Package 'gpboost'

August 26, 2024

Type Package

Title Combining Tree-Boosting with Gaussian Process and Mixed Effects Models

Version 1.5.1.2

Date 2024-08-26

Description An R package that allows for combining tree-boosting with Gaussian process and mixed effects models. It also allows for independently doing tree-boosting as well as inference and prediction for Gaussian process and mixed effects models. See <https://github.com/fabsig/GPBoost> for more information on the software and Sigrist (2022, JMLR) <https://www.jmlr.org/papers/v23/20-322.html> and Sigrist (2023, TPAMI) <doi:10.1109/TPAMI.2022.3168152> for more information on the methodology.

Encoding UTF-8

License Apache License (== 2.0) | file LICENSE

URL https://github.com/fabsig/GPBoost

BugReports https://github.com/fabsig/GPBoost/issues

NeedsCompilation yes

Biarch true

Suggests testthat

Depends R (>= 3.5), R6 (>= 2.0)

Imports data.table (>= 1.9.6), graphics, RJSONIO, Matrix (>= 1.1-0), methods, utils

SystemRequirements C++17

RoxygenNote 6.0.1

Author Fabio Sigrist [aut, cre],
Tim Gyger [aut],
Pascal Kuendig [aut],
Benoit Jacob [cph],
Gael Guennebaud [cph],
Nicolas Carre [cph],

Pierre Zoppitelli [cph],
Gauthier Brun [cph],
Jean Ceccato [cph],
Jitse Niesen [cph],
Other authors of Eigen for the included version of Eigen [ctb, cph],
Timothy A. Davis [cph],
Guolin Ke [ctb],
Damien Soukhavong [ctb],
James Lamb [ctb],
Other authors of LightGBM for the included version of LightGBM [ctb],
Microsoft Corporation [cph],
Dropbox, Inc. [cph],
Jay Loden [cph],
Dave Daeschler [cph],
Giampaolo Rodola [cph],
Alberto Ferreira [ctb],
Daniel Lemire [ctb],
Victor Zverovich [cph],
IBM Corporation [ctb],
Keith O'Hara [cph],
Stephen L. Moshier [cph],
Jorge Nocedal [cph],
Naoaki Okazaki [cph],
Yixuan Qiu [cph],
Dirk Toewe [cph]

Maintainer Fabio Sigrist <fabiosigrist@gmail.com>

Repository CRAN

Date/Publication 2024-08-26 18:20:02 UTC

Contents

`agaricus.test` *Test part from Mushroom Data Set*

Description

This data set is originally from the Mushroom data set, UCI Machine Learning Repository. This data set includes the following fields:

- `label`: the label for each record
- `data`: a sparse Matrix of `dgCMatrix` class, with 126 columns.

Usage

```
data(agaricus.test)
```

Format

A list containing a label vector, and a dgCMatrix object with 1611 rows and 126 variables

References

https://archive.ics.uci.edu/ml/datasets/Mushroom

Bache, K. & Lichman, M. (2013). UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science.

`agaricus.train` *Training part from Mushroom Data Set*

Description

This data set is originally from the Mushroom data set, UCI Machine Learning Repository. This data set includes the following fields:

- `label`: the label for each record
- `data`: a sparse Matrix of `dgCMatrix` class, with 126 columns.

Usage

```
data(agaricus.train)
```

Format

A list containing a label vector, and a dgCMatrix object with 6513 rows and 127 variables

References

https://archive.ics.uci.edu/ml/datasets/Mushroom

Bache, K. & Lichman, M. (2013). UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science.

---

bank *Bank Marketing Data Set*

Description

This data set is originally from the Bank Marketing data set, UCI Machine Learning Repository.

It contains only the following: bank.csv with 10 randomly selected from 3 (older version of this dataset with less inputs).

Usage

```
data(bank)
```

Format

A data.table with 4521 rows and 17 variables

References

http://archive.ics.uci.edu/ml/datasets/Bank+Marketing

S. Moro, P. Cortez and P. Rita. (2014) A Data-Driven Approach to Predict the Success of Bank Telemarketing. Decision Support Systems

---

coords *Example data for the GPBoost package*

Description

A matrix with spatial coordinates for the example data of the GPBoost package

Usage

```
data(GPBoost_data)
```

coords_test *Example data for the GPBoost package*

Description

A matrix with spatial coordinates for predictions for the example data of the GPBoost package

Usage

```
data(GPBoost_data)
```

dim.gpb.Dataset *Dimensions of an* gpb.Dataset

Description

Returns a vector of numbers of rows and of columns in an gpb.Dataset.

Usage

```
## S3 method for class 'gpb.Dataset'
dim(x, ...)
```

Arguments

x	Object of class gpb.Dataset
...	other parameters

Details

Note: since nrow and ncol internally use dim, they can also be directly used with an gpb.Dataset object.

Value

a vector of numbers of rows and of columns

Examples

```
data(agaricus.train, package = "gpboost")
train <- agaricus.train
dtrain <- gpb.Dataset(train$data, label = train$label)

stopifnot(nrow(dtrain) == nrow(train$data))
stopifnot(ncol(dtrain) == ncol(train$data))
stopifnot(all(dim(dtrain) == dim(train$data)))
```

dimnames.gpb.Dataset *Handling of column names of* gpb.Dataset

Description

Only column names are supported for gpb.Dataset, thus setting of row names would have no effect and returned row names would be NULL.

Usage

```
## S3 method for class 'gpb.Dataset'
dimnames(x)

## S3 replacement method for class 'gpb.Dataset'
dimnames(x) <- value
```

Arguments

x	object of class gpb.Dataset
value	a list of two elements: the first one is ignored and the second one is column names

Details

Generic dimnames methods are used by colnames. Since row names are irrelevant, it is recommended to use colnames directly.

Value

A list with the dimension names of the dataset

A list with the dimension names of the dataset

Examples

```
data(agaricus.train, package = "gpboost")
train <- agaricus.train
dtrain <- gpb.Dataset(train$data, label = train$label)
gpb.Dataset.construct(dtrain)
dimnames(dtrain)
colnames(dtrain)
colnames(dtrain) <- make.names(seq_len(ncol(train$data)))
print(dtrain, verbose = TRUE)
```

`fit` *Generic 'fit' method for a* `GPModel`

Description

Generic 'fit' method for a `GPModel`

Usage

```
fit(gp_model, y, X, params, offset = NULL, fixed_effects = NULL)
```

Arguments

`gp_model`	a `GPModel`
`y`	A `vector` with response variable data
`X`	A `matrix` with numeric covariate data for the fixed effects linear regression term (if there is one)
`params`	A `list` with parameters for the estimation / optimization

- optimizer_cov: `string` (default = "lbfgs"). Optimizer used for estimating covariance parameters. Options: "gradient_descent", "lbfgs", "fisher_scoring", "newton", "nelder_mead", "adam". If there are additional auxiliary parameters for non-Gaussian likelihoods, 'optimizer_cov' is also used for those
- optimizer_coef: `string` (default = "wls" for Gaussian likelihoods and "lbfgs" for other likelihoods). Optimizer used for estimating linear regression coefficients, if there are any (for the GPBoost algorithm there are usually none). Options: "gradient_descent", "lbfgs", "wls", "nelder_mead", "adam". Gradient descent steps are done simultaneously with gradient descent steps for the covariance parameters. "wls" refers to doing coordinate descent for the regression coefficients using weighted least squares. If 'optimizer_cov' is set to "nelder_mead", "lbfgs", or "adam", 'optimizer_coef' is automatically also set to the same value.
- maxit: `integer` (default = 1000). Maximal number of iterations for optimization algorithm
- delta_rel_conv: `numeric` (default = 1E-6 except for "nelder_mead" for which the default is 1E-8). Convergence tolerance. The algorithm stops if the relative change in either the (approximate) log-likelihood or the parameters is below this value. For "adam", the L2 norm of the gradient is used instead of the relative change in the log-likelihood. If < 0, internal default values are used
- convergence_criterion: `string` (default = "relative_change_in_log_likelihood"). The convergence criterion used for terminating the optimization algorithm. Options: "relative_change_in_log_likelihood" or "relative_change_in_parameters"
- init_coef: `vector` with `numeric` elements (default = NULL). Initial values for the regression coefficients (if there are any, can be NULL)

- init_cov_pars: `vector` with `numeric` elements (default = NULL). Initial values for covariance parameters of Gaussian process and random effects (can be NULL). The order it the same as the order of the parameters in the summary function: first is the error variance (only for "gaussian" likelihood), next follow the variances of the grouped random effects (if there are any, in the order provided in 'group_data'), and then follow the marginal variance and the range of the Gaussian process. If there are multiple Gaussian processes, then the variances and ranges follow alternatingly. If 'init_cov_pars = NULL', an internal choice is used that depends on the likelihood and the random effects type and covariance function. If you select the option 'trace = TRUE' in the 'params' argument, you will see the first initial covariance parameters in iteration 0.
- lr_coef: `numeric` (default = 0.1). Learning rate for fixed effect regression coefficients if gradient descent is used
- lr_cov: `numeric` (default = 0.1 for "gradient_descent" and 1. otherwise). Initial learning rate for covariance parameters if a gradient-based optimization method is used
  - If lr_cov < 0, internal default values are used (0.1 for "gradient_descent" and 1. otherwise)
  - If there are additional auxiliary parameters for non-Gaussian likelihoods, 'lr_cov' is also used for those
  - For "lbfgs", this is divided by the norm of the gradient in the first iteration
- use_nesterov_acc: `boolean` (default = TRUE). If TRUE Nesterov acceleration is used. This is used only for gradient descent
- acc_rate_coef: `numeric` (default = 0.5). Acceleration rate for regression coefficients (if there are any) for Nesterov acceleration
- acc_rate_cov: `numeric` (default = 0.5). Acceleration rate for covariance parameters for Nesterov acceleration
- momentum_offset: `integer` (Default = 2). Number of iterations for which no momentum is applied in the beginning.
- trace: `boolean` (default = FALSE). If TRUE, information on the progress of the parameter optimization is printed
- std_dev: `boolean` (default = TRUE). If TRUE, approximate standard deviations are calculated for the covariance and linear regression parameters (= square root of diagonal of the inverse Fisher information for Gaussian likelihoods and square root of diagonal of a numerically approximated inverse Hessian for non-Gaussian likelihoods)
- init_aux_pars: `vector` with `numeric` elements (default = NULL). Initial values for additional parameters for non-Gaussian likelihoods (e.g., shape parameter of a gamma or negative_binomial likelihood)
- estimate_aux_pars: `boolean` (default = TRUE). If TRUE, additional parameters for non-Gaussian likelihoods are also estimated (e.g., shape parameter of a gamma or negative_binomial likelihood)
- cg_max_num_it: `integer` (default = 1000). Maximal number of iterations for conjugate gradient algorithms

- cg_max_num_it_tridiag: `integer` (default = 1000). Maximal number of iterations for conjugate gradient algorithm when being run as Lanczos algorithm for tridiagonalization
- cg_delta_conv: `numeric` (default = 1E-2). Tolerance level for L2 norm of residuals for checking convergence in conjugate gradient algorithm when being used for parameter estimation
- num_rand_vec_trace: `integer` (default = 50). Number of random vectors (e.g., Rademacher) for stochastic approximation of the trace of a matrix
- reuse_rand_vec_trace: `boolean` (default = TRUE). If true, random vectors (e.g., Rademacher) for stochastic approximations of the trace of a matrix are sampled only once at the beginning of the parameter estimation and reused in later trace approximations. Otherwise they are sampled every time a trace is calculated
- seed_rand_vec_trace: `integer` (default = 1). Seed number to generate random vectors (e.g., Rademacher)
- piv_chol_rank: `integer` (default = 50). Rank of the pivoted Cholesky decomposition used as preconditioner in conjugate gradient algorithms
- cg_preconditioner_type: `string`. Type of preconditioner used for conjugate gradient algorithms.
  - Options for non-Gaussian likelihoods and gp_approx = "vecchia":
    - "Sigma_inv_plus_BtWB" (= default): (B^T * (D^-1 + W) * B) as preconditioner for inverting (B^T * D^-1 * B + W), where B^T * D^-1 * B approx= Sigma^-1
  - "piv_chol_on_Sigma": (Lk * Lk^T + W^-1) as preconditioner for inverting (B^-1 * D * B^-T + W^-1), where Lk is a low-rank pivoted Cholesky approximation for Sigma and B^-1 * D * B^-T approx= Sigma
  - Options for likelihood = "gaussian" and gp_approx = "full_scale_tapering":
    - "predictive_process_plus_diagonal" (= default): predictive process preconditiioner
    - "none": no preconditioner

`offset` A `numeric vector` with additional fixed effects contributions that are added to the linear predictor (= offset). The length of this vector needs to equal the number of training data points.

`fixed_effects` This is discontinued. Use the renamed equivalent argument `offset` instead

Author(s)

Fabio Sigrist

`fit.GPModel` *Fits a* `GPModel`

Description

Estimates the parameters of a `GPModel` by maximizing the marginal likelihood

Usage

```
## S3 method for class 'GPModel'
fit(gp_model, y, X = NULL, params = list(),
  offset = NULL, fixed_effects = NULL)
```

Arguments

gp_model	a GPModel
y	A vector with response variable data
X	A matrix with numeric covariate data for the fixed effects linear regression term (if there is one)
params	A list with parameters for the estimation / optimization

- optimizer_cov: string (default = "lbfgs"). Optimizer used for estimating covariance parameters. Options: "gradient_descent", "lbfgs", "fisher_scoring", "newton", "nelder_mead", "adam". If there are additional auxiliary parameters for non-Gaussian likelihoods, 'optimizer_cov' is also used for those
- optimizer_coef: string (default = "wls" for Gaussian likelihoods and "lbfgs" for other likelihoods). Optimizer used for estimating linear regression coefficients, if there are any (for the GPBoost algorithm there are usually none). Options: "gradient_descent", "lbfgs", "wls", "nelder_mead", "adam". Gradient descent steps are done simultaneously with gradient descent steps for the covariance parameters. "wls" refers to doing coordinate descent for the regression coefficients using weighted least squares. If 'optimizer_cov' is set to "nelder_mead", "lbfgs", or "adam", 'optimizer_coef' is automatically also set to the same value.
- maxit: integer (default = 1000). Maximal number of iterations for optimization algorithm
- delta_rel_conv: numeric (default = 1E-6 except for "nelder_mead" for which the default is 1E-8). Convergence tolerance. The algorithm stops if the relative change in either the (approximate) log-likelihood or the parameters is below this value. For "adam", the L2 norm of the gradient is used instead of the relative change in the log-likelihood. If < 0, internal default values are used
- convergence_criterion: string (default = "relative_change_in_log_likelihood"). The convergence criterion used for terminating the optimization algorithm. Options: "relative_change_in_log_likelihood" or "relative_change_in_parameters"
- init_coef: vector with numeric elements (default = NULL). Initial values for the regression coefficients (if there are any, can be NULL)
- init_cov_pars: vector with numeric elements (default = NULL). Initial values for covariance parameters of Gaussian process and random effects (can be NULL). The order it the same as the order of the parameters in the summary function: first is the error variance (only for "gaussian" likelihood), next follow the variances of the grouped random effects (if there are any, in the order provided in 'group_data'), and then follow the marginal variance and the range of the Gaussian process. If there are multiple Gaussian processes, then the variances and ranges follow alternatingly. If 'init_cov_pars

= NULL', an internal choice is used that depends on the likelihood and the random effects type and covariance function. If you select the option 'trace = TRUE' in the 'params' argument, you will see the first initial covariance parameters in iteration 0.

- lr_coef: `numeric` (default = 0.1). Learning rate for fixed effect regression coefficients if gradient descent is used
- lr_cov: `numeric` (default = 0.1 for "gradient_descent" and 1. otherwise). Initial learning rate for covariance parameters if a gradient-based optimization method is used
  - If lr_cov < 0, internal default values are used (0.1 for "gradient_descent" and 1. otherwise)
  - If there are additional auxiliary parameters for non-Gaussian likelihoods, 'lr_cov' is also used for those
  - For "lbfgs", this is divided by the norm of the gradient in the first iteration
- use_nesterov_acc: `boolean` (default = TRUE). If TRUE Nesterov acceleration is used. This is used only for gradient descent
- acc_rate_coef: `numeric` (default = 0.5). Acceleration rate for regression coefficients (if there are any) for Nesterov acceleration
- acc_rate_cov: `numeric` (default = 0.5). Acceleration rate for covariance parameters for Nesterov acceleration
- momentum_offset: `integer` (Default = 2). Number of iterations for which no momentum is applied in the beginning.
- trace: `boolean` (default = FALSE). If TRUE, information on the progress of the parameter optimization is printed
- std_dev: `boolean` (default = TRUE). If TRUE, approximate standard deviations are calculated for the covariance and linear regression parameters (= square root of diagonal of the inverse Fisher information for Gaussian likelihoods and square root of diagonal of a numerically approximated inverse Hessian for non-Gaussian likelihoods)
- init_aux_pars: `vector` with `numeric` elements (default = NULL). Initial values for additional parameters for non-Gaussian likelihoods (e.g., shape parameter of a gamma or negative_binomial likelihood)
- estimate_aux_pars: `boolean` (default = TRUE). If TRUE, additional parameters for non-Gaussian likelihoods are also estimated (e.g., shape parameter of a gamma or negative_binomial likelihood)
- cg_max_num_it: `integer` (default = 1000). Maximal number of iterations for conjugate gradient algorithms
- cg_max_num_it_tridiag: `integer` (default = 1000). Maximal number of iterations for conjugate gradient algorithm when being run as Lanczos algorithm for tridiagonalization
- cg_delta_conv: `numeric` (default = 1E-2). Tolerance level for L2 norm of residuals for checking convergence in conjugate gradient algorithm when being used for parameter estimation
- num_rand_vec_trace: `integer` (default = 50). Number of random vectors (e.g., Rademacher) for stochastic approximation of the trace of a matrix

- reuse_rand_vec_trace: `boolean` (default = TRUE). If true, random vectors (e.g., Rademacher) for stochastic approximations of the trace of a matrix are sampled only once at the beginning of the parameter estimation and reused in later trace approximations. Otherwise they are sampled every time a trace is calculated
- seed_rand_vec_trace: `integer` (default = 1). Seed number to generate random vectors (e.g., Rademacher)
- piv_chol_rank: `integer` (default = 50). Rank of the pivoted Cholesky decomposition used as preconditioner in conjugate gradient algorithms
- cg_preconditioner_type: `string`. Type of preconditioner used for conjugate gradient algorithms.
  - Options for non-Gaussian likelihoods and gp_approx = "vecchia":
    - "Sigma_inv_plus_BtWB" (= default): (B^T * (D^-1 + W) * B) as preconditioner for inverting (B^T * D^-1 * B + W), where B^T * D^-1 * B approx= Sigma^-1
  - "piv_chol_on_Sigma": (Lk * Lk^T + W^-1) as preconditioner for inverting (B^-1 * D * B^-T + W^-1), where Lk is a low-rank pivoted Cholesky approximation for Sigma and B^-1 * D * B^-T approx= Sigma
  - Options for likelihood = "gaussian" and gp_approx = "full_scale_tapering":
    - "predictive_process_plus_diagonal" (= default): predictive process preconditiioner
    - "none": no preconditioner

`offset` A `numeric vector` with additional fixed effects contributions that are added to the linear predictor (= offset). The length of this vector needs to equal the number of training data points.

`fixed_effects` This is discontinued. Use the renamed equivalent argument `offset` instead

Value

A fitted `GPModel`

Author(s)

Fabio Sigrist

Examples

```
# See https://github.com/fabsig/GPBoost/tree/master/R-package for more examples


data(GPBoost_data, package = "gpboost")
# Add intercept column
X1 <- cbind(rep(1,dim(X)[1]),X)
X_test1 <- cbind(rep(1,dim(X_test)[1]),X_test)

#--------------------Grouped random effects model: single-level random effect----------------
gp_model <- GPModel(group_data = group_data[,1], likelihood="gaussian")
```

```
fit(gp_model, y = y, X = X1, params = list(std_dev = TRUE))
summary(gp_model)
# Make predictions
pred <- predict(gp_model, group_data_pred = group_data_test[,1],
                X_pred = X_test1, predict_var = TRUE)
pred$mu # Predicted mean
pred$var # Predicted variances
# Also predict covariance matrix
pred <- predict(gp_model, group_data_pred = group_data_test[,1],
                X_pred = X_test1, predict_cov_mat = TRUE)
pred$mu # Predicted mean
pred$cov # Predicted covariance

#--------------------Gaussian process model----------------
gp_model <- GPModel(gp_coords = coords, cov_function = "exponential",
                    likelihood="gaussian")
fit(gp_model, y = y, X = X1, params = list(std_dev = TRUE))
summary(gp_model)
# Make predictions
pred <- predict(gp_model, gp_coords_pred = coords_test,
                X_pred = X_test1, predict_cov_mat = TRUE)
pred$mu # Predicted (posterior) mean of GP
pred$cov # Predicted (posterior) covariance matrix of GP
```

`fitGPModel` *Fits a* `GPModel`

Description

Estimates the parameters of a `GPModel` by maximizing the marginal likelihood

Usage

```
fitGPModel(likelihood = "gaussian", group_data = NULL,
  group_rand_coef_data = NULL, ind_effect_group_rand_coef = NULL,
  drop_intercept_group_rand_effect = NULL, gp_coords = NULL,
  gp_rand_coef_data = NULL, cov_function = "exponential",
  cov_fct_shape = 0.5, gp_approx = "none", cov_fct_taper_range = 1,
  cov_fct_taper_shape = 0, num_neighbors = 20L,
  vecchia_ordering = "random", ind_points_selection = "kmeans++",
  num_ind_points = 500L, cover_tree_radius = 1,
  matrix_inversion_method = "cholesky", seed = 0L, cluster_ids = NULL,
  free_raw_data = FALSE, y, X = NULL, params = list(),
  vecchia_approx = NULL, vecchia_pred_type = NULL,
  num_neighbors_pred = NULL, offset = NULL, fixed_effects = NULL)
```

Arguments

`likelihood` A `string` specifying the likelihood function (distribution) of the response variable. Available options:

- "gaussian"
- "bernoulli_probit": binary data with Bernoulli likelihood and a probit link function
- "bernoulli_logit": binary data with Bernoulli likelihood and a logit link function
- "gamma": gamma distribution with a with log link function
- "poisson": Poisson distribution with a with log link function
- "negative_binomial": negative binomial distribution with a with log link function
- Note: other likelihoods could be implemented upon request

`group_data` A `vector` or `matrix` whose columns are categorical grouping variables. The elements being group levels defining grouped random effects. The elements of 'group_data' can be integer, double, or character. The number of columns corresponds to the number of grouped (intercept) random effects

`group_rand_coef_data` A `vector` or `matrix` with numeric covariate data for grouped random coefficients

`ind_effect_group_rand_coef` A `vector` with integer indices that indicate the corresponding categorical grouping variable (=columns) in 'group_data' for every covariate in 'group_rand_coef_data'. Counting starts at 1. The length of this index vector must equal the number of covariates in 'group_rand_coef_data'. For instance, c(1,1,2) means that the first two covariates (=first two columns) in 'group_rand_coef_data' have random coefficients corresponding to the first categorical grouping variable (=first column) in 'group_data', and the third covariate (=third column) in 'group_rand_coef_data' has a random coefficient corresponding to the second grouping variable (=second column) in 'group_data'

`drop_intercept_group_rand_effect` A `vector` of type `logical` (boolean). Indicates whether intercept random effects are dropped (only for random coefficients). If drop_intercept_group_rand_effect[k] is TRUE, the intercept random effect number k is dropped / not included. Only random effects with random slopes can be dropped.

`gp_coords` A `matrix` with numeric coordinates (= inputs / features) for defining Gaussian processes

`gp_rand_coef_data` A `vector` or `matrix` with numeric covariate data for Gaussian process random coefficients

`cov_function` A `string` specifying the covariance function for the Gaussian process. Available options:

- "exponential": Exponential covariance function (using the parametrization of Diggle and Ribeiro, 2007)

- "gaussian": Gaussian, aka squared exponential, covariance function (using the parametrization of Diggle and Ribeiro, 2007)
- "matern": Matern covariance function with the smoothness specified by the `cov_fct_shape` parameter (using the parametrization of Rasmussen and Williams, 2006)
- "powered_exponential": powered exponential covariance function with the exponent specified by the `cov_fct_shape` parameter (using the parametrization of Diggle and Ribeiro, 2007)
- "wendland": Compactly supported Wendland covariance function (using the parametrization of Bevilacqua et al., 2019, AOS)
- "matern_space_time": Spatio-temporal Matern covariance function with different range parameters for space and time. Note that the first column in `gp_coords` must correspond to the time dimension
- "matern_ard": anisotropic Matern covariance function with Automatic Relevance Determination (ARD), i.e., with a different range parameter for every coordinate dimension / column of `gp_coords`
- "gaussian_ard": anisotropic Gaussian, aka squared exponential, covariance function with Automatic Relevance Determination (ARD), i.e., with a different range parameter for every coordinate dimension / column of `gp_coords`

`cov_fct_shape` A `numeric` specifying the shape parameter of the covariance function (=smoothness parameter for Matern covariance) This parameter is irrelevant for some covariance functions such as the exponential or Gaussian

`gp_approx` A `string` specifying the large data approximation for Gaussian processes. Available options:

- "none": No approximation
- "vecchia": A Vecchia approximation; see Sigrist (2022, JMLR) for more details
- "tapering": The covariance function is multiplied by a compactly supported Wendland correlation function
- "fitc": Fully Independent Training Conditional approximation aka modified predictive process approximation; see Gyger, Furrer, and Sigrist (2024) for more details
- "full_scale_tapering": A full scale approximation combining an inducing point / predictive process approximation with tapering on the residual process; see Gyger, Furrer, and Sigrist (2024) for more details

`cov_fct_taper_range` A `numeric` specifying the range parameter of the Wendland covariance function and Wendland correlation taper function. We follow the notation of Bevilacqua et al. (2019, AOS)

`cov_fct_taper_shape` A `numeric` specifying the shape (=smoothness) parameter of the Wendland covariance function and Wendland correlation taper function. We follow the notation of Bevilacqua et al. (2019, AOS)

`num_neighbors` An `integer` specifying the number of neighbors for the Vecchia approximation. Note: for prediction, the number of neighbors can be set through the

'num_neighbors_pred' parameter in the 'set_prediction_data' function. By default, num_neighbors_pred = 2 * num_neighbors. Further, the type of Vecchia approximation used for making predictions is set through the 'vecchia_pred_type' parameter in the 'set_prediction_data' function

`vecchia_ordering`
: A `string` specifying the ordering used in the Vecchia approximation. Available options:
  - "none": the default ordering in the data is used
  - "random": a random ordering
  - "time": ordering accorrding to time (only for space-time models)
  - "time_random_space": ordering according to time and randomly for all spatial points with the same time points (only for space-time models)

`ind_points_selection`
: A `string` specifying the method for choosing inducing points Available options:
  - "kmeans++: the k-means++ algorithm
  - "cover_tree": the cover tree algorithm
  - "random": random selection from data points

`num_ind_points`
: An `integer` specifying the number of inducing points / knots for, e.g., a predictive process approximation

`cover_tree_radius`
: A `numeric` specifying the radius (= "spatial resolution") for the cover tree algorithm

`matrix_inversion_method`
: A `string` specifying the method used for inverting covariance matrices. Available options:
  - "cholesky": Cholesky factorization
  - "iterative": iterative methods. A combination of conjugate gradient, Lanczos algorithm, and other methods.
    This is currently only supported for the following cases:
    - likelihood != "gaussian" and gp_approx == "vecchia" (non-Gaussian likelihoods with a Vecchia-Laplace approximation)
    - likelihood == "gaussian" and gp_approx == "full_scale_tapering" (Gaussian likelihood with a full-scale tapering approximation)

`seed`
: An `integer` specifying the seed used for model creation (e.g., random ordering in Vecchia approximation)

`cluster_ids`
: A `vector` with elements indicating independent realizations of random effects / Gaussian processes (same values = same process realization). The elements of 'cluster_ids' can be integer, double, or character.

`free_raw_data`
: A `boolean`. If TRUE, the data (groups, coordinates, covariate data for random coefficients) is freed in R after initialization

`y`
: A `vector` with response variable data

`X`
: A `matrix` with numeric covariate data for the fixed effects linear regression term (if there is one)

`params`
: A `list` with parameters for the estimation / optimization

- optimizer_cov: `string` (default = "lbfgs"). Optimizer used for estimating covariance parameters. Options: "gradient_descent", "lbfgs", "fisher_scoring", "newton", "nelder_mead", "adam". If there are additional auxiliary parameters for non-Gaussian likelihoods, 'optimizer_cov' is also used for those
- optimizer_coef: `string` (default = "wls" for Gaussian likelihoods and "lbfgs" for other likelihoods). Optimizer used for estimating linear regression coefficients, if there are any (for the GPBoost algorithm there are usually none). Options: "gradient_descent", "lbfgs", "wls", "nelder_mead", "adam". Gradient descent steps are done simultaneously with gradient descent steps for the covariance parameters. "wls" refers to doing coordinate descent for the regression coefficients using weighted least squares. If 'optimizer_cov' is set to "nelder_mead", "lbfgs", or "adam", 'optimizer_coef' is automatically also set to the same value.
- maxit: `integer` (default = 1000). Maximal number of iterations for optimization algorithm
- delta_rel_conv: `numeric` (default = 1E-6 except for "nelder_mead" for which the default is 1E-8). Convergence tolerance. The algorithm stops if the relative change in either the (approximate) log-likelihood or the parameters is below this value. For "adam", the L2 norm of the gradient is used instead of the relative change in the log-likelihood. If < 0, internal default values are used
- convergence_criterion: `string` (default = "relative_change_in_log_likelihood"). The convergence criterion used for terminating the optimization algorithm. Options: "relative_change_in_log_likelihood" or "relative_change_in_parameters"
- init_coef: `vector` with `numeric` elements (default = NULL). Initial values for the regression coefficients (if there are any, can be NULL)
- init_cov_pars: `vector` with `numeric` elements (default = NULL). Initial values for covariance parameters of Gaussian process and random effects (can be NULL). The order it the same as the order of the parameters in the summary function: first is the error variance (only for "gaussian" likelihood), next follow the variances of the grouped random effects (if there are any, in the order provided in 'group_data'), and then follow the marginal variance and the range of the Gaussian process. If there are multiple Gaussian processes, then the variances and ranges follow alternatingly. If 'init_cov_pars = NULL', an internal choice is used that depends on the likelihood and the random effects type and covariance function. If you select the option 'trace = TRUE' in the 'params' argument, you will see the first initial covariance parameters in iteration 0.
- lr_coef: `numeric` (default = 0.1). Learning rate for fixed effect regression coefficients if gradient descent is used
- lr_cov: `numeric` (default = 0.1 for "gradient_descent" and 1. otherwise). Initial learning rate for covariance parameters if a gradient-based optimization method is used
  - If lr_cov < 0, internal default values are used (0.1 for "gradient_descent" and 1. otherwise)
  - If there are additional auxiliary parameters for non-Gaussian likelihoods, 'lr_cov' is also used for those

  - For "lbfgs", this is divided by the norm of the gradient in the first iteration
- use_nesterov_acc: `boolean` (default = TRUE). If TRUE Nesterov acceleration is used. This is used only for gradient descent
- acc_rate_coef: `numeric` (default = 0.5). Acceleration rate for regression coefficients (if there are any) for Nesterov acceleration
- acc_rate_cov: `numeric` (default = 0.5). Acceleration rate for covariance parameters for Nesterov acceleration
- momentum_offset: `integer` (Default = 2). Number of iterations for which no momentum is applied in the beginning.
- trace: `boolean` (default = FALSE). If TRUE, information on the progress of the parameter optimization is printed
- std_dev: `boolean` (default = TRUE). If TRUE, approximate standard deviations are calculated for the covariance and linear regression parameters (= square root of diagonal of the inverse Fisher information for Gaussian likelihoods and square root of diagonal of a numerically approximated inverse Hessian for non-Gaussian likelihoods)
- init_aux_pars: `vector` with `numeric` elements (default = NULL). Initial values for additional parameters for non-Gaussian likelihoods (e.g., shape parameter of a gamma or negative_binomial likelihood)
- estimate_aux_pars: `boolean` (default = TRUE). If TRUE, additional parameters for non-Gaussian likelihoods are also estimated (e.g., shape parameter of a gamma or negative_binomial likelihood)
- cg_max_num_it: `integer` (default = 1000). Maximal number of iterations for conjugate gradient algorithms
- cg_max_num_it_tridiag: `integer` (default = 1000). Maximal number of iterations for conjugate gradient algorithm when being run as Lanczos algorithm for tridiagonalization
- cg_delta_conv: `numeric` (default = 1E-2). Tolerance level for L2 norm of residuals for checking convergence in conjugate gradient algorithm when being used for parameter estimation
- num_rand_vec_trace: `integer` (default = 50). Number of random vectors (e.g., Rademacher) for stochastic approximation of the trace of a matrix
- reuse_rand_vec_trace: `boolean` (default = TRUE). If true, random vectors (e.g., Rademacher) for stochastic approximations of the trace of a matrix are sampled only once at the beginning of the parameter estimation and reused in later trace approximations. Otherwise they are sampled every time a trace is calculated
- seed_rand_vec_trace: `integer` (default = 1). Seed number to generate random vectors (e.g., Rademacher)
- piv_chol_rank: `integer` (default = 50). Rank of the pivoted Cholesky decomposition used as preconditioner in conjugate gradient algorithms
- cg_preconditioner_type: `string`. Type of preconditioner used for conjugate gradient algorithms.
  - Options for non-Gaussian likelihoods and gp_approx = "vecchia":

    * "Sigma_inv_plus_BtWB" (= default): (B^T * (D^-1 + W) * B) as preconditioner for inverting (B^T * D^-1 * B + W), where B^T * D^-1 * B approx= Sigma^-1
  - "piv_chol_on_Sigma": (Lk * Lk^T + W^-1) as preconditioner for inverting (B^-1 * D * B^-T + W^-1), where Lk is a low-rank pivoted Cholesky approximation for Sigma and B^-1 * D * B^-T approx= Sigma
  - Options for likelihood = "gaussian" and gp_approx = "full_scale_tapering":
    * "predictive_process_plus_diagonal" (= default): predictive process preconditiioner
    * "none": no preconditioner

`vecchia_approx` Discontinued. Use the argument `gp_approx` instead

`vecchia_pred_type`
: A `string` specifying the type of Vecchia approximation used for making predictions. This is discontinued here. Use the function 'set_prediction_data' to specify this

`num_neighbors_pred`
: an `integer` specifying the number of neighbors for making predictions. This is discontinued here. Use the function 'set_prediction_data' to specify this

`offset`
: A `numeric vector` with additional fixed effects contributions that are added to the linear predictor (= offset). The length of this vector needs to equal the number of training data points.

`fixed_effects` This is discontinued. Use the renamed equivalent argument `offset` instead

Value

A fitted `GPModel`

Author(s)

Fabio Sigrist

Examples

```
# See https://github.com/fabsig/GPBoost/tree/master/R-package for more examples


data(GPBoost_data, package = "gpboost")
# Add intercept column
X1 <- cbind(rep(1,dim(X)[1]),X)
X_test1 <- cbind(rep(1,dim(X_test)[1]),X_test)

#--------------------Grouped random effects model: single-level random effect----------------
gp_model <- fitGPModel(group_data = group_data[,1], y = y, X = X1,
                       likelihood="gaussian", params = list(std_dev = TRUE))
summary(gp_model)
# Make predictions
pred <- predict(gp_model, group_data_pred = group_data_test[,1],
```

```
                X_pred = X_test1, predict_var = TRUE)
pred$mu # Predicted mean
pred$var # Predicted variances
# Also predict covariance matrix
pred <- predict(gp_model, group_data_pred = group_data_test[,1],
                X_pred = X_test1, predict_cov_mat = TRUE)
pred$mu # Predicted mean
pred$cov # Predicted covariance

#--------------------Two crossed random effects and a random slope----------------
gp_model <- fitGPModel(group_data = group_data, likelihood="gaussian",
                       group_rand_coef_data = X[,2],
                       ind_effect_group_rand_coef = 1,
                       y = y, X = X1, params = list(std_dev = TRUE))
summary(gp_model)

#--------------------Gaussian process model----------------
gp_model <- fitGPModel(gp_coords = coords, cov_function = "exponential",
                     likelihood="gaussian", y = y, X = X1, params = list(std_dev = TRUE))
summary(gp_model)
# Make predictions
pred <- predict(gp_model, gp_coords_pred = coords_test,
                X_pred = X_test1, predict_cov_mat = TRUE)
pred$mu # Predicted (posterior) mean of GP
pred$cov # Predicted (posterior) covariance matrix of GP

#--------------------Gaussian process model with Vecchia approximation----------------
gp_model <- fitGPModel(gp_coords = coords, cov_function = "exponential",
                       gp_approx = "vecchia", num_neighbors = 20,
                       likelihood="gaussian", y = y)
summary(gp_model)

#--------------------Gaussian process model with random coefficients----------------
gp_model <- fitGPModel(gp_coords = coords, cov_function = "exponential",
                       gp_rand_coef_data = X[,2], y=y,
                       likelihood = "gaussian", params = list(std_dev = TRUE))
summary(gp_model)

#--------------------Combine Gaussian process with grouped random effects----------------
gp_model <- fitGPModel(group_data = group_data,
                       gp_coords = coords, cov_function = "exponential",
                     likelihood = "gaussian", y = y, X = X1, params = list(std_dev = TRUE))
summary(gp_model)
```

`getinfo` *Get information of an* `gpb.Dataset` *object*

Description

Get one attribute of a `gpb.Dataset`

Usage

```
getinfo(dataset, ...)

## S3 method for class 'gpb.Dataset'
getinfo(dataset, name, ...)
```

Arguments

dataset	Object of class gpb.Dataset
...	other parameters
name	the name of the information field to get (see details)

Details

The name field can be one of the following:

- label: label gpboost learn from ;
- weight: to do a weight rescale ;
- group: used for learning-to-rank tasks. An integer vector describing how to group rows together as ordered results from the same set of candidate results to be ranked. For example, if you have a 100-document dataset with group = c(10, 20, 40, 10, 10, 10), that means that you have 6 groups, where the first 10 records are in the first group, records 11-30 are in the second group, etc.
- init_score: initial score is the base prediction gpboost will boost from.

Value

info data

info data

Examples

```
data(agaricus.train, package = "gpboost")
train <- agaricus.train
dtrain <- gpb.Dataset(train$data, label = train$label)
gpb.Dataset.construct(dtrain)

labels <- gpboost::getinfo(dtrain, "label")
gpboost::setinfo(dtrain, "label", 1 - labels)

labels2 <- gpboost::getinfo(dtrain, "label")
stopifnot(all(labels2 == 1 - labels))
```

get_aux_pars *Get (estimated) auxiliary (additional) parameters of the likelihood*

Description

Get (estimated) auxiliary (additional) parameters of the likelihood such as the shape parameter of a gamma or a negative binomial distribution. Some likelihoods (e.g., bernoulli_logit or poisson) have no auxiliary parameters

Usage

```
get_aux_pars(gp_model)
```

Arguments

gp_model	A GPModel

Author(s)

Fabio Sigrist

Examples

```
data(GPBoost_data, package = "gpboost")
X1 <- cbind(rep(1,dim(X)[1]),X) # Add intercept column
y_pos <- exp(y)
gp_model <- fitGPModel(group_data = group_data[,1], y = y_pos, X = X1, likelihood="gamma")
get_aux_pars(gp_model)
```

get_aux_pars.GPModel *Get (estimated) auxiliary (additional) parameters of the likelihood*

Description

Get (estimated) auxiliary (additional) parameters of the likelihood such as the shape parameter of a gamma or a negative binomial distribution. Some likelihoods (e.g., bernoulli_logit or poisson) have no auxiliary parameters

Usage

```
## S3 method for class 'GPModel'
get_aux_pars(gp_model)
```

Arguments

gp_model A GPModel

Value

A GPModel

Author(s)

Fabio Sigrist

Examples

```
data(GPBoost_data, package = "gpboost")
X1 <- cbind(rep(1,dim(X)[1]),X) # Add intercept column
y_pos <- exp(y)
gp_model <- fitGPModel(group_data = group_data[,1], y = y_pos, X = X1, likelihood="gamma")
get_aux_pars(gp_model)
```

---

get_coef *Get (estimated) linear regression coefficients*

Description

Get (estimated) linear regression coefficients and standard deviations (if std_dev=TRUE was set in fit)

Usage

```
get_coef(gp_model)
```

Arguments

gp_model A GPModel

Author(s)

Fabio Sigrist

Examples

```
data(GPBoost_data, package = "gpboost")
X1 <- cbind(rep(1,dim(X)[1]),X) # Add intercept column
gp_model <- fitGPModel(group_data = group_data[,1], y = y, X = X1, likelihood="gaussian")
get_coef(gp_model)
```

get_coef.GPModel *Get (estimated) linear regression coefficients*

Description

Get (estimated) linear regression coefficients and standard deviations (if std_dev=TRUE was set in `fit`)

Usage

```
## S3 method for class 'GPModel'
get_coef(gp_model)
```

Arguments

gp_model	A `GPModel`

Value

A `GPModel`

Author(s)

Fabio Sigrist

Examples

```
data(GPBoost_data, package = "gpboost")
X1 <- cbind(rep(1,dim(X)[1]),X) # Add intercept column
gp_model <- fitGPModel(group_data = group_data[,1], y = y, X = X1, likelihood="gaussian")
get_coef(gp_model)
```

get_cov_pars *Get (estimated) covariance parameters*

Description

Get (estimated) covariance parameters and standard deviations (if std_dev=TRUE was set in `fit`)

Usage

```
get_cov_pars(gp_model)
```

Arguments

gp_model	A `GPModel`

Author(s)

Fabio Sigrist

Examples

```
data(GPBoost_data, package = "gpboost")
X1 <- cbind(rep(1,dim(X)[1]),X) # Add intercept column
gp_model <- fitGPModel(group_data = group_data[,1], y = y, X = X1, likelihood="gaussian")
get_cov_pars(gp_model)
```

---

get_cov_pars.GPModel *Get (estimated) covariance parameters*

Description

Get (estimated) covariance parameters and standard deviations (if std_dev=TRUE was set in `fit`)

Usage

```
## S3 method for class 'GPModel'
get_cov_pars(gp_model)
```

Arguments

gp_model	A `GPModel`

Value

A `GPModel`

Author(s)

Fabio Sigrist

Examples

```
data(GPBoost_data, package = "gpboost")
X1 <- cbind(rep(1,dim(X)[1]),X) # Add intercept column
gp_model <- fitGPModel(group_data = group_data[,1], y = y, X = X1, likelihood="gaussian")
get_cov_pars(gp_model)
```

get_nested_categories *Auxiliary function to create categorical variables for nested grouped random effects*

Description

Auxiliary function to create categorical variables for nested grouped random effects

Usage

```
get_nested_categories(outer_var, inner_var)
```

Arguments

outer_var	A vector containing the outer categorical grouping variable within which the inner_var is nested in. Can be of type integer, double, or character.
inner_var	A vector containing the inner nested categorical grouping variable

Value

A vector containing a categorical variable such that inner_var is nested in outer_var

Author(s)

Fabio Sigrist

Examples

```
# Fit a model with Time as categorical fixed effects variables and Diet and Chick
#   as random effects, where Chick is nested in Diet using lme4
chick_nested_diet <- get_nested_categories(ChickWeight$Diet, ChickWeight$Chick)
fixed_effects_matrix <- model.matrix(weight ~ as.factor(Time), data = ChickWeight)
mod_gpb <- fitGPModel(X = fixed_effects_matrix,
                      group_data = cbind(diet=ChickWeight$Diet, chick_nested_diet),
                      y = ChickWeight$weight, params = list(std_dev = TRUE))
summary(mod_gpb)
# This does (almost) the same thing as the following code using lme4:
# mod_lme4 <- lmer(weight ~ as.factor(Time) + (1 | Diet/Chick), data = ChickWeight, REML = FALSE)
# summary(mod_lme4)
```

gpb.convert_with_rules *Data preparator for GPBoost datasets with rules (integer)*

Description

Attempts to prepare a clean dataset to prepare to put in a gpb.Dataset. Factor, character, and logical columns are converted to integer. Missing values in factors and characters will be filled with 0L. Missing values in logicals will be filled with -1L.

This function returns and optionally takes in "rules" the describe exactly how to convert values in columns.

Columns that contain only NA values will be converted by this function but will not show up in the returned rules.

Usage

```
gpb.convert_with_rules(data, rules = NULL)
```

Arguments

data	A data.frame or data.table to prepare.
rules	A set of rules from the data preparator, if already used. This should be an R list, where names are column names in data and values are named character vectors whose names are column values and whose values are new values to replace them with.

Value

A list with the cleaned dataset (data) and the rules (rules). Note that the data must be converted to a matrix format (as.matrix) for input in gpb.Dataset.

Examples

```
data(iris)

str(iris)

new_iris <- gpb.convert_with_rules(data = iris)
str(new_iris$data)

data(iris) # Erase iris dataset
iris$Species[1L] <- "NEW FACTOR" # Introduce junk factor (NA)

# Use conversion using known rules
# Unknown factors become 0, excellent for sparse datasets
newer_iris <- gpb.convert_with_rules(data = iris, rules = new_iris$rules)
```

```
# Unknown factor is now zero, perfect for sparse datasets
newer_iris$data[1L, ] # Species became 0 as it is an unknown factor

newer_iris$data[1L, 5L] <- 1.0 # Put back real initial value

# Is the newly created dataset equal? YES!
all.equal(new_iris$data, newer_iris$data)

# Can we test our own rules?
data(iris) # Erase iris dataset

# We remapped values differently
personal_rules <- list(
  Species = c(
    "setosa" = 3L
    , "versicolor" = 2L
    , "virginica" = 1L
  )
)
newest_iris <- gpb.convert_with_rules(data = iris, rules = personal_rules)
str(newest_iris$data) # SUCCESS!
```

---

gpb.cv *CV function for number of boosting iterations*

Description

Cross validation function for determining number of boosting iterations

Usage

```
gpb.cv(params = list(), data, nrounds = 100L, gp_model = NULL,
  line_search_step_length = FALSE, use_gp_model_for_validation = TRUE,
  fit_GP_cov_pars_OOS = FALSE, train_gp_model_cov_pars = TRUE,
  folds = NULL, nfold = 4L, label = NULL, weight = NULL, obj = NULL,
  eval = NULL, verbose = 1L, record = TRUE, eval_freq = 1L,
  showsd = FALSE, stratified = TRUE, init_model = NULL, colnames = NULL,
  categorical_feature = NULL, early_stopping_rounds = NULL,
  callbacks = list(), reset_data = FALSE, delete_boosters_folds = FALSE,
  ...)
```

Arguments

params	list of "tuning" parameters. See the parameter documentation for more information. A few key parameters: • `learning_rate`: The learning rate, also called shrinkage or damping parameter (default = 0.1). An important tuning parameter for boosting. Lower values usually lead to higher predictive accuracy but more boosting iterations are needed

- num_leaves: Number of leaves in a tree. Tuning parameter for tree-boosting (default = 31)
- max_depth: Maximal depth of a tree. Tuning parameter for tree-boosting (default = no limit)
- min_data_in_leaf: Minimal number of samples per leaf. Tuning parameter for tree-boosting (default = 20)
- lambda_l2: L2 regularization (default = 0)
- lambda_l1: L1 regularization (default = 0)
- max_bin: Maximal number of bins that feature values will be bucketed in (default = 255)
- line_search_step_length (default = FALSE): If TRUE, a line search is done to find the optimal step length for every boosting update (see, e.g., Friedman 2001). This is then multiplied by the learning rate
- train_gp_model_cov_pars (default = TRUE): If TRUE, the covariance parameters of the Gaussian process are estimated in every boosting iterations, otherwise the gp_model parameters are not estimated. In the latter case, you need to either estimate them beforehand or provide values via the 'init_cov_pars' parameter when creating the gp_model
- use_gp_model_for_validation (default = TRUE): If TRUE, the Gaussian process is also used (in addition to the tree model) for calculating predictions on the validation data
- leaves_newton_update (default = FALSE): Set this to TRUE to do a Newton update step for the tree leaves after the gradient step. Applies only to Gaussian process boosting (GPBoost algorithm)
- num_threads: Number of threads. For the best speed, set this to the number of real CPU cores(parallel::detectCores(logical = FALSE)), not the number of threads (most CPU using hyper-threading to generate 2 threads per CPU core).

data
: a gpb.Dataset object, used for training. Some functions, such as gpb.cv, may allow you to pass other types of data like matrix and then separately supply label as a keyword argument.

nrounds
: number of boosting iterations (= number of trees). This is the most important tuning parameter for boosting

gp_model
: A GPModel object that contains the random effects (Gaussian process and / or grouped random effects) model

line_search_step_length
: Boolean. If TRUE, a line search is done to find the optimal step length for every boosting update (see, e.g., Friedman 2001). This is then multiplied by the learning_rate. Applies only to the GPBoost algorithm

use_gp_model_for_validation
: Boolean. If TRUE, the gp_model (Gaussian process and/or random effects) is also used (in addition to the tree model) for calculating predictions on the validation data. If FALSE, the gp_model (random effects part) is ignored for making predictions and only the tree ensemble is used for making predictions for calculating the validation / test error.

`fit_GP_cov_pars_OOS`
: Boolean (default = FALSE). If TRUE, the covariance parameters of the `gp_model` model are estimated using the out-of-sample (OOS) predictions on the validation data using the optimal number of iterations (after performing the CV). This corresponds to the GPBoostOOS algorithm.

`train_gp_model_cov_pars`
: Boolean. If TRUE, the covariance parameters of the `gp_model` (Gaussian process and/or random effects) are estimated in every boosting iterations, otherwise the `gp_model` parameters are not estimated. In the latter case, you need to either estimate them beforehand or provide the values via the `init_cov_pars` parameter when creating the `gp_model`

`folds`
: `list` provides a possibility to use a list of pre-defined CV folds (each element must be a vector of test fold's indices). When folds are supplied, the `nfold` and `stratified` parameters are ignored.

`nfold`
: the original dataset is randomly partitioned into `nfold` equal size subsamples.

`label`
: Vector of labels, used if `data` is not an `gpb.Dataset`

`weight`
: vector of response values. If not NULL, will set to dataset

`obj`
: (character) The distribution of the response variable (=label) conditional on fixed and random effects. This only needs to be set when doing independent boosting without random effects / Gaussian processes.

`eval`
: Evaluation metric to be monitored when doing CV and parameter tuning. This can be a string, function, or list with a mixture of strings and functions.

  - **a. character vector**: Non-exhaustive list of supported metrics: "test_neg_log_likelihood", "mse", "rmse", "mae", "auc", "average_precision", "binary_logloss", "binary_error" See the "metric" section of the parameter documentation for a complete list of valid metrics.
  - **b. function**: You can provide a custom evaluation function. This should accept the keyword arguments `preds` and `dtrain` and should return a named list with three elements:
    - `name`: A string with the name of the metric, used for printing and storing results.
    - `value`: A single number indicating the value of the metric for the given predictions and true values
    - `higher_better`: A boolean indicating whether higher values indicate a better fit. For example, this would be `FALSE` for metrics like MAE or RMSE.
  - **c. list**: If a list is given, it should only contain character vectors and functions. These should follow the requirements from the descriptions above.

`verbose`
: verbosity for output, if <= 0, also will disable the print of evaluation during training

`record`
: Boolean, TRUE will record iteration message to `booster$record_evals`

`eval_freq`
: evaluation output frequency, only effect when verbose > 0

`showsd`
: `boolean`, whether to show standard deviation of cross validation. This parameter defaults to `TRUE`.

`stratified` a boolean indicating whether sampling of folds should be stratified by the values of outcome labels.

`init_model` path of model file of gpb.Booster object, will continue training from this model

`colnames` feature names, if not null, will use this to overwrite the names in dataset

`categorical_feature`
categorical features. This can either be a character vector of feature names or an integer vector with the indices of the features (e.g. `c(1L, 10L)` to say "the first and tenth columns").

`early_stopping_rounds`
int. Activates early stopping. Requires at least one validation data and one metric. When this parameter is non-null, training will stop if the evaluation of any metric on any validation set fails to improve for `early_stopping_rounds` consecutive boosting rounds. If training stops early, the returned model will have attribute `best_iter` set to the iteration number of the best iteration.

`callbacks` List of callback functions that are applied at each iteration.

`reset_data` Boolean, setting it to TRUE (not the default value) will transform the booster model into a predictor model which frees up memory and the original datasets

`delete_boosters_folds`
Boolean, setting it to TRUE (not the default value) will delete the boosters of the individual folds

`...` other parameters, see Parameters.rst for more information.

Value

a trained model gpb.CVBooster.

Early Stopping

"early stopping" refers to stopping the training process if the model's performance on a given validation set does not improve for several consecutive iterations.

If multiple arguments are given to `eval`, their order will be preserved. If you enable early stopping by setting `early_stopping_rounds` in `params`, by default all metrics will be considered for early stopping.

If you want to only consider the first metric for early stopping, pass `first_metric_only = TRUE` in `params`. Note that if you also specify `metric` in `params`, that metric will be considered the "first" one. If you omit `metric`, a default metric will be used based on your choice for the parameter `obj` (keyword argument) or `objective` (passed into `params`).

Author(s)

Authors of the LightGBM R package, Fabio Sigrist

Examples

```
# See https://github.com/fabsig/GPBoost/tree/master/R-package for more examples

library(gpboost)
```

```
data(GPBoost_data, package = "gpboost")

# Create random effects model and dataset
gp_model <- GPModel(group_data = group_data[,1], likelihood="gaussian")
dtrain <- gpb.Dataset(X, label = y)
params <- list(learning_rate = 0.05,
               max_depth = 6,
               min_data_in_leaf = 5)
# Run CV
cvbst <- gpb.cv(params = params,
                data = dtrain,
                gp_model = gp_model,
                nrounds = 100,
                nfold = 4,
                eval = "l2",
                early_stopping_rounds = 5,
                use_gp_model_for_validation = TRUE)
print(paste0("Optimal number of iterations: ", cvbst$best_iter,
             ", best test error: ", cvbst$best_score))
```

`gpb.Dataset` *Construct* `gpb.Dataset` *object*

Description

Construct `gpb.Dataset` object from dense matrix, sparse matrix or local file (that was created previously by saving an `gpb.Dataset`).

Usage

```
gpb.Dataset(data, params = list(), reference = NULL, colnames = NULL,
  categorical_feature = NULL, free_raw_data = FALSE, info = list(), ...)
```

Arguments

`data`	a `matrix` object, a `dgCMatrix` object or a character representing a filename
`params`	a list of parameters. See the "Dataset Parameters" section of the parameter documentation for a list of parameters and valid values.
`reference`	reference dataset. When GPBoost creates a Dataset, it does some preprocessing like binning continuous features into histograms. If you want to apply the same bin boundaries from an existing dataset to new `data`, pass that existing Dataset to this argument.
`colnames`	names of columns
`categorical_feature`	categorical features. This can either be a character vector of feature names or an integer vector with the indices of the features (e.g. `c(1L, 10L)` to say "the first and tenth columns").

free_raw_data	GPBoost constructs its data format, called a "Dataset", from tabular data. By default, this Dataset object on the R side does keep a copy of the raw data. If you set free_raw_data = TRUE, no copy of the raw data is kept (this reduces memory usage)
info	a list of information of the gpb.Dataset object
...	other information to pass to info or parameters pass to params

Value

constructed dataset

Examples

```
data(agaricus.train, package = "gpboost")
train <- agaricus.train
dtrain <- gpb.Dataset(train$data, label = train$label)
data_file <- tempfile(fileext = ".data")
gpb.Dataset.save(dtrain, data_file)
dtrain <- gpb.Dataset(data_file)
gpb.Dataset.construct(dtrain)
```

---

gpb.Dataset.construct *Construct Dataset explicitly*

Description

Construct Dataset explicitly

Usage

```
gpb.Dataset.construct(dataset)
```

Arguments

dataset	Object of class gpb.Dataset

Value

constructed dataset

Examples

```
data(agaricus.train, package = "gpboost")
train <- agaricus.train
dtrain <- gpb.Dataset(train$data, label = train$label)
gpb.Dataset.construct(dtrain)
```

gpb.Dataset.create.valid *Construct validation data*

Description

Construct validation data according to training data

Usage

```
gpb.Dataset.create.valid(dataset, data, info = list(), ...)
```

Arguments

dataset	gpb.Dataset object, training data
data	a matrix object, a dgCMatrix object or a character representing a filename
info	a list of information of the gpb.Dataset object
...	other information to pass to info.

Value

constructed dataset

Examples

```
data(agaricus.train, package = "gpboost")
train <- agaricus.train
dtrain <- gpb.Dataset(train$data, label = train$label)
data(agaricus.test, package = "gpboost")
test <- agaricus.test
dtest <- gpb.Dataset.create.valid(dtrain, test$data, label = test$label)
```

gpb.Dataset.save *Save* gpb.Dataset *to a binary file*

Description

Please note that init_score is not saved in binary file. If you need it, please set it again after loading Dataset.

Usage

```
gpb.Dataset.save(dataset, fname)
```

Arguments

dataset	object of class gpb.Dataset
fname	object filename of output file

Value

the dataset you passed in

Examples

```
data(agaricus.train, package = "gpboost")
train <- agaricus.train
dtrain <- gpb.Dataset(train$data, label = train$label)
gpb.Dataset.save(dtrain, tempfile(fileext = ".bin"))
```

---

gpb.Dataset.set.categorical

Set categorical feature of gpb.Dataset

Description

Set the categorical features of an gpb.Dataset object. Use this function to tell GPBoost which features should be treated as categorical.

Usage

```
gpb.Dataset.set.categorical(dataset, categorical_feature)
```

Arguments

dataset	object of class gpb.Dataset
categorical_feature	categorical features. This can either be a character vector of feature names or an integer vector with the indices of the features (e.g. c(1L, 10L) to say "the first and tenth columns").

Value

the dataset you passed in

Examples

```
data(agaricus.train, package = "gpboost")
train <- agaricus.train
dtrain <- gpb.Dataset(train$data, label = train$label)
data_file <- tempfile(fileext = ".data")
gpb.Dataset.save(dtrain, data_file)
dtrain <- gpb.Dataset(data_file)
gpb.Dataset.set.categorical(dtrain, 1L:2L)
```

---

gpb.Dataset.set.reference

Set reference of gpb.Dataset

Description

If you want to use validation data, you should set reference to training data

Usage

```
gpb.Dataset.set.reference(dataset, reference)
```

Arguments

dataset	object of class gpb.Dataset
reference	object of class gpb.Dataset

Value

the dataset you passed in

Examples

```
data(agaricus.train, package ="gpboost")
train <- agaricus.train
dtrain <- gpb.Dataset(train$data, label = train$label)
data(agaricus.test, package = "gpboost")
test <- agaricus.test
dtest <- gpb.Dataset(test$data, test = train$label)
gpb.Dataset.set.reference(dtest, dtrain)
```

gpb.dump *Dump GPBoost model to json*

Description

Dump GPBoost model to json

Usage

```
gpb.dump(booster, num_iteration = NULL)
```

Arguments

booster	Object of class gpb.Booster
num_iteration	number of iteration want to predict with, NULL or <= 0 means use best iteration

Value

json format of model

Examples

```
library(gpboost)
data(agaricus.train, package = "gpboost")
train <- agaricus.train
dtrain <- gpb.Dataset(train$data, label = train$label)
data(agaricus.test, package = "gpboost")
test <- agaricus.test
dtest <- gpb.Dataset.create.valid(dtrain, test$data, label = test$label)
params <- list(objective = "regression", metric = "l2")
valids <- list(test = dtest)
model <- gpb.train(
  params = params
  , data = dtrain
  , nrounds = 10L
  , valids = valids
  , min_data = 1L
  , learning_rate = 1.0
  , early_stopping_rounds = 5L
)
json_model <- gpb.dump(model)
```

gpb.get.eval.result *Get record evaluation result from booster*

Description

Given a gpb.Booster, return evaluation results for a particular metric on a particular dataset.

Usage

```
gpb.get.eval.result(booster, data_name, eval_name, iters = NULL,
  is_err = FALSE)
```

Arguments

booster	Object of class gpb.Booster
data_name	Name of the dataset to return evaluation results for.
eval_name	Name of the evaluation metric to return results for.
iters	An integer vector of iterations you want to get evaluation results for. If NULL (the default), evaluation results for all iterations will be returned.
is_err	TRUE will return evaluation error instead

Value

numeric vector of evaluation result

Examples

```
# train a regression model
data(agaricus.train, package = "gpboost")
train <- agaricus.train
dtrain <- gpb.Dataset(train$data, label = train$label)
data(agaricus.test, package = "gpboost")
test <- agaricus.test
dtest <- gpb.Dataset.create.valid(dtrain, test$data, label = test$label)
params <- list(objective = "regression", metric = "l2")
valids <- list(test = dtest)
model <- gpb.train(
  params = params
  , data = dtrain
  , nrounds = 5L
  , valids = valids
  , min_data = 1L
  , learning_rate = 1.0
)

# Examine valid data_name values
print(setdiff(names(model$record_evals), "start_iter"))
```

```
# Examine valid eval_name values for dataset "test"
print(names(model$record_evals[["test"]]))

# Get L2 values for "test" dataset
gpb.get.eval.result(model, "test", "l2")
```

---

gpb.grid.search.tune.parameters

Function for choosing tuning parameters

Description

Function that allows for choosing tuning parameters from a grid in a determinstic or random way using cross validation or validation data sets.

Usage

```
gpb.grid.search.tune.parameters(param_grid, data, params = list(),
  num_try_random = NULL, nrounds = 100L, gp_model = NULL,
  line_search_step_length = FALSE, use_gp_model_for_validation = TRUE,
  train_gp_model_cov_pars = TRUE, folds = NULL, nfold = 4L,
  label = NULL, weight = NULL, obj = NULL, eval = NULL,
  verbose_eval = 1L, stratified = TRUE, init_model = NULL,
  colnames = NULL, categorical_feature = NULL,
  early_stopping_rounds = NULL, callbacks = list(),
  return_all_combinations = FALSE, ...)
```

Arguments

param_grid	list with candidate parameters defining the grid over which a search is done
data	a gpb.Dataset object, used for training. Some functions, such as gpb.cv, may allow you to pass other types of data like matrix and then separately supply label as a keyword argument.
params	list with other parameters not included in param_grid
num_try_random	integer with number of random trial on parameter grid. If NULL, a deterministic search is done
nrounds	number of boosting iterations (= number of trees). This is the most important tuning parameter for boosting
gp_model	A GPModel object that contains the random effects (Gaussian process and / or grouped random effects) model
line_search_step_length	Boolean. If TRUE, a line search is done to find the optimal step length for every boosting update (see, e.g., Friedman 2001). This is then multiplied by the learning_rate. Applies only to the GPBoost algorithm

use_gp_model_for_validation
: Boolean. If TRUE, the gp_model (Gaussian process and/or random effects) is also used (in addition to the tree model) for calculating predictions on the validation data. If FALSE, the gp_model (random effects part) is ignored for making predictions and only the tree ensemble is used for making predictions for calculating the validation / test error.

train_gp_model_cov_pars
: Boolean. If TRUE, the covariance parameters of the gp_model (Gaussian process and/or random effects) are estimated in every boosting iterations, otherwise the gp_model parameters are not estimated. In the latter case, you need to either estimate them beforehand or provide the values via the init_cov_pars parameter when creating the gp_model

folds
: list provides a possibility to use a list of pre-defined CV folds (each element must be a vector of test fold's indices). When folds are supplied, the nfold and stratified parameters are ignored.

nfold
: the original dataset is randomly partitioned into nfold equal size subsamples.

label
: Vector of labels, used if data is not an gpb.Dataset

weight
: vector of response values. If not NULL, will set to dataset

obj
: (character) The distribution of the response variable (=label) conditional on fixed and random effects. This only needs to be set when doing independent boosting without random effects / Gaussian processes.

eval
: Evaluation metric to be monitored when doing CV and parameter tuning. This can be a string, function, or list with a mixture of strings and functions.

  - **a. character vector**: Non-exhaustive list of supported metrics: "test_neg_log_likelihood", "mse", "rmse", "mae", "auc", "average_precision", "binary_logloss", "binary_error" See the "metric" section of the parameter documentation for a complete list of valid metrics.
  - **b. function**: You can provide a custom evaluation function. This should accept the keyword arguments preds and dtrain and should return a named list with three elements:
    - name: A string with the name of the metric, used for printing and storing results.
    - value: A single number indicating the value of the metric for the given predictions and true values
    - higher_better: A boolean indicating whether higher values indicate a better fit. For example, this would be FALSE for metrics like MAE or RMSE.
  - **c. list**: If a list is given, it should only contain character vectors and functions. These should follow the requirements from the descriptions above.

verbose_eval
: integer. Whether to display information on the progress of tuning parameter choice. If None or 0, verbose is of. If = 1, summary progress information is displayed for every parameter combination. If >= 2, detailed progress is displayed at every boosting stage for every parameter combination.

stratified
: a boolean indicating whether sampling of folds should be stratified by the values of outcome labels.

`init_model` path of model file of `gpb.Booster` object, will continue training from this model

`colnames` feature names, if not null, will use this to overwrite the names in dataset

`categorical_feature`
categorical features. This can either be a character vector of feature names or an integer vector with the indices of the features (e.g. `c(1L, 10L)` to say "the first and tenth columns").

`early_stopping_rounds`
int. Activates early stopping. Requires at least one validation data and one metric. When this parameter is non-null, training will stop if the evaluation of any metric on any validation set fails to improve for `early_stopping_rounds` consecutive boosting rounds. If training stops early, the returned model will have attribute `best_iter` set to the iteration number of the best iteration.

`callbacks` List of callback functions that are applied at each iteration.

`return_all_combinations`
a `boolean` indicating whether all tried parameter combinations are returned

`...` other parameters, see Parameters.rst for more information.

Value

A `list` with the best parameter combination and score The list has the following format: list("best_params" = best_params, "best_iter" = best_iter, "best_score" = best_score) If return_all_combinations is TRUE, then the list contains an additional entry 'all_combinations'

Early Stopping

"early stopping" refers to stopping the training process if the model's performance on a given validation set does not improve for several consecutive iterations.

If multiple arguments are given to `eval`, their order will be preserved. If you enable early stopping by setting `early_stopping_rounds` in `params`, by default all metrics will be considered for early stopping.

If you want to only consider the first metric for early stopping, pass `first_metric_only = TRUE` in `params`. Note that if you also specify `metric` in `params`, that metric will be considered the "first" one. If you omit `metric`, a default metric will be used based on your choice for the parameter `obj` (keyword argument) or `objective` (passed into `params`).

Author(s)

Fabio Sigrist

Examples

```
# See https://github.com/fabsig/GPBoost/tree/master/R-package for more examples

library(gpboost)
data(GPBoost_data, package = "gpboost")

# Create random effects model, dataset, and define parameter grid
gp_model <- GPModel(group_data = group_data[,1], likelihood="gaussian")
```

```
dataset <- gpb.Dataset(X, label = y)
param_grid = list("learning_rate" = c(1,0.1,0.01),
                  "min_data_in_leaf" = c(10,100,1000),
                  "max_depth" = c(1,2,3,5,10),
                  "lambda_l2" = c(0,1,10))
other_params <- list(num_leaves = 2^10)
# Note: here we try different values for 'max_depth' and thus set 'num_leaves' to a large value.
#       An alternative strategy is to impose no limit on 'max_depth',
#       and try different values for 'num_leaves' as follows:
# param_grid = list("learning_rate" = c(1,0.1,0.01),
#                   "min_data_in_leaf" = c(10,100,1000),
#                   "num_leaves" = 2^(1:10),
#                   "lambda_l2" = c(0,1,10))
# other_params <- list(max_depth = -1)
set.seed(1)
opt_params <- gpb.grid.search.tune.parameters(param_grid = param_grid, params = other_params,
                                              num_try_random = NULL, nfold = 4,
                                              data = dataset, gp_model = gp_model,
                                       use_gp_model_for_validation=TRUE, verbose_eval = 1,
                                              nrounds = 1000, early_stopping_rounds = 10)
print(paste0("Best parameters: ",
             paste0(unlist(lapply(seq_along(opt_params$best_params),
                                  function(y, n, i) { paste0(n[[i]],": ", y[[i]]) },
                                  y=opt_params$best_params,
                                  n=names(opt_params$best_params))), collapse=", ")))
print(paste0("Best number of iterations: ", opt_params$best_iter))
print(paste0("Best score: ", round(opt_params$best_score, digits=3)))
# Note: other scoring / evaluation metrics can be chosen using the
#       'metric' argument, e.g., metric = "l1"

# Using manually defined validation data instead of cross-validation
valid_tune_idx <- sample.int(length(y), as.integer(0.2*length(y)))
folds = list(valid_tune_idx)
opt_params <- gpb.grid.search.tune.parameters(param_grid = param_grid, params = other_params,
                                              num_try_random = NULL, folds = folds,
                                              data = dataset, gp_model = gp_model,
                                       use_gp_model_for_validation=TRUE, verbose_eval = 1,
                                              nrounds = 1000, early_stopping_rounds = 10)
```

---

gpb.importance *Compute feature importance in a model*

Description

Creates a `data.table` of feature importances in a model.

Usage

```
gpb.importance(model, percentage = TRUE)
```

Arguments

model	object of class gpb.Booster.
percentage	whether to show importance in relative percentage.

Value

For a tree model, a data.table with the following columns:

- Feature: Feature names in the model.
- Gain: The total gain of this feature's splits.
- Cover: The number of observation related to this feature.
- Frequency: The number of times a feature splited in trees.

Examples

```
data(agaricus.train, package = "gpboost")
train <- agaricus.train
dtrain <- gpb.Dataset(train$data, label = train$label)

params <- list(
  objective = "binary"
  , learning_rate = 0.1
  , max_depth = -1L
  , min_data_in_leaf = 1L
  , min_sum_hessian_in_leaf = 1.0
)
model <- gpb.train(
    params = params
    , data = dtrain
    , nrounds = 5L
)

tree_imp1 <- gpb.importance(model, percentage = TRUE)
tree_imp2 <- gpb.importance(model, percentage = FALSE)
```

---

gpb.interprete *Compute feature contribution of prediction*

Description

Computes feature contribution components of rawscore prediction.

Usage

```
gpb.interprete(model, data, idxset, num_iteration = NULL)
```

Arguments

model	object of class gpb.Booster.
data	a matrix object or a dgCMatrix object.
idxset	an integer vector of indices of rows needed.
num_iteration	number of iteration want to predict with, NULL or <= 0 means use best iteration.

Value

For regression, binary classification and lambdarank model, a list of data.table with the following columns:

- Feature: Feature names in the model.
- Contribution: The total contribution of this feature's splits.

For multiclass classification, a list of data.table with the Feature column and Contribution columns to each class.

Examples

```
Logit <- function(x) log(x / (1.0 - x))
data(agaricus.train, package = "gpboost")
train <- agaricus.train
dtrain <- gpb.Dataset(train$data, label = train$label)
setinfo(dtrain, "init_score", rep(Logit(mean(train$label)), length(train$label)))
data(agaricus.test, package = "gpboost")
test <- agaricus.test

params <- list(
    objective = "binary"
    , learning_rate = 0.1
    , max_depth = -1L
    , min_data_in_leaf = 1L
    , min_sum_hessian_in_leaf = 1.0
)
model <- gpb.train(
    params = params
    , data = dtrain
    , nrounds = 3L
)

tree_interpretation <- gpb.interprete(model, test$data, 1L:5L)
```

gpb.load *Load GPBoost model*

Description

Load GPBoost takes in either a file path or model string. If both are provided, Load will default to loading from file Boosters with gp_models can only be loaded from file.

Usage

```
gpb.load(filename = NULL, model_str = NULL)
```

Arguments

filename	path of model file
model_str	a str containing the model

Value

gpb.Booster

Author(s)

Fabio Sigrist, authors of the LightGBM R package

Examples

```
library(gpboost)
data(GPBoost_data, package = "gpboost")

# Train model and make prediction
gp_model <- GPModel(group_data = group_data[,1], likelihood = "gaussian")
bst <- gpboost(data = X, label = y, gp_model = gp_model, nrounds = 16,
               learning_rate = 0.05, max_depth = 6, min_data_in_leaf = 5,
               verbose = 0)
pred <- predict(bst, data = X_test, group_data_pred = group_data_test[,1],
                predict_var= TRUE, pred_latent = TRUE)
# Save model to file
filename <- tempfile(fileext = ".json")
gpb.save(bst,filename = filename)
# Load from file and make predictions again
bst_loaded <- gpb.load(filename = filename)
pred_loaded <- predict(bst_loaded, data = X_test, group_data_pred = group_data_test[,1],
                       predict_var= TRUE, pred_latent = TRUE)
# Check equality
pred$fixed_effect - pred_loaded$fixed_effect
pred$random_effect_mean - pred_loaded$random_effect_mean
pred$random_effect_cov - pred_loaded$random_effect_cov
```

gpb.model.dt.tree *Parse a GPBoost model json dump*

Description

Parse a GPBoost model json dump into a data.table structure.

Usage

```
gpb.model.dt.tree(model, num_iteration = NULL)
```

Arguments

model	object of class gpb.Booster
num_iteration	number of iterations you want to predict with. NULL or <= 0 means use best iteration

Value

A data.table with detailed information about model trees' nodes and leafs.

The columns of the data.table are:

- tree_index: ID of a tree in a model (integer)
- split_index: ID of a node in a tree (integer)
- split_feature: for a node, it's a feature name (character); for a leaf, it simply labels it as "NA"
- node_parent: ID of the parent node for current node (integer)
- leaf_index: ID of a leaf in a tree (integer)
- leaf_parent: ID of the parent node for current leaf (integer)
- split_gain: Split gain of a node
- threshold: Splitting threshold value of a node
- decision_type: Decision type of a node
- default_left: Determine how to handle NA value, TRUE -> Left, FALSE -> Right
- internal_value: Node value
- internal_count: The number of observation collected by a node
- leaf_value: Leaf value
- leaf_count: The number of observation collected by a leaf

Examples

```
data(agaricus.train, package = "gpboost")
train <- agaricus.train
dtrain <- gpb.Dataset(train$data, label = train$label)

params <- list(
  objective = "binary"
  , learning_rate = 0.01
  , num_leaves = 63L
  , max_depth = -1L
  , min_data_in_leaf = 1L
  , min_sum_hessian_in_leaf = 1.0
)
model <- gpb.train(params, dtrain, 10L)

tree_dt <- gpb.model.dt.tree(model)
```

---

gpb.plot.importance *Plot feature importance as a bar graph*

Description

Plot previously calculated feature importance: Gain, Cover and Frequency, as a bar graph.

Usage

```
gpb.plot.importance(tree_imp, top_n = 10L, measure = "Gain",
  left_margin = 10L, cex = NULL, ...)
```

Arguments

tree_imp	a data.table returned by gpb.importance.
top_n	maximal number of top features to include into the plot.
measure	the name of importance measure to plot, can be "Gain", "Cover" or "Frequency".
left_margin	(base R barplot) allows to adjust the left margin size to fit feature names.
cex	(base R barplot) passed as cex.names parameter to barplot. Set a number smaller than 1.0 to make the bar labels smaller than R's default and values greater than 1.0 to make them larger.
...	other parameters passed to graphics::barplot

Details

The graph represents each feature as a horizontal bar of length proportional to the defined importance of a feature. Features are shown ranked in a decreasing importance order.

Value

The gpb.plot.importance function creates a barplot and silently returns a processed data.table with top_n features sorted by defined importance.

Examples

```
data(agaricus.train, package = "gpboost")
train <- agaricus.train
dtrain <- gpb.Dataset(train$data, label = train$label)

params <- list(
    objective = "binary"
    , learning_rate = 0.1
    , min_data_in_leaf = 1L
    , min_sum_hessian_in_leaf = 1.0
)

model <- gpb.train(
    params = params
    , data = dtrain
    , nrounds = 5L
)

tree_imp <- gpb.importance(model, percentage = TRUE)
gpb.plot.importance(tree_imp, top_n = 5L, measure = "Gain")
```

---

gpb.plot.interpretation

Plot feature contribution as a bar graph

Description

Plot previously calculated feature contribution as a bar graph.

Usage

```
gpb.plot.interpretation(tree_interpretation_dt, top_n = 10L, cols = 1L,
  left_margin = 10L, cex = NULL)
```

Arguments

tree_interpretation_dt	a data.table returned by gpb.interprete.
top_n	maximal number of top features to include into the plot.
cols	the column numbers of layout, will be used only for multiclass classification feature contribution.
left_margin	(base R barplot) allows to adjust the left margin size to fit feature names.
cex	(base R barplot) passed as cex.names parameter to barplot.

Details

The graph represents each feature as a horizontal bar of length proportional to the defined contribution of a feature. Features are shown ranked in a decreasing contribution order.

Value

The `gpb.plot.interpretation` function creates a `barplot`.

Examples

```
Logit <- function(x) {
  log(x / (1.0 - x))
}
data(agaricus.train, package = "gpboost")
labels <- agaricus.train$label
dtrain <- gpb.Dataset(
  agaricus.train$data
  , label = labels
)
setinfo(dtrain, "init_score", rep(Logit(mean(labels)), length(labels)))

data(agaricus.test, package = "gpboost")

params <- list(
  objective = "binary"
  , learning_rate = 0.1
  , max_depth = -1L
  , min_data_in_leaf = 1L
  , min_sum_hessian_in_leaf = 1.0
)
model <- gpb.train(
  params = params
  , data = dtrain
  , nrounds = 5L
)

tree_interpretation <- gpb.interprete(
  model = model
  , data = agaricus.test$data
  , idxset = 1L:5L
)
gpb.plot.interpretation(
  tree_interpretation_dt = tree_interpretation[[1L]]
  , top_n = 3L
)
```

gpb.plot.part.dep.interact *Plot interaction partial dependence plots*

Description

Plot interaction partial dependence plots

Usage

```
gpb.plot.part.dep.interact(model, data, variables, n.pt.per.var = 20,
  subsample = pmin(1, n.pt.per.var^2 * 100/nrow(data)),
  discrete.variables = c(FALSE, FALSE), which.class = NULL,
  type = "filled.contour", nlevels = 20, xlab = variables[1],
  ylab = variables[2], zlab = "", main = "", return_plot_data = FALSE,
  ...)
```

Arguments

model	A gpb.Booster model object
data	A matrix with data for creating partial dependence plots
variables	A vector of length two of type string with names of the columns or integer with indices of the columns in data for which an interaction dependence plot is created
n.pt.per.var	Number of grid points per variable (used only if a variable is not discrete) For continuous variables, the two-dimensional grid for the interaction plot has dimension c(n.pt.per.var, n.pt.per.var)
subsample	Fraction of random samples in data to be used for calculating the partial dependence plot
discrete.variables	A vector of length two of type boolean. If an entry is TRUE, the evaluation grid of the corresponding variable is set to the unique values of the variable
which.class	An integer indicating the class in multi-class classification (value from 0 to num_class - 1)
type	A character string indicating the type of the plot. Supported values: "filled.contour" and "contour"
nlevels	Parameter passed to the filled.contour or contour function
xlab	Parameter passed to the filled.contour or contour function
ylab	Parameter passed to the filled.contour or contour function
zlab	Parameter passed to the filled.contour or contour function
main	Parameter passed to the filled.contour or contour function
return_plot_data	A boolean. If TRUE, the data for creating the partial dependence plot is returned
...	Additional parameters passed to the filled.contour or contour function

Value

A `list` with three entries for creating the partial dependence plot: the first two entries are `vectors` with x and y coordinates. The third is a two-dimensional `matrix` of dimension c(length(x), length(y)) with z-coordinates. This is only returned if `return_plot_data==TRUE`

Author(s)

Fabio Sigrist

Examples

```
library(gpboost)
data(GPBoost_data, package = "gpboost")
gp_model <- GPModel(group_data = group_data[,1], likelihood = "gaussian")
gpboost_model <- gpboost(data = X,
                         label = y,
                         gp_model = gp_model,
                         nrounds = 16,
                         learning_rate = 0.05,
                         max_depth = 6,
                         min_data_in_leaf = 5,
                         verbose = 0)
gpb.plot.part.dep.interact(gpboost_model, X, variables = c(1,2))
```

---

gpb.plot.partial.dependence

Plot partial dependence plots

Description

Plot partial dependence plots

Usage

```
gpb.plot.partial.dependence(model, data, variable, n.pt = 100,
  subsample = pmin(1, n.pt * 100/nrow(data)), discrete.x = FALSE,
  which.class = NULL, xlab = deparse(substitute(variable)), ylab = "",
  type = if (discrete.x) "p" else "b", main = "",
  return_plot_data = FALSE, ...)
```

Arguments

`model`	A `gpb.Booster` model object
`data`	A `matrix` with data for creating partial dependence plots
`variable`	A `string` with a name of the column or an `integer` with an index of the column in `data` for which a dependence plot is created

n.pt	Evaluation grid size (used only if x is not discrete)
subsample	Fraction of random samples in data to be used for calculating the partial dependence plot
discrete.x	A boolean. If TRUE, the evaluation grid is set to the unique values of x
which.class	An integer indicating the class in multi-class classification (value from 0 to num_class - 1)
xlab	Parameter passed to plot
ylab	Parameter passed to plot
type	Parameter passed to plot
main	Parameter passed to plot
return_plot_data	A boolean. If TRUE, the data for creating the partial dependence plot is returned
...	Additional parameters passed to plot

Value

A two-dimensional matrix with data for creating the partial dependence plot. This is only returned if return_plot_data==TRUE

Author(s)

Fabio Sigrist (adapted from a version by Michael Mayer)

Examples

```
library(gpboost)
data(GPBoost_data, package = "gpboost")

gp_model <- GPModel(group_data = group_data[,1], likelihood = "gaussian")
gpboost_model <- gpboost(data = X,
                         label = y,
                         gp_model = gp_model,
                         nrounds = 16,
                         learning_rate = 0.05,
                         max_depth = 6,
                         min_data_in_leaf = 5,
                         verbose = 0)
gpb.plot.partial.dependence(gpboost_model, X, variable = 1)
```

gpb.save *Save GPBoost model*

Description

Save GPBoost model

Usage

```
gpb.save(booster, filename, start_iteration = NULL, num_iteration = NULL,
  save_raw_data = FALSE, ...)
```

Arguments

booster	Object of class gpb.Booster
filename	saved filename
start_iteration	int or NULL, optional (default=NULL) Start index of the iteration to predict. If NULL or <= 0, starts from the first iteration.
num_iteration	int or NULL, optional (default=NULL) Limit number of iterations in the prediction. If NULL, if the best iteration exists and start_iteration is NULL or <= 0, the best iteration is used; otherwise, all iterations from start_iteration are used. If <= 0, all iterations from start_iteration are used (no limits).
save_raw_data	If TRUE, the raw data (predictor / covariate data) for the Booster is also saved. Enable this option if you want to change start_iteration or num_iteration at prediction time after loading.
...	Additional named arguments passed to the predict() method of the gpb.Booster object passed to object. This is only used when there is a gp_model and when save_raw_data=FALSE

Value

gpb.Booster

Author(s)

Fabio Sigrist, authors of the LightGBM R package

Examples

```
library(gpboost)
data(GPBoost_data, package = "gpboost")

# Train model and make prediction
gp_model <- GPModel(group_data = group_data[,1], likelihood = "gaussian")
bst <- gpboost(data = X, label = y, gp_model = gp_model, nrounds = 16,
               learning_rate = 0.05, max_depth = 6, min_data_in_leaf = 5,
```

```
                verbose = 0)
pred <- predict(bst, data = X_test, group_data_pred = group_data_test[,1],
                predict_var= TRUE, pred_latent = TRUE)
# Save model to file
filename <- tempfile(fileext = ".json")
gpb.save(bst,filename = filename)
# Load from file and make predictions again
bst_loaded <- gpb.load(filename = filename)
pred_loaded <- predict(bst_loaded, data = X_test, group_data_pred = group_data_test[,1],
                       predict_var= TRUE, pred_latent = TRUE)
# Check equality
pred$fixed_effect - pred_loaded$fixed_effect
pred$random_effect_mean - pred_loaded$random_effect_mean
pred$random_effect_cov - pred_loaded$random_effect_cov
```

`gpb.train` *Main training logic for GBPoost*

Description

Logic to train with GBPoost

Usage

```
gpb.train(params = list(), data, nrounds = 100L, gp_model = NULL,
  use_gp_model_for_validation = TRUE, train_gp_model_cov_pars = TRUE,
  valids = list(), obj = NULL, eval = NULL, verbose = 1L,
  record = TRUE, eval_freq = 1L, init_model = NULL, colnames = NULL,
  categorical_feature = NULL, early_stopping_rounds = NULL,
  callbacks = list(), reset_data = FALSE, ...)
```

Arguments

`params` list of "tuning" parameters. See the parameter documentation for more information. A few key parameters:

- `learning_rate`: The learning rate, also called shrinkage or damping parameter (default = 0.1). An important tuning parameter for boosting. Lower values usually lead to higher predictive accuracy but more boosting iterations are needed
- `num_leaves`: Number of leaves in a tree. Tuning parameter for tree-boosting (default = 31)
- `max_depth`: Maximal depth of a tree. Tuning parameter for tree-boosting (default = no limit)
- `min_data_in_leaf`: Minimal number of samples per leaf. Tuning parameter for tree-boosting (default = 20)
- `lambda_l2`: L2 regularization (default = 0)

- lambda_l1: L1 regularization (default = 0)
- max_bin: Maximal number of bins that feature values will be bucketed in (default = 255)
- line_search_step_length (default = FALSE): If TRUE, a line search is done to find the optimal step length for every boosting update (see, e.g., Friedman 2001). This is then multiplied by the learning rate
- train_gp_model_cov_pars (default = TRUE): If TRUE, the covariance parameters of the Gaussian process are estimated in every boosting iterations, otherwise the gp_model parameters are not estimated. In the latter case, you need to either estimate them beforehand or provide values via the 'init_cov_pars' parameter when creating the gp_model
- use_gp_model_for_validation (default = TRUE): If TRUE, the Gaussian process is also used (in addition to the tree model) for calculating predictions on the validation data
- leaves_newton_update (default = FALSE): Set this to TRUE to do a Newton update step for the tree leaves after the gradient step. Applies only to Gaussian process boosting (GPBoost algorithm)
- num_threads: Number of threads. For the best speed, set this to the number of real CPU cores(parallel::detectCores(logical = FALSE)), not the number of threads (most CPU using hyper-threading to generate 2 threads per CPU core).

data — a gpb.Dataset object, used for training. Some functions, such as gpb.cv, may allow you to pass other types of data like matrix and then separately supply label as a keyword argument.

nrounds — number of boosting iterations (= number of trees). This is the most important tuning parameter for boosting

gp_model — A GPModel object that contains the random effects (Gaussian process and / or grouped random effects) model

use_gp_model_for_validation — Boolean. If TRUE, the gp_model (Gaussian process and/or random effects) is also used (in addition to the tree model) for calculating predictions on the validation data. If FALSE, the gp_model (random effects part) is ignored for making predictions and only the tree ensemble is used for making predictions for calculating the validation / test error.

train_gp_model_cov_pars — Boolean. If TRUE, the covariance parameters of the gp_model (Gaussian process and/or random effects) are estimated in every boosting iterations, otherwise the gp_model parameters are not estimated. In the latter case, you need to either estimate them beforehand or provide the values via the init_cov_pars parameter when creating the gp_model

valids — a list of gpb.Dataset objects, used for validation

obj — (character) The distribution of the response variable (=label) conditional on fixed and random effects. This only needs to be set when doing independent boosting without random effects / Gaussian processes.

eval — Evaluation metric to be monitored when doing CV and parameter tuning. This can be a string, function, or list with a mixture of strings and functions.

- **a. character vector**: Non-exhaustive list of supported metrics: "test_neg_log_likelihood", "mse", "rmse", "mae", "auc", "average_precision", "binary_logloss", "binary_error" See the "metric" section of the parameter documentation for a complete list of valid metrics.
- **b. function**: You can provide a custom evaluation function. This should accept the keyword arguments `preds` and `dtrain` and should return a named list with three elements:
  - `name`: A string with the name of the metric, used for printing and storing results.
  - `value`: A single number indicating the value of the metric for the given predictions and true values
  - `higher_better`: A boolean indicating whether higher values indicate a better fit. For example, this would be `FALSE` for metrics like MAE or RMSE.
- **c. list**: If a list is given, it should only contain character vectors and functions. These should follow the requirements from the descriptions above.

`verbose` verbosity for output, if <= 0, also will disable the print of evaluation during training

`record` Boolean, TRUE will record iteration message to `booster$record_evals`

`eval_freq` evaluation output frequency, only effect when verbose > 0

`init_model` path of model file of `gpb.Booster` object, will continue training from this model

`colnames` feature names, if not null, will use this to overwrite the names in dataset

`categorical_feature`
categorical features. This can either be a character vector of feature names or an integer vector with the indices of the features (e.g. `c(1L, 10L)` to say "the first and tenth columns").

`early_stopping_rounds`
int. Activates early stopping. Requires at least one validation data and one metric. When this parameter is non-null, training will stop if the evaluation of any metric on any validation set fails to improve for `early_stopping_rounds` consecutive boosting rounds. If training stops early, the returned model will have attribute `best_iter` set to the iteration number of the best iteration.

`callbacks` List of callback functions that are applied at each iteration.

`reset_data` Boolean, setting it to TRUE (not the default value) will transform the booster model into a predictor model which frees up memory and the original datasets

`...` other parameters, see the parameter documentation for more information.

Value

a trained booster model `gpb.Booster`.

Early Stopping

"early stopping" refers to stopping the training process if the model's performance on a given validation set does not improve for several consecutive iterations.

If multiple arguments are given to eval, their order will be preserved. If you enable early stopping by setting early_stopping_rounds in params, by default all metrics will be considered for early stopping.

If you want to only consider the first metric for early stopping, pass first_metric_only = TRUE in params. Note that if you also specify metric in params, that metric will be considered the "first" one. If you omit metric, a default metric will be used based on your choice for the parameter obj (keyword argument) or objective (passed into params).

Author(s)

Fabio Sigrist, authors of the LightGBM R package

Examples

```
# See https://github.com/fabsig/GPBoost/tree/master/R-package for more examples


library(gpboost)
data(GPBoost_data, package = "gpboost")

#--------------------Combine tree-boosting and grouped random effects model----------------
# Create random effects model
gp_model <- GPModel(group_data = group_data[,1], likelihood = "gaussian")
# The default optimizer for covariance parameters (hyperparameters) is
# Nesterov-accelerated gradient descent.
# This can be changed to, e.g., Nelder-Mead as follows:
# re_params <- list(optimizer_cov = "nelder_mead")
# gp_model$set_optim_params(params=re_params)
# Use trace = TRUE to monitor convergence:
# re_params <- list(trace = TRUE)
# gp_model$set_optim_params(params=re_params)
dtrain <- gpb.Dataset(data = X, label = y)
# Train model
bst <- gpb.train(data = dtrain, gp_model = gp_model, nrounds = 16,
                 learning_rate = 0.05, max_depth = 6, min_data_in_leaf = 5,
                 verbose = 0)
# Estimated random effects model
summary(gp_model)
# Make predictions
pred <- predict(bst, data = X_test, group_data_pred = group_data_test[,1],
                predict_var= TRUE)
pred$random_effect_mean # Predicted mean
pred$random_effect_cov # Predicted variances
pred$fixed_effect # Predicted fixed effect from tree ensemble
# Sum them up to otbain a single prediction
pred$random_effect_mean + pred$fixed_effect

#--------------------Combine tree-boosting and Gaussian process model----------------
# Create Gaussian process model
gp_model <- GPModel(gp_coords = coords, cov_function = "exponential",
                    likelihood = "gaussian")
# Train model
```

```
dtrain <- gpb.Dataset(data = X, label = y)
bst <- gpb.train(data = dtrain, gp_model = gp_model, nrounds = 16,
                 learning_rate = 0.05, max_depth = 6, min_data_in_leaf = 5,
                 verbose = 0)
# Estimated random effects model
summary(gp_model)
# Make predictions
pred <- predict(bst, data = X_test, gp_coords_pred = coords_test,
                predict_cov_mat =TRUE)
pred$random_effect_mean # Predicted (posterior) mean of GP
pred$random_effect_cov # Predicted (posterior) covariance matrix of GP
pred$fixed_effect # Predicted fixed effect from tree ensemble
# Sum them up to otbain a single prediction
pred$random_effect_mean + pred$fixed_effect


#--------------------Using validation data-------------------------
set.seed(1)
train_ind <- sample.int(length(y),size=250)
dtrain <- gpb.Dataset(data = X[train_ind,], label = y[train_ind])
dtest <- gpb.Dataset.create.valid(dtrain, data = X[-train_ind,], label = y[-train_ind])
valids <- list(test = dtest)
gp_model <- GPModel(group_data = group_data[train_ind,1], likelihood="gaussian")
# Need to set prediction data for gp_model
gp_model$set_prediction_data(group_data_pred = group_data[-train_ind,1])
# Training with validation data and use_gp_model_for_validation = TRUE
bst <- gpb.train(data = dtrain, gp_model = gp_model, nrounds = 100,
                 learning_rate = 0.05, max_depth = 6, min_data_in_leaf = 5,
                 verbose = 1, valids = valids,
                 early_stopping_rounds = 10, use_gp_model_for_validation = TRUE)
print(paste0("Optimal number of iterations: ", bst$best_iter,
             ", best test error: ", bst$best_score))
# Plot validation error
val_error <- unlist(bst$record_evals$test$l2$eval)
plot(1:length(val_error), val_error, type="l", lwd=2, col="blue",
   xlab="iteration", ylab="Validation error", main="Validation error vs. boosting iteration")


#--------------------Do Newton updates for tree leaves---------------
# Note: run the above examples first
bst <- gpb.train(data = dtrain, gp_model = gp_model, nrounds = 100,
                 learning_rate = 0.05, max_depth = 6, min_data_in_leaf = 5,
                 verbose = 1, valids = valids,
                 early_stopping_rounds = 5, use_gp_model_for_validation = FALSE,
                 leaves_newton_update = TRUE)
print(paste0("Optimal number of iterations: ", bst$best_iter,
             ", best test error: ", bst$best_score))
# Plot validation error
val_error <- unlist(bst$record_evals$test$l2$eval)
plot(1:length(val_error), val_error, type="l", lwd=2, col="blue",
   xlab="iteration", ylab="Validation error", main="Validation error vs. boosting iteration")
```

```
#--------------------GPBoostOOS algorithm: GP parameters estimated out-of-sample----------------
# Create random effects model and dataset
gp_model <- GPModel(group_data = group_data[,1], likelihood="gaussian")
dtrain <- gpb.Dataset(X, label = y)
params <- list(learning_rate = 0.05,
               max_depth = 6,
               min_data_in_leaf = 5)
# Stage 1: run cross-validation to (i) determine to optimal number of iterations
#           and (ii) to estimate the GPModel on the out-of-sample data
cvbst <- gpb.cv(params = params,
                data = dtrain,
                gp_model = gp_model,
                nrounds = 100,
                nfold = 4,
                eval = "l2",
                early_stopping_rounds = 5,
                use_gp_model_for_validation = TRUE,
                fit_GP_cov_pars_OOS = TRUE)
print(paste0("Optimal number of iterations: ", cvbst$best_iter))
# Estimated random effects model
# Note: ideally, one would have to find the optimal combination of
#               other tuning parameters such as the learning rate, tree depth, etc.)
summary(gp_model)
# Stage 2: Train tree-boosting model while holding the GPModel fix
bst <- gpb.train(data = dtrain,
                 gp_model = gp_model,
                 nrounds = cvbst$best_iter,
                 learning_rate = 0.05,
                 max_depth = 6,
                 min_data_in_leaf = 5,
                 verbose = 0,
                 train_gp_model_cov_pars = FALSE)
# The GPModel has not changed:
summary(gp_model)
```

---

gpboost *Train a GPBoost model*

Description

Simple interface for training a GPBoost model.

Usage

```
gpboost(data, label = NULL, weight = NULL, params = list(),
  nrounds = 100L, gp_model = NULL, line_search_step_length = FALSE,
  use_gp_model_for_validation = TRUE, train_gp_model_cov_pars = TRUE,
  valids = list(), obj = NULL, eval = NULL, verbose = 1L,
  record = TRUE, eval_freq = 1L, early_stopping_rounds = NULL,
```

```
  init_model = NULL, colnames = NULL, categorical_feature = NULL,
  callbacks = list(), ...)
```

Arguments

data	a gpb.Dataset object, used for training. Some functions, such as gpb.cv, may allow you to pass other types of data like matrix and then separately supply label as a keyword argument.
label	Vector of response values / labels, used if data is not an gpb.Dataset
weight	Vector of weights. The GPBoost algorithm currently does not support weights
params	list of "tuning" parameters. See the parameter documentation for more information. A few key parameters: • learning_rate: The learning rate, also called shrinkage or damping parameter (default = 0.1). An important tuning parameter for boosting. Lower values usually lead to higher predictive accuracy but more boosting iterations are needed • num_leaves: Number of leaves in a tree. Tuning parameter for tree-boosting (default = 31) • max_depth: Maximal depth of a tree. Tuning parameter for tree-boosting (default = no limit) • min_data_in_leaf: Minimal number of samples per leaf. Tuning parameter for tree-boosting (default = 20) • lambda_l2: L2 regularization (default = 0) • lambda_l1: L1 regularization (default = 0) • max_bin: Maximal number of bins that feature values will be bucketed in (default = 255) • line_search_step_length (default = FALSE): If TRUE, a line search is done to find the optimal step length for every boosting update (see, e.g., Friedman 2001). This is then multiplied by the learning rate • train_gp_model_cov_pars (default = TRUE): If TRUE, the covariance parameters of the Gaussian process are estimated in every boosting iterations, otherwise the gp_model parameters are not estimated. In the latter case, you need to either estimate them beforehand or provide values via the 'init_cov_pars' parameter when creating the gp_model • use_gp_model_for_validation (default = TRUE): If TRUE, the Gaussian process is also used (in addition to the tree model) for calculating predictions on the validation data • leaves_newton_update (default = FALSE): Set this to TRUE to do a Newton update step for the tree leaves after the gradient step. Applies only to Gaussian process boosting (GPBoost algorithm) • num_threads: Number of threads. For the best speed, set this to the number of real CPU cores(parallel::detectCores(logical = FALSE)), not the number of threads (most CPU using hyper-threading to generate 2 threads per CPU core).
nrounds	number of boosting iterations (= number of trees). This is the most important tuning parameter for boosting

`gp_model` A `GPModel` object that contains the random effects (Gaussian process and / or grouped random effects) model

`line_search_step_length`
Boolean. If TRUE, a line search is done to find the optimal step length for every boosting update (see, e.g., Friedman 2001). This is then multiplied by the `learning_rate`. Applies only to the GPBoost algorithm

`use_gp_model_for_validation`
Boolean. If TRUE, the `gp_model` (Gaussian process and/or random effects) is also used (in addition to the tree model) for calculating predictions on the validation data. If FALSE, the `gp_model` (random effects part) is ignored for making predictions and only the tree ensemble is used for making predictions for calculating the validation / test error.

`train_gp_model_cov_pars`
Boolean. If TRUE, the covariance parameters of the `gp_model` (Gaussian process and/or random effects) are estimated in every boosting iterations, otherwise the `gp_model` parameters are not estimated. In the latter case, you need to either estimate them beforehand or provide the values via the `init_cov_pars` parameter when creating the `gp_model`

`valids` a list of `gpb.Dataset` objects, used for validation

`obj` (character) The distribution of the response variable (=label) conditional on fixed and random effects. This only needs to be set when doing independent boosting without random effects / Gaussian processes.

`eval` Evaluation metric to be monitored when doing CV and parameter tuning. This can be a string, function, or list with a mixture of strings and functions.

- **a. character vector**: Non-exhaustive list of supported metrics: "test_neg_log_likelihood", "mse", "rmse", "mae", "auc", "average_precision", "binary_logloss", "binary_error" See the "metric" section of the parameter documentation for a complete list of valid metrics.
- **b. function**: You can provide a custom evaluation function. This should accept the keyword arguments `preds` and `dtrain` and should return a named list with three elements:
  - `name`: A string with the name of the metric, used for printing and storing results.
  - `value`: A single number indicating the value of the metric for the given predictions and true values
  - `higher_better`: A boolean indicating whether higher values indicate a better fit. For example, this would be `FALSE` for metrics like MAE or RMSE.
- **c. list**: If a list is given, it should only contain character vectors and functions. These should follow the requirements from the descriptions above.

`verbose` verbosity for output, if <= 0, also will disable the print of evaluation during training

`record` Boolean, TRUE will record iteration message to `booster$record_evals`

`eval_freq` evaluation output frequency, only effect when verbose > 0

early_stopping_rounds
: int. Activates early stopping. Requires at least one validation data and one metric. When this parameter is non-null, training will stop if the evaluation of any metric on any validation set fails to improve for `early_stopping_rounds` consecutive boosting rounds. If training stops early, the returned model will have attribute `best_iter` set to the iteration number of the best iteration.

init_model
: path of model file of `gpb.Booster` object, will continue training from this model

colnames
: feature names, if not null, will use this to overwrite the names in dataset

categorical_feature
: categorical features. This can either be a character vector of feature names or an integer vector with the indices of the features (e.g. `c(1L, 10L)` to say "the first and tenth columns").

callbacks
: List of callback functions that are applied at each iteration.

...
: Additional arguments passed to `gpb.train`. For example

- `valids`: a list of `gpb.Dataset` objects, used for validation
- `eval`: evaluation function, can be (a list of) character or custom eval function
- `record`: Boolean, TRUE will record iteration message to `booster$record_evals`
- `colnames`: feature names, if not null, will use this to overwrite the names in dataset
- `categorical_feature`: categorical features. This can either be a character vector of feature names or an integer vector with the indices of the features (e.g. `c(1L, 10L)` to say "the first and tenth columns").
- `reset_data`: Boolean, setting it to TRUE (not the default value) will transform the booster model into a predictor model which frees up memory and the original datasets

Value

a trained `gpb.Booster`

Early Stopping

"early stopping" refers to stopping the training process if the model's performance on a given validation set does not improve for several consecutive iterations.

If multiple arguments are given to `eval`, their order will be preserved. If you enable early stopping by setting `early_stopping_rounds` in `params`, by default all metrics will be considered for early stopping.

If you want to only consider the first metric for early stopping, pass `first_metric_only = TRUE` in `params`. Note that if you also specify `metric` in `params`, that metric will be considered the "first" one. If you omit `metric`, a default metric will be used based on your choice for the parameter `obj` (keyword argument) or `objective` (passed into `params`).

Author(s)

Fabio Sigrist, authors of the LightGBM R package

Examples

```
# See https://github.com/fabsig/GPBoost/tree/master/R-package for more examples


library(gpboost)
data(GPBoost_data, package = "gpboost")

#--------------------Combine tree-boosting and grouped random effects model----------------
# Create random effects model
gp_model <- GPModel(group_data = group_data[,1], likelihood = "gaussian")
# The default optimizer for covariance parameters (hyperparameters) is
# Nesterov-accelerated gradient descent.
# This can be changed to, e.g., Nelder-Mead as follows:
# re_params <- list(optimizer_cov = "nelder_mead")
# gp_model$set_optim_params(params=re_params)
# Use trace = TRUE to monitor convergence:
# re_params <- list(trace = TRUE)
# gp_model$set_optim_params(params=re_params)

# Train model
bst <- gpboost(data = X, label = y, gp_model = gp_model, nrounds = 16,
               learning_rate = 0.05, max_depth = 6, min_data_in_leaf = 5,
               verbose = 0)
# Estimated random effects model
summary(gp_model)

# Make predictions
# Predict latent variables
pred <- predict(bst, data = X_test, group_data_pred = group_data_test[,1],
                predict_var = TRUE, pred_latent = TRUE)
pred$random_effect_mean # Predicted latent random effects mean
pred$random_effect_cov # Predicted random effects variances
pred$fixed_effect # Predicted fixed effects from tree ensemble
# Predict response variable
pred_resp <- predict(bst, data = X_test, group_data_pred = group_data_test[,1],
                     predict_var = TRUE, pred_latent = FALSE)
pred_resp$response_mean # Predicted response mean
# For Gaussian data: pred$random_effect_mean + pred$fixed_effect = pred_resp$response_mean
pred$random_effect_mean + pred$fixed_effect - pred_resp$response_mean

#--------------------Combine tree-boosting and Gaussian process model----------------
# Create Gaussian process model
gp_model <- GPModel(gp_coords = coords, cov_function = "exponential",
                    likelihood = "gaussian")
# Train model
bst <- gpboost(data = X, label = y, gp_model = gp_model, nrounds = 8,
               learning_rate = 0.1, max_depth = 6, min_data_in_leaf = 5,
               verbose = 0)
# Estimated random effects model
summary(gp_model)
# Make predictions
pred <- predict(bst, data = X_test, gp_coords_pred = coords_test,
```

```
                predict_var = TRUE, pred_latent = TRUE)
pred$random_effect_mean # Predicted latent random effects mean
pred$random_effect_cov # Predicted random effects variances
pred$fixed_effect # Predicted fixed effects from tree ensemble
# Predict response variable
pred_resp <- predict(bst, data = X_test, gp_coords_pred = coords_test,
                     predict_var = TRUE, pred_latent = FALSE)
pred_resp$response_mean # Predicted response mean
```

`GPBoost_data` *Example data for the GPBoost package*

Description

Simulated example data for the GPBoost package This data set includes the following fields:

- `y`: response variable
- `X`: a matrix with covariate information
- `group_data`: a matrix with categorical grouping variables
- `coords`: a matrix with spatial coordinates
- `X_test`: a matrix with covariate information for predictions
- `group_data_test`: a matrix with categorical grouping variables for predictions
- `coords_test`: a matrix with spatial coordinates for predictions

Usage

```
data(GPBoost_data)
```

`GPModel` *Create a* `GPModel` *object*

Description

Create a `GPModel` which contains a Gaussian process and / or mixed effects model with grouped random effects

Usage

```
GPModel(likelihood = "gaussian", group_data = NULL,
  group_rand_coef_data = NULL, ind_effect_group_rand_coef = NULL,
  drop_intercept_group_rand_effect = NULL, gp_coords = NULL,
  gp_rand_coef_data = NULL, cov_function = "exponential",
  cov_fct_shape = 0.5, gp_approx = "none", cov_fct_taper_range = 1,
  cov_fct_taper_shape = 0, num_neighbors = 20L,
  vecchia_ordering = "random", ind_points_selection = "kmeans++",
  num_ind_points = 500L, cover_tree_radius = 1,
  matrix_inversion_method = "cholesky", seed = 0L, cluster_ids = NULL,
  free_raw_data = FALSE, vecchia_approx = NULL, vecchia_pred_type = NULL,
  num_neighbors_pred = NULL)
```

Arguments

`likelihood` A `string` specifying the likelihood function (distribution) of the response variable. Available options:

- "gaussian"
- "bernoulli_probit": binary data with Bernoulli likelihood and a probit link function
- "bernoulli_logit": binary data with Bernoulli likelihood and a logit link function
- "gamma": gamma distribution with a with log link function
- "poisson": Poisson distribution with a with log link function
- "negative_binomial": negative binomial distribution with a with log link function
- Note: other likelihoods could be implemented upon request

`group_data` A `vector` or `matrix` whose columns are categorical grouping variables. The elements being group levels defining grouped random effects. The elements of 'group_data' can be integer, double, or character. The number of columns corresponds to the number of grouped (intercept) random effects

`group_rand_coef_data` A `vector` or `matrix` with numeric covariate data for grouped random coefficients

`ind_effect_group_rand_coef` A `vector` with integer indices that indicate the corresponding categorical grouping variable (=columns) in 'group_data' for every covariate in 'group_rand_coef_data'. Counting starts at 1. The length of this index vector must equal the number of covariates in 'group_rand_coef_data'. For instance, c(1,1,2) means that the first two covariates (=first two columns) in 'group_rand_coef_data' have random coefficients corresponding to the first categorical grouping variable (=first column) in 'group_data', and the third covariate (=third column) in 'group_rand_coef_data' has a random coefficient corresponding to the second grouping variable (=second column) in 'group_data'

`drop_intercept_group_rand_effect` A `vector` of type `logical` (boolean). Indicates whether intercept random effects are dropped (only for random coefficients). If drop_intercept_group_rand_effect[k]

is TRUE, the intercept random effect number k is dropped / not included. Only random effects with random slopes can be dropped.

`gp_coords` A `matrix` with numeric coordinates (= inputs / features) for defining Gaussian processes

`gp_rand_coef_data` A `vector` or `matrix` with numeric covariate data for Gaussian process random coefficients

`cov_function` A `string` specifying the covariance function for the Gaussian process. Available options:

- "exponential": Exponential covariance function (using the parametrization of Diggle and Ribeiro, 2007)
- "gaussian": Gaussian, aka squared exponential, covariance function (using the parametrization of Diggle and Ribeiro, 2007)
- "matern": Matern covariance function with the smoothness specified by the `cov_fct_shape` parameter (using the parametrization of Rasmussen and Williams, 2006)
- "powered_exponential": powered exponential covariance function with the exponent specified by the `cov_fct_shape` parameter (using the parametrization of Diggle and Ribeiro, 2007)
- "wendland": Compactly supported Wendland covariance function (using the parametrization of Bevilacqua et al., 2019, AOS)
- "matern_space_time": Spatio-temporal Matern covariance function with different range parameters for space and time. Note that the first column in `gp_coords` must correspond to the time dimension
- "matern_ard": anisotropic Matern covariance function with Automatic Relevance Determination (ARD), i.e., with a different range parameter for every coordinate dimension / column of `gp_coords`
- "gaussian_ard": anisotropic Gaussian, aka squared exponential, covariance function with Automatic Relevance Determination (ARD), i.e., with a different range parameter for every coordinate dimension / column of `gp_coords`

`cov_fct_shape` A `numeric` specifying the shape parameter of the covariance function (=smoothness parameter for Matern covariance) This parameter is irrelevant for some covariance functions such as the exponential or Gaussian

`gp_approx` A `string` specifying the large data approximation for Gaussian processes. Available options:

- "none": No approximation
- "vecchia": A Vecchia approximation; see Sigrist (2022, JMLR) for more details
- "tapering": The covariance function is multiplied by a compactly supported Wendland correlation function
- "fitc": Fully Independent Training Conditional approximation aka modified predictive process approximation; see Gyger, Furrer, and Sigrist (2024) for more details
- "full_scale_tapering": A full scale approximation combining an inducing point / predictive process approximation with tapering on the residual process; see Gyger, Furrer, and Sigrist (2024) for more details

`cov_fct_taper_range`
: A `numeric` specifying the range parameter of the Wendland covariance function and Wendland correlation taper function. We follow the notation of Bevilacqua et al. (2019, AOS)

`cov_fct_taper_shape`
: A `numeric` specifying the shape (=smoothness) parameter of the Wendland covariance function and Wendland correlation taper function. We follow the notation of Bevilacqua et al. (2019, AOS)

`num_neighbors`
: An `integer` specifying the number of neighbors for the Vecchia approximation. Note: for prediction, the number of neighbors can be set through the 'num_neighbors_pred' parameter in the 'set_prediction_data' function. By default, num_neighbors_pred = 2 * num_neighbors. Further, the type of Vecchia approximation used for making predictions is set through the 'vecchia_pred_type' parameter in the 'set_prediction_data' function

`vecchia_ordering`
: A `string` specifying the ordering used in the Vecchia approximation. Available options:
  - "none": the default ordering in the data is used
  - "random": a random ordering
  - "time": ordering accorrding to time (only for space-time models)
  - "time_random_space": ordering according to time and randomly for all spatial points with the same time points (only for space-time models)

`ind_points_selection`
: A `string` specifying the method for choosing inducing points Available options:
  - "kmeans++: the k-means++ algorithm
  - "cover_tree": the cover tree algorithm
  - "random": random selection from data points

`num_ind_points`
: An `integer` specifying the number of inducing points / knots for, e.g., a predictive process approximation

`cover_tree_radius`
: A `numeric` specifying the radius (= "spatial resolution") for the cover tree algorithm

`matrix_inversion_method`
: A `string` specifying the method used for inverting covariance matrices. Available options:
  - "cholesky": Cholesky factorization
  - "iterative": iterative methods. A combination of conjugate gradient, Lanczos algorithm, and other methods.
    This is currently only supported for the following cases:
    - likelihood != "gaussian" and gp_approx == "vecchia" (non-Gaussian likelihoods with a Vecchia-Laplace approximation)
    - likelihood == "gaussian" and gp_approx == "full_scale_tapering" (Gaussian likelihood with a full-scale tapering approximation)

`seed`
: An `integer` specifying the seed used for model creation (e.g., random ordering in Vecchia approximation)

cluster_ids A vector with elements indicating independent realizations of random effects / Gaussian processes (same values = same process realization). The elements of 'cluster_ids' can be integer, double, or character.

free_raw_data A boolean. If TRUE, the data (groups, coordinates, covariate data for random coefficients) is freed in R after initialization

vecchia_approx Discontinued. Use the argument gp_approx instead

vecchia_pred_type
A string specifying the type of Vecchia approximation used for making predictions. This is discontinued here. Use the function 'set_prediction_data' to specify this

num_neighbors_pred
an integer specifying the number of neighbors for making predictions. This is discontinued here. Use the function 'set_prediction_data' to specify this

Value

A GPModel containing ontains a Gaussian process and / or mixed effects model with grouped random effects

Author(s)

Fabio Sigrist

Examples

```
# See https://github.com/fabsig/GPBoost/tree/master/R-package for more examples

data(GPBoost_data, package = "gpboost")

#--------------------Grouped random effects model: single-level random effect----------------
gp_model <- GPModel(group_data = group_data[,1], likelihood="gaussian")

#--------------------Gaussian process model----------------
gp_model <- GPModel(gp_coords = coords, cov_function = "exponential",
                    likelihood="gaussian")

#--------------------Combine Gaussian process with grouped random effects----------------
gp_model <- GPModel(group_data = group_data,
                    gp_coords = coords, cov_function = "exponential",
                    likelihood="gaussian")
```

GPModel_shared_params *Documentation for parameters shared by* GPModel*,* gpb.cv*, and* gpboost

---

Description

Documentation for parameters shared by GPModel, gpb.cv, and gpboost

Arguments

`likelihood` A `string` specifying the likelihood function (distribution) of the response variable. Available options:

- "gaussian"
- "bernoulli_probit": binary data with Bernoulli likelihood and a probit link function
- "bernoulli_logit": binary data with Bernoulli likelihood and a logit link function
- "gamma": gamma distribution with a with log link function
- "poisson": Poisson distribution with a with log link function
- "negative_binomial": negative binomial distribution with a with log link function
- Note: other likelihoods could be implemented upon request

`group_data` A `vector` or `matrix` whose columns are categorical grouping variables. The elements being group levels defining grouped random effects. The elements of 'group_data' can be integer, double, or character. The number of columns corresponds to the number of grouped (intercept) random effects

`group_rand_coef_data` A `vector` or `matrix` with numeric covariate data for grouped random coefficients

`ind_effect_group_rand_coef` A `vector` with integer indices that indicate the corresponding categorical grouping variable (=columns) in 'group_data' for every covariate in 'group_rand_coef_data'. Counting starts at 1. The length of this index vector must equal the number of covariates in 'group_rand_coef_data'. For instance, c(1,1,2) means that the first two covariates (=first two columns) in 'group_rand_coef_data' have random coefficients corresponding to the first categorical grouping variable (=first column) in 'group_data', and the third covariate (=third column) in 'group_rand_coef_data' has a random coefficient corresponding to the second grouping variable (=second column) in 'group_data'

`drop_intercept_group_rand_effect` A `vector` of type `logical` (boolean). Indicates whether intercept random effects are dropped (only for random coefficients). If drop_intercept_group_rand_effect[k] is TRUE, the intercept random effect number k is dropped / not included. Only random effects with random slopes can be dropped.

`gp_coords` A `matrix` with numeric coordinates (= inputs / features) for defining Gaussian processes

`gp_rand_coef_data` A `vector` or `matrix` with numeric covariate data for Gaussian process random coefficients

`cov_function` A `string` specifying the covariance function for the Gaussian process. Available options:

- "exponential": Exponential covariance function (using the parametrization of Diggle and Ribeiro, 2007)

- "gaussian": Gaussian, aka squared exponential, covariance function (using the parametrization of Diggle and Ribeiro, 2007)
- "matern": Matern covariance function with the smoothness specified by the `cov_fct_shape` parameter (using the parametrization of Rasmussen and Williams, 2006)
- "powered_exponential": powered exponential covariance function with the exponent specified by the `cov_fct_shape` parameter (using the parametrization of Diggle and Ribeiro, 2007)
- "wendland": Compactly supported Wendland covariance function (using the parametrization of Bevilacqua et al., 2019, AOS)
- "matern_space_time": Spatio-temporal Matern covariance function with different range parameters for space and time. Note that the first column in `gp_coords` must correspond to the time dimension
- "matern_ard": anisotropic Matern covariance function with Automatic Relevance Determination (ARD), i.e., with a different range parameter for every coordinate dimension / column of `gp_coords`
- "gaussian_ard": anisotropic Gaussian, aka squared exponential, covariance function with Automatic Relevance Determination (ARD), i.e., with a different range parameter for every coordinate dimension / column of `gp_coords`

`cov_fct_shape` A `numeric` specifying the shape parameter of the covariance function (=smoothness parameter for Matern covariance) This parameter is irrelevant for some covariance functions such as the exponential or Gaussian

`gp_approx` A `string` specifying the large data approximation for Gaussian processes. Available options:

- "none": No approximation
- "vecchia": A Vecchia approximation; see Sigrist (2022, JMLR) for more details
- "tapering": The covariance function is multiplied by a compactly supported Wendland correlation function
- "fitc": Fully Independent Training Conditional approximation aka modified predictive process approximation; see Gyger, Furrer, and Sigrist (2024) for more details
- "full_scale_tapering": A full scale approximation combining an inducing point / predictive process approximation with tapering on the residual process; see Gyger, Furrer, and Sigrist (2024) for more details

`cov_fct_taper_range` A `numeric` specifying the range parameter of the Wendland covariance function and Wendland correlation taper function. We follow the notation of Bevilacqua et al. (2019, AOS)

`cov_fct_taper_shape` A `numeric` specifying the shape (=smoothness) parameter of the Wendland covariance function and Wendland correlation taper function. We follow the notation of Bevilacqua et al. (2019, AOS)

`num_neighbors` An `integer` specifying the number of neighbors for the Vecchia approximation. Note: for prediction, the number of neighbors can be set through the

'num_neighbors_pred' parameter in the 'set_prediction_data' function. By default, num_neighbors_pred = 2 * num_neighbors. Further, the type of Vecchia approximation used for making predictions is set through the 'vecchia_pred_type' parameter in the 'set_prediction_data' function

`vecchia_ordering`
: A `string` specifying the ordering used in the Vecchia approximation. Available options:
  - "none": the default ordering in the data is used
  - "random": a random ordering
  - "time": ordering accorrding to time (only for space-time models)
  - "time_random_space": ordering according to time and randomly for all spatial points with the same time points (only for space-time models)

`ind_points_selection`
: A `string` specifying the method for choosing inducing points Available options:
  - "kmeans++: the k-means++ algorithm
  - "cover_tree": the cover tree algorithm
  - "random": random selection from data points

`num_ind_points`
: An `integer` specifying the number of inducing points / knots for, e.g., a predictive process approximation

`cover_tree_radius`
: A `numeric` specifying the radius (= "spatial resolution") for the cover tree algorithm

`matrix_inversion_method`
: A `string` specifying the method used for inverting covariance matrices. Available options:
  - "cholesky": Cholesky factorization
  - "iterative": iterative methods. A combination of conjugate gradient, Lanczos algorithm, and other methods.
    This is currently only supported for the following cases:
    - likelihood != "gaussian" and gp_approx == "vecchia" (non-Gaussian likelihoods with a Vecchia-Laplace approximation)
    - likelihood == "gaussian" and gp_approx == "full_scale_tapering" (Gaussian likelihood with a full-scale tapering approximation)

`seed`
: An `integer` specifying the seed used for model creation (e.g., random ordering in Vecchia approximation)

`vecchia_pred_type`
: A `string` specifying the type of Vecchia approximation used for making predictions. Default value if vecchia_pred_type = NULL: "order_obs_first_cond_obs_only". Available options:
  - "order_obs_first_cond_obs_only": Vecchia approximation for the observable process and observed training data is ordered first and the neighbors are only observed training data points
  - "order_obs_first_cond_all": Vecchia approximation for the observable process and observed training data is ordered first and the neighbors are selected among all points (training + prediction)

- "latent_order_obs_first_cond_obs_only": Vecchia approximation for the latent process and observed data is ordered first and neighbors are only observed points
- "latent_order_obs_first_cond_all": Vecchia approximation for the latent process and observed data is ordered first and neighbors are selected among all points
- "order_pred_first": Vecchia approximation for the observable process and prediction data is ordered first for making predictions. This option is only available for Gaussian likelihoods

`num_neighbors_pred`
: an `integer` specifying the number of neighbors for the Vecchia approximation for making predictions. Default value if NULL: num_neighbors_pred = 2 * num_neighbors

`cg_delta_conv_pred`
: a `numeric` specifying the tolerance level for L2 norm of residuals for checking convergence in conjugate gradient algorithms when being used for prediction Default value if NULL: 1e-3

`nsim_var_pred`
: an `integer` specifying the number of samples when simulation is used for calculating predictive variances Default value if NULL: 1000

`rank_pred_approx_matrix_lanczos`
: an `integer` specifying the rank of the matrix for approximating predictive covariances obtained using the Lanczos algorithm Default value if NULL: 1000

`cluster_ids`
: A `vector` with elements indicating independent realizations of random effects / Gaussian processes (same values = same process realization). The elements of 'cluster_ids' can be integer, double, or character.

`free_raw_data`
: A `boolean`. If TRUE, the data (groups, coordinates, covariate data for random coefficients) is freed in R after initialization

`y`
: A `vector` with response variable data

`X`
: A `matrix` with numeric covariate data for the fixed effects linear regression term (if there is one)

`params`
: A `list` with parameters for the estimation / optimization

- optimizer_cov: `string` (default = "lbfgs"). Optimizer used for estimating covariance parameters. Options: "gradient_descent", "lbfgs", "fisher_scoring", "newton", "nelder_mead", "adam". If there are additional auxiliary parameters for non-Gaussian likelihoods, 'optimizer_cov' is also used for those
- optimizer_coef: `string` (default = "wls" for Gaussian likelihoods and "lbfgs" for other likelihoods). Optimizer used for estimating linear regression coefficients, if there are any (for the GPBoost algorithm there are usually none). Options: "gradient_descent", "lbfgs", "wls", "nelder_mead", "adam". Gradient descent steps are done simultaneously with gradient descent steps for the covariance parameters. "wls" refers to doing coordinate descent for the regression coefficients using weighted least squares. If 'optimizer_cov' is set to "nelder_mead", "lbfgs", or "adam", 'optimizer_coef' is automatically also set to the same value.
- maxit: `integer` (default = 1000). Maximal number of iterations for optimization algorithm

- delta_rel_conv: `numeric` (default = 1E-6 except for "nelder_mead" for which the default is 1E-8). Convergence tolerance. The algorithm stops if the relative change in either the (approximate) log-likelihood or the parameters is below this value. For "adam", the L2 norm of the gradient is used instead of the relative change in the log-likelihood. If < 0, internal default values are used
- convergence_criterion: `string` (default = "relative_change_in_log_likelihood"). The convergence criterion used for terminating the optimization algorithm. Options: "relative_change_in_log_likelihood" or "relative_change_in_parameters"
- init_coef: `vector` with `numeric` elements (default = NULL). Initial values for the regression coefficients (if there are any, can be NULL)
- init_cov_pars: `vector` with `numeric` elements (default = NULL). Initial values for covariance parameters of Gaussian process and random effects (can be NULL). The order it the same as the order of the parameters in the summary function: first is the error variance (only for "gaussian" likelihood), next follow the variances of the grouped random effects (if there are any, in the order provided in 'group_data'), and then follow the marginal variance and the range of the Gaussian process. If there are multiple Gaussian processes, then the variances and ranges follow alternatingly. If 'init_cov_pars = NULL', an internal choice is used that depends on the likelihood and the random effects type and covariance function. If you select the option 'trace = TRUE' in the 'params' argument, you will see the first initial covariance parameters in iteration 0.
- lr_coef: `numeric` (default = 0.1). Learning rate for fixed effect regression coefficients if gradient descent is used
- lr_cov: `numeric` (default = 0.1 for "gradient_descent" and 1. otherwise). Initial learning rate for covariance parameters if a gradient-based optimization method is used
  - If lr_cov < 0, internal default values are used (0.1 for "gradient_descent" and 1. otherwise)
  - If there are additional auxiliary parameters for non-Gaussian likelihoods, 'lr_cov' is also used for those
  - For "lbfgs", this is divided by the norm of the gradient in the first iteration
- use_nesterov_acc: `boolean` (default = TRUE). If TRUE Nesterov acceleration is used. This is used only for gradient descent
- acc_rate_coef: `numeric` (default = 0.5). Acceleration rate for regression coefficients (if there are any) for Nesterov acceleration
- acc_rate_cov: `numeric` (default = 0.5). Acceleration rate for covariance parameters for Nesterov acceleration
- momentum_offset: `integer` (Default = 2). Number of iterations for which no momentum is applied in the beginning.
- trace: `boolean` (default = FALSE). If TRUE, information on the progress of the parameter optimization is printed
- std_dev: `boolean` (default = TRUE). If TRUE, approximate standard deviations are calculated for the covariance and linear regression parameters (=

square root of diagonal of the inverse Fisher information for Gaussian likelihoods and square root of diagonal of a numerically approximated inverse Hessian for non-Gaussian likelihoods)

- init_aux_pars: `vector` with `numeric` elements (default = NULL). Initial values for additional parameters for non-Gaussian likelihoods (e.g., shape parameter of a gamma or negative_binomial likelihood)
- estimate_aux_pars: `boolean` (default = TRUE). If TRUE, additional parameters for non-Gaussian likelihoods are also estimated (e.g., shape parameter of a gamma or negative_binomial likelihood)
- cg_max_num_it: `integer` (default = 1000). Maximal number of iterations for conjugate gradient algorithms
- cg_max_num_it_tridiag: `integer` (default = 1000). Maximal number of iterations for conjugate gradient algorithm when being run as Lanczos algorithm for tridiagonalization
- cg_delta_conv: `numeric` (default = 1E-2). Tolerance level for L2 norm of residuals for checking convergence in conjugate gradient algorithm when being used for parameter estimation
- num_rand_vec_trace: `integer` (default = 50). Number of random vectors (e.g., Rademacher) for stochastic approximation of the trace of a matrix
- reuse_rand_vec_trace: `boolean` (default = TRUE). If true, random vectors (e.g., Rademacher) for stochastic approximations of the trace of a matrix are sampled only once at the beginning of the parameter estimation and reused in later trace approximations. Otherwise they are sampled every time a trace is calculated
- seed_rand_vec_trace: `integer` (default = 1). Seed number to generate random vectors (e.g., Rademacher)
- piv_chol_rank: `integer` (default = 50). Rank of the pivoted Cholesky decomposition used as preconditioner in conjugate gradient algorithms
- cg_preconditioner_type: `string`. Type of preconditioner used for conjugate gradient algorithms.
  - Options for non-Gaussian likelihoods and gp_approx = "vecchia":
    - "Sigma_inv_plus_BtWB" (= default): (B^T * (D^-1 + W) * B) as preconditioner for inverting (B^T * D^-1 * B + W), where B^T * D^-1 * B approx= Sigma^-1
  - "piv_chol_on_Sigma": (Lk * Lk^T + W^-1) as preconditioner for inverting (B^-1 * D * B^-T + W^-1), where Lk is a low-rank pivoted Cholesky approximation for Sigma and B^-1 * D * B^-T approx= Sigma
  - Options for likelihood = "gaussian" and gp_approx = "full_scale_tapering":
    - "predictive_process_plus_diagonal" (= default): predictive process preconditiioner
    - "none": no preconditioner

`offset` A `numeric vector` with additional fixed effects contributions that are added to the linear predictor (= offset). The length of this vector needs to equal the number of training data points.

`fixed_effects` This is discontinued. Use the renamed equivalent argument `offset` instead

group_data_pred
: A vector or matrix with elements being group levels for which predictions are made (if there are grouped random effects in the GPModel)

group_rand_coef_data_pred
: A vector or matrix with covariate data for grouped random coefficients (if there are some in the GPModel)

gp_coords_pred
: A matrix with prediction coordinates (=features) for Gaussian process (if there is a GP in the GPModel)

gp_rand_coef_data_pred
: A vector or matrix with covariate data for Gaussian process random coefficients (if there are some in the GPModel)

cluster_ids_pred
: A vector with elements indicating the realizations of random effects / Gaussian processes for which predictions are made (set to NULL if you have not specified this when creating the GPModel)

X_pred
: A matrix with prediction covariate data for the fixed effects linear regression term (if there is one in the GPModel)

predict_cov_mat
: A boolean. If TRUE, the (posterior) predictive covariance is calculated in addition to the (posterior) predictive mean

predict_var
: A boolean. If TRUE, the (posterior) predictive variances are calculated

vecchia_approx
: Discontinued. Use the argument gp_approx instead

---

group_data *Example data for the GPBoost package*

Description

A matrix with categorical grouping variables for the example data of the GPBoost package

Usage

```
data(GPBoost_data)
```

---

group_data_test *Example data for the GPBoost package*

Description

A matrix with categorical grouping variables for predictions for the example data of the GPBoost package

Usage

```
data(GPBoost_data)
```

loadGPModel *Load a* GPModel *from a file*

Description

Load a GPModel from a file

Usage

```
loadGPModel(filename)
```

Arguments

filename	filename for loading

Value

A GPModel

Author(s)

Fabio Sigrist

Examples

```
data(GPBoost_data, package = "gpboost")
# Add intercept column
X1 <- cbind(rep(1,dim(X)[1]),X)
X_test1 <- cbind(rep(1,dim(X_test)[1]),X_test)

gp_model <- fitGPModel(group_data = group_data[,1], y = y, X = X1, likelihood="gaussian")
pred <- predict(gp_model, group_data_pred = group_data_test[,1],
                X_pred = X_test1, predict_var = TRUE)
# Save model to file
filename <- tempfile(fileext = ".json")
saveGPModel(gp_model,filename = filename)
# Load from file and make predictions again
gp_model_loaded <- loadGPModel(filename = filename)
pred_loaded <- predict(gp_model_loaded, group_data_pred = group_data_test[,1],
                       X_pred = X_test1, predict_var = TRUE)
# Check equality
pred$mu - pred_loaded$mu
pred$var - pred_loaded$var
```

neg_log_likelihood *Evaluate the negative log-likelihood*

Description

Evaluate the negative log-likelihood. If there is a linear fixed effects predictor term, this needs to be calculated "manually" prior to calling this function (see example below)

Usage

```
neg_log_likelihood(gp_model, cov_pars, y, fixed_effects = NULL,
  aux_pars = NULL)
```

Arguments

gp_model	A GPModel
cov_pars	A vector with numeric elements. Covariance parameters of Gaussian process and random effects
y	A vector with response variable data
fixed_effects	A numeric vector with fixed effects, e.g., containing a linear predictor. The length of this vector needs to equal the number of training data points.
aux_pars	A vector with numeric elements. Additional parameters for non-Gaussian likelihoods (e.g., shape parameter of a gamma or negative_binomial likelihood)

Author(s)

Fabio Sigrist

Examples

```
data(GPBoost_data, package = "gpboost")
gp_model <- GPModel(group_data = group_data, likelihood="gaussian")
X1 <- cbind(rep(1,dim(X)[1]), X)
coef <- c(0.1, 0.1, 0.1)
fixed_effects <- as.numeric(X1 %*% coef)
neg_log_likelihood(gp_model, y = y, cov_pars = c(0.1,1,1),
                   fixed_effects = fixed_effects)
```

neg_log_likelihood.GPModel *Evaluate the negative log-likelihood*

Description

Evaluate the negative log-likelihood. If there is a linear fixed effects predictor term, this needs to be calculated "manually" prior to calling this function (see example below)

Usage

```
## S3 method for class 'GPModel'
neg_log_likelihood(gp_model, cov_pars, y,
  fixed_effects = NULL, aux_pars = NULL)
```

Arguments

gp_model	A GPModel
cov_pars	A vector with numeric elements. Covariance parameters of Gaussian process and random effects
y	A vector with response variable data
fixed_effects	A numeric vector with fixed effects, e.g., containing a linear predictor. The length of this vector needs to equal the number of training data points.
aux_pars	A vector with numeric elements. Additional parameters for non-Gaussian likelihoods (e.g., shape parameter of a gamma or negative_binomial likelihood)

Value

A GPModel

Author(s)

Fabio Sigrist

Examples

```
data(GPBoost_data, package = "gpboost")
gp_model <- GPModel(group_data = group_data, likelihood="gaussian")
X1 <- cbind(rep(1,dim(X)[1]), X)
coef <- c(0.1, 0.1, 0.1)
fixed_effects <- as.numeric(X1 %*% coef)
neg_log_likelihood(gp_model, y = y, cov_pars = c(0.1,1,1),
                   fixed_effects = fixed_effects)
```

predict.gpb.Booster *Prediction function for* gpb.Booster *objects*

Description

Prediction function for gpb.Booster objects

Usage

```
## S3 method for class 'gpb.Booster'
predict(object, data, start_iteration = NULL,
  num_iteration = NULL, pred_latent = FALSE, predleaf = FALSE,
  predcontrib = FALSE, header = FALSE, reshape = FALSE,
  group_data_pred = NULL, group_rand_coef_data_pred = NULL,
  gp_coords_pred = NULL, gp_rand_coef_data_pred = NULL,
  cluster_ids_pred = NULL, predict_cov_mat = FALSE, predict_var = FALSE,
  cov_pars = NULL, ignore_gp_model = FALSE, rawscore = NULL,
  vecchia_pred_type = NULL, num_neighbors_pred = NULL, ...)
```

Arguments

object	Object of class gpb.Booster
data	a matrix object, a dgCMatrix object or a character representing a filename
start_iteration	int or NULL, optional (default=NULL) Start index of the iteration to predict. If NULL or <= 0, starts from the first iteration.
num_iteration	int or NULL, optional (default=NULL) Limit number of iterations in the prediction. If NULL, if the best iteration exists and start_iteration is NULL or <= 0, the best iteration is used; otherwise, all iterations from start_iteration are used. If <= 0, all iterations from start_iteration are used (no limits).
pred_latent	If TRUE latent variables, both fixed effects (tree-ensemble) and random effects (gp_model) are predicted. Otherwise, the response variable (label) is predicted. Depending on how the argument 'pred_latent' is set, different values are returned from this function; see the 'Value' section for more details. If there is no gp_model, this argument corresponds to 'raw_score' in LightGBM.
predleaf	whether predict leaf index instead.
predcontrib	return per-feature contributions for each record.
header	only used for prediction for text file. True if text file has header
reshape	whether to reshape the vector of predictions to a matrix form when there are several prediction outputs per case.
group_data_pred	A vector or matrix with elements being group levels for which predictions are made (if there are grouped random effects in the GPModel)

group_rand_coef_data_pred
: A vector or matrix with covariate data for grouped random coefficients (if there are some in the GPModel)

gp_coords_pred
: A matrix with prediction coordinates (=features) for Gaussian process (if there is a GP in the GPModel)

gp_rand_coef_data_pred
: A vector or matrix with covariate data for Gaussian process random coefficients (if there are some in the GPModel)

cluster_ids_pred
: A vector with elements indicating the realizations of random effects / Gaussian processes for which predictions are made (set to NULL if you have not specified this when creating the GPModel)

predict_cov_mat
: A boolean. If TRUE, the (posterior) predictive covariance is calculated in addition to the (posterior) predictive mean

predict_var
: A boolean. If TRUE, the (posterior) predictive variances are calculated

cov_pars
: A vector containing covariance parameters which are used if the gp_model has not been trained or if predictions should be made for other parameters than the trained ones

ignore_gp_model
: A boolean. If TRUE, predictions are only made for the tree ensemble part and the gp_model is ignored

rawscore
: This is discontinued. Use the renamed equivalent argument pred_latent instead

vecchia_pred_type
: A string specifying the type of Vecchia approximation used for making predictions. This is discontinued here. Use the function 'set_prediction_data' to specify this

num_neighbors_pred
: an integer specifying the number of neighbors for making predictions. This is discontinued here. Use the function 'set_prediction_data' to specify this

...
: Additional named arguments passed to the predict() method of the gpb.Booster object passed to object.

Value

either a list with vectors or a single vector / matrix depending on whether there is a gp_model or not If there is a gp_model, the result dict contains the following entries. 1. If pred_latent is TRUE, the dict contains the following 3 entries: - result["fixed_effect"] are the predictions from the tree-ensemble. - result["random_effect_mean"] are the predicted means of the gp_model. - result["random_effect_cov"] are the predicted covariances or variances of the gp_model (only if 'predict_var' or 'predict_cov' is TRUE). 2. If pred_latent is FALSE, the dict contains the following 2 entries: - result["response_mean"] are the predicted means of the response variable (Label) taking into account both the fixed effects (tree-ensemble) and the random effects (gp_model) - result["response_var"] are the predicted covariances or variances of the response variable (only if 'predict_var' or 'predict_cov' is TRUE) If there is no gp_model or predcontrib or ignore_gp_model are TRUE, the result contains predictions from the tree-booster only.

Author(s)

Fabio Sigrist, authors of the LightGBM R package

Examples

```
# See https://github.com/fabsig/GPBoost/tree/master/R-package for more examples


library(gpboost)
data(GPBoost_data, package = "gpboost")

#--------------------Combine tree-boosting and grouped random effects model----------------
# Create random effects model
gp_model <- GPModel(group_data = group_data[,1], likelihood = "gaussian")
# The default optimizer for covariance parameters (hyperparameters) is
# Nesterov-accelerated gradient descent.
# This can be changed to, e.g., Nelder-Mead as follows:
# re_params <- list(optimizer_cov = "nelder_mead")
# gp_model$set_optim_params(params=re_params)
# Use trace = TRUE to monitor convergence:
# re_params <- list(trace = TRUE)
# gp_model$set_optim_params(params=re_params)

# Train model
bst <- gpboost(data = X, label = y, gp_model = gp_model, nrounds = 16,
               learning_rate = 0.05, max_depth = 6, min_data_in_leaf = 5,
               verbose = 0)
# Estimated random effects model
summary(gp_model)

# Make predictions
# Predict latent variables
pred <- predict(bst, data = X_test, group_data_pred = group_data_test[,1],
                predict_var = TRUE, pred_latent = TRUE)
pred$random_effect_mean # Predicted latent random effects mean
pred$random_effect_cov # Predicted random effects variances
pred$fixed_effect # Predicted fixed effects from tree ensemble
# Predict response variable
pred_resp <- predict(bst, data = X_test, group_data_pred = group_data_test[,1],
                     predict_var = TRUE, pred_latent = FALSE)
pred_resp$response_mean # Predicted response mean
# For Gaussian data: pred$random_effect_mean + pred$fixed_effect = pred_resp$response_mean
pred$random_effect_mean + pred$fixed_effect - pred_resp$response_mean

#--------------------Combine tree-boosting and Gaussian process model----------------
# Create Gaussian process model
gp_model <- GPModel(gp_coords = coords, cov_function = "exponential",
                    likelihood = "gaussian")
# Train model
bst <- gpboost(data = X, label = y, gp_model = gp_model, nrounds = 8,
               learning_rate = 0.1, max_depth = 6, min_data_in_leaf = 5,
               verbose = 0)
```

```
# Estimated random effects model
summary(gp_model)
# Make predictions
pred <- predict(bst, data = X_test, gp_coords_pred = coords_test,
                predict_var = TRUE, pred_latent = TRUE)
pred$random_effect_mean # Predicted latent random effects mean
pred$random_effect_cov # Predicted random effects variances
pred$fixed_effect # Predicted fixed effects from tree ensemble
# Predict response variable
pred_resp <- predict(bst, data = X_test, gp_coords_pred = coords_test,
                     predict_var = TRUE, pred_latent = FALSE)
pred_resp$response_mean # Predicted response mean
```

`predict.GPModel` *Make predictions for a* `GPModel`

Description

Make predictions for a `GPModel`

Usage

```
## S3 method for class 'GPModel'
predict(object, y = NULL, group_data_pred = NULL,
  group_rand_coef_data_pred = NULL, gp_coords_pred = NULL,
  gp_rand_coef_data_pred = NULL, cluster_ids_pred = NULL,
  predict_cov_mat = FALSE, predict_var = FALSE, cov_pars = NULL,
  X_pred = NULL, use_saved_data = FALSE, predict_response = TRUE,
  offset = NULL, offset_pred = NULL, fixed_effects = NULL,
  fixed_effects_pred = NULL, vecchia_pred_type = NULL,
  num_neighbors_pred = NULL, ...)
```

Arguments

`object`	a `GPModel`
`y`	Observed data (can be NULL, e.g. when the model has been estimated already and the same data is used for making predictions)
`group_data_pred`	A `vector` or `matrix` with elements being group levels for which predictions are made (if there are grouped random effects in the `GPModel`)
`group_rand_coef_data_pred`	A `vector` or `matrix` with covariate data for grouped random coefficients (if there are some in the `GPModel`)
`gp_coords_pred`	A `matrix` with prediction coordinates (=features) for Gaussian process (if there is a GP in the `GPModel`)

gp_rand_coef_data_pred
: A vector or matrix with covariate data for Gaussian process random coefficients (if there are some in the GPModel)

cluster_ids_pred
: A vector with elements indicating the realizations of random effects / Gaussian processes for which predictions are made (set to NULL if you have not specified this when creating the GPModel)

predict_cov_mat
: A boolean. If TRUE, the (posterior) predictive covariance is calculated in addition to the (posterior) predictive mean

predict_var
: A boolean. If TRUE, the (posterior) predictive variances are calculated

cov_pars
: A vector containing covariance parameters which are used if the GPModel has not been trained or if predictions should be made for other parameters than the trained ones

X_pred
: A matrix with prediction covariate data for the fixed effects linear regression term (if there is one in the GPModel)

use_saved_data
: A boolean. If TRUE, predictions are done using a priory set data via the function '$set_prediction_data' (this option is not used by users directly)

predict_response
: A boolean. If TRUE, the response variable (label) is predicted, otherwise the latent random effects

offset
: A numeric vector with additional fixed effects contributions that are added to the linear predictor (= offset). The length of this vector needs to equal the number of training data points.

offset_pred
: A numeric vector with additional fixed effects contributions that are added to the linear predictor for the prediction points (= offset). The length of this vector needs to equal the number of prediction points.

fixed_effects
: This is discontinued. Use the renamed equivalent argument offset instead

fixed_effects_pred
: This is discontinued. Use the renamed equivalent argument offset_pred instead

vecchia_pred_type
: A string specifying the type of Vecchia approximation used for making predictions. This is discontinued here. Use the function 'set_prediction_data' to specify this

num_neighbors_pred
: an integer specifying the number of neighbors for making predictions. This is discontinued here. Use the function 'set_prediction_data' to specify this

...
: (not used, ignore this, simply here that there is no CRAN warning)

Value

Predictions from a GPModel. A list with three entries is returned:

- "mu" (first entry): predictive (=posterior) mean. For (generalized) linear mixed effects models, i.e., models with a linear regression term, this consists of the sum of fixed effects and random effects predictions

- "cov" (second entry): predictive (=posterior) covariance matrix. This is NULL if 'predict_cov_mat=FALSE'
- "var" (third entry) : predictive (=posterior) variances. This is NULL if 'predict_var=FALSE'

Author(s)

Fabio Sigrist

Examples

```
# See https://github.com/fabsig/GPBoost/tree/master/R-package for more examples


data(GPBoost_data, package = "gpboost")
# Add intercept column
X1 <- cbind(rep(1,dim(X)[1]),X)
X_test1 <- cbind(rep(1,dim(X_test)[1]),X_test)

#--------------------Grouped random effects model: single-level random effect----------------
gp_model <- fitGPModel(group_data = group_data[,1], y = y, X = X1,
                       likelihood="gaussian", params = list(std_dev = TRUE))
summary(gp_model)
# Make predictions
pred <- predict(gp_model, group_data_pred = group_data_test[,1],
                X_pred = X_test1, predict_var = TRUE)
pred$mu # Predicted mean
pred$var # Predicted variances
# Also predict covariance matrix
pred <- predict(gp_model, group_data_pred = group_data_test[,1],
                X_pred = X_test1, predict_cov_mat = TRUE)
pred$mu # Predicted mean
pred$cov # Predicted covariance


#--------------------Gaussian process model----------------
gp_model <- fitGPModel(gp_coords = coords, cov_function = "exponential",
                     likelihood="gaussian", y = y, X = X1, params = list(std_dev = TRUE))
summary(gp_model)
# Make predictions
pred <- predict(gp_model, gp_coords_pred = coords_test,
                X_pred = X_test1, predict_cov_mat = TRUE)
pred$mu # Predicted (posterior) mean of GP
pred$cov # Predicted (posterior) covariance matrix of GP
```

`predict_training_data_random_effects`
Predict ("estimate") training data random effects for a `GPModel`

Description

Predict ("estimate") training data random effects for a GPModel

Usage

```
predict_training_data_random_effects(gp_model, predict_var = FALSE)
```

Arguments

gp_model	A GPModel
predict_var	A boolean. If TRUE, the (posterior) predictive variances are calculated

Value

A GPModel

Author(s)

Fabio Sigrist

Examples

```
data(GPBoost_data, package = "gpboost")
# Add intercept column
X1 <- cbind(rep(1,dim(X)[1]),X)
X_test1 <- cbind(rep(1,dim(X_test)[1]),X_test)

gp_model <- fitGPModel(group_data = group_data[,1], y = y, X = X1, likelihood="gaussian")
all_training_data_random_effects <- predict_training_data_random_effects(gp_model)
first_occurences <- match(unique(group_data[,1]), group_data[,1])
unique_training_data_random_effects <- all_training_data_random_effects[first_occurences]
head(unique_training_data_random_effects)
```

---

predict_training_data_random_effects.GPModel

Predict ("estimate") training data random effects for a GPModel

---

Description

Predict ("estimate") training data random effects for a GPModel

Usage

```
## S3 method for class 'GPModel'
predict_training_data_random_effects(gp_model,
  predict_var = FALSE)
```

Arguments

gp_model	A GPModel
predict_var	A boolean. If TRUE, the (posterior) predictive variances are calculated

Value

A GPModel

Author(s)

Fabio Sigrist

Examples

```
data(GPBoost_data, package = "gpboost")
# Add intercept column
X1 <- cbind(rep(1,dim(X)[1]),X)
X_test1 <- cbind(rep(1,dim(X_test)[1]),X_test)

gp_model <- fitGPModel(group_data = group_data[,1], y = y, X = X1, likelihood="gaussian")
all_training_data_random_effects <- predict_training_data_random_effects(gp_model)
first_occurences <- match(unique(group_data[,1]), group_data[,1])
unique_training_data_random_effects <- all_training_data_random_effects[first_occurences]
head(unique_training_data_random_effects)
```

---

readRDS.gpb.Booster *readRDS for* gpb.Booster *models*

Description

Attempts to load a model stored in a .rds file, using readRDS

Usage

```
readRDS.gpb.Booster(file, refhook = NULL)
```

Arguments

file	a connection or the name of the file where the R object is saved to or read from.
refhook	a hook function for handling reference objects.

Value

gpb.Booster

Examples

```
library(gpboost)
data(agaricus.train, package = "gpboost")
train <- agaricus.train
dtrain <- gpb.Dataset(train$data, label = train$label)
data(agaricus.test, package = "gpboost")
test <- agaricus.test
dtest <- gpb.Dataset.create.valid(dtrain, test$data, label = test$label)
params <- list(objective = "regression", metric = "l2")
valids <- list(test = dtest)
model <- gpb.train(
  params = params
  , data = dtrain
  , nrounds = 10L
  , valids = valids
  , min_data = 1L
  , learning_rate = 1.0
  , early_stopping_rounds = 5L
)
model_file <- tempfile(fileext = ".rds")
saveRDS.gpb.Booster(model, model_file)
new_model <- readRDS.gpb.Booster(model_file)
```

---

saveGPModel *Save a* GPModel

Description

Save a GPModel

Usage

```
saveGPModel(gp_model, filename)
```

Arguments

gp_model	a GPModel
filename	filename for saving

Value

A GPModel

Author(s)

Fabio Sigrist

Examples

```
data(GPBoost_data, package = "gpboost")
# Add intercept column
X1 <- cbind(rep(1,dim(X)[1]),X)
X_test1 <- cbind(rep(1,dim(X_test)[1]),X_test)

gp_model <- fitGPModel(group_data = group_data[,1], y = y, X = X1, likelihood="gaussian")
pred <- predict(gp_model, group_data_pred = group_data_test[,1],
                X_pred = X_test1, predict_var = TRUE)
# Save model to file
filename <- tempfile(fileext = ".json")
saveGPModel(gp_model,filename = filename)
# Load from file and make predictions again
gp_model_loaded <- loadGPModel(filename = filename)
pred_loaded <- predict(gp_model_loaded, group_data_pred = group_data_test[,1],
                       X_pred = X_test1, predict_var = TRUE)
# Check equality
pred$mu - pred_loaded$mu
pred$var - pred_loaded$var
```

---

saveRDS.gpb.Booster *saveRDS for* gpb.Booster *models*

Description

Attempts to save a model using RDS. Has an additional parameter (raw) which decides whether to save the raw model or not.

Usage

```
saveRDS.gpb.Booster(object, file, ascii = FALSE, version = NULL,
  compress = TRUE, refhook = NULL, raw = TRUE)
```

Arguments

object	R object to serialize.
file	a connection or the name of the file where the R object is saved to or read from.
ascii	a logical. If TRUE or NA, an ASCII representation is written; otherwise (default), a binary one is used. See the comments in the help for save.
version	the workspace format version to use. NULL specifies the current default version (2). Versions prior to 2 are not supported, so this will only be relevant when there are later versions.
compress	a logical specifying whether saving to a named file is to use "gzip" compression, or one of "gzip", "bzip2" or "xz" to indicate the type of compression to be used. Ignored if file is a connection.
refhook	a hook function for handling reference objects.

raw — whether to save the model in a raw variable or not, recommended to leave it to TRUE.

Value

NULL invisibly.

Examples

```
library(gpboost)
data(agaricus.train, package = "gpboost")
train <- agaricus.train
dtrain <- gpb.Dataset(train$data, label = train$label)
data(agaricus.test, package = "gpboost")
test <- agaricus.test
dtest <- gpb.Dataset.create.valid(dtrain, test$data, label = test$label)
params <- list(objective = "regression", metric = "l2")
valids <- list(test = dtest)
model <- gpb.train(
    params = params
    , data = dtrain
    , nrounds = 10L
    , valids = valids
    , min_data = 1L
    , learning_rate = 1.0
    , early_stopping_rounds = 5L
)
model_file <- tempfile(fileext = ".rds")
saveRDS.gpb.Booster(model, model_file)
```

---

setinfo *Set information of an* gpb.Dataset *object*

Description

Set one attribute of a gpb.Dataset

Usage

```
setinfo(dataset, ...)

## S3 method for class 'gpb.Dataset'
setinfo(dataset, name, info, ...)
```

Arguments

dataset	Object of class gpb.Dataset
...	other parameters
name	the name of the field to get
info	the specific field of information to set

Details

The name field can be one of the following:

- label: vector of labels to use as the target variable
- weight: to do a weight rescale
- init_score: initial score is the base prediction gpboost will boost from
- group: used for learning-to-rank tasks. An integer vector describing how to group rows together as ordered results from the same set of candidate results to be ranked. For example, if you have a 100-document dataset with group = c(10, 20, 40, 10, 10, 10), that means that you have 6 groups, where the first 10 records are in the first group, records 11-30 are in the second group, etc.

Value

the dataset you passed in

the dataset you passed in

Examples

```
data(agaricus.train, package = "gpboost")
train <- agaricus.train
dtrain <- gpb.Dataset(train$data, label = train$label)
gpb.Dataset.construct(dtrain)

labels <- gpboost::getinfo(dtrain, "label")
gpboost::setinfo(dtrain, "label", 1 - labels)

labels2 <- gpboost::getinfo(dtrain, "label")
stopifnot(all.equal(labels2, 1 - labels))
```

---

set_optim_params *Set parameters for estimation of the covariance parameters*

Description

Set parameters for optimization of the covariance parameters of a GPModel

Usage

```
set_optim_params(gp_model, params = list())
```

Arguments

gp_model	A GPModel
params	A list with parameters for the estimation / optimization

- optimizer_cov: string (default = "lbfgs"). Optimizer used for estimating covariance parameters. Options: "gradient_descent", "lbfgs", "fisher_scoring", "newton", "nelder_mead", "adam". If there are additional auxiliary parameters for non-Gaussian likelihoods, 'optimizer_cov' is also used for those
- optimizer_coef: string (default = "wls" for Gaussian likelihoods and "lbfgs" for other likelihoods). Optimizer used for estimating linear regression coefficients, if there are any (for the GPBoost algorithm there are usually none). Options: "gradient_descent", "lbfgs", "wls", "nelder_mead", "adam". Gradient descent steps are done simultaneously with gradient descent steps for the covariance parameters. "wls" refers to doing coordinate descent for the regression coefficients using weighted least squares. If 'optimizer_cov' is set to "nelder_mead", "lbfgs", or "adam", 'optimizer_coef' is automatically also set to the same value.
- maxit: integer (default = 1000). Maximal number of iterations for optimization algorithm
- delta_rel_conv: numeric (default = 1E-6 except for "nelder_mead" for which the default is 1E-8). Convergence tolerance. The algorithm stops if the relative change in either the (approximate) log-likelihood or the parameters is below this value. For "adam", the L2 norm of the gradient is used instead of the relative change in the log-likelihood. If < 0, internal default values are used
- convergence_criterion: string (default = "relative_change_in_log_likelihood"). The convergence criterion used for terminating the optimization algorithm. Options: "relative_change_in_log_likelihood" or "relative_change_in_parameters"
- init_coef: vector with numeric elements (default = NULL). Initial values for the regression coefficients (if there are any, can be NULL)
- init_cov_pars: vector with numeric elements (default = NULL). Initial values for covariance parameters of Gaussian process and random effects (can be NULL). The order it the same as the order of the parameters in the summary function: first is the error variance (only for "gaussian" likelihood), next follow the variances of the grouped random effects (if there are any, in the order provided in 'group_data'), and then follow the marginal variance and the range of the Gaussian process. If there are multiple Gaussian processes, then the variances and ranges follow alternatingly. If 'init_cov_pars = NULL', an internal choice is used that depends on the likelihood and the random effects type and covariance function. If you select the option 'trace = TRUE' in the 'params' argument, you will see the first initial covariance parameters in iteration 0.
- lr_coef: numeric (default = 0.1). Learning rate for fixed effect regression coefficients if gradient descent is used

- lr_cov: numeric (default = 0.1 for "gradient_descent" and 1. otherwise). Initial learning rate for covariance parameters if a gradient-based optimization method is used
  - If lr_cov < 0, internal default values are used (0.1 for "gradient_descent" and 1. otherwise)
  - If there are additional auxiliary parameters for non-Gaussian likelihoods, 'lr_cov' is also used for those
  - For "lbfgs", this is divided by the norm of the gradient in the first iteration
- use_nesterov_acc: boolean (default = TRUE). If TRUE Nesterov acceleration is used. This is used only for gradient descent
- acc_rate_coef: numeric (default = 0.5). Acceleration rate for regression coefficients (if there are any) for Nesterov acceleration
- acc_rate_cov: numeric (default = 0.5). Acceleration rate for covariance parameters for Nesterov acceleration
- momentum_offset: integer (Default = 2). Number of iterations for which no momentum is applied in the beginning.
- trace: boolean (default = FALSE). If TRUE, information on the progress of the parameter optimization is printed
- std_dev: boolean (default = TRUE). If TRUE, approximate standard deviations are calculated for the covariance and linear regression parameters (= square root of diagonal of the inverse Fisher information for Gaussian likelihoods and square root of diagonal of a numerically approximated inverse Hessian for non-Gaussian likelihoods)
- init_aux_pars: vector with numeric elements (default = NULL). Initial values for additional parameters for non-Gaussian likelihoods (e.g., shape parameter of a gamma or negative_binomial likelihood)
- estimate_aux_pars: boolean (default = TRUE). If TRUE, additional parameters for non-Gaussian likelihoods are also estimated (e.g., shape parameter of a gamma or negative_binomial likelihood)
- cg_max_num_it: integer (default = 1000). Maximal number of iterations for conjugate gradient algorithms
- cg_max_num_it_tridiag: integer (default = 1000). Maximal number of iterations for conjugate gradient algorithm when being run as Lanczos algorithm for tridiagonalization
- cg_delta_conv: numeric (default = 1E-2). Tolerance level for L2 norm of residuals for checking convergence in conjugate gradient algorithm when being used for parameter estimation
- num_rand_vec_trace: integer (default = 50). Number of random vectors (e.g., Rademacher) for stochastic approximation of the trace of a matrix
- reuse_rand_vec_trace: boolean (default = TRUE). If true, random vectors (e.g., Rademacher) for stochastic approximations of the trace of a matrix are sampled only once at the beginning of the parameter estimation and reused in later trace approximations. Otherwise they are sampled every time a trace is calculated
- seed_rand_vec_trace: integer (default = 1). Seed number to generate random vectors (e.g., Rademacher)

- piv_chol_rank: `integer` (default = 50). Rank of the pivoted Cholesky decomposition used as preconditioner in conjugate gradient algorithms
- cg_preconditioner_type: `string`. Type of preconditioner used for conjugate gradient algorithms.
  - Options for non-Gaussian likelihoods and gp_approx = "vecchia":
    - "Sigma_inv_plus_BtWB" (= default): (B^T * (D^-1 + W) * B) as preconditioner for inverting (B^T * D^-1 * B + W), where B^T * D^-1 * B approx= Sigma^-1
  - "piv_chol_on_Sigma": (Lk * Lk^T + W^-1) as preconditioner for inverting (B^-1 * D * B^-T + W^-1), where Lk is a low-rank pivoted Cholesky approximation for Sigma and B^-1 * D * B^-T approx= Sigma
  - Options for likelihood = "gaussian" and gp_approx = "full_scale_tapering":
    - "predictive_process_plus_diagonal" (= default): predictive process preconditiioner
    - "none": no preconditioner

Author(s)

Fabio Sigrist

Examples

```
data(GPBoost_data, package = "gpboost")
gp_model <- GPModel(group_data = group_data, likelihood="gaussian")
set_optim_params(gp_model, params=list(optimizer_cov="nelder_mead"))
```

`set_optim_params.GPModel`

Set parameters for estimation of the covariance parameters

Description

Set parameters for optimization of the covariance parameters of a `GPModel`

Usage

```
## S3 method for class 'GPModel'
set_optim_params(gp_model, params = list())
```

Arguments

gp_model	A GPModel
params	A list with parameters for the estimation / optimization

- optimizer_cov: string (default = "lbfgs"). Optimizer used for estimating covariance parameters. Options: "gradient_descent", "lbfgs", "fisher_scoring", "newton", "nelder_mead", "adam". If there are additional auxiliary parameters for non-Gaussian likelihoods, 'optimizer_cov' is also used for those
- optimizer_coef: string (default = "wls" for Gaussian likelihoods and "lbfgs" for other likelihoods). Optimizer used for estimating linear regression coefficients, if there are any (for the GPBoost algorithm there are usually none). Options: "gradient_descent", "lbfgs", "wls", "nelder_mead", "adam". Gradient descent steps are done simultaneously with gradient descent steps for the covariance parameters. "wls" refers to doing coordinate descent for the regression coefficients using weighted least squares. If 'optimizer_cov' is set to "nelder_mead", "lbfgs", or "adam", 'optimizer_coef' is automatically also set to the same value.
- maxit: integer (default = 1000). Maximal number of iterations for optimization algorithm
- delta_rel_conv: numeric (default = 1E-6 except for "nelder_mead" for which the default is 1E-8). Convergence tolerance. The algorithm stops if the relative change in either the (approximate) log-likelihood or the parameters is below this value. For "adam", the L2 norm of the gradient is used instead of the relative change in the log-likelihood. If < 0, internal default values are used
- convergence_criterion: string (default = "relative_change_in_log_likelihood"). The convergence criterion used for terminating the optimization algorithm. Options: "relative_change_in_log_likelihood" or "relative_change_in_parameters"
- init_coef: vector with numeric elements (default = NULL). Initial values for the regression coefficients (if there are any, can be NULL)
- init_cov_pars: vector with numeric elements (default = NULL). Initial values for covariance parameters of Gaussian process and random effects (can be NULL). The order it the same as the order of the parameters in the summary function: first is the error variance (only for "gaussian" likelihood), next follow the variances of the grouped random effects (if there are any, in the order provided in 'group_data'), and then follow the marginal variance and the range of the Gaussian process. If there are multiple Gaussian processes, then the variances and ranges follow alternatingly. If 'init_cov_pars = NULL', an internal choice is used that depends on the likelihood and the random effects type and covariance function. If you select the option 'trace = TRUE' in the 'params' argument, you will see the first initial covariance parameters in iteration 0.
- lr_coef: numeric (default = 0.1). Learning rate for fixed effect regression coefficients if gradient descent is used
- lr_cov: numeric (default = 0.1 for "gradient_descent" and 1. otherwise). Initial learning rate for covariance parameters if a gradient-based optimization method is used

  - If lr_cov < 0, internal default values are used (0.1 for "gradient_descent" and 1. otherwise)
  - If there are additional auxiliary parameters for non-Gaussian likelihoods, 'lr_cov' is also used for those
  - For "lbfgs", this is divided by the norm of the gradient in the first iteration
- use_nesterov_acc: `boolean` (default = TRUE). If TRUE Nesterov acceleration is used. This is used only for gradient descent
- acc_rate_coef: `numeric` (default = 0.5). Acceleration rate for regression coefficients (if there are any) for Nesterov acceleration
- acc_rate_cov: `numeric` (default = 0.5). Acceleration rate for covariance parameters for Nesterov acceleration
- momentum_offset: `integer` (Default = 2). Number of iterations for which no momentum is applied in the beginning.
- trace: `boolean` (default = FALSE). If TRUE, information on the progress of the parameter optimization is printed
- std_dev: `boolean` (default = TRUE). If TRUE, approximate standard deviations are calculated for the covariance and linear regression parameters (= square root of diagonal of the inverse Fisher information for Gaussian likelihoods and square root of diagonal of a numerically approximated inverse Hessian for non-Gaussian likelihoods)
- init_aux_pars: `vector` with `numeric` elements (default = NULL). Initial values for additional parameters for non-Gaussian likelihoods (e.g., shape parameter of a gamma or negative_binomial likelihood)
- estimate_aux_pars: `boolean` (default = TRUE). If TRUE, additional parameters for non-Gaussian likelihoods are also estimated (e.g., shape parameter of a gamma or negative_binomial likelihood)
- cg_max_num_it: `integer` (default = 1000). Maximal number of iterations for conjugate gradient algorithms
- cg_max_num_it_tridiag: `integer` (default = 1000). Maximal number of iterations for conjugate gradient algorithm when being run as Lanczos algorithm for tridiagonalization
- cg_delta_conv: `numeric` (default = 1E-2). Tolerance level for L2 norm of residuals for checking convergence in conjugate gradient algorithm when being used for parameter estimation
- num_rand_vec_trace: `integer` (default = 50). Number of random vectors (e.g., Rademacher) for stochastic approximation of the trace of a matrix
- reuse_rand_vec_trace: `boolean` (default = TRUE). If true, random vectors (e.g., Rademacher) for stochastic approximations of the trace of a matrix are sampled only once at the beginning of the parameter estimation and reused in later trace approximations. Otherwise they are sampled every time a trace is calculated
- seed_rand_vec_trace: `integer` (default = 1). Seed number to generate random vectors (e.g., Rademacher)
- piv_chol_rank: `integer` (default = 50). Rank of the pivoted Cholesky decomposition used as preconditioner in conjugate gradient algorithms

- cg_preconditioner_type: `string`. Type of preconditioner used for conjugate gradient algorithms.
  - Options for non-Gaussian likelihoods and gp_approx = "vecchia":
    - "Sigma_inv_plus_BtWB" (= default): (B^T * (D^-1 + W) * B) as preconditioner for inverting (B^T * D^-1 * B + W), where B^T * D^-1 * B approx= Sigma^-1
  - "piv_chol_on_Sigma": (Lk * Lk^T + W^-1) as preconditioner for inverting (B^-1 * D * B^-T + W^-1), where Lk is a low-rank pivoted Cholesky approximation for Sigma and B^-1 * D * B^-T approx= Sigma
  - Options for likelihood = "gaussian" and gp_approx = "full_scale_tapering":
    - "predictive_process_plus_diagonal" (= default): predictive process preconditiioner
    - "none": no preconditioner

Value

A `GPModel`

Author(s)

Fabio Sigrist

Examples

```
data(GPBoost_data, package = "gpboost")
gp_model <- GPModel(group_data = group_data, likelihood="gaussian")
set_optim_params(gp_model, params=list(optimizer_cov="nelder_mead"))
```

---

set_prediction_data *Set prediction data for a* `GPModel`

Description

Set the data required for making predictions with a `GPModel`

Usage

```
set_prediction_data(gp_model, vecchia_pred_type = NULL,
  num_neighbors_pred = NULL, cg_delta_conv_pred = NULL,
  nsim_var_pred = NULL, rank_pred_approx_matrix_lanczos = NULL,
  group_data_pred = NULL, group_rand_coef_data_pred = NULL,
  gp_coords_pred = NULL, gp_rand_coef_data_pred = NULL,
  cluster_ids_pred = NULL, X_pred = NULL)
```

Arguments

`gp_model` A `GPModel`

`vecchia_pred_type`
A `string` specifying the type of Vecchia approximation used for making predictions. Default value if vecchia_pred_type = NULL: "order_obs_first_cond_obs_only". Available options:

- "order_obs_first_cond_obs_only": Vecchia approximation for the observable process and observed training data is ordered first and the neighbors are only observed training data points
- "order_obs_first_cond_all": Vecchia approximation for the observable process and observed training data is ordered first and the neighbors are selected among all points (training + prediction)
- "latent_order_obs_first_cond_obs_only": Vecchia approximation for the latent process and observed data is ordered first and neighbors are only observed points
- "latent_order_obs_first_cond_all": Vecchia approximation for the latent process and observed data is ordered first and neighbors are selected among all points
- "order_pred_first": Vecchia approximation for the observable process and prediction data is ordered first for making predictions. This option is only available for Gaussian likelihoods

`num_neighbors_pred`
an `integer` specifying the number of neighbors for the Vecchia approximation for making predictions. Default value if NULL: num_neighbors_pred = 2 * num_neighbors

`cg_delta_conv_pred`
a `numeric` specifying the tolerance level for L2 norm of residuals for checking convergence in conjugate gradient algorithms when being used for prediction Default value if NULL: 1e-3

`nsim_var_pred` an `integer` specifying the number of samples when simulation is used for calculating predictive variances Default value if NULL: 1000

`rank_pred_approx_matrix_lanczos`
an `integer` specifying the rank of the matrix for approximating predictive covariances obtained using the Lanczos algorithm Default value if NULL: 1000

`group_data_pred`
A `vector` or `matrix` with elements being group levels for which predictions are made (if there are grouped random effects in the `GPModel`)

`group_rand_coef_data_pred`
A `vector` or `matrix` with covariate data for grouped random coefficients (if there are some in the `GPModel`)

`gp_coords_pred` A `matrix` with prediction coordinates (=features) for Gaussian process (if there is a GP in the `GPModel`)

`gp_rand_coef_data_pred`
A `vector` or `matrix` with covariate data for Gaussian process random coefficients (if there are some in the `GPModel`)

cluster_ids_pred
: A vector with elements indicating the realizations of random effects / Gaussian processes for which predictions are made (set to NULL if you have not specified this when creating the GPModel)

X_pred
: A matrix with prediction covariate data for the fixed effects linear regression term (if there is one in the GPModel)

Author(s)

Fabio Sigrist

Examples

```
data(GPBoost_data, package = "gpboost")
set.seed(1)
train_ind <- sample.int(length(y),size=250)
gp_model <- GPModel(group_data = group_data[train_ind,1], likelihood="gaussian")
set_prediction_data(gp_model, group_data_pred = group_data[-train_ind,1])
```

---

set_prediction_data.GPModel

Set prediction data for a GPModel

Description

Set the data required for making predictions with a GPModel

Usage

```
## S3 method for class 'GPModel'
set_prediction_data(gp_model, vecchia_pred_type = NULL,
  num_neighbors_pred = NULL, cg_delta_conv_pred = NULL,
  nsim_var_pred = NULL, rank_pred_approx_matrix_lanczos = NULL,
  group_data_pred = NULL, group_rand_coef_data_pred = NULL,
  gp_coords_pred = NULL, gp_rand_coef_data_pred = NULL,
  cluster_ids_pred = NULL, X_pred = NULL)
```

Arguments

gp_model
: A GPModel

vecchia_pred_type
: A string specifying the type of Vecchia approximation used for making predictions. Default value if vecchia_pred_type = NULL: "order_obs_first_cond_obs_only". Available options:

- "order_obs_first_cond_obs_only": Vecchia approximation for the observable process and observed training data is ordered first and the neighbors are only observed training data points
- "order_obs_first_cond_all": Vecchia approximation for the observable process and observed training data is ordered first and the neighbors are selected among all points (training + prediction)
- "latent_order_obs_first_cond_obs_only": Vecchia approximation for the latent process and observed data is ordered first and neighbors are only observed points
- "latent_order_obs_first_cond_all": Vecchia approximation for the latent process and observed data is ordered first and neighbors are selected among all points
- "order_pred_first": Vecchia approximation for the observable process and prediction data is ordered first for making predictions. This option is only available for Gaussian likelihoods

`num_neighbors_pred`
: an `integer` specifying the number of neighbors for the Vecchia approximation for making predictions. Default value if NULL: num_neighbors_pred = 2 * num_neighbors

`cg_delta_conv_pred`
: a `numeric` specifying the tolerance level for L2 norm of residuals for checking convergence in conjugate gradient algorithms when being used for prediction Default value if NULL: 1e-3

`nsim_var_pred`
: an `integer` specifying the number of samples when simulation is used for calculating predictive variances Default value if NULL: 1000

`rank_pred_approx_matrix_lanczos`
: an `integer` specifying the rank of the matrix for approximating predictive covariances obtained using the Lanczos algorithm Default value if NULL: 1000

`group_data_pred`
: A `vector` or `matrix` with elements being group levels for which predictions are made (if there are grouped random effects in the `GPModel`)

`group_rand_coef_data_pred`
: A `vector` or `matrix` with covariate data for grouped random coefficients (if there are some in the `GPModel`)

`gp_coords_pred`
: A `matrix` with prediction coordinates (=features) for Gaussian process (if there is a GP in the `GPModel`)

`gp_rand_coef_data_pred`
: A `vector` or `matrix` with covariate data for Gaussian process random coefficients (if there are some in the `GPModel`)

`cluster_ids_pred`
: A `vector` with elements indicating the realizations of random effects / Gaussian processes for which predictions are made (set to NULL if you have not specified this when creating the `GPModel`)

`X_pred`
: A `matrix` with prediction covariate data for the fixed effects linear regression term (if there is one in the `GPModel`)

Value

A GPModel

Author(s)

Fabio Sigrist

Examples

```
data(GPBoost_data, package = "gpboost")
set.seed(1)
train_ind <- sample.int(length(y),size=250)
gp_model <- GPModel(group_data = group_data[train_ind,1], likelihood="gaussian")
set_prediction_data(gp_model, group_data_pred = group_data[-train_ind,1])
```

---

slice *Slice a dataset*

Description

Get a new gpb.Dataset containing the specified rows of original gpb.Dataset object

Usage

```
slice(dataset, ...)

## S3 method for class 'gpb.Dataset'
slice(dataset, idxset, ...)
```

Arguments

dataset	Object of class gpb.Dataset
...	other parameters (currently not used)
idxset	an integer vector of indices of rows needed

Value

constructed sub dataset

Examples

```
data(agaricus.train, package = "gpboost")
train <- agaricus.train
dtrain <- gpb.Dataset(train$data, label = train$label)

dsub <- gpboost::slice(dtrain, seq_len(42L))
gpb.Dataset.construct(dsub)
labels <- gpboost::getinfo(dsub, "label")
```

summary.GPModel *Summary for a* GPModel

Description

Summary for a GPModel

Usage

```
## S3 method for class 'GPModel'
summary(object, ...)
```

Arguments

object	a GPModel
...	(not used, ignore this, simply here that there is no CRAN warning)

Value

Summary of a (fitted) GPModel

Author(s)

Fabio Sigrist

Examples

```
# See https://github.com/fabsig/GPBoost/tree/master/R-package for more examples

data(GPBoost_data, package = "gpboost")
# Add intercept column
X1 <- cbind(rep(1,dim(X)[1]),X)
X_test1 <- cbind(rep(1,dim(X_test)[1]),X_test)

#--------------------Grouped random effects model: single-level random effect----------------
gp_model <- fitGPModel(group_data = group_data[,1], y = y, X = X1,
                       likelihood="gaussian", params = list(std_dev = TRUE))
summary(gp_model)



#--------------------Gaussian process model----------------
gp_model <- fitGPModel(gp_coords = coords, cov_function = "exponential",
                       likelihood="gaussian", y = y, X = X1, params = list(std_dev = TRUE))
summary(gp_model)
```

X — *Example data for the GPBoost package*

Description

A matrix with covariate data for the example data of the GPBoost package

Usage

```
data(GPBoost_data)
```

X_test — *Example data for the GPBoost package*

Description

A matrix with covariate information for the predictions for the example data of the GPBoost package

Usage

```
data(GPBoost_data)
```

y — *Example data for the GPBoost package*

Description

Response variable for the example data of the GPBoost package

Usage

```
data(GPBoost_data)
```

Index