Package ‘SeBR’

July 21, 2025

Type Package

Title Semiparametric Bayesian Regression Analysis

Version 1.1.0

Description Monte Carlo sampling algorithms for semiparametric Bayesian regression analysis. These models feature a nonparametric (unknown) transformation of the data paired with widely-used regression models including linear regression, spline regression, quantile regression, and Gaussian processes. The transformation enables broader applicability of these key models, including for real-valued, positive, and compactly-supported data with challenging distributional features. The samplers prioritize computational scalability and, for most cases, Monte Carlo (not MCMC) sampling for greater efficiency. Details of the methods and algorithms are provided in Kowal and Wu (2024) <doi:10.1080/01621459.2024.2395586>.

License MIT + file LICENSE

URL `https://github.com/drkowal/SeBR`, `https://drkowal.github.io/SeBR/`

BugReports `https://github.com/drkowal/SeBR/issues`

Imports graphics, stats

Suggests fields, GpGp, knitr, MASS, plyr, quantreg, rmarkdown, spikeSlabGAM, statmod

VignetteBuilder knitr

Encoding UTF-8

RoxygenNote 7.2.3

NeedsCompilation no

Author Dan Kowal [aut, cre, cph] (ORCID: <`https://orcid.org/0000-0003-0917-3007`>)

Maintainer Dan Kowal <`daniel.r.kowal@gmail.com`>

Repository CRAN

Date/Publication 2025-06-16 22:20:02 UTC

Contents

`all_subsets` *Compute all subsets of a set*

Description

Given a set of variables, compute the inclusion indicators for all possible subsets.

Usage

```
all_subsets(set)
```

Arguments

`set`	the set from which to compute all subsets (e.g., `1:p`)

Value

a data frame where the rows indicate the `2^p` different subsets and the columns indicate inclusion (logical) for each element in that subset

References

Code adapted from <https://www.r-bloggers.com/2012/04/generating-all-subsets-of-a-set/>

`bb` *Bayesian bootstrap posterior sampler for the CDF*

Description

Compute one Monte Carlo draw from the Bayesian bootstrap (BB) posterior distribution of the cumulative distribution function (CDF).

Usage

```
bb(y)
```

Arguments

`y`	the data from which to infer the CDF (preferably sorted)

Details

Assuming the data `y` are iid from an unknown distribution, the Bayesian bootstrap (BB) is a nonparametric model for this distribution. The BB is a limiting case of a Dirichlet process prior (without any hyperparameters) that admits direct Monte Carlo (not MCMC) sampling.

This function computes one draw from the BB posterior distribution for the CDF `Fy`.

Value

a function that can evaluate the sampled CDF at any argument(s)

Note

This code is inspired by `ggdist::weighted_ecdf`.

Examples

```
# Simulate data:
y = rnorm(n = 100)

# One draw from the BB posterior:
Fy = bb(y)

class(Fy) # this is a function
Fy(0) # some example use (for this one draw)
Fy(c(.5, 1.2))

# Plot several draws from the BB posterior distribution:
ys = seq(-3, 3, length.out=1000)
plot(ys, ys, type='n', ylim = c(0,1),
     main = 'Draws from BB posterior', xlab = 'y', ylab = 'F(y)')
for(s in 1:50) lines(ys, bb(y)(ys), col='gray')

# Add ECDF for reference:
lines(ys, ecdf(y)(ys), lty=2)
```

---

bgp_bc *Bayesian Gaussian processes with a Box-Cox transformation*

Description

MCMC sampling for Bayesian Gaussian process regression with a (known or unknown) Box-Cox transformation.

Usage

```
bgp_bc(
  y,
  locs,
  X = NULL,
  covfun_name = "matern_isotropic",
  locs_test = locs,
  X_test = NULL,
  nn = 30,
  emp_bayes = TRUE,
  lambda = NULL,
  sample_lambda = TRUE,
  nsave = 1000,
  nburn = 1000,
  nskip = 0
)
```

Arguments

`y`	`n x 1` response vector
`locs`	`n x d` matrix of locations
`X`	`n x p` design matrix; if unspecified, use intercept only
`covfun_name`	string name of a covariance function; see ?GpGp
`locs_test`	`n_test x d` matrix of locations at which predictions are needed; default is `locs`
`X_test`	`n_test x p` design matrix for test data; default is `X`
`nn`	number of nearest neighbors to use; default is 30 (larger values improve the approximation but increase computing cost)
`emp_bayes`	logical; if TRUE, use a (faster!) empirical Bayes approach for estimating the mean function
`lambda`	Box-Cox transformation; if NULL, estimate this parameter
`sample_lambda`	logical; if TRUE, sample lambda, otherwise use the fixed value of lambda above or the MLE (if lambda unspecified)
`nsave`	number of MCMC iterations to save
`nburn`	number of MCMC iterations to discard
`nskip`	number of MCMC iterations to skip between saving iterations, i.e., save every (nskip + 1)th draw

Details

This function provides Bayesian inference for transformed Gaussian processes. The transformation is parametric from the Box-Cox family, which has one parameter `lambda`. That parameter may be fixed in advanced or learned from the data. For computational efficiency, the Gaussian process parameters are fixed at point estimates, and the latent Gaussian process is only sampled when `emp_bayes` = FALSE.

Value

a list with the following elements:

- `coefficients` the posterior mean of the regression coefficients
- `fitted.values` the posterior predictive mean at the test points `locs_test`
- `fit_gp` the fitted `GpGp_fit` object, which includes covariance parameter estimates and other model information
- `post_ypred`: `nsave x n_test` samples from the posterior predictive distribution at `locs_test`
- `post_g`: `nsave` posterior samples of the transformation evaluated at the unique `y` values
- `post_lambda` `nsave` posterior samples of lambda
- `model`: the model fit (here, `bgp_bc`)

as well as the arguments passed in.

Note

Box-Cox transformations may be useful in some cases, but in general we recommend the nonparametric transformation (with Monte Carlo, not MCMC sampling) in `sbgp`.

Examples

```
# Simulate some data:
n = 200 # sample size
x = seq(0, 1, length = n) # observation points

# Transform a noisy, periodic function:
y = g_inv_bc(
  sin(2*pi*x) + sin(4*pi*x) + rnorm(n, sd = .5),
             lambda = .5) # Signed square-root transformation

# Package we use for fast computing w/ Gaussian processes:
library(GpGp)

# Fit a Bayesian Gaussian process with Box-Cox transformation:
fit = bgp_bc(y = y, locs = x)
names(fit) # what is returned
coef(fit) # estimated regression coefficients (here, just an intercept)
class(fit$fit_gp) # the GpGp object is also returned
round(quantile(fit$post_lambda), 3) # summary of unknown Box-Cox parameter

# Plot the model predictions (point and interval estimates):
pi_y = t(apply(fit$post_ypred, 2, quantile, c(0.05, .95))) # 90% PI
plot(x, y, type='n', ylim = range(pi_y,y),
     xlab = 'x', ylab = 'y', main = paste('Fitted values and prediction intervals'))
polygon(c(x, rev(x)),c(pi_y[,2], rev(pi_y[,1])),col='gray', border=NA)
lines(x, y, type='p')
lines(x, fitted(fit), lwd = 3)
```

`blm_bc` *Bayesian linear model with a Box-Cox transformation*

Description

MCMC sampling for Bayesian linear regression with a (known or unknown) Box-Cox transformation. A g-prior is assumed for the regression coefficients.

Usage

```
blm_bc(
  y,
  X,
  X_test = X,
```

```
  psi = length(y),
  lambda = NULL,
  sample_lambda = TRUE,
  nsave = 1000,
  nburn = 1000,
  nskip = 0,
  verbose = TRUE
)
```

Arguments

`y`	`n x 1` vector of observed counts
`X`	`n x p` matrix of predictors (no intercept)
`X_test`	`n_test x p` matrix of predictors for test data; default is the observed covariates `X`
`psi`	prior variance (g-prior)
`lambda`	Box-Cox transformation; if NULL, estimate this parameter
`sample_lambda`	logical; if TRUE, sample lambda, otherwise use the fixed value of lambda above or the MLE (if lambda unspecified)
`nsave`	number of MCMC iterations to save
`nburn`	number of MCMC iterations to discard
`nskip`	number of MCMC iterations to skip between saving iterations, i.e., save every (nskip + 1)th draw
`verbose`	logical; if TRUE, print time remaining

Details

This function provides fully Bayesian inference for a transformed linear model via MCMC sampling. The transformation is parametric from the Box-Cox family, which has one parameter `lambda`. That parameter may be fixed in advanced or learned from the data.

Value

a list with the following elements:

- `coefficients` the posterior mean of the regression coefficients
- `fitted.values` the posterior predictive mean at the test points `X_test`
- `post_theta`: `nsave x p` samples from the posterior distribution of the regression coefficients
- `post_ypred`: `nsave x n_test` samples from the posterior predictive distribution at test points `X_test`
- `post_g`: `nsave` posterior samples of the transformation evaluated at the unique `y` values
- `post_lambda` `nsave` posterior samples of lambda
- `post_sigma` `nsave` posterior samples of sigma
- `model`: the model fit (here, `blm_bc`)

as well as the arguments passed in.

Note

Box-Cox transformations may be useful in some cases, but in general we recommend the nonparametric transformation (with Monte Carlo, not MCMC sampling) in `sblm`.

An intercept is automatically added to `X` and `X_test`. The coefficients reported do *not* include this intercept parameter, since it is not identified under more general transformation models (e.g., `sblm`).

Examples

```
# Simulate some data:
dat = simulate_tlm(n = 100, p = 5, g_type = 'step')
y = dat$y; X = dat$X # training data
y_test = dat$y_test; X_test = dat$X_test # testing data

hist(y, breaks = 25) # marginal distribution

# Fit the Bayesian linear model with a Box-Cox transformation:
fit = blm_bc(y = y, X = X, X_test = X_test)
names(fit) # what is returned
round(quantile(fit$post_lambda), 3) # summary of unknown Box-Cox parameter
```

`blm_bc_hs` *Bayesian linear model with a Box-Cox transformation and a horseshoe prior*

Description

MCMC sampling for Bayesian linear regression with 1) a (known or unknown) Box-Cox transformation and 2) a horseshoe prior for the (possibly high-dimensional) regression coefficients.

Usage

```
blm_bc_hs(
  y,
  X,
  X_test = X,
  lambda = NULL,
  sample_lambda = TRUE,
  only_theta = FALSE,
  nsave = 1000,
  nburn = 1000,
  nskip = 0,
  verbose = TRUE
)
```

Arguments

`y`	`n x 1` vector of observed counts
`X`	`n x p` matrix of predictors (no intercept)
`X_test`	`n_test x p` matrix of predictors for test data; default is the observed covariates `X`
`lambda`	Box-Cox transformation; if NULL, estimate this parameter
`sample_lambda`	logical; if TRUE, sample lambda, otherwise use the fixed value of lambda above or the MLE (if lambda unspecified)
`only_theta`	logical; if TRUE, only return posterior draws of the regression coefficients (for speed)
`nsave`	number of MCMC iterations to save
`nburn`	number of MCMC iterations to discard
`nskip`	number of MCMC iterations to skip between saving iterations, i.e., save every (nskip + 1)th draw
`verbose`	logical; if TRUE, print time remaining

Details

This function provides fully Bayesian inference for a transformed linear model via MCMC sampling. The transformation is parametric from the Box-Cox family, which has one parameter `lambda`. That parameter may be fixed in advanced or learned from the data.

The horseshoe prior is especially useful for high-dimensional settings with many (possibly correlated) covariates. This function uses a fast Cholesky-forward/backward sampler when `p < n` and the Bhattacharya et al. (<https://doi.org/10.1093/biomet/asw042>) sampler when `p > n`. Thus, the sampler can scale linear in `n` (for fixed/small `p`) or linear in `p` (for fixed/small `n`).

Value

a list with the following elements:

- `coefficients` the posterior mean of the regression coefficients
- `fitted.values` the posterior predictive mean at the test points `X_test`
- `post_theta`: `nsave x p` samples from the posterior distribution of the regression coefficients
- `post_ypred`: `nsave x n_test` samples from the posterior predictive distribution at test points `X_test`
- `post_g`: `nsave` posterior samples of the transformation evaluated at the unique `y` values
- `post_lambda`: `nsave` posterior samples of lambda
- `post_sigma`: `nsave` posterior samples of sigma
- `model`: the model fit (here, `blm_bc_hs`)

as well as the arguments passed in.

Note

Box-Cox transformations may be useful in some cases, but in general we recommend the nonparametric transformation in `sblm_hs`.

An intercept is automatically added to `X` and `X_test`. The coefficients reported do *not* include this intercept parameter, since it is not identified under more general transformation models (e.g., `sblm_hs`).

Examples

```
# Simulate data from a transformed (sparse) linear model:
dat = simulate_tlm(n = 100, p = 50, g_type = 'step', prop_sig = 0.1)
y = dat$y; X = dat$X # training data

hist(y, breaks = 25) # marginal distribution

# Fit the Bayesian linear model with a Box-Cox transformation & a horseshoe prior:
fit = blm_bc_hs(y = y, X = X, verbose = FALSE)
names(fit) # what is returned
```

`bqr` *Bayesian quantile regression*

Description

MCMC sampling for Bayesian quantile regression. An asymmetric Laplace distribution is assumed for the errors, so the regression models targets the specified quantile. A g-prior is assumed for the regression coefficients.

Usage

```
bqr(
  y,
  X,
  tau = 0.5,
  X_test = X,
  psi = length(y),
  nsave = 1000,
  nburn = 1000,
  nskip = 0,
  verbose = TRUE
)
```

Arguments

`y`	n x 1 vector of observed counts
`X`	n x p matrix of predictors (no intercept)

`tau`	the target quantile (between zero and one)
`X_test`	`n_test` x `p` matrix of predictors for test data; default is the observed covariates `X`
`psi`	prior variance (g-prior)
`nsave`	number of MCMC iterations to save
`nburn`	number of MCMC iterations to discard
`nskip`	number of MCMC iterations to skip between saving iterations, i.e., save every (nskip + 1)th draw
`verbose`	logical; if TRUE, print time remaining

Value

a list with the following elements:

- `coefficients` the posterior mean of the regression coefficients
- `fitted.values` the estimated `tau`th quantile at test points `X_test`
- `post_theta`: `nsave` x `p` samples from the posterior distribution of the regression coefficients
- `post_ypred`: `nsave` x `n_test` samples from the posterior predictive distribution at test points `X_test`
- `post_qtau`: `nsave` x `n_test` samples of the `tau`th conditional quantile at test points `X_test`
- `model`: the model fit (here, `bqr`)

as well as the arguments passed

Note

The asymmetric Laplace distribution is advantageous because it links the regression model (`X%*%theta`) to a pre-specified quantile (`tau`). However, it is often a poor model for observed data, and the semi-parametric version `sbqr` is recommended in general.

An intercept is automatically added to `X` and `X_test`. The coefficients reported do *not* include this intercept parameter.

Examples

```
# Simulate some heteroskedastic data (no transformation):
dat = simulate_tlm(n = 100, p = 5, g_type = 'box-cox', heterosked = TRUE, lambda = 1)
y = dat$y; X = dat$X # training data
y_test = dat$y_test; X_test = dat$X_test # testing data

# Target this quantile:
tau = 0.05

# Fit the Bayesian quantile regression model:
fit = bqr(y = y, X = X, tau = tau, X_test = X_test)
names(fit) # what is returned

# Posterior predictive checks on testing data: empirical CDF
```

```
y0 = sort(unique(y_test))
plot(y0, y0, type='n', ylim = c(0,1),
     xlab='y', ylab='F_y', main = 'Posterior predictive ECDF')
temp = sapply(1:nrow(fit$post_ypred), function(s)
  lines(y0, ecdf(fit$post_ypred[s,])(y0), # ECDF of posterior predictive draws
        col='gray', type ='s'))
lines(y0, ecdf(y_test)(y0),  # ECDF of testing data
     col='black', type = 's', lwd = 3)

# The posterior predictive checks usually do not pass!
# try ?sbqr instead...
```

`bsm_bc` *Bayesian spline model with a Box-Cox transformation*

Description

MCMC sampling for Bayesian spline regression with a (known or unknown) Box-Cox transformation.

Usage

```
bsm_bc(
  y,
  x = NULL,
  x_test = x,
  psi = NULL,
  lambda = NULL,
  sample_lambda = TRUE,
  nsave = 1000,
  nburn = 1000,
  nskip = 0,
  verbose = TRUE
)
```

Arguments

`y`	n x 1 vector of observed counts
`x`	n x 1 vector of observation points; if NULL, assume equally-spaced on [0,1]
`x_test`	`n_test` x 1 vector of testing points; if NULL, assume equal to `x`
`psi`	prior variance (inverse smoothing parameter); if NULL, sample this parameter
`lambda`	Box-Cox transformation; if NULL, estimate this parameter
`sample_lambda`	logical; if TRUE, sample lambda, otherwise use the fixed value of lambda above or the MLE (if lambda unspecified)
`nsave`	number of MCMC iterations to save

`nburn`	number of MCMC iterations to discard
`nskip`	number of MCMC iterations to skip between saving iterations, i.e., save every (nskip + 1)th draw
`verbose`	logical; if TRUE, print time remaining

Details

This function provides fully Bayesian inference for a transformed spline model via MCMC sampling. The transformation is parametric from the Box-Cox family, which has one parameter `lambda`. That parameter may be fixed in advanced or learned from the data.

Value

a list with the following elements:

- `coefficients` the posterior mean of the regression coefficients
- `fitted.values` the posterior predictive mean at the test points `x_test`
- `post_theta`: `nsave` x `p` samples from the posterior distribution of the regression coefficients
- `post_ypred`: `nsave` x `n_test` samples from the posterior predictive distribution at `x_test`
- `post_g`: `nsave` posterior samples of the transformation evaluated at the unique `y` values
- `post_lambda` `nsave` posterior samples of lambda
- `model`: the model fit (here, `sbsm_bc`)

as well as the arguments passed in.

Note

Box-Cox transformations may be useful in some cases, but in general we recommend the nonparametric transformation (with Monte Carlo, not MCMC sampling) in `sbsm`.

Examples

```
# Simulate some data:
n = 100 # sample size
x = sort(runif(n)) # observation points

# Transform a noisy, periodic function:
y = g_inv_bc(
  sin(2*pi*x) + sin(4*pi*x) + rnorm(n, sd = .5),
             lambda = .5) # Signed square-root transformation

# Fit the Bayesian spline model with a Box-Cox transformation:
fit = bsm_bc(y = y, x = x)
names(fit) # what is returned
round(quantile(fit$post_lambda), 3) # summary of unknown Box-Cox parameter

# Plot the model predictions (point and interval estimates):
pi_y = t(apply(fit$post_ypred, 2, quantile, c(0.05, .95))) # 90% PI
plot(x, y, type='n', ylim = range(pi_y,y),
```

```
     xlab = 'x', ylab = 'y', main = paste('Fitted values and prediction intervals'))
polygon(c(x, rev(x)),c(pi_y[,2], rev(pi_y[,1])),col='gray', border=NA)
lines(x, y, type='p')
lines(x, fitted(fit), lwd = 3)
```

`computeTimeRemaining` *Estimate the remaining time in the algorithm*

Description

Estimate the remaining time in the algorithm

Usage

```
computeTimeRemaining(nsi, timer0, nsims, nprints = 2)
```

Arguments

`nsi`	current iteration
`timer0`	initial timer value from `proc.time()[3]`
`nsims`	total number of simulations
`nprints`	total number of printed updates

Value

estimate of remaining time

`concen_hbb` *Posterior sampling algorithm for the HBB concentration hyperparameters*

Description

Compute Monte Carlo draws from the (marginal) posterior distribution of the concentration hyperparameters of the hierarchical Bayesian bootstrap (hbb). The HBB is a nonparametric model for group-specific distributions; each group has a concentration parameter, where larger values encourage more shrinkage toward a common distribution.

Usage

```
concen_hbb(
  groups,
  shape_alphas = NULL,
  rate_alphas = NULL,
  nsave = 1000,
  ngrid = 500
)
```

Arguments

`groups`	the group assignments in the observed data
`shape_alphas`	(optional) shape parameter for the Gamma prior
`rate_alphas`	(optional) rate parameter for the Gamma prior
`nsave`	(optional) number of Monte Carlo simulations
`ngrid`	(optional) number of grid points

Details

The concentration hyperparameters are assigned independent Gamma(`shape_alphas`, `rate_alphas`) priors. This function uses a grid approximation to the marginal posterior with the goal of producing a simple algorithm. Because this is a *marginal* posterior sampler, it can be used with the `hbb` sampler (which conditions on `alphas`) to provide a joint Monte Carlo (not MCMC) sampling algorithm for the concentration hyperparameters, the group-specific CDFs, and the common CDF. Note that diffuse priors on `alphas` tend to put posterior mass on large values, which leads to more aggressive shrinkage toward the common distribution (complete pooling). For moderate shrinkage, we use the default values `shape_alphas = 30*K` and `rate_alphas = 1`, where `K` is the number of groups.

Value

`nsave x K` samples of the concentration hyperparameters corresponding to the `K` groups

References

Oganisian et al. (<https://doi.org/10.1515/ijb-2022-0051>)

Examples

```
# Dimensions:
n = 500 # number of observations
K = 3 # number of groups

# Assign groups w/ unequal probabilities:
ugroups = paste('g', 1:K, sep='') # groups
groups = sample(ugroups,
                size = n,
                replace = TRUE,
                prob = 1:K) # unequally weighted (unnormalized)

# Summarize:
table(groups)/n

# Marginal posterior sampling for alpha:
post_alpha = concen_hbb(groups)

# Summarize: posterior distributions
for(c in 1:K) {
  hist(post_alpha[,c],
       main = paste("Concentration parameter: group", ugroups[c]),
```

```
      xlim = range(post_alpha))
  abline(v = mean(post_alpha[,c]), lwd=3) # posterior mean
}
```

`contract_grid` *Grid contraction*

Description

Contract the grid if the evaluation points exceed some threshold. This removes the corresponding z values. We can add points back to achieve the same (approximate) length.

Usage

```
contract_grid(z, Fz, lower, upper, add_back = TRUE, monotone = TRUE)
```

Arguments

`z`	grid points (ordered)
`Fz`	function evaluated at those grid points
`lower`	lower threshold at which to check Fz
`upper`	upper threshold at which to check Fz
`add_back`	logical; if true, expand the grid to (about) the original size
`monotone`	logical; if true, enforce monotonicity on the expanded grid

Value

a list containing the grid points `z` and the (interpolated) function `Fz` at those points

---

`Fz_fun` *Compute the latent data CDF*

Description

Assuming a Gaussian latent data distribution (given x), compute the CDF on a grid of points

Usage

```
Fz_fun(z, weights = NULL, mean_vec = NULL, sd_vec)
```

Arguments

`z`	vector of points at which the CDF of z is evaluated
`weights`	`n`-dimensional vector of weights; if NULL, assume 1/n
`mean_vec`	`n`-dimensional vector of means; if NULL, assume mean zero
`sd_vec`	`n`-dimensional vector of standard deviations

Value

CDF of z evaluated at `z`

`g_bc` *Box-Cox transformation*

Description

Evaluate the Box-Cox transformation, which is a scaled power transformation to preserve continuity in the index `lambda` at zero. Negative values are permitted.

Usage

```
g_bc(t, lambda)
```

Arguments

`t`	argument(s) at which to evaluate the function
`lambda`	Box-Cox parameter

Value

The evaluation(s) of the Box-Cox function at the given input(s) `t`.

Note

Special cases include the identity transformation (`lambda = 1`), the square-root transformation (`lambda = 1/2`), and the log transformation (`lambda = 0`).

Examples

```
# Log-transformation:
g_bc(1:5, lambda = 0); log(1:5)

# Square-root transformation: note the shift and scaling
g_bc(1:5, lambda = 1/2); sqrt(1:5)
```

`g_fun` *Compute the transformation*

Description

Given the CDFs of z and y, compute a smoothed function to evaluate the transformation

Usage

```
g_fun(y, Fy_eval, z, Fz_eval)
```

Arguments

`y`	vector of points at which the CDF of y is evaluated
`Fy_eval`	CDF of y evaluated at `y`
`z`	vector of points at which the CDF of z is evaluated
`Fz_eval`	CDF of z evaluated at `z`

Value

A smooth monotone function which can be used for evaluations of the transformation.

`g_inv_approx` *Approximate inverse transformation*

Description

Compute the inverse function of a transformation g based on a grid search.

Usage

```
g_inv_approx(g, t_grid)
```

Arguments

`g`	the transformation function
`t_grid`	grid of arguments at which to evaluate the transformation function

Value

A function which can be used for evaluations of the (approximate) inverse transformation function.

`g_inv_bc` *Inverse Box-Cox transformation*

Description

Evaluate the inverse Box-Cox transformation. Negative values are permitted.

Usage

```
g_inv_bc(s, lambda)
```

Arguments

`s`	argument(s) at which to evaluate the function
`lambda`	Box-Cox parameter

Value

The evaluation(s) of the inverse Box-Cox function at the given input(s) `s`.

Note

Special cases include the identity transformation (`lambda = 1`), the square-root transformation (`lambda = 1/2`), and the log transformation (`lambda = 0`).

Examples

```
# (Inverse) log-transformation:
g_inv_bc(1:5, lambda = 0); exp(1:5)

# (Inverse) square-root transformation: note the shift and scaling
g_inv_bc(1:5, lambda = 1/2); (1:5)^2
```

`hbb` *Hierarchical Bayesian bootstrap posterior sampler*

Description

Compute one Monte Carlo draw from the hierarchical Bayesian bootstrap (HBB) posterior distribution of the cumulative distribution function (CDF) for each group. The common (BB) and group-specific (HBB) weights are also returned.

Usage

```
hbb(
  y,
  groups,
  sample_alphas = FALSE,
  shape_alphas = NULL,
  rate_alphas = NULL,
  alphas = NULL,
  M = 30
)
```

Arguments

`y`	the data from which to infer the group-specific CDFs
`groups`	the group assignment for each element of `y`
`sample_alphas`	logical; if TRUE, sample the concentration hyperparameters from their marginal posterior distribution
`shape_alphas`	(optional) shape parameter for the Gamma prior on each `alphas` (if sampled)
`rate_alphas`	(optional) rate parameter for the Gamma prior on each `alphas` (if sampled)
`alphas`	(optional) vector of fixed concentration hyperparameters corresponding to the unique levels in `groups` (used when `sample_alphas = FALSE`)
`M`	a positive scaling term to set a default value of `alphas` when it is unspecified (`alphas = NULL`) and not sampled (`sample_alphas = FALSE`)

Details

Assuming the data `y` are independent with unknown, group-specific distributions, the hierarchical Bayesian bootstrap (HBB) from Oganisian et al. (<https://doi.org/10.1515/ijb-2022-0051>) is a nonparametric model for each distribution. The HBB includes hierarchical shrinkage across these groups toward a common distribution (the `bb`). The HBB admits direct Monte Carlo (not MCMC) sampling.

The shrinkage toward this common distribution is determined by the concentration hyperparameters `alphas`. Each component of `alphas` corresponds to one of the groups. Larger values encourage more shrinkage toward the common distribution, while smaller values allow more substantial deviations for that group.

When `sample_alphas=TRUE`, each component of `alphas` is sampled from its marginal posterior distribution, assuming independent Gamma(`shape_alphas`, `rate_alphas`) priors. This step uses a simple grid approximation to enable efficient sampling that preserves joint Monte Carlo sampling with the group-specific and common distributions. See `concen_hbb` for details. Note that diffuse priors on `alphas` tends to produce more aggressive shrinkage toward the common distribution (complete pooling). For moderate shrinkage, we use the default values `shape_alphas = 30*K` and `rate_alphas = 1` where `K` is the number of groups.

When `sample_alphas=FALSE`, these concentration hyperparameters are fixed at user-specified values. That can be done by specifying `alphas` directly. Alternatively, if `alphas` is left unspecified (`alphas = NULL`), we adopt the default from Oganisian et al. which sets the `c`th entry to `M*n/nc` where `M` is user-specified and `nc` is the number of observations in group `c`. For further guidance on the choice of `M`:

- `M = 0.01/K` approximates separate BB's by group (no pooling);
- `M` between 10 and 100 gives moderate shrinkage (partial pooling); and
- `M = 100*max(nc)` approximates a common BB (complete pooling).

Value

a list with the following elements:

- `Fyc`: a list of functions where each entry corresponds to a group and that group-specific function can evaluate the sampled CDF at any argument(s)
- `weights_y`: sampled weights from the common (BB) distribution (`n`-dimensional)
- `weights_yc`: sampled weights from each of the `K` groups (`K x n`)
- `alphas`: the (fixed or sampled) concentration hyperparameters

Note

If supplying `alphas` with distinct entries, make sure that the groups are ordered properly; these entries should match `sort(unique(groups))`.

References

Oganisian et al. (<https://doi.org/10.1515/ijb-2022-0051>)

Examples

```
# Sample size and number of groups:
n = 500
K = 3

# Define the groups, then assign:
ugroups = paste('g', 1:K, sep='') # groups
groups = sample(ugroups, n, replace = TRUE) # assignments

# Simulate the data: iid normal, then add group-specific features
y = rnorm(n = n) # data
for(g in ugroups)
  y[groups==g] = y[groups==g] + 3*rnorm(1) # group-specific

# One draw from the HBB posterior of the CDF:
samp_hbb = hbb(y, groups)

names(samp_hbb) # items returned
Fyc = samp_hbb$Fyc # list of CDFs
class(Fyc) # this is a list
class(Fyc[[1]]) # each element is a function

c = 1 # try: vary in 1:K
```

```
Fyc[[c]](0) # some example use (for this one draw)
Fyc[[c]](c(.5, 1.2))

# Plot several draws from the HBB posterior distribution:
ys = seq(min(y), max(y), length.out=1000)
plot(ys, ys, type='n', ylim = c(0,1),
     main = 'Draws from HBB posteriors', xlab = 'y', ylab = 'F_c(y)')
for(s in 1:50){ # some draws

  # BB CDF:
  Fy = bb(y)
  lines(ys, Fy(ys), lwd=3) # plot CDF

  # HBB:
  Fyc = hbb(y, groups)$Fyc

  # Plot CDFs by group:
  for(c in 1:K) lines(ys, Fyc[[c]](ys), col=c+1, lwd=3)
}

# For reference, add the ECDFs by group:
for(c in 1:K) lines(ys, ecdf(y[groups==ugroups[c]])(ys), lty=2)

legend('bottomright', c('BB', paste('HBB:', ugroups)), col = 1:(K+1), lwd=3)
```

`plot_pptest` *Plot point and interval predictions on testing data*

Description

Given posterior predictive samples at `X_test`, plot the point and interval estimates and compare to the actual testing data `y_test`.

Usage

```
plot_pptest(post_ypred, y_test, alpha_level = 0.1)
```

Arguments

`post_ypred`	`nsave` x `n_test` samples from the posterior predictive distribution at test points `X_test`
`y_test`	`n_test` testing points
`alpha_level`	alpha-level for prediction intervals

Value

plot of the testing data, point and interval predictions, and a summary of the empirical coverage

Examples

```
# Simulate some data:
dat = simulate_tlm(n = 100, p = 5, g_type = 'step')

# Fit a semiparametric Bayesian linear model:
fit = sblm(y = dat$y, X = dat$X, X_test = dat$X_test)

# Evaluate posterior predictive means and intervals on the testing data:
plot_pptest(fit$post_ypred, dat$y_test,
            alpha_level = 0.10) # coverage should be about 90%
```

`rank_approx` *Rank-based estimation of the linear regression coefficients*

Description

For a transformed Gaussian linear model, compute point estimates of the regression coefficients. This approach uses the ranks of the data and does not require the transformation, but must expand the sample size to `n^2` and thus can be slow.

Usage

```
rank_approx(y, X)
```

Arguments

`y`	`n x 1` response vector
`X`	`n x p` matrix of predictors (should not include an intercept!)

Value

the estimated linear coefficients

Examples

```
# Simulate some data:
dat = simulate_tlm(n = 200, p = 10, g_type = 'step')

# Point estimates for the linear coefficients:
theta_hat = suppressWarnings(
  rank_approx(y = dat$y,
              X = dat$X[,-1]) # remove intercept
) # warnings occur from glm.fit (fitted probabilities 0 or 1)

# Check: correlation with true coefficients
cor(dat$beta_true[-1], # excluding the intercept
    theta_hat)
```

`sampleFastGaussian` *Sample a Gaussian vector using Bhattacharya et al. (2016)*

Description

Sample from N(mu, Sigma) where Sigma = solve(crossprod(Phi) + solve(D)) and mu = Sigma*crossprod(Phi, alpha):

Usage

```
sampleFastGaussian(Phi, Ddiag, alpha)
```

Arguments

`Phi`	`n x p` matrix (of predictors)
`Ddiag`	`p x 1` vector of diagonal components (of prior variance)
`alpha`	`n x 1` vector (of data, scaled by variance)

Value

Draw from N(mu, Sigma), which is `p x 1`, and is computed in `O(n^2*p)`

Note

Assumes D is diagonal, but extensions are available

References

Bhattacharya, Chakraborty, and Mallick (2016, <https://doi.org/10.1093/biomet/asw042>)

`sbgp` *Semiparametric Bayesian Gaussian processes*

Description

Monte Carlo sampling for Bayesian Gaussian process regression with an unknown (nonparametric) transformation.

Usage

```
sbgp(
  y,
  locs,
  X = NULL,
  covfun_name = "matern_isotropic",
  locs_test = locs,
  X_test = NULL,
  nn = 30,
  emp_bayes = TRUE,
  fixedX = (length(y) >= 500),
  approx_g = FALSE,
  nsave = 1000,
  ngrid = 100
)
```

Arguments

`y`	`n x 1` response vector
`locs`	`n x d` matrix of locations
`X`	`n x p` design matrix; if unspecified, use intercept only
`covfun_name`	string name of a covariance function; see ?GpGp
`locs_test`	`n_test x d` matrix of locations at which predictions are needed; default is `locs`
`X_test`	`n_test x p` design matrix for test data; default is `X`
`nn`	number of nearest neighbors to use; default is 30 (larger values improve the approximation but increase computing cost)
`emp_bayes`	logical; if TRUE, use a (faster!) empirical Bayes approach for estimating the mean function
`fixedX`	logical; if TRUE, treat the design as fixed (non-random) when sampling the transformation; otherwise treat covariates as random with an unknown distribution
`approx_g`	logical; if TRUE, apply large-sample approximation for the transformation
`nsave`	number of Monte Carlo simulations
`ngrid`	number of grid points for inverse approximations

Details

This function provides Bayesian inference for a transformed Gaussian process model using Monte Carlo (not MCMC) sampling. The transformation is modeled as unknown and learned jointly with the regression function (unless `approx_g = TRUE`, which then uses a point approximation). This model applies for real-valued data, positive data, and compactly-supported data (the support is automatically deduced from the observed `y` values). The results are typically unchanged whether `laplace_approx` is TRUE/FALSE; setting it to TRUE may reduce sensitivity to the prior, while setting it to FALSE may speed up computations for very large datasets. For computational efficiency, the Gaussian process parameters are fixed at point estimates, and the latent Gaussian process is only sampled when `emp_bayes = FALSE`. However, the uncertainty from this term is often negligible compared to the observation errors, and the transformation serves as an additional layer of robustness. By default, `fixedX` is set to FALSE for smaller datasets (`n < 500`) and TRUE for larger datasets (`n >= 500`).

Value

a list with the following elements:

- `coefficients` the estimated regression coefficients
- `fitted.values` the posterior predictive mean at the test points `locs_test`
- `fit_gp` the fitted `GpGp_fit` object, which includes covariance parameter estimates and other model information
- `post_ypred`: `nsave x ntest` samples from the posterior predictive distribution at `locs_test`
- `post_g`: `nsave` posterior samples of the transformation evaluated at the unique `y` values
- `model`: the model fit (here, sbgp)

as well as the arguments passed in.

Examples

```
# Simulate some data:
n = 200 # sample size
x = seq(0, 1, length = n) # observation points

# Transform a noisy, periodic function:
y = g_inv_bc(
  sin(2*pi*x) + sin(4*pi*x) + rnorm(n),
             lambda = .5) # Signed square-root transformation

# Package we use for fast computing w/ Gaussian processes:
library(GpGp)

# Fit the semiparametric Bayesian Gaussian process:
fit = sbgp(y = y, locs = x)
names(fit) # what is returned
coef(fit) # estimated regression coefficients (here, just an intercept)
class(fit$fit_gp) # the GpGp object is also returned

# Plot the model predictions (point and interval estimates):
pi_y = t(apply(fit$post_ypred, 2, quantile, c(0.05, .95))) # 90% PI
plot(x, y, type='n', ylim = range(pi_y,y),
     xlab = 'x', ylab = 'y', main = paste('Fitted values and prediction intervals'))
polygon(c(x, rev(x)),c(pi_y[,2], rev(pi_y[,1])),col='gray', border=NA)
lines(x, y, type='p') # observed points
lines(x, fitted(fit), lwd = 3) # fitted curve
```

`sblm` *Semiparametric Bayesian linear model*

Description

Monte Carlo sampling for Bayesian linear regression with an unknown (nonparametric) transformation. A g-prior is assumed for the regression coefficients.

Usage

```
sblm(
  y,
  X,
  X_test = X,
  psi = length(y),
  laplace_approx = TRUE,
  fixedX = (length(y) >= 500),
  approx_g = FALSE,
  nsave = 1000,
  ngrid = 100,
  verbose = TRUE
)
```

Arguments

`y`	`n` x `1` response vector
`X`	`n` x `p` matrix of predictors (no intercept)
`X_test`	`n_test` x `p` matrix of predictors for test data; default is the observed covariates `X`
`psi`	prior variance (g-prior)
`laplace_approx`	logical; if TRUE, use a normal approximation to the posterior in the definition of the transformation; otherwise the prior is used
`fixedX`	logical; if TRUE, treat the design as fixed (non-random) when sampling the transformation; otherwise treat covariates as random with an unknown distribution
`approx_g`	logical; if TRUE, apply large-sample approximation for the transformation
`nsave`	number of Monte Carlo simulations
`ngrid`	number of grid points for inverse approximations
`verbose`	logical; if TRUE, print time remaining

Details

This function provides fully Bayesian inference for a transformed linear model using Monte Carlo (not MCMC) sampling. The transformation is modeled as unknown and learned jointly with the regression coefficients (unless `approx_g = TRUE`, which then uses a point approximation). This model applies for real-valued data, positive data, and compactly-supported data (the support is automatically deduced from the observed `y` values). The results are typically unchanged whether `laplace_approx` is TRUE/FALSE; setting it to TRUE may reduce sensitivity to the prior, while setting it to FALSE may speed up computations for very large datasets. By default, `fixedX` is set to FALSE for smaller datasets (`n < 500`) and TRUE for larger datasets (`n >= 500`).

Value

a list with the following elements:

- `coefficients` the posterior mean of the regression coefficients
- `fitted.values` the posterior predictive mean at the test points `X_test`
- `post_theta`: `nsave x p` samples from the posterior distribution of the regression coefficients
- `post_ypred`: `nsave x n_test` samples from the posterior predictive distribution at test points `X_test`
- `post_g`: `nsave` posterior samples of the transformation evaluated at the unique `y` values
- `model`: the model fit (here, `sblm`)

as well as the arguments passed in.

Note

The location (intercept) and scale (`sigma_epsilon`) are not identified, so any intercepts in `X` and `X_test` will be removed. The model-fitting *does* include an internal location-scale adjustment, but the function only outputs inferential summaries for the identifiable parameters.

Examples

```
# Simulate some data:
dat = simulate_tlm(n = 100, p = 5, g_type = 'step')
y = dat$y; X = dat$X # training data
y_test = dat$y_test; X_test = dat$X_test # testing data

hist(y, breaks = 25) # marginal distribution

# Fit the semiparametric Bayesian linear model:
fit = sblm(y = y, X = X, X_test = X_test)
names(fit) # what is returned

# Note: this is Monte Carlo sampling...no need for MCMC diagnostics!

# Evaluate posterior predictive means and intervals on the testing data:
plot_pptest(fit$post_ypred, y_test,
            alpha_level = 0.10) # coverage should be about 90%
```

```
# Check: correlation with true coefficients
cor(dat$beta_true, coef(fit))

# Summarize the transformation:
y0 = sort(unique(y)) # posterior draws of g are evaluated at the unique y observations
plot(y0, fit$post_g[1,], type='n', ylim = range(fit$post_g),
     xlab = 'y', ylab = 'g(y)', main = "Posterior draws of the transformation")
temp = sapply(1:nrow(fit$post_g), function(s)
  lines(y0, fit$post_g[s,], col='gray')) # posterior draws
lines(y0, colMeans(fit$post_g), lwd = 3) # posterior mean
lines(y, dat$g_true, type='p', pch=2) # true transformation
legend('bottomright', c('Truth'), pch = 2) # annotate the true transformation

# Posterior predictive checks on testing data: empirical CDF
y0 = sort(unique(y_test))
plot(y0, y0, type='n', ylim = c(0,1),
     xlab='y', ylab='F_y', main = 'Posterior predictive ECDF')
temp = sapply(1:nrow(fit$post_ypred), function(s)
  lines(y0, ecdf(fit$post_ypred[s,])(y0), # ECDF of posterior predictive draws
        col='gray', type ='s'))
lines(y0, ecdf(y_test)(y0),  # ECDF of testing data
     col='black', type = 's', lwd = 3)
```

`sblm_hs` *Semiparametric Bayesian linear model with horseshoe priors for high-dimensional data*

Description

MCMC sampling for semiparametric Bayesian linear regression with 1) an unknown (nonparametric) transformation and 2) a horseshoe prior for the (possibly high-dimensional) regression coefficients. Here, unlike `sblm`, Gibbs sampling is needed for the regression coefficients and the horseshoe prior variance components. The transformation g is still sampled unconditionally on the regression coefficients, which provides a more efficient blocking within the Gibbs sampler.

Usage

```
sblm_hs(
  y,
  X,
  X_test = X,
  fixedX = (length(y) >= 500),
  approx_g = FALSE,
  init_screen = NULL,
  pilot_hs = FALSE,
  nsave = 1000,
  nburn = 1000,
  ngrid = 100,
```

```
  verbose = TRUE
)
```

Arguments

`y`	`n x 1` response vector
`X`	`n x p` matrix of predictors (no intercept)
`X_test`	`n_test x p` matrix of predictors for test data; default is the observed covariates `X`
`fixedX`	logical; if TRUE, treat the design as fixed (non-random) when sampling the transformation; otherwise treat covariates as random with an unknown distribution
`approx_g`	logical; if TRUE, apply large-sample approximation for the transformation
`init_screen`	for the initial approximation, number of covariates to pre-screen (necessary when `p > n`); if NULL, use `n/log(n)`
`pilot_hs`	logical; if TRUE, use a short pilot run with a horseshoe prior to estimate the marginal CDF of the latent z (otherwise, use a sparse Laplace approximation)
`nsave`	number of MCMC simulations to save
`nburn`	number of MCMC iterations to discard
`ngrid`	number of grid points for inverse approximations
`verbose`	logical; if TRUE, print time remaining

Details

This function provides fully Bayesian inference for a transformed linear model with horseshoe priors using efficiently-blocked Gibbs sampling. The transformation is modeled as unknown and learned jointly with the regression coefficients (unless `approx_g` = TRUE, which then uses a point approximation). This model applies for real-valued data, positive data, and compactly-supported data (the support is automatically deduced from the observed `y` values).

The horseshoe prior is especially useful for high-dimensional settings with many (possibly correlated) covariates. Compared to sparse or spike-and-slab alternatives (see `sblm_ssvs`), the horseshoe prior delivers more scalable computing in `p`. This function uses a fast Cholesky-forward/backward sampler when `p < n` and the Bhattacharya et al. (<https://doi.org/10.1093/biomet/asw042>) sampler when `p > n`. Thus, the sampler can scale linear in `n` (for fixed/small `p`) or linear in `p` (for fixed/small `n`). Empirically, the horseshoe prior performs best under sparse regimes, i.e., when the number of true signals (nonzero regression coefficients) is a small fraction of the total number of variables.

To learn the transformation, `SeBR` infers the marginal CDF of the latent data model `Fz` by integrating over the covariates `X` and the coefficients `theta`. When `fixedX = TRUE`, the `X` averaging is empirical; otherwise it uses the Bayesian bootstrap (`bb`). By default, `fixedX` is set to `FALSE` for smaller datasets (`n < 500`) and `TRUE` for larger datasets. When `pilot_hs = TRUE`, the algorithm fits an initial linear regression model with a horseshoe prior (`blm_bc_hs`) to transformed data (under a preliminary point estimate of the transformation) and uses that posterior distribution to integrate over `theta`. Otherwise, this marginalization is done using a sparse Laplace approximation for speed and simplicity.

Value

a list with the following elements:

- `coefficients` the posterior mean of the regression coefficients
- `fitted.values` the posterior predictive mean at the test points `X_test`
- `post_theta`: `nsave` x `p` samples from the posterior distribution of the regression coefficients
- `post_ypred`: `nsave` x `n_test` samples from the posterior predictive distribution at test points `X_test`
- `post_g`: `nsave` posterior samples of the transformation evaluated at the unique `y` values
- `model`: the model fit (here, `sblm_hs`)

as well as the arguments passed in.

Note

The location (intercept) and scale (`sigma_epsilon`) are not identified, so any intercepts in `X` and `X_test` will be removed. The model-fitting *does* include an internal location-scale adjustment, but the function only outputs inferential summaries for the identifiable parameters.

Examples

```
# Simulate data from a transformed (sparse) linear model:
dat = simulate_tlm(n = 100, p = 50, g_type = 'step', prop_sig = 0.1)
y = dat$y; X = dat$X # training data
y_test = dat$y_test; X_test = dat$X_test # testing data

hist(y, breaks = 25) # marginal distribution

# Fit the semiparametric Bayesian linear model with a horseshoe prior:
fit = sblm_hs(y = y, X = X, X_test = X_test)
names(fit) # what is returned

# Evaluate posterior predictive means and intervals on the testing data:
plot_pptest(fit$post_ypred, y_test,
            alpha_level = 0.10) # coverage should be about 90%

# Check: correlation with true coefficients
cor(dat$beta_true, coef(fit))

# Compute 95% credible intervals for the coefficients:
ci_theta = t(apply(fit$post_theta, 2, quantile, c(0.05/2, 1 - 0.05/2)))

# True positive/negative rates for "selected" coefficients:
selected = ((ci_theta[,1] >0 | ci_theta[,2] < 0)) # intervals exclude zero
sigs_true = dat$beta_true != 0 # true signals
(TPR = sum(selected & sigs_true)/sum(sigs_true))
(TNR = sum(!selected & !sigs_true)/sum(!sigs_true))

# Summarize the transformation:
y0 = sort(unique(y)) # posterior draws of g are evaluated at the unique y observations
```

```
plot(y0, fit$post_g[1,], type='n', ylim = range(fit$post_g),
     xlab = 'y', ylab = 'g(y)', main = "Posterior draws of the transformation")
temp = sapply(1:nrow(fit$post_g), function(s)
  lines(y0, fit$post_g[s,], col='gray')) # posterior draws
lines(y0, colMeans(fit$post_g), lwd = 3) # posterior mean
lines(y, dat$g_true, type='p', pch=2) # true transformation
legend('bottomright', c('Truth'), pch = 2) # annotate the true transformation

# Posterior predictive checks on testing data: empirical CDF
y0 = sort(unique(y_test))
plot(y0, y0, type='n', ylim = c(0,1),
     xlab='y', ylab='F_y', main = 'Posterior predictive ECDF')
temp = sapply(1:nrow(fit$post_ypred), function(s)
  lines(y0, ecdf(fit$post_ypred[s,])(y0), # ECDF of posterior predictive draws
        col='gray', type ='s'))
lines(y0, ecdf(y_test)(y0),  # ECDF of testing data
     col='black', type = 's', lwd = 3)
```

`sblm_modelsel` *Model selection for semiparametric Bayesian linear regression*

Description

Compute model probabilities for semiparametric Bayesian linear regression with 1) an unknown (nonparametric) transformation and 2) a sparsity prior on the regression coefficients. The model probabilities are computed using direct Monte Carlo (not MCMC) sampling.

Usage

```
sblm_modelsel(
  y,
  X,
  prob_inclusion = 0.5,
  psi = length(y),
  fixedX = (length(y) >= 500),
  init_screen = NULL,
  nsave = 1000,
  override = FALSE,
  ngrid = 100,
  verbose = TRUE
)
```

Arguments

`y`	n x 1 response vector
`X`	n x p matrix of predictors (no intercept)
`prob_inclusion`	prior inclusion probability for each variable

`psi`	prior variance (g-prior)
`fixedX`	logical; if TRUE, treat the design as fixed (non-random) when sampling the transformation; otherwise treat covariates as random with an unknown distribution
`init_screen`	for the initial approximation, number of covariates to pre-screen (necessary when `p > n`); if NULL, use `n/log(n)`
`nsave`	number of Monte Carlo simulations
`override`	logical; if TRUE, the user may override the default cancellation of the function call when `p > 15`
`ngrid`	number of grid points for inverse approximations
`verbose`	logical; if TRUE, print time remaining

Details

This function provides fully Bayesian model selection for a transformed linear model with sparse g-priors on the regression coefficients. The transformation is modeled as unknown and learned jointly with the model probabilities. This model applies for real-valued data, positive data, and compactly-supported data (the support is automatically deduced from the observed `y` values). By default, `fixedX` is set to `FALSE` for smaller datasets (`n < 500`) and `TRUE` for larger datasets.

Enumeration of all possible subsets is computationally demanding and should be reserved only for small `p`. The function will exit for `p > 15` unless `override = TRUE`.

This function exclusively computes model probabilities and does not provide other coefficient inference or prediction. These additions would be straightforward, but are omitted to save on computing time. For prediction, inference, and computation with moderate to large `p`, use `sblm_ssvs`.

Value

a list with the following elements:

- `post_probs` the posterior probabilities for each model
- `all_models`: `2^p x p` matrix where each row corresponds to a model from `post_probs` and each column indicates inclusion (TRUE) or exclusion (FALSE) for that variable
- `model`: the model fit (here, `sblm_modelsel`)

as well as the arguments passed in.

Note

The location (intercept) and scale (`sigma_epsilon`) are not identified, so any intercept in `X` will be removed. The model-fitting *does* include an internal location-scale adjustment, but the model probabilities only refer to the non-intercept variables in `X`.

Examples

```
# Simulate data from a transformed (sparse) linear model:
dat = simulate_tlm(n = 100, p = 5, g_type = 'beta')
y = dat$y; X = dat$X

hist(y, breaks = 25) # marginal distribution

# Package for conveniently computing all subsets:
library(plyr)

# Fit the semiparametric Bayesian linear model with model selection:
fit = sblm_modelsel(y = y, X = X)
names(fit) # what is returned

# Summarize the probabilities of each model (by size):
plot(rowSums(fit$all_models), fit$post_probs,
     xlab = 'Model sizes', ylab = 'p(model | data)',
     main = 'Posterior model probabilities', pch = 2, ylim = c(0,1))

# Highest probability model:
hpm = which.max(fit$post_probs)
fit$post_probs[hpm] # probability
which(fit$all_models[hpm,]) # which variables
which(dat$beta_true != 0) # ground truth
```

`sblm_ssvs` *Semiparametric Bayesian linear model with stochastic search variable selection*

Description

MCMC sampling for semiparametric Bayesian linear regression with 1) an unknown (nonparametric) transformation and 2) a sparsity prior on the (possibly high-dimensional) regression coefficients. Here, unlike `sblm`, Gibbs sampling is used for the variable inclusion indicator variables `gamma`, referred to as stochastic search variable selection (SSVS). All remaining terms–including the transformation `g`, the regression coefficients `theta`, and any predictive draws–are drawn directly from the joint posterior (predictive) distribution.

Usage

```
sblm_ssvs(
  y,
  X,
  X_test = X,
  psi = length(y),
  fixedX = (length(y) >= 500),
  approx_g = FALSE,
```

```
  init_screen = NULL,
  a_pi = 1,
  b_pi = 1,
  nsave = 1000,
  nburn = 1000,
  ngrid = 100,
  verbose = TRUE
)
```

Arguments

`y`	`n` x `1` response vector
`X`	`n` x `p` matrix of predictors (no intercept)
`X_test`	`n_test` x `p` matrix of predictors for test data; default is the observed covariates `X`
`psi`	prior variance (g-prior)
`fixedX`	logical; if TRUE, treat the design as fixed (non-random) when sampling the transformation; otherwise treat covariates as random with an unknown distribution
`approx_g`	logical; if TRUE, apply large-sample approximation for the transformation
`init_screen`	for the initial approximation, number of covariates to pre-screen (necessary when `p` > `n`); if NULL, use `n/log(n)`
`a_pi`	shape1 parameter of the (Beta) prior inclusion probability
`b_pi`	shape2 parameter of the (Beta) prior inclusion probability
`nsave`	number of MCMC simulations to save
`nburn`	number of MCMC iterations to discard
`ngrid`	number of grid points for inverse approximations
`verbose`	logical; if TRUE, print time remaining

Details

This function provides fully Bayesian inference for a transformed linear model with sparse g-priors on the regression coefficients. The transformation is modeled as unknown and learned jointly with the regression coefficients (unless `approx_g` = TRUE, which then uses a point approximation). This model applies for real-valued data, positive data, and compactly-supported data (the support is automatically deduced from the observed `y` values). By default, `fixedX` is set to `FALSE` for smaller datasets (`n < 500`) and `TRUE` for larger datasets.

The sparsity prior is especially useful for variable selection. Compared to the horseshoe prior version (`sblm_hs`), the sparse g-prior is advantageous because 1) it truly allows for sparse (i.e., exactly zero) coefficients in the prior and posterior, 2) it incorporates covariate dependencies via the g-prior structure, and 3) it tends to perform well under both sparse and non-sparse regimes, while the horseshoe version only performs well under sparse regimes. The disadvantage is that SSVS does not scale nearly as well in `p`.

Following Scott and Berger (<https://doi.org/10.1214/10-AOS792>), we include a `Beta(a_pi, b_pi)` prior on the prior inclusion probability. This term is then sampled with the variable inclusion indicators `gamma` in a Gibbs sampling block. All other terms are sampled using direct Monte Carlo (not MCMC) sampling.

Alternatively, model probabilities can be computed directly (by Monte Carlo, not MCMC/Gibbs sampling) using `sblm_modelsel`.

Value

a list with the following elements:

- `coefficients` the posterior mean of the regression coefficients
- `fitted.values` the posterior predictive mean at the test points `X_test`
- `selected`: the variables (columns of `X`) selected by the median probability model
- `pip`: (marginal) posterior inclusion probabilities for each variable
- `post_theta`: `nsave x p` samples from the posterior distribution of the regression coefficients
- `post_gamma`: `nsave x p` samples from the posterior distribution of the variable inclusion indicators
- `post_ypred`: `nsave x n_test` samples from the posterior predictive distribution at test points `X_test`
- `post_g`: `nsave` posterior samples of the transformation evaluated at the unique `y` values
- `model`: the model fit (here, `sblm_ssvs`)

as well as the arguments passed in.

Note

The location (intercept) and scale (`sigma_epsilon`) are not identified, so any intercepts in `X` and `X_test` will be removed. The model-fitting *does* include an internal location-scale adjustment, but the function only outputs inferential summaries for the identifiable parameters.

Examples

```
# Simulate data from a transformed (sparse) linear model:
dat = simulate_tlm(n = 100, p = 15, g_type = 'step')
y = dat$y; X = dat$X # training data
y_test = dat$y_test; X_test = dat$X_test # testing data

hist(y, breaks = 25) # marginal distribution

# Fit the semiparametric Bayesian linear model with sparsity priors:
fit = sblm_ssvs(y = y, X = X, X_test = X_test)
names(fit) # what is returned

# Evaluate posterior predictive means and intervals on the testing data:
plot_pptest(fit$post_ypred, y_test,
            alpha_level = 0.10) # coverage should be about 90%

# Check: correlation with true coefficients
cor(dat$beta_true, coef(fit))
```

```
# Selected coefficients under median probability model:
fit$selected

# True signals:
which(dat$beta_true != 0)

# Summarize the transformation:
y0 = sort(unique(y)) # posterior draws of g are evaluated at the unique y observations
plot(y0, fit$post_g[1,], type='n', ylim = range(fit$post_g),
     xlab = 'y', ylab = 'g(y)', main = "Posterior draws of the transformation")
temp = sapply(1:nrow(fit$post_g), function(s)
  lines(y0, fit$post_g[s,], col='gray')) # posterior draws
lines(y0, colMeans(fit$post_g), lwd = 3) # posterior mean
lines(y, dat$g_true, type='p', pch=2) # true transformation

# Posterior predictive checks on testing data: empirical CDF
y0 = sort(unique(y_test))
plot(y0, y0, type='n', ylim = c(0,1),
     xlab='y', ylab='F_y', main = 'Posterior predictive ECDF')
temp = sapply(1:nrow(fit$post_ypred), function(s)
  lines(y0, ecdf(fit$post_ypred[s,])(y0), # ECDF of posterior predictive draws
        col='gray', type ='s'))
lines(y0, ecdf(y_test)(y0),  # ECDF of testing data
     col='black', type = 's', lwd = 3)
```

`sbqr` *Semiparametric Bayesian quantile regression*

Description

MCMC sampling for Bayesian quantile regression with an unknown (nonparametric) transformation. Like in traditional Bayesian quantile regression, an asymmetric Laplace distribution is assumed for the errors, so the regression models targets the specified quantile. However, these models are often woefully inadequate for describing observed data. We introduce a nonparametric transformation to improve model adequacy while still providing inference for the regression coefficients and the specified quantile. A g-prior is assumed for the regression coefficients.

Usage

```
sbqr(
  y,
  X,
  tau = 0.5,
  X_test = X,
  psi = length(y),
  laplace_approx = TRUE,
  fixedX = TRUE,
```

```
  approx_g = FALSE,
  nsave = 1000,
  nburn = 100,
  ngrid = 100,
  verbose = TRUE
)
```

Arguments

`y`	`n x 1` response vector
`X`	`n x p` matrix of predictors (no intercept)
`tau`	the target quantile (between zero and one)
`X_test`	`n_test x p` matrix of predictors for test data; default is the observed covariates `X`
`psi`	prior variance (g-prior)
`laplace_approx`	logical; if TRUE, use a normal approximation to the posterior in the definition of the transformation; otherwise the prior is used
`fixedX`	logical; if TRUE, treat the design as fixed (non-random) when sampling the transformation; otherwise treat covariates as random with an unknown distribution
`approx_g`	logical; if TRUE, apply large-sample approximation for the transformation
`nsave`	number of MCMC iterations to save
`nburn`	number of MCMC iterations to discard
`ngrid`	number of grid points for inverse approximations
`verbose`	logical; if TRUE, print time remaining

Details

This function provides fully Bayesian inference for a transformed quantile linear model. The transformation is modeled as unknown and learned jointly with the regression coefficients (unless `approx_g` = TRUE, which then uses a point approximation). This model applies for real-valued data, positive data, and compactly-supported data (the support is automatically deduced from the observed `y` values). The results are typically unchanged whether `laplace_approx` is TRUE/FALSE; setting it to TRUE may reduce sensitivity to the prior, while setting it to FALSE may speed up computations for very large datasets. Similarly, treating the covariates as fixed (`fixedX = TRUE`) can substantially improve computing efficiency, so we make this the default.

Value

a list with the following elements:

- `coefficients` the posterior mean of the regression coefficients
- `fitted.values` the estimated `tau`th quantile at test points `X_test`
- `post_theta`: `nsave x p` samples from the posterior distribution of the regression coefficients
- `post_ypred`: `nsave x n_test` samples from the posterior predictive distribution at test points `X_test`

- `post_qtau`: `nsave` x `n_test` samples of the `tau`th conditional quantile at test points `X_test`
- `post_g`: `nsave` posterior samples of the transformation evaluated at the unique `y` values
- `model`: the model fit (here, `sbqr`)

as well as the arguments passed in.

Note

The location (intercept) is not identified, so any intercepts in `X` and `X_test` will be removed. The model-fitting *does* include an internal location-scale adjustment, but the function only outputs inferential summaries for the identifiable parameters.

Examples

```
# Simulate some heteroskedastic data (no transformation):
dat = simulate_tlm(n = 200, p = 10, g_type = 'box-cox', heterosked = TRUE, lambda = 1)
y = dat$y; X = dat$X # training data
y_test = dat$y_test; X_test = dat$X_test # testing data

# Target this quantile:
tau = 0.05

# Fit the semiparametric Bayesian quantile regression model:
fit = sbqr(y = y, X = X, tau = tau, X_test = X_test)
names(fit) # what is returned

# Posterior predictive checks on testing data: empirical CDF
y0 = sort(unique(y_test))
plot(y0, y0, type='n', ylim = c(0,1),
     xlab='y', ylab='F_y', main = 'Posterior predictive ECDF')
temp = sapply(1:nrow(fit$post_ypred), function(s)
  lines(y0, ecdf(fit$post_ypred[s,])(y0), # ECDF of posterior predictive draws
        col='gray', type ='s'))
lines(y0, ecdf(y_test)(y0),  # ECDF of testing data
     col='black', type = 's', lwd = 3)
```

`sbsm` *Semiparametric Bayesian spline model*

Description

Monte Carlo sampling for Bayesian spline regression with an unknown (nonparametric) transformation. Cubic B-splines are used with a prior that penalizes roughness.

Usage

```
sbsm(
  y,
  x = NULL,
  x_test = x,
  psi = NULL,
  laplace_approx = TRUE,
  fixedX = (length(y) >= 500),
  approx_g = FALSE,
  nsave = 1000,
  ngrid = 100,
  verbose = TRUE
)
```

Arguments

`y`	`n x 1` response vector
`x`	`n x 1` vector of observation points; if NULL, assume equally-spaced on [0,1]
`x_test`	`n_test x 1` vector of testing points; if NULL, assume equal to `x`
`psi`	prior variance (inverse smoothing parameter); if NULL, sample this parameter
`laplace_approx`	logical; if TRUE, use a normal approximation to the posterior in the definition of the transformation; otherwise the prior is used
`fixedX`	logical; if TRUE, treat the design as fixed (non-random) when sampling the transformation; otherwise treat covariates as random with an unknown distribution
`approx_g`	logical; if TRUE, apply large-sample approximation for the transformation
`nsave`	number of Monte Carlo simulations
`ngrid`	number of grid points for inverse approximations
`verbose`	logical; if TRUE, print time remaining

Details

This function provides fully Bayesian inference for a transformed spline regression model using Monte Carlo (not MCMC) sampling. The transformation is modeled as unknown and learned jointly with the regression function (unless `approx_g = TRUE`, which then uses a point approximation). This model applies for real-valued data, positive data, and compactly-supported data (the support is automatically deduced from the observed `y` values). The results are typically unchanged whether `laplace_approx` is TRUE/FALSE; setting it to TRUE may reduce sensitivity to the prior, while setting it to FALSE may speed up computations for very large datasets. By default, `fixedX` is set to FALSE for smaller datasets (`n < 500`) and TRUE for larger datasets (`n >= 500`).

Value

a list with the following elements:

- `coefficients` the posterior mean of the regression coefficients

- `fitted.values` the posterior predictive mean at the test points `x_test`
- `post_theta`: `nsave` x `p` samples from the posterior distribution of the regression coefficients
- `post_ypred`: `nsave` x `n_test` samples from the posterior predictive distribution at `x_test`
- `post_g`: `nsave` posterior samples of the transformation evaluated at the unique `y` values
- `model`: the model fit (here, `sbsm`)

as well as the arguments passed in.

Examples

```
# Simulate some data:
n = 200 # sample size
x = sort(runif(n)) # observation points

# Transform a noisy, periodic function:
y = g_inv_bc(
  sin(2*pi*x) + sin(4*pi*x) + rnorm(n),
             lambda = .5) # Signed square-root transformation

# Fit the semiparametric Bayesian spline model:
fit = sbsm(y = y, x = x)
names(fit) # what is returned

# Note: this is Monte Carlo sampling...no need for MCMC diagnostics!

# Plot the model predictions (point and interval estimates):
pi_y = t(apply(fit$post_ypred, 2, quantile, c(0.05, .95))) # 90% PI
plot(x, y, type='n', ylim = range(pi_y,y),
     xlab = 'x', ylab = 'y', main = paste('Fitted values and prediction intervals'))
polygon(c(x, rev(x)),c(pi_y[,2], rev(pi_y[,1])),col='gray', border=NA)
lines(x, y, type='p') # observed points
lines(x, fitted(fit), lwd = 3) # fitted curve
```

`simulate_tlm` *Simulate a transformed linear model*

Description

Generate training data (X, y) and testing data (X_test, y_test) for a transformed linear model. The covariates are correlated Gaussian variables. A user-specified proportion (`prop_sig`) of the regression coefficients are nonozero (= 1) and the rest are zero. There are multiple options for the transformation, which define the support of the data (see below).

Usage

```
simulate_tlm(
  n,
  p,
  g_type = "beta",
  n_test = 1000,
  heterosked = FALSE,
  lambda = 1,
  prop_sig = 0.5
)
```

Arguments

`n`	number of observations in the training data
`p`	number of covariates
`g_type`	type of transformation; must be one of `beta`, `step`, or `box-cox`
`n_test`	number of observations in the testing data
`heterosked`	logical; if TRUE, simulate the latent data with heteroskedasticity
`lambda`	Box-Cox parameter (only applies for `g_type = 'box-cox'`)
`prop_sig`	proportion of signals (nonzero coefficients)

Details

The transformations vary in complexity and support for the observed data, and include the following options: `beta` yields marginally Beta(0.1, 0.5) data supported on [0,1]; `step` generates a locally-linear inverse transformation and produces positive data; and `box-cox` refers to the signed Box-Cox family indexed by `lambda`, which generates real-valued data with examples including identity, square-root, and log transformations.

Value

a list with the following elements:

- `y`: the response variable in the training data
- `X`: the covariates in the training data
- `y_test`: the response variable in the testing data
- `X_test`: the covariates in the testing data
- `beta_true`: the true regression coefficients
- `g_true`: the true transformation, evaluated at y

Note

The design matrices `X` and `X_test` do not include an intercept and there is no intercept parameter in `beta_true`. The location/scale of the data are not identified in general transformed regression models, so recovering them is not a goal.

Examples

```
# Simulate data:
dat = simulate_tlm(n = 100, p = 5, g_type = 'beta')
names(dat) # what is returned
hist(dat$y, breaks = 25) # marginal distribution
```

`sir_adjust` *Post-processing with importance sampling*

Description

Given Monte Carlo draws from the surrogate posterior, apply sampling importance reweighting (SIR) to correct for the true model likelihood.

Usage

```
sir_adjust(
  fit,
  sir_frac = 0.3,
  nsims_prior = 100,
  marg_x = FALSE,
  verbose = TRUE
)
```

Arguments

`fit`	a fitted model object that includes • `coefficients` the posterior mean of the regression coefficients • `post_theta`: `nsave` x `p` samples from the posterior distribution of the regression coefficients • `post_ypred`: `nsave` x `n_test` samples from the posterior predictive distribution at test points `X_test` • `post_g`: `nsave` posterior samples of the transformation evaluated at the unique `y` values • `model`: the model fit (`sblm` or `sbsm`)
`sir_frac`	fraction of draws to sample for SIR
`nsims_prior`	number of draws from the prior
`marg_x`	logical; if TRUE, compute the weights marginal over the covariates
`verbose`	logical; if TRUE, print time remaining

Details

The Monte Carlo sampling for `sblm` and `sbsm` uses a surrogate likelihood for posterior inference, which enables much faster and easier computing. SIR provides a correction for the actual (specified) likelihood. However, this correction step typically does not produce any noticeable discrepancies, even for small sample sizes.

Value

the fitted model object with the posterior draws subsampled based on the SIR adjustment

Note

SIR sampling is done *without* replacement, so `sir_frac` is typically between 0.1 and 0.5. The `nsims_priors` draws are used to approximate a prior expectation, but larger values can significantly slow down this function. The importance weights can be computed conditionally (`marg_x = FALSE`) or unconditionally (`marg_x = TRUE`) on the covariates, corresponding to whether or not the covariates are marginalized out in the likelihood. The conditional version is much faster.

Examples

```
# Simulate some data:
dat = simulate_tlm(n = 50, p = 5, g_type = 'step')
y = dat$y; X = dat$X # training data
y_test = dat$y_test; X_test = dat$X_test # testing data

hist(y, breaks = 10) # marginal distribution

# Fit the semiparametric Bayesian linear model:
fit = sblm(y = y, X = X, X_test = X_test)
names(fit) # what is returned

# Update with SIR:
fit_sir = sir_adjust(fit)
names(fit_sir) # what is returned

# Prediction: unadjusted vs. adjusted?

# Point estimates:
y_hat = fitted(fit)
y_hat_sir = fitted(fit_sir)
cor(y_hat, y_hat_sir) # similar

# Interval estimates:
pi_y = t(apply(fit$post_ypred, 2, quantile, c(0.05, .95))) # 90% PI
pi_y_sir = t(apply(fit_sir$post_ypred, 2, quantile, c(0.05, .95))) # 90% PI

# PI overlap (%):
overlaps = 100*sapply(1:length(y_test), function(i){
  # innermost part
  (min(pi_y[i,2], pi_y_sir[i,2]) - max(pi_y[i,1], pi_y_sir[i,1]))/
    # outermost part
    (max(pi_y[i,2], pi_y_sir[i,2]) - min(pi_y[i,1], pi_y_sir[i,1]))
})
summary(overlaps) # mostly close to 100%

# Coverage of PIs on testing data (should be ~ 90%)
mean((pi_y[,1] <= y_test)*(pi_y[,2] >= y_test)) # unadjusted
mean((pi_y_sir[,1] <= y_test)*(pi_y_sir[,2] >= y_test)) # adjusted
```

```
# Plot together with testing data:
plot(y_test, y_test, type='n', ylim = range(pi_y, pi_y_sir, y_test),
     xlab = 'y_test', ylab = 'y_hat', main = paste('Prediction intervals: testing data'))
abline(0,1) # reference line
suppressWarnings(
  arrows(y_test, pi_y[,1], y_test, pi_y[,2],
         length=0.15, angle=90, code=3, col='gray', lwd=2)
) # plot the PIs (unadjusted)
suppressWarnings(
  arrows(y_test, pi_y_sir[,1], y_test, pi_y_sir[,2],
         length=0.15, angle=90, code=3, col='darkgray', lwd=2)
) # plot the PIs (adjusted)
lines(y_test, y_hat, type='p', pch=2) # plot the means (unadjusted)
lines(y_test, y_hat_sir, type='p', pch=3) # plot the means (adjusted)
```

`square_stabilize` *Numerically stabilize the squared elements*

Description

Given a vector to be squared, add a numeric buffer for the elements very close to zero.

Usage

```
square_stabilize(vec)
```

Arguments

`vec`	vector of inputs to be squared

Value

a vector of the same length as ‘vec‘

`SSR_gprior` *Compute the sum-squared-residuals term under Zellner’s g-prior*

Description

These sum-squared-residuals (SSR) arise in the variance (or precision) term under 1) Zellner’s g-prior on the coefficients and a Gamma prior on the error precision and 2) marginalization over the coefficients.

Usage

```
SSR_gprior(y, X = NULL, psi)
```

Arguments

`y`	vector of response variables
`X`	matrix of covariates; if NULL, return `sum(y^2)`
`psi`	prior variance (g-prior)

Value

a positive scalar

`uni.slice` *Univariate Slice Sampler from Neal (2008)*

Description

Compute a draw from a univariate distribution using the code provided by Radford M. Neal. The documentation below is also reproduced from Neal (2008).

Usage

```
uni.slice(x0, g, w = 1, m = Inf, lower = -Inf, upper = +Inf, gx0 = NULL)
```

Arguments

`x0`	Initial point
`g`	Function returning the log of the probability density (plus constant)
`w`	Size of the steps for creating interval (default 1)
`m`	Limit on steps (default infinite)
`lower`	Lower bound on support of the distribution (default -Inf)
`upper`	Upper bound on support of the distribution (default +Inf)
`gx0`	Value of g(x0), if known (default is not known)

Value

The point sampled, with its log density attached as an attribute.

Note

The log density function may return -Inf for points outside the support of the distribution. If a lower and/or upper bound is specified for the support, the log density function will not be called outside such limits.

Index