian, 3 = log linear variance).

`leaf_value` Constant leaf value(s) to be fixed for each tree in the ensemble indexed by `forest_num`. Can be either a single number or a vector, depending on the forest's leaf dimension.

Method `adjust_residual()`**:** Adjusts residual based on the predictions of a forest

This is typically run just once at the beginning of a forest sampling algorithm. After trees are initialized with constant root node predictions, their root predictions are subtracted out of the residual.

Usage:

```
ForestSamples$adjust_residual(
  dataset,
  outcome,
  forest_model,
  requires_basis,
  forest_num,
  add
)
```

Arguments:

`dataset` `ForestDataset` object storing the covariates and bases for a given forest

`outcome` `Outcome` object storing the residuals to be updated based on forest predictions

`forest_model` `ForestModel` object storing tracking structures used in training / sampling

`requires_basis` Whether or not a forest requires a basis for prediction

`forest_num` Index of forest used to update residuals

`add` Whether forest predictions should be added to or subtracted from residuals

Method `save_json()`**:** Store the trees and metadata of `ForestDataset` class in a json file

Usage:

```
ForestSamples$save_json(json_filename)
```

Arguments:

`json_filename` Name of output json file (must end in ".json")

Method `load_json()`**:** Load trees and metadata for an ensemble from a json file. Note that any trees and metadata already present in `ForestDataset` class will be overwritten.

Usage:

```
ForestSamples$load_json(json_filename)
```

Arguments:

`json_filename` Name of model input json file (must end in ".json")

Method `num_samples()`**:** Return number of samples in a `ForestContainer` object

Usage:

```
ForestSamples$num_samples()
```

Returns: Sample count

Method `num_trees()`**:** Return number of trees in each ensemble of a `ForestContainer` object

Usage:

```
ForestSamples$num_trees()
```

Returns: Tree count

Method `leaf_dimension()`**:** Return output dimension of trees in a `ForestContainer` object

Usage:

```
ForestSamples$leaf_dimension()
```

Returns: Leaf node parameter size

Method `is_leaf_constant()`**:** Return constant leaf status of trees in a `ForestContainer` object

Usage:

```
ForestSamples$is_leaf_constant()
```

Returns: `TRUE` if leaves are constant, `FALSE` otherwise

Method `is_exponentiated()`**:** Return exponentiation status of trees in a `ForestContainer` object

Usage:

```
ForestSamples$is_exponentiated()
```

Returns: `TRUE` if leaf predictions must be exponentiated, `FALSE` otherwise

Method `add_forest_with_constant_leaves()`**:** Add a new all-root ensemble to the container, with all of the leaves set to the value / vector provided

Usage:

```
ForestSamples$add_forest_with_constant_leaves(leaf_value)
```

Arguments:

`leaf_value` Value (or vector of values) to initialize root nodes in tree

Method `add_numeric_split_tree()`**:** Add a numeric (i.e. `X[,i] <= c`) split to a given tree in the ensemble

Usage:

```
ForestSamples$add_numeric_split_tree(
  forest_num,
  tree_num,
  leaf_num,
  feature_num,
  split_threshold,
  left_leaf_value,
  right_leaf_value
)
```

Arguments:

`forest_num` Index of the forest which contains the tree to be split

`tree_num` Index of the tree to be split

`leaf_num` Leaf to be split

`feature_num` Feature that defines the new split

`split_threshold` Value that defines the cutoff of the new split

`left_leaf_value` Value (or vector of values) to assign to the newly created left node

`right_leaf_value` Value (or vector of values) to assign to the newly created right node

Method `get_tree_leaves()`**:** Retrieve a vector of indices of leaf nodes for a given tree in a given forest

Usage:

```
ForestSamples$get_tree_leaves(forest_num, tree_num)
```

Arguments:

`forest_num` Index of the forest which contains tree `tree_num`

`tree_num` Index of the tree for which leaf indices will be retrieved

Method `get_tree_split_counts()`**:** Retrieve a vector of split counts for every training set variable in a given tree in a given forest

Usage:

```
ForestSamples$get_tree_split_counts(forest_num, tree_num, num_features)
```

Arguments:

`forest_num` Index of the forest which contains tree `tree_num`

`tree_num` Index of the tree for which split counts will be retrieved

`num_features` Total number of features in the training set

Method `get_forest_split_counts()`**:** Retrieve a vector of split counts for every training set variable in a given forest

Usage:

```
ForestSamples$get_forest_split_counts(forest_num, num_features)
```

Arguments:

`forest_num` Index of the forest for which split counts will be retrieved

`num_features` Total number of features in the training set

Method `get_aggregate_split_counts()`**:** Retrieve a vector of split counts for every training set variable in a given forest, aggregated across ensembles and trees

Usage:

```
ForestSamples$get_aggregate_split_counts(num_features)
```

Arguments:

`num_features` Total number of features in the training set

Method `get_granular_split_counts()`**:** Retrieve a vector of split counts for every training set variable in a given forest, reported separately for each ensemble and tree

Usage:

```
ForestSamples$get_granular_split_counts(num_features)
```

Arguments:

`num_features` Total number of features in the training set

Method `ensemble_tree_max_depth()`**:** Maximum depth of a specific tree in a specific ensemble in a `ForestSamples` object

Usage:

```
ForestSamples$ensemble_tree_max_depth(ensemble_num, tree_num)
```

Arguments:

`ensemble_num` Ensemble number

`tree_num` Tree index within ensemble `ensemble_num`

Returns: Maximum leaf depth

Method `average_ensemble_max_depth()`**:** Average the maximum depth of each tree in a given ensemble in a `ForestSamples` object

Usage:

```
ForestSamples$average_ensemble_max_depth(ensemble_num)
```

Arguments:

`ensemble_num` Ensemble number

Returns: Average maximum depth

Method `average_max_depth()`**:** Average the maximum depth of each tree in each ensemble in a `ForestContainer` object

Usage:

```
ForestSamples$average_max_depth()
```

Returns: Average maximum depth

Method `num_forest_leaves()`**:** Number of leaves in a given ensemble in a `ForestSamples` object

Usage:

```
ForestSamples$num_forest_leaves(forest_num)
```

Arguments:

`forest_num` Index of the ensemble to be queried

Returns: Count of leaves in the ensemble stored at `forest_num`

Method `sum_leaves_squared()`**:** Sum of squared (raw) leaf values in a given ensemble in a `ForestSamples` object

Usage:

```
ForestSamples$sum_leaves_squared(forest_num)
```

Arguments:

`forest_num` Index of the ensemble to be queried

Returns: Average maximum depth

Method `is_leaf_node()`**:** Whether or not a given node of a given tree in a given forest in the `ForestSamples` is a leaf

Usage:

```
ForestSamples$is_leaf_node(forest_num, tree_num, node_id)
```

Arguments:

`forest_num` Index of the forest to be queried

`tree_num` Index of the tree to be queried

`node_id` Index of the node to be queried

Returns: `TRUE` if node is a leaf, `FALSE` otherwise

Method `is_numeric_split_node()`**:** Whether or not a given node of a given tree in a given forest in the `ForestSamples` is a numeric split node

Usage:

```
ForestSamples$is_numeric_split_node(forest_num, tree_num, node_id)
```

Arguments:

`forest_num` Index of the forest to be queried

`tree_num` Index of the tree to be queried

`node_id` Index of the node to be queried

Returns: `TRUE` if node is a numeric split node, `FALSE` otherwise

Method `is_categorical_split_node()`**:** Whether or not a given node of a given tree in a given forest in the `ForestSamples` is a categorical split node

Usage:

```
ForestSamples$is_categorical_split_node(forest_num, tree_num, node_id)
```

Arguments:

`forest_num` Index of the forest to be queried

`tree_num` Index of the tree to be queried

`node_id` Index of the node to be queried

Returns: `TRUE` if node is a categorical split node, `FALSE` otherwise

Method `parent_node()`**:** Parent node of given node of a given tree in a given forest in a `ForestSamples` object

Usage:

```
ForestSamples$parent_node(forest_num, tree_num, node_id)
```

Arguments:

`forest_num` Index of the forest to be queried

`tree_num` Index of the tree to be queried

`node_id` Index of the node to be queried

Returns: Integer ID of the parent node

Method `left_child_node()`**:** Left child node of given node of a given tree in a given forest in a `ForestSamples` object

Usage:

```
ForestSamples$left_child_node(forest_num, tree_num, node_id)
```

Arguments:

`forest_num` Index of the forest to be queried

`tree_num` Index of the tree to be queried

`node_id` Index of the node to be queried

Returns: Integer ID of the left child node

Method `right_child_node()`**:** Right child node of given node of a given tree in a given forest in a `ForestSamples` object

Usage:

```
ForestSamples$right_child_node(forest_num, tree_num, node_id)
```

Arguments:

`forest_num` Index of the forest to be queried

`tree_num` Index of the tree to be queried

`node_id` Index of the node to be queried

Returns: Integer ID of the right child node

Method `node_depth()`**:** Depth of given node of a given tree in a given forest in a `ForestSamples` object, with 0 depth for the root node.

Usage:

```
ForestSamples$node_depth(forest_num, tree_num, node_id)
```

Arguments:

`forest_num` Index of the forest to be queried

`tree_num` Index of the tree to be queried

`node_id` Index of the node to be queried

Returns: Integer valued depth of the node

Method `node_split_index()`**:** Split index of given node of a given tree in a given forest in a `ForestSamples` object. Returns `-1` is node is a leaf.

Usage:

```
ForestSamples$node_split_index(forest_num, tree_num, node_id)
```

Arguments:

`forest_num` Index of the forest to be queried

`tree_num` Index of the tree to be queried

`node_id` Index of the node to be queried

Returns: Integer valued depth of the node

Method `node_split_threshold()`**:** Threshold that defines a numeric split for a given node of a given tree in a given forest in a `ForestSamples` object. Returns `Inf` if the node is a leaf or a categorical split node.

Usage:

```
ForestSamples$node_split_threshold(forest_num, tree_num, node_id)
```

Arguments:

`forest_num` Index of the forest to be queried

`tree_num` Index of the tree to be queried

`node_id` Index of the node to be queried

Returns: Threshold defining a split for the node

Method `node_split_categories()`**:** Array of category indices that define a categorical split for a given node of a given tree in a given forest in a `ForestSamples` object. Returns `c(Inf)` if the node is a leaf or a numeric split node.

Usage:

```
ForestSamples$node_split_categories(forest_num, tree_num, node_id)
```

Arguments:

`forest_num` Index of the forest to be queried

`tree_num` Index of the tree to be queried

`node_id` Index of the node to be queried

Returns: Categories defining a split for the node

Method `node_leaf_values()`**:** Leaf node value(s) for a given node of a given tree in a given forest in a `ForestSamples` object. Values are stale if the node is a split node.

Usage:

```
ForestSamples$node_leaf_values(forest_num, tree_num, node_id)
```

Arguments:

`forest_num` Index of the forest to be queried

`tree_num` Index of the tree to be queried

`node_id` Index of the node to be queried

Returns: Vector (often univariate) of leaf values

Method `num_nodes()`**:** Number of nodes in a given tree in a given forest in a `ForestSamples` object.

Usage:

```
ForestSamples$num_nodes(forest_num, tree_num)
```

Arguments:

`forest_num` Index of the forest to be queried

`tree_num` Index of the tree to be queried

Returns: Count of total tree nodes

Method `num_leaves()`**:** Number of leaves in a given tree in a given forest in a `ForestSamples` object.

Usage:

```
ForestSamples$num_leaves(forest_num, tree_num)
```

Arguments:

`forest_num` Index of the forest to be queried

`tree_num` Index of the tree to be queried

Returns: Count of total tree leaves

Method `num_leaf_parents()`**:** Number of leaf parents (split nodes with two leaves as children) in a given tree in a given forest in a `ForestSamples` object.

Usage:

```
ForestSamples$num_leaf_parents(forest_num, tree_num)
```

Arguments:

`forest_num` Index of the forest to be queried

`tree_num` Index of the tree to be queried

Returns: Count of total tree leaf parents

Method `num_split_nodes()`**:** Number of split nodes in a given tree in a given forest in a `ForestSamples` object.

Usage:

```
ForestSamples$num_split_nodes(forest_num, tree_num)
```

Arguments:

`forest_num` Index of the forest to be queried

`tree_num` Index of the tree to be queried

Returns: Count of total tree split nodes

Method `nodes()`**:** Array of node indices in a given tree in a given forest in a `ForestSamples` object.

Usage:

```
ForestSamples$nodes(forest_num, tree_num)
```

Arguments:

`forest_num` Index of the forest to be queried

`tree_num` Index of the tree to be queried

Returns: Indices of tree nodes

Method `leaves()`**:** Array of leaf indices in a given tree in a given forest in a `ForestSamples` object.

Usage:

```
ForestSamples$leaves(forest_num, tree_num)
```

Arguments:

`forest_num` Index of the forest to be queried

`tree_num` Index of the tree to be queried

Returns: Indices of leaf nodes

Method `delete_sample()`**:** Modify the `ForestSamples` object by removing the forest sample indexed by ‘forest_num

Usage:

```
ForestSamples$delete_sample(forest_num)
```

Arguments:

`forest_num` Index of the forest to be removed

ForestSamplesSerialization

Forest Container Serialization Routines

Description

While the BARTSerialization and BCFSerialization topics focus on JSON serialization / deserialization for entire `bartmodel` and `bcfmodel` objects, this function group provides an interface for a more focused use case: loading a single ForestSamples container from a broader BART / BCF model (which may include multiple forests and other parametric terms).

`loadForestContainerJson` converts a CppJson object representing a BART or BCF model into a ForestSamples container by extracting the JSON indexed by a forest label (i.e. `"forest_0"`) and deserializing it into a ForestSamples object.

Both `loadForestContainerJson` and `loadForestContainerCombinedJson` operate similarly, but on a list of CppJson or JSON string representations of BART / BCF models with the same structure.

These functions are intended for advanced use cases in which users require detailed control of sampling algorithms and data structures. Minimal input validation and error checks are performed – users are responsible for providing the correct inputs. For tutorials on the "proper" usage of the stochtree's advanced workflow, we provide several vignettes at https://stochtree.ai/

Usage

```
loadForestContainerJson(json_object, json_forest_label)

loadForestContainerCombinedJson(json_object_list, json_forest_label)

loadForestContainerCombinedJsonString(json_string_list, json_forest_label)
```

Arguments

`json_object`	Object of class `CppJson`
`json_forest_label`	Label referring to a particular forest (i.e. "forest_0") in the overall json hierarchy (must exist in every json object in a list if a list is provided)
`json_object_list`	List of objects of class `CppJson`
`json_string_list`	List of strings that parse into objects of type `CppJson`

Value

Each of the functions in this group returns a `ForestSamples` object.

Examples

```
X <- matrix(runif(10*100), ncol = 10)
y <- -5 + 10*(X[,1] > 0.5) + rnorm(100)
bart_model <- bart(X, y, num_gfr=0, num_mcmc=10)
bart_json <- saveBARTModelToJson(bart_model)
bart_json_string <- saveBARTModelToJsonString(bart_model)
bart_json_list <- list(bart_json)
bart_json_string_list <- list(bart_json_string)
mean_forest <- loadForestContainerJson(bart_json, "forest_0")
mean_forest <- loadForestContainerCombinedJson(bart_json_list, "forest_0")
mean_forest <- loadForestContainerCombinedJsonString(bart_json_string_list, "forest_0")
```

ForestStateManagement *Forest State Management Routines*

Description

A forest sampler features two types of state: ephemeral and persistent. Persistent state includes objects like ForestSamples and RandomEffectSamples which constitute part of the final sampled model. Ephemeral state supports the sampling computations, but is not retained after the sampler finishes.

The two primary forest-based bits of ephemeral state are the Forest and ForestModel classes, which represent the current state of a forest and its corresponding tracking data structures.

In a linear sampling loop, this ephemeral state is updated with each iteration of the sampler and any retained forests are copied to a ForestSamples object. However, in multi-chain settings, the state of a forest must typically be "reset" at the beginning of a new chain. These function enable this process by synchronizing the state of a Forest and ForestModel with a corresponding element of a ForestSamples object, or by resetting both to their default (root) state.

`resetActiveForest` resets a Forest object, either from a specific forest in a ForestSamples object or to an ensemble of single-node (i.e. root) trees. `resetForestModel` re-initializes a forest model (tracking data structures) from a specific forest in a ForestSamples object.

These functions are intended for advanced use cases in which users require detailed control of sampling algorithms and data structures. Minimal input validation and error checks are performed – users are responsible for providing the correct inputs. For tutorials on the "proper" usage of the stochtree's advanced workflow, we provide several vignettes at https://stochtree.ai/

Usage

```
resetActiveForest(active_forest, forest_samples = NULL, forest_num = NULL)

resetForestModel(forest_model, forest, dataset, residual, is_mean_model)
```

Arguments

active_forest	Current active forest
forest_samples	(Optional) Container of forest samples from which to re-initialize active forest. If not provided, active forest will be reset to an ensemble of single-node (i.e. root) trees.
forest_num	(Optional) Index of forest samples from which to initialize active forest. If not provided, active forest will be reset to an ensemble of single-node (i.e. root) trees.
forest_model	Forest model with tracking data structures
forest	Forest from which to re-initialize forest model
dataset	Training dataset object
residual	Residual which will also be updated
is_mean_model	Whether the model being updated is a conditional mean model

Value

Both functions have no return type and operate in-place on the relevant Forest or ForestModel objects

Examples

```
n <- 100
p <- 10
num_trees <- 100
leaf_dimension <- 1
is_leaf_constant <- TRUE
is_exponentiated <- FALSE
alpha <- 0.95
beta <- 2.0
min_samples_leaf <- 2
max_depth <- 10
feature_types <- as.integer(rep(0, p))
leaf_model <- 0
sigma2 <- 1.0
leaf_scale <- as.matrix(1.0)
variable_weights <- rep(1/p, p)
a_forest <- 1
b_forest <- 1
cutpoint_grid_size <- 100
X <- matrix(runif(n*p), ncol = p)
forest_dataset <- createForestDataset(X)
y <- -5 + 10*(X[,1] > 0.5) + rnorm(n)
outcome <- createOutcome(y)
rng <- createCppRNG(1234)
global_model_config <- createGlobalModelConfig(global_error_variance=sigma2)
forest_model_config <- createForestModelConfig(feature_types=feature_types,
                                                num_trees=num_trees, num_observations=n,
                                                num_features=p, alpha=alpha, beta=beta,
                                                min_samples_leaf=min_samples_leaf,
```

```
                                                max_depth=max_depth,
                                                variable_weights=variable_weights,
                                                cutpoint_grid_size=cutpoint_grid_size,
                                                leaf_model_type=leaf_model,
                                                leaf_model_scale=leaf_scale)
forest_model <- createForestModel(forest_dataset, forest_model_config, global_model_config)
active_forest <- createForest(num_trees, leaf_dimension, is_leaf_constant, is_exponentiated)
forest_samples <- createForestSamples(num_trees, leaf_dimension,
                                      is_leaf_constant, is_exponentiated)
active_forest$prepare_for_sampler(forest_dataset, outcome, forest_model, 0, 0.)
forest_model$sample_one_iteration(
    forest_dataset, outcome, forest_samples, active_forest,
    rng, forest_model_config, global_model_config,
    keep_forest = TRUE, gfr = FALSE
)
resetActiveForest(active_forest, forest_samples, 0)
resetForestModel(forest_model, active_forest, forest_dataset, outcome, TRUE)
```

`getRandomEffectSamples` *Extract Random Effect Samples Generic Function*

Description

Generic function for extracting random effect samples from a model object (BCF, BART, etc...)

Usage

```
getRandomEffectSamples(object, ...)
```

Arguments

`object`	Fitted model object from which to extract random effects
`...`	Other parameters to be used in random effects extraction

Value

List of random effect samples

Examples

```
n <- 100
p <- 10
X <- matrix(runif(n*p), ncol = p)
rfx_group_ids <- sample(1:2, size = n, replace = TRUE)
rfx_basis <- rep(1.0, n)
y <- (-5 + 10*(X[,1] > 0.5)) + (-2*(rfx_group_ids==1)+2*(rfx_group_ids==2)) + rnorm(n)
bart_model <- bart(X_train=X, y_train=y, rfx_group_ids_train=rfx_group_ids,
                   rfx_basis_train = rfx_basis, num_gfr=0, num_mcmc=10)
rfx_samples <- getRandomEffectSamples(bart_model)
```

`getRandomEffectSamples.bartmodel` *Extract Random Effects Samples from BART Model*

Description

Extract raw sample values for each of the random effect parameter terms.

Usage

```
## S3 method for class 'bartmodel'
getRandomEffectSamples(object, ...)
```

Arguments

`object`	Object of type `bartmodel` containing draws of a BART model and associated sampling outputs.
`...`	Other parameters to be used in random effects extraction

Value

List of arrays. The alpha array has dimension (`num_components`, `num_samples`) and is simply a vector if `num_components = 1`. The xi and beta arrays have dimension (`num_components`, `num_groups`, `num_samples`) and is simply a matrix if `num_components = 1`. The sigma array has dimension (`num_components`, `num_samples`) and is simply a vector if `num_components = 1`.

Examples

```
n <- 100
p <- 5
X <- matrix(runif(n*p), ncol = p)
f_XW <- (
    ((0 <= X[,1]) & (0.25 > X[,1])) * (-7.5) +
    ((0.25 <= X[,1]) & (0.5 > X[,1])) * (-2.5) +
    ((0.5 <= X[,1]) & (0.75 > X[,1])) * (2.5) +
    ((0.75 <= X[,1]) & (1 > X[,1])) * (7.5)
)
snr <- 3
group_ids <- rep(c(1,2), n %/% 2)
rfx_coefs <- matrix(c(-1, -1, 1, 1), nrow=2, byrow=TRUE)
rfx_basis <- cbind(1, runif(n, -1, 1))
rfx_term <- rowSums(rfx_coefs[group_ids,] * rfx_basis)
E_y <- f_XW + rfx_term
y <- E_y + rnorm(n, 0, 1)*(sd(E_y)/snr)
test_set_pct <- 0.2
n_test <- round(test_set_pct*n)
n_train <- n - n_test
test_inds <- sort(sample(1:n, n_test, replace = FALSE))
train_inds <- (1:n)[!((1:n) %in% test_inds)]
```

```
X_test <- X[test_inds,]
X_train <- X[train_inds,]
y_test <- y[test_inds]
y_train <- y[train_inds]
rfx_group_ids_test <- group_ids[test_inds]
rfx_group_ids_train <- group_ids[train_inds]
rfx_basis_test <- rfx_basis[test_inds,]
rfx_basis_train <- rfx_basis[train_inds,]
rfx_term_test <- rfx_term[test_inds]
rfx_term_train <- rfx_term[train_inds]
bart_model <- bart(X_train = X_train, y_train = y_train, X_test = X_test,
                   rfx_group_ids_train = rfx_group_ids_train,
                   rfx_group_ids_test = rfx_group_ids_test,
                   rfx_basis_train = rfx_basis_train,
                   rfx_basis_test = rfx_basis_test,
                   num_gfr = 10, num_burnin = 0, num_mcmc = 10)
rfx_samples <- getRandomEffectSamples(bart_model)
```

`getRandomEffectSamples.bcfmodel`
Extract Random Effect Samples from BCF Model

Description

Extract raw sample values for each of the random effect parameter terms.

Usage

```
## S3 method for class 'bcfmodel'
getRandomEffectSamples(object, ...)
```

Arguments

`object`	Object of type `bcfmodel` containing draws of a Bayesian causal forest model and associated sampling outputs.
`...`	Other parameters to be used in random effects extraction

Value

List of arrays. The alpha array has dimension (`num_components`, `num_samples`) and is simply a vector if `num_components = 1`. The xi and beta arrays have dimension (`num_components`, `num_groups`, `num_samples`) and is simply a matrix if `num_components = 1`. The sigma array has dimension (`num_components`, `num_samples`) and is simply a vector if `num_components = 1`.

Examples

```
n <- 500
p <- 5
X <- matrix(runif(n*p), ncol = p)
mu_x <- (
    ((0 <= X[,1]) & (0.25 > X[,1])) * (-7.5) +
    ((0.25 <= X[,1]) & (0.5 > X[,1])) * (-2.5) +
    ((0.5 <= X[,1]) & (0.75 > X[,1])) * (2.5) +
    ((0.75 <= X[,1]) & (1 > X[,1])) * (7.5)
)
pi_x <- (
    ((0 <= X[,1]) & (0.25 > X[,1])) * (0.2) +
    ((0.25 <= X[,1]) & (0.5 > X[,1])) * (0.4) +
    ((0.5 <= X[,1]) & (0.75 > X[,1])) * (0.6) +
    ((0.75 <= X[,1]) & (1 > X[,1])) * (0.8)
)
tau_x <- (
    ((0 <= X[,2]) & (0.25 > X[,2])) * (0.5) +
    ((0.25 <= X[,2]) & (0.5 > X[,2])) * (1.0) +
    ((0.5 <= X[,2]) & (0.75 > X[,2])) * (1.5) +
    ((0.75 <= X[,2]) & (1 > X[,2])) * (2.0)
)
Z <- rbinom(n, 1, pi_x)
E_XZ <- mu_x + Z*tau_x
snr <- 3
rfx_group_ids <- rep(c(1,2), n %/% 2)
rfx_coefs <- matrix(c(-1, -1, 1, 1), nrow=2, byrow=TRUE)
rfx_basis <- cbind(1, runif(n, -1, 1))
rfx_term <- rowSums(rfx_coefs[rfx_group_ids,] * rfx_basis)
y <- E_XZ + rfx_term + rnorm(n, 0, 1)*(sd(E_XZ)/snr)
test_set_pct <- 0.2
n_test <- round(test_set_pct*n)
n_train <- n - n_test
test_inds <- sort(sample(1:n, n_test, replace = FALSE))
train_inds <- (1:n)[!((1:n) %in% test_inds)]
X_test <- X[test_inds,]
X_train <- X[train_inds,]
pi_test <- pi_x[test_inds]
pi_train <- pi_x[train_inds]
Z_test <- Z[test_inds]
Z_train <- Z[train_inds]
y_test <- y[test_inds]
y_train <- y[train_inds]
mu_test <- mu_x[test_inds]
mu_train <- mu_x[train_inds]
tau_test <- tau_x[test_inds]
tau_train <- tau_x[train_inds]
rfx_group_ids_test <- rfx_group_ids[test_inds]
rfx_group_ids_train <- rfx_group_ids[train_inds]
rfx_basis_test <- rfx_basis[test_inds,]
rfx_basis_train <- rfx_basis[train_inds,]
rfx_term_test <- rfx_term[test_inds]
```

```
rfx_term_train <- rfx_term[train_inds]
mu_params <- list(sample_sigma2_leaf = TRUE)
tau_params <- list(sample_sigma2_leaf = FALSE)
bcf_model <- bcf(X_train = X_train, Z_train = Z_train, y_train = y_train,
                 propensity_train = pi_train,
                 rfx_group_ids_train = rfx_group_ids_train,
                 rfx_basis_train = rfx_basis_train, X_test = X_test,
                 Z_test = Z_test, propensity_test = pi_test,
                 rfx_group_ids_test = rfx_group_ids_test,
                 rfx_basis_test = rfx_basis_test,
                 num_gfr = 10, num_burnin = 0, num_mcmc = 10,
                 prognostic_forest_params = mu_params,
                 treatment_effect_forest_params = tau_params)
rfx_samples <- getRandomEffectSamples(bcf_model)
```

`GlobalModelConfig` *Global Model Configuration Object*

Description

Object used to get / set global parameters and other global model configuration options in the "low-level" stochtree interface. The "low-level" stochtree interface enables a high degreee of sampler customization, in which users employ R wrappers around C++ objects like ForestDataset, Outcome, CppRng, and ForestModel to run the Gibbs sampler of a BART model with custom modifications. GlobalModelConfig allows users to specify / query the global parameters of a model they wish to run.

This class is intended for advanced use cases in which users require detailed control of sampling algorithms and data structures. Minimal input validation and error checks are performed – users are responsible for providing the correct inputs. For tutorials on the "proper" usage of the stochtree's advanced workflow, we provide several vignettes at https://stochtree.ai/

Value

Global error variance parameter

Public fields

`global_error_variance` Global error variance parameter Create a new GlobalModelConfig object.

Methods

Public methods:

- `GlobalModelConfig$new()`
- `GlobalModelConfig$update_global_error_variance()`
- `GlobalModelConfig$get_global_error_variance()`

Method `new()`:

Usage:

```
GlobalModelConfig$new(global_error_variance = 1)
```

Arguments:

`global_error_variance` Global error variance parameter (default: `1.0`)

Returns: A new GlobalModelConfig object.

Method `update_global_error_variance()`**:** Update global error variance parameter

Usage:

```
GlobalModelConfig$update_global_error_variance(global_error_variance)
```

Arguments:

`global_error_variance` Global error variance parameter

Method `get_global_error_variance()`**:** Query global error variance parameter for this GlobalModelConfig object

Usage:

```
GlobalModelConfig$get_global_error_variance()
```

`JSONCreation` *JSON Creation Routines*

Description

The CppJson class wraps an external pointer to C++ JSON object for a seamless R serialization interface. This function group provides several utilities for creating CppJson objects: from scratch (i.e. initializing an empty JSON object), from a JSON file, or from a string that parses to valid JSON.

`createCppJson` creates a new (empty) CppJson object. `createCppJsonFile` creates and populates a CppJson object from data in a JSON file. `createCppJsonString` creates and populates a CppJson object from a JSON string.

These functions are intended for advanced use cases in which users require detailed control of sampling algorithms and data structures. Minimal input validation and error checks are performed – users are responsible for providing the correct inputs. For tutorials on the "proper" usage of the stochtree's advanced workflow, we provide several vignettes at https://stochtree.ai/

Usage

```
createCppJson()

createCppJsonFile(json_filename)

createCppJsonString(json_string)
```

Arguments

`json_filename`	Name of JSON file to read. Must end in `.json`.
`json_string`	JSON string

Value

Each of the functions in this group returns a CppJson object.

Examples

```
example_vec <- runif(10)
example_json <- createCppJson()
example_json$add_vector("myvec", example_vec)
tmpjson <- tempfile(fileext = ".json")
example_json$save_file(file.path(tmpjson))
example_json_roundtrip <- createCppJsonFile(file.path(tmpjson))
unlink(tmpjson)
example_json_string <- example_json$return_json_string()
example_json_roundtrip <- createCppJsonString(example_json_string)
```

---

loadScalarJson *Load Scalar from JSON*

Description

Load a scalar from json

This function is intended for advanced use cases in which users require detailed control of sampling algorithms and data structures. Minimal input validation and error checks are performed – users are responsible for providing the correct inputs. For tutorials on the "proper" usage of the stochtree's advanced workflow, we provide several vignettes at https://stochtree.ai/

Usage

```
loadScalarJson(json_object, json_scalar_label, subfolder_name = NULL)
```

Arguments

json_object	Object of class CppJson
json_scalar_label	Label referring to a particular scalar / string value (i.e. "num_samples") in the overall json hierarchy
subfolder_name	(Optional) Name of the subfolder / hierarchy under which vector sits

Value

R vector

Examples

```
example_scalar <- 5.4
example_json <- createCppJson()
example_json$add_scalar("myscalar", example_scalar)
roundtrip_scalar <- loadScalarJson(example_json, "myscalar")
```

loadVectorJson *Load Vector from JSON*

Description

Load a vector from json

This function is intended for advanced use cases in which users require detailed control of sampling algorithms and data structures. Minimal input validation and error checks are performed – users are responsible for providing the correct inputs. For tutorials on the "proper" usage of the stochtree's advanced workflow, we provide several vignettes at https://stochtree.ai/

Usage

```
loadVectorJson(json_object, json_vector_label, subfolder_name = NULL)
```

Arguments

json_object	Object of class CppJson
json_vector_label	Label referring to a particular vector (i.e. "sigma2_global_samples") in the overall json hierarchy
subfolder_name	(Optional) Name of the subfolder / hierarchy under which vector sits

Value

R vector

Examples

```
example_vec <- runif(10)
example_json <- createCppJson()
example_json$add_vector("myvec", example_vec)
roundtrip_vec <- loadVectorJson(example_json, "myvec")
```

Outcome *Outcome Data C++ Wrapper*

Description

Outcome / partial residual used to sample an additive model. The outcome class is a wrapper around a vector of (mutable) outcomes for ML tasks (supervised learning, causal inference). When an additive tree ensemble is sampled, the outcome used to sample a specific model term is the "partial residual" consisting of the outcome minus the predictions of every other model term (trees, group random effects, etc...).

This class is intended for advanced use cases in which users require detailed control of sampling algorithms and data structures. Minimal input validation and error checks are performed – users are responsible for providing the correct inputs. For tutorials on the "proper" usage of the stochtree's advanced workflow, we provide several vignettes at https://stochtree.ai/

Public fields

`data_ptr` External pointer to a C++ `Outcome` class

Methods

Public methods:

- `Outcome$new()`
- `Outcome$get_data()`
- `Outcome$add_vector()`
- `Outcome$subtract_vector()`
- `Outcome$update_data()`

Method `new()`**:** Create a new Outcome object.

Usage:

```
Outcome$new(outcome)
```

Arguments:

`outcome` Vector of outcome values

Returns: A new `Outcome` object.

Method `get_data()`**:** Extract raw data in R from the underlying C++ object

Usage:

```
Outcome$get_data()
```

Returns: R vector containing (copy of) the values in `Outcome` object

Method `add_vector()`**:** Update the current state of the outcome (i.e. partial residual) data by adding the values of `update_vector`

Usage:

```
Outcome$add_vector(update_vector)
```

Arguments:

`update_vector` Vector to be added to outcome

Returns: None

Method `subtract_vector()`**:** Update the current state of the outcome (i.e. partial residual) data by subtracting the values of `update_vector`

Usage:

```
Outcome$subtract_vector(update_vector)
```

Arguments:

`update_vector` Vector to be subtracted from outcome

Returns: None

Method `update_data()`**:** Update the current state of the outcome (i.e. partial residual) data by replacing each element with the elements of `new_vector`

Usage:

```
Outcome$update_data(new_vector)
```

Arguments:

`new_vector` Vector from which to overwrite the current data

Returns: None

`OutcomeModel` *Create a New Outcome Model Object*

Description

Create a New Outcome Model Object

Usage

```
OutcomeModel(outcome = "continuous", link = NULL)
```

Arguments

`outcome`	Character string specifying the outcome type.
`link`	Character string specifying the link function.

Value

An object of class `outcome_model`.

Examples

```
my_model <- OutcomeModel(outcome = "continuous", link = "identity")
```

plot.bartmodel *Plot BART Model Fit*

Description

Plot the BART model fit and any relevant sampled quantities. This will default to a traceplot of the global error scale and the in-sample mean forest predictions for the first train set observation. Since `stochtree::bart()` is flexible and it's possible to sample a model with a fixed global error scale and no mean forest, this procedure is adaptive and will attempt to plot a trace of whichever model terms are included if these two default terms are omitted.

Usage

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

Arguments

`x`	The BART model object
`...`	Additional arguments

Value

BART model object unchanged after summarizing

plot.bcfmodel *Plot BCF Model*

Description

Plot the BCF model fit and any relevant sampled quantities. This will default to a traceplot of the global error scale and the in-sample mean forest predictions for the first train set observation. Since `stochtree::bcf()` is flexible and it's possible to sample a model with a fixed global error scale and no mean forest, this procedure will throw an error if these two default terms are omitted.

Usage

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

Arguments

`x`	The BCF model object
`...`	Additional arguments

Value

BCF model object unchanged after summarizing

`predict.bartmodel` *Predict from a BART Model*

Description

Predict from a sampled BART model on new data

Usage

```
## S3 method for class 'bartmodel'
predict(
  object,
  X,
  leaf_basis = NULL,
  rfx_group_ids = NULL,
  rfx_basis = NULL,
  type = "posterior",
  terms = "all",
  scale = "linear",
  ...
)
```

Arguments

`object`	Object of type `bart` containing draws of a regression forest and associated sampling outputs.
`X`	Covariates used to determine tree leaf predictions for each observation. Must be passed as a matrix or dataframe.
`leaf_basis`	(Optional) Bases used for prediction (by e.g. dot product with leaf values). Default: `NULL`.
`rfx_group_ids`	(Optional) Test set group labels used for an additive random effects model. We do not currently support (but plan to in the near future), test set evaluation for group labels that were not in the training set.
`rfx_basis`	(Optional) Test set basis for "random-slope" regression in additive random effects model.
`type`	(Optional) Type of prediction to return. Options are "mean", which averages the predictions from every draw of a BART model, and "posterior", which returns the entire matrix of posterior predictions. Default: "posterior".
`terms`	(Optional) Which model terms to include in the prediction. This can be a single term or a list of model terms. Options include "y_hat", "mean_forest", "rfx", "variance_forest", or "all". If a model doesn't have mean forest, random effects, or variance forest predictions, but one of those terms is request, the request will

simply be ignored. If none of the requested terms are present in a model, this function will return NULL along with a warning. Default: "all".

scale (Optional) Scale of mean function predictions. Options are "linear", which returns predictions on the original scale of the mean forest / RFX terms, "probability", which transforms predictions into class probabilities for models with discrete outcomes, and "class", which returns predicted outcome categories for discrete outcome models. "probability" is only valid for outcome models with outcome == 'binary' or outcome == 'ordinal'. For binary outcomes, this will return the probability that y == 1, and for ordinal outcomes, this will return probabilities for each outcome label. Default: "linear".

... (Optional) Other prediction parameters.

Value

List of prediction matrices or single prediction matrix / vector, depending on the terms requested.

Examples

```
n <- 100
p <- 5
X <- matrix(runif(n*p), ncol = p)
f_XW <- (
    ((0 <= X[,1]) & (0.25 > X[,1])) * (-7.5) +
    ((0.25 <= X[,1]) & (0.5 > X[,1])) * (-2.5) +
    ((0.5 <= X[,1]) & (0.75 > X[,1])) * (2.5) +
    ((0.75 <= X[,1]) & (1 > X[,1])) * (7.5)
)
noise_sd <- 1
y <- f_XW + rnorm(n, 0, noise_sd)
test_set_pct <- 0.2
n_test <- round(test_set_pct*n)
n_train <- n - n_test
test_inds <- sort(sample(1:n, n_test, replace = FALSE))
train_inds <- (1:n)[!((1:n) %in% test_inds)]
X_test <- X[test_inds,]
X_train <- X[train_inds,]
y_test <- y[test_inds]
y_train <- y[train_inds]
bart_model <- bart(X_train = X_train, y_train = y_train,
                   num_gfr = 10, num_burnin = 0, num_mcmc = 10)
y_hat_test <- predict(bart_model, X=X_test)$y_hat
```

---

predict.bcfmodel *Predict from BCF Model*

Description

Predict from a sampled BCF model on new data

Usage

```
## S3 method for class 'bcfmodel'
predict(
  object,
  X,
  Z,
  propensity = NULL,
  rfx_group_ids = NULL,
  rfx_basis = NULL,
  type = "posterior",
  terms = "all",
  scale = "linear",
  ...
)
```

Arguments

object	Object of type bcfmodel containing draws of a Bayesian causal forest model and associated sampling outputs.
X	Covariates used to determine tree leaf predictions for each observation. Must be passed as a matrix or dataframe.
Z	Treatments used for prediction.
propensity	(Optional) Propensities used for prediction.
rfx_group_ids	(Optional) Test set group labels used for an additive random effects model. We do not currently support (but plan to in the near future), test set evaluation for group labels that were not in the training set.
rfx_basis	(Optional) Test set basis for "random-slope" regression in additive random effects model. If the model was sampled with a random effects model_spec of "intercept_only" or "intercept_plus_treatment", this is optional, but if it is provided, it will be used.
type	(Optional) Type of prediction to return. Options are "mean", which averages the predictions from every draw of a BCF model, and "posterior", which returns the entire matrix of posterior predictions. Default: "posterior".
terms	(Optional) Which model terms to include in the prediction. Options include "y_hat", "prognostic_function", "mu", "cate", "tau", "rfx", "variance_forest", or "all".

The treatment effect terms follow a three-level hierarchy:

- "tau" returns tau_0 + tau(X): the parametric treatment intercept (if sampled) plus the treatment forest. This matches model$tau_hat_train / model$tau_hat_test.
- "cate" additionally folds in the random slope on treatment when random effects are fit with rfx_model_spec = "intercept_plus_treatment"; otherwise it is identical to "tau".
- The raw forest-only component (without tau_0) is not directly returned by this method; use model$forests_tau to access it.

Similarly for the prognostic term: `"mu"` returns the prognostic forest only, while `"prognostic_function"` additionally folds in the random intercept when `rfx_model_spec` is `"intercept_only"` or `"intercept_plus_treatment"`; otherwise the two are identical.

If a model doesn't have random effects or variance forest predictions but one of those terms is requested, the request will simply be ignored. If none of the requested terms are present, this function will return `NULL` along with a warning. Default: `"all"`.

`scale` (Optional) Scale of mean function predictions. Options are "linear", which returns predictions on the original scale of the mean forest / RFX terms, and "probability", which transforms predictions into a probability of observing `y == 1`. "probability" is only valid for models fit with a probit outcome model. Default: "linear".

`...` (Optional) Other prediction parameters.

Value

List of prediction matrices or single prediction matrix / vector, depending on the terms requested.

Examples

```
n <- 500
p <- 5
X <- matrix(runif(n*p), ncol = p)
mu_x <- (
    ((0 <= X[,1]) & (0.25 > X[,1])) * (-7.5) +
    ((0.25 <= X[,1]) & (0.5 > X[,1])) * (-2.5) +
    ((0.5 <= X[,1]) & (0.75 > X[,1])) * (2.5) +
    ((0.75 <= X[,1]) & (1 > X[,1])) * (7.5)
)
pi_x <- (
    ((0 <= X[,1]) & (0.25 > X[,1])) * (0.2) +
    ((0.25 <= X[,1]) & (0.5 > X[,1])) * (0.4) +
    ((0.5 <= X[,1]) & (0.75 > X[,1])) * (0.6) +
    ((0.75 <= X[,1]) & (1 > X[,1])) * (0.8)
)
tau_x <- (
    ((0 <= X[,2]) & (0.25 > X[,2])) * (0.5) +
    ((0.25 <= X[,2]) & (0.5 > X[,2])) * (1.0) +
    ((0.5 <= X[,2]) & (0.75 > X[,2])) * (1.5) +
    ((0.75 <= X[,2]) & (1 > X[,2])) * (2.0)
)
Z <- rbinom(n, 1, pi_x)
noise_sd <- 1
y <- mu_x + tau_x*Z + rnorm(n, 0, noise_sd)
test_set_pct <- 0.2
n_test <- round(test_set_pct*n)
n_train <- n - n_test
test_inds <- sort(sample(1:n, n_test, replace = FALSE))
train_inds <- (1:n)[!((1:n) %in% test_inds)]
X_test <- X[test_inds,]
```

```
X_train <- X[train_inds,]
pi_test <- pi_x[test_inds]
pi_train <- pi_x[train_inds]
Z_test <- Z[test_inds]
Z_train <- Z[train_inds]
y_test <- y[test_inds]
y_train <- y[train_inds]
mu_test <- mu_x[test_inds]
mu_train <- mu_x[train_inds]
tau_test <- tau_x[test_inds]
tau_train <- tau_x[train_inds]
bcf_model <- bcf(X_train = X_train, Z_train = Z_train, y_train = y_train,
                 propensity_train = pi_train, num_gfr = 10,
                 num_burnin = 0, num_mcmc = 10)
preds <- predict(bcf_model, X_test, Z_test, pi_test)
```

`print.bartmodel` *Print Summary of BART Model*

Description

Prints a summary of the BART model, including the model terms and their specifications.

Usage

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

Arguments

`x`	The BART model object
`...`	Additional arguments

Value

BART model object unchanged after printing summary

`print.bcfmodel` *Print Summary of BCF Model*

Description

Prints a summary of the BCF model, including the model terms and their specifications.

Usage

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

Arguments

x	The BCF model object
...	Additional arguments (currently unused)

Value

BCF model object unchanged after printing summary

---

print.ForestSamples *Summarize a ForestSamples Object*

Description

Prints a summary of the ForestSamples object, including number of forests and the underlying model of each forest.

Usage

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

Arguments

x	ForestSamples object
...	Additional arguments

Value

ForestSamples object unchanged after printing summary

---

print.RandomEffectSamples *Summarize a RandomEffectSamples Object*

Description

Prints a summary of the RandomEffectSamples object, including number of forests and the underlying model of each forest.

Usage

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

Arguments

x	RandomEffectSamples object
...	Additional arguments

Value

RandomEffectSamples object unchanged after printing summary

RandomEffectSamples *Random Effect Container C++ Wrapper*

Description

Class that wraps the "persistent" aspects of a C++ random effects model, including draws of the parameters and a map from the original label indices to the 0-indexed label numbers used to place group samples in memory (i.e. the first label is stored in column 0 of the sample matrix, the second label is store in column 1 of the sample matrix, etc...)

This class is intended for advanced use cases in which users require detailed control of sampling algorithms and data structures. Minimal input validation and error checks are performed – users are responsible for providing the correct inputs. For tutorials on the "proper" usage of the stochtree's advanced workflow, we provide several vignettes at https://stochtree.ai/

Coordinates various C++ random effects classes and persists those needed for prediction / serialization

Public fields

`rfx_container_ptr` External pointer to a C++ StochTree::RandomEffectsContainer class

`label_mapper_ptr` External pointer to a C++ StochTree::LabelMapper class

`training_group_ids` Unique vector of group IDs that were in the training dataset

Methods

Public methods:

- `RandomEffectSamples$new()`
- `RandomEffectSamples$load_in_session()`
- `RandomEffectSamples$load_from_json()`
- `RandomEffectSamples$append_from_json()`
- `RandomEffectSamples$load_from_json_string()`
- `RandomEffectSamples$append_from_json_string()`
- `RandomEffectSamples$predict()`
- `RandomEffectSamples$extract_parameter_samples()`
- `RandomEffectSamples$delete_sample()`
- `RandomEffectSamples$extract_label_mapping()`

- RandomEffectSamples$num_samples()
- RandomEffectSamples$num_components()
- RandomEffectSamples$num_groups()

Method `new()`**:** Create a new RandomEffectSamples object.

Usage:

```
RandomEffectSamples$new()
```

Returns: A new `RandomEffectSamples` object.

Method `load_in_session()`**:** Construct RandomEffectSamples object from other "in-session" R objects

Usage:

```
RandomEffectSamples$load_in_session(
  num_components,
  num_groups,
  random_effects_tracker
)
```

Arguments:

`num_components` Number of "components" or bases defining the random effects regression

`num_groups` Number of random effects groups

`random_effects_tracker` Object of type `RandomEffectsTracker`

Returns: None

Method `load_from_json()`**:** Construct RandomEffectSamples object from a json object

Usage:

```
RandomEffectSamples$load_from_json(
  json_object,
  json_rfx_container_label,
  json_rfx_mapper_label,
  json_rfx_groupids_label
)
```

Arguments:

`json_object` Object of class `CppJson`

`json_rfx_container_label` Label referring to a particular rfx sample container (i.e. "random_effect_container_0") in the overall json hierarchy

`json_rfx_mapper_label` Label referring to a particular rfx label mapper (i.e. "random_effect_label_mapper_0") in the overall json hierarchy

`json_rfx_groupids_label` Label referring to a particular set of rfx group IDs (i.e. "random_effect_groupids_0") in the overall json hierarchy

Returns: A new `RandomEffectSamples` object.

Method `append_from_json()`**:** Append random effect draws to `RandomEffectSamples` object from a json object

Usage:

```
RandomEffectSamples$append_from_json(
  json_object,
  json_rfx_container_label,
  json_rfx_mapper_label,
  json_rfx_groupids_label
)
```

Arguments:

`json_object` Object of class `CppJson`

`json_rfx_container_label` Label referring to a particular rfx sample container (i.e. "random_effect_container_0") in the overall json hierarchy

`json_rfx_mapper_label` Label referring to a particular rfx label mapper (i.e. "random_effect_label_mapper_0") in the overall json hierarchy

`json_rfx_groupids_label` Label referring to a particular set of rfx group IDs (i.e. "random_effect_groupids_0") in the overall json hierarchy

Returns: None

Method `load_from_json_string()`**:** Construct RandomEffectSamples object from a json object

Usage:

```
RandomEffectSamples$load_from_json_string(
  json_string,
  json_rfx_container_label,
  json_rfx_mapper_label,
  json_rfx_groupids_label
)
```

Arguments:

`json_string` JSON string which parses into object of class `CppJson`

`json_rfx_container_label` Label referring to a particular rfx sample container (i.e. "random_effect_container_0") in the overall json hierarchy

`json_rfx_mapper_label` Label referring to a particular rfx label mapper (i.e. "random_effect_label_mapper_0") in the overall json hierarchy

`json_rfx_groupids_label` Label referring to a particular set of rfx group IDs (i.e. "random_effect_groupids_0") in the overall json hierarchy

Returns: A new `RandomEffectSamples` object.

Method `append_from_json_string()`**:** Append random effect draws to `RandomEffectSamples` object from a json object

Usage:

```
RandomEffectSamples$append_from_json_string(
  json_string,
  json_rfx_container_label,
  json_rfx_mapper_label,
  json_rfx_groupids_label
)
```

Arguments:

`json_string` JSON string which parses into object of class `CppJson`

`json_rfx_container_label` Label referring to a particular rfx sample container (i.e. "random_effect_container_0") in the overall json hierarchy

`json_rfx_mapper_label` Label referring to a particular rfx label mapper (i.e. "random_effect_label_mapper_0") in the overall json hierarchy

`json_rfx_groupids_label` Label referring to a particular set of rfx group IDs (i.e. "random_effect_groupids_0") in the overall json hierarchy

Returns: None

Method `predict()`**:** Predict random effects for each observation implied by `rfx_group_ids` and `rfx_basis`. If a random effects model is "intercept-only" the `rfx_basis` will be a vector of ones of size `length(rfx_group_ids)`.

Usage:

```
RandomEffectSamples$predict(rfx_group_ids, rfx_basis = NULL)
```

Arguments:

`rfx_group_ids` Indices of random effects groups in a prediction set

`rfx_basis` (Optional) Basis used for random effects prediction

Returns: Matrix with as many rows as observations provided and as many columns as samples drawn of the model.

Method `extract_parameter_samples()`**:** Extract the random effects parameters sampled. With the "redundant parameterization" of Gelman et al (2008), this includes four parameters: alpha (the "working parameter" shared across every group), xi (the "group parameter" sampled separately for each group), beta (the product of alpha and xi, which corresponds to the overall group-level random effects), and sigma (group-independent prior variance for each component of xi).

Usage:

```
RandomEffectSamples$extract_parameter_samples()
```

Returns: List of arrays. The alpha array has dimension (`num_components`, `num_samples`) and is simply a vector if `num_components = 1`. The xi and beta arrays have dimension (`num_components`, `num_groups`, `num_samples`) and are simply matrices if `num_components = 1`. The sigma array has dimension (`num_components`, `num_samples`) and is simply a vector if `num_components = 1`.

Method `delete_sample()`**:** Modify the `RandomEffectsSamples` object by removing the parameter samples index by `sample_num`.

Usage:

```
RandomEffectSamples$delete_sample(sample_num)
```

Arguments:

`sample_num` Index of the RFX sample to be removed

Method `extract_label_mapping()`**:** Convert the mapping of group IDs to random effect components indices from C++ to R native format

Usage:

```
RandomEffectSamples$extract_label_mapping()
```

Returns: List mapping group ID to random effect components.

Method `num_samples()`**:** Query the number of samples in the `RandomEffectsSamples` object.

Usage:

```
RandomEffectSamples$num_samples()
```

Returns: Integer number of samples

Method `num_components()`**:** Query the number of components in the `RandomEffectsSamples` object.

Usage:

```
RandomEffectSamples$num_components()
```

Returns: Integer number of components

Method `num_groups()`**:** Query the number of groups in the `RandomEffectsSamples` object.

Usage:

```
RandomEffectSamples$num_groups()
```

Returns: Integer number of groups

`RandomEffectSamplesSerialization`

Random Effects Container Serialization Routines

Description

While the BARTSerialization and BCFSerialization topics focus on JSON serialization / deserialization for entire `bartmodel` and `bcfmodel` objects, this function group provides an interface for a more focused use case: loading a single RandomEffectSamples container from a broader BART / BCF model (which may include forests and other parametric terms).

`loadRandomEffectSamplesJson` converts a CppJson object representing a BART or BCF model into a RandomEffectSamples container by extracting the JSON indexed by an integer label (i.e. `0`) and deserializing it into a RandomEffectSamples object.

Both `loadRandomEffectSamplesJson` and `loadRandomEffectSamplesCombinedJson` operate similarly, but on a list of CppJson or JSON string representations of BART / BCF models with the same structure.

These functions are intended for advanced use cases in which users require detailed control of sampling algorithms and data structures. Minimal input validation and error checks are performed – users are responsible for providing the correct inputs. For tutorials on the "proper" usage of the stochtree's advanced workflow, we provide several vignettes at `https://stochtree.ai/`

Usage

```
loadRandomEffectSamplesJson(json_object, json_rfx_num)

loadRandomEffectSamplesCombinedJson(json_object_list, json_rfx_num)

loadRandomEffectSamplesCombinedJsonString(json_string_list, json_rfx_num)
```

Arguments

`json_object`	Object of class `CppJson`
`json_rfx_num`	Integer index indicating the position of the random effects term to be unpacked (must exist in every json object in a list if a list is provided)
`json_object_list`	List of objects of class `CppJson`
`json_string_list`	List of objects of class `CppJson`

Value

Each of the functions in this group returns a `RandomEffectSamples` object.

Examples

```
n <- 100
p <- 10
X <- matrix(runif(n*p), ncol = p)
rfx_group_ids <- sample(1:2, size = n, replace = TRUE)
rfx_basis <- rep(1.0, n)
y <- (-5 + 10*(X[,1] > 0.5)) + (-2*(rfx_group_ids==1)+2*(rfx_group_ids==2))
bart_model <- bart(X_train=X, y_train=y, rfx_group_ids_train=rfx_group_ids,
                   rfx_basis_train = rfx_basis, num_gfr=0, num_mcmc=10)
bart_json <- saveBARTModelToJson(bart_model)
bart_json_string <- saveBARTModelToJsonString(bart_model)
bart_json_list <- list(bart_json)
bart_json_string_list <- list(bart_json_string)
rfx_container <- loadRandomEffectSamplesJson(bart_json, 0)
rfx_container <- loadRandomEffectSamplesCombinedJson(bart_json_list, 0)
rfx_container <- loadRandomEffectSamplesCombinedJsonString(bart_json_string_list, 0)
```

`RandomEffectsDataset` *Random Effects Dataset C++ Wrapper*

Description

Dataset used to sample a random effects model. A random effects dataset consists of three matrices / vectors: group labels, bases, and variance weights. Variance weights are optional.

This class is intended for advanced use cases in which users require detailed control of sampling algorithms and data structures. Minimal input validation and error checks are performed – users are responsible for providing the correct inputs. For tutorials on the "proper" usage of the stochtree's advanced workflow, we provide several vignettes at https://stochtree.ai/

Public fields

`data_ptr` External pointer to a C++ RandomEffectsDataset class

Methods

Public methods:

- `RandomEffectsDataset$new()`
- `RandomEffectsDataset$update_basis()`
- `RandomEffectsDataset$update_variance_weights()`
- `RandomEffectsDataset$num_observations()`
- `RandomEffectsDataset$num_basis()`
- `RandomEffectsDataset$get_group_labels()`
- `RandomEffectsDataset$get_basis()`
- `RandomEffectsDataset$get_variance_weights()`
- `RandomEffectsDataset$has_group_labels()`
- `RandomEffectsDataset$has_basis()`
- `RandomEffectsDataset$has_variance_weights()`

Method `new()`**:** Create a new RandomEffectsDataset object.

Usage:

```
RandomEffectsDataset$new(group_labels, basis, variance_weights = NULL)
```

Arguments:

`group_labels` Vector of group labels

`basis` Matrix of bases used to define the random effects regression (for an intercept-only model, pass an array of ones)

`variance_weights` (Optional) Vector of observation-specific variance weights

Returns: A new `RandomEffectsDataset` object.

Method `update_basis()`**:** Update basis matrix in a dataset

Usage:

```
RandomEffectsDataset$update_basis(basis)
```

Arguments:

`basis` Updated matrix of bases used to define random slopes / intercepts

Method `update_variance_weights()`**:** Update variance_weights in a dataset

Usage:

```
RandomEffectsDataset$update_variance_weights(
  variance_weights,
  exponentiate = F
)
```

Arguments:

`variance_weights` Updated vector of variance weights used to define individual variance / case weights

`exponentiate` Whether or not input vector should be exponentiated before being written to the RandomEffectsDataset's variance weights. Default: F.

Method `num_observations()`**:** Return number of observations in a `RandomEffectsDataset` object

Usage:

```
RandomEffectsDataset$num_observations()
```

Returns: Observation count

Method `num_basis()`**:** Return dimension of the basis matrix in a `RandomEffectsDataset` object

Usage:

```
RandomEffectsDataset$num_basis()
```

Returns: Basis vector count

Method `get_group_labels()`**:** Return group labels as an R vector

Usage:

```
RandomEffectsDataset$get_group_labels()
```

Returns: Group label data

Method `get_basis()`**:** Return bases as an R matrix

Usage:

```
RandomEffectsDataset$get_basis()
```

Returns: Basis data

Method `get_variance_weights()`**:** Return variance weights as an R vector

Usage:

```
RandomEffectsDataset$get_variance_weights()
```

Returns: Variance weight data

Method `has_group_labels()`**:** Whether or not a dataset has group label indices

Usage:

```
RandomEffectsDataset$has_group_labels()
```

Returns: True if group label vector is loaded, false otherwise

Method `has_basis()`**:** Whether or not a dataset has a basis matrix

Usage:

```
RandomEffectsDataset$has_basis()
```

Returns: True if basis matrix is loaded, false otherwise

Method `has_variance_weights()`**:** Whether or not a dataset has variance weights

Usage:

```
RandomEffectsDataset$has_variance_weights()
```

Returns: True if variance weights are loaded, false otherwise

---

`RandomEffectsModel` *Random Effects Model C++ Wrapper*

Description

The core "model" class for sampling random effects. Stores current model state, prior parameters, and procedures for sampling from the conditional posterior of each parameter.

This class is intended for advanced use cases in which users require detailed control of sampling algorithms and data structures. Minimal input validation and error checks are performed – users are responsible for providing the correct inputs. For tutorials on the "proper" usage of the stochtree's advanced workflow, we provide several vignettes at https://stochtree.ai/

Public fields

`rfx_model_ptr` External pointer to a C++ StochTree::RandomEffectsModel class

`num_groups` Number of groups in the random effects model

`num_components` Number of components (i.e. dimension of basis) in the random effects model

Methods

Public methods:

- `RandomEffectsModel$new()`
- `RandomEffectsModel$sample_random_effect()`
- `RandomEffectsModel$predict()`
- `RandomEffectsModel$set_working_parameter()`
- `RandomEffectsModel$set_group_parameters()`
- `RandomEffectsModel$set_working_parameter_cov()`
- `RandomEffectsModel$set_group_parameter_cov()`
- `RandomEffectsModel$set_variance_prior_shape()`
- `RandomEffectsModel$set_variance_prior_scale()`

Method `new()`**:** Create a new RandomEffectsModel object.

Usage:

```
RandomEffectsModel$new(num_components, num_groups)
```

Arguments:

`num_components` Number of "components" or bases defining the random effects regression

`num_groups` Number of random effects groups

Returns: A new `RandomEffectsModel` object.

Method `sample_random_effect()`**:** Sample from random effects model.

Usage:

```
RandomEffectsModel$sample_random_effect(
  rfx_dataset,
  residual,
  rfx_tracker,
  rfx_samples,
  keep_sample,
  global_variance,
  rng
)
```

Arguments:

`rfx_dataset` Object of type `RandomEffectsDataset`

`residual` Object of type `Outcome`

`rfx_tracker` Object of type `RandomEffectsTracker`

`rfx_samples` Object of type `RandomEffectSamples`

`keep_sample` Whether sample should be retained in `rfx_samples`. If `FALSE`, the state of `rfx_tracker` will be updated, but the parameter values will not be added to the sample container. Samples are commonly discarded due to burn-in or thinning.

`global_variance` Scalar global variance parameter

`rng` Object of type `CppRNG`

Returns: None

Method `predict()`**:** Predict from (a single sample of a) random effects model.

Usage:

```
RandomEffectsModel$predict(rfx_dataset, rfx_tracker)
```

Arguments:

`rfx_dataset` Object of type `RandomEffectsDataset`

`rfx_tracker` Object of type `RandomEffectsTracker`

Returns: Vector of predictions with size matching number of observations in rfx_dataset

Method `set_working_parameter()`**:** Set value for the "working parameter." This is typically used for initialization, but could also be used to interrupt or override the sampler.

Usage:

```
RandomEffectsModel$set_working_parameter(value)
```

Arguments:

`value` Parameter input

Returns: None

Method `set_group_parameters()`: Set value for the "group parameters." This is typically used for initialization, but could also be used to interrupt or override the sampler.

Usage:

```
RandomEffectsModel$set_group_parameters(value)
```

Arguments:

`value` Parameter input

Returns: None

Method `set_working_parameter_cov()`: Set value for the working parameter covariance. This is typically used for initialization, but could also be used to interrupt or override the sampler.

Usage:

```
RandomEffectsModel$set_working_parameter_cov(value)
```

Arguments:

`value` Parameter input

Returns: None

Method `set_group_parameter_cov()`: Set value for the group parameter covariance. This is typically used for initialization, but could also be used to interrupt or override the sampler.

Usage:

```
RandomEffectsModel$set_group_parameter_cov(value)
```

Arguments:

`value` Parameter input

Returns: None

Method `set_variance_prior_shape()`: Set shape parameter for the group parameter variance prior.

Usage:

```
RandomEffectsModel$set_variance_prior_shape(value)
```

Arguments:

`value` Parameter input

Returns: None

Method `set_variance_prior_scale()`: Set shape parameter for the group parameter variance prior.

Usage:

```
RandomEffectsModel$set_variance_prior_scale(value)
```

Arguments:

`value` Parameter input

Returns: None

RandomEffectStateManagement

Random Effects State Management Routines

Description

A forest sampler features two types of state: ephemeral and persistent. Persistent state includes objects like ForestSamples and RandomEffectSamples which constitute part of the final sampled model. Ephemeral state supports the sampling computations, but is not retained after the sampler finishes.

The two primary random-effects-based bits of ephemeral state are the RandomEffectsModel and RandomEffectsTracker classes, which represent the current state of a random effects model and its corresponding tracking data structures.

In a linear sampling loop, this ephemeral state is updated with each iteration of the sampler and any retained forests are copied to a RandomEffectSamples object. However, in multi-chain settings, the state of a random effects model must typically be "reset" at the beginning of a new chain. These function enable this process by synchronizing the state of a RandomEffectsModel and RandomEffectsTracker with a corresponding element of a RandomEffectSamples object, or by resetting both to their default (root) state.

`resetRandomEffectsModel` resets a `RandomEffectsModel` object based on the parameters indexed by `sample_num` in a `RandomEffectsSamples` object. `resetRandomEffectsTracker` resets a `RandomEffectsTracker` object based on the parameters indexed by `sample_num` in a `RandomEffectsSamples` object.

`rootResetRandomEffectsModel` resets a `RandomEffectsModel` object to its "default" state. `rootResetRandomEffectsTrac` resets a `RandomEffectsTracker` object to its "default" state.

These functions are intended for advanced use cases in which users require detailed control of sampling algorithms and data structures. Minimal input validation and error checks are performed – users are responsible for providing the correct inputs. For tutorials on the "proper" usage of the stochtree's advanced workflow, we provide several vignettes at https://stochtree.ai/

Usage

```
resetRandomEffectsModel(rfx_model, rfx_samples, sample_num, sigma_alpha_init)

resetRandomEffectsTracker(
  rfx_tracker,
  rfx_model,
  rfx_dataset,
  residual,
  rfx_samples
)

rootResetRandomEffectsModel(
  rfx_model,
  alpha_init,
```

```
  xi_init,
  sigma_alpha_init,
  sigma_xi_init,
  sigma_xi_shape,
  sigma_xi_scale
)

rootResetRandomEffectsTracker(rfx_tracker, rfx_model, rfx_dataset, residual)
```

Arguments

rfx_model	Object of type RandomEffectsModel.
rfx_samples	Object of type RandomEffectSamples.
sample_num	Index of sample stored in rfx_samples from which to reset the state of a random effects model. Zero-indexed, so resetting based on the first sample would require setting sample_num = 0.
sigma_alpha_init	Initial value of the "working parameter" scale parameter.
rfx_tracker	Object of type RandomEffectsTracker.
rfx_dataset	Object of type RandomEffectsDataset.
residual	Object of type Outcome.
alpha_init	Initial value of the "working parameter".
xi_init	Initial value of the "group parameters".
sigma_xi_init	Initial value of the "group parameters" scale parameter.
sigma_xi_shape	Shape parameter for the inverse gamma variance model on the group parameters.
sigma_xi_scale	Scale parameter for the inverse gamma variance model on the group parameters.

Value

All four functions have no return type and operate in-place on the relevant RandomEffectsModel or RandomEffectsTracker objects

Examples

```
n <- 100
p <- 10
rfx_group_ids <- sample(1:2, size = n, replace = TRUE)
rfx_basis <- matrix(rep(1.0, n), ncol=1)
rfx_dataset <- createRandomEffectsDataset(rfx_group_ids, rfx_basis)
y <- (-2*(rfx_group_ids==1)+2*(rfx_group_ids==2)) + rnorm(n)
y_std <- (y-mean(y))/sd(y)
outcome <- createOutcome(y_std)
rng <- createCppRNG(1234)
num_groups <- length(unique(rfx_group_ids))
num_components <- ncol(rfx_basis)
rfx_model <- createRandomEffectsModel(num_components, num_groups)
rfx_tracker <- createRandomEffectsTracker(rfx_group_ids)
```

```
rfx_samples <- createRandomEffectSamples(num_components, num_groups, rfx_tracker)
alpha_init <- rep(1,num_components)
xi_init <- matrix(rep(alpha_init, num_groups),num_components,num_groups)
sigma_alpha_init <- diag(1,num_components,num_components)
sigma_xi_init <- diag(1,num_components,num_components)
sigma_xi_shape <- 1
sigma_xi_scale <- 1
rfx_model$set_working_parameter(alpha_init)
rfx_model$set_group_parameters(xi_init)
rfx_model$set_working_parameter_cov(sigma_alpha_init)
rfx_model$set_group_parameter_cov(sigma_xi_init)
rfx_model$set_variance_prior_shape(sigma_xi_shape)
rfx_model$set_variance_prior_scale(sigma_xi_scale)
for (i in 1:3) {
    rfx_model$sample_random_effect(rfx_dataset=rfx_dataset, residual=outcome,
                                   rfx_tracker=rfx_tracker, rfx_samples=rfx_samples,
                                   keep_sample=TRUE, global_variance=1.0, rng=rng)
}
resetRandomEffectsModel(rfx_model, rfx_samples, 0, 1.0)
resetRandomEffectsTracker(rfx_tracker, rfx_model, rfx_dataset, outcome, rfx_samples)
rootResetRandomEffectsModel(rfx_model, alpha_init, xi_init, sigma_alpha_init,
                            sigma_xi_init, sigma_xi_shape, sigma_xi_scale)
rootResetRandomEffectsTracker(rfx_tracker, rfx_model, rfx_dataset, outcome)
```

`RandomEffectsTracker` *Random Effects Tracker C++ Wrapper*

Description

Class that defines a "tracker" for random effects models, most notably storing the data indices available in each group for quicker posterior computation and sampling of random effects terms. The class stores a mapping from every observation to its group index, a mapping from group indices to the training sample observations available in that group, and predictions for each observation.

This class is intended for advanced use cases in which users require detailed control of sampling algorithms and data structures. Minimal input validation and error checks are performed – users are responsible for providing the correct inputs. For tutorials on the "proper" usage of the stochtree's advanced workflow, we provide several vignettes at https://stochtree.ai/

Public fields

`rfx_tracker_ptr` External pointer to a C++ StochTree::RandomEffectsTracker class

Methods

Public methods:

- `RandomEffectsTracker$new()`

Method `new()`**:** Create a new RandomEffectsTracker object.

Usage:

```
RandomEffectsTracker$new(rfx_group_indices)
```

Arguments:

`rfx_group_indices` Integer indices indicating groups used to define random effects

Returns: A new `RandomEffectsTracker` object.

`sample_without_replacement`
Sample Without Replacement

Description

Draw `sample_size` samples from `population_vector` without replacement, weighted by `sampling_probabilities`

This function is intended for advanced use cases in which users require detailed control of sampling algorithms and data structures. Minimal input validation and error checks are performed – users are responsible for providing the correct inputs. For tutorials on the "proper" usage of the stochtree's advanced workflow, we provide several vignettes at https://stochtree.ai/

Usage

```
sample_without_replacement(
  population_vector,
  sampling_probabilities,
  sample_size
)
```

Arguments

`population_vector`	Vector from which to draw samples.
`sampling_probabilities`	Vector of probabilities of drawing each element of `population_vector`.
`sample_size`	Number of samples to draw from `population_vector`. Must be less than or equal to `length(population_vector)`

Value

Vector of size `sample_size`

Examples

```
a <- as.integer(c(4,3,2,5,1,9,7))
p <- c(0.7,0.2,0.05,0.02,0.01,0.01,0.01)
num_samples <- 5
sample_without_replacement(a, p, num_samples)
```

`sampleBARTPosteriorPredictive` *Sample BART Posterior Predictive*

Description

Sample from the posterior predictive distribution for outcomes modeled by BART

Usage

```
sampleBARTPosteriorPredictive(
  model_object,
  X = NULL,
  leaf_basis = NULL,
  rfx_group_ids = NULL,
  rfx_basis = NULL,
  num_draws_per_sample = NULL
)
```

Arguments

`model_object`	A fitted BART model object of class `bartmodel`.
`X`	A matrix or data frame of covariates. Required if the BART model depends on covariates (e.g., contains a mean or variance forest).
`leaf_basis`	A matrix of bases for mean forest models with regression defined in the leaves. Required for "leaf regression" models.
`rfx_group_ids`	A vector of group IDs for random effects model. Required if the BART model includes random effects.
`rfx_basis`	A matrix of bases for random effects model. Required if the BART model includes random effects.
`num_draws_per_sample`	The number of posterior predictive samples to draw for each posterior sample. Defaults to a heuristic based on the number of samples in a BART model (i.e. if the BART model has >1000 draws, we use 1 draw from the likelihood per sample, otherwise we upsample to ensure intervals are based on at least 1000 posterior predictive draws).

Value

Array of posterior predictive samples with dimensions (num_observations, num_posterior_samples, num_draws_per_sample) if num_draws_per_sample > 1, otherwise (num_observations, num_posterior_samples).

Examples

```
n <- 100
p <- 5
X <- matrix(rnorm(n * p), nrow = n, ncol = p)
y <- 2 * X[,1] + rnorm(n)
bart_model <- bart(y_train = y, X_train = X)
ppd_samples <- sampleBARTPosteriorPredictive(
  model_object = bart_model, X = X
)
```

---

sampleBCFPosteriorPredictive

Sample BCF Posterior Predictive

Description

Sample from the posterior predictive distribution for outcomes modeled by BCF

Usage

```
sampleBCFPosteriorPredictive(
  model_object,
  X = NULL,
  Z = NULL,
  propensity = NULL,
  rfx_group_ids = NULL,
  rfx_basis = NULL,
  num_draws_per_sample = NULL
)
```

Arguments

`model_object`	A fitted BCF model object of class `bcfmodel`.
`X`	A matrix or data frame of covariates.
`Z`	A vector or matrix of treatment assignments.
`propensity`	(Optional) A vector or matrix of propensity scores. Required if the underlying model depends on user-provided propensities.
`rfx_group_ids`	(Optional) A vector of group IDs for random effects model. Required if the BCF model includes random effects.
`rfx_basis`	(Optional) A matrix of bases for random effects model. Required if the BCF model includes random effects.
`num_draws_per_sample`	(Optional) The number of samples to draw from the likelihood for each draw of the posterior. Defaults to a heuristic based on the number of samples in a BCF model (i.e. if the BCF model has >1000 draws, we use 1 draw from the likelihood per sample, otherwise we upsample to ensure at least 1000 posterior predictive draws).

Value

Array of posterior predictive samples with dimensions (num_observations, num_posterior_samples, num_draws_per_sample) if num_draws_per_sample > 1, otherwise (num_observations, num_posterior_samples).

Examples

```
n <- 100
p <- 5
X <- matrix(rnorm(n * p), nrow = n, ncol = p)
pi_X <- pnorm(X[,1] / 2)
Z <- rbinom(n, 1, pi_X)
y <- 2 * X[,2] + 0.5 * X[,2] * Z + rnorm(n)
bcf_model <- bcf(X_train = X, Z_train = Z, y_train = y, propensity_train = pi_X)
ppd_samples <- sampleBCFPosteriorPredictive(
  model_object = bcf_model, X = X,
  Z = Z, propensity = pi_X
)
```

---

sampleGlobalErrorVarianceOneIteration
Sample Global Error Variance

Description

Sample one iteration of the (inverse gamma) global variance model

This function is intended for advanced use cases in which users require detailed control of sampling algorithms and data structures. Minimal input validation and error checks are performed – users are responsible for providing the correct inputs. For tutorials on the "proper" usage of the stochtree's advanced workflow, we provide several vignettes at https://stochtree.ai/

Usage

```
sampleGlobalErrorVarianceOneIteration(residual, dataset, rng, a, b)
```

Arguments

residual	Outcome class
dataset	ForestDataset class
rng	C++ random number generator
a	Global variance shape parameter
b	Global variance scale parameter

Value

None

Examples

```
X <- matrix(runif(10*100), ncol = 10)
y <- -5 + 10*(X[,1] > 0.5) + rnorm(100)
y_std <- (y-mean(y))/sd(y)
forest_dataset <- createForestDataset(X)
outcome <- createOutcome(y_std)
rng <- createCppRNG(1234)
a <- 1.0
b <- 1.0
sigma2 <- sampleGlobalErrorVarianceOneIteration(outcome, forest_dataset, rng, a, b)
```

---

sampleLeafVarianceOneIteration

Sample Leaf Scale

Description

Sample one iteration of the leaf parameter variance model (only for univariate basis and constant leaf!)

This function is intended for advanced use cases in which users require detailed control of sampling algorithms and data structures. Minimal input validation and error checks are performed – users are responsible for providing the correct inputs. For tutorials on the "proper" usage of the stochtree's advanced workflow, we provide several vignettes at https://stochtree.ai/

Usage

```
sampleLeafVarianceOneIteration(forest, rng, a, b)
```

Arguments

forest	C++ forest
rng	C++ random number generator
a	Leaf variance shape parameter
b	Leaf variance scale parameter

Value

None

Examples

```
num_trees <- 100
leaf_dimension <- 1
is_leaf_constant <- TRUE
is_exponentiated <- FALSE
active_forest <- createForest(num_trees, leaf_dimension, is_leaf_constant, is_exponentiated)
rng <- createCppRNG(1234)
```

```
a <- 1.0
b <- 1.0
tau <- sampleLeafVarianceOneIteration(active_forest, rng, a, b)
```

`summary.bartmodel` *Summarize BART Model Fit and Parameters*

Description

Summarize a BART fit with a description of the model that was fit and numeric summaries of any sampled quantities.

Usage

```
## S3 method for class 'bartmodel'
summary(object, ...)
```

Arguments

`object`	The BART model object
`...`	Additional arguments

Value

BART model object unchanged after summarizing

`summary.bcfmodel` *Summarize BCF Model*

Description

Summarize a BCF fit with a description of the model that was fit and numeric summaries of any sampled quantities.

Usage

```
## S3 method for class 'bcfmodel'
summary(object, ...)
```

Arguments

`object`	The BCF model object
`...`	Additional arguments

Value

BCF model object unchanged after summarizing

Index

bart, 5
bart(), *51*
BARTSerialization, 11, *51*, *92*, *116*
bcf, 13
bcf(), *51*
BCFSerialization, 21, *51*, *92*, *116*

calibrateInverseGammaErrorVariance, 23
computeBARTPosteriorInterval, 25
computeBCFPosteriorInterval, 26
computeContrastBARTModel, 28
computeContrastBCFModel, 30
computeForestLeafIndices
        *(*ForestKernelComputation*)*, 65
computeForestLeafVariances
        *(*ForestKernelComputation*)*, 65
computeForestMaxLeafIndex
        *(*ForestKernelComputation*)*, 65
CppJson, 33, *92*, *100*, *116*
CppRNG, 40
createBARTModelFromCombinedJson
        *(*BARTSerialization*)*, 11
createBARTModelFromCombinedJsonString
        *(*BARTSerialization*)*, 11
createBARTModelFromJson
        *(*BARTSerialization*)*, 11
createBARTModelFromJsonFile
        *(*BARTSerialization*)*, 11
createBARTModelFromJsonString
        *(*BARTSerialization*)*, 11
createBCFModelFromCombinedJson
        *(*BCFSerialization*)*, 21
createBCFModelFromCombinedJsonString
        *(*BCFSerialization*)*, 21
createBCFModelFromJson
        *(*BCFSerialization*)*, 21
createBCFModelFromJsonFile
        *(*BCFSerialization*)*, 21
createBCFModelFromJsonString
        *(*BCFSerialization*)*, 21
createCppJson *(*JSONCreation*)*, 100
createCppJsonFile *(*JSONCreation*)*, 100
createCppJsonString *(*JSONCreation*)*, 100
createCppRNG, 40
createForest, 41
createForestDataset, 42
createForestModel, 42
createForestModelConfig, 43
createForestSamples, 45
createGlobalModelConfig, 46
createOutcome, 47
createPreprocessorFromJson
        *(*DataPreprocessing*)*, 50
createPreprocessorFromJsonString
        *(*DataPreprocessing*)*, 50
createRandomEffectSamples, 47
createRandomEffectsDataset, 48
createRandomEffectsModel, 49
createRandomEffectsTracker, 50

DataPreprocessing, 50

extractParameter, 52
extractParameter.bartmodel, 53
extractParameter.bcfmodel, 54

Forest, 56, *93*, *94*
ForestDataset, 61
ForestKernelComputation, 65
ForestModel, 67, *93*, *94*
ForestModelConfig, 71
ForestSamples, 78, *92*, *93*, *123*
ForestSamplesSerialization, 92
ForestStateManagement, 93

getRandomEffectSamples, 95
getRandomEffectSamples.bartmodel, 96
getRandomEffectSamples.bcfmodel, 97
GlobalModelConfig, 99

JSONCreation, 100

loadForestContainerCombinedJson
(ForestSamplesSerialization), 92
loadForestContainerCombinedJsonString
(ForestSamplesSerialization), 92
loadForestContainerJson
(ForestSamplesSerialization), 92
loadRandomEffectSamplesCombinedJson
(RandomEffectSamplesSerialization), 116
loadRandomEffectSamplesCombinedJsonString
(RandomEffectSamplesSerialization), 116
loadRandomEffectSamplesJson
(RandomEffectSamplesSerialization), 116
loadScalarJson, 101
loadVectorJson, 102

Outcome, 102
OutcomeModel, 104

plot.bartmodel, 105
plot.bcfmodel, 105
predict.bartmodel, 106
predict.bcfmodel, 107
preprocessPredictionData
(DataPreprocessing), 50
preprocessTrainData
(DataPreprocessing), 50
print.bartmodel, 110
print.bcfmodel, 110
print.ForestSamples, 111
print.RandomEffectSamples, 111

RandomEffectSamples, *93*, 112, *116*, *123*
RandomEffectSamplesSerialization, 116
RandomEffectsDataset, 117
RandomEffectsModel, 120, *123, 124*
RandomEffectStateManagement, 123
RandomEffectsTracker, *123, 124*, 125
resetActiveForest
(ForestStateManagement), 93
resetForestModel
(ForestStateManagement), 93
resetRandomEffectsModel
(RandomEffectStateManagement), 123
resetRandomEffectsTracker
(RandomEffectStateManagement), 123
rootResetRandomEffectsModel
(RandomEffectStateManagement), 123
rootResetRandomEffectsTracker
(RandomEffectStateManagement), 123

sample_without_replacement, 126
sampleBARTPosteriorPredictive, 127
sampleBCFPosteriorPredictive, 128
sampleGlobalErrorVarianceOneIteration, 129
sampleLeafVarianceOneIteration, 130
saveBARTModelToJson
(BARTSerialization), 11
saveBARTModelToJsonFile
(BARTSerialization), 11
saveBARTModelToJsonString
(BARTSerialization), 11
saveBCFModelToJson (BCFSerialization), 21
saveBCFModelToJsonFile
(BCFSerialization), 21
saveBCFModelToJsonString
(BCFSerialization), 21
savePreprocessorToJson
(DataPreprocessing), 50
savePreprocessorToJsonString
(DataPreprocessing), 50
stochtree (stochtree-package), 4
stochtree-package, 4
summary.bartmodel, 131
summary.bcfmodel, 131