Package 'gpboost'

October 19, 2026

Type Package

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

Version 1.8.0

Date 2026-10-19

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

Encoding UTF-8

License Apache License (== 2.0) | file LICENSE

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

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

NeedsCompilation yes

Biarch true

Suggests testthat

Depends R (>= 3.5), R6 (>= 2.4.0)

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

SystemRequirements C++17

Config/roxygen2/version 8.0.0

RoxygenNote 6.0.1

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

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

Maintainer Fabio Sigrist <fabiosigrist@gmail.com>

Repository CRAN

Date/Publication 2026-10-19 18:50:02 UTC

Contents

agaricus.test 4
agaricus.train 5
bank 5
coords 6
coords_test 6
dim.gpb.Dataset 6
dimnames.gpb.Dataset 7
fit 8
fit.GPModel 13
fitGPModel 19
getinfo 38
get_aux_pars 39
get_aux_pars.GPModel 40

get_coef . . . 41
get_coef.GPModel . . . 42
get_cov_pars . . . 43
get_cov_pars.GPModel . . . 43
get_nested_categories . . . 44
gpb.convert_with_rules . . . 45
gpb.cv . . . 46
gpb.Dataset . . . 51
gpb.Dataset.construct . . . 52
gpb.Dataset.create.valid . . . 52
gpb.Dataset.save . . . 53
gpb.Dataset.set.categorical . . . 54
gpb.Dataset.set.reference . . . 54
gpb.dump . . . 55
gpb.get.default.num.threads . . . 56
gpb.get.eval.result . . . 57
gpb.get.num.threads . . . 58
gpb.grid.search.tune.parameters . . . 58
gpb.importance . . . 62
gpb.interprete . . . 63
gpb.load . . . 64
gpb.model.dt.tree . . . 65
gpb.plot.importance . . . 67
gpb.plot.interpretation . . . 68
gpb.plot.part.dep.interact . . . 69
gpb.plot.partial.dependence . . . 71
gpb.save . . . 72
gpb.set.default.num.threads . . . 74
gpb.set.num.threads . . . 75
gpb.train . . . 76
gpb.tune.num.threads . . . 81
gpboost . . . 84
GPBoost_data . . . 88
GPModel . . . 89
group_data . . . 101
group_data_test . . . 101
loadGPModel . . . 102
neg_log_likelihood . . . 103
neg_log_likelihood.GPModel . . . 104
predict.gpb.Booster . . . 105
predict.GPModel . . . 108
predict_training_data_random_effects . . . 111
predict_training_data_random_effects.GPModel . . . 112
readRDS.gpb.Booster . . . 113
saveGPModel . . . 114
saveRDS.gpb.Booster . . . 115
setinfo . . . 116
set_optim_params . . . 117

set_optim_params.GPModel 123
set_prediction_data 128
set_prediction_data.GPModel 131
slice 133
summary.GPModel 134
X 135
X_test 135
y 135

Index **136**

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

Description

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

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

Usage

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

Format

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

References

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

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

agaricus.train *Training part from Mushroom Data Set*

Description

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

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

Usage

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

Format

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

References

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

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

bank *Bank Marketing Data Set*

Description

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

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

Usage

```
data(bank)
```

Format

A data.table with 4521 rows and 17 variables

References

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

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

coords *Coordinates for example data for the GPBoost package*

Description

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

Usage

```
data(GPBoost_data)
```

coords_test *Test coordinates for example data for the GPBoost package*

Description

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

Usage

```
data(GPBoost_data)
```

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

Description

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

Usage

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

Arguments

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

Details

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

Value

a vector of numbers of rows and of columns

Examples

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

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

---

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

---

Description

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

Usage

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

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

Arguments

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

Details

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

Value

A list with the dimension names of the dataset

A list with the dimension names of the dataset

Examples

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

---

fit *Generic 'fit' method for a* GPModel

Description

Generic 'fit' method for a GPModel

Usage

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

Arguments

gp_model	a GPModel
y	A vector with response variable data
X	A matrix with numeric covariate data for the fixed effects linear regression term (if there is one)
params	A list with parameters for the estimation / optimization • trace: boolean (default = FALSE). If TRUE, information on the progress of the parameter optimization is printed • init_cov_pars: vector with numeric elements (default = NULL). Initial values for covariance parameters of Gaussian process and random effects (can be NULL). The order is same as the order of the parameters in the summary function: first is the error variance (only for "gaussian" likelihood), next follow the variances of the grouped random effects (if there are any, in the order provided in 'group_data'), and then follow the marginal variance and the ranges of the Gaussian process. If there are multiple Gaussian processes, then the variances and ranges follow alternatingly. If 'init_cov_pars = NULL', an internal choice is used that depends on the likelihood and the random effects type and covariance function. If you select the option 'trace = TRUE' in the 'params' argument, you will see the first initial covariance parameters in iteration 0. • init_coef: vector with numeric elements (default = NULL). Initial values for the regression coefficients (if there are any, can be NULL)

- init_aux_pars: `vector` with `numeric` elements (default = NULL). Initial values for additional parameters for non-Gaussian likelihoods (e.g., shape parameter of a gamma or negative_binomial likelihood)
- init_coef_aux_pars_from_iid_model: `boolean` (default = TRUE). If TRUE, regression coefficients and auxiliary parameters are initialized from an iid model (only for models with a linear regression term). This option is ignored if init_coef is provided. If init_aux_pars is provided but init_coef is not, only regression coefficients are initialized from an iid model. For likelihood = "gaussian_heteroscedastic_fixed_and_random", if init_cov_pars is not provided, the initial covariance parameters are obtained from a homoscedastic Gaussian process model, and the regression coefficients are then initialized from this model instead: the coefficients of the mean from its coefficients and the intercept of the log-error variance from the logarithm of its error variance.
- estimate_cov_par_index: `vector` with `integer` (default = -1). This allows for disabling the estimation of some (or all) covariance parameters. If 'estimate_cov_par_index' = -1, all covariance parameters are estimated. If estimate_cov_par_index != -1, this should be a vector with length equal to the number of covariance parameters, and estimate_cov_par_index[i] should be of bool type indicating whether parameter number i is estimated or not. For instance, estimate_cov_par_index = c(1,1,0) means that the first two covariance parameters are estimated and the last one not. Parameters that are not estimated are kept at their initial values (see 'init_cov_pars'). They are treated as known constants when calculating the standard errors of the estimated parameters, and their own standard errors are NaN.
- estimate_aux_pars: `boolean` (default = TRUE). If TRUE, additional parameters for non-Gaussian likelihoods are also estimated (e.g., shape parameter of a gamma or negative_binomial likelihood)
- optimizer_cov: `string` (default = "lbfgs"). Optimizer used for estimating covariance parameters. Options: "lbfgs", "gradient_descent", "fisher_scoring", "newton", "nelder_mead". If there are additional auxiliary parameters for non-Gaussian likelihoods, 'optimizer_cov' is also used for those
- optimizer_coef: `string` (default = "wls" for Gaussian likelihoods and "lbfgs" for other likelihoods). Optimizer used for estimating linear regression coefficients, if there are any (for the GPBoost algorithm there are usually none). Options: "gradient_descent", "lbfgs", "wls", "nelder_mead". Gradient descent steps are done simultaneously with gradient descent steps for the covariance parameters. "wls" refers to doing coordinate descent for the regression coefficients using weighted least squares. If 'optimizer_cov' is set to "nelder_mead" or "lbfgs", 'optimizer_coef' is automatically also set to the same value.
- maxit: `integer` (default = 1000). Maximal number of iterations for optimization algorithm. If maxit = -999, internal default values are used
- delta_rel_conv: `numeric` (default = 1E-6 except for "nelder_mead" for which the default is 1E-8). Convergence tolerance. The algorithm stops if the relative change in either the (approximate) log-likelihood or the parameters is below this value. If delta_rel_conv = -999, internal default values are used

- cg_max_num_it: `integer` (default = 1000). Maximal number of iterations for conjugate gradient algorithms. If cg_max_num_it = -999, internal default values are used
- cg_max_num_it_tridiag: `integer` (default = 1000). Maximal number of iterations for conjugate gradient algorithm when being run as Lanczos algorithm for tridiagonalization. If cg_max_num_it_tridiag = -999, internal default values are used
- cg_delta_conv: `numeric` (default = 1E-2). Tolerance level for L2 norm of residuals for checking convergence in conjugate gradient algorithm when being used for parameter estimation. This is the tolerance of the "absolute" criterion only, the "relative" criterion is controlled by cg_rel_tol and cg_abs_tol. If cg_delta_conv = -999, internal default values are used
- cg_convergence_criterion: `string` (default = "absolute"). Stopping rule of conjugate gradient algorithms.
  - "absolute": stop when ||r||_2 < cg_delta_conv
  - "relative": stop when ||r||_2 <= max(cg_abs_tol, cg_rel_tol * ||b||_2), where b is the right-hand side of the linear system. The cg_abs_tol floor makes this robust for right-hand sides that are zero or very small
- cg_rel_tol: `numeric` (default = 1E-6). Relative tolerance of the "relative" stopping rule. If cg_rel_tol = -999, internal default values are used
- cg_abs_tol: `numeric` (default = 1E-8). Absolute tolerance floor of the "relative" stopping rule, which makes it robust for right-hand sides that are zero or very small. This is independent of cg_delta_conv, so that the two criteria can be varied separately. If cg_abs_tol = -999, internal default values are used
- cg_multi_rhs_convergence: `string` (default = "average"). How the stopping rule is aggregated over the columns of a linear system with several right-hand sides, such as the stochastic Lanczos quadrature. For cg_convergence_criterion = "relative", every right-hand side is normalized by its own norm, i.e. q_j = ||r_j||_2 / max(cg_abs_tol, cg_rel_tol * ||b_j||_2):
  - "average": stop when mean(q_j) <= 1
  - "max": stop when max(q_j) <= 1
  - "per_rhs": stop iterating on column j individually as soon as q_j <= 1

  For cg_convergence_criterion = "absolute" there is no per-column normalization:
  - "average": stop when mean(||r_j||_2) < cg_delta_conv. This is the historic behavior and the default
  - "max": stop when ||r_j||_2 < cg_delta_conv for all j
  - "per_rhs": stop iterating on column j as soon as ||r_j||_2 < cg_delta_conv
- num_rand_vec_trace: `integer` (default = 50). Number of random vectors (e.g., Rademacher) for stochastic approximation of the trace of a matrix. If num_rand_vec_trace = -999, internal default values are used
- reuse_rand_vec_trace: `boolean` (default = TRUE). If true, random vectors (e.g., Rademacher) for stochastic approximations of the trace of a matrix are sampled only once at the beginning of the parameter estimation and reused in later trace approximations. Otherwise they are sampled every time a trace is calculated

- seed_rand_vec_trace: `integer` (default = 1). Seed number to generate random vectors (e.g., Rademacher)
- cg_preconditioner_type (`string`): Type of preconditioner used for conjugate gradient algorithms.
  - Options for grouped random effects:
    * "ssor" (= default): SSOR preconditioner
    * "incomplete_cholesky": zero fill-in incomplete Cholesky factorization
  - Options for likelihood != "gaussian" and gp_approx == "vecchia" or likelihood == "gaussian" and gp_approx == "vecchia_latent":
    * "vadu" (= default): (B^T * (D^-1 + W) * B) as preconditioner for inverting (B^T * D^-1 * B + W), where B^T * D^-1 * B approx= Sigma^-1
    * "fitc": FITC / modified predictive process preconditioner for inverting (B^-1 * D * B^-T + W^-1)
    * "pivoted_cholesky": (Lk * Lk^T + W^-1) as preconditioner for inverting (B^-1 * D * B^-T + W^-1), where Lk is a low-rank pivoted Cholesky approximation for Sigma and B^-1 * D * B^-T approx= Sigma
    * "incomplete_cholesky": zero fill-in incomplete (reverse) Cholesky factorization of (B^T * D^-1 * B + W) using the sparsity pattern of B^T * D^-1 * B approx= Sigma^-1
  - Options for likelihood != "gaussian" and gp_approx == "full_scale_vecchia":
    * "fitc" (= default): FITC / modified predictive process preconditioner
    * "vifdu": VIF with diagonal update preconditioner
  - Options for likelihood == "gaussian" and gp_approx == "full_scale_tapering":
    * "fitc" (= default): modified predictive process preconditioner
    * "none": no preconditioner
- fitc_piv_chol_preconditioner_rank (`integer`): Rank of the FITC and pivoted Cholesky decomposition preconditioners for iterative methods for Vecchia and VIF approximations (for full_scale_tapering, the same inducing points as in the approximation as used). If fitc_piv_chol_preconditioner_rank = -999, internal default values are used:
  - 200 for the FITC preconditioner
  - 50 for the pivoted Cholesky decomposition preconditioner
- convergence_criterion: `string` (default = "relative_change_in_log_likelihood", only relevant for "gradient_descent", "fisher_scoring", and "newton"). The convergence criterion used for terminating the optimization algorithm. Options: "relative_change_in_log_likelihood" or "relative_change_in_parameters". If convergence_criterion = "default", internal default values are used
- lr_cov: `numeric` (default = 0.1 for "gradient_descent" and 1. otherwise, relevant for "gradient_descent", "fisher_scoring", "newton", and "lbfgs"). Initial learning rate for covariance parameters if a gradient-based optimization method is used
  - If lr_cov = -999, internal default values are used (0.1 for "gradient_descent" and 1. otherwise)

  - If there are additional auxiliary parameters for non-Gaussian likelihoods, 'lr_cov' is also used for those
  - For "gradient_descent", "fisher_scoring", and "newton", this is the learning rate by which the search direction is multiplied. It can be decreased, and increased again, during the optimization
  - For "lbfgs" and "lbfgs_linesearch_nocedal_wright", this is not a learning rate by which every update is multiplied. It only determines the first trial step, which is the negative gradient divided by its norm and multiplied by 'lr_cov', i.e., a step of length 'lr_cov'. The subsequent steps are determined by the approximate Hessian of the L-BFGS algorithm and the line search. 'lr_cov' has no effect when the approximate Hessian of a previous call is reused (see 'reuse_learning_rates_gp_model' of the GPBoost algorithm)
- lr_coef: `numeric` (default = 0.1, only relevant for "gradient_descent", "fisher_scoring", and "newton"). Learning rate for fixed effect regression coefficients if gradient descent is used. If lr_coef = -999, internal default values are used
- use_nesterov_acc: `boolean` (default = TRUE, only relevant for "gradient_descent"). If TRUE Nesterov acceleration is used. This is used only for gradient descent
- nesterov_schedule_version: `integer` (Default = 0, only relevant for "gradient_descent"). Which version of Nesterov schedule should be used. If nesterov_schedule_version = -999, internal default values are used.
- acc_rate_coef: `numeric` (default = 0.5, only relevant for "gradient_descent"). Acceleration rate for regression coefficients (if there are any) for Nesterov acceleration. If acc_rate_coef = -999, internal default values are used
- acc_rate_cov: `numeric` (default = 0.5, only relevant for "gradient_descent"). Acceleration rate for covariance parameters for Nesterov acceleration. If acc_rate_cov = -999, internal default values are used
- momentum_offset: `integer` (Default = 2, only relevant for "gradient_descent"). Number of iterations for which no momentum is applied in the beginning. If momentum_offset = -999, internal default values are used.
- m_lbfgs: `integer` (Default = 6). Number of corrections to approximate the inverse Hessian matrix for the "lbfgs" optimizer. If m_lbfgs = -999, internal default values are used
- cold_restart_lbfgs: `boolean` (Default = TRUE). If TRUE, restarts of the "lbfgs" optimizers are "cold" restarts: the regression coefficients, the auxiliary parameters, and the modes of the Laplace approximations are reset to their initial values and only the covariance parameters are kept. Such restarts can escape a local optimum in which the optimizer is stuck. If FALSE, "warm" restarts are done: the optimization simply continues from the current parameters with a re-initialized approximate Hessian. Only relevant if `max_num_restarts_lbfgs` > 0. Note that every cold restart is a full optimization with up to `maxit` iterations, and the number of iterations reported is the sum over all restarts
- max_num_restarts_lbfgs: `integer` (Default = 0). Maximal number of restarts of the "lbfgs" optimizers after they have terminated. A restart continues from the current parameters with a re-initialized approximate Hessian. What

else is reset depends on cold_restart_lbfgs. Restarts are only done as long as the negative log-likelihood still decreases by more than delta_rel_conv and as long as the total number of iterations is below maxit. This can help when the line search of "lbfgs" fails and the optimizer thus terminates prematurely without having converged. This can happen, e.g., for the "asymmetric_laplace" (= quantile regression) likelihood, whose approximate marginal likelihood is not smooth

- reset_modes_line_search_lbfgs: boolean (Default = FALSE). If TRUE, the mode finding of the Laplace approximation starts at the modes of the current parameters for every trial point of the line search of the "lbfgs" optimizer, and not at the modes of the previous trial point. Only relevant for non-Gaussian likelihoods and optimizer_cov = "lbfgs" or "lbfgs_linesearch_nocedal_wright
- delta_conv_mode_finding: numeric (Default = 1E-8, and 1E-10 for the likelihood "gaussian_heteroscedastic_fixed_and_random"). Convergence tolerance in mode finding algorithm for Laplace approximation for non-Gaussian likelihoods. If delta_conv_mode_finding = -999, internal default values are used

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

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

Author(s)

Fabio Sigrist

---

fit.GPModel *Fits a* GPModel

Description

Estimates the parameters of a GPModel by maximizing the marginal likelihood

Usage

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

Arguments

gp_model — a GPModel

y — A vector with response variable data

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

params — A list with parameters for the estimation / optimization

- trace: `boolean` (default = FALSE). If TRUE, information on the progress of the parameter optimization is printed
- init_cov_pars: `vector` with `numeric` elements (default = NULL). Initial values for covariance parameters of Gaussian process and random effects (can be NULL). The order is same as the order of the parameters in the summary function: first is the error variance (only for "gaussian" likelihood), next follow the variances of the grouped random effects (if there are any, in the order provided in 'group_data'), and then follow the marginal variance and the ranges of the Gaussian process. If there are multiple Gaussian processes, then the variances and ranges follow alternatingly. If 'init_cov_pars = NULL', an internal choice is used that depends on the likelihood and the random effects type and covariance function. If you select the option 'trace = TRUE' in the 'params' argument, you will see the first initial covariance parameters in iteration 0.
- init_coef: `vector` with `numeric` elements (default = NULL). Initial values for the regression coefficients (if there are any, can be NULL)
- init_aux_pars: `vector` with `numeric` elements (default = NULL). Initial values for additional parameters for non-Gaussian likelihoods (e.g., shape parameter of a gamma or negative_binomial likelihood)
- init_coef_aux_pars_from_iid_model: `boolean` (default = TRUE). If TRUE, regression coefficients and auxiliary parameters are initialized from an iid model (only for models with a linear regression term). This option is ignored if init_coef is provided. If init_aux_pars is provided but init_coef is not, only regression coefficients are initialized from an iid model. For likelihood = "gaussian_heteroscedastic_fixed_and_random", if init_cov_pars is not provided, the initial covariance parameters are obtained from a homoscedastic Gaussian process model, and the regression coefficients are then initialized from this model instead: the coefficients of the mean from its coefficients and the intercept of the log-error variance from the logarithm of its error variance.
- estimate_cov_par_index: `vector` with `integer` (default = -1). This allows for disabling the estimation of some (or all) covariance parameters. If 'estimate_cov_par_index' = -1, all covariance parameters are estimated. If estimate_cov_par_index != -1, this should be a vector with length equal to the number of covariance parameters, and estimate_cov_par_index[i] should be of bool type indicating whether parameter number i is estimated or not. For instance, estimate_cov_par_index = c(1,1,0) means that the first two covariance parameters are estimated and the last one not. Parameters that are not estimated are kept at their initial values (see 'init_cov_pars'). They are treated as known constants when calculating the standard errors of the estimated parameters, and their own standard errors are NaN.
- estimate_aux_pars: `boolean` (default = TRUE). If TRUE, additional parameters for non-Gaussian likelihoods are also estimated (e.g., shape parameter of a gamma or negative_binomial likelihood)
- optimizer_cov: `string` (default = "lbfgs"). Optimizer used for estimating covariance parameters. Options: "lbfgs", "gradient_descent", "fisher_scoring", "newton", "nelder_mead". If there are additional auxiliary parameters for non-Gaussian likelihoods, 'optimizer_cov' is also used for those

- optimizer_coef: `string` (default = "wls" for Gaussian likelihoods and "lbfgs" for other likelihoods). Optimizer used for estimating linear regression coefficients, if there are any (for the GPBoost algorithm there are usually none). Options: "gradient_descent", "lbfgs", "wls", "nelder_mead". Gradient descent steps are done simultaneously with gradient descent steps for the covariance parameters. "wls" refers to doing coordinate descent for the regression coefficients using weighted least squares. If 'optimizer_cov' is set to "nelder_mead" or "lbfgs", 'optimizer_coef' is automatically also set to the same value.
- maxit: `integer` (default = 1000). Maximal number of iterations for optimization algorithm. If maxit = -999, internal default values are used
- delta_rel_conv: `numeric` (default = 1E-6 except for "nelder_mead" for which the default is 1E-8). Convergence tolerance. The algorithm stops if the relative change in either the (approximate) log-likelihood or the parameters is below this value. If delta_rel_conv = -999, internal default values are used
- cg_max_num_it: `integer` (default = 1000). Maximal number of iterations for conjugate gradient algorithms. If cg_max_num_it = -999, internal default values are used
- cg_max_num_it_tridiag: `integer` (default = 1000). Maximal number of iterations for conjugate gradient algorithm when being run as Lanczos algorithm for tridiagonalization. If cg_max_num_it_tridiag = -999, internal default values are used
- cg_delta_conv: `numeric` (default = 1E-2). Tolerance level for L2 norm of residuals for checking convergence in conjugate gradient algorithm when being used for parameter estimation. This is the tolerance of the "absolute" criterion only, the "relative" criterion is controlled by cg_rel_tol and cg_abs_tol. If cg_delta_conv = -999, internal default values are used
- cg_convergence_criterion: `string` (default = "absolute"). Stopping rule of conjugate gradient algorithms.
  - "absolute": stop when ||r||_2 < cg_delta_conv
  - "relative": stop when ||r||_2 <= max(cg_abs_tol, cg_rel_tol * ||b||_2), where b is the right-hand side of the linear system. The cg_abs_tol floor makes this robust for right-hand sides that are zero or very small
- cg_rel_tol: `numeric` (default = 1E-6). Relative tolerance of the "relative" stopping rule. If cg_rel_tol = -999, internal default values are used
- cg_abs_tol: `numeric` (default = 1E-8). Absolute tolerance floor of the "relative" stopping rule, which makes it robust for right-hand sides that are zero or very small. This is independent of cg_delta_conv, so that the two criteria can be varied separately. If cg_abs_tol = -999, internal default values are used
- cg_multi_rhs_convergence: `string` (default = "average"). How the stopping rule is aggregated over the columns of a linear system with several right-hand sides, such as the stochastic Lanczos quadrature. For cg_convergence_criterion = "relative", every right-hand side is normalized by its own norm, i.e. q_j = ||r_j||_2 / max(cg_abs_tol, cg_rel_tol * ||b_j||_2):
  - "average": stop when mean(q_j) <= 1

  - "max": stop when max(q_j) <= 1
  - "per_rhs": stop iterating on column j individually as soon as q_j <= 1

  For cg_convergence_criterion = "absolute" there is no per-column normalization:

  - "average": stop when mean(||r_j||_2) < cg_delta_conv. This is the historic behavior and the default
  - "max": stop when ||r_j||_2 < cg_delta_conv for all j
  - "per_rhs": stop iterating on column j as soon as ||r_j||_2 < cg_delta_conv

- num_rand_vec_trace: `integer` (default = 50). Number of random vectors (e.g., Rademacher) for stochastic approximation of the trace of a matrix. If num_rand_vec_trace = -999, internal default values are used
- reuse_rand_vec_trace: `boolean` (default = TRUE). If true, random vectors (e.g., Rademacher) for stochastic approximations of the trace of a matrix are sampled only once at the beginning of the parameter estimation and reused in later trace approximations. Otherwise they are sampled every time a trace is calculated
- seed_rand_vec_trace: `integer` (default = 1). Seed number to generate random vectors (e.g., Rademacher)
- cg_preconditioner_type (`string`): Type of preconditioner used for conjugate gradient algorithms.
  - Options for grouped random effects:
    * "ssor" (= default): SSOR preconditioner
    * "incomplete_cholesky": zero fill-in incomplete Cholesky factorization
  - Options for likelihood != "gaussian" and gp_approx == "vecchia" or likelihood == "gaussian" and gp_approx == "vecchia_latent":
    * "vadu" (= default): (B^T * (D^-1 + W) * B) as preconditioner for inverting (B^T * D^-1 * B + W), where B^T * D^-1 * B approx= Sigma^-1
    * "fitc": FITC / modified predictive process preconditioner for inverting (B^-1 * D * B^-T + W^-1)
    * "pivoted_cholesky": (Lk * Lk^T + W^-1) as preconditioner for inverting (B^-1 * D * B^-T + W^-1), where Lk is a low-rank pivoted Cholesky approximation for Sigma and B^-1 * D * B^-T approx= Sigma
    * "incomplete_cholesky": zero fill-in incomplete (reverse) Cholesky factorization of (B^T * D^-1 * B + W) using the sparsity pattern of B^T * D^-1 * B approx= Sigma^-1
  - Options for likelihood != "gaussian" and gp_approx == "full_scale_vecchia":
    * "fitc" (= default): FITC / modified predictive process preconditioner
    * "vifdu": VIF with diagonal update preconditioner
  - Options for likelihood == "gaussian" and gp_approx == "full_scale_tapering":
    * "fitc" (= default): modified predictive process preconditioner
    * "none": no preconditioner

- fitc_piv_chol_preconditioner_rank (`integer`): Rank of the FITC and pivoted Cholesky decomposition preconditioners for iterative methods for Vecchia and VIF approximations (for full_scale_tapering, the same inducing points as in the approximation as used). If fitc_piv_chol_preconditioner_rank = -999, internal default values are used:
  - 200 for the FITC preconditioner
  - 50 for the pivoted Cholesky decomposition preconditioner
- convergence_criterion: `string` (default = "relative_change_in_log_likelihood", only relevant for "gradient_descent", "fisher_scoring", and "newton"). The convergence criterion used for terminating the optimization algorithm. Options: "relative_change_in_log_likelihood" or "relative_change_in_parameters". If convergence_criterion = "default", internal default values are used
- lr_cov: `numeric` (default = 0.1 for "gradient_descent" and 1. otherwise, relevant for "gradient_descent", "fisher_scoring", "newton", and "lbfgs"). Initial learning rate for covariance parameters if a gradient-based optimization method is used
  - If lr_cov = -999, internal default values are used (0.1 for "gradient_descent" and 1. otherwise)
  - If there are additional auxiliary parameters for non-Gaussian likelihoods, 'lr_cov' is also used for those
  - For "gradient_descent", "fisher_scoring", and "newton", this is the learning rate by which the search direction is multiplied. It can be decreased, and increased again, during the optimization
  - For "lbfgs" and "lbfgs_linesearch_nocedal_wright", this is not a learning rate by which every update is multiplied. It only determines the first trial step, which is the negative gradient divided by its norm and multiplied by 'lr_cov', i.e., a step of length 'lr_cov'. The subsequent steps are determined by the approximate Hessian of the L-BFGS algorithm and the line search. 'lr_cov' has no effect when the approximate Hessian of a previous call is reused (see 'reuse_learning_rates_gp_model' of the GPBoost algorithm)
- lr_coef: `numeric` (default = 0.1, only relevant for "gradient_descent", "fisher_scoring", and "newton"). Learning rate for fixed effect regression coefficients if gradient descent is used. If lr_coef = -999, internal default values are used
- use_nesterov_acc: `boolean` (default = TRUE, only relevant for "gradient_descent"). If TRUE Nesterov acceleration is used. This is used only for gradient descent
- nesterov_schedule_version: `integer` (Default = 0, only relevant for "gradient_descent"). Which version of Nesterov schedule should be used. If nesterov_schedule_version = -999, internal default values are used.
- acc_rate_coef: `numeric` (default = 0.5, only relevant for "gradient_descent"). Acceleration rate for regression coefficients (if there are any) for Nesterov acceleration. If acc_rate_coef = -999, internal default values are used
- acc_rate_cov: `numeric` (default = 0.5, only relevant for "gradient_descent"). Acceleration rate for covariance parameters for Nesterov acceleration. If acc_rate_cov = -999, internal default values are used

- momentum_offset: `integer` (Default = 2, only relevant for "gradient_descent"). Number of iterations for which no momentum is applied in the beginning. If momentum_offset = -999, internal default values are used.
- m_lbfgs: `integer` (Default = 6). Number of corrections to approximate the inverse Hessian matrix for the "lbfgs" optimizer. If m_lbfgs = -999, internal default values are used
- cold_restart_lbfgs: `boolean` (Default = TRUE). If TRUE, restarts of the "lbfgs" optimizers are "cold" restarts: the regression coefficients, the auxiliary parameters, and the modes of the Laplace approximations are reset to their initial values and only the covariance parameters are kept. Such restarts can escape a local optimum in which the optimizer is stuck. If FALSE, "warm" restarts are done: the optimization simply continues from the current parameters with a re-initialized approximate Hessian. Only relevant if `max_num_restarts_lbfgs` > 0. Note that every cold restart is a full optimization with up to `maxit` iterations, and the number of iterations reported is the sum over all restarts
- max_num_restarts_lbfgs: `integer` (Default = 0). Maximal number of restarts of the "lbfgs" optimizers after they have terminated. A restart continues from the current parameters with a re-initialized approximate Hessian. What else is reset depends on `cold_restart_lbfgs`. Restarts are only done as long as the negative log-likelihood still decreases by more than `delta_rel_conv` and as long as the total number of iterations is below `maxit`. This can help when the line search of "lbfgs" fails and the optimizer thus terminates prematurely without having converged. This can happen, e.g., for the `"asymmetric_laplace"` (= quantile regression) likelihood, whose approximate marginal likelihood is not smooth
- reset_modes_line_search_lbfgs: `boolean` (Default = FALSE). If TRUE, the mode finding of the Laplace approximation starts at the modes of the current parameters for every trial point of the line search of the "lbfgs" optimizer, and not at the modes of the previous trial point. Only relevant for non-Gaussian likelihoods and `optimizer_cov` = `"lbfgs"` or `"lbfgs_linesearch_nocedal_wright`
- delta_conv_mode_finding: `numeric` (Default = 1E-8, and 1E-10 for the likelihood "gaussian_heteroscedastic_fixed_and_random"). Convergence tolerance in mode finding algorithm for Laplace approximation for non-Gaussian likelihoods. If delta_conv_mode_finding = -999, internal default values are used

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

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

Value

A fitted `GPModel`

Author(s)

Fabio Sigrist

Examples

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


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

#--------------------Grouped random effects model: single-level random effect----------------
gp_model <- GPModel(group_data = group_data[,1], likelihood="gaussian")
fit(gp_model, y = y, X = X1)
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 = "matern", cov_fct_shape = 1.5,
                    likelihood="gaussian")
fit(gp_model, y = y, X = X1)
summary(gp_model)
# Make predictions
pred <- predict(gp_model, gp_coords_pred = coords_test,
                X_pred = X_test1, predict_cov_mat = TRUE)
pred$mu # Predicted (posterior) mean of GP
pred$cov # Predicted (posterior) covariance matrix of GP
```

`fitGPModel` *Fits a* `GPModel`

Description

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

Usage

```
fitGPModel(likelihood = "gaussian", group_data = NULL,
  group_rand_coef_data = NULL, ind_effect_group_rand_coef = NULL,
  drop_intercept_group_rand_effect = NULL, gp_coords = NULL,
```

```
gp_rand_coef_data = NULL, cov_function = "matern", cov_fct_shape = 1.5,
gp_approx = "none", num_parallel_threads = NULL, GPU_use = FALSE,
matrix_inversion_method = "default", weights = NULL,
additional_likelihood_data = NULL,
likelihood_learning_rate = 1, cov_fct_taper_range = 1,
cov_fct_taper_shape = 1, num_neighbors = NULL,
vecchia_ordering = "random", ind_points_selection = "kmeans++",
num_ind_points = NULL, cover_tree_radius = 1, seed = 0L,
cluster_ids = NULL, fidelity_specific_mean = TRUE, cov_fct_order = 1L,
free_raw_data = FALSE, y, X = NULL, params = list(),
vecchia_approx = NULL, vecchia_pred_type = NULL,
num_neighbors_pred = NULL, offset = NULL, fixed_effects = NULL,
likelihood_additional_param = NULL)
```

Arguments

`likelihood` A `string` specifying the likelihood function (distribution) of the response variable. The currently supported likelihoods, grouped by response type, are:

Response type	Support	Likelihoods
Continuous	y in (-inf, inf)	"gaussian", "t", "t_fix_df", "quantile_regressi
Positive continuous	y in (0, inf)	"gamma", "gamma_varying_shape", "lognorm
Non-negative continuous / semicontinuous	y in [0, inf) with point mass at 0	"tweedie", "tweedie_fixed_p", "tweedie_joint
Count	y in {0, 1, 2, ...}	"poisson", "negative_binomial", "negative_bi
Binary	y in {0, 1}	"bernoulli_logit", "bernoulli_probit"
Proportion / fractional / bounded	y in [0, 1]	"quasi_bernoulli_logit", "quasi_bernoulli_pro

The table lists the main likelihood names. Aliases, heteroscedastic and varying-shape variants, and the "_regression" variants of the two-part likelihoods are documented in the detailed descriptions below.

Continuous response: y in (-inf, inf)

- "gaussian": Gaussian likelihood
- "t": t-distribution (e.g., for robust regression). The default approximation is Fisher-Laplace, i.e., the Fisher information is used instead of the observed Hessian in the Laplace approximation of the marginal likelihood.
- "t_fix_df": t-distribution with the degrees-of-freedom (df) held fixed and not estimated
  - The degrees-of-freedom (df) can be set via the `likelihood_additional_param` parameter. The default is df = 2
- "quantile_regression" / "asymmetric_laplace" : an asymmetric Laplace likelihood for quantile regression. Both names are accepted, as is the alias "quantile"
  - The quantile must be supplied through the `likelihood_additional_param` parameter and must be strictly between 0 and 1

  The default approximation is Fisher-Laplace, i.e., the Fisher information is used instead of the observed Hessian in the Laplace approximation of

the marginal likelihood. The triangular-kernel-curvature (TKC) approximation can be enabled by appending "_triangular_kernel_curvature" or the shorthand "_tkc", for example, "quantile_regression_tkc" or "asymmetric_laplace_triangular_kernel_curvature".

  - experimental: the log-likelihood is not differentiable at the kinks where 'y_i' equals the location parameter, and the quasi-Newton mode finding can therefore stall at a point that is not the exact posterior mode. Appending "_ssn_alm", for example "quantile_regression_ssn_alm", enables an exact check of the (non-smooth) optimality conditions after the mode finding and, if the check fails, a refinement of the mode with a semismooth Newton method applied to the subproblems of an augmented Lagrangian method. This makes the approximate marginal likelihood a well-defined function of the parameters, i.e., independent of the path taken by the optimizer and of the number of threads, at the price of additional linear solves during mode finding. This matters for standard errors and for likelihood-based model comparison, but it does not necessarily improve predictive accuracy; increasing `max_num_restarts_lbfgs` can be a cheaper way of avoiding poor optima. Appending "_admm_ssn_alm" instead warm starts the refinement with an ADMM phase, which typically halves its cost when `matrix_inversion_method = "cholesky"` and is not recommended for iterative methods

- "gaussian_heteroscedastic": Gaussian likelihood where the mean is related to fixed and random effects and the log-error variance is related to fixed effects only (linear predictor or GPBoost algorithm). The estimated coefficients of the log-variance model are returned alongside the mean-model coefficients (with the suffix "_scale"). Fisher-Laplace is the default and currently the only implemented approximation.
- "gaussian_heteroscedastic_fixed_and_random": Gaussian likelihood where both the mean and the variance are related to fixed and random effects. This is currently only implemented for GPs with a 'vecchia' approximation. Fisher-Laplace is the default and currently the only implemented approximation. By default, Cholesky factorizations are used. If init_cov_pars is not given, the initial values are obtained from a homoscedastic Gaussian process model (unless some weights are zero).

Positive continuous response: y in (0, inf)

- "gamma": Gamma likelihood with a log link function
- "gamma_varying_shape": As "gamma", but the gamma shape varies across observations and is modeled by an additional fixed-effects-only predictor: log(shape) = F_s(X) (F_s(X) = linear predictor or the GPBoost algorithm), while the log mean log(mu) = F(X) + Zb is related to both fixed and random effects. Note that the shape also governs the dispersion, var(y) = mu^2 / shape. The estimated coefficients of the log-shape model are returned alongside the mean-model coefficients (with the suffix "_shape"). See also the corresponding two-part likelihoods "hurdle_gamma_varying_shape" and "hurdle_regression_gamma_varying_shape" below
- "lognormal": Log-normal likelihood with a log link function
- "gpd": Generalized Pareto likelihood for finite positive responses. The log scale parameter equals the latent predictor eta (sum of fixed and random

effects), sigma = exp(eta), and the estimated auxiliary parameter is shape > -0.5

- "egpd_power": Naveau power-carrier extended generalized Pareto likelihood with auxiliary parameters shape and kappa
- "egpd_power_mixture": Naveau ordered power-mixture carrier with auxiliary parameters shape, kappa1, delta_kappa, and p
- "egpd_beta": Naveau beta-carrier extended generalized Pareto likelihood with auxiliary parameters shape and delta
- "egpd_power_beta": Naveau power-beta carrier with auxiliary parameters shape, delta, and kappa

Non-negative continuous / semicontinuous response: y in [0, inf) with point mass at 0

- "tweedie": Compound Poisson–Gamma Tweedie likelihood with a log link, variance phi * mu^p, and 1.01 < p < 1.99. The dispersion phi and power p are estimated
- "tweedie_fixed_p": The same Tweedie likelihood with p fixed through `likelihood_additional_pa` only phi is estimated. The fixed power must satisfy 1.01 < p < 1.99
- "tweedie_joint", "tweedie_joint_fixed_p": The same Tweedie model, but the observed number of events (e.g., claims) N, given in the first column of `additional_likelihood_data`, is used jointly with the aggregate response y. The likelihood is the joint density of (y, N) of the compound Poisson–Gamma representation, N ~ Poisson(mu^(2-p) / (phi * (2-p))) and y | N = n ~ Gamma(n * (2-p) / (p-1), scale = phi * (p-1) * mu^(p-1)), see Jorgensen and de Souza (1994, Scandinavian Actuarial Journal). Given phi and p, the mean model is the same as for "tweedie"; N adds information for estimating phi and p. N must be a non-negative integer that is 0 if and only if y is 0. Predictions are the same as for "tweedie", and the "test_neg_log_likelihood" metric uses the marginal density of y since N is not known for new data
- "tweedie_varying_dispersion", "tweedie_varying_dispersion_fixed_p": As "tweedie" and "tweedie_fixed_p", but the dispersion phi varies across observations: log(phi) = F_d(X) is related to fixed effects only (linear predictor or GPBoost algorithm), while log(mu) = F(X) + Zb is related to both fixed and random effects. The power p is then the only (auxiliary) parameter. The estimated coefficients of the log-dispersion model are returned alongside the mean-model coefficients (with the suffix "_dispersion")
- "tweedie_joint_varying_dispersion", "tweedie_joint_varying_dispersion_fixed_p": The joint (y, N) likelihood of "tweedie_joint" with the varying dispersion of "tweedie_varying_dispersion"
- "hurdle_<base>": Two-part likelihoods for non-negative response variables with an excess probability 'p0' of exact zeros. They combine a point mass 'p0' at zero with a base distribution with support 'y > 0' for the remaining mass '1 - p0'. The fixed effects 'F(X)' and random effects 'Zb' enter only through the base component: 'exp(F(X) + Zb)' is its mean or scale parameter, not the unconditional response mean, which is E(y) = (1 - p0) * base_mean. The structural-zero probability 'p0' is estimated

jointly with the base auxiliary parameters. Currently supported bases: "hurdle_gamma", "hurdle_lognormal", and the extreme-value bases "hurdle_gpd", "hurdle_egpd_power", "hurdle_egpd_power_mixture", "hurdle_egpd_beta", "hurdle_egpd_power_beta". For all these bases, the "zero_inflated" prefix is accepted as an alias, e.g. "zero_inflated_gamma" maps to "hurdle_gamma". See "zero_inflated_<base>" under count responses for the corresponding count-data family

- "hurdle_regression_<base>": As "hurdle_<base>", but the structural-zero probability is modeled as a logistic regression on the covariates instead of as a constant (e.g. "hurdle_regression_gamma", "hurdle_regression_lognormal", "hurdle_regression_gpd", "hurdle_regression_egpd_power", "hurdle_regression_egpd_power_mixtu "hurdle_regression_egpd_beta", "hurdle_regression_egpd_power_beta"). The zero probability is then pi_i = 1 / (1 + exp(-x_i'alpha)), modeled through a second fixed-effects-only predictor that reuses the same design matrix X as the response model (the response predictor carries the random effects, the zero predictor does not). The estimated zero-model coefficients alpha are returned by get_coef() alongside the response-model coefficients, with the suffix "_zero"
- "hurdle_gamma_varying_shape", "hurdle_regression_gamma_varying_shape": As "hurdle_gamma" and "hurdle_regression_gamma", but with the observation-specific gamma shape of "gamma_varying_shape". For "hurdle_regression_gamma_varying_shape" there are three predictors: the response mean, the structural-zero logit (coefficients with the suffix "_zero"), and log(shape)
- "zero_censored_power_transformed_normal": Likelihood of a censored and power-transformed normal variable for modeling data with a point mass at 0 and a continuous distribution for y > 0. The model used is Y = max(0,X)^lambda, X ~ N(mu, sigma^2), where mu = F(X) + Zb, and sigma and lambda are (auxiliary) parameters that are estimated. For more details on this model, see Sigrist et al. (2012, AOAS) "A dynamic nonstationary spatio-temporal model for short term prediction of precipitation"
- "zero_censored_power_transformed_normal_heteroscedastic": As "zero_censored_power_transform but the standard deviation sigma of the latent normal variable varies across observations: log(sigma) = F_2(X) is related to fixed effects only (linear predictor or GPBoost algorithm), while mu = F(X) + Zb is related to both fixed and random effects. lambda is then the only (auxiliary) parameter that is estimated
- "zero_censored_shifted_gamma": Zero-censored shifted gamma likelihood for modeling data with a point mass at 0 and a continuous distribution for y > 0. The model used is Y = max(Z - xi, 0), where Z follows a gamma distribution with mean mu = exp(F(X) + Zb) and shape k. The shape k and shift xi are (auxiliary) parameters that are estimated. This is the version of "zero_one_censored_shifted_gamma" (Sigrist and Stahel, 2011) without the upper censoring at 1
- "zero_censored_shifted_gamma_varying_shape": As "zero_censored_shifted_gamma", but the shape k varies across observations: log(k) = F_s(X) is related to fixed effects only (covariates and / or the GPBoost tree-boosting algorithm; no random effects / GPs for the shape), while log(mu) = F(X) + Zb is related to both fixed and random effects. The shift xi is then the only (auxiliary)

parameter that is estimated. The estimated coefficients of the log-shape model are returned alongside the mean-model coefficients (with the suffix "_shape")

Count response: y in {0, 1, 2, ...}

- "poisson": Poisson likelihood with a log link function
- "negative_binomial": negative binomial likelihood with a log link function (aka "nbinom2", "negative_binomial_2"). The variance is mu * (mu + r) / r, mu = mean, r = shape, with this parametrization
- "negative_binomial_1": Negative binomial 1 (aka "nbinom1") likelihood with a log link function. The variance is mu * (1 + phi), mu = mean, phi = dispersion, with this parametrization
- "zero_inflated_<base>": Two-part count likelihoods that combine a point mass 'p0' at zero with a count base distribution for the remaining mass '1 - p0' (the base can itself generate additional zeros). As for the hurdle likelihoods, 'exp(F(X) + Zb)' is the mean of the (non-structural) base component - not the unconditional response mean - so E(y) = (1 - p0) * base_mean, and 'p0' is estimated jointly with the base auxiliary parameters. Currently supported bases: "zero_inflated_poisson", "zero_inflated_negative_binomial" (aux: shape; aka "zero_inflated_nbinom2"), "zero_inflated_negative_binomial_1" (aux: dispersion; aka "zero_inflated_nbinom1")
- "zero_inflated_regression_<base>": As "zero_inflated_<base>", but with the logistic structural-zero model described under "hurdle_regression_<base>" above: "zero_inflated_regression_poisson", "zero_inflated_regression_negative_binomial", and "zero_inflated_regression_negative_binomial_1"

Binary response: y in {0, 1}

- "bernoulli_logit": Bernoulli likelihood with a logit link function for binary classification. Aliases: "binary", "binary_logit"
- "bernoulli_probit": Bernoulli likelihood with a probit link function for binary classification. Aliases: "binary_probit"

Proportion / fractional / bounded response: y in [0, 1]

- "quasi_bernoulli_logit": quasi-Bernoulli likelihood with a logit link function for y in [0,1]. Aliases: "quasi_binary", "quasi_binary_logit"
- "quasi_bernoulli_probit": quasi-Bernoulli likelihood with a probit link function for y in [0,1]. Aliases: "quasi_binary_probit"
- "binomial_logit": Binomial likelihood with a logit link function. The response variable `y` needs to contain proportions of successes / trials, and the `weights` parameter needs to contain the numbers of trials. Aliases: "binomial"
- "binomial_probit": Binomial likelihood with a probit link function. The response variable `y` needs to contain proportions of successes / trials, and the `weights` parameter needs to contain the numbers of trials
- "beta_binomial": Beta-binomial likelihood with a logit link function. The response variable `y` needs to contain proportions of successes / trials, and the `weights` parameter needs to contain the numbers of trials. Aliases: "betabinomial", "beta-binomial"

- "beta" : Beta likelihood with a logit link function (parametrization of Ferrari and Cribari-Neto, 2004). The response must lie strictly in (0,1)
- "zoctn": Zero-one censored transformed normal likelihood for modeling data in [0,1] with point masses at 0 and 1 and a continuous distribution on (0,1). The model used is T ~ N(mu, sigma^2), W = max(min(T,1),0), and Y = g(W), where g(x) = expit(a + b * logit(x)) for x in (0,1), mu = F(X) + Z_RE u, u denotes the random effects, Z_RE is their design matrix, and sigma, a, and b are (auxiliary) parameters that are estimated. For more details on this model, see Qiang and Sigrist (2026)
- "zero_one_censored_transformed_beta": Zero-one censored transformed beta likelihood for modeling data in [0,1] with point masses at 0 and 1 and a continuous distribution on (0,1). If T follows a beta distribution with mean mu = expit(F(X) + Zb) and precision phi, the observed response is obtained by applying the linear transformation Y = (1 + 2u) * T - u and censoring the result to [0,1]. The precision phi and shift u are (auxiliary) parameters that are estimated. For more details on this model, see Kosmidis and Zeileis (2025)
- "zero_one_censored_shifted_gamma": Zero-one censored shifted gamma likelihood for modeling data in [0,1] with point masses at 0 and 1 and a continuous distribution on (0,1). The model used is Y = min(max(Z - xi, 0), 1), where Z follows a gamma distribution with mean mu = exp(F(X) + Zb) and shape k. The shape k and shift xi are (auxiliary) parameters that are estimated. For more details on this model, see Sigrist and Stahel (2011)

Notes

- The first lines in the likelihoods source file contain additional comments on the specific parametrizations used
- Other likelihoods can be implemented upon request

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

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

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

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

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

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

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

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

- "matern": Matern covariance function with the smoothness specified by the `cov_fct_shape` parameter (using the parametrization of Rasmussen and Williams, 2006)
- "matern_estimate_shape": same as "matern" but the smoothness parameter is also estimated
- "matern_space_time": Spatio-temporal Matern covariance function with different range parameters for space and time. Note that the first column in `gp_coords` must correspond to the time dimension
- "space_time_gneiting": Spatio-temporal covariance function given in Eq. (16) of Gneiting (2002). Note that the first column in `gp_coords` must correspond to the time dimension. This covariance has seven parameters (in the following order: sigma2, a, c, alpha, nu, beta, delta) which are all estimated by default. You can disable the estimation of some of these parameter using the 'estimate_cov_par_index' argument of the `params` argument in either the `fit` function of a `gp_model` object or the `set_optim_params` function prior to estimation.
- "matern_ard": anisotropic Matern covariance function with Automatic Relevance Determination (ARD), i.e., with a different range parameter for every coordinate of `gp_coords`
- "matern_ard_estimate_shape": same as "matern_ard" but the smoothness parameter is also estimated
- "exponential": Exponential covariance function (using the parametrization of Diggle and Ribeiro, 2007)
- "gaussian": Gaussian, aka squared exponential, covariance function (using the parametrization of Diggle and Ribeiro, 2007)
- "gaussian_ard": anisotropic Gaussian, aka squared exponential, covariance function with Automatic Relevance Determination (ARD), i.e., with a different range parameter for every coordinate of `gp_coords`
- "powered_exponential": powered exponential covariance function with the exponent specified by the `cov_fct_shape` parameter (using the parametrization of Diggle and Ribeiro, 2007)
- "wendland": Compactly supported Wendland covariance function (using the parametrization of Bevilacqua et al., 2019, AOS)
- "linear": linear covariance function. This corresponds to a Bayesian linear regression model with a Gaussian prior on the coefficients with a constant variance diagonal prior covariance, and the prior variance is estimated using empirical Bayes.

- "hurst": Hurst covariance function cov(s, s') = (sigma2 / 2) * (||s||^(2H) + ||s'||^(2H) - ||s - s'||^(2H)), 0 < H < 1. For H = 0.5, this corresponds to Brownian motion in one dimension (-> see the 'estimate_cov_par_index' argument). This is the covariance for `cov_fct_order` = 1 (default). For `cov_fct_order` = 2, the second-order Hurst covariance cov(s, s') = sigma2 / (2 (2H - 1)) * (||s - s'||^(2H) - ||s||^(2H) - ||s'||^(2H) + 2H (s^T s') (||s||^(2H-2) + ||s'||^(2H-2))), 1 < H < 2, is used. In one dimension and for s, s' >= 0, H = 1.5 corresponds to an integrated Wiener process. See `cov_fct_order` for more details
- "hurst_ard": Hurst covariance function with with Automatic Relevance Determination (ARD), i.e., with a different range parameter for every coordinate of "gp_coords" except for the first coordinate which has a range parameter of 1 due to identifiability with the marginal variance: $cov(s,s') = (\sigma^2/2)\left[\left(s_1^2+\sum_{k=2}^d (s_k/l_k)^2\right)^H + \left(s'^2_1+\sum_{k=2}^d (s'_k/l_k)^2\right)^H - \left((s_1-s'_1)^2+\sum_{k=2}^d((s_k-s'_k$
  For `cov_fct_order` = 2, the second-order Hurst covariance (see "hurst") is applied to the scaled coordinates (s_1, s_2/l_2, ..., s_d/l_d)
- "ar1_mf_<base>": two-level autoregressive multifidelity covariance defined by $f_H(x) = \rho f_L(x) + \delta(x)$, where f_L and δ are independent Gaussian processes using the same base covariance type but separate parameter vectors.
  - For example, use "ar1_mf_matern", "ar1_mf_matern_ard", or "ar1_mf_matern_estimate_shape"
  - The last column of `gp_coords` must be 0 for low fidelity or 1 for high fidelity. All preceding columns are input coordinates for the GPs.
  - Covariance parameters are ordered as [low-fidelity base parameters, discrepancy base parameters, rho]. The two base blocks use the ordinary parameter ordering of the base covariance, and rho is unrestricted and can be negative.
  - Any supported base covariance except "wendland" can be used. Correlation tapering and Gaussian-process random coefficients are currently not supported for this model.

`cov_fct_shape` A `numeric` specifying the shape parameter of the covariance function (e.g., smoothness parameter for Matern and Wendland covariance) This parameter is irrelevant for some covariance functions such as the exponential or Gaussian

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

- "none": No approximation
- "vecchia": Vecchia approximation; see Sigrist (2022, JMLR) for more details. For "matern_space_time" and the anisotropic ARD covariance functions ("matern_ard", "gaussian_ard", "matern_ard_estimate_shape"), neighbors are selected by absolute correlation, i.e., using distances in coordinates scaled by the range parameters, and they are redetermined during parameter estimation. For "space_time_gneiting" and "ar1_mf_<base>", neighbors are also selected by absolute correlation by default. Use gp_approx = "vecchia_euclidean" for Euclidean-distance selection.
- "full_scale_vecchia": Vecchia-inducing points full-scale (VIF) approximation; see Gyger, Furrer, and Sigrist (2026, JMLR) for more details

- "tapering": The covariance function is multiplied by a compactly supported Wendland correlation function
- "fitc": Fully Independent Training Conditional approximation aka modified predictive process approximation
- "full_scale_tapering": Full-scale approximation combining an inducing point / predictive process approximation with tapering on the residual process; see Gyger, Furrer, and Sigrist (2026, JUQ) for more details
- "vecchia_latent": similar as "vecchia" but a Vecchia approximation is applied to the latent Gaussian process for likelihood == "gaussian". For likelihood != "gaussian", "vecchia" and "vecchia_latent" are equivalent

`num_parallel_threads`
: An `integer` specifying the number of parallel threads for OMP. If num_parallel_threads = NULL, the default number of threads of the session is used, see `gpb.get.default.num.threads`. Initially, this is the number of physical performance cores, which is determined automatically as follows. On CPUs whose cores have different speeds (e.g., the performance and efficiency cores of recent Intel CPUs or Apple silicon), the fastest cores are counted, and hyperthreads are counted only once, since slow threads can slow down an entire model. The next fastest cores are added if the fastest ones alone would leave only a single thread. On Linux, a CPU bandwidth limit of a control group (e.g., of a container) is respected as well. If the environment variable OMP_NUM_THREADS is set, it takes precedence over both the topology of the CPU and the limit of the control group, and the number of threads that OpenMP is configured to use is taken as it is if the cores of the CPU cannot be determined. In either case, the limits of the OpenMP runtime itself still apply, in particular the limit of the contention group (OMP_THREAD_LIMIT). For ordinary use, leave num_parallel_threads unspecified to use this default. The default of the session can be changed with `gpb.tune.num.threads`, which benchmarks different numbers of threads, or with `gpb.set.default.num.threads`. Setting num_parallel_threads = 1 disables the OpenMP parallelization of the model and can substantially increase the runtime. A single thread should be chosen deliberately, e.g., to distribute the resources among concurrent model fits or for a specific test; it is not required for correctness or reproducibility

`GPU_use`
: A `boolean`. If TRUE, GPU acceleration will be used if supported

`matrix_inversion_method`
: A `string` specifying the method used for inverting covariance matrices. Available options:

- "default": iterative methods where possible, otherwise Cholesky factorization. Cholesky factorization is also used for likelihood = "gaussian_heteroscedastic_fixed_and_rando
- "cholesky": Cholesky factorization
- "iterative": iterative methods. A combination of the conjugate gradient, the Lanczos algorithm, and other methods.
  This is currently only supported for the following cases:
  - grouped random effects with more than one level
  - likelihood != "gaussian" and gp_approx == "vecchia" (non-Gaussian likelihoods with a Vecchia-Laplace approximation)

- likelihood != "gaussian" and gp_approx == "full_scale_vecchia" (non-Gaussian likelihoods with a VIF approximation)
- likelihood == "gaussian" and gp_approx == "full_scale_tapering" (Gaussian likelihood with a full-scale tapering approximation)

`weights` A `vector` with sample weights. For a Gaussian likelihood, the error variance ("nugget") for observation `i` is divided by `weights[i]`. For non-Gaussian likelihoods, the conditional log-likelihood contribution of observation `i` is multiplied by `weights[i]`. Consequently, weights affect the estimation of both random and fixed effects. Note that a Gaussian likelihood is calculated via the Laplace approximation when `gp_approx = "vecchia_latent"`, when `likelihood = "gaussian_latent"`, and when grouped random effects are combined with a Vecchia-approximated Gaussian process. In these cases, the weights act as for a non-Gaussian likelihood, i.e., the Gaussian log-likelihood contribution of observation `i` is multiplied by `weights[i]` instead of the error variance being divided by it.

`additional_likelihood_data`
A `vector` or `matrix` with observation-level data that some likelihoods require in addition to the response variable `y` (one row per data point). Currently, this is only used by the joint Tweedie likelihoods ("tweedie_joint" and its variants), for which its first column contains the observed number of events (e.g., claims) N. N must be a non-negative integer that is 0 if and only if y is 0

`likelihood_learning_rate`
A `numeric` with a learning rate for the likelihood for generalized Bayesian inference (only non-Gaussian likelihoods)

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

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

`num_neighbors` An `integer` specifying the number of neighbors for the Vecchia and VIF approximations. Internal default values if NULL:

- 20 for gp_approx = "vecchia"
- 30 for gp_approx = "full_scale_vecchia"

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

`vecchia_ordering`
A `string` specifying the ordering used in the Vecchia approximation. Available options:

- "none": the default ordering in the data is used
- "random": a random ordering
- "time": ordering accorrding to time (only for space-time models)

- "time_random_space": ordering according to time and randomly for all spatial points with the same time points (only for space-time models)

`ind_points_selection`

A `string` specifying the method for choosing inducing points Available options:

- "kmeans++: the k-means++ algorithm
- "cover_tree": the cover tree algorithm
- "random": random selection from data points

`num_ind_points` An `integer` specifying the number of inducing points / knots for FITC, full_scale_tapering, and VIF approximations. Internal default values if NULL:

- 500 for gp_approx = "FITC" and gp_approx = "full_scale_tapering"
- 200 for gp_approx = "full_scale_vecchia"

`cover_tree_radius`

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

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

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

`fidelity_specific_mean`

A `logical`. For an `ar1_mf_<base>` covariance, whether marginal mean models are fitted independently for low and high fidelity. If `TRUE` (the default), linear covariates supplied to `fit()` are internally expanded into separate low- and high-fidelity coefficient blocks, and the fidelity indicator is automatically added as a feature when the model is used in the GPBoost algorithm. Ignored for other covariance functions.

`cov_fct_order` An `integer` specifying the order m of the "hurst" and "hurst_ard" covariance functions (also when used as base covariance in "ar1_mf_hurst" and "ar1_mf_hurst_ard"). Currently, the orders 1 (default) and 2 are supported. The Hurst exponent H of the order m satisfies m - 1 < H < m, and H = m - 0.5 is used as initial value. Order 1 is a fractional Brownian motion / field anchored at the origin (b(0) = 0), and order 2 is a second-order Hurst process / field anchored at the origin such that both the process and its gradient are zero there (b(0) = 0, grad b(0) = 0). The origin of the coordinates thus is a part of the model, and the coordinates are not centered internally. The order is not estimated. It can be chosen, e.g., by comparing the marginal likelihoods or by cross-validation.

- Continuous-time RW1 prior: use `cov_function = "hurst"`, `cov_fct_order = 1`, and fix H = 0.5. This is a Brownian motion with b(0) = 0 and b'(t) = sqrt(q) W'(t), where W' denotes white noise and q = sigma2
- Continuous-time RW2 prior: use `cov_function = "hurst"`, `cov_fct_order = 2`, and fix H = 1.5. This is an integrated Wiener process with b(0) = b'(0) = 0 and b"(t) = sqrt(q) W'(t), where q = 3 * sigma2

For these priors, use a one-dimensional time coordinate t >= 0 whose origin t = 0 is the desired anchor (shift the time points if necessary). H can be fixed using the 'init_cov_pars' and 'estimate_cov_par_index' arguments of the `params`

argument, see the examples of `fitGPModel`. If the polynomial null space of the intrinsic model should not be penalized, include an intercept (RW1) or an intercept and a linear time trend (RW2) as fixed effects. Note that the order 2 model with H = 1.5 is the continuous-time RW2 prior evaluated at the time points and not the usual discrete intrinsic RW2 prior on a lattice: for equally spaced time points, adjacent second differences are correlated (correlation 1/4) whereas they are independent for the discrete RW2

`free_raw_data` A `boolean`. If TRUE, the data (groups, coordinates, covariate data for random coefficients, cluster IDs, weights, and additional likelihood data) is freed in R after initialization

`y` A `vector` with response variable data

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

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

- trace: `boolean` (default = FALSE). If TRUE, information on the progress of the parameter optimization is printed
- init_cov_pars: `vector` with `numeric` elements (default = NULL). Initial values for covariance parameters of Gaussian process and random effects (can be NULL). The order is same as the order of the parameters in the summary function: first is the error variance (only for "gaussian" likelihood), next follow the variances of the grouped random effects (if there are any, in the order provided in 'group_data'), and then follow the marginal variance and the ranges of the Gaussian process. If there are multiple Gaussian processes, then the variances and ranges follow alternatingly. If 'init_cov_pars = NULL', an internal choice is used that depends on the likelihood and the random effects type and covariance function. If you select the option 'trace = TRUE' in the 'params' argument, you will see the first initial covariance parameters in iteration 0.
- init_coef: `vector` with `numeric` elements (default = NULL). Initial values for the regression coefficients (if there are any, can be NULL)
- init_aux_pars: `vector` with `numeric` elements (default = NULL). Initial values for additional parameters for non-Gaussian likelihoods (e.g., shape parameter of a gamma or negative_binomial likelihood)
- init_coef_aux_pars_from_iid_model: `boolean` (default = TRUE). If TRUE, regression coefficients and auxiliary parameters are initialized from an iid model (only for models with a linear regression term). This option is ignored if init_coef is provided. If init_aux_pars is provided but init_coef is not, only regression coefficients are initialized from an iid model. For likelihood = "gaussian_heteroscedastic_fixed_and_random", if init_cov_pars is not provided, the initial covariance parameters are obtained from a homoscedastic Gaussian process model, and the regression coefficients are then initialized from this model instead: the coefficients of the mean from its coefficients and the intercept of the log-error variance from the logarithm of its error variance.
- estimate_cov_par_index: `vector` with `integer` (default = -1). This allows for disabling the estimation of some (or all) covariance parameters. If 'esti-

mate_cov_par_index' = -1, all covariance parameters are estimated. If estimate_cov_par_index != -1, this should be a vector with length equal to the number of covariance parameters, and estimate_cov_par_index[i] should be of bool type indicating whether parameter number i is estimated or not. For instance, estimate_cov_par_index = c(1,1,0) means that the first two covariance parameters are estimated and the last one not. Parameters that are not estimated are kept at their initial values (see 'init_cov_pars'). They are treated as known constants when calculating the standard errors of the estimated parameters, and their own standard errors are NaN.

- estimate_aux_pars: `boolean` (default = TRUE). If TRUE, additional parameters for non-Gaussian likelihoods are also estimated (e.g., shape parameter of a gamma or negative_binomial likelihood)
- optimizer_cov: `string` (default = "lbfgs"). Optimizer used for estimating covariance parameters. Options: "lbfgs", "gradient_descent", "fisher_scoring", "newton", "nelder_mead". If there are additional auxiliary parameters for non-Gaussian likelihoods, 'optimizer_cov' is also used for those
- optimizer_coef: `string` (default = "wls" for Gaussian likelihoods and "lbfgs" for other likelihoods). Optimizer used for estimating linear regression coefficients, if there are any (for the GPBoost algorithm there are usually none). Options: "gradient_descent", "lbfgs", "wls", "nelder_mead". Gradient descent steps are done simultaneously with gradient descent steps for the covariance parameters. "wls" refers to doing coordinate descent for the regression coefficients using weighted least squares. If 'optimizer_cov' is set to "nelder_mead" or "lbfgs", 'optimizer_coef' is automatically also set to the same value.
- maxit: `integer` (default = 1000). Maximal number of iterations for optimization algorithm. If maxit = -999, internal default values are used
- delta_rel_conv: `numeric` (default = 1E-6 except for "nelder_mead" for which the default is 1E-8). Convergence tolerance. The algorithm stops if the relative change in either the (approximate) log-likelihood or the parameters is below this value. If delta_rel_conv = -999, internal default values are used
- cg_max_num_it: `integer` (default = 1000). Maximal number of iterations for conjugate gradient algorithms. If cg_max_num_it = -999, internal default values are used
- cg_max_num_it_tridiag: `integer` (default = 1000). Maximal number of iterations for conjugate gradient algorithm when being run as Lanczos algorithm for tridiagonalization. If cg_max_num_it_tridiag = -999, internal default values are used
- cg_delta_conv: `numeric` (default = 1E-2). Tolerance level for L2 norm of residuals for checking convergence in conjugate gradient algorithm when being used for parameter estimation. This is the tolerance of the "absolute" criterion only, the "relative" criterion is controlled by cg_rel_tol and cg_abs_tol. If cg_delta_conv = -999, internal default values are used
- cg_convergence_criterion: `string` (default = "absolute"). Stopping rule of conjugate gradient algorithms.
  - "absolute": stop when ||r||_2 < cg_delta_conv

  - "relative": stop when ||r||_2 <= max(cg_abs_tol, cg_rel_tol * ||b||_2), where b is the right-hand side of the linear system. The cg_abs_tol floor makes this robust for right-hand sides that are zero or very small
- cg_rel_tol: `numeric` (default = 1E-6). Relative tolerance of the "relative" stopping rule. If cg_rel_tol = -999, internal default values are used
- cg_abs_tol: `numeric` (default = 1E-8). Absolute tolerance floor of the "relative" stopping rule, which makes it robust for right-hand sides that are zero or very small. This is independent of cg_delta_conv, so that the two criteria can be varied separately. If cg_abs_tol = -999, internal default values are used
- cg_multi_rhs_convergence: `string` (default = "average"). How the stopping rule is aggregated over the columns of a linear system with several right-hand sides, such as the stochastic Lanczos quadrature. For cg_convergence_criterion = "relative", every right-hand side is normalized by its own norm, i.e. q_j = ||r_j||_2 / max(cg_abs_tol, cg_rel_tol * ||b_j||_2):
  - "average": stop when mean(q_j) <= 1
  - "max": stop when max(q_j) <= 1
  - "per_rhs": stop iterating on column j individually as soon as q_j <= 1

  For cg_convergence_criterion = "absolute" there is no per-column normalization:
  - "average": stop when mean(||r_j||_2) < cg_delta_conv. This is the historic behavior and the default
  - "max": stop when ||r_j||_2 < cg_delta_conv for all j
  - "per_rhs": stop iterating on column j as soon as ||r_j||_2 < cg_delta_conv
- num_rand_vec_trace: `integer` (default = 50). Number of random vectors (e.g., Rademacher) for stochastic approximation of the trace of a matrix. If num_rand_vec_trace = -999, internal default values are used
- reuse_rand_vec_trace: `boolean` (default = TRUE). If true, random vectors (e.g., Rademacher) for stochastic approximations of the trace of a matrix are sampled only once at the beginning of the parameter estimation and reused in later trace approximations. Otherwise they are sampled every time a trace is calculated
- seed_rand_vec_trace: `integer` (default = 1). Seed number to generate random vectors (e.g., Rademacher)
- cg_preconditioner_type (`string`): Type of preconditioner used for conjugate gradient algorithms.
  - Options for grouped random effects:
    * "ssor" (= default): SSOR preconditioner
    * "incomplete_cholesky": zero fill-in incomplete Cholesky factorization
  - Options for likelihood != "gaussian" and gp_approx == "vecchia" or likelihood == "gaussian" and gp_approx == "vecchia_latent":
    * "vadu" (= default): (B^T * (D^-1 + W) * B) as preconditioner for inverting (B^T * D^-1 * B + W), where B^T * D^-1 * B approx= Sigma^-1

    * "fitc": FITC / modified predictive process preconditioner for inverting (B^-1 * D * B^-T + W^-1)
    * "pivoted_cholesky": (Lk * Lk^T + W^-1) as preconditioner for inverting (B^-1 * D * B^-T + W^-1), where Lk is a low-rank pivoted Cholesky approximation for Sigma and B^-1 * D * B^-T approx= Sigma
    * "incomplete_cholesky": zero fill-in incomplete (reverse) Cholesky factorization of (B^T * D^-1 * B + W) using the sparsity pattern of B^T * D^-1 * B approx= Sigma^-1
  - Options for likelihood != "gaussian" and gp_approx == "full_scale_vecchia":
    * "fitc" (= default): FITC / modified predictive process preconditioner
    * "vifdu": VIF with diagonal update preconditioner
  - Options for likelihood == "gaussian" and gp_approx == "full_scale_tapering":
    * "fitc" (= default): modified predictive process preconditioner
    * "none": no preconditioner
- fitc_piv_chol_preconditioner_rank (`integer`): Rank of the FITC and pivoted Cholesky decomposition preconditioners for iterative methods for Vecchia and VIF approximations (for full_scale_tapering, the same inducing points as in the approximation as used). If fitc_piv_chol_preconditioner_rank = -999, internal default values are used:
  - 200 for the FITC preconditioner
  - 50 for the pivoted Cholesky decomposition preconditioner
- convergence_criterion: `string` (default = "relative_change_in_log_likelihood", only relevant for "gradient_descent", "fisher_scoring", and "newton"). The convergence criterion used for terminating the optimization algorithm. Options: "relative_change_in_log_likelihood" or "relative_change_in_parameters". If convergence_criterion = "default", internal default values are used
- lr_cov: `numeric` (default = 0.1 for "gradient_descent" and 1. otherwise, relevant for "gradient_descent", "fisher_scoring", "newton", and "lbfgs"). Initial learning rate for covariance parameters if a gradient-based optimization method is used
  - If lr_cov = -999, internal default values are used (0.1 for "gradient_descent" and 1. otherwise)
  - If there are additional auxiliary parameters for non-Gaussian likelihoods, 'lr_cov' is also used for those
  - For "gradient_descent", "fisher_scoring", and "newton", this is the learning rate by which the search direction is multiplied. It can be decreased, and increased again, during the optimization
  - For "lbfgs" and "lbfgs_linesearch_nocedal_wright", this is not a learning rate by which every update is multiplied. It only determines the first trial step, which is the negative gradient divided by its norm and multiplied by 'lr_cov', i.e., a step of length 'lr_cov'. The subsequent steps are determined by the approximate Hessian of the L-BFGS algorithm and the line search. 'lr_cov' has no effect when the approximate Hessian of a previous call is reused (see 'reuse_learning_rates_gp_model' of the GPBoost algorithm)

- lr_coef: numeric (default = 0.1, only relevant for "gradient_descent", "fisher_scoring", and "newton"). Learning rate for fixed effect regression coefficients if gradient descent is used. If lr_coef = -999, internal default values are used
- use_nesterov_acc: boolean (default = TRUE, only relevant for "gradient_descent"). If TRUE Nesterov acceleration is used. This is used only for gradient descent
- nesterov_schedule_version: integer (Default = 0, only relevant for "gradient_descent"). Which version of Nesterov schedule should be used. If nesterov_schedule_version = -999, internal default values are used.
- acc_rate_coef: numeric (default = 0.5, only relevant for "gradient_descent"). Acceleration rate for regression coefficients (if there are any) for Nesterov acceleration. If acc_rate_coef = -999, internal default values are used
- acc_rate_cov: numeric (default = 0.5, only relevant for "gradient_descent"). Acceleration rate for covariance parameters for Nesterov acceleration. If acc_rate_cov = -999, internal default values are used
- momentum_offset: integer (Default = 2, only relevant for "gradient_descent"). Number of iterations for which no momentum is applied in the beginning. If momentum_offset = -999, internal default values are used.
- m_lbfgs: integer (Default = 6). Number of corrections to approximate the inverse Hessian matrix for the "lbfgs" optimizer. If m_lbfgs = -999, internal default values are used
- cold_restart_lbfgs: boolean (Default = TRUE). If TRUE, restarts of the "lbfgs" optimizers are "cold" restarts: the regression coefficients, the auxiliary parameters, and the modes of the Laplace approximations are reset to their initial values and only the covariance parameters are kept. Such restarts can escape a local optimum in which the optimizer is stuck. If FALSE, "warm" restarts are done: the optimization simply continues from the current parameters with a re-initialized approximate Hessian. Only relevant if max_num_restarts_lbfgs > 0. Note that every cold restart is a full optimization with up to maxit iterations, and the number of iterations reported is the sum over all restarts
- max_num_restarts_lbfgs: integer (Default = 0). Maximal number of restarts of the "lbfgs" optimizers after they have terminated. A restart continues from the current parameters with a re-initialized approximate Hessian. What else is reset depends on cold_restart_lbfgs. Restarts are only done as long as the negative log-likelihood still decreases by more than delta_rel_conv and as long as the total number of iterations is below maxit. This can help when the line search of "lbfgs" fails and the optimizer thus terminates prematurely without having converged. This can happen, e.g., for the "asymmetric_laplace" (= quantile regression) likelihood, whose approximate marginal likelihood is not smooth
- reset_modes_line_search_lbfgs: boolean (Default = FALSE). If TRUE, the mode finding of the Laplace approximation starts at the modes of the current parameters for every trial point of the line search of the "lbfgs" optimizer, and not at the modes of the previous trial point. Only relevant for non-Gaussian likelihoods and optimizer_cov = "lbfgs" or "lbfgs_linesearch_nocedal_wright
- delta_conv_mode_finding: numeric (Default = 1E-8, and 1E-10 for the likelihood "gaussian_heteroscedastic_fixed_and_random"). Convergence

tolerance in mode finding algorithm for Laplace approximation for non-Gaussian likelihoods. If delta_conv_mode_finding = -999, internal default values are used

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

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

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

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

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

`likelihood_additional_param`
A `numeric` specifying an additional parameter for the `likelihood` which cannot be estimated for this `likelihood` (e.g., degrees of freedom for `likelihood = "t_fix_df"`). This is not to be confused with any auxiliary parameters that can be estimated and accessed through the function `get_aux_pars` after estimation. Note that this `likelihood_additional_param` parameter is irrelevant for many likelihoods. If `likelihood_additional_param = NULL`, the following default and mandatory-value rules apply:

- df = 2 for likelihood = "t_fix_df"
- No default is used for likelihood = "asymmetric_laplace"; a quantile strictly between 0 and 1 is required
- No default is used for the Tweedie likelihoods with a fixed power ("tweedie_fixed_p" and the other "_fixed_p" variants); a power strictly between 1.01 and 1.99 is required

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")
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)
summary(gp_model)

#--------------------Gaussian process model----------------
gp_model <- fitGPModel(gp_coords = coords, cov_function = "matern", cov_fct_shape = 1.5,
                       likelihood="gaussian", y = y, X = X1)
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 = "matern", cov_fct_shape = 1.5,
                       gp_approx = "vecchia", num_neighbors = 20,
                       likelihood="gaussian", y = y)
summary(gp_model)

#--------------------Gaussian process model with random coefficients----------------
gp_model <- fitGPModel(gp_coords = coords, cov_function = "matern", cov_fct_shape = 1.5,
                       gp_rand_coef_data = X[,2], y=y,
                       likelihood = "gaussian")
summary(gp_model)

#--------------------Combine Gaussian process with grouped random effects----------------
gp_model <- fitGPModel(group_data = group_data,
                       gp_coords = coords, cov_function = "matern", cov_fct_shape = 1.5,
                       likelihood = "gaussian", y = y, X = X1)
summary(gp_model)

#--------------------Continuous-time RW2 prior for a time series----------------
# The prior is anchored at time t = 0, i.e., b(0) = b'(0) = 0
time <- seq(0, 1, length.out = 100)
```

```
y_time <- sin(2 * pi * time) + rnorm(100, sd = 0.2)
X_time <- cbind(Intercept = 1, trend = time) # unpenalized intercept and linear trend
# The covariance parameters are (error variance, GP variance, H), and H = 1.5 is not estimated
gp_model <- fitGPModel(gp_coords = time, cov_function = "hurst", cov_fct_order = 2,
                       likelihood = "gaussian", y = y_time, X = X_time,
                       params = list(init_cov_pars = c(0.1, 1, 1.5),
                                     estimate_cov_par_index = c(1, 1, 0)))
summary(gp_model)
# Continuous-time RW1 prior: cov_fct_order = 1, H fixed to 0.5, and only an intercept
```

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

Description

Get one attribute of a `gpb.Dataset`

Usage

```
getinfo(dataset, ...)

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

Arguments

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

Details

The `name` field can be one of the following:

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

Value

info data

info data

Examples

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

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

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

---

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

Description

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

Usage

```
get_aux_pars(gp_model, std_err = FALSE)
```

Arguments

gp_model	A GPModel
std_err	A boolean. If TRUE, (approximate) standard errors are calculated (= 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). For non-Gaussian likelihoods, no standard errors of the covariance and auxiliary parameters are calculated if the model is used in the GPBoost algorithm, since they require the fixed effects given by the tree ensemble

Author(s)

Fabio Sigrist

Examples

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

---

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

Description

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

Usage

```
## S3 method for class 'GPModel'
get_aux_pars(gp_model, std_err = FALSE)
```

Arguments

gp_model	A GPModel
std_err	A boolean. If TRUE, (approximate) standard errors are calculated (= 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). For non-Gaussian likelihoods, no standard errors of the covariance and auxiliary parameters are calculated if the model is used in the GPBoost algorithm, since they require the fixed effects given by the tree ensemble

Value

A GPModel

Author(s)

Fabio Sigrist

Examples

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

---

get_coef *Get (estimated) linear regression coefficients*

Description

Get (estimated) linear regression coefficients and standard errors (if std_err=TRUE)

Usage

```
get_coef(gp_model, std_err = FALSE)
```

Arguments

gp_model	A GPModel
std_err	A boolean. If TRUE, (approximate) standard errors are calculated (= 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). For non-Gaussian likelihoods, no standard errors of the covariance and auxiliary parameters are calculated if the model is used in the GPBoost algorithm, since they require the fixed effects given by the tree ensemble

Author(s)

Fabio Sigrist

Examples

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

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

Description

Get (estimated) linear regression coefficients and standard errors (if std_err=TRUE)

Usage

```
## S3 method for class 'GPModel'
get_coef(gp_model, std_err = FALSE)
```

Arguments

`gp_model`	A `GPModel`
`std_err`	A `boolean`. If TRUE, (approximate) standard errors are calculated (= 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). For non-Gaussian likelihoods, no standard errors of the covariance and auxiliary parameters are calculated if the model is used in the GPBoost algorithm, since they require the fixed effects given by the tree ensemble

Value

A `GPModel`

Author(s)

Fabio Sigrist

Examples

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

get_cov_pars *Get (estimated) covariance parameters*

Description

Get (estimated) covariance parameters and standard error (if std_err=TRUE)

Usage

```
get_cov_pars(gp_model, std_err = FALSE)
```

Arguments

gp_model	A GPModel
std_err	A boolean. If TRUE, (approximate) standard errors are calculated (= 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). For non-Gaussian likelihoods, no standard errors of the covariance and auxiliary parameters are calculated if the model is used in the GPBoost algorithm, since they require the fixed effects given by the tree ensemble

Author(s)

Fabio Sigrist

Examples

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

get_cov_pars.GPModel *Get (estimated) covariance parameters*

Description

Get (estimated) covariance parameters and standard errors (if std_err=TRUE)

Usage

```
## S3 method for class 'GPModel'
get_cov_pars(gp_model, std_err = FALSE)
```

Arguments

gp_model	A GPModel
std_err	A boolean. If TRUE, (approximate) standard errors are calculated (= 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). For non-Gaussian likelihoods, no standard errors of the covariance and auxiliary parameters are calculated if the model is used in the GPBoost algorithm, since they require the fixed effects given by the tree ensemble

Value

A GPModel

Author(s)

Fabio Sigrist

Examples

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

---

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

Description

Auxiliary function to create categorical variables for nested grouped random effects

Usage

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

Arguments

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

Value

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

Author(s)

Fabio Sigrist

Examples

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

---

gpb.convert_with_rules

Data preparator for GPBoost datasets with rules (integer)

Description

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

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

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

Usage

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

Arguments

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

Value

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

Examples

```
data(iris)

str(iris)

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

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

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

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

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

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

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

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

---

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

Description

Cross validation function for determining number of boosting iterations

Usage

```
gpb.cv(params = list(), data, gp_model = NULL, nrounds = 1000L,
  early_stopping_rounds = NULL, folds = NULL, nfold = 5L, metric = NULL,
```

```
verbose = 1L, use_gp_model_for_validation = TRUE,
fit_GP_cov_pars_OOS = FALSE, train_gp_model_cov_pars = TRUE,
label = NULL, weight = NULL, obj = NULL, eval = NULL, record = TRUE,
eval_freq = 1L, showsd = FALSE, stratified = TRUE, init_model = NULL,
colnames = NULL, categorical_feature = NULL, callbacks = list(),
reset_data = FALSE, delete_boosters_folds = FALSE, ...)
```

Arguments

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

- `learning_rate`: The learning rate, also called shrinkage or damping parameter (default = 0.1). An important tuning parameter for boosting. Lower values usually lead to higher predictive accuracy but more boosting iterations are needed
- `num_leaves`: Number of leaves in a tree. Tuning parameter for tree-boosting (default = 31)
- `max_depth`: Maximal depth of a tree. Tuning parameter for tree-boosting (default = no limit)
- `min_data_in_leaf`: Minimal number of samples per leaf. Tuning parameter for tree-boosting (default = 20)
- `lambda_l2`: L2 regularization (default = 0)
- `lambda_l1`: L1 regularization (default = 0)
- `max_bin`: Maximal number of bins that feature values will be bucketed in (default = 255)
- `line_search_step_length` (default = FALSE): If TRUE, a line search is done to find the optimal step length for every boosting update (see, e.g., Friedman 2001). This is then multiplied by the learning rate
- `train_gp_model_cov_pars` (default = TRUE): If TRUE, the covariance parameters of the Gaussian process are estimated in every boosting iterations, otherwise the gp_model parameters are not estimated. In the latter case, you need to either estimate them beforehand or provide values via the 'init_cov_pars' parameter when creating the gp_model
- `use_gp_model_for_validation` (default = TRUE): If TRUE, the Gaussian process is also used (in addition to the tree model) for calculating predictions on the validation data
- `leaves_newton_update` (default = FALSE): Set this to TRUE to do a Newton update step for the tree leaves after the gradient step. Applies only to Gaussian process boosting (GPBoost algorithm)
- num_threads: Number of threads. For the best speed, set this to the number of real CPU cores(`parallel::detectCores(logical = FALSE)`), not the number of threads (most CPU using hyper-threading to generate 2 threads per CPU core). For ordinary use, leave this unspecified to use the default number of threads. Setting num_threads = 1 disables the parallelization and can substantially increase the runtime. A single thread should be chosen deliberately, e.g., to distribute the resources among concurrent model fits or for a specific test; it is not required for correctness or reproducibility.

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.

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

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

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.

folds
: (list) This provides the 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-sized subsamples. This allows for doing nfold-CV

metric
: Evaluation metric to be monitored when doing CV and parameter tuning. Can be a character string or vector of character strings. If not NULL, the metric in params will be overridden.

  - Default ="test_neg_log_likelihood" if there is a GPModel
  - Non-exhaustive list of supported metrics: "test_neg_log_likelihood", "mse", "rmse", "mae", "crps_gaussian", "auc", "average_precision", "binary_logloss", "binary_error"
  - See the "metric" section of the parameter documentation for a complete list of valid metrics

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Value

a trained model gpb.CVBooster.

Early Stopping

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

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

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

Author(s)

Authors of the LightGBM R package, Fabio Sigrist

Examples

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

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

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

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

Description

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

Usage

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

Arguments

data	a matrix object, a dgCMatrix object or a character representing a filename
params	a list of parameters. See the "Dataset Parameters" section of the parameter documentation for a list of parameters and valid values.
reference	reference dataset. When GPBoost creates a Dataset, it does some preprocessing like binning continuous features into histograms. If you want to apply the same bin boundaries from an existing dataset to new data, pass that existing Dataset to this argument.
colnames	names of columns
categorical_feature	categorical features. This can either be a character vector of feature names or an integer vector with the indices of the features (e.g. c(1L, 10L) to say "the first and tenth columns").
free_raw_data	GPBoost constructs its data format, called a "Dataset", from tabular data. By default, this Dataset object on the R side does keep a copy of the raw data. If you set free_raw_data = TRUE, no copy of the raw data is kept (this reduces memory usage)
info	a list of information of the gpb.Dataset object
...	other information to pass to info or parameters pass to params

Value

constructed dataset

Examples

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

```
gpb.Dataset.construct(dtrain)
```

---

gpb.Dataset.construct *Construct Dataset explicitly*

Description

Construct Dataset explicitly

Usage

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

Arguments

dataset	Object of class gpb.Dataset

Value

constructed dataset

Examples

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

---

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

Description

Construct validation data according to training data

Usage

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

Arguments

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

Value

constructed dataset

Examples

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

---

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

Description

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

Usage

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

Arguments

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

Value

the dataset you passed in

Examples

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

gpb.Dataset.set.categorical

Set categorical feature of gpb.Dataset

Description

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

Usage

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

Arguments

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

Value

the dataset you passed in

Examples

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

gpb.Dataset.set.reference

Set reference of gpb.Dataset

Description

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

Usage

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

Arguments

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

Value

the dataset you passed in

Examples

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

---

gpb.dump *Dump GPBoost model to json*

Description

Dump GPBoost model to json

Usage

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

Arguments

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

Value

json format of model

Examples

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

```
params <- list(objective = "regression", metric = "l2")
valids <- list(test = dtest)
model <- gpb.train(
  params = params
  , data = dtrain
  , nrounds = 10L
  , valids = valids
  , min_data = 1L
  , learning_rate = 1.0
  , early_stopping_rounds = 5L
)
json_model <- gpb.dump(model)
```

`gpb.get.default.num.threads` *Get the number of threads used by models for which no number of threads is specified*

Description

Returns the number of threads that a `GPModel` uses when no number of threads has been specified for it via the `num_parallel_threads` argument. This is the number of threads that has been set for the session, either by `gpb.tune.num.threads` or by `gpb.set.default.num.threads`, and the automatically selected number of threads (normally based on the number of physical performance cores, subject to the settings of OpenMP and of the system) if no such number has been set.

This is not the number of threads that OMP currently uses, see `gpb.get.num.threads`: a model for which no number of threads has been specified sets the number of threads returned here (and resets it again) whenever it does calculations, irrespective of the number of threads of the process.

Usage

```
gpb.get.default.num.threads()
```

Value

An `integer` of length one: the number of threads used by models for which no number of threads is specified

Author(s)

Fabio Sigrist

Examples

```
num_threads <- gpb.get.default.num.threads()
```

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

Description

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

Usage

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

Arguments

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

Value

numeric vector of evaluation result

Examples

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

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

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

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

`gpb.get.num.threads` *Get the number of threads used for parallelization*

Description

Returns the number of threads that OMP currently uses for parallelization, i.e. `omp_get_max_threads()`. Note that a team of threads can be smaller than this if the OpenMP runtime limits it, in particular through the limit of the contention group (`OMP_THREAD_LIMIT`). Note that models for which the number of threads has been specified via the `num_parallel_threads` argument of `GPModel` are not affected by this number: such models set (and reset) the number of threads themselves whenever they do calculations.

Usage

```
gpb.get.num.threads()
```

Value

An `integer` of length one: the current maximum number of threads of OpenMP, as returned by `omp_get_max_threads()`

Author(s)

Fabio Sigrist

Examples

```
num_threads <- gpb.get.num.threads()
```

`gpb.grid.search.tune.parameters`
Function for choosing tuning parameters

Description

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

Usage

```
gpb.grid.search.tune.parameters(param_grid, num_try_random = NULL, data,
  gp_model = NULL, params = list(), nrounds = 1000L,
  early_stopping_rounds = NULL, folds = NULL, nfold = 5L, metric = NULL,
  verbose_eval = 1L, cv_seed = NULL, use_gp_model_for_validation = TRUE,
  train_gp_model_cov_pars = TRUE, label = NULL, weight = NULL,
  obj = NULL, eval = NULL, stratified = TRUE, init_model = NULL,
  colnames = NULL, categorical_feature = NULL, callbacks = list(),
  return_all_combinations = FALSE, ...)
```

Arguments

param_grid	list with candidate parameters defining the grid over which a search is done
num_try_random	integer with number of random trial on parameter grid. If NULL, a deterministic search is done
data	a gpb.Dataset object, used for training. Some functions, such as gpb.cv, may allow you to pass other types of data like matrix and then separately supply label as a keyword argument.
gp_model	A GPModel object that contains the random effects (Gaussian process and / or grouped random effects) model
params	list with other parameters not included in param_grid
nrounds	number of boosting iterations (= number of trees). This is the most important tuning parameter for boosting
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.
folds	(list) This provides the 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-sized subsamples. This allows for doing nfold-CV
metric	Evaluation metric to be monitored when doing CV and parameter tuning. Can be a character string or vector of character strings. If not NULL, the metric in params will be overridden. • Default ="test_neg_log_likelihood" if there is a GPModel • Non-exhaustive list of supported metrics: "test_neg_log_likelihood", "mse", "rmse", "mae", "crps_gaussian", "auc", "average_precision", "binary_logloss", "binary_error" • See the "metric" section of the parameter documentation for a complete list of valid metrics

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

cv_seed Seed for generating folds when doing nfold CV

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Value

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

Early Stopping

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

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

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

Author(s)

Fabio Sigrist

Examples

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

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

n <- length(y)
param_grid <- list("learning_rate" = c(0.001, 0.01, 0.1, 1, 10),
                   "min_data_in_leaf" = c(1, 10, 100, 1000),
                   "max_depth" = c(-1),
                   "num_leaves" = 2^(1:10),
                   "lambda_l2" = c(0, 1, 10, 100),
                   "max_bin" = c(250, 500, 1000, min(n,10000)),
                   "line_search_step_length" = c(TRUE, FALSE))
# Note: "max_depth" = c(-1) means no depth limit as we tune 'num_leaves'.
#    Can also additionally tune 'max_depth', e.g., "max_depth" = c(-1, 1, 2, 3, 5, 10)
metric = "mse" # Define metric
```

```
# Note: can also use metric = "test_neg_log_likelihood".
# See https://github.com/fabsig/GPBoost/blob/master/docs/Parameters.rst#metric-parameters
gp_model <- GPModel(group_data = group_data[,1], likelihood="gaussian")
data_train <- gpb.Dataset(data = X, label = y)
set.seed(1)
opt_params <- gpb.grid.search.tune.parameters(param_grid = param_grid,
                                              data = data_train, gp_model = gp_model,
                                              num_try_random = 100, nfold = 5,
                                             nrounds = 1000, early_stopping_rounds = 20,
                                          verbose_eval = 1, metric = metric, cv_seed = 4)
print(paste0("Best parameters: ",
             paste0(unlist(lapply(seq_along(opt_params$best_params),
                                  function(y, n, i) { paste0(n[[i]],": ", y[[i]]) },
                                  y=opt_params$best_params,
                                  n=names(opt_params$best_params))), collapse=", ")))
print(paste0("Best number of iterations: ", opt_params$best_iter))
print(paste0("Best score: ", round(opt_params$best_score, digits=3)))
# Alternatively and faster: using manually defined validation data instead of cross-validation
# use 20% of the data as validation data
valid_tune_idx <- sample.int(length(y), as.integer(0.2*length(y)))
folds <- list(valid_tune_idx)
opt_params <- gpb.grid.search.tune.parameters(param_grid = param_grid,
                                              data = data_train, gp_model = gp_model,
                                              num_try_random = 100, folds = folds,
                                             nrounds = 1000, early_stopping_rounds = 20,
                                          verbose_eval = 1, metric = metric, cv_seed = 4)
```

---

gpb.importance *Compute feature importance in a model*

Description

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

Usage

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

Arguments

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

Value

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

- Feature: Feature names in the model.

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

Examples

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

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

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

---

gpb.interprete *Compute feature contribution of prediction*

Description

Computes feature contribution components of rawscore prediction.

Usage

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

Arguments

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

Value

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

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

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

Examples

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

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

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

---

gpb.load *Load GPBoost model*

Description

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

Usage

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

Arguments

filename	path of model file
model_str	a str containing the model

Value

gpb.Booster

Author(s)

Fabio Sigrist, authors of the LightGBM R package

Examples

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

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

---

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

Description

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

Usage

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

Arguments

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

Value

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

The columns of the data.table are:

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

Examples

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

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

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

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

Description

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

Usage

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

Arguments

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

Details

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

Value

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

Examples

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

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

model <- gpb.train(
```

```
    params = params
    , data = dtrain
    , nrounds = 5L
)

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

---

gpb.plot.interpretation *Plot feature contribution as a bar graph*

Description

Plot previously calculated feature contribution as a bar graph.

Usage

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

Arguments

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

Details

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

Value

The gpb.plot.interpretation function creates a barplot.

Examples

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

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

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

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

---

gpb.plot.part.dep.interact

Plot interaction partial dependence plots

Description

Plot interaction partial dependence plots

Usage

```
gpb.plot.part.dep.interact(model, data, variables, latent_scale = TRUE,
```

```
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
latent_scale	If TRUE, the plot is done on the scale of the tree-ensemble. If FALSE, the plot is done on the (potentially transformed) response variable scale (e.g. probabilities for classification)
n.pt.per.var	Number of grid points per variable (used only if a variable is not discrete) For continuous variables, the two-dimensional grid for the interaction plot has dimension c(n.pt.per.var, n.pt.per.var)
subsample	Fraction of random samples in data to be used for calculating the partial dependence plot
discrete.variables	A vector of length two of type boolean. If an entry is TRUE, the evaluation grid of the corresponding variable is set to the unique values of the variable
which.class	An integer indicating the class in multi-class classification (value from 0 to num_class - 1)
type	A character string indicating the type of the plot. Supported values: "filled.contour" and "contour"
nlevels	Parameter passed to the filled.contour or contour function
xlab	Parameter passed to the filled.contour or contour function
ylab	Parameter passed to the filled.contour or contour function
zlab	Parameter passed to the filled.contour or contour function
main	Parameter passed to the filled.contour or contour function
return_plot_data	A boolean. If TRUE, the data for creating the partial dependence plot is returned
...	Additional parameters passed to the filled.contour or contour function

Value

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

Author(s)

Fabio Sigrist

Examples

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

---

gpb.plot.partial.dependence

Plot partial dependence plots

Description

Plot partial dependence plots

Usage

```
gpb.plot.partial.dependence(model, data, variable, latent_scale = TRUE,
  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
latent_scale	If TRUE, the plot is done on the scale of the tree-ensemble. If FALSE, the plot is done on the (potentially transformed) response variable scale (e.g. probabilities for classification)
n.pt	Evaluation grid size (used only if x is not discrete)
subsample	Fraction of random samples in data to be used for calculating the partial dependence plot

discrete.x	A boolean. If TRUE, the evaluation grid is set to the unique values of x
which.class	An integer indicating the class in multi-class classification (value from 0 to num_class - 1)
xlab	Parameter passed to plot
ylab	Parameter passed to plot
type	Parameter passed to plot
main	Parameter passed to plot
return_plot_data	A boolean. If TRUE, the data for creating the partial dependence plot is returned
...	Additional parameters passed to plot

Value

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

Author(s)

Fabio Sigrist (adapted from a version by Michael Mayer)

Examples

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

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

---

gpb.save *Save GPBoost model (booster and GPModel)*

Description

Save GPBoost model (booster and GPModel)

Usage

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

Arguments

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

Value

gpb.Booster. After loading, the GPModel can be accessed via bst_loaded$.__enclos_env__$private$gp_model

Author(s)

Fabio Sigrist, authors of the LightGBM R package

Examples

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

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

```
# Check equality
pred$fixed_effect - pred_loaded$fixed_effect
pred$random_effect_mean - pred_loaded$random_effect_mean
pred$random_effect_cov - pred_loaded$random_effect_cov
```

`gpb.set.default.num.threads` *Set the number of threads used by models for which no number of threads is specified*

Description

Sets the number of threads that a `GPModel` uses when no number of threads has been specified for it via the `num_parallel_threads` argument. Use this to apply a number of threads that has been determined by `gpb.tune.num.threads` in an earlier session, without running the benchmark again.

In contrast to `gpb.set.num.threads`, this does not change the number of threads that OMP currently uses: it only changes the number of threads that models use when nothing else is requested. The number of threads specified for an individual model always takes precedence.

Usage

```
gpb.set.default.num.threads(num_threads)
```

Arguments

`num_threads`	An `integer` specifying the number of threads. It is limited by the largest number of threads that GPBoost uses on its own: the number of threads that OMP uses when GPBoost determines its default (usually the number of logical processors, or the value of the environment variable `OMP_NUM_THREADS` if it is set), the limit of the contention group of OpenMP (`OMP_THREAD_LIMIT`), and, unless `OMP_NUM_THREADS` is set, a CPU bandwidth limit of a control group on Linux. A number of threads that is specified for an individual model via `num_parallel_threads` is not limited by this. If `num_threads` is not positive, the automatically selected number of threads is used again

Value

This function does not return anything

Author(s)

Fabio Sigrist

Examples

```
num_threads_old <- gpb.get.default.num.threads()
gpb.set.default.num.threads(2L)
gpb.set.default.num.threads(num_threads_old)
```

gpb.set.num.threads *Set the number of threads used for parallelization*

Description

Sets the number of threads used by OMP and Eigen in the entire R process. Note that models for which the number of threads has been specified via the num_parallel_threads argument of GPModel are not affected by this: such models set (and reset) the number of threads themselves whenever they do calculations.

Usage

```
gpb.set.num.threads(num_threads)
```

Arguments

num_threads	An integer specifying the number of threads. If num_threads is not positive, the default number of threads of the session is used, see gpb.get.default.num.threads. Initially, this is the automatically selected number of threads (normally based on the number of physical performance cores, subject to the settings of OpenMP and of the system, see the num_parallel_threads argument of GPModel), and it is the number of threads set by gpb.tune.num.threads or gpb.set.default.num.threads once one of them has been called

Value

This function does not return anything

Author(s)

Fabio Sigrist

Examples

```
num_threads_old <- gpb.get.num.threads()
gpb.set.num.threads(2L)
gpb.set.num.threads(num_threads_old)
```

gpb.train *Main training logic for GBPoost*

Description

Logic to train with GBPoost

Usage

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

Arguments

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

- `learning_rate`: The learning rate, also called shrinkage or damping parameter (default = 0.1). An important tuning parameter for boosting. Lower values usually lead to higher predictive accuracy but more boosting iterations are needed
- `num_leaves`: Number of leaves in a tree. Tuning parameter for tree-boosting (default = 31)
- `max_depth`: Maximal depth of a tree. Tuning parameter for tree-boosting (default = no limit)
- `min_data_in_leaf`: Minimal number of samples per leaf. Tuning parameter for tree-boosting (default = 20)
- `lambda_l2`: L2 regularization (default = 0)
- `lambda_l1`: L1 regularization (default = 0)
- `max_bin`: Maximal number of bins that feature values will be bucketed in (default = 255)
- `line_search_step_length` (default = FALSE): If TRUE, a line search is done to find the optimal step length for every boosting update (see, e.g., Friedman 2001). This is then multiplied by the learning rate
- `train_gp_model_cov_pars` (default = TRUE): If TRUE, the covariance parameters of the Gaussian process are estimated in every boosting iterations, otherwise the gp_model parameters are not estimated. In the latter case, you need to either estimate them beforehand or provide values via the 'init_cov_pars' parameter when creating the gp_model
- `use_gp_model_for_validation` (default = TRUE): If TRUE, the Gaussian process is also used (in addition to the tree model) for calculating predictions on the validation data

- leaves_newton_update (default = FALSE): Set this to TRUE to do a Newton update step for the tree leaves after the gradient step. Applies only to Gaussian process boosting (GPBoost algorithm)
- num_threads: Number of threads. For the best speed, set this to the number of real CPU cores(parallel::detectCores(logical = FALSE)), not the number of threads (most CPU using hyper-threading to generate 2 threads per CPU core). For ordinary use, leave this unspecified to use the default number of threads. Setting num_threads = 1 disables the parallelization and can substantially increase the runtime. A single thread should be chosen deliberately, e.g., to distribute the resources among concurrent model fits or for a specific test; it is not required for correctness or reproducibility.

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

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

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

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

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

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

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

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

- **a. character vector**: Non-exhaustive list of supported metrics: "test_neg_log_likelihood", "mse", "rmse", "mae", "auc", "average_precision", "binary_logloss", "binary_error" See the "metric" section of the parameter documentation for a complete list of valid metrics.
- **b. function**: You can provide a custom evaluation function. This should accept the keyword arguments preds and dtrain and should return a named list with three elements:
  - name: A string with the name of the metric, used for printing and storing results.

  - `value`: A single number indicating the value of the metric for the given predictions and true values
  - `higher_better`: A boolean indicating whether higher values indicate a better fit. For example, this would be `FALSE` for metrics like MAE or RMSE.
- **c. list**: If a list is given, it should only contain character vectors and functions. These should follow the requirements from the descriptions above.

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

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

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

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

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

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

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

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

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

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

Value

a trained booster model `gpb.Booster`.

Early Stopping

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

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

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

Author(s)

Fabio Sigrist, authors of the LightGBM R package

Examples

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


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

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

#--------------------Combine tree-boosting and Gaussian process model----------------
# Create Gaussian process model
gp_model <- GPModel(gp_coords = coords, cov_function = "exponential",
                    likelihood = "gaussian")
# Train model
dtrain <- gpb.Dataset(data = X, label = y)
bst <- gpb.train(data = dtrain, gp_model = gp_model, nrounds = 16,
                 learning_rate = 0.05, max_depth = 6, min_data_in_leaf = 5,
                 verbose = 0)
# Estimated random effects model
summary(gp_model)
# Make predictions
pred <- predict(bst, data = X_test, gp_coords_pred = coords_test,
                predict_cov_mat =TRUE)
```

```
pred$random_effect_mean # Predicted (posterior) mean of GP
pred$random_effect_cov # Predicted (posterior) covariance matrix of GP
pred$fixed_effect # Predicted fixed effect from tree ensemble
# Sum them up to otbain a single prediction
pred$random_effect_mean + pred$fixed_effect


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


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


#--------------------GPBoostOOS algorithm: GP parameters estimated out-of-sample----------------
# Create random effects model and dataset
gp_model <- GPModel(group_data = group_data[,1], likelihood="gaussian")
dtrain <- gpb.Dataset(X, label = y)
params <- list(learning_rate = 0.05,
               max_depth = 6,
               min_data_in_leaf = 5)
# Stage 1: run cross-validation to (i) determine to optimal number of iterations
#           and (ii) to estimate the GPModel on the out-of-sample data
```

```
cvbst <- gpb.cv(params = params,
                data = dtrain,
                gp_model = gp_model,
                nrounds = 100,
                nfold = 4,
                eval = "l2",
                early_stopping_rounds = 5,
                use_gp_model_for_validation = TRUE,
                fit_GP_cov_pars_OOS = TRUE)
print(paste0("Optimal number of iterations: ", cvbst$best_iter))
# Estimated random effects model
# Note: ideally, one would have to find the optimal combination of
#               other tuning parameters such as the learning rate, tree depth, etc.)
summary(gp_model)
# Stage 2: Train tree-boosting model while holding the GPModel fix
bst <- gpb.train(data = dtrain,
                 gp_model = gp_model,
                 nrounds = cvbst$best_iter,
                 learning_rate = 0.05,
                 max_depth = 6,
                 min_data_in_leaf = 5,
                 verbose = 0,
                 train_gp_model_cov_pars = FALSE)
# The GPModel has not changed:
summary(gp_model)
```

`gpb.tune.num.threads` *Benchmark different numbers of threads and select a tuned default*

Description

Measures how long representative GPBoost calculations take with different numbers of OpenMP threads and selects a number of threads that is used by all models for which no number of threads is specified via the `num_parallel_threads` argument of `GPModel`, for the rest of the session.

The benchmark is never run automatically: the number of threads that GPBoost uses without it is the automatically selected one, which is normally based on the number of physical performance cores. The selected number of threads is a tuned default and not an optimal number of threads: the best number of threads depends on the model, on the size of the data and on the machine. Use the `num_parallel_threads` argument of `GPModel` for a model whose number of threads should differ from the default.

The measured times are the times of evaluations of the negative log-likelihood, which is the calculation that dominates the estimation of a model. The simulation of the data and the setup of the models (e.g. the search for the neighbors of the Vecchia approximation) happen outside of the timed sections.

Usage

```
gpb.tune.num.threads(workloads = "all", num_threads_candidates = NULL,
  n_rep = 5L, tolerance = 0.03, max_time = 120,
  max_relative_slowdown = 0.25, set_default = TRUE, verbose = TRUE,
  workload_size = "default")
```

Arguments

workloads	A character vector specifying the workloads that are benchmarked: • "all" (= default): all workloads below • "grouped_re": a single-level grouped random effect model. The calculations are cheap per data point, which makes the overhead of the parallelization visible • "vecchia_non_gaussian": a non-Gaussian Gaussian process model with a Vecchia approximation and iterative methods • "crossed_re_iterative": crossed grouped random effects with iterative methods Restrict the workloads to the model class that you mainly use if you know it
num_threads_candidates	An integer vector with the numbers of threads that are benchmarked. If NULL, powers of two, the automatically selected number of threads and the largest number of threads that GPBoost uses on its own are benchmarked
n_rep	An integer specifying the number of repeated measurements per workload and number of threads. The median of the repetitions is used. Every workload is measured at least twice if n_rep is at least two, since a single measurement has no spread from which the noise could be estimated
tolerance	A numeric specifying the relative difference in runtime that is considered negligible. The smallest number of threads whose aggregated runtime is within this tolerance of the best aggregated runtime is selected, so that more threads are not used for a negligible gain. If the measurements are noisier than this, the observed noise is used instead
max_time	A numeric specifying approximately how many seconds the benchmark may take. Repetitions are dropped if the measurements take longer. This is a soft limit: every workload simulates its data, creates its models and is measured at least once, and the remaining time is divided among the workloads that are still to be measured
max_relative_slowdown	A numeric specifying how much slower than the fastest number of threads a number of threads may be for a single workload and still be selected. This prevents that a number of threads which is much slower for one model class is selected because the aggregate over all workloads looks acceptable
set_default	A logical. If TRUE, the selected number of threads is set as the default of the session, see gpb.set.default.num.threads. Set this to FALSE to only measure
verbose	A logical. If TRUE, the progress and the results are printed

workload_size A string, either "default" or "small". The small workloads run in a few seconds, but they are too small for the number of threads to matter, so they are only useful for checking that the benchmark runs and the default of the session is never changed for them

Value

A list, invisibly, with the following elements:

- num_threads: the selected number of threads
- num_threads_automatic: the automatically selected number of threads
- num_threads_max: the largest number of threads that GPBoost uses on its own
- default_was_set: whether the default of the session has been changed
- timings: a data.frame with the measurements per workload and number of threads (median, minimum and relative median absolute deviation of the repetitions), or NULL if only one number of threads is left to benchmark, in which case nothing is measured and that number of threads is the selected one
- aggregate: a data.frame with the aggregated relative runtime per number of threads, i.e. the geometric mean over the workloads of the runtime relative to the fastest measurement of the workload, and whether the number of threads is acceptable, i.e. whether it is not slower than max_relative_slowdown for a single workload. It is NULL whenever timings is
- num_threads_before: the default of the session before the benchmark
- tolerance_used: the tolerance that has been used for the selection
- safeguard_was_relaxed: TRUE if no number of threads was within max_relative_slowdown of the fastest one for every workload, i.e. if the workloads disagree. The number of threads whose slowest workload is the least slow is selected then

Author(s)

Fabio Sigrist

Examples

```
# Benchmark all workloads and use the result for the rest of the session
results <- gpb.tune.num.threads()

# Benchmark only the model class that is mainly used
results <- gpb.tune.num.threads(workloads = "grouped_re")

# Only measure, without changing the default
results <- gpb.tune.num.threads(set_default = FALSE)
```

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 gpb.Dataset object, used for training. Some functions, such as gpb.cv, may allow you to pass other types of data like matrix and then separately supply label as a keyword argument.
label	Vector of response values / labels, used if data is not an gpb.Dataset
weight	Vector of weights. The GPBoost algorithm currently does not support weights
params	list of "tuning" parameters. See the parameter documentation for more information. A few key parameters: • learning_rate: The learning rate, also called shrinkage or damping parameter (default = 0.1). An important tuning parameter for boosting. Lower values usually lead to higher predictive accuracy but more boosting iterations are needed • num_leaves: Number of leaves in a tree. Tuning parameter for tree-boosting (default = 31) • max_depth: Maximal depth of a tree. Tuning parameter for tree-boosting (default = no limit) • min_data_in_leaf: Minimal number of samples per leaf. Tuning parameter for tree-boosting (default = 20) • lambda_l2: L2 regularization (default = 0) • lambda_l1: L1 regularization (default = 0) • max_bin: Maximal number of bins that feature values will be bucketed in (default = 255) • line_search_step_length (default = FALSE): If TRUE, a line search is done to find the optimal step length for every boosting update (see, e.g., Friedman 2001). This is then multiplied by the learning rate

- `train_gp_model_cov_pars` (default = TRUE): If TRUE, the covariance parameters of the Gaussian process are estimated in every boosting iterations, otherwise the gp_model parameters are not estimated. In the latter case, you need to either estimate them beforehand or provide values via the 'init_cov_pars' parameter when creating the gp_model
- `use_gp_model_for_validation` (default = TRUE): If TRUE, the Gaussian process is also used (in addition to the tree model) for calculating predictions on the validation data
- `leaves_newton_update` (default = FALSE): Set this to TRUE to do a Newton update step for the tree leaves after the gradient step. Applies only to Gaussian process boosting (GPBoost algorithm)
- num_threads: Number of threads. For the best speed, set this to the number of real CPU cores(`parallel::detectCores(logical = FALSE)`), not the number of threads (most CPU using hyper-threading to generate 2 threads per CPU core). For ordinary use, leave this unspecified to use the default number of threads. Setting num_threads = 1 disables the parallelization and can substantially increase the runtime. A single thread should be chosen deliberately, e.g., to distribute the resources among concurrent model fits or for a specific test; it is not required for correctness or reproducibility.

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

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

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

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

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

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

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

- **a. character vector**: Non-exhaustive list of supported metrics: "test_neg_log_likelihood", "mse", "rmse", "mae", "auc", "average_precision", "binary_logloss", "binary_error" See the "metric" section of the parameter documentation for a complete list of valid metrics.

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

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

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

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

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

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

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

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

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

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

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

Value

a trained `gpb.Booster`

Early Stopping

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

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

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

Author(s)

Fabio Sigrist, authors of the LightGBM R package

Examples

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


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

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

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

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

```
                     predict_var = TRUE, pred_latent = FALSE)
pred_resp$response_mean # Predicted response mean
# For Gaussian data: pred$random_effect_mean + pred$fixed_effect = pred_resp$response_mean
pred$random_effect_mean + pred$fixed_effect - pred_resp$response_mean

#--------------------Combine tree-boosting and Gaussian process model----------------
# Create Gaussian process model
gp_model <- GPModel(gp_coords = coords, cov_function = "exponential",
                    likelihood = "gaussian")
# Train model
bst <- gpboost(data = X, label = y, gp_model = gp_model, nrounds = 8,
               learning_rate = 0.1, max_depth = 6, min_data_in_leaf = 5,
               verbose = 0)
# Estimated random effects model
summary(gp_model)
# Make predictions
pred <- predict(bst, data = X_test, gp_coords_pred = coords_test,
                predict_var = TRUE, pred_latent = TRUE)
pred$random_effect_mean # Predicted latent random effects mean
pred$random_effect_cov # Predicted random effects variances
pred$fixed_effect # Predicted fixed effects from tree ensemble
# Predict response variable
pred_resp <- predict(bst, data = X_test, gp_coords_pred = coords_test,
                     predict_var = TRUE, pred_latent = FALSE)
pred_resp$response_mean # Predicted response mean
```

`GPBoost_data` *Example data for the GPBoost package*

Description

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

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

Usage

```
data(GPBoost_data)
```

GPModel *Create a* GPModel *object*

Description

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

Usage

```
GPModel(likelihood = "gaussian", group_data = NULL,
  group_rand_coef_data = NULL, ind_effect_group_rand_coef = NULL,
  drop_intercept_group_rand_effect = NULL, gp_coords = NULL,
  gp_rand_coef_data = NULL, cov_function = "matern", cov_fct_shape = 1.5,
  gp_approx = "none", num_parallel_threads = NULL, GPU_use = FALSE,
  matrix_inversion_method = "default", weights = NULL,
  additional_likelihood_data = NULL,
  likelihood_learning_rate = 1, cov_fct_taper_range = 1,
  cov_fct_taper_shape = 1, num_neighbors = NULL,
  vecchia_ordering = "random", ind_points_selection = "kmeans++",
  num_ind_points = NULL, cover_tree_radius = 1, seed = 0L,
  cluster_ids = NULL, likelihood_additional_param = NULL,
  fidelity_specific_mean = TRUE, cov_fct_order = 1L, num_data = NULL,
  free_raw_data = FALSE, vecchia_approx = NULL, vecchia_pred_type = NULL,
  num_neighbors_pred = NULL)
```

Arguments

likelihood — A string specifying the likelihood function (distribution) of the response variable. The currently supported likelihoods, grouped by response type, are:

Response type	Support	Likelihoods
Continuous	y in (-inf, inf)	"gaussian", "t", "t_fix_df", "quantile_regressi
Positive continuous	y in (0, inf)	"gamma", "gamma_varying_shape", "lognorm
Non-negative continuous / semicontinuous	y in [0, inf) with point mass at 0	"tweedie", "tweedie_fixed_p", "tweedie_joint
Count	y in {0, 1, 2, ...}	"poisson", "negative_binomial", "negative_bi
Binary	y in {0, 1}	"bernoulli_logit", "bernoulli_probit"
Proportion / fractional / bounded	y in [0, 1]	"quasi_bernoulli_logit", "quasi_bernoulli_pro

The table lists the main likelihood names. Aliases, heteroscedastic and varying-shape variants, and the "_regression" variants of the two-part likelihoods are documented in the detailed descriptions below.

Continuous response: y in (-inf, inf)

- "gaussian": Gaussian likelihood

- "t": t-distribution (e.g., for robust regression). The default approximation is Fisher-Laplace, i.e., the Fisher information is used instead of the observed Hessian in the Laplace approximation of the marginal likelihood.
- "t_fix_df": t-distribution with the degrees-of-freedom (df) held fixed and not estimated
  - The degrees-of-freedom (df) can be set via the `likelihood_additional_param` parameter. The default is df = 2
- "quantile_regression" / "asymmetric_laplace" : an asymmetric Laplace likelihood for quantile regression. Both names are accepted, as is the alias "quantile"
  - The quantile must be supplied through the `likelihood_additional_param` parameter and must be strictly between 0 and 1

  The default approximation is Fisher-Laplace, i.e., the Fisher information is used instead of the observed Hessian in the Laplace approximation of the marginal likelihood. The triangular-kernel-curvature (TKC) approximation can be enabled by appending "_triangular_kernel_curvature" or the shorthand "_tkc", for example, "quantile_regression_tkc" or "asymmetric_laplace_triangular_kernel_curvature".
  - experimental: the log-likelihood is not differentiable at the kinks where 'y_i' equals the location parameter, and the quasi-Newton mode finding can therefore stall at a point that is not the exact posterior mode. Appending "_ssn_alm", for example "quantile_regression_ssn_alm", enables an exact check of the (non-smooth) optimality conditions after the mode finding and, if the check fails, a refinement of the mode with a semismooth Newton method applied to the subproblems of an augmented Lagrangian method. This makes the approximate marginal likelihood a well-defined function of the parameters, i.e., independent of the path taken by the optimizer and of the number of threads, at the price of additional linear solves during mode finding. This matters for standard errors and for likelihood-based model comparison, but it does not necessarily improve predictive accuracy; increasing `max_num_restarts_lbfgs` can be a cheaper way of avoiding poor optima. Appending "_admm_ssn_alm" instead warm starts the refinement with an ADMM phase, which typically halves its cost when `matrix_inversion_method = "cholesky"` and is not recommended for iterative methods
- "gaussian_heteroscedastic": Gaussian likelihood where the mean is related to fixed and random effects and the log-error variance is related to fixed effects only (linear predictor or GPBoost algorithm). The estimated coefficients of the log-variance model are returned alongside the mean-model coefficients (with the suffix "_scale"). Fisher-Laplace is the default and currently the only implemented approximation.
- "gaussian_heteroscedastic_fixed_and_random": Gaussian likelihood where both the mean and the variance are related to fixed and random effects. This is currently only implemented for GPs with a 'vecchia' approximation. Fisher-Laplace is the default and currently the only implemented approximation. By default, Cholesky factorizations are used. If init_cov_pars is not given, the initial values are obtained from a homoscedastic Gaussian process model (unless some weights are zero).

Positive continuous response: y in (0, inf)

- "gamma": Gamma likelihood with a log link function
- "gamma_varying_shape": As "gamma", but the gamma shape varies across observations and is modeled by an additional fixed-effects-only predictor: log(shape) = F_s(X) (F_s(X) = linear predictor or the GPBoost algorithm), while the log mean log(mu) = F(X) + Zb is related to both fixed and random effects. Note that the shape also governs the dispersion, var(y) = mu^2 / shape. The estimated coefficients of the log-shape model are returned alongside the mean-model coefficients (with the suffix "_shape"). See also the corresponding two-part likelihoods "hurdle_gamma_varying_shape" and "hurdle_regression_gamma_varying_shape" below
- "lognormal": Log-normal likelihood with a log link function
- "gpd": Generalized Pareto likelihood for finite positive responses. The log scale parameter equals the latent predictor eta (sum of fixed and random effects), sigma = exp(eta), and the estimated auxiliary parameter is shape > -0.5
- "egpd_power": Naveau power-carrier extended generalized Pareto likelihood with auxiliary parameters shape and kappa
- "egpd_power_mixture": Naveau ordered power-mixture carrier with auxiliary parameters shape, kappa1, delta_kappa, and p
- "egpd_beta": Naveau beta-carrier extended generalized Pareto likelihood with auxiliary parameters shape and delta
- "egpd_power_beta": Naveau power-beta carrier with auxiliary parameters shape, delta, and kappa

Non-negative continuous / semicontinuous response: y in [0, inf) with point mass at 0

- "tweedie": Compound Poisson–Gamma Tweedie likelihood with a log link, variance phi * mu^p, and 1.01 < p < 1.99. The dispersion phi and power p are estimated
- "tweedie_fixed_p": The same Tweedie likelihood with p fixed through `likelihood_additional_pa` only phi is estimated. The fixed power must satisfy 1.01 < p < 1.99
- "tweedie_joint", "tweedie_joint_fixed_p": The same Tweedie model, but the observed number of events (e.g., claims) N, given in the first column of `additional_likelihood_data`, is used jointly with the aggregate response y. The likelihood is the joint density of (y, N) of the compound Poisson–Gamma representation, N ~ Poisson(mu^(2-p) / (phi * (2-p))) and y | N = n ~ Gamma(n * (2-p) / (p-1), scale = phi * (p-1) * mu^(p-1)), see Jorgensen and de Souza (1994, Scandinavian Actuarial Journal). Given phi and p, the mean model is the same as for "tweedie"; N adds information for estimating phi and p. N must be a non-negative integer that is 0 if and only if y is 0. Predictions are the same as for "tweedie", and the "test_neg_log_likelihood" metric uses the marginal density of y since N is not known for new data
- "tweedie_varying_dispersion", "tweedie_varying_dispersion_fixed_p": As "tweedie" and "tweedie_fixed_p", but the dispersion phi varies across observations: log(phi) = F_d(X) is related to fixed effects only (linear predictor or GPBoost algorithm), while log(mu) = F(X) + Zb is related to both

fixed and random effects. The power p is then the only (auxiliary) parameter. The estimated coefficients of the log-dispersion model are returned alongside the mean-model coefficients (with the suffix "_dispersion")

- "tweedie_joint_varying_dispersion", "tweedie_joint_varying_dispersion_fixed_p": The joint (y, N) likelihood of "tweedie_joint" with the varying dispersion of "tweedie_varying_dispersion"
- "hurdle_<base>": Two-part likelihoods for non-negative response variables with an excess probability 'p0' of exact zeros. They combine a point mass 'p0' at zero with a base distribution with support 'y > 0' for the remaining mass '1 - p0'. The fixed effects 'F(X)' and random effects 'Zb' enter only through the base component: 'exp(F(X) + Zb)' is its mean or scale parameter, not the unconditional response mean, which is E(y) = (1 - p0) * base_mean. The structural-zero probability 'p0' is estimated jointly with the base auxiliary parameters. Currently supported bases: "hurdle_gamma", "hurdle_lognormal", and the extreme-value bases "hurdle_gpd", "hurdle_egpd_power", "hurdle_egpd_power_mixture", "hurdle_egpd_beta", "hurdle_egpd_power_beta". For all these bases, the "zero_inflated" prefix is accepted as an alias, e.g. "zero_inflated_gamma" maps to "hurdle_gamma". See "zero_inflated_<base>" under count responses for the corresponding count-data family
- "hurdle_regression_<base>": As "hurdle_<base>", but the structural-zero probability is modeled as a logistic regression on the covariates instead of as a constant (e.g. "hurdle_regression_gamma", "hurdle_regression_lognormal", "hurdle_regression_gpd", "hurdle_regression_egpd_power", "hurdle_regression_egpd_power_mixtu "hurdle_regression_egpd_beta", "hurdle_regression_egpd_power_beta"). The zero probability is then pi_i = 1 / (1 + exp(-x_i'alpha)), modeled through a second fixed-effects-only predictor that reuses the same design matrix X as the response model (the response predictor carries the random effects, the zero predictor does not). The estimated zero-model coefficients alpha are returned by get_coef() alongside the response-model coefficients, with the suffix "_zero"
- "hurdle_gamma_varying_shape", "hurdle_regression_gamma_varying_shape": As "hurdle_gamma" and "hurdle_regression_gamma", but with the observation-specific gamma shape of "gamma_varying_shape". For "hurdle_regression_gamma_varying_shape" there are three predictors: the response mean, the structural-zero logit (coefficients with the suffix "_zero"), and log(shape)
- "zero_censored_power_transformed_normal": Likelihood of a censored and power-transformed normal variable for modeling data with a point mass at 0 and a continuous distribution for y > 0. The model used is Y = max(0,X)^lambda, X ~ N(mu, sigma^2), where mu = F(X) + Zb, and sigma and lambda are (auxiliary) parameters that are estimated. For more details on this model, see Sigrist et al. (2012, AOAS) "A dynamic nonstationary spatio-temporal model for short term prediction of precipitation"
- "zero_censored_power_transformed_normal_heteroscedastic": As "zero_censored_power_transforn but the standard deviation sigma of the latent normal variable varies across observations: log(sigma) = F_2(X) is related to fixed effects only (linear predictor or GPBoost algorithm), while mu = F(X) + Zb is related to both fixed and random effects. lambda is then the only (auxiliary) parameter that

is estimated

- "zero_censored_shifted_gamma": Zero-censored shifted gamma likelihood for modeling data with a point mass at 0 and a continuous distribution for y > 0. The model used is Y = max(Z - xi, 0), where Z follows a gamma distribution with mean mu = exp(F(X) + Zb) and shape k. The shape k and shift xi are (auxiliary) parameters that are estimated. This is the version of "zero_one_censored_shifted_gamma" (Sigrist and Stahel, 2011) without the upper censoring at 1
- "zero_censored_shifted_gamma_varying_shape": As "zero_censored_shifted_gamma", but the shape k varies across observations: log(k) = F_s(X) is related to fixed effects only (covariates and / or the GPBoost tree-boosting algorithm; no random effects / GPs for the shape), while log(mu) = F(X) + Zb is related to both fixed and random effects. The shift xi is then the only (auxiliary) parameter that is estimated. The estimated coefficients of the log-shape model are returned alongside the mean-model coefficients (with the suffix "_shape")

Count response: y in {0, 1, 2, ...}

- "poisson": Poisson likelihood with a log link function
- "negative_binomial": negative binomial likelihood with a log link function (aka "nbinom2", "negative_binomial_2"). The variance is mu * (mu + r) / r, mu = mean, r = shape, with this parametrization
- "negative_binomial_1": Negative binomial 1 (aka "nbinom1") likelihood with a log link function. The variance is mu * (1 + phi), mu = mean, phi = dispersion, with this parametrization
- "zero_inflated_<base>": Two-part count likelihoods that combine a point mass 'p0' at zero with a count base distribution for the remaining mass '1 - p0' (the base can itself generate additional zeros). As for the hurdle likelihoods, 'exp(F(X) + Zb)' is the mean of the (non-structural) base component - not the unconditional response mean - so E(y) = (1 - p0) * base_mean, and 'p0' is estimated jointly with the base auxiliary parameters. Currently supported bases: "zero_inflated_poisson", "zero_inflated_negative_binomial" (aux: shape; aka "zero_inflated_nbinom2"), "zero_inflated_negative_binomial_1" (aux: dispersion; aka "zero_inflated_nbinom1")
- "zero_inflated_regression_<base>": As "zero_inflated_<base>", but with the logistic structural-zero model described under "hurdle_regression_<base>" above: "zero_inflated_regression_poisson", "zero_inflated_regression_negative_binomial", and "zero_inflated_regression_negative_binomial_1"

Binary response: y in {0, 1}

- "bernoulli_logit": Bernoulli likelihood with a logit link function for binary classification. Aliases: "binary", "binary_logit"
- "bernoulli_probit": Bernoulli likelihood with a probit link function for binary classification. Aliases: "binary_probit"

Proportion / fractional / bounded response: y in [0, 1]

- "quasi_bernoulli_logit": quasi-Bernoulli likelihood with a logit link function for y in [0,1]. Aliases: "quasi_binary", "quasi_binary_logit"

- "quasi_bernoulli_probit": quasi-Bernoulli likelihood with a probit link function for y in [0,1]. Aliases: "quasi_binary_probit"
- "binomial_logit": Binomial likelihood with a logit link function. The response variable `y` needs to contain proportions of successes / trials, and the `weights` parameter needs to contain the numbers of trials. Aliases: "binomial"
- "binomial_probit": Binomial likelihood with a probit link function. The response variable `y` needs to contain proportions of successes / trials, and the `weights` parameter needs to contain the numbers of trials
- "beta_binomial": Beta-binomial likelihood with a logit link function. The response variable `y` needs to contain proportions of successes / trials, and the `weights` parameter needs to contain the numbers of trials. Aliases: "betabinomial", "beta-binomial"
- "beta" : Beta likelihood with a logit link function (parametrization of Ferrari and Cribari-Neto, 2004). The response must lie strictly in (0,1)
- "zoctn": Zero-one censored transformed normal likelihood for modeling data in [0,1] with point masses at 0 and 1 and a continuous distribution on (0,1). The model used is T ~ N(mu, sigma^2), W = max(min(T,1),0), and Y = g(W), where g(x) = expit(a + b * logit(x)) for x in (0,1), mu = F(X) + Z_RE u, u denotes the random effects, Z_RE is their design matrix, and sigma, a, and b are (auxiliary) parameters that are estimated. For more details on this model, see Qiang and Sigrist (2026)
- "zero_one_censored_transformed_beta": Zero-one censored transformed beta likelihood for modeling data in [0,1] with point masses at 0 and 1 and a continuous distribution on (0,1). If T follows a beta distribution with mean mu = expit(F(X) + Zb) and precision phi, the observed response is obtained by applying the linear transformation Y = (1 + 2u) * T - u and censoring the result to [0,1]. The precision phi and shift u are (auxiliary) parameters that are estimated. For more details on this model, see Kosmidis and Zeileis (2025)
- "zero_one_censored_shifted_gamma": Zero-one censored shifted gamma likelihood for modeling data in [0,1] with point masses at 0 and 1 and a continuous distribution on (0,1). The model used is Y = min(max(Z - xi, 0), 1), where Z follows a gamma distribution with mean mu = exp(F(X) + Zb) and shape k. The shape k and shift xi are (auxiliary) parameters that are estimated. For more details on this model, see Sigrist and Stahel (2011)

Notes

- The first lines in the likelihoods source file contain additional comments on the specific parametrizations used
- Other likelihoods can be implemented upon request

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

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

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

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

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

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

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

  - "matern": Matern covariance function with the smoothness specified by the cov_fct_shape parameter (using the parametrization of Rasmussen and Williams, 2006)
  - "matern_estimate_shape": same as "matern" but the smoothness parameter is also estimated
  - "matern_space_time": Spatio-temporal Matern covariance function with different range parameters for space and time. Note that the first column in gp_coords must correspond to the time dimension
  - "space_time_gneiting": Spatio-temporal covariance function given in Eq. (16) of Gneiting (2002). Note that the first column in gp_coords must correspond to the time dimension. This covariance has seven parameters (in the following order: sigma2, a, c, alpha, nu, beta, delta) which are all estimated by default. You can disable the estimation of some of these parameter using the 'estimate_cov_par_index' argument of the params argument in either the fit function of a gp_model object or the set_optim_params function prior to estimation.
  - "matern_ard": anisotropic Matern covariance function with Automatic Relevance Determination (ARD), i.e., with a different range parameter for every coordinate of gp_coords
  - "matern_ard_estimate_shape": same as "matern_ard" but the smoothness parameter is also estimated
  - "exponential": Exponential covariance function (using the parametrization of Diggle and Ribeiro, 2007)
  - "gaussian": Gaussian, aka squared exponential, covariance function (using the parametrization of Diggle and Ribeiro, 2007)

- "gaussian_ard": anisotropic Gaussian, aka squared exponential, covariance function with Automatic Relevance Determination (ARD), i.e., with a different range parameter for every coordinate of gp_coords
- "powered_exponential": powered exponential covariance function with the exponent specified by the cov_fct_shape parameter (using the parametrization of Diggle and Ribeiro, 2007)
- "wendland": Compactly supported Wendland covariance function (using the parametrization of Bevilacqua et al., 2019, AOS)
- "linear": linear covariance function. This corresponds to a Bayesian linear regression model with a Gaussian prior on the coefficients with a constant variance diagonal prior covariance, and the prior variance is estimated using empirical Bayes.
- "hurst": Hurst covariance function cov(s, s') = (sigma2 / 2) * (||s||^(2H) + ||s'||^(2H) - ||s - s'||^(2H)), 0 < H < 1. For H = 0.5, this corresponds to Brownian motion in one dimension (-> see the 'estimate_cov_par_index' argument). This is the covariance for cov_fct_order = 1 (default). For cov_fct_order = 2, the second-order Hurst covariance cov(s, s') = sigma2 / (2 (2H - 1)) * (||s - s'||^(2H) - ||s||^(2H) - ||s'||^(2H) + 2H (s^T s') (||s||^(2H-2) + ||s'||^(2H-2))), 1 < H < 2, is used. In one dimension and for s, s' >= 0, H = 1.5 corresponds to an integrated Wiener process. See cov_fct_order for more details
- "hurst_ard": Hurst covariance function with with Automatic Relevance Determination (ARD), i.e., with a different range parameter for every coordinate of "gp_coords" except for the first coordinate which has a range parameter of 1 due to identifiability with the marginal variance: $cov(s, s') = (\sigma^2/2)\left[\left(s_1^2 + \sum_{k=2}^{d}(s_k/l_k)^2\right)^H + \left(s'^2_1 + \sum_{k=2}^{d}(s'_k/l_k)^2\right)^H - \left((s_1 - s'_1)^2 + \sum_{k=2}^{d}((s_k - s'_k$
  For cov_fct_order = 2, the second-order Hurst covariance (see "hurst") is applied to the scaled coordinates (s_1, s_2/l_2, ..., s_d/l_d)
- "ar1_mf_<base>": two-level autoregressive multifidelity covariance defined by $f_H(x) = \rho f_L(x) + \delta(x)$, where f_L and δ are independent Gaussian processes using the same base covariance type but separate parameter vectors.
  - For example, use "ar1_mf_matern", "ar1_mf_matern_ard", or "ar1_mf_matern_estimate_shape"
  - The last column of gp_coords must be 0 for low fidelity or 1 for high fidelity. All preceding columns are input coordinates for the GPs.
  - Covariance parameters are ordered as [low-fidelity base parameters, discrepancy base parameters, rho]. The two base blocks use the ordinary parameter ordering of the base covariance, and rho is unrestricted and can be negative.
  - Any supported base covariance except "wendland" can be used. Correlation tapering and Gaussian-process random coefficients are currently not supported for this model.

cov_fct_shape — A numeric specifying the shape parameter of the covariance function (e.g., smoothness parameter for Matern and Wendland covariance) This parameter is irrelevant for some covariance functions such as the exponential or Gaussian

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

- "none": No approximation
- "vecchia": Vecchia approximation; see Sigrist (2022, JMLR) for more details. For "matern_space_time" and the anisotropic ARD covariance functions ("matern_ard", "gaussian_ard", "matern_ard_estimate_shape"), neighbors are selected by absolute correlation, i.e., using distances in coordinates scaled by the range parameters, and they are redetermined during parameter estimation. For "space_time_gneiting" and "ar1_mf_<base>", neighbors are also selected by absolute correlation by default. Use gp_approx = "vecchia_euclidean" for Euclidean-distance selection.
- "full_scale_vecchia": Vecchia-inducing points full-scale (VIF) approximation; see Gyger, Furrer, and Sigrist (2026, JMLR) for more details
- "tapering": The covariance function is multiplied by a compactly supported Wendland correlation function
- "fitc": Fully Independent Training Conditional approximation aka modified predictive process approximation
- "full_scale_tapering": Full-scale approximation combining an inducing point / predictive process approximation with tapering on the residual process; see Gyger, Furrer, and Sigrist (2026, JUQ) for more details
- "vecchia_latent": similar as "vecchia" but a Vecchia approximation is applied to the latent Gaussian process for likelihood == "gaussian". For likelihood != "gaussian", "vecchia" and "vecchia_latent" are equivalent

`num_parallel_threads`
: An `integer` specifying the number of parallel threads for OMP. If num_parallel_threads = NULL, the default number of threads of the session is used, see `gpb.get.default.num.threads`. Initially, this is the number of physical performance cores, which is determined automatically as follows. On CPUs whose cores have different speeds (e.g., the performance and efficiency cores of recent Intel CPUs or Apple silicon), the fastest cores are counted, and hyperthreads are counted only once, since slow threads can slow down an entire model. The next fastest cores are added if the fastest ones alone would leave only a single thread. On Linux, a CPU bandwidth limit of a control group (e.g., of a container) is respected as well. If the environment variable OMP_NUM_THREADS is set, it takes precedence over both the topology of the CPU and the limit of the control group, and the number of threads that OpenMP is configured to use is taken as it is if the cores of the CPU cannot be determined. In either case, the limits of the OpenMP runtime itself still apply, in particular the limit of the contention group (OMP_THREAD_LIMIT). For ordinary use, leave num_parallel_threads unspecified to use this default. The default of the session can be changed with `gpb.tune.num.threads`, which benchmarks different numbers of threads, or with `gpb.set.default.num.threads`. Setting num_parallel_threads = 1 disables the OpenMP parallelization of the model and can substantially increase the runtime. A single thread should be chosen deliberately, e.g., to distribute the resources among concurrent model fits or for a specific test; it is not required for correctness or reproducibility

`GPU_use`
: A `boolean`. If TRUE, GPU acceleration will be used if supported

`matrix_inversion_method`
: A `string` specifying the method used for inverting covariance matrices. Available options:

- "default": iterative methods where possible, otherwise Cholesky factorization. Cholesky factorization is also used for likelihood = "gaussian_heteroscedastic_fixed_and_rando
- "cholesky": Cholesky factorization
- "iterative": iterative methods. A combination of the conjugate gradient, the Lanczos algorithm, and other methods.
  This is currently only supported for the following cases:
  - grouped random effects with more than one level
  - likelihood != "gaussian" and gp_approx == "vecchia" (non-Gaussian likelihoods with a Vecchia-Laplace approximation)
  - likelihood != "gaussian" and gp_approx == "full_scale_vecchia" (non-Gaussian likelihoods with a VIF approximation)
  - likelihood == "gaussian" and gp_approx == "full_scale_tapering" (Gaussian likelihood with a full-scale tapering approximation)

`weights` A `vector` with sample weights. For a Gaussian likelihood, the error variance ("nugget") for observation `i` is divided by `weights[i]`. For non-Gaussian likelihoods, the conditional log-likelihood contribution of observation `i` is multiplied by `weights[i]`. Consequently, weights affect the estimation of both random and fixed effects. Note that a Gaussian likelihood is calculated via the Laplace approximation when `gp_approx = "vecchia_latent"`, when `likelihood = "gaussian_latent"`, and when grouped random effects are combined with a Vecchia-approximated Gaussian process. In these cases, the weights act as for a non-Gaussian likelihood, i.e., the Gaussian log-likelihood contribution of observation `i` is multiplied by `weights[i]` instead of the error variance being divided by it.

`additional_likelihood_data`
A `vector` or `matrix` with observation-level data that some likelihoods require in addition to the response variable `y` (one row per data point). Currently, this is only used by the joint Tweedie likelihoods ("tweedie_joint" and its variants), for which its first column contains the observed number of events (e.g., claims) N. N must be a non-negative integer that is 0 if and only if y is 0

`likelihood_learning_rate`
A `numeric` with a learning rate for the likelihood for generalized Bayesian inference (only non-Gaussian likelihoods)

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

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

`num_neighbors` An `integer` specifying the number of neighbors for the Vecchia and VIF approximations. Internal default values if NULL:

- 20 for gp_approx = "vecchia"
- 30 for gp_approx = "full_scale_vecchia"

Note: for prediction, the number of neighbors can be set through the 'num_neighbors_pred' parameter in the 'set_prediction_data' function. By default, num_neighbors_pred

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

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

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

`num_ind_points`
: An `integer` specifying the number of inducing points / knots for FITC, full_scale_tapering, and VIF approximations. Internal default values if NULL:
  - 500 for gp_approx = "FITC" and gp_approx = "full_scale_tapering"
  - 200 for gp_approx = "full_scale_vecchia"

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

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

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

`likelihood_additional_param`
: A `numeric` specifying an additional parameter for the `likelihood` which cannot be estimated for this `likelihood` (e.g., degrees of freedom for `likelihood = "t_fix_df"`). This is not to be confused with any auxiliary parameters that can be estimated and accessed through the function `get_aux_pars` after estimation. Note that this `likelihood_additional_param` parameter is irrelevant for many likelihoods. If `likelihood_additional_param = NULL`, the following default and mandatory-value rules apply:
  - df = 2 for likelihood = "t_fix_df"
  - No default is used for likelihood = "asymmetric_laplace"; a quantile strictly between 0 and 1 is required
  - No default is used for the Tweedie likelihoods with a fixed power ("tweedie_fixed_p" and the other "_fixed_p" variants); a power strictly between 1.01 and 1.99 is required

`fidelity_specific_mean`
: A `logical`. For an `ar1_mf_<base>` covariance, whether marginal mean models are fitted independently for low and high fidelity. If `TRUE` (the default), linear covariates supplied to `fit()` are internally expanded into separate low- and

high-fidelity coefficient blocks, and the fidelity indicator is automatically added as a feature when the model is used in the GPBoost algorithm. Ignored for other covariance functions.

`cov_fct_order` An `integer` specifying the order m of the "hurst" and "hurst_ard" covariance functions (also when used as base covariance in "ar1_mf_hurst" and "ar1_mf_hurst_ard"). Currently, the orders 1 (default) and 2 are supported. The Hurst exponent H of the order m satisfies m - 1 < H < m, and H = m - 0.5 is used as initial value. Order 1 is a fractional Brownian motion / field anchored at the origin (b(0) = 0), and order 2 is a second-order Hurst process / field anchored at the origin such that both the process and its gradient are zero there (b(0) = 0, grad b(0) = 0). The origin of the coordinates thus is a part of the model, and the coordinates are not centered internally. The order is not estimated. It can be chosen, e.g., by comparing the marginal likelihoods or by cross-validation.

- Continuous-time RW1 prior: use `cov_function = "hurst"`, `cov_fct_order = 1`, and fix H = 0.5. This is a Brownian motion with b(0) = 0 and b'(t) = sqrt(q) W'(t), where W' denotes white noise and q = sigma2
- Continuous-time RW2 prior: use `cov_function = "hurst"`, `cov_fct_order = 2`, and fix H = 1.5. This is an integrated Wiener process with b(0) = b'(0) = 0 and b"(t) = sqrt(q) W'(t), where q = 3 * sigma2

For these priors, use a one-dimensional time coordinate t >= 0 whose origin t = 0 is the desired anchor (shift the time points if necessary). H can be fixed using the 'init_cov_pars' and 'estimate_cov_par_index' arguments of the `params` argument, see the examples of `fitGPModel`. If the polynomial null space of the intrinsic model should not be penalized, include an intercept (RW1) or an intercept and a linear time trend (RW2) as fixed effects. Note that the order 2 model with H = 1.5 is the continuous-time RW2 prior evaluated at the time points and not the usual discrete intrinsic RW2 prior on a lattice: for equally spaced time points, adjacent second differences are correlated (correlation 1/4) whereas they are independent for the discrete RW2

`num_data` A `numeric` with the number of samples. This is only used for iid models

`free_raw_data` A `boolean`. If TRUE, the data (groups, coordinates, covariate data for random coefficients, cluster IDs, weights, and additional likelihood data) is freed in R after initialization

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

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

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

Value

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

Author(s)

Fabio Sigrist

Examples

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

data(GPBoost_data, package = "gpboost")

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

#--------------------Gaussian process model----------------
gp_model <- GPModel(gp_coords = coords, cov_function = "matern", cov_fct_shape = 1.5,
                    likelihood="gaussian")

#--------------------Combine Gaussian process with grouped random effects----------------
gp_model <- GPModel(group_data = group_data,
                    gp_coords = coords, cov_function = "matern", cov_fct_shape = 1.5,
                    likelihood="gaussian")
```

---

group_data *Gouping data for 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 *Test grouping data for example data for the GPBoost package*

Description

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

Usage

```
data(GPBoost_data)
```

`loadGPModel` *Load a* `GPModel` *from a file*

Description

Load a `GPModel` from a file

Usage

```
loadGPModel(filename)
```

Arguments

`filename`	filename for loading

Value

A `GPModel`

Author(s)

Fabio Sigrist

Examples

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

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

neg_log_likelihood *Evaluate the negative log-likelihood*

Description

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

Usage

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

Arguments

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

Author(s)

Fabio Sigrist

Examples

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

neg_log_likelihood.GPModel *Evaluate the negative log-likelihood*

Description

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

Usage

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

Arguments

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

Value

A GPModel

Author(s)

Fabio Sigrist

Examples

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

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

Description

Prediction function for gpb.Booster objects

Usage

```
## S3 method for class 'gpb.Booster'
predict(object, data, start_iteration = NULL,
  num_iteration = NULL, pred_latent = FALSE, predleaf = FALSE,
  predcontrib = FALSE, header = FALSE, reshape = FALSE,
  group_data_pred = NULL, group_rand_coef_data_pred = NULL,
  gp_coords_pred = NULL, gp_rand_coef_data_pred = NULL,
  cluster_ids_pred = NULL, predict_cov_mat = FALSE, predict_var = FALSE,
  sample_posterior = FALSE, num_post_samples = 100, cov_pars = NULL,
  offset_pred = NULL, ignore_gp_model = FALSE, rawscore = NULL,
  vecchia_pred_type = NULL, num_neighbors_pred = NULL, ...)
```

Arguments

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

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

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

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

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

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

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

sample_posterior
: A boolean. If TRUE, samples from the posterior are drawn

num_post_samples
: A numeric with the number of posterior samples to draw if 'sample_posterior=TRUE'

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

offset_pred
: A numeric vector. Offsets for prediction: additional fixed effects contributions that are added to the predictor for the prediction points. The length of this vector needs to equal the number of prediction points times the number of fixed-effect sets.

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

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

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

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

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

Value

either a list with vectors or a single vector / matrix depending on whether there is a gp_model or not

- If there is a gp_model, the result dict contains the following entries.
  - 1. If pred_latent is FALSE (=default), the dict contains the following entries:
    - result["response_mean"] are the predictive 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 predictive covariances or variances of the response variable (only if 'predict_var' or 'predict_cov' is TRUE)
    - result["posterior_samples"] samples of the posterior of the response variable (if 'sample_posterior' is TRUE)
  - 2. If pred_latent is TRUE, the dict contains the following entries:
    - result["fixed_effect"] are the predictions from the tree-ensemble
    - result["random_effect_mean"] are the predictive means of the gp_model excluding the 'fixed_effect'
    - result["random_effect_cov"] are the predictive covariances or variances of the gp_model (only if 'predict_var' or 'predict_cov' is TRUE)
    - result["posterior_samples"] samples of the posterior latent variable including both the fixed effects and the gp_model (if 'sample_posterior' is TRUE)
- If there is no gp_model or predcontrib or ignore_gp_model are TRUE, the result contains predictions from the tree-booster only.

Author(s)

Fabio Sigrist, authors of the LightGBM R package

Examples

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


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

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

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

```
summary(gp_model)

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

#--------------------Combine tree-boosting and Gaussian process model----------------
# Create Gaussian process model
gp_model <- GPModel(gp_coords = coords, cov_function = "exponential",
                    likelihood = "gaussian")
# Train model
bst <- gpboost(data = X, label = y, gp_model = gp_model, nrounds = 8,
               learning_rate = 0.1, max_depth = 6, min_data_in_leaf = 5,
               verbose = 0)
# Estimated random effects model
summary(gp_model)
# Make predictions
pred <- predict(bst, data = X_test, gp_coords_pred = coords_test,
                predict_var = TRUE, pred_latent = TRUE)
pred$random_effect_mean # Predicted latent random effects mean
pred$random_effect_cov # Predicted random effects variances
pred$fixed_effect # Predicted fixed effects from tree ensemble
# Predict response variable
pred_resp <- predict(bst, data = X_test, gp_coords_pred = coords_test,
                     predict_var = TRUE, pred_latent = FALSE)
pred_resp$response_mean # Predicted response mean
```

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

Description

Make predictions for a `GPModel`

Usage

```
## S3 method for class 'GPModel'
predict(object, predict_response = TRUE,
  predict_var = FALSE, predict_cov_mat = FALSE, sample_posterior = FALSE,
```

```
  sample_prior = FALSE, num_post_samples = 100, num_prior_samples = 100,
  y = NULL, cov_pars = NULL, group_data_pred = NULL,
  group_rand_coef_data_pred = NULL, gp_coords_pred = NULL,
  gp_rand_coef_data_pred = NULL, cluster_ids_pred = NULL, X_pred = NULL,
  use_saved_data = FALSE, offset = NULL, offset_pred = NULL,
  fixed_effects = NULL, fixed_effects_pred = NULL,
  vecchia_pred_type = NULL, num_neighbors_pred = NULL, ...)
```

Arguments

object	a GPModel
predict_response	A boolean. If TRUE, the response variable (label) is predicted, otherwise the latent random effects
predict_var	A boolean. If TRUE, the (posterior) predictive variances are calculated
predict_cov_mat	A boolean. If TRUE, the (posterior) predictive covariance is calculated in addition to the (posterior) predictive mean
sample_posterior	A boolean. If TRUE, samples from the posterior are drawn
sample_prior	A boolean. If TRUE, samples from the prior are drawn
num_post_samples	A numeric with the number of posterior samples to draw if 'sample_posterior=TRUE'
num_prior_samples	A numeric with the number of prior samples to draw if 'sample_prior=TRUE'
y	Observed data (can be NULL, e.g. when the model has been estimated already and the same data is used for making predictions)
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
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)
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)
offset	A numeric vector with additional fixed effects contributions that are added to the linear predictor (= offset). The length of this vector needs to equal the number of training data points times the number of fixed-effect sets.
offset_pred	A numeric vector with additional fixed effects contributions that are added to the linear predictor for the prediction points (= offset). The length of this vector needs to equal the number of prediction points times the number of fixed-effect sets.
fixed_effects	This is discontinued. Use the renamed equivalent argument offset instead
fixed_effects_pred	This is discontinued. Use the renamed equivalent argument offset_pred instead
vecchia_pred_type	A string specifying the type of Vecchia approximation used for making predictions. This is discontinued here. Use the function 'set_prediction_data' to specify this
num_neighbors_pred	an integer specifying the number of neighbors for making predictions. This is discontinued here. Use the function 'set_prediction_data' to specify this
...	(not used, ignore this, simply here that there is no CRAN warning)

Value

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

- "mu" (first entry): predictive (=posterior) mean. For (generalized) linear mixed effects models, i.e., models with a linear regression term, this consists of the sum of fixed effects and random effects predictions
- "cov" (second entry): predictive (=posterior) covariance matrix. This is NULL if 'predict_cov_mat=FALSE'
- "var" (third entry) : predictive (=posterior) variances. This is NULL if 'predict_var=FALSE'

Author(s)

Fabio Sigrist

Examples

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


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

```
#--------------------Grouped random effects model: single-level random effect----------------
gp_model <- fitGPModel(group_data = group_data[,1], y = y, X = X1,
                       likelihood="gaussian")
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 = "matern", cov_fct_shape = 1.5,
                       likelihood="gaussian", y = y, X = X1)
summary(gp_model)
# Make predictions
pred <- predict(gp_model, gp_coords_pred = coords_test,
                X_pred = X_test1, predict_cov_mat = TRUE)
pred$mu # Predicted (posterior) mean of GP
pred$cov # Predicted (posterior) covariance matrix of GP
```

---

predict_training_data_random_effects

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

---

Description

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

Usage

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

Arguments

gp_model	A GPModel or a gpb.Booster with a GPModel
predict_var	A boolean. If TRUE, the (posterior) predictive variances are calculated
...	Additional arguments passed to methods.

Value

A GPModel

Author(s)

Fabio Sigrist

Examples

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

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

---

predict_training_data_random_effects.GPModel

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

---

Description

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

Usage

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

## S3 method for class 'gpb.Booster'
predict_training_data_random_effects(gp_model,
  predict_var = FALSE, ...)
```

Arguments

gp_model	A GPModel or a gpb.Booster with a GPModel
predict_var	A boolean. If TRUE, the (posterior) predictive variances are calculated
offset	A numeric vector with additional fixed effects contributions that are added to the linear predictor (= offset). The length of this vector needs to equal the number of training data points times the number of fixed-effect sets.
...	Additional arguments passed to methods.

Value

A GPModel

Author(s)

Fabio Sigrist

Examples

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

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

---

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

Description

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

Usage

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

Arguments

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

Value

`gpb.Booster`

Examples

```
library(gpboost)
data(agaricus.train, package = "gpboost")
train <- agaricus.train
dtrain <- gpb.Dataset(train$data, label = train$label)
data(agaricus.test, package = "gpboost")
test <- agaricus.test
dtest <- gpb.Dataset.create.valid(dtrain, test$data, label = test$label)
params <- list(objective = "regression", metric = "l2")
valids <- list(test = dtest)
model <- gpb.train(
```

```
  params = params
  , data = dtrain
  , nrounds = 10L
  , valids = valids
  , min_data = 1L
  , learning_rate = 1.0
  , early_stopping_rounds = 5L
)
model_file <- tempfile(fileext = ".rds")
saveRDS.gpb.Booster(model, model_file)
new_model <- readRDS.gpb.Booster(model_file)
```

`saveGPModel` *Save a* `GPModel`

Description

Save a `GPModel`

Usage

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

Arguments

`gp_model`	a `GPModel`
`filename`	filename for saving

Value

A `GPModel`

Author(s)

Fabio Sigrist

Examples

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

gp_model <- fitGPModel(group_data = group_data[,1], y = y, X = X1, likelihood="gaussian")
pred <- predict(gp_model, group_data_pred = group_data_test[,1],
                X_pred = X_test1, predict_var = TRUE)
# Save model to file
filename <- tempfile(fileext = ".json")
```

```
saveGPModel(gp_model,filename = filename)
# Load from file and make predictions again
gp_model_loaded <- loadGPModel(filename = filename)
pred_loaded <- predict(gp_model_loaded, group_data_pred = group_data_test[,1],
                       X_pred = X_test1, predict_var = TRUE)
# Check equality
pred$mu - pred_loaded$mu
pred$var - pred_loaded$var
```

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

Description

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

Usage

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

Arguments

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

Value

NULL invisibly.

Examples

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

---

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

Description

Set one attribute of a gpb.Dataset

Usage

```
setinfo(dataset, ...)

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

Arguments

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

Details

The name field can be one of the following:

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

Value

the dataset you passed in

the dataset you passed in

Examples

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

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

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

---

set_optim_params *Set parameters for estimation of the covariance parameters*

Description

Set parameters for optimization of the covariance parameters of a GPModel

Usage

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

Arguments

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

- trace: boolean (default = FALSE). If TRUE, information on the progress of the parameter optimization is printed
- init_cov_pars: vector with numeric elements (default = NULL). Initial values for covariance parameters of Gaussian process and random effects (can be NULL). The order is same as the order of the parameters in the summary function: first is the error variance (only for "gaussian" likelihood), next follow the variances of the grouped random effects (if there are any, in the order provided in 'group_data'), and then follow the marginal variance and the ranges of the Gaussian process. If there are multiple Gaussian processes, then the variances and ranges follow alternatingly. If 'init_cov_pars = NULL', an internal choice is used that depends on the likelihood and the random effects type and covariance function. If you select the option 'trace = TRUE' in the 'params' argument, you will see the first initial covariance parameters in iteration 0.
- init_coef: vector with numeric elements (default = NULL). Initial values for the regression coefficients (if there are any, can be NULL)
- init_aux_pars: vector with numeric elements (default = NULL). Initial values for additional parameters for non-Gaussian likelihoods (e.g., shape parameter of a gamma or negative_binomial likelihood)
- init_coef_aux_pars_from_iid_model: boolean (default = TRUE). If TRUE, regression coefficients and auxiliary parameters are initialized from an iid model (only for models with a linear regression term). This option is ignored if init_coef is provided. If init_aux_pars is provided but init_coef is not, only regression coefficients are initialized from an iid model. For likelihood = "gaussian_heteroscedastic_fixed_and_random", if init_cov_pars is not provided, the initial covariance parameters are obtained from a homoscedastic Gaussian process model, and the regression coefficients are then initialized from this model instead: the coefficients of the mean from its coefficients and the intercept of the log-error variance from the logarithm of its error variance.
- estimate_cov_par_index: vector with integer (default = -1). This allows for disabling the estimation of some (or all) covariance parameters. If 'estimate_cov_par_index' = -1, all covariance parameters are estimated. If estimate_cov_par_index != -1, this should be a vector with length equal to the number of covariance parameters, and estimate_cov_par_index[i] should be of bool type indicating whether parameter number i is estimated or not. For instance, estimate_cov_par_index = c(1,1,0) means that the first two covariance parameters are estimated and the last one not. Parameters that are not estimated are kept at their initial values (see 'init_cov_pars'). They are treated as known constants when calculating the standard errors of the estimated parameters, and their own standard errors are NaN.
- estimate_aux_pars: boolean (default = TRUE). If TRUE, additional parameters for non-Gaussian likelihoods are also estimated (e.g., shape parameter of a gamma or negative_binomial likelihood)

- optimizer_cov: `string` (default = "lbfgs"). Optimizer used for estimating covariance parameters. Options: "lbfgs", "gradient_descent", "fisher_scoring", "newton", "nelder_mead". If there are additional auxiliary parameters for non-Gaussian likelihoods, 'optimizer_cov' is also used for those
- optimizer_coef: `string` (default = "wls" for Gaussian likelihoods and "lbfgs" for other likelihoods). Optimizer used for estimating linear regression coefficients, if there are any (for the GPBoost algorithm there are usually none). Options: "gradient_descent", "lbfgs", "wls", "nelder_mead". Gradient descent steps are done simultaneously with gradient descent steps for the covariance parameters. "wls" refers to doing coordinate descent for the regression coefficients using weighted least squares. If 'optimizer_cov' is set to "nelder_mead" or "lbfgs", 'optimizer_coef' is automatically also set to the same value.
- maxit: `integer` (default = 1000). Maximal number of iterations for optimization algorithm. If maxit = -999, internal default values are used
- delta_rel_conv: `numeric` (default = 1E-6 except for "nelder_mead" for which the default is 1E-8). Convergence tolerance. The algorithm stops if the relative change in either the (approximate) log-likelihood or the parameters is below this value. If delta_rel_conv = -999, internal default values are used
- cg_max_num_it: `integer` (default = 1000). Maximal number of iterations for conjugate gradient algorithms. If cg_max_num_it = -999, internal default values are used
- cg_max_num_it_tridiag: `integer` (default = 1000). Maximal number of iterations for conjugate gradient algorithm when being run as Lanczos algorithm for tridiagonalization. If cg_max_num_it_tridiag = -999, internal default values are used
- cg_delta_conv: `numeric` (default = 1E-2). Tolerance level for L2 norm of residuals for checking convergence in conjugate gradient algorithm when being used for parameter estimation. This is the tolerance of the "absolute" criterion only, the "relative" criterion is controlled by cg_rel_tol and cg_abs_tol. If cg_delta_conv = -999, internal default values are used
- cg_convergence_criterion: `string` (default = "absolute"). Stopping rule of conjugate gradient algorithms.
  - "absolute": stop when ||r||_2 < cg_delta_conv
  - "relative": stop when ||r||_2 <= max(cg_abs_tol, cg_rel_tol * ||b||_2), where b is the right-hand side of the linear system. The cg_abs_tol floor makes this robust for right-hand sides that are zero or very small
- cg_rel_tol: `numeric` (default = 1E-6). Relative tolerance of the "relative" stopping rule. If cg_rel_tol = -999, internal default values are used
- cg_abs_tol: `numeric` (default = 1E-8). Absolute tolerance floor of the "relative" stopping rule, which makes it robust for right-hand sides that are zero or very small. This is independent of cg_delta_conv, so that the two criteria can be varied separately. If cg_abs_tol = -999, internal default values are used
- cg_multi_rhs_convergence: `string` (default = "average"). How the stopping rule is aggregated over the columns of a linear system with several

right-hand sides, such as the stochastic Lanczos quadrature. For cg_convergence_criterion = "relative", every right-hand side is normalized by its own norm, i.e. q_j = ||r_j||_2 / max(cg_abs_tol, cg_rel_tol * ||b_j||_2):

  - "average": stop when mean(q_j) <= 1
  - "max": stop when max(q_j) <= 1
  - "per_rhs": stop iterating on column j individually as soon as q_j <= 1

For cg_convergence_criterion = "absolute" there is no per-column normalization:

  - "average": stop when mean(||r_j||_2) < cg_delta_conv. This is the historic behavior and the default
  - "max": stop when ||r_j||_2 < cg_delta_conv for all j
  - "per_rhs": stop iterating on column j as soon as ||r_j||_2 < cg_delta_conv

- num_rand_vec_trace: `integer` (default = 50). Number of random vectors (e.g., Rademacher) for stochastic approximation of the trace of a matrix. If num_rand_vec_trace = -999, internal default values are used
- reuse_rand_vec_trace: `boolean` (default = TRUE). If true, random vectors (e.g., Rademacher) for stochastic approximations of the trace of a matrix are sampled only once at the beginning of the parameter estimation and reused in later trace approximations. Otherwise they are sampled every time a trace is calculated
- seed_rand_vec_trace: `integer` (default = 1). Seed number to generate random vectors (e.g., Rademacher)
- cg_preconditioner_type (`string`): Type of preconditioner used for conjugate gradient algorithms.
  - Options for grouped random effects:
    * "ssor" (= default): SSOR preconditioner
    * "incomplete_cholesky": zero fill-in incomplete Cholesky factorization
  - Options for likelihood != "gaussian" and gp_approx == "vecchia" or likelihood == "gaussian" and gp_approx == "vecchia_latent":
    * "vadu" (= default): (B^T * (D^-1 + W) * B) as preconditioner for inverting (B^T * D^-1 * B + W), where B^T * D^-1 * B approx= Sigma^-1
    * "fitc": FITC / modified predictive process preconditioner for inverting (B^-1 * D * B^-T + W^-1)
    * "pivoted_cholesky": (Lk * Lk^T + W^-1) as preconditioner for inverting (B^-1 * D * B^-T + W^-1), where Lk is a low-rank pivoted Cholesky approximation for Sigma and B^-1 * D * B^-T approx= Sigma
    * "incomplete_cholesky": zero fill-in incomplete (reverse) Cholesky factorization of (B^T * D^-1 * B + W) using the sparsity pattern of B^T * D^-1 * B approx= Sigma^-1
  - Options for likelihood != "gaussian" and gp_approx == "full_scale_vecchia":
    * "fitc" (= default): FITC / modified predictive process preconditioner
    * "vifdu": VIF with diagonal update preconditioner

  - Options for likelihood == "gaussian" and gp_approx == "full_scale_tapering":
    * "fitc" (= default): modified predictive process preconditioner
    * "none": no preconditioner
- fitc_piv_chol_preconditioner_rank (`integer`): Rank of the FITC and pivoted Cholesky decomposition preconditioners for iterative methods for Vecchia and VIF approximations (for full_scale_tapering, the same inducing points as in the approximation as used). If fitc_piv_chol_preconditioner_rank = -999, internal default values are used:
  - 200 for the FITC preconditioner
  - 50 for the pivoted Cholesky decomposition preconditioner
- convergence_criterion: `string` (default = "relative_change_in_log_likelihood", only relevant for "gradient_descent", "fisher_scoring", and "newton"). The convergence criterion used for terminating the optimization algorithm. Options: "relative_change_in_log_likelihood" or "relative_change_in_parameters". If convergence_criterion = "default", internal default values are used
- lr_cov: `numeric` (default = 0.1 for "gradient_descent" and 1. otherwise, relevant for "gradient_descent", "fisher_scoring", "newton", and "lbfgs"). Initial learning rate for covariance parameters if a gradient-based optimization method is used
  - If lr_cov = -999, internal default values are used (0.1 for "gradient_descent" and 1. otherwise)
  - If there are additional auxiliary parameters for non-Gaussian likelihoods, 'lr_cov' is also used for those
  - For "gradient_descent", "fisher_scoring", and "newton", this is the learning rate by which the search direction is multiplied. It can be decreased, and increased again, during the optimization
  - For "lbfgs" and "lbfgs_linesearch_nocedal_wright", this is not a learning rate by which every update is multiplied. It only determines the first trial step, which is the negative gradient divided by its norm and multiplied by 'lr_cov', i.e., a step of length 'lr_cov'. The subsequent steps are determined by the approximate Hessian of the L-BFGS algorithm and the line search. 'lr_cov' has no effect when the approximate Hessian of a previous call is reused (see 'reuse_learning_rates_gp_model' of the GPBoost algorithm)
- lr_coef: `numeric` (default = 0.1, only relevant for "gradient_descent", "fisher_scoring", and "newton"). Learning rate for fixed effect regression coefficients if gradient descent is used. If lr_coef = -999, internal default values are used
- use_nesterov_acc: `boolean` (default = TRUE, only relevant for "gradient_descent"). If TRUE Nesterov acceleration is used. This is used only for gradient descent
- nesterov_schedule_version: `integer` (Default = 0, only relevant for "gradient_descent"). Which version of Nesterov schedule should be used. If nesterov_schedule_version = -999, internal default values are used.
- acc_rate_coef: `numeric` (default = 0.5, only relevant for "gradient_descent"). Acceleration rate for regression coefficients (if there are any) for Nesterov acceleration. If acc_rate_coef = -999, internal default values are used

- acc_rate_cov: `numeric` (default = 0.5, only relevant for "gradient_descent"). Acceleration rate for covariance parameters for Nesterov acceleration. If acc_rate_cov = -999, internal default values are used
- momentum_offset: `integer` (Default = 2, only relevant for "gradient_descent"). Number of iterations for which no momentum is applied in the beginning. If momentum_offset = -999, internal default values are used.
- m_lbfgs: `integer` (Default = 6). Number of corrections to approximate the inverse Hessian matrix for the "lbfgs" optimizer. If m_lbfgs = -999, internal default values are used
- cold_restart_lbfgs: `boolean` (Default = TRUE). If TRUE, restarts of the "lbfgs" optimizers are "cold" restarts: the regression coefficients, the auxiliary parameters, and the modes of the Laplace approximations are reset to their initial values and only the covariance parameters are kept. Such restarts can escape a local optimum in which the optimizer is stuck. If FALSE, "warm" restarts are done: the optimization simply continues from the current parameters with a re-initialized approximate Hessian. Only relevant if `max_num_restarts_lbfgs` > 0. Note that every cold restart is a full optimization with up to `maxit` iterations, and the number of iterations reported is the sum over all restarts
- max_num_restarts_lbfgs: `integer` (Default = 0). Maximal number of restarts of the "lbfgs" optimizers after they have terminated. A restart continues from the current parameters with a re-initialized approximate Hessian. What else is reset depends on `cold_restart_lbfgs`. Restarts are only done as long as the negative log-likelihood still decreases by more than `delta_rel_conv` and as long as the total number of iterations is below `maxit`. This can help when the line search of "lbfgs" fails and the optimizer thus terminates prematurely without having converged. This can happen, e.g., for the `"asymmetric_laplace"` (= quantile regression) likelihood, whose approximate marginal likelihood is not smooth
- reset_modes_line_search_lbfgs: `boolean` (Default = FALSE). If TRUE, the mode finding of the Laplace approximation starts at the modes of the current parameters for every trial point of the line search of the "lbfgs" optimizer, and not at the modes of the previous trial point. Only relevant for non-Gaussian likelihoods and `optimizer_cov` = `"lbfgs"` or `"lbfgs_linesearch_nocedal_wright`
- delta_conv_mode_finding: `numeric` (Default = 1E-8, and 1E-10 for the likelihood "gaussian_heteroscedastic_fixed_and_random"). Convergence tolerance in mode finding algorithm for Laplace approximation for non-Gaussian likelihoods. If delta_conv_mode_finding = -999, internal default values are used

Author(s)

Fabio Sigrist

Examples

```
data(GPBoost_data, package = "gpboost")
gp_model <- GPModel(group_data = group_data, likelihood="gaussian")
```

```
set_optim_params(gp_model, params=list(optimizer_cov="nelder_mead"))
```

`set_optim_params.GPModel` *Set parameters for estimation of the covariance parameters*

Description

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

Usage

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

Arguments

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

- trace: `boolean` (default = FALSE). If TRUE, information on the progress of the parameter optimization is printed
- init_cov_pars: `vector` with `numeric` elements (default = NULL). Initial values for covariance parameters of Gaussian process and random effects (can be NULL). The order is same as the order of the parameters in the summary function: first is the error variance (only for "gaussian" likelihood), next follow the variances of the grouped random effects (if there are any, in the order provided in 'group_data'), and then follow the marginal variance and the ranges of the Gaussian process. If there are multiple Gaussian processes, then the variances and ranges follow alternatingly. If 'init_cov_pars = NULL', an internal choice is used that depends on the likelihood and the random effects type and covariance function. If you select the option 'trace = TRUE' in the 'params' argument, you will see the first initial covariance parameters in iteration 0.
- init_coef: `vector` with `numeric` elements (default = NULL). Initial values for the regression coefficients (if there are any, can be NULL)
- init_aux_pars: `vector` with `numeric` elements (default = NULL). Initial values for additional parameters for non-Gaussian likelihoods (e.g., shape parameter of a gamma or negative_binomial likelihood)
- init_coef_aux_pars_from_iid_model: `boolean` (default = TRUE). If TRUE, regression coefficients and auxiliary parameters are initialized from an iid model (only for models with a linear regression term). This option is ignored if init_coef is provided. If init_aux_pars is provided but init_coef is not, only regression coefficients are initialized from an iid model. For likelihood = "gaussian_heteroscedastic_fixed_and_random", if init_cov_pars

is not provided, the initial covariance parameters are obtained from a homoscedastic Gaussian process model, and the regression coefficients are then initialized from this model instead: the coefficients of the mean from its coefficients and the intercept of the log-error variance from the logarithm of its error variance.

- estimate_cov_par_index: `vector` with `integer` (default = -1). This allows for disabling the estimation of some (or all) covariance parameters. If 'estimate_cov_par_index' = -1, all covariance parameters are estimated. If estimate_cov_par_index != -1, this should be a vector with length equal to the number of covariance parameters, and estimate_cov_par_index[i] should be of bool type indicating whether parameter number i is estimated or not. For instance, estimate_cov_par_index = c(1,1,0) means that the first two covariance parameters are estimated and the last one not. Parameters that are not estimated are kept at their initial values (see 'init_cov_pars'). They are treated as known constants when calculating the standard errors of the estimated parameters, and their own standard errors are NaN.
- estimate_aux_pars: `boolean` (default = TRUE). If TRUE, additional parameters for non-Gaussian likelihoods are also estimated (e.g., shape parameter of a gamma or negative_binomial likelihood)
- optimizer_cov: `string` (default = "lbfgs"). Optimizer used for estimating covariance parameters. Options: "lbfgs", "gradient_descent", "fisher_scoring", "newton", "nelder_mead". If there are additional auxiliary parameters for non-Gaussian likelihoods, 'optimizer_cov' is also used for those
- optimizer_coef: `string` (default = "wls" for Gaussian likelihoods and "lbfgs" for other likelihoods). Optimizer used for estimating linear regression coefficients, if there are any (for the GPBoost algorithm there are usually none). Options: "gradient_descent", "lbfgs", "wls", "nelder_mead". Gradient descent steps are done simultaneously with gradient descent steps for the covariance parameters. "wls" refers to doing coordinate descent for the regression coefficients using weighted least squares. If 'optimizer_cov' is set to "nelder_mead" or "lbfgs", 'optimizer_coef' is automatically also set to the same value.
- maxit: `integer` (default = 1000). Maximal number of iterations for optimization algorithm. If maxit = -999, internal default values are used
- delta_rel_conv: `numeric` (default = 1E-6 except for "nelder_mead" for which the default is 1E-8). Convergence tolerance. The algorithm stops if the relative change in either the (approximate) log-likelihood or the parameters is below this value. If delta_rel_conv = -999, internal default values are used
- cg_max_num_it: `integer` (default = 1000). Maximal number of iterations for conjugate gradient algorithms. If cg_max_num_it = -999, internal default values are used
- cg_max_num_it_tridiag: `integer` (default = 1000). Maximal number of iterations for conjugate gradient algorithm when being run as Lanczos algorithm for tridiagonalization. If cg_max_num_it_tridiag = -999, internal default values are used
- cg_delta_conv: `numeric` (default = 1E-2). Tolerance level for L2 norm of residuals for checking convergence in conjugate gradient algorithm when

being used for parameter estimation. This is the tolerance of the "absolute" criterion only, the "relative" criterion is controlled by cg_rel_tol and cg_abs_tol. If cg_delta_conv = -999, internal default values are used

- cg_convergence_criterion: `string` (default = "absolute"). Stopping rule of conjugate gradient algorithms.
  - "absolute": stop when ||r||_2 < cg_delta_conv
  - "relative": stop when ||r||_2 <= max(cg_abs_tol, cg_rel_tol * ||b||_2), where b is the right-hand side of the linear system. The cg_abs_tol floor makes this robust for right-hand sides that are zero or very small
- cg_rel_tol: `numeric` (default = 1E-6). Relative tolerance of the "relative" stopping rule. If cg_rel_tol = -999, internal default values are used
- cg_abs_tol: `numeric` (default = 1E-8). Absolute tolerance floor of the "relative" stopping rule, which makes it robust for right-hand sides that are zero or very small. This is independent of cg_delta_conv, so that the two criteria can be varied separately. If cg_abs_tol = -999, internal default values are used
- cg_multi_rhs_convergence: `string` (default = "average"). How the stopping rule is aggregated over the columns of a linear system with several right-hand sides, such as the stochastic Lanczos quadrature. For cg_convergence_criterion = "relative", every right-hand side is normalized by its own norm, i.e. q_j = ||r_j||_2 / max(cg_abs_tol, cg_rel_tol * ||b_j||_2):
  - "average": stop when mean(q_j) <= 1
  - "max": stop when max(q_j) <= 1
  - "per_rhs": stop iterating on column j individually as soon as q_j <= 1

  For cg_convergence_criterion = "absolute" there is no per-column normalization:
  - "average": stop when mean(||r_j||_2) < cg_delta_conv. This is the historic behavior and the default
  - "max": stop when ||r_j||_2 < cg_delta_conv for all j
  - "per_rhs": stop iterating on column j as soon as ||r_j||_2 < cg_delta_conv
- num_rand_vec_trace: `integer` (default = 50). Number of random vectors (e.g., Rademacher) for stochastic approximation of the trace of a matrix. If num_rand_vec_trace = -999, internal default values are used
- reuse_rand_vec_trace: `boolean` (default = TRUE). If true, random vectors (e.g., Rademacher) for stochastic approximations of the trace of a matrix are sampled only once at the beginning of the parameter estimation and reused in later trace approximations. Otherwise they are sampled every time a trace is calculated
- seed_rand_vec_trace: `integer` (default = 1). Seed number to generate random vectors (e.g., Rademacher)
- cg_preconditioner_type (`string`): Type of preconditioner used for conjugate gradient algorithms.
  - Options for grouped random effects:
    * "ssor" (= default): SSOR preconditioner
    * "incomplete_cholesky": zero fill-in incomplete Cholesky factorization

  - Options for likelihood != "gaussian" and gp_approx == "vecchia" or likelihood == "gaussian" and gp_approx == "vecchia_latent":
    * "vadu" (= default): (B^T * (D^-1 + W) * B) as preconditioner for inverting (B^T * D^-1 * B + W), where B^T * D^-1 * B approx= Sigma^-1
    * "fitc": FITC / modified predictive process preconditioner for inverting (B^-1 * D * B^-T + W^-1)
    * "pivoted_cholesky": (Lk * Lk^T + W^-1) as preconditioner for inverting (B^-1 * D * B^-T + W^-1), where Lk is a low-rank pivoted Cholesky approximation for Sigma and B^-1 * D * B^-T approx= Sigma
    * "incomplete_cholesky": zero fill-in incomplete (reverse) Cholesky factorization of (B^T * D^-1 * B + W) using the sparsity pattern of B^T * D^-1 * B approx= Sigma^-1
  - Options for likelihood != "gaussian" and gp_approx == "full_scale_vecchia":
    * "fitc" (= default): FITC / modified predictive process preconditioner
    * "vifdu": VIF with diagonal update preconditioner
  - Options for likelihood == "gaussian" and gp_approx == "full_scale_tapering":
    * "fitc" (= default): modified predictive process preconditioner
    * "none": no preconditioner
- fitc_piv_chol_preconditioner_rank (`integer`): Rank of the FITC and pivoted Cholesky decomposition preconditioners for iterative methods for Vecchia and VIF approximations (for full_scale_tapering, the same inducing points as in the approximation as used). If fitc_piv_chol_preconditioner_rank = -999, internal default values are used:
  - 200 for the FITC preconditioner
  - 50 for the pivoted Cholesky decomposition preconditioner
- convergence_criterion: `string` (default = "relative_change_in_log_likelihood", only relevant for "gradient_descent", "fisher_scoring", and "newton"). The convergence criterion used for terminating the optimization algorithm. Options: "relative_change_in_log_likelihood" or "relative_change_in_parameters". If convergence_criterion = "default", internal default values are used
- lr_cov: `numeric` (default = 0.1 for "gradient_descent" and 1. otherwise, relevant for "gradient_descent", "fisher_scoring", "newton", and "lbfgs"). Initial learning rate for covariance parameters if a gradient-based optimization method is used
  - If lr_cov = -999, internal default values are used (0.1 for "gradient_descent" and 1. otherwise)
  - If there are additional auxiliary parameters for non-Gaussian likelihoods, 'lr_cov' is also used for those
  - For "gradient_descent", "fisher_scoring", and "newton", this is the learning rate by which the search direction is multiplied. It can be decreased, and increased again, during the optimization
  - For "lbfgs" and "lbfgs_linesearch_nocedal_wright", this is not a learning rate by which every update is multiplied. It only determines the first

trial step, which is the negative gradient divided by its norm and multiplied by 'lr_cov', i.e., a step of length 'lr_cov'. The subsequent steps are determined by the approximate Hessian of the L-BFGS algorithm and the line search. 'lr_cov' has no effect when the approximate Hessian of a previous call is reused (see 'reuse_learning_rates_gp_model' of the GPBoost algorithm)

- lr_coef: `numeric` (default = 0.1, only relevant for "gradient_descent", "fisher_scoring", and "newton"). Learning rate for fixed effect regression coefficients if gradient descent is used. If lr_coef = -999, internal default values are used
- use_nesterov_acc: `boolean` (default = TRUE, only relevant for "gradient_descent"). If TRUE Nesterov acceleration is used. This is used only for gradient descent
- nesterov_schedule_version: `integer` (Default = 0, only relevant for "gradient_descent"). Which version of Nesterov schedule should be used. If nesterov_schedule_version = -999, internal default values are used.
- acc_rate_coef: `numeric` (default = 0.5, only relevant for "gradient_descent"). Acceleration rate for regression coefficients (if there are any) for Nesterov acceleration. If acc_rate_coef = -999, internal default values are used
- acc_rate_cov: `numeric` (default = 0.5, only relevant for "gradient_descent"). Acceleration rate for covariance parameters for Nesterov acceleration. If acc_rate_cov = -999, internal default values are used
- momentum_offset: `integer` (Default = 2, only relevant for "gradient_descent"). Number of iterations for which no momentum is applied in the beginning. If momentum_offset = -999, internal default values are used.
- m_lbfgs: `integer` (Default = 6). Number of corrections to approximate the inverse Hessian matrix for the "lbfgs" optimizer. If m_lbfgs = -999, internal default values are used
- cold_restart_lbfgs: `boolean` (Default = TRUE). If TRUE, restarts of the "lbfgs" optimizers are "cold" restarts: the regression coefficients, the auxiliary parameters, and the modes of the Laplace approximations are reset to their initial values and only the covariance parameters are kept. Such restarts can escape a local optimum in which the optimizer is stuck. If FALSE, "warm" restarts are done: the optimization simply continues from the current parameters with a re-initialized approximate Hessian. Only relevant if `max_num_restarts_lbfgs` > 0. Note that every cold restart is a full optimization with up to `maxit` iterations, and the number of iterations reported is the sum over all restarts
- max_num_restarts_lbfgs: `integer` (Default = 0). Maximal number of restarts of the "lbfgs" optimizers after they have terminated. A restart continues from the current parameters with a re-initialized approximate Hessian. What else is reset depends on `cold_restart_lbfgs`. Restarts are only done as long as the negative log-likelihood still decreases by more than `delta_rel_conv` and as long as the total number of iterations is below `maxit`. This can help when the line search of "lbfgs" fails and the optimizer thus terminates prematurely without having converged. This can happen, e.g., for the `"asymmetric_laplace"` (= quantile regression) likelihood, whose approximate marginal likelihood is not smooth

- reset_modes_line_search_lbfgs: boolean (Default = FALSE). If TRUE, the mode finding of the Laplace approximation starts at the modes of the current parameters for every trial point of the line search of the "lbfgs" optimizer, and not at the modes of the previous trial point. Only relevant for non-Gaussian likelihoods and optimizer_cov = "lbfgs" or "lbfgs_linesearch_nocedal_wright
- delta_conv_mode_finding: numeric (Default = 1E-8, and 1E-10 for the likelihood "gaussian_heteroscedastic_fixed_and_random"). Convergence tolerance in mode finding algorithm for Laplace approximation for non-Gaussian likelihoods. If delta_conv_mode_finding = -999, internal default values are used

Value

A GPModel

Author(s)

Fabio Sigrist

Examples

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

---

set_prediction_data *Set prediction data for a* GPModel

Description

Set the data required for making predictions with a GPModel

Usage

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

Arguments

`gp_model` A `GPModel`

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

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

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

`cg_delta_conv_pred`
a `numeric` specifying the tolerance level for L2 norm of residuals for checking convergence in conjugate gradient algorithms when being used for prediction. This is the tolerance of the "absolute" criterion only, the "relative" criterion is controlled by cg_rel_tol_pred and cg_abs_tol_pred. Default value if NULL: 1e-3

`cg_convergence_criterion_pred`
a `string` specifying the stopping rule of conjugate gradient algorithms when being used for prediction: "absolute" (stop when ||r||_2 < cg_delta_conv_pred) or "relative" (stop when ||r||_2 <= max(cg_abs_tol_pred, cg_rel_tol_pred * ||b||_2)). Default value if NULL: the value of cg_convergence_criterion used for the parameter estimation

`cg_rel_tol_pred`
a `numeric` specifying the relative tolerance of the "relative" stopping rule when being used for prediction. Default value if NULL: the value of cg_rel_tol used for the parameter estimation. This is inherited independently of cg_convergence_criterion_pred

`cg_abs_tol_pred`
a `numeric` specifying the absolute tolerance floor of the "relative" stopping rule when being used for prediction. This makes the rule robust for right-hand sides that are zero or very small. Default value if NULL: the value of cg_abs_tol used for the parameter estimation. This is inherited independently of the other two prediction options

`nsim_var_pred` an `integer` specifying the number of samples when simulation is used for calculating predictive variances Internal default values if NULL:

- 500 for grouped random effects
- 1000 for gp_approx = "vecchia" and gp_approx = "full_scale_tapering"
- 100 for gp_approx = "full_scale_vecchia"

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

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

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

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

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

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

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

Author(s)

Fabio Sigrist

Examples

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

`set_prediction_data.GPModel` *Set prediction data for a* `GPModel`

Description

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

Usage

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

Arguments

`gp_model` A `GPModel`

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

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

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

cg_delta_conv_pred
: a numeric specifying the tolerance level for L2 norm of residuals for checking convergence in conjugate gradient algorithms when being used for prediction. This is the tolerance of the "absolute" criterion only, the "relative" criterion is controlled by cg_rel_tol_pred and cg_abs_tol_pred. Default value if NULL: 1e-3

cg_convergence_criterion_pred
: a string specifying the stopping rule of conjugate gradient algorithms when being used for prediction: "absolute" (stop when ||r||_2 < cg_delta_conv_pred) or "relative" (stop when ||r||_2 <= max(cg_abs_tol_pred, cg_rel_tol_pred * ||b||_2)). Default value if NULL: the value of cg_convergence_criterion used for the parameter estimation

cg_rel_tol_pred
: a numeric specifying the relative tolerance of the "relative" stopping rule when being used for prediction. Default value if NULL: the value of cg_rel_tol used for the parameter estimation. This is inherited independently of cg_convergence_criterion_pred

cg_abs_tol_pred
: a numeric specifying the absolute tolerance floor of the "relative" stopping rule when being used for prediction. This makes the rule robust for right-hand sides that are zero or very small. Default value if NULL: the value of cg_abs_tol used for the parameter estimation. This is inherited independently of the other two prediction options

nsim_var_pred
: an integer specifying the number of samples when simulation is used for calculating predictive variances Internal default values if NULL:
  - 500 for grouped random effects
  - 1000 for gp_approx = "vecchia" and gp_approx = "full_scale_tapering"
  - 100 for gp_approx = "full_scale_vecchia"

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

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

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

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

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

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

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

Value

A GPModel

Author(s)

Fabio Sigrist

Examples

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

---

slice *Slice a dataset*

Description

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

Usage

```
slice(dataset, ...)

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

Arguments

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

Value

constructed sub dataset

Examples

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

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

summary.GPModel *Summary for a* GPModel

Description

Summary for a GPModel

Usage

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

Arguments

object	a GPModel
std_err	A boolean. If TRUE, (approximate) standard errors are calculated (= 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). For non-Gaussian likelihoods, no standard errors of the covariance and auxiliary parameters are calculated if the model is used in the GPBoost algorithm, since they require the fixed effects given by the tree ensemble
...	(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")
summary(gp_model)
```

```
#--------------------Gaussian process model----------------
gp_model <- fitGPModel(gp_coords = coords, cov_function = "matern", cov_fct_shape = 1.5,
                       likelihood="gaussian", y = y, X = X1)
summary(gp_model)
```

X *Predictor variable data for 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 *Test predictor variable data for 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 *Response variable data for 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, 4
agaricus.train, 5
bank, 5
coords, 6
coords_test, 6
GPBoost_data, 88
group_data, 101
group_data_test, 101
X, 135
X_test, 135
y, 135

agaricus.test, 4
agaricus.train, 5

bank, 5
barplot, *67*

coords, 6
coords_test, 6

dim.gpb.Dataset, 6
dimnames.gpb.Dataset, 7
dimnames<-.gpb.Dataset *(*dimnames.gpb.Dataset*)*, 7

fit, 8
fit.GPModel, 13
fitGPModel, 19, *31*, *100*

get_aux_pars, 39
get_aux_pars.GPModel, 40
get_coef, 41
get_coef.GPModel, 42
get_cov_pars, 43
get_cov_pars.GPModel, 43
get_nested_categories, 44
getinfo, 38
gpb.convert_with_rules, 45
gpb.cv, 46, *48*, *59*, *77*, *84*
gpb.Dataset, *49*, 51, *60*, *84*
gpb.Dataset.construct, 52
gpb.Dataset.create.valid, 52
gpb.Dataset.save, 53
gpb.Dataset.set.categorical, 54
gpb.Dataset.set.reference, 54
gpb.dump, 55
gpb.get.default.num.threads, *28*, 56, *75*, *97*
gpb.get.eval.result, 57
gpb.get.num.threads, *56*, 58
gpb.grid.search.tune.parameters, 58
gpb.importance, 62, *67*
gpb.interprete, 63, *68*
gpb.load, 64
gpb.model.dt.tree, 65
gpb.plot.importance, 67
gpb.plot.interpretation, 68
gpb.plot.part.dep.interact, 69
gpb.plot.partial.dependence, 71
gpb.save, 72
gpb.set.default.num.threads, *28*, *56*, 74, *75*, *82*, *97*
gpb.set.num.threads, *74*, 75
gpb.train, 76, *86*
gpb.tune.num.threads, *28*, *56*, *74, 75*, 81, *97*
gpboost, 84
GPBoost_data, 88
GPModel, *56*, *58*, *74, 75*, *81*, 89
group_data, 101
group_data_test, 101

loadGPModel, 102

neg_log_likelihood, 103
neg_log_likelihood.GPModel, 104

predict.gpb.Booster, 105
predict.GPModel, 108

predict_training_data_random_effects, 111
predict_training_data_random_effects.gpb.Booster (predict_training_data_random_effects.GPModel), 112
predict_training_data_random_effects.GPModel, 112

readRDS, *113*
readRDS.gpb.Booster, 113

saveGPModel, 114
saveRDS.gpb.Booster, 115
set_optim_params, 117
set_optim_params.GPModel, 123
set_prediction_data, 128
set_prediction_data.GPModel, 131
setinfo, 116
slice, 133
summary.GPModel, 134

X, 135
X_test, 135

y, 135