

# Package ‘gpboost’

August 25, 2022

**Type** Package

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

**Version** 0.7.9

**Date** 2022-08-25

**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 (2020) <[arXiv:2004.02653](https://arxiv.org/abs/2004.02653)> and Sigrist (2021) <[arXiv:2105.08966](https://arxiv.org/abs/2105.08966)> 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++11

**RoxygenNote** 6.0.1

**Author** Fabio Sigrist [aut, cre],  
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]

**Maintainer** Fabio Sigrist <fabiosigrist@gmail.com>

**Repository** CRAN

**Date/Publication** 2022-08-25 12:40:02 UTC

## R topics documented:

agaricus.test . . . . .	3
agaricus.train . . . . .	4
bank . . . . .	5
coords . . . . .	5
coords_test . . . . .	5
dim.gpb.Dataset . . . . .	6
dimnames.gpb.Dataset . . . . .	7
fit . . . . .	8
fit.GPModel . . . . .	9
fitGPModel . . . . .	11
getinfo . . . . .	16
get_nested_categories . . . . .	17
gpb.convert_with_rules . . . . .	18
gpb.cv . . . . .	19
gpb.Dataset . . . . .	23
gpb.Dataset.construct . . . . .	24
gpb.Dataset.create.valid . . . . .	25
gpb.Dataset.save . . . . .	26
gpb.Dataset.set.categorical . . . . .	26
gpb.Dataset.set.reference . . . . .	27
gpb.dump . . . . .	28
gpb.get.eval.result . . . . .	29
gpb.grid.search.tune.parameters . . . . .	30

gpb.importance . . . . .	33
gpb.interprete . . . . .	34
gpb.load . . . . .	35
gpb.model.dt.tree . . . . .	36
gpb.plot.importance . . . . .	37
gpb.plot.interpretation . . . . .	38
gpb.plot.part.dep.interact . . . . .	40
gpb.plot.partial.dependence . . . . .	41
gpb.save . . . . .	43
gpb.train . . . . .	44
gpboost . . . . .	49
GBoost_data . . . . .	54
GPMModel . . . . .	54
GPMModel_shared_params . . . . .	57
group_data . . . . .	60
group_data_test . . . . .	61
loadGPMModel . . . . .	61
predict.gpb.Booster . . . . .	62
predict.GPMModel . . . . .	65
predict_training_data_random_effects . . . . .	68
predict_training_data_random_effects.GPMModel . . . . .	69
readRDS.gpb.Booster . . . . .	70
saveGPMModel . . . . .	71
saveRDS.gpb.Booster . . . . .	72
setinfo . . . . .	73
set_prediction_data . . . . .	74
set_prediction_data.GPMModel . . . . .	75
slice . . . . .	77
summary.GPMModel . . . . .	78
X . . . . .	79
X_test . . . . .	79
y . . . . .	79

**Index** **80**

---

agaricus.test	<i>Test part from Mushroom Data Set</i>
---------------	---

---

**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 dgCMatrx class, with 126 columns.

**Usage**

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

**Format**

A list containing a label vector, and a dgCMatrx 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	<i>Training part from Mushroom Data Set</i>
----------------	---

---

**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 dgCMatrx class, with 126 columns.

**Usage**

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

**Format**

A list containing a label vector, and a dgCMatrx 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	<i>Bank Marketing Data Set</i>
------	--------------------------------

---

**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	<i>Example data for the GPBoost package</i>
--------	---

---

**Description**

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

**Usage**

```
data(GPBoost_data)
```

---

coords_test	<i>Example data for the GPBoost package</i>
-------------	---

---

**Description**

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

**Usage**

```
data(GPBoost_data)
```

---

dim.gpb.Dataset	<i>Dimensions of an gpb.Dataset</i>
-----------------	-------------------------------------

---

## 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

<code>x</code>	Object of class <code>gpb.Dataset</code>
<code>...</code>	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

<code>x</code>	object of class <code>gpb.Dataset</code>
<code>value</code>	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	<i>Generic 'fit' method for a GPModel</i>
-----	---

---

### Description

Generic 'fit' method for a GPModel

### Usage

```
fit(gp_model, y, X, params, 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   | <p>A list with parameters for the model fitting / optimization</p> <ul style="list-style-type: none"> <li>• optimizer_cov Optimizer used for estimating covariance parameters. Options: "gradient_descent", "fisher_scoring", "nelder_mead", "bfgs", "adam". Default= gradient_descent"</li> <li>• optimizer_coef Optimizer used for estimating linear regression coefficients, if there are any (for the GPBoost algorithm there are usually none). Options: "gradient_descent", "wls", "nelder_mead", "bfgs", "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. Default="wls" for Gaussian data and "gradient_descent" for other likelihoods. If 'optimizer_cov' is set to "nelder_mead", "bfgs", or "adam", 'optimizer_coef' is automatically also set to the same value.</li> <li>• maxit Maximal number of iterations for optimization algorithm. Default=1000</li> <li>• delta_rel_conv Convergence tolerance. The algorithm stops if the relative change in either the (approximate) log-likelihood or the parameters is below this value. For "bfgs" and "adam", the L2 norm of the gradient is used instead of the relative change in the log-likelihood. Default=1E-6</li> <li>• convergence_criterion The convergence criterion used for terminating the optimization algorithm. Options: "relative_change_in_log_likelihood" (default) or "relative_change_in_parameters"</li> <li>• init_coef Initial values for the regression coefficients (if there are any, can be NULL). Default=NULL</li> <li>• init_cov_pars Initial values for covariance parameters of Gaussian process and random effects (can be NULL). Default=NULL</li> <li>• lr_coef Learning rate for fixed effect regression coefficients if gradient descent is used. Default=0.1</li> </ul> |



- `lr_cov` Learning rate for covariance parameters. If  $\leq 0$ , internal default values are used. Default value = 0.1 for "gradient\_descent" and 1. for "fisher\_scoring"
- `use_nesterov_acc` If TRUE Nesterov acceleration is used. This is used only for gradient descent. Default=TRUE
- `acc_rate_coef` Acceleration rate for regression coefficients (if there are any) for Nesterov acceleration. Default=0.5
- `acc_rate_cov` Acceleration rate for covariance parameters for Nesterov acceleration. Default=0.5
- `momentum_offset` Number of iterations for which no momentum is applied in the beginning. Default=2
- `trace` If TRUE, information on the progress of the parameter optimization is printed. Default=FALSE
- `std_dev` 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)

`fixed_effects` A vector of optional external fixed effects which are held fixed during training.

### Author(s)

Fabio Sigrist

---

fit.GPModel

*Fits a GPModel*

---

### Description

Estimates the parameters of a GPModel using maximum likelihood estimation

### Usage

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

### Arguments

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

- `optimizer_cov` Optimizer used for estimating covariance parameters. Options: "gradient\_descent", "fisher\_scoring", "nelder\_mead", "bfgs", "adam". Default= gradient\_descent"
- `optimizer_coef` Optimizer used for estimating linear regression coefficients, if there are any (for the GPBoost algorithm there are usually none). Options: "gradient\_descent", "wls", "nelder\_mead", "bfgs", "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. Default="wls" for Gaussian data and "gradient\_descent" for other likelihoods. If 'optimizer\_cov' is set to "nelder\_mead", "bfgs", or "adam", 'optimizer\_coef' is automatically also set to the same value.
- `maxit` Maximal number of iterations for optimization algorithm. Default=1000
- `delta_rel_conv` Convergence tolerance. The algorithm stops if the relative change in either the (approximate) log-likelihood or the parameters is below this value. For "bfgs" and "adam", the L2 norm of the gradient is used instead of the relative change in the log-likelihood. Default=1E-6
- `convergence_criterion` The convergence criterion used for terminating the optimization algorithm. Options: "relative\_change\_in\_log\_likelihood" (default) or "relative\_change\_in\_parameters"
- `init_coef` Initial values for the regression coefficients (if there are any, can be NULL). Default=NULL
- `init_cov_pars` Initial values for covariance parameters of Gaussian process and random effects (can be NULL). Default=NULL
- `lr_coef` Learning rate for fixed effect regression coefficients if gradient descent is used. Default=0.1
- `lr_cov` Learning rate for covariance parameters. If  $\leq 0$ , internal default values are used. Default value = 0.1 for "gradient\_descent" and 1. for "fisher\_scoring"
- `use_nesterov_acc` If TRUE Nesterov acceleration is used. This is used only for gradient descent. Default=TRUE
- `acc_rate_coef` Acceleration rate for regression coefficients (if there are any) for Nesterov acceleration. Default=0.5
- `acc_rate_cov` Acceleration rate for covariance parameters for Nesterov acceleration. Default=0.5
- `momentum_offset` Number of iterations for which no momentum is applied in the beginning. Default=2
- `trace` If TRUE, information on the progress of the parameter optimization is printed. Default=FALSE
- `std_dev` 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)

`fixed_effects` A vector of optional external fixed effects which are held fixed during training.

**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
```

**Description**

Estimates the parameters of a GPMoel using maximum likelihood estimation

**Usage**

```
fitGPMoel(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, cov_fct_taper_range = 1, vecchia_approx = FALSE,
  num_neighbors = 30L, vecchia_ordering = "none",
  vecchia_pred_type = "order_obs_first_cond_obs_only",
  num_neighbors_pred = num_neighbors, cluster_ids = NULL,
  free_raw_data = FALSE, likelihood = "gaussian", y, X = NULL,
  params = list())
```

**Arguments**

- 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. The following covariance functions are available: "exponential", "gaussian", "matern",

"powered\_exponential", "wendland", and "exponential\_tapered". For "exponential", "gaussian", and "powered\_exponential", we follow the notation and parametrization of Diggle and Ribeiro (2007). For "matern", we follow the notation of Rasmussen and Williams (2006). For "wendland", we follow the notation of Bevilacqua et al. (2019). A covariance function with the suffix "\_tapered" refers to a covariance function that is multiplied by a compactly supported Wendland covariance function (= tapering)

cov_fct_shape	A numeric specifying the shape parameter of the covariance function (=smoothness parameter for Matern and Wendland covariance). For the Wendland covariance function, we follow the notation of Bevilacqua et al. (2019)). This parameter is irrelevant for some covariance functions such as the exponential or Gaussian
cov_fct_taper_range	A numeric specifying the range parameter of the Wendland covariance function / taper. We follow the notation of Bevilacqua et al. (2019)
vecchia_approx	A boolean. If TRUE, the Vecchia approximation is used
num_neighbors	An integer specifying the number of neighbors for the Vecchia approximation
vecchia_ordering	A string specifying the ordering used in the Vecchia approximation. "none" means the default ordering is used, "random" uses a random ordering
vecchia_pred_type	A string specifying the type of Vecchia approximation used for making predictions. "order_obs_first_cond_obs_only" = observed data is ordered first and the neighbors are only observed points, "order_obs_first_cond_all" = observed data is ordered first and the neighbors are selected among all points (observed + predicted), "order_pred_first" = predicted data is ordered first for making predictions, "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
num_neighbors_pred	an integer specifying the number of neighbors for the Vecchia approximation for making predictions
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
likelihood	A string specifying the likelihood function (distribution) of the response variable Default = "gaussian"
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 model fitting / optimization

- `optimizer_cov` Optimizer used for estimating covariance parameters. Options: "gradient\_descent", "fisher\_scoring", "nelder\_mead", "bfgs", "adam". Default= gradient\_descent"
- `optimizer_coef` Optimizer used for estimating linear regression coefficients, if there are any (for the GPBoost algorithm there are usually none). Options: "gradient\_descent", "wls", "nelder\_mead", "bfgs", "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. Default="wls" for Gaussian data and "gradient\_descent" for other likelihoods. If 'optimizer\_cov' is set to "nelder\_mead", "bfgs", or "adam", 'optimizer\_coef' is automatically also set to the same value.
- `maxit` Maximal number of iterations for optimization algorithm. Default=1000
- `delta_rel_conv` Convergence tolerance. The algorithm stops if the relative change in either the (approximate) log-likelihood or the parameters is below this value. For "bfgs" and "adam", the L2 norm of the gradient is used instead of the relative change in the log-likelihood. Default=1E-6
- `convergence_criterion` The convergence criterion used for terminating the optimization algorithm. Options: "relative\_change\_in\_log\_likelihood" (default) or "relative\_change\_in\_parameters"
- `init_coef` Initial values for the regression coefficients (if there are any, can be NULL). Default=NULL
- `init_cov_pars` Initial values for covariance parameters of Gaussian process and random effects (can be NULL). Default=NULL
- `lr_coef` Learning rate for fixed effect regression coefficients if gradient descent is used. Default=0.1
- `lr_cov` Learning rate for covariance parameters. If  $\leq 0$ , internal default values are used. Default value = 0.1 for "gradient\_descent" and 1. for "fisher\_scoring"
- `use_nesterov_acc` If TRUE Nesterov acceleration is used. This is used only for gradient descent. Default=TRUE
- `acc_rate_coef` Acceleration rate for regression coefficients (if there are any) for Nesterov acceleration. Default=0.5
- `acc_rate_cov` Acceleration rate for covariance parameters for Nesterov acceleration. Default=0.5
- `momentum_offset` Number of iterations for which no momentum is applied in the beginning. Default=2
- `trace` If TRUE, information on the progress of the parameter optimization is printed. Default=FALSE
- `std_dev` 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)

**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",
                      vecchia_approx = TRUE, num_neighbors = 30,
                      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 gpbboost 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 gboost will boost from.

**Value**

info data

info data

**Examples**

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

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

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

---

`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!

```

**Description**

Cross validation function for determining number of boosting iterations

## Usage

```
gpb.cv(params = list(), data, nrounds = 100L, gp_model = NULL,
  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)
  - `min_data_in_leaf` Minimal number of samples per leaf. Tuning parameter for tree-boosting (default = 20)
  - `max_depth` Maximal depth of a tree. Tuning parameter for tree-boosting (default = no limit)
  - `leaves_newton_update` 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)
  - `train_gp_model_cov_pars` 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 the values via the `'init_cov_pars'` parameter when creating the `gp_model` (default = TRUE).
  - `use_gp_model_for_validation` If TRUE, the Gaussian process is also used (in addition to the tree model) for calculating predictions on the validation data (default = TRUE)
  - `use_nesterov_acc` Set this to TRUE to do boosting with Nesterov acceleration (default = FALSE). Can currently only be used for `tree_learner = "serial"` (default option)
  - `nesterov_acc_rate` Acceleration rate for momentum step in case Nesterov accelerated boosting is used (default = 0.5)
  - `boosting` Boosting type. "gbdt", "rf", "dart" or "goss". Only "gbdt" allows for doing Gaussian process boosting.
  - `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 <code>gpb.Dataset</code> object, used for training. Some functions, such as <code>gpb.cv</code> , 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. Default = 100
gp_model	A <code>GPMoDel</code> object that contains the random effects (Gaussian process and / or grouped random effects) model
use_gp_model_for_validation	Boolean (default = TRUE). If TRUE, the <code>gp_model</code> (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 <code>gp_model</code> (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_00S	Boolean (default = FALSE). If TRUE, the covariance parameters of the <code>gp_model</code> 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 <code>GPBoostOOS</code> algorithm.
train_gp_model_cov_pars	Boolean (default = TRUE). If TRUE, the covariance parameters of the <code>gp_model</code> (Gaussian process and/or random effects) are estimated in every boosting iterations, otherwise the <code>gp_model</code> parameters are not estimated. In the latter case, you need to either estimate them beforehand or provide the values via the <code>init_cov_pars</code> parameter when creating the <code>gp_model</code>
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 <code>nfold</code> and <code>stratified</code> parameters are ignored.
nfold	the original dataset is randomly partitioned into <code>nfold</code> equal size subsamples.
label	Vector of labels, used if data is not an <code>gpb.Dataset</code>
weight	vector of response values. If not NULL, will set to dataset
obj	objective function, can be character or custom objective function. Examples include <code>regression</code> , <code>regression_l1</code> , <code>huber</code> , <code>binary</code> , <code>lambdarank</code> , <code>multiclass</code> , <code>multiclass</code>
eval	evaluation function(s). This can be a character vector, function, or list with a mixture of strings and functions. <ul style="list-style-type: none"> <li>• <b>a. character vector:</b> If you provide a character vector to this argument, it should contain strings with valid evaluation metrics. See the "<a href="#">metric</a>" section of the parameter documentation for a list of valid metrics.</li> <li>• <b>b. function:</b> You can provide a custom evaluation function. This should accept the keyword arguments <code>preds</code> and <code>dtrain</code> and should return a named list with three elements: <ul style="list-style-type: none"> <li>– name: A string with the name of the metric, used for printing and storing results.</li> <li>– value: A single number indicating the value of the metric for the given predictions and true values</li> </ul> </li> </ul>

- `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.

<code>verbose</code>	verbosity for output, if $\leq 0$ , also will disable the print of evaluation during training
<code>record</code>	Boolean, <code>TRUE</code> will record iteration message to <code>booster\$record_evals</code>
<code>eval_freq</code>	evaluation output frequency, only effect when <code>verbose &gt; 0</code>
<code>showsd</code>	boolean, whether to show standard deviation of cross validation. This parameter defaults to <code>TRUE</code> .
<code>stratified</code>	a boolean indicating whether sampling of folds should be stratified by the values of outcome labels.
<code>init_model</code>	path of model file of <code>gpb</code> . <code>Booster</code> object, will continue training from this model
<code>colnames</code>	feature names, if not null, will use this to overwrite the names in dataset
<code>categorical_feature</code>	categorical features. This can either be a character vector of feature names or an integer vector with the indices of the features (e.g. <code>c(1L, 10L)</code> to say "the first and tenth columns").
<code>early_stopping_rounds</code>	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 <code>early_stopping_rounds</code> consecutive boosting rounds. If training stops early, the returned model will have attribute <code>best_iter</code> set to the iteration number of the best iteration.
<code>callbacks</code>	List of callback functions that are applied at each iteration.
<code>reset_data</code>	Boolean, setting it to <code>TRUE</code> (not the default value) will transform the booster model into a predictor model which frees up memory and the original datasets
<code>delete_boosters_folds</code>	Boolean, setting it to <code>TRUE</code> (not the default value) will delete the boosters of the individual folds
<code>...</code>	other parameters, see <code>Parameters.rst</code> 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 Sigrüst

### 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,
              objective = "regression_l2")

# 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 " <a href="#">Dataset Parameters</a> " section of the <a href="#">parameter documentation</a> 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 dgCMatix 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 <code>gpb.Dataset</code>
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

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

```

## Arguments

<code>param_grid</code>	list with candidate parameters defining the grid over which a search is done
<code>data</code>	a <code>gpb.Dataset</code> object, used for training. Some functions, such as <code>gpb.cv</code> , may allow you to pass other types of data like matrix and then separately supply label as a keyword argument.
<code>params</code>	list with other parameters not included in <code>param_grid</code>
<code>num_try_random</code>	integer with number of random trial on parameter grid. If <code>NULL</code> , a deterministic search is done
<code>nrounds</code>	number of boosting iterations (= number of trees). This is the most important tuning parameter for boosting. Default = 100
<code>gp_model</code>	A <code>GPMoDel</code> object that contains the random effects (Gaussian process and / or grouped random effects) model

use_gp_model_for_validation	Boolean (default = TRUE). 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 (default = TRUE). 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 <a href="#">gpb.Dataset</a>
weight	vector of response values. If not NULL, will set to dataset
obj	objective function, can be character or custom objective function. Examples include regression, regression_l1, huber, binary, lambdarank, multiclass, multiclass
eval	evaluation function(s). This can be a character vector, function, or list with a mixture of strings and functions. <ul style="list-style-type: none"> <li>• <b>a. character vector:</b> If you provide a character vector to this argument, it should contain strings with valid evaluation metrics. See <a href="#">the "metric" section of the parameter documentation</a> for a list of valid metrics.</li> <li>• <b>b. function:</b> 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: <ul style="list-style-type: none"> <li>– name: A string with the name of the metric, used for printing and storing results.</li> <li>– value: A single number indicating the value of the metric for the given predictions and true values</li> <li>– higher_better: A boolean indicating whether higher values indicate a better fit. For example, this would be FALSE for metrics like MAE or RMSE.</li> </ul> </li> <li>• <b>c. list:</b> If a list is given, it should only contain character vectors and functions. These should follow the requirements from the descriptions above.</li> </ul>
verbose_eval	integer. Whether to display information on the progress of tuning parameter choice. If None or 0, verbose is off. 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. <code>c(1L, 10L)</code> 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 <code>early_stopping_rounds</code> consecutive boosting rounds. If training stops early, the returned model will have attribute <code>best_iter</code> set to the iteration number of the best iteration.
callbacks	List of callback functions that are applied at each iteration.
...	other parameters, see <code>Parameters.rst</code> 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)`

### 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")
dtrain <- gpb.Dataset(X, label = y)
params <- list(objective = "regression_l2")
param_grid = list("learning_rate" = c(0.1,0.01), "min_data_in_leaf" = c(20),
                 "max_depth" = c(5,10), "num_leaves" = 2^17, "max_bin" = c(255,1000))
# Parameter tuning using cross-validation and deterministic grid search
```



```

set.seed(1)
opt_params <- gpb.grid.search.tune.parameters(param_grid = param_grid,
                                             params = params,
                                             num_try_random = NULL,
                                             nfold = 4,
                                             data = dtrain,
                                             gp_model = gp_model,
                                             verbose_eval = 1,
                                             nrounds = 1000,
                                             early_stopping_rounds = 5,
                                             eval = "l2")

# Parameter tuning using cross-validation and random grid search
set.seed(1)
opt_params <- gpb.grid.search.tune.parameters(param_grid = param_grid,
                                             params = params,
                                             num_try_random = 4,
                                             nfold = 4,
                                             data = dtrain,
                                             gp_model = gp_model,
                                             verbose_eval = 1,
                                             nrounds = 1000,
                                             early_stopping_rounds = 5,
                                             eval = "l2")

```

---

gpb.importance	<i>Compute feature importance in a model</i>
----------------	--

---

### 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 splitted in trees.

**Examples**

```

data(agaricus.train, package = "gboost")
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 <code>gpb.Booster</code> .
data	a matrix object or a <code>dgCMatrx</code> object.
idxset	an integer vector of indices of rows needed.
num_iteration	number of iteration want to predict with, <code>NULL</code> or <code>&lt;= 0</code> 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,
               objective = "regression_l2", 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 leaves.

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 = "gboost")
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 <code>gpb.importance</code> .
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 <code>cex.names</code> parameter to <code>barplot</code> . 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.

**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 <a href="#">gpb.interprete</a> .
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  $c(\text{length}(x), \text{length}(y))$  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,
                          objective = "regression_l2",
                          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,
        objective = "regression_l2",
        verbose = 0)
gpb.plot.partial.dependence(gpboost_model, X, variable = 1)

```

---

gpb.save	<i>Save GPBoost model</i>
----------	---------------------------

---

## Description

Save GPBoost model

## Usage

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

## Arguments

booster	Object of class <code>gpb.Booster</code>
filename	saved filename
start_iteration	int or NULL, optional (default=NULL) Start index of the iteration to predict. If NULL or $\leq 0$ , 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 $\leq 0$ , the best iteration is used; otherwise, all iterations from start_iteration are used. If $\leq 0$ , 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 <code>predict()</code> method of the <code>gpb.Booster</code> object passed to <code>object</code> . This is only used when there is a <code>gp_model</code> and when <code>save_raw_data=FALSE</code>

## 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,
              objective = "regression_l2", 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 <a href="#">the parameter documentation</a> for more information. A few key parameters: <ul style="list-style-type: none"> <li>• <code>learning_rate</code> 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</li> </ul> |
|--------|--|

- `num_leaves` Number of leaves in a tree. Tuning parameter for tree-boosting (default = 31)
  - `min_data_in_leaf` Minimal number of samples per leaf. Tuning parameter for tree-boosting (default = 20)
  - `max_depth` Maximal depth of a tree. Tuning parameter for tree-boosting (default = no limit)
  - `leaves_newton_update` 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)
  - `train_gp_model_cov_pars` 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 the values via the `'init_cov_pars'` parameter when creating the `gp_model` (default = TRUE).
  - `use_gp_model_for_validation` If TRUE, the Gaussian process is also used (in addition to the tree model) for calculating predictions on the validation data (default = TRUE)
  - `use_nesterov_acc` Set this to TRUE to do boosting with Nesterov acceleration (default = FALSE). Can currently only be used for `tree_learner = "serial"` (default option)
  - `nesterov_acc_rate` Acceleration rate for momentum step in case Nesterov accelerated boosting is used (default = 0.5)
  - `boosting` Boosting type. "gbdt", "rf", "dart" or "goss". Only "gbdt" allows for doing Gaussian process boosting.
  - `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. Default = 100
- `gp_model` A `GPMoDel` object that contains the random effects (Gaussian process and / or grouped random effects) model
- `use_gp_model_for_validation` Boolean (default = TRUE). 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 (default = TRUE). 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	objective function, can be character or custom objective function. Examples include regression, regression_l1, huber, binary, lambdarank, multiclass, multiclass
eval	<p>evaluation function(s). This can be a character vector, function, or list with a mixture of strings and functions.</p> <ul style="list-style-type: none"> <li>• <b>a. character vector:</b> If you provide a character vector to this argument, it should contain strings with valid evaluation metrics. See <a href="#">the "metric" section of the parameter documentation</a> for a list of valid metrics.</li> <li>• <b>b. function:</b> 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: <ul style="list-style-type: none"> <li>– name: A string with the name of the metric, used for printing and storing results.</li> <li>– value: A single number indicating the value of the metric for the given predictions and true values</li> <li>– higher_better: A boolean indicating whether higher values indicate a better fit. For example, this would be FALSE for metrics like MAE or RMSE.</li> </ul> </li> <li>• <b>c. list:</b> If a list is given, it should only contain character vectors and functions. These should follow the requirements from the descriptions above.</li> </ul>
verbose	verbosity for output, if $\leq 0$ , 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 <a href="#">the parameter documentation</a> 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,
                 objective = "regression_l2", 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$fixed_effect # Predicted fixed effect from tree ensemble
# Sum them up to obtain a single prediction
pred$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,
                 objective = "regression_l2", 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$fixed_effect # Predicted fixed effect from tree ensemble
# Sum them up to obtain a single prediction
pred$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$set_prediction_data(group_data_pred = group_data[-train_ind,1])
# Training with validation data and use_gp_model_for_validation = TRUE
bst <- gpb.train(data = dtrain, gp_model = gp_model, nrounds = 100,
                 learning_rate = 0.05, max_depth = 6, min_data_in_leaf = 5,
                 objective = "regression_l2", verbose = 1, valids = valids,
                 early_stopping_rounds = 10, use_gp_model_for_validation = TRUE)
print(paste0("Optimal number of iterations: ", bst$best_iter,
            ", best test error: ", bst$best_score))
# Plot validation error
val_error <- unlist(bst$record_evals$test$12$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,
                 objective = "regression_l2", 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$12$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,
              objective = "regression_l2")
# 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,
                 objective = "regression_l2",
                 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, 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 <code>gpb.Dataset</code> object, used for training. Some functions, such as <code>gpb.cv</code> , may allow you to pass other types of data like matrix and then separately supply label as a keyword argument.   |
| label  | Vector of labels, used if data is not an <code>gpb.Dataset</code>   |
| weight | vector of response values. If not NULL, will set to dataset   |
| params | list of ("tuning") parameters. See <a href="#">the parameter documentation</a> for more information. A few key parameters: <ul style="list-style-type: none"> <li>• <code>learning_rate</code> 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</li> <li>• <code>num_leaves</code> Number of leaves in a tree. Tuning parameter for tree-boosting (default = 31)</li> <li>• <code>min_data_in_leaf</code> Minimal number of samples per leaf. Tuning parameter for tree-boosting (default = 20)</li> <li>• <code>max_depth</code> Maximal depth of a tree. Tuning parameter for tree-boosting (default = no limit)</li> <li>• <code>leaves_newton_update</code> 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)</li> <li>• <code>train_gp_model_cov_pars</code> If TRUE, the covariance parameters of the Gaussian process are estimated in every boosting iterations, otherwise the <code>gp_model</code> 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 <code>gp_model</code> (default = TRUE).</li> <li>• <code>use_gp_model_for_validation</code> If TRUE, the Gaussian process is also used (in addition to the tree model) for calculating predictions on the validation data (default = TRUE)</li> <li>• <code>use_nesterov_acc</code> Set this to TRUE to do boosting with Nesterov acceleration (default = FALSE). Can currently only be used for <code>tree_learner = "serial"</code> (default option)</li> <li>• <code>nesterov_acc_rate</code> Acceleration rate for momentum step in case Nesterov accelerated boosting is used (default = 0.5)</li> <li>• <code>boosting</code> Boosting type. "gbdt", "rf", "dart" or "goss". Only "gbdt" allows for doing Gaussian process boosting.</li> </ul> |

	<ul style="list-style-type: none"> <li>• <b>num_threads</b> Number of threads. For the best speed, set this to the number of real CPU cores(<code>parallel::detectCores(logical = FALSE)</code>), not the number of threads (most CPU using hyper-threading to generate 2 threads per CPU core).</li> </ul>
nrounds	number of boosting iterations (= number of trees). This is the most important tuning parameter for boosting. Default = 100
gp_model	A GPMoDel object that contains the random effects (Gaussian process and / or grouped random effects) model
use_gp_model_for_validation	Boolean (default = TRUE). 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 (default = TRUE). 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 <code>init_cov_pars</code> parameter when creating the gp_model
valids	a list of <code>gpb.Dataset</code> objects, used for validation
obj	objective function, can be character or custom objective function. Examples include <code>regression</code> , <code>regression_l1</code> , <code>huber</code> , <code>binary</code> , <code>lambdaRank</code> , <code>multiclass</code> , <code>multiclass</code>
eval	<p>evaluation function(s). This can be a character vector, function, or list with a mixture of strings and functions.</p> <ul style="list-style-type: none"> <li>• <b>a. character vector:</b> If you provide a character vector to this argument, it should contain strings with valid evaluation metrics. See <a href="#">the "metric" section of the parameter documentation</a> for a list of valid metrics.</li> <li>• <b>b. function:</b> You can provide a custom evaluation function. This should accept the keyword arguments <code>preds</code> and <code>dtrain</code> and should return a named list with three elements: <ul style="list-style-type: none"> <li>– <code>name</code>: A string with the name of the metric, used for printing and storing results.</li> <li>– <code>value</code>: A single number indicating the value of the metric for the given predictions and true values</li> <li>– <code>higher_better</code>: A boolean indicating whether higher values indicate a better fit. For example, this would be FALSE for metrics like MAE or RMSE.</li> </ul> </li> <li>• <b>c. list:</b> If a list is given, it should only contain character vectors and functions. These should follow the requirements from the descriptions above.</li> </ul>
verbose	verbosity for output, if $\leq 0$ , also will disable the print of evaluation during training
record	Boolean, TRUE will record iteration message to <code>booster\$record_evals</code>
eval_freq	evaluation output frequency, only effect when <code>verbose &gt; 0</code>

<code>early_stopping_rounds</code>	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 <code>early_stopping_rounds</code> consecutive boosting rounds. If training stops early, the returned model will have attribute <code>best_iter</code> set to the iteration number of the best iteration.
<code>init_model</code>	path of model file of <code>gpb.Booster</code> object, will continue training from this model
<code>colnames</code>	feature names, if not null, will use this to overwrite the names in dataset
<code>categorical_feature</code>	categorical features. This can either be a character vector of feature names or an integer vector with the indices of the features (e.g. <code>c(1L, 10L)</code> to say "the first and tenth columns").
<code>callbacks</code>	List of callback functions that are applied at each iteration.
<code>...</code>	Additional arguments passed to <code>gpb.train</code> . For example <ul style="list-style-type: none"> <li>• <code>valids</code>: a list of <code>gpb.Dataset</code> objects, used for validation</li> <li>• <code>obj</code>: objective function, can be character or custom objective function. Examples include <code>regression</code>, <code>regression_l1</code>, <code>huber</code>, <code>binary</code>, <code>lambdarank</code>, <code>multiclass</code>, <code>multiclass</code></li> <li>• <code>eval</code>: evaluation function, can be (a list of) character or custom eval function</li> <li>• <code>record</code>: Boolean, TRUE will record iteration message to <code>booster\$record_evals</code></li> <li>• <code>colnames</code>: feature names, if not null, will use this to overwrite the names in dataset</li> <li>• <code>categorical_feature</code>: categorical features. This can either be a character vector of feature names or an integer vector with the indices of the features (e.g. <code>c(1L, 10L)</code> to say "the first and tenth columns").</li> <li>• <code>reset_data</code>: 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</li> </ul>

**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,
               objective = "regression_l2", 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,
               objective = "regression_l2", 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**

```
GPMoel(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, cov_fct_taper_range = 1, vecchia_approx = FALSE,
  num_neighbors = 30L, vecchia_ordering = "none",
  vecchia_pred_type = "order_obs_first_cond_obs_only",
  num_neighbors_pred = num_neighbors, cluster_ids = NULL,
  free_raw_data = FALSE, likelihood = "gaussian")
```

**Arguments**

- 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. The following covariance functions are available: "exponential", "gaussian", "matern", "powered\_exponential", "wendland", and "exponential\_tapered". For "exponential", "gaussian", and "powered\_exponential", we follow the notation and parametrization of Diggle and Ribeiro (2007). For "matern", we follow the notation of Rasmussen and Williams (2006). For "wendland", we follow the notation of Bevilacqua et al. (2019). A covariance function with the suffix

	"_tapered" refers to a covariance function that is multiplied by a compactly supported Wendland covariance function (= tapering)
cov_fct_shape	A numeric specifying the shape parameter of the covariance function (=smoothness parameter for Matern and Wendland covariance). For the Wendland covariance function, we follow the notation of Bevilacqua et al. (2019)). This parameter is irrelevant for some covariance functions such as the exponential or Gaussian
cov_fct_taper_range	A numeric specifying the range parameter of the Wendland covariance function / taper. We follow the notation of Bevilacqua et al. (2019)
vecchia_approx	A boolean. If TRUE, the Vecchia approximation is used
num_neighbors	An integer specifying the number of neighbors for the Vecchia approximation
vecchia_ordering	A string specifying the ordering used in the Vecchia approximation. "none" means the default ordering is used, "random" uses a random ordering
vecchia_pred_type	A string specifying the type of Vecchia approximation used for making predictions. "order_obs_first_cond_obs_only" = observed data is ordered first and the neighbors are only observed points, "order_obs_first_cond_all" = observed data is ordered first and the neighbors are selected among all points (observed + predicted), "order_pred_first" = predicted data is ordered first for making predictions, "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
num_neighbors_pred	an integer specifying the number of neighbors for the Vecchia approximation for making predictions
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
likelihood	A string specifying the likelihood function (distribution) of the response variable Default = "gaussian"

**Value**

A GPModel containing contains 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 *Shared parameter docs*

---

## Description

Parameter docs shared by GPMoDel, gpb.cv, and gpboost

## Arguments

likelihood	A string specifying the likelihood function (distribution) of the response variable Default = "gaussian"
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. The following covariance functions are available: "exponential", "gaussian", "matern", "powered_exponential", "wendland", and "exponential_tapered". For "exponential", "gaussian", and "powered_exponential", we follow the notation and parametrization of Diggle and Ribeiro (2007). For "matern", we follow the notation of Rasmussen and Williams (2006). For "wendland", we follow the notation of Bevilacqua et al. (2019). A covariance function with the suffix "_tapered" refers to a covariance function that is multiplied by a compactly supported Wendland covariance function (= tapering)
cov_fct_shape	A numeric specifying the shape parameter of the covariance function (=smoothness parameter for Matern and Wendland covariance). For the Wendland covariance function, we follow the notation of Bevilacqua et al. (2019)). This parameter is irrelevant for some covariance functions such as the exponential or Gaussian
cov_fct_taper_range	A numeric specifying the range parameter of the Wendland covariance function / taper. We follow the notation of Bevilacqua et al. (2019)
vecchia_approx	A boolean. If TRUE, the Vecchia approximation is used
num_neighbors	An integer specifying the number of neighbors for the Vecchia approximation
vecchia_ordering	A string specifying the ordering used in the Vecchia approximation. "none" means the default ordering is used, "random" uses a random ordering
vecchia_pred_type	A string specifying the type of Vecchia approximation used for making predictions. "order_obs_first_cond_obs_only" = observed data is ordered first and the neighbors are only observed points, "order_obs_first_cond_all" = observed data is ordered first and the neighbors are selected among all points (observed + predicted), "order_pred_first" = predicted data is ordered first for making predictions, "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
num_neighbors_pred	an integer specifying the number of neighbors for the Vecchia approximation for making predictions

cg_delta_conv_pred	a numeric specifying the tolerance level for L2 norm of residuals for checking convergence in conjugate gradient algorithm when being used for prediction
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 model fitting / optimization <ul style="list-style-type: none"> <li>optimizer_cov Optimizer used for estimating covariance parameters. Options: "gradient_descent", "fisher_scoring", "nelder_mead", "bfgs", "adam". Default= gradient_descent"</li> <li>optimizer_coef Optimizer used for estimating linear regression coefficients, if there are any (for the GPBoost algorithm there are usually none). Options: "gradient_descent", "wls", "nelder_mead", "bfgs", "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. Default="wls" for Gaussian data and "gradient_descent" for other likelihoods. If 'optimizer_cov' is set to "nelder_mead", "bfgs", or "adam", 'optimizer_coef' is automatically also set to the same value.</li> <li>maxit Maximal number of iterations for optimization algorithm. Default=1000</li> <li>delta_rel_conv Convergence tolerance. The algorithm stops if the relative change in either the (approximate) log-likelihood or the parameters is below this value. For "bfgs" and "adam", the L2 norm of the gradient is used instead of the relative change in the log-likelihood. Default=1E-6</li> <li>convergence_criterion The convergence criterion used for terminating the optimization algorithm. Options: "relative_change_in_log_likelihood" (default) or "relative_change_in_parameters"</li> <li>init_coef Initial values for the regression coefficients (if there are any, can be NULL). Default=NULL</li> <li>init_cov_pars Initial values for covariance parameters of Gaussian process and random effects (can be NULL). Default=NULL</li> <li>lr_coef Learning rate for fixed effect regression coefficients if gradient descent is used. Default=0.1</li> <li>lr_cov Learning rate for covariance parameters. If &lt;= 0, internal default values are used. Default value = 0.1 for "gradient_descent" and 1. for "fisher_scoring"</li> <li>use_nesterov_acc If TRUE Nesterov acceleration is used. This is used only for gradient descent. Default=TRUE</li> <li>acc_rate_coef Acceleration rate for regression coefficients (if there are any) for Nesterov acceleration. Default=0.5</li> </ul>

- acc\_rate\_cov Acceleration rate for covariance parameters for Nesterov acceleration. Default=0.5
- momentum\_offset Number of iterations for which no momentum is applied in the beginning. Default=2
- trace If TRUE, information on the progress of the parameter optimization is printed. Default=FALSE
- std\_dev 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)

fixed\_effects A vector of optional external fixed effects which are held fixed during training.

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

---

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	<i>Example data for the GPBoost package</i>
-----------------	---

---

**Description**

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

**Usage**

```
data(GPBoost_data)
```

---

loadGPModel	<i>Load a GPModel from a file</i>
-------------	-----------------------------------

---

**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

```

---

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, vecchia_pred_type = NULL,
        num_neighbors_pred = -1, predict_cov_mat = FALSE, predict_var = FALSE,
        ignore_gp_model = FALSE, rawscore = 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)
vecchia_pred_type	A string specifying the type of Vecchia approximation used for making predictions. "order_obs_first_cond_obs_only" = observed data is ordered first and the neighbors are only observed points, "order_obs_first_cond_all" = observed data is ordered first and the neighbors are selected among all points (observed + predicted), "order_pred_first" = predicted data is ordered first for making predictions, "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
num_neighbors_pred	an integer specifying the number of neighbors for the Vecchia approximation for making predictions
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
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 deprecated, use the renamed equivalent argument pred_latent instead
...	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 variances of the response variable. If there is no `gp_model` or `predcontrib` or `ignore_gp_model` are `TRUE`, the result contains predictions from the tree-boost 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,
               objective = "regression_l2", 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,
              objective = "regression_l2", 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, vecchia_pred_type = NULL,
        num_neighbors_pred = NULL, cg_delta_conv_pred = NULL,
        predict_response = FALSE, ...)

```

**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)
vecchia_pred_type	A string specifying the type of Vecchia approximation used for making predictions. "order_obs_first_cond_obs_only" = observed data is ordered first and the neighbors are only observed points, "order_obs_first_cond_all" = observed data is ordered first and the neighbors are selected among all points (observed + predicted), "order_pred_first" = predicted data is ordered first for making predictions, "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
num_neighbors_pred	an integer specifying the number of neighbors for the Vecchia approximation for making predictions

```

cg_delta_conv_pred
    a numeric specifying the tolerance level for L2 norm of residuals for checking
    convergence in conjugate gradient algorithm when being used for prediction
predict_response
    A boolean. If TRUE, the response variable (label) is predicted, otherwise the
    latent random effects (this is only relevant for non-Gaussian data)
...
    (not used, ignore this, simply here that there is no CRAN warning)

```

**Value**

Predictions made using a GPModel. It returns a list of length three. The first entry ('mu') is the predicted mean, the second entry ('cov') is the predicted covariance matrix (=NULL if 'predict\_cov\_mat=FALSE'), and the third entry ('var') are predicted variances (=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
```

*Generic 'predict\_training\_data\_random\_effects' method for a GPMoDel*

---

### Description

Generic 'predict\_training\_data\_random\_effects' method for a GPMoDel

### Usage

```
predict_training_data_random_effects(gp_model)
```

### Arguments

gp\_model            A GPMoDel

### Value

A GPMoDel

### Author(s)

Fabio Sigris

### 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)
```

## Arguments

gp\_model            A GPModel

## 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	<i>Save a GPModel</i>
-------------	-----------------------

---

**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

<code>object</code>	R object to serialize.
<code>file</code>	a connection or the name of the file where the R object is saved to or read from.
<code>ascii</code>	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.
<code>version</code>	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.
<code>compress</code>	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.
<code>refhook</code>	a hook function for handling reference objects.
<code>raw</code>	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 gpbboost 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\_prediction\_data    *Generic 'set\_prediction\_data' method for a GPModel*

---

**Description**

Generic 'set\_prediction\_data' method for a GPModel

**Usage**

```
set_prediction_data(gp_model, 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,
  vecchia_pred_type = NULL, num_neighbors_pred = NULL,
  cg_delta_conv_pred = NULL)
```

**Arguments**

gp_model	A GPModel
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)
vecchia_pred_type	A string specifying the type of Vecchia approximation used for making predictions. "order_obs_first_cond_obs_only" = observed data is ordered first and the neighbors are only observed points, "order_obs_first_cond_all" = observed data is ordered first and the neighbors are selected among all points (observed + predicted), "order_pred_first" = predicted data is ordered first for making predictions, "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
num_neighbors_pred	an integer specifying the number of neighbors for the Vecchia approximation for making predictions
cg_delta_conv_pred	a numeric specifying the tolerance level for L2 norm of residuals for checking convergence in conjugate gradient algorithm when being used for prediction

**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, 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,
  vecchia_pred_type = NULL, num_neighbors_pred = NULL,
  cg_delta_conv_pred = NULL)
```

**Arguments**

gp_model	A GPModel
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)
vecchia_pred_type	A string specifying the type of Vecchia approximation used for making predictions. "order_obs_first_cond_obs_only" = observed data is ordered first and the neighbors are only observed points, "order_obs_first_cond_all" = observed data is ordered first and the neighbors are selected among all points (observed + predicted), "order_pred_first" = predicted data is ordered first for making predictions, "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
num_neighbors_pred	an integer specifying the number of neighbors for the Vecchia approximation for making predictions

cg\_delta\_conv\_pred

a numeric specifying the tolerance level for L2 norm of residuals for checking convergence in conjugate gradient algorithm when being used for prediction

### 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 <code>gpb.Dataset</code>
...	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

## \* datasets

- agaricus.test, 3
  - agaricus.train, 4
  - bank, 5
  - coords, 5
  - coords\_test, 5
  - GPBoost\_data, 54
  - group\_data, 60
  - group\_data\_test, 61
  - X, 79
  - X\_test, 79
  - y, 79
- agaricus.test, 3
- agaricus.train, 4
- bank, 5
- barplot, 38
- coords, 5
- coords\_test, 5
- dim.gpb.Dataset, 6
- dimnames.gpb.Dataset, 7
- dimnames<- .gpb.Dataset  
(dimnames.gpb.Dataset), 7
- fit, 8
- fit.GPModel, 9
- fitGPModel, 11
- get\_nested\_categories, 17
- getinfo, 16
- gpb.convert\_with\_rules, 18
- gpb.cv, 19, 21, 30, 45, 50
- gpb.Dataset, 21, 23, 31, 50
- gpb.Dataset.construct, 24
- gpb.Dataset.create.valid, 25
- gpb.Dataset.save, 26
- gpb.Dataset.set.categorical, 26
- gpb.Dataset.set.reference, 27
- gpb.dump, 28
- gpb.get.eval.result, 29
- gpb.grid.search.tune.parameters, 30
- gpb.importance, 33, 38
- gpb.interprete, 34, 39
- gpb.load, 35
- gpb.model.dt.tree, 36
- gpb.plot.importance, 37
- gpb.plot.interpretation, 38
- gpb.plot.part.dep.interact, 40
- gpb.plot.partial.dependence, 41
- gpb.save, 43
- gpb.train, 44, 52
- gpboost, 49
- GPBoost\_data, 54
- GPModel, 54
- GPModel\_shared\_params, 57
- group\_data, 60
- group\_data\_test, 61
- loadGPModel, 61
- predict.gpb.Booster, 62
- predict.GPModel, 65
- predict\_training\_data\_random\_effects,  
68
- predict\_training\_data\_random\_effects.GPModel,  
69
- readRDS, 70
- readRDS.gpb.Booster, 70
- saveGPModel, 71
- saveRDS.gpb.Booster, 72
- set\_prediction\_data, 74
- set\_prediction\_data.GPModel, 75
- setinfo, 73
- slice, 77
- summary.GPModel, 78
- X, 79



$\chi$ \_test, 79

y, 79