Package 'lgspline'

July 22, 2025

Type Package

Title Lagrangian Multiplier Smoothing Splines for Smooth Function Estimation

Version 0.2.0

Description Implements Lagrangian multiplier smoothing splines for flexible nonparametric regression and function estimation. Provides tools for fitting, prediction, and inference using a constrained optimization approach to enforce smoothness.
Supports generalized linear models, Weibull accelerated failure time (AFT) models, quadratic programming problems, and customizable arbitrary correlation structures. Options for fitting in parallel are provided. The method builds upon the framework described by Ezhov et al. (2018) <doi:10.1515/jag-2017-0029> using Lagrangian multipliers to fit cubic splines. For more information on correlation structure estimation, see Searle et al. (2009) <ISBN:978-0470009598>. For quadratic programming and constrained optimization in general, see Nocedal & Wright (2006) <doi:10.1007/978-0-387-40065-5>.
For a comprehensive background on smoothing splines, see Wahba (1990) <doi:10.1137/1.9781611970128> and Wood (2006) <ISBN:978-1584884743> ``Generalized Additive Models: An Introduction with R".

License MIT + file LICENSE

Language en-US

Depends R (>= 3.5.0)

Imports Rcpp (>= 1.0.7), RcppArmadillo, FNN, RColorBrewer, plotly, quadprog, methods, stats

LinkingTo Rcpp, RcppArmadillo

Suggests testthat (>= 3.0.0), spelling, knitr, rmarkdown, parallel, survival, graphics

URL https://github.com/matthewlouisdavisBioStat/lgspline

BugReports https://github.com/matthewlouisdavisBioStat/lgspline/issues

Encoding UTF-8

RoxygenNote 7.3.2

Config/testthat/edition 3

NeedsCompilation yes

Author Matthew Davis [aut, cre] (ORCID:
<https://orcid.org/0000-0001-9714-1018>)

Maintainer Matthew Davis <matthewlouisdavis@gmail.com>

Repository CRAN

Date/Publication 2025-04-24 11:40:13 UTC

Contents

`coef.lgspline` *Extract model coefficients*

Description

Extracts polynomial coefficients for each partition from a fitted lgspline model.

Usage

```
## S3 method for class 'lgspline'
coef(object, ...)
```

Arguments

`object`	A fitted lgspline model object containing coefficient vectors.
`...`	Not used.

Details

For each partition, coefficients represent a polynomial expansion of the predictor(s) by column index, for example:

- intercept: Constant term
- v: Linear term
- v_^2: Quadratic term
- v^3: Cubic term
- _v_x_w_: Interaction between v and w

If column/variable names are present, indices will be replaced with column/variable names.

Coefficients can be accessed either as separate vectors per partition or combined into a single matrix using `Reduce('cbind', coef(model_fit))`.

Value

A list where each element corresponds to a partition and contains a single-column matrix of coefficient values for that partition. Row names indicate the term type. Returns NULL if coefficients are not found in the object.

partition1, partition2, ... Matrices containing coefficients for each partition.

See Also

lgspline

Examples

```
## Simulate some data and fit using default settings
set.seed(1234)
t <- runif(1000, -10, 10)
y <- 2*sin(t) + -0.06*t^2 + rnorm(length(t))
model_fit <- lgspline(t, y)

## Extract coefficients
coefficients <- coef(model_fit)

## Print coefficients for first partition
print(coefficients[[1]])
```

```
## Compare coefficients across all partitions
print(Reduce('cbind', coefficients))
```

`create_onehot` *Create One-Hot Encoded Matrix*

Description

Converts a categorical vector into a one-hot encoded matrix where each unique value becomes a binary column.

Usage

```
create_onehot(x)
```

Arguments

x	A vector containing categorical values (factors, character, etc.)

Details

The function creates dummy variables for each unique value in the input vector using `model.matrix()` with dummy-intercept coding. Column names are cleaned by removing the 'x' prefix added by `model.matrix()`.

Value

A data frame containing the one-hot encoded binary columns with cleaned column names

Examples

```
## lgspline will not accept this format of "catvar", because inputting data
# this way can cause difficult-to-diagnose issues in formula parsing
# all variables must be numeric
df <- data.frame(numvar = rnorm(100),
                 catvar = rep(LETTERS[1:4],
                              25))
print(head(df))

## Instead, replace with dummy-intercept coding by
# 1) applying one-hot encoding
# 2) dropping the first column
# 3) appending to our data

dummy_intercept_coding <- create_onehot(df$catvar)[,-1]
df$catvar <- NULL
df <- cbind(df, dummy_intercept_coding)
print(head(df))
```

`Details` *Lagrangian Multiplier Smoothing Splines: Mathematical Details*

Description

This document provides the mathematical and implementation details for Lagrangian Multiplier Smoothing Splines.

Statistical Problem Formulation

Consider a dataset with observed predictors $\mathbf{T}$ (an $N \times q$ matrix) and corresponding response values $\mathbf{y}$ (an $N \times 1$ vector). We assume the relationship follows a generalized linear model with an unknown smooth function f:

$$y_i \sim \mathcal{D}(g^{-1}(f(\mathbf{t}_i)), \sigma^2)$$

where $\mathcal{D}$ is a distribution, g is a link function, $\mathbf{t}_i$ is the ith row of $\mathbf{T}$, and σ^2 is a dispersion parameter.

The objective is to estimate the unknown function f that balances:

- Goodness of fit: How well the model fits the observed data
- Smoothness: Avoiding excessive fluctuations in the estimated function
- Interpretability: Understanding the relationship between predictors and response

Lagrangian Multiplier Smoothing Splines address this by:

1. Partitioning the predictor space into $K + 1$ regions
2. Fitting local polynomial models within each partition
3. Explicitly enforcing smoothness where partitions meet using Lagrangian multipliers
4. Penalizing the integrated squared second derivative of the estimated function

Unlike other smoothing spline formulations, this technique ensures that no post-fitting algebraic rearrangement or disentangelement of a spline basis is needed to obtain interpretable models. The relationship between predictor and response is explicit, and the basis expansions for each partition are homogeneous.

Overview

Lagrangian Multiplier Smoothing Splines fit piecewise polynomial regression models to partitioned data, with smoothness at partition boundaries enforced through Lagrangian multipliers. The approach is penalized by the integrated squared second derivative of the estimated function.

Unlike traditional smoothing splines that implicitly derive piecewise polynomials through optimization, or regression splines using specialized bases (e.g., B-splines), this method:

- Explicitly represents polynomial basis functions in a natural form

- Uses Lagrangian multipliers to enforce smoothness constraints
- Maintains interpretability of coefficient estimates

The fitted model is directly interpretable without the need for reparameterization following fitting. This implementation accommodates non-spline terms and interactions, GLMs, correlation structures, and inequality constraints in addition to linear regression assuming Gaussian response. Extensive customization is offered for users to adapt lgspline for their own modelling frameworks.

Core notation:

- $\mathbf{y}_{(N\times 1)}$: Response vector
- $\mathbf{T}_{(N\times q)}$: Matrix of predictors
- $\mathbf{X}_{(N\times P)}$: Block-diagonal matrix of polynomial expansions
- $\mathbf{\Lambda}_{(P\times P)}$: Penalty matrix
- $\tilde{\boldsymbol{\beta}}_{(P\times 1)}$: Constrained coefficient estimates
- $\mathbf{G}_{(P\times P)}$: $(\mathbf{X}^T\mathbf{W}\mathbf{X}+\mathbf{\Lambda})^{-1}$
- $\mathbf{A}_{(P\times r)}$: Constraint matrix ensuring smoothness
- $\mathbf{U}_{(P\times P)}$: $\mathbf{I}-\mathbf{G}\mathbf{A}(\mathbf{A}^T\mathbf{G}\mathbf{A})^{-1}\mathbf{A}^T$

Model Formulation and Estimation

Model Structure: The method decomposes the predictor space into K+1 partitions and fits polynomial regression models within each partition, constraining the fitted function to be smooth at partition boundaries.

For a single predictor, the function within each partition k is represented as:

$$f_k(t) = \beta_k^{(0)} + \beta_k^{(1)}t + \beta_k^{(2)}t^2 + \beta_k^{(3)}t^3 + ...$$

More generally, for each partition k, the model takes the form:

$$f_k(\mathbf{t}) = \mathbf{x}^T\boldsymbol{\beta}_k$$

Where $\mathbf{x}$ contains polynomial basis functions (intercept, linear, quadratic, cubic terms, and their interactions) and $\boldsymbol{\beta}_k$ are the corresponding coefficients.

Smoothness constraints enforce that the function value, first and second derivatives match at adjacent partition boundaries:

$$f_k(t_k) = f_{k+1}(t_k)$$
$$f_k'(t_k) = f_{k+1}'(t_k)$$
$$f_k''(t_k) = f_{k+1}''(t_k)$$

These constraints are expressed as linear equations in the $\mathbf{A}$ matrix such that $\mathbf{A}^T\boldsymbol{\beta} = \mathbf{0}$ implies the smoothness conditions are satisfied.

Estimation Procedure: The estimation procedure follows these key steps:

1. Unconstrained estimation:

$$\hat{\boldsymbol{\beta}} = \mathbf{G}\mathbf{X}^T\mathbf{y}$$

where $\mathbf{G} = (\mathbf{X}^T\mathbf{W}\mathbf{X} + \mathbf{\Lambda})^{-1}$ for weighted design matrix $\mathbf{W}$ and penalty matrix $\mathbf{\Lambda}$.

2. Apply smoothness constraints:

$$\tilde{\boldsymbol{\beta}} = \hat{\boldsymbol{\beta}} - \mathbf{G}\mathbf{A}(\mathbf{A}^T\mathbf{G}\mathbf{A})^{-1}\mathbf{A}^T\hat{\boldsymbol{\beta}}$$

This can be computed efficiently as:

$$\tilde{\boldsymbol{\beta}} = \mathbf{U}\hat{\boldsymbol{\beta}}$$

where $\mathbf{U} = \mathbf{I} - \mathbf{G}\mathbf{A}(\mathbf{A}^T\mathbf{G}\mathbf{A})^{-1}\mathbf{A}^T$.

3. For GLMs, iterative refinement:
   - Update $\mathbf{G}$ based on current estimates
   - Recompute $\tilde{\boldsymbol{\beta}}$ using the new $\mathbf{G}$
   - Continue until convergence
4. For inequality constraints (shape or range constraints):
   - Sequential quadratic programming using `solve.QP`
   - Initial values from the equality-constrained estimates
5. Variance estimation:

$$\tilde{\sigma}^2 = \frac{\|\mathbf{y} - \mathbf{X}\tilde{\boldsymbol{\beta}}\|^2}{N - \text{trace}(\mathbf{X}\mathbf{U}\mathbf{G}\mathbf{X}^T)}$$

The variance-covariance matrix of $\tilde{\boldsymbol{\beta}}$ is estimated as:

$$\text{Var}(\tilde{\boldsymbol{\beta}}) = \tilde{\sigma}^2\mathbf{U}\mathbf{G}$$

This approach offers computational efficiency through:

- Parallel computation for each partition
- Inversion of only small matrices (one per partition)
- Efficient calculation of $\mathbf{X}\mathbf{U}\mathbf{G}\mathbf{X}^T$ trace

Knot Selection and Partitioning

Univariate Case: For a single predictor, knots are placed at evenly-spaced quantiles of the predictor variable:

- The predictor range is divided into K+1 regions using K interior knots
- Each observation is assigned to a partition based on these knot boundaries
- Custom knots can be specified through the `custom_knots` parameter

Multivariate Case: For multiple predictors, a k-means clustering approach is used:

1. K+1 cluster centers are identified using k-means on standardized predictors
2. Neighboring centers are determined using a midpoint criterion:
   - Centers i and j are neighbors if the midpoint between them is closer to both i and j than to any other center
3. Observations are assigned to the nearest cluster center

The default number of knots (K) is determined adaptively based on:

- Sample size (N)
- Number of basis terms per partition (p)
- Number of predictors (q)
- Model complexity (GLM family)

Custom Model Specification

The package provides interfaces for extending to custom distributions and estimation procedures.

Family Structure: A list containing GLM components:

family Character name of distribution

link Character name of link function

linkfun Function transforming response to linear predictor scale

linkinv Function transforming linear predictor to response scale

custom_dev.resids Optional function for GCV optimization criterion

Unconstrained Fitting: Core function for unconstrained coefficient estimation per partition:

```
function(X, y, LambdaHalf, Lambda, keep_weighted_Lambda, family,
         tol, K, parallel, cl, chunk_size, num_chunks, rem_chunks,
         order_indices, weights, status, ...) {
  # Returns p-length coefficient vector
}
```

Dispersion Estimation: Computes scale/dispersion parameter:

```
function(mu, y, order_indices, family, observation_weights, ...) {
  # Returns scalar dispersion estimate
  # Defaults to 1 (no dispersion)
}
```

GLM Weights: Computes observation weights:

```
function(mu, y, order_indices, family, dispersion,
         observation_weights, ...) {
  # Returns weights vector
  # Defaults to family variance with optional weights
}
```

Schur Corrections: Adjusts covariance matrix for parameter uncertainty:

```
function(X, y, B, dispersion, order_list, K, family,
         observation_weights, ...) {
 # Required for valid standard errors when dispersion affects coefficient estimation
}
```

Constraint Systems

Linear Equality Constraints: Linear constraints enforce exact relationships between coefficients, implemented through the Lagrangian multiplier approach and integrated with smoothness constraints.

Constraints are specified as $\mathbf{h}^T\boldsymbol{\beta} = \mathbf{h}^T\boldsymbol{\beta}_0$ via:

```
lgspline(
  y ~ spl(x),
  constraint_vectors = h,     # Matrix of constraint vectors
  constraint_values = beta0   # Target values
)
```

Common applications include:

- No-intercept models (via `no_intercept = TRUE`)
- Offset terms with coefficients constrained to 1
- Hypothesis testing with nested models

Inequality Constraints: Inequality constraints enforce bounds or directional relationships on the fitted function through sequential quadratic programming.

Built-in constraints include:

- Range constraints: `qp_range_lower` and `qp_range_upper`
- Monotonicity: `qp_monotonic_increase` or `qp_monotonic_decrease`
- Derivative constraints: `qp_positive_derivative` or `qp_negative_derivative`

Custom inequality constraints can be defined through:

- `qp_Amat_fxn`: Function returning constraint matrix
- `qp_bvec_fxn`: Function returning constraint vector
- `qp_meq_fxn`: Function returning number of equality constraints

Correlation Structure Estimation

Correlation patterns in clustered data are modeled using a matrix whitening approach based on restricted maximum likelihood (REML). The correlation structure is parameterized through the square root inverse matrix $\mathbf{V}^{-1/2}$.

Available Correlation Structures:

- **Exchangeable**: Constant correlation within clusters
- **Spatial Exponential**: Exponential decay with distance
- **AR(1)**: Correlation based on rank differences
- **Gaussian/Squared Exponential**: Smooth decay with squared distance
- **Spherical**: Polynomial decay with exact cutoff
- **Matern**: Flexible correlation with adjustable smoothness
- **Gamma-Cosine**: Combined gamma decay with oscillation
- **Gaussian-Cosine**: Combined Gaussian decay with oscillation

The REML objective function combines correlation structure, parameter estimates, and smoothness constraints:

$$\frac{1}{N}\big(- \log|\mathbf{V}^{-1/2}| + \frac{N}{2}\log(\sigma^2) + \frac{1}{2\sigma^2}(\mathbf{y} - \mathbf{X}\tilde{\boldsymbol{\beta}})^T\mathbf{V}^{-1}(\mathbf{y} - \mathbf{X}\tilde{\boldsymbol{\beta}}) + \frac{1}{2}\log|\sigma^2\mathbf{U}\mathbf{G}|\big)$$

Analytical gradients are provided for efficient optimization of correlation parameters.

Custom Correlation Functions: Custom correlation structures can be specified through:

- `VhalfInv_fxn`: Creates $\mathbf{V}^{-1/2}$
- `Vhalf_fxn`: Creates $\mathbf{V}^{1/2}$ (for non-Gaussian responses)
- `REML_grad`: Provides analytical gradient
- `VhalfInv_logdet`: Efficient determinant computation
- `custom_VhalfInv_loss`: Alternative optimization objective

Penalty Construction and Optimization

Smoothing Spline Penalty: The penalty matrix $\mathbf{\Lambda}$ is constructed as the integrated squared second derivative of the estimated function over the support of the predictors:

$$\int_a^b \|f''(t)\|^2 dt = \int_a^b \|\mathbf{x}''^\top \boldsymbol{\beta}_k\|^2 dt = \boldsymbol{\beta}_k^T \left\{ \int_a^b \mathbf{x}''\mathbf{x}''^\top dt \right\} \boldsymbol{\beta}_k = \boldsymbol{\beta}_k^T \mathbf{\Lambda}_s \boldsymbol{\beta}_k$$

For a one-dimensional cubic function, the second derivative basis is $\mathbf{x}'' = (0, 0, 2, 6t)^T$ and the penalty matrix has a closed-form expression:

$$\mathbf{\Lambda}_s = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 4(b-a) & 6(b^2-a^2) \\ 0 & 0 & 6(b^2-a^2) & 12(b^3-a^3) \end{pmatrix}$$

The full penalty matrix combines the smoothing spline penalty with a ridge penalty on non-spline terms and optional predictor-specific or partition-specific penalties:

$$\mathbf{\Lambda} = \lambda_s(\mathbf{\Lambda}_s + \lambda_r \mathbf{\Lambda}_r + \sum_{l=1}^{L} \lambda_l \mathbf{\Lambda}_l)$$

where:

- λ_s is the global smoothing parameter (`wiggle_penalty`)
- $\mathbf{\Lambda}_s$ is the smoothing spline penalty
- $\mathbf{\Lambda}_r$ is a ridge penalty on lower-order terms (`flat_ridge_penalty`)
- λ_l are optional predictor/partition-specific penalties

This penalty structure defines a meaningful metric for function smoothness, pulling estimates toward linear functions rather than simply shrinking coefficients toward zero.

Penalty Optimization: A penalized variant of Generalized Cross Validation (GCV) is used to find optimal penalties:

$$\text{GCV} = \frac{\sum_{i=1}^{N} r_i^2}{N\left(1 - \frac{1}{N}\text{trace}(\mathbf{XUGX}^T)\right)^2} + \frac{mp}{2} \sum_{l=1}^{L} (\log(1+\lambda_l) - 1)^2$$

where:

- $r_i = y_i - \tilde{y}_i$ are residuals (or replaced with custom alternative for GLMs)
- trace$(\mathbf{XUGX}^T)$ represents effective degrees of freedom
- mp is the "meta-penalty" term that regularizes predictor/partition-specific penalties

For GLMs, a pseudo-count approach is used to ensure valid link transformations, or `custom_dev.resids` can be provided to replace the sum-of-squared errors.

Optimization employs:

- Grid search for initial values
- Damped BFGS with analytical gradients
- Automated restarts and error handling
- Inflation factor $((N+2)/(N-2))^2$ for final penalties

References

Buse, A., & Lim, L. (1977). Cubic Splines as a Special Case of Restricted Least Squares. Journal of the American Statistical Association, 72, 64-68.

Craven, P., & Wahba, G. (1978). Smoothing noisy data with spline functions. Numerische Mathematik, 31, 377-403.

Eilers, P. H. C., & Marx, B. D. (1996). Flexible smoothing with B-splines and penalties. Statistical Science, 11, 89-121.

Ezhov, N., Neitzel, F., & Petrovic, S. (2018). Spline approximation, Part 1: Basic methodology. Journal of Applied Geodesy, 12, 139-155.

Kisi, O., Heddam, S., Parmar, K. S., Petroselli, A., Külls, C., & Zounemat-Kermani, M. (2025). Integration of Gaussian process regression and K means clustering for enhanced short term rainfall runoff modeling. Scientific Reports, 15, 7444.

Nocedal, J., & Wright, S. J. (2006). Numerical Optimization (2nd ed.). Springer.

Reinsch, C. H. (1967). Smoothing by spline functions. Numerische Mathematik, 10, 177-183.

Searle, S. R., Casella, G., & McCulloch, C. E. (2009). Variance Components. Wiley Series in Probability and Statistics. Wiley.

Silverman, B. W. (1984). Spline Smoothing: The Equivalent Variable Kernel Method. The Annals of Statistics, 12, 898-916.

Wahba, G. (1990). Spline Models for Observational Data. Society for Industrial and Applied Mathematics.

Wood, S. N. (2006). Generalized Additive Models: An Introduction with R. Chapman & Hall/CRC, Boca Raton.

`find_extremum` *Find Extremum of Fitted Lagrangian Multiplier Smoothing Spline*

Description

Finds global extrema of a fitted lgspline model using deterministic or stochastic optimization strategies. Supports custom objective functions for advanced applications like Bayesian optimization acquisition functions.

Usage

```
find_extremum(
  object,
  vars = NULL,
  quick_heuristic = TRUE,
  initial = NULL,
  B_predict = NULL,
  minimize = FALSE,
  stochastic = FALSE,
  stochastic_draw = function(mu, sigma, ...) {
```

```
    N <- length(mu)
    rnorm(N, mu,
   sigma)
},
 sigmasq_predict = object$sigmasq_tilde,
 custom_objective_function = NULL,
 custom_objective_derivative = NULL,
 ...
)
```

Arguments

`object`	A fitted lgspline model object containing partition information and fitted values
`vars`	Vector; A vector of numeric indices (or character variable names) of predictors to optimize for. If NULL (by default), all predictors will be optimized.
`quick_heuristic`	Logical; whether to search only the top-performing partition. When TRUE (default), optimizes within the best partition. When FALSE, initiates searches from all partition local maxima.
`initial`	Numeric vector; Optional initial values for optimization. Useful for fixing binary predictors or providing starting points. Default NULL
`B_predict`	Matrix; Optional custom coefficient list for prediction. Useful for posterior draws in Bayesian optimization. Default NULL
`minimize`	Logical; whether to find minimum instead of maximum. Default FALSE
`stochastic`	Logical; whether to add noise for stochastic optimization. Enables better exploration of the function space. Default FALSE
`stochastic_draw`	Function; Generates random noise/modifies predictions for stochastic optimization, analogous to posterior_predictive_draw. Takes three arguments: • mu: Vector of predicted values • sigma: Vector of standard deviations (square-root of sigmasq_tilde) • ...: Additional arguments to pass through Default `rnorm(length(mu), mu, sigma)`
`sigmasq_predict`	Numeric; Variance parameter for stochastic optimization. Controls the magnitude of random perturbations. Defaults to object$sigmasq_tilde
`custom_objective_function`	Function; Optional custom objective function for optimization. Takes arguments: • mu: Vector of predicted response values • sigma: Vector of standard deviations • y_best: Numeric; Best observed response value • ...: Additional arguments passed through Default NULL

`custom_objective_derivative`
: Function; Optional gradient function for custom optimization objective. Takes arguments:
  - mu: Vector of predicted response values
  - sigma: Vector of standard deviations
  - y_best: Numeric; Best observed response value
  - d_mu: Gradient of fitted function (for chain-rule computations)
  - ...: Additional arguments passed through

  Default NULL

`...`
: Additional arguments passed to internal optimization routines.

Details

This method finds extrema (maxima or minima) of the fitted function or composite functions of the fit. The optimization process can be customized through several approaches:

- Partition-based search: Either focuses on the top-performing partition (quick_heuristic = TRUE) or searches across all partition local maxima
- Stochastic optimization: Adds random noise during optimization for better exploration
- Custom objectives: Supports user-defined objective functions and gradients for specialized optimization tasks like Bayesian optimization

Value

A list containing the following components:

t Numeric vector of input values at the extremum.

y Numeric value of the objective function at the extremum.

See Also

`lgspline` for fitting the model, `generate_posterior` for generating posterior draws

Examples

```
## Basic usage with simulated data
set.seed(1234)
t <- runif(1000, -10, 10)
y <- 2*sin(t) + -0.06*t^2 + rnorm(length(t))
model_fit <- lgspline(t, y)
plot(model_fit)

## Find global maximum and minimum
max_point <- find_extremum(model_fit)
min_point <- find_extremum(model_fit, minimize = TRUE)
abline(v = max_point$t, col = 'blue')  # Add maximum point
abline(v = min_point$t, col = 'red')   # Add minimum point

## Advanced usage: custom objective functions
```

```
# expected improvement acquisition function
ei_custom_objective_function = function(mu, sigma, y_best, ...) {
  d <- y_best - mu
  d * pnorm(d/sigma) + sigma * dnorm(d/sigma)
}
# derivative of ei
ei_custom_objective_derivative = function(mu, sigma, y_best, d_mu, ...) {
  d <- y_best - mu
  z <- d/sigma
  d_z <- -d_mu/sigma
  pnorm(z)*d_mu - d*dnorm(z)*d_z + sigma*z*dnorm(z)*d_z
}

## Single iteration of Bayesian optimization
post_draw <- generate_posterior(model_fit)
acq <- find_extremum(model_fit,
                     stochastic = TRUE,  # Enable stochastic exploration
                     B_predict = post_draw$post_draw_coefficients,
                     sigmasq_predict = post_draw$post_draw_sigmasq,
                     custom_objective_function = ei_custom_objective_function,
                     custom_objective_derivative = ei_custom_objective_derivative)
abline(v = acq$t, col = 'green')  # Add acquisition point
```

---

generate_posterior *Generate Posterior Samples from Fitted Lagrangian Multiplier Smoothing Spline*

Description

Draws samples from the posterior distribution of model parameters and optionally generates posterior predictive samples. Uses Laplace approximation for non-Gaussian responses.

Usage

```
generate_posterior(
  object,
  new_sigmasq_tilde = object$sigmasq_tilde,
  new_predictors = object$X[[1]],
  theta_1 = 0,
  theta_2 = 0,
  posterior_predictive_draw = function(N, mean, sqrt_dispersion, ...) {
     rnorm(N,
    mean, sqrt_dispersion)
 },
  draw_dispersion = TRUE,
  include_posterior_predictive = FALSE,
  num_draws = 1,
```

```
  ...
)
```

Arguments

`object`	A fitted lgspline model object containing model parameters and fit statistics
`new_sigmasq_tilde`	Numeric; Dispersion parameter for sampling. Controls variance of posterior draws. Default object$sigmasq_tilde
`new_predictors`	Matrix; New data matrix for posterior predictive sampling. Should match structure of original predictors. Default = predictors as input to `lgspline`.
`theta_1`	Numeric; Shape parameter for prior gamma distribution of inverse-dispersion. Default 0 implies uniform prior
`theta_2`	Numeric; Rate parameter for prior gamma distribution of inverse-dispersion. Default 0 implies uniform prior
`posterior_predictive_draw`	Function; Random number generator for posterior predictive samples. Takes arguments: • N: Integer; Number of samples to draw • mean: Numeric vector; Predicted mean values • sqrt_dispersion: Numeric vector; Square root of dispersion parameter • ...: Additional arguments to pass through
`draw_dispersion`	Logical; whether to sample the dispersion parameter from its posterior distribution. When FALSE, uses point estimate. Default TRUE
`include_posterior_predictive`	Logical; whether to generate posterior predictive samples for new observations. Default FALSE
`num_draws`	Integer; Number of posterior draws to generate. Default 1
`...`	Additional arguments passed to internal sampling routines.

Details

Implements posterior sampling using the following approach:

- Coefficient posterior: Assumes sqrt(N)B ~ N(Btilde, sigma^2UG)
- Dispersion parameter: Sampled from inverse-gamma distribution with user-specified prior parameters (theta_1, theta_2) and model-based sufficient statistics
- Posterior predictive: Generated using custom sampling function, defaulting to Gaussian for standard normal responses

For the dispersion parameter, the sampling process follows for a fitted lgspline object "model_fit" (where unbias_dispersion is coerced to 1 if TRUE, 0 if FALSE)

```
shape <-  theta_1 + 0.5 * (model_fit$N - model_fit$unbias_dispersion * model_fit$trace_XUGX)
rate <- theta_2 + 0.5 * (model_fit$N - model_fit$unbias_dispersion * model_fit$trace_XUGX) * new_sigmas
post_draw_sigmasq <- 1/rgamma(1, shape, rate)
```

Users can modify sufficient statistics by adjusting theta_1 and theta_2 relative to the default model-based values.

Value

A list containing the following components:

post_draw_coefficients List of length num_draws containing posterior coefficient samples.

post_draw_sigmasq List of length num_draws containing posterior dispersion parameter samples (or repeated point estimate if draw_dispersion = FALSE).

post_pred_draw List of length num_draws containing posterior predictive samples (only if include_posterior_predictive = TRUE).

See Also

lgspline for model fitting, wald_univariate for Wald-type inference

Examples

```
## Generate example data
t <- runif(1000, -10, 10)
true_y <- 2*sin(t) + -0.06*t^2
y <- rnorm(length(true_y), true_y, 1)

## Fit model (using unstandardized expansions for consistent inference)
model_fit <- lgspline(t, y,
                      K = 7,
                      standardize_expansions_for_fitting = FALSE)

## Compare Wald (= t-intervals here) to Monte Carlo credible intervals
# Get Wald intervals
wald <- wald_univariate(model_fit,
                        cv = qt(0.975, df = model_fit$trace_XUGX))
wald_bounds <- cbind(wald[["interval_lb"]], wald[["interval_ub"]])

## Generate posterior samples (uniform prior)
post_draws <- generate_posterior(model_fit,
                                 theta_1 = -1,
                                 theta_2 = 0,
                                 num_draws = 2000)

## Convert to matrix and compute credible intervals
post_mat <- Reduce('cbind',
                   lapply(post_draws$post_draw_coefficients,
                          function(x) Reduce("rbind", x)))
post_bounds <- t(apply(post_mat, 1, quantile, c(0.025, 0.975)))

## Compare intervals
print(round(cbind(wald_bounds, post_bounds), 4))
```

`leave_one_out` *Compute Leave-One-Out Cross-Validated predictions for Gaussian Response/Identity Link under Constraint.*

Description

Computes the leave-one-out cross-validated predictions from a model fit, assuming Gaussian-distributed response with identity link.

Usage

```
leave_one_out(model_fit)
```

Arguments

`model_fit`	A fitted Lagrangian smoothing spline model

Value

A vector of leave-one-out cross-validated predictions

Examples

```
## Basic usage with Gaussian response, computing PRESS
set.seed(1234)
t <- rnorm(50)
y <- sin(t) + rnorm(50, 0, .25)
fit <- lgspline(t, y)
loo <- leave_one_out(fit)
press <- mean((y-loo)^2)

plot(loo, y,
   main = "Leave-One-Out Cross-Validation Prediction vs. Observed Response",
   xlab = 'Prediction', ylab = 'Response')
abline(0, 1)
```

`lgspline` *Fit Lagrangian Multiplier Smoothing Splines*

Description

A comprehensive software package for fitting a variant of smoothing splines as a constrained optimization problem, avoiding the need to algebraically disentangle a spline basis after fitting, and allowing for interpretable interactions and non-spline effects to be included.

`lgspline` fits piecewise polynomial regression splines constrained to be smooth where they meet, penalized by the squared, integrated, second-derivative of the estimated function with respect to predictors.

The method of Lagrangian multipliers is used to enforce the following:

- Equivalent fitted values at knots
- Equivalent first derivatives at knots, with respect to predictors
- Equivalent second derivatives at knots, with respect to predictors

The coefficients are penalized by an analytical form of the traditional cubic smoothing spline penalty, as well as tunable modifications that allow for unique penalization of multiple predictors and partitions.

This package supports model fitting for multiple spline and non-spline effects, GLM families, Weibull accelerated failure time (AFT) models, correlation structures, quadratic programming constraints, and extensive customization for user-defined models.

In addition, parallel processing capabilities and comprehensive tools for visualization, frequentist, and Bayesian inference are provided.

Usage

```
lgspline(predictors = NULL, y = NULL, formula = NULL, response = NULL,
            standardize_response = TRUE, standardize_predictors_for_knots = TRUE,
                standardize_expansions_for_fitting = TRUE, family = gaussian(),
           glm_weight_function = function(mu, y, order_indices, family, dispersion,
                                          observation_weights, ...) {
                  if(any(!is.null(observation_weights))){
                    family$variance(mu) * observation_weights
                  } else {
                    family$variance(mu)
                  }
           }, shur_correction_function = function(X, y, B, dispersion, order_list, K,
                                        family, observation_weights, ...) {
                  lapply(1:(K+1), function(k) 0)
                }, need_dispersion_for_estimation = FALSE,
                dispersion_function = function(mu, y, order_indices, family,
                                          observation_weights, ...) { 1 },
               K = NULL, custom_knots = NULL, cluster_on_indicators = FALSE,
               make_partition_list = NULL, previously_tuned_penalties = NULL,
             smoothing_spline_penalty = NULL, opt = TRUE, use_custom_bfgs = TRUE,
               delta = NULL, tol = 10*sqrt(.Machine$double.eps),
               invsoftplus_initial_wiggle = c(-25, 20, -15, -10, -5),
               invsoftplus_initial_flat = c(-14, -7), wiggle_penalty = 2e-07,
               flat_ridge_penalty = 0.5, unique_penalty_per_partition = TRUE,
```

```
     unique_penalty_per_predictor = TRUE, meta_penalty = 1e-08,
     predictor_penalties = NULL, partition_penalties = NULL,
     include_quadratic_terms = TRUE, include_cubic_terms = TRUE,
     include_quartic_terms = NULL, include_2way_interactions = TRUE,
include_3way_interactions = TRUE, include_quadratic_interactions = FALSE,
     offset = c(), just_linear_with_interactions = NULL,
just_linear_without_interactions = NULL, exclude_interactions_for = NULL,
     exclude_these_expansions = NULL, custom_basis_fxn = NULL,
include_constrain_fitted = TRUE, include_constrain_first_deriv = TRUE,
     include_constrain_second_deriv = TRUE,
  include_constrain_interactions = TRUE, cl = NULL, chunk_size = NULL,
  parallel_eigen = TRUE, parallel_trace = FALSE, parallel_aga = FALSE,
     parallel_matmult = FALSE, parallel_unconstrained = TRUE,
     parallel_find_neighbors = FALSE, parallel_penalty = FALSE,
     parallel_make_constraint = FALSE,
     unconstrained_fit_fxn = unconstrained_fit_default,
     keep_weighted_Lambda = FALSE, iterate_tune = TRUE,
     iterate_final_fit = TRUE, blockfit = FALSE,
     qp_score_function = function(X, y, mu, order_list, dispersion,
                                  VhalfInv, observation_weights, ...) {
       if(!is.null(observation_weights)) {
         crossprod(X, cbind((y - mu)*observation_weights))
       } else {
         crossprod(X, cbind(y - mu))
       }
 }, qp_observations = NULL, qp_Amat = NULL, qp_bvec = NULL, qp_meq = 0,
     qp_positive_derivative = FALSE, qp_negative_derivative = FALSE,
     qp_monotonic_increase = FALSE, qp_monotonic_decrease = FALSE,
    qp_range_upper = NULL, qp_range_lower = NULL, qp_Amat_fxn = NULL,
  qp_bvec_fxn = NULL, qp_meq_fxn = NULL, constraint_values = cbind(),
  constraint_vectors = cbind(), return_G = TRUE, return_Ghalf = TRUE,
return_U = TRUE, estimate_dispersion = TRUE, unbias_dispersion = NULL,
     return_varcovmat = TRUE, custom_penalty_mat = NULL,
     cluster_args = c(custom_centers = NA, nstart = 10),
     dummy_dividor = 1.2345672152894e-22,
     dummy_adder = 2.234567210529e-18, verbose = FALSE,
     verbose_tune = FALSE, expansions_only = FALSE,
     observation_weights = NULL, do_not_cluster_on_these = c(),
     neighbor_tolerance = 1 + 1e-08, null_constraint = NULL,
    critical_value = qnorm(1 - 0.05/2), data = NULL, weights = NULL,
     no_intercept = FALSE, correlation_id = NULL, spacetime = NULL,
     correlation_structure = NULL, VhalfInv = NULL, Vhalf = NULL,
     VhalfInv_fxn = NULL, Vhalf_fxn = NULL, VhalfInv_par_init = c(),
REML_grad = NULL, custom_VhalfInv_loss = NULL, VhalfInv_logdet = NULL,
     include_warnings = TRUE, ...)
```

Arguments

`predictors` Default: NULL. Numeric matrix or data frame of predictor variables. Supports direct matrix input or formula interface when used with 'data' argument. Must contain numeric predictors, with categorical variables pre-converted to numeric indicators.

`y` Default: NULL. Numeric response variable vector representing the target/outcome/dependent variable etc. to be modeled.

`formula` Default: NULL. Optional statistical formula for model specification, serving as an alternative to direct matrix input. Supports standard R formula syntax with special 'spl()' function for defining spline terms.

`response` Default: NULL. Alternative name for response variable, providing compatibility with different naming conventions. Takes precedence only if 'y' is not supplied.

`standardize_response` Default: TRUE. Logical indicator controlling whether the response variable should be centered by mean and scaled by standard deviation before model fitting. When TRUE, tends to improve numerical stability. Only offered for identity link functions (when family$link == 'identity')

`standardize_predictors_for_knots` Default: TRUE. Logical flag determining whether predictor variables should be standardized before knot placement. Ensures consistent knot selection across different predictor scales, and that no one predictor dominates in terms of influence on knot placement. For all expansions (x), standardization corresponds to dividing by the difference in 69 and 31st percentiles = x / (quantile(x, 0.69) - quantile(x, 0.31)).

`standardize_expansions_for_fitting` Default: TRUE. Logical switch to standardize polynomial basis expansions during model fitting. Provides computational stability during penalty tuning without affecting statistical inference, as design matrices, variance-covariance matrices, and coefficient estimates are systematically backtransformed after fitting to account for the standardization. If TRUE, $\mathbf{U}$ and $\mathbf{G}$ will remain on the transformed scale, and B_raw as returned will correspond to the coefficients fitted on the expansion-standardized scale.

`family` Default: gaussian(). Generalized linear model (GLM) distribution family specifying the error distribution and link function for model fitting. Defaults to Gaussian distribution with identity link. Supports custom family specifications, including user-defined link functions and optional custom tuning loss criteria. Minimally requires 1) family name (family) 2) link name (link) 3) linkfun (link function) 4) linkinv (link function inverse) 5) variance (mean variance relationship function, $\text{Var}(Y|\mu)$).

`glm_weight_function` Default: function that returns family$variance(mu) * observation_weights if weights exist, family$variance(mu) otherwise. Codes the mean-variance relationship of a GLM or GLM-like model, the diagonal $\mathbf{W}$ matrix of $\mathbf{X}^T\mathbf{W}\mathbf{X}$ that appears in the information. This can be replaced with a user-specified function. It is used for updating $\mathbf{G} = (\mathbf{X}^T\mathbf{W}\mathbf{X} + \mathbf{L})^{-1}$ after obtaining constrained estimates of coefficients. This is not used for fitting unconstrained models, but for iterating between updates of $\mathbf{U}$, $\mathbf{G}$, and beta coefficients afterwards.

`shur_correction_function`
: Default: function that returns list of zeros. Advanced function for computing Schur complements **S** to add to **G** to properly account for uncertainty in dispersion or other nuisance parameter estimation. The effective information becomes $\mathbf{G}^* = (\mathbf{G}^{-1} + \mathbf{S})^{-1}$.

`need_dispersion_for_estimation`
: Default: FALSE. Logical indicator specifying whether a dispersion parameter is required for coefficient estimation. This is not needed for canonical regular exponential family models, but is often needed otherwise (such as fitting Weibull AFT models).

`dispersion_function`
: Default: function that returns 1. Custom function for estimating the dispersion parameter. Unless `need_dispersion_for_estimation` is TRUE, this will not affect coefficient estimates.

`K`
: Default: NULL. Integer specifying the number of knot locations for spline partitions. This can intuitively be considered the total number of partitions - 1.

`custom_knots`
: Default: NULL. Optional matrix providing user-specified knot locations in 1-D.

`cluster_on_indicators`
: Default: FALSE. Logical flag determining whether indicator variables should be used for clustering knot locations.

`make_partition_list`
: Default: NULL. Optional list allowing direct specification of custom partition assignments. The intent is that the make_partition_list returned by one model can be supplied here to keep the same knot locations for another.

`previously_tuned_penalties`
: Default: NULL. Optional list of pre-computed penalty components from previous model fits.

`smoothing_spline_penalty`
: Default: NULL. Optional custom smoothing spline penalty matrix for fine-tuned complexity control.

`opt`
: Default: TRUE. Logical switch controlling whether model penalties should be automatically optimized via generalized cross-validation. Turn this off if previously_tuned_penalties are supplied AND desired, otherwise, the previously_tuned_penalties will be ignored.

`use_custom_bfgs`
: Default: TRUE. Logical indicator selecting between a native implementation of damped-BFGS optimization method with analytical gradients or base R's BFGS implementation with finite-difference approximation of gradients.

`delta`
: Default: NULL. Numeric pseudocount used for stabilizing optimization in non-identity link function scenarios.

`tol`
: Default: 10*sqrt(.Machine$double.eps). Numeric convergence tolerance controlling the precision of optimization algorithms.

`invsoftplus_initial_wiggle`
: Default: c(-25, 20, -15, -10, -5). Numeric vector of initial grid points for wiggle penalty optimization, specified on the inverse-softplus (softplus$(x) = \log(1 + e^x)$) scale.

`invsoftplus_initial_flat`
: Default: c(-7, 0). Numeric vector of initial grid points for ridge penalty optimization, specified on the inverse-softplus (softplus$(x) = \log(1 + e^x)$) scale.

`wiggle_penalty` Default: 2e-7. Numeric penalty controlling the integrated squared second derivative, governing function smoothness. Applied to both smoothing spline penalty (alone) and is multiplied by `flat_ridge_penalty` for penalizing linear terms.

`flat_ridge_penalty`
: Default: 0.5. Numeric flat ridge penalty for additional regularization on only intercepts and linear terms (won't affect interactions or quadratic/cubic/quartic terms by default). If `custom_penalty_mat` is supplied, the penalty will be for the custom penalty matrix instead. This penalty is multiplied with `wiggle_penalty` to obtain the total ridge penalty - hence, by default, the ridge penalization on linear terms is half of the magnitude of non-linear terms.

`unique_penalty_per_partition`
: Default: TRUE. Logical flag allowing the magnitude of the smoothing spline penalty to differ across partition.

`unique_penalty_per_predictor`
: Default: TRUE. Logical flag allowing the magnitude of the smoothing spline penalty to differ between predictors.

`meta_penalty` Default: 1e-8. Numeric "meta-penalty" applied to predictor and partition penalties during tuning. The minimization of GCV is modified to be a penalized minimization problem, with penalty $0.5 \times$ meta_penalty $\times (\sum \log(\text{penalty}))^2$, such that penalties are pulled towards 1 on the absolute scale and thus, their multiplicative effect towards 0.

`predictor_penalties`
: Default: NULL. Optional list of custom penalties specified per predictor.

`partition_penalties`
: Default: NULL. Optional list of custom penalties specified per partition.

`include_quadratic_terms`
: Default: TRUE. Logical switch to include squared predictor terms in basis expansions.

`include_cubic_terms`
: Default: TRUE. Logical switch to include cubic predictor terms in basis expansions.

`include_quartic_terms`
: Default: NULL. Logical switch to include quartic predictor terms in basis expansions. This is highly recommended for fitting models with multiple predictors to avoid over-specified constraints. When NULL (by default), will internally set to FALSE if only one predictor present, and TRUE otherwise.

`include_2way_interactions`
: Default: TRUE. Logical switch to include linear two-way interactions between predictors.

`include_3way_interactions`
: Default: TRUE. Logical switch to include three-way interactions between predictors.

`include_quadratic_interactions`
: Default: FALSE. Logical switch to include linear-quadratic interaction terms.

`offset` Default: Empty vector. When non-missing, this is a vector of column indices/names to include as offsets. `lgspline` will automatically introduce constraints such that the coefficient for offset terms are 1.

`just_linear_with_interactions`
Default: NULL. Integer vector specifying columns to retain linear terms with interactions.

`just_linear_without_interactions`
Default: NULL. Integer vector specifying columns to retain only linear terms without interactions.

`exclude_interactions_for`
Default: NULL. Integer vector indicating columns to exclude from all interaction terms.

`exclude_these_expansions`
Default: NULL. Character vector specifying basis expansions to be excluded from the model. These must be named columns of the data, or in the form "_1_", "_2_", "_1_x_2_", "_2_^2" etc. where "1" and "2" indicate column indices of predictor matrix input.

`custom_basis_fxn`
Default: NULL. Optional user-defined function for generating custom basis expansions. See `get_polynomial_expansions`.

`include_constrain_fitted`
Default: TRUE. Logical switch to constrain fitted values at knot points.

`include_constrain_first_deriv`
Default: TRUE. Logical switch to constrain first derivatives at knot points.

`include_constrain_second_deriv`
Default: TRUE. Logical switch to constrain second derivatives at knot points.

`include_constrain_interactions`
Default: TRUE. Logical switch to constrain interaction terms at knot points.

`cl` Default: NULL. Parallel processing cluster object for distributed computation (use `parallel::makeCluster()`).

`chunk_size` Default: NULL. Integer specifying custom fixed chunk size for parallel processing.

`parallel_eigen` Default: TRUE. Logical flag to enable parallel processing for eigenvalue decomposition computations.

`parallel_trace` Default: FALSE. Logical flag to enable parallel processing for trace computation.

`parallel_aga` Default: FALSE. Logical flag to enable parallel processing for specific matrix operations involving G and A.

`parallel_matmult`
Default: FALSE. Logical flag to enable parallel processing for block-diagonal matrix multiplication.

`parallel_unconstrained`
Default: TRUE. Logical flag to enable parallel processing for unconstrained maximum likelihood estimation.

`parallel_find_neighbors`
: Default: FALSE. Logical flag to enable parallel processing for neighbor identification (which partitions are neighbors).

`parallel_penalty`
: Default: FALSE. Logical flag to enable parallel processing for penalty matrix construction.

`parallel_make_constraint`
: Default: FALSE. Logical flag to enable parallel processing for constraint matrix generation.

`unconstrained_fit_fxn`
: Default: `unconstrained_fit_default`. Custom function for fitting unconstrained models per partition.

`keep_weighted_Lambda`
: Default: FALSE. Logical flag to retain generalized linear model weights in penalty constraints using Tikhonov parameterization. It is advised to turn this to TRUE when fitting non-canonical GLMs. The default `unconstrained_fit_default` by default assumes canonical GLMs for setting up estimating equations; this is not valid with non-canonical GLMs. With `keep_weighted_Lambda = TRUE`, the Tikhonov parameterization binds $\mathbf{\Lambda}^{1/2}$, the square-root penalty matrix, to the design matrix $\mathbf{X}_k$ for each partition k, and family$linkinv(0) to the response vector $\mathbf{y}_k$ for each partition before finding unconstrained estimates using base R's `glm.fit` function. The potential issue is that the weights of the information matrix will appear in the penalty, such that the effective penalty is $\mathbf{\Lambda}_{\text{eff}} = \mathbf{L}^{1/2}\mathbf{W}\mathbf{L}^{1/2}$ rather than just $\mathbf{L}^{1/2}\mathbf{L}^{1/2}$. If FALSE, this approach will only be used to supply initial values to a native implementation of damped Newton-Rapshon for fitting GLM models (see `damped_newton_r` and `unconstrained_fit_default`). For Gamma with log-link, this is fortunately a non-issue since the mean-variance relationship is essentially stabilized, so `keep_weighted_Lambda = TRUE` is strongly advised.

`iterate_tune`
: Default: TRUE. Logical switch to use iterative optimization during penalty tuning. If FALSE, $\mathbf{G}$ and $\mathbf{U}$ are constructed from unconstrained $\boldsymbol{\beta}$ estimates when tuning.

`iterate_final_fit`
: Default: TRUE. Logical switch to use iterative optimization for final model fitting. If FALSE, $\mathbf{G}$ and $\mathbf{U}$ are constructed from unconstrained $\boldsymbol{\beta}$ estimates when fitting the final model after tuning.

`blockfit`
: Default: FALSE. Logical switch to abandon per-partition fitting for non-spline effects without interactions, collapse all matrices into block-diagonal single-matrix form, and fit agnostic to partition. This would be more efficient for many non-spline effects without interactions and relatively few spline effects or non-spline effects with interactions. Ignored if `length(just_linear_without_interactions) = 0` after processing formulas and input.

`qp_score_function`
: Default: $\mathbf{X}^T(\mathbf{y} - \text{E}[\mathbf{y}])$, where $\text{E}[\mathbf{y}] = \boldsymbol{\mu}$. A function returning the score of the log-likelihood for optimization (excluding penalization/priors involving $\mathbf{\Lambda}$), which is needed for the formulation of quadratic programming problems, when `blockfit = TRUE`, and correlation-structure fitting for GLMs, all relying on `solve.QP`.

Accepts arguments "X, y, mu, order_list, dispersion, VhalfInv, observation_weights, ..." in order. As shown in the examples below, a gamma log-link model requires $\mathbf{X}^T\mathbf{W}(\mathbf{y} - \mathrm{E}[\mathbf{y}])$ instead, with $\mathbf{W}$ a diagonal matrix of $\mathrm{E}[\mathbf{y}]^2$ (Note: This example might be incorrect; check the specific score equation for Gamma log-link). This argument is not needed when fitting non-canonical GLMs without quadratic programming constraints or correlation structures, situations for which `keep_weighted_Lambda=TRUE` is sufficient.

`qp_observations`
: Default: NULL. Numeric vector of observations to apply constraints to for monotonic and range quadratic programming constraints. Useful for saving computational resources.

`qp_Amat`
: Default: NULL. Constraint matrix for quadratic programming formulation. The `Amat` argument of `solve.QP`.

`qp_bvec`
: Default: NULL. Constraint vector for quadratic programming formulation. The `bvec` argument of `solve.QP`.

`qp_meq`
: Default: 0. Number of equality constraints in quadratic programming setup. The `meq` argument of `solve.QP`.

`qp_positive_derivative`, `qp_monotonic_increase`
: Default: FALSE. Logical flags to constrain the function to have positive first derivatives/be monotonically increasing using quadratic programming with respect to the order (ascending rows) of the input data set.

`qp_negative_derivative`, `qp_monotonic_decrease`
: Default: FALSE. Logical flags to constrain the function to have negative first derivatives/be monotonically decreasing using quadratic programming with respect to the order (ascending rows) of the input data set.

`qp_range_upper`
: Default: NULL. Numeric upper bound for constrained fitted values using quadratic programming.

`qp_range_lower`
: Default: NULL. Numeric lower bound for constrained fitted values using quadratic programming.

`qp_Amat_fxn`
: Default: NULL. Custom function for generating Amat matrix in quadratic programming.

`qp_bvec_fxn`
: Default: NULL. Custom function for generating bvec vector in quadratic programming.

`qp_meq_fxn`
: Default: NULL. Custom function for determining meq equality constraints in quadratic programming.

`constraint_values`
: Default: `cbind()`. Matrix of constraint values for sum constraints. The constraint enforces $\mathbf{C}^T(\boldsymbol{\beta} - \mathbf{c}) = \mathbf{0}$ in addition to smoothing constraints, where $\mathbf{C}$ = `constraint_vectors` and $\mathbf{c}$ = `constraint_values`.

`constraint_vectors`
: Default: `cbind()`. Matrix of vectors for sum constraints. The constraint enforces $\mathbf{C}^T(\boldsymbol{\beta} - \mathbf{c}) = \mathbf{0}$ in addition to smoothing constraints, where $\mathbf{C}$ = `constraint_vectors` and $\mathbf{c}$ = `constraint_values`.

`return_G`
: Default: TRUE. Logical switch to return the unscaled variance-covariance matrix without smoothing constraints ($\mathbf{G}$).

`return_Ghalf` Default: TRUE. Logical switch to return the matrix square root of the unscaled variance-covariance matrix without smoothing constraints ($\mathbf{G}^{1/2}$).

`return_U` Default: TRUE. Logical switch to return the constraint projection matrix $\mathbf{U}$.

`estimate_dispersion`
Default: TRUE. Logical flag to estimate the dispersion parameter after fitting.

`unbias_dispersion`
Default NULL. Logical switch to multiply final dispersion estimates by $N/(N-\text{trace}(\mathbf{XUGX}^T))$, which in the case of Gaussian-distributed errors with identity link function, provides unbiased estimates of variance. When NULL (by default), gets set to TRUE for Gaussian + identity link and FALSE otherwise.

`return_varcovmat`
Default: TRUE. Logical switch to return the variance-covariance matrix of the estimated coefficients. This is needed for performing Wald inference.

`custom_penalty_mat`
Default: NULL. Optional $p \times p$ custom penalty matrix for individual partitions to replace the default ridge penalty applied to linear-and-intercept terms only. This can be interpreted as proportional to the prior correlation matrix of coefficients for non-spline effects, and will appear in the penalty matrix for all partitions. It is recommended to first run the function using `expansions_only = TRUE` so you have an idea of where the expansions appear in each partition, what "p" is, and you can carefully customize your penalty matrix after.

`cluster_args` Default: `c(custom_centers = NA, nstart = 10)`. Named vector of arguments controlling clustering procedures. If the first argument is not NA, this will be treated as custom cluster centers and all other arguments ignored. Otherwise, default base R k-means clustering will be used with all other arguments supplied to `kmeans` (for example, by default, the "nstart" argument as provided). Custom centers must be a $K \times q$ matrix with one column for each predictor in order of their appearance in input predictor/data, and one row for each center.

`dummy_dividor` Default: 0.00000000000000000000012345672152894. Small numeric constant to prevent division by zero in computational routines.

`dummy_adder` Default: 0.00000000000000000022345672105290. Small numeric constant to prevent division by zero in computational routines.

`verbose` Default: FALSE. Logical flag to print general progress messages during model fitting (does not include during tuning).

`verbose_tune` Default: FALSE. Logical flag to print detailed progress messages during penalty tuning specifically.

`expansions_only`
Default: FALSE. Logical switch to return only basis expansions without full model fitting. Useful for setting up custom constraints and penalties.

`observation_weights`
Default: NULL. Numeric vector of observation-specific weights for generalized least squares estimation.

`do_not_cluster_on_these`
Default: c(). Vector specifying predictor columns to exclude from clustering procedures, in addition to the non-spline effects by default.

`neighbor_tolerance`
: Default: 1 + 1e-8. Numeric tolerance for determining neighboring partitions using k-means clustering. Greater values means more partitions are likely to be considered neighbors. Intended for internal use only (modify at your own risk!).

`null_constraint`
: Default: NULL. Alternative parameterization of constraint values.

`critical_value`
: Default: `qnorm(1-0.05/2)`. Numeric critical value value used for constructing Wald confidence intervals of the form estimate$\pm$critical_value$\times$(standard error).

`data`
: Default: NULL. Optional data frame providing context for formula-based model specification.

`weights`
: Default: NULL. Alternative name for observation weights, maintained for interface compatibility.

`no_intercept`
: Default: FALSE. Logical flag to remove intercept, constraining it to 0. The function automatically constructs constraint_vectors and constraint_values to achieve this. Calling formulas with a "0+" in it like `y ~ 0 + .` will set this option to TRUE.

`correlation_id`, `spacetime`
: Default: NULL. N-length vector and N-row matrix of cluster (or subject, group etc.) ids and longitudinal/spatial variables respectively, whereby observations within each grouping of `correlation_id` are correlated with respect to the variables submitted to `spacetime`.

`correlation_structure`
: Default: NULL. Native implementations of popular variance-covariance structures. Offers options for "exchangeable", "spatial-exponential", "squared-exponential", "ar(1)", "spherical", "gaussian-cosine", "gamma-cosine", and "matern", along with their aliases. The eponymous correlation structure is fit along with coefficients and dispersion, with correlation estimated using a REML objective. See section "Correlation Structure Estimation" for more details.

`VhalfInv`
: Default: NULL. Matrix representing a fixed, custom square-root-inverse covariance structure for the response variable of longitudinal and spatial modeling. Must be an $N \times N$ matrix where N is number of observations. This matrix $\mathbf{V}^{-1/2}$ serves as a fixed transformation matrix for the response, equivalent to GLS with known covariance $\mathbf{V}$. This is known as "whitening" in some literature.

`Vhalf`
: Default: NULL. Matrix representing a fixed, custom square-root covariance structure for the response variable of longitudinal and spatial modeling. Must be an $N \times N$ matrix where N is number of observations. This matrix $\mathbf{V}^{1/2}$ is used when backtransforming coefficients for fitting GLMs with arbitrary correlation structure.

`VhalfInv_fxn`
: Default: NULL. Function for parametric modeling of the covariance structure $\mathbf{V}^{-1/2}$. Must take a single numeric vector argument "par" and return an $N \times N$ matrix. When provided with `VhalfInv_par_init`, this function is optimized via BFGS to find optimal covariance parameters that minimize the negative REML log-likelihood (or custom loss if `custom_VhalfInv_loss` is specified). The function must return a valid square root of the inverse covariance matrix - i.e., if $\mathbf{V}$ is the true covariance, `VhalfInv_fxn` should return $\mathbf{V}^{-1/2}$ such that `VhalfInv_fxn(par) * VhalfInv_fxn(par)` = $\mathbf{V}^{-1}$.

`Vhalf_fxn` Default: NULL. Function for efficient computation of $\mathbf{V}^{1/2}$, used only when optimizing correlation structures with non-canonical-Gaussian response.

`VhalfInv_par_init`
Default: c(). Numeric vector of initial parameter values for `VhalfInv_fxn` optimization. When provided with `VhalfInv_fxn`, triggers optimization of the covariance structure. Length determines the dimension of the parameter space. For example, for AR(1) correlation, this could be a single correlation parameter; for unstructured correlation, this could be all unique elements of the correlation matrix.

`REML_grad` Default: NULL. Function for evaluating the gradient of the objective function (negative REML or custom loss) with respect to the parameters of `VhalfInv_fxn`. Must take the same "par" argument as `VhalfInv_fxn`, as well as second argument "model_fit" for the output of `lgspline.fit` and ellipses "..." as a third argument. It should return a vector of partial derivatives matching the length of par. When provided, enables more efficient optimization via analytical gradients rather than numerical approximation. Optional - if NULL, BFGS uses numerical gradients.

`custom_VhalfInv_loss`
Default: NULL. Alternative to negative REML for serving as the objective function for optimizing correlation parameters. Must take the same "par" argument as `VhalfInv_fxn`, as well as second argument "model_fit" for the output of `lgspline.fit` and ellipses "..." as a third argument. It should return a numeric scalar.

`VhalfInv_logdet`
Default: NULL. Function for efficient computation of $\log|\mathbf{V}^{-1/2}|$ that bypasses construction of the full $\mathbf{V}^{-1/2}$ matrix. Must take the same parameter vector 'par' as `VhalfInv_fxn` and return a scalar value equal to $\log(\det(\mathbf{V}^{-1/2}))$. When NULL, the determinant is computed directly from `VhalfInv`, which can be computationally expensive for large matrices.

`include_warnings`
Default: TRUE. Logical switch to control display of warning messages during model fitting.

`...` Additional arguments passed to the unconstrained model fitting function.

Details

A flexible and interpretable implementation of smoothing splines including:

- Multiple predictors and interaction terms
- Various GLM families and link functions
- Correlation structures for longitudinal/clustered data
- Shape constraints via quadratic programming
- Parallel computation for large datasets
- Comprehensive inference tools

Value

A list containing the following components:

y Original response vector.

ytilde Fitted/predicted values on the scale of the response.

X List of design matrices $\mathbf{X}_k$ for each partition k, containing basis expansions including intercept, linear, quadratic, cubic, and interaction terms as specified.

A Constraint matrix **A** encoding smoothness constraints at knot points and any user-specified linear constraints.

B List of fitted coefficients $\boldsymbol{\beta}_k$ for each partition k on the original, unstandardized scale of the predictors and response.

B_raw List of fitted coefficients for each partition on the predictor-and-response standardized scale.

K Number of interior knots with one predictor (number of partitions minus 1 with > 1 predictor).

p Number of basis expansions of predictors per partition.

q Number of predictor variables.

P Total number of coefficients ($p \times (K + 1)$).

N Number of observations.

penalties List containing optimized penalty matrices and components:

- Lambda: Combined penalty matrix ($\boldsymbol{\Lambda}$), includes $\mathbf{L}_{\text{predictor\_list}}$ contributions but not $\mathbf{L}_{\text{partition\_list}}$.
- L1: Smoothing spline penalty matrix ($\mathbf{L}_1$).
- L2: Ridge penalty matrix ($\mathbf{L}_2$).
- L predictor list: Predictor-specific penalty matrices ($\mathbf{L}_{\text{predictor\_list}}$).
- L partition list: Partition-specific penalty matrices ($\mathbf{L}_{\text{partition\_list}}$).

knot_scale_transf Function for transforming predictors to standardized scale used for knot placement.

knot_scale_inv_transf Function for transforming standardized predictors back to original scale.

knots Matrix of knot locations on original unstandarized predictor scale for one predictor.

partition_codes Vector assigning observations to partitions.

partition_bounds Vector or matrix specifying the boundaries between partitions.

knot_expand_function Internal function for expanding data according to partition structure.

predict Function for generating predictions on new data.

assign_partition Function for assigning new observations to partitions.

family GLM family object specifying the error distribution and link function.

estimate_dispersion Logical indicating whether dispersion parameter was estimated.

unbias_dispersion Logical indicating whether dispersion estimates should be unbiased.

backtransform_coefficients Function for converting standardized coefficients to original scale.

forwtransform_coefficients Function for converting coefficients to standardized scale.

mean_y, sd_y Mean and standard deviation of response if standardized.

og_order Original ordering of observations before partitioning.

order_list List containing observation indices for each partition.

constraint_values, constraint_vectors Matrices specifying linear equality constraints if provided.

make_partition_list List containing partition information for > 1-D cases.

expansion_scales Vector of scaling factors used for standardizing basis expansions.

take_derivative, take_interaction_2ndderivative Functions for computing derivatives of basis expansions.

get_all_derivatives_insample Function for computing all derivatives on training data.

numerics Indices of numeric predictors used in basis expansions.

power1_cols, power2_cols, power3_cols, power4_cols Column indices for linear through quartic terms.

quad_cols Column indices for all quadratic terms (including interactions).

interaction_single_cols, interaction_quad_cols Column indices for linear-linear and linear-quadratic interactions.

triplet_cols Column indices for three-way interactions.

nonspline_cols Column indices for terms excluded from spline expansion.

return_varcovmat Logical indicating whether variance-covariance matrix was computed.

raw_expansion_names Names of basis expansion terms.

std_X, unstd_X Functions for standardizing/unstandardizing design matrices.

parallel_cluster_supplied Logical indicating whether a parallel cluster was supplied.

weights List of observation weights per partition.

G List of unscaled unconstrained variance-covariance matrices $\mathbf{G}_k$ per partition k if `return_G=TRUE`. Computed as $(\mathbf{X}_k^T\mathbf{X}_k + \mathbf{\Lambda}_{\text{eff}})^{-1}$ for partition k.

Ghalf List of $\mathbf{G}_k^{1/2}$ matrices if `return_Ghalf=TRUE`.

U Constraint projection matrix $\mathbf{U}$ if `return_U=TRUE`. For K=0 and no constraints, returns identity. Otherwise, returns $\mathbf{U} = \mathbf{I} - \mathbf{GA}(\mathbf{A}^T\mathbf{GA})^{-1}\mathbf{A}^T$. Used for computing the variance-covariance matrix $\sigma^2\mathbf{UG}$.

sigmasq_tilde Estimated (or fixed) dispersion parameter $\tilde{\sigma}^2$.

trace_XUGX Effective degrees of freedom (trace$(\mathbf{XUGX}^T)$).

varcovmat Variance-covariance matrix of coefficient estimates if `return_varcovmat=TRUE`.

VhalfInv The $\mathbf{V}^{-1/2}$ matrix used for implementing correlation structures, if specified.

VhalfInv_fxn, Vhalf_fxn, VhalfInv_logdet, REML_grad Functions for generating $\mathbf{V}^{-1/2}$, $\mathbf{V}^{1/2}$, $\log|\mathbf{V}^{-1/2}|$, and gradient of REML if provided.

VhalfInv_params_estimates Vector of estimated correlation parameters when using `VhalfInv_fxn`.

VhalfInv_params_vcov Approximate variance-covariance matrix of estimated correlation parameters from BFGS optimization.

wald_univariate Function for computing univariate Wald statistics and confidence intervals.

critical_value Critical value used for confidence interval construction.

generate_posterior Function for drawing from the posterior distribution of coefficients.

find_extremum Function for optimizing the fitted function.

plot Function for visualizing fitted curves.

quadprog_list List containing quadratic programming components if applicable.

When `expansions_only=TRUE` is used, a reduced list is returned containing only the following prior to any fitting or tuning:

X Design matrices $\mathbf{X}_k$

y Response vectors $\mathbf{y}_k$

A Constraint matrix $\mathbf{A}$

penalties Penalty matrices

order_list, og_order Ordering information

expansion_scales, colnm_expansions Scaling and naming information

K, knots Knot information

make_partition_list, partition_codes, partition_bounds Partition information

constraint_vectors, constraint_values Constraint information

quadprog_list Quadratic programming components if applicable

The returned object has class "lgspline" and provides comprehensive tools for model interpretation, inference, prediction, and visualization. All coefficients and predictions can be transformed between standardized and original scales using the provided transformation functions. The object includes both frequentist and Bayesian inference capabilities through Wald statistics and posterior sampling. Advanced customization options are available for analyzing arbitrarily complex study designs. See `Details` for descriptions of the model fitting process.

See Also

- `solve.QP` for quadratic programming optimization
- `plot_ly` for interactive plotting
- `kmeans` for k-means clustering
- `optim` for general purpose optimization routines

Examples

```
## ## ## ## Simple Examples ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ##
## Simulate some data, fit using default settings, and plot
set.seed(1234)
t <- runif(2500, -10, 10)
y <- 2*sin(t) + -0.06*t^2 + rnorm(length(t))
model_fit <- lgspline(t, y)
plot(t, y, main = 'Observed Data vs. Fitted Function, Colored by Partition')
plot(model_fit, add = TRUE)

## Repeat using logistic regression, with univariate inference shown
# and alternative function call
y <- rbinom(length(y), 1, 1/(1+exp(-std(y))))
df <- data.frame(t = t, y = y)
```

```
model_fit <- lgspline(y ~ spl(t),
                      df,
                      opt = FALSE, # no tuning penalties
                      family = quasibinomial())
plot(t, y, main = 'Observed Data vs Fitted Function with Formulas and Derivatives',
  ylim = c(-0.5, 1.05), cex.main = 0.8)
plot(model_fit,
     show_formulas = TRUE,
     text_size_formula = 0.65,
     legend_pos = 'bottomleft',
     legend_args = list(y.intersp = 1.1),
     add = TRUE)
## Notice how the coefficients match the formula, and expansions are
# homogenous across partitions without reparameterization
print(summary(model_fit))

## Overlay first and second derivatives of fitted function respectively
derivs <- predict(model_fit,
                  new_predictors = sort(t),
                  take_first_derivatives = TRUE,
                  take_second_derivatives = TRUE)
points(sort(t), derivs$first_deriv, col = 'gold', type = 'l')
points(sort(t), derivs$second_deriv, col = 'goldenrod', type = 'l')
legend('bottomright',
       col = c('gold','goldenrod'),
       lty = 1,
       legend = c('First Derivative', 'Second Derivative'))

## Simple 2D example - including a non-spline effect
z <- seq(-2, 2, length.out = length(y))
df <- data.frame(Predictor1 = t,
                 Predictor2 = z,
                 Response = sin(y)+0.1*z)
model_fit <- lgspline(Response ~ spl(Predictor1) + Predictor1*Predictor2,
                      df)

## Notice, while spline effects change over partitions,
# interactions and non-spline effects are constrained to remain the same
coefficients <- Reduce('cbind', coef(model_fit))
colnames(coefficients) <- paste0('Partition ', 1:(model_fit$K+1))
print(coefficients)

## One or two variables can be selected for plotting at a time
# even when >= 3 predictors are present
plot(model_fit,
      custom_title = 'Marginal Relationship of Predictor 1 and Response',
      vars = 'Predictor1',
      custom_response_lab = 'Response',
      show_formulas = TRUE,
      legend_pos = 'bottomright',
      digits = 4,
      text_size_formula = 0.5)
```

```
## 3D plots are implemented as well, retaining analytical formulas
my_plot <- plot(model_fit,
                show_formulas = TRUE,
                custom_response_lab = 'Response')
my_plot


## ## ## ## More Detailed 1D Example ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ##
## 1D data generating functions
t <- seq(-9, 9, length.out = 1000)
slinky <- function(x) {
  (50 * cos(x * 2) -2 * x^2 + (0.25 * x)^4 + 80)
}
coil <- function(x) {
  (100 * cos(x * 2) +-1.5 * x^2 + (0.1 * x)^4 +
  (0.05 * x^3) + (-0.01 * x^5) +
     (0.00002 * x^6) -(0.000001 * x^7) + 100)
}
exponential_log <- function(x) {
  unlist(c(sapply(x, function(xx) {
    if (xx <= 1) {
      100 * (exp(xx) - exp(1))
    } else {
      100 * (log(xx))
    }
  })))
}
scaled_abs_gamma <- function(x) {
  2*sqrt(gamma(abs(x)))
}

## Composite function
fxn <- function(x)(slinky(t) +
                   coil(t) +
                   exponential_log(t) +
                   scaled_abs_gamma(t))

## Bind together with random noise
dat <- cbind(t, fxn(t) + rnorm(length(t), 0, 50))
colnames(dat) <- c('t', 'y')
x <- dat[,'t']
y <- dat[,'y']

## Fit Model, 4 equivalent ways are shown below
model_fit <- lgspline(t, y)
model_fit <- lgspline(y ~ spl(t), as.data.frame(dat))
model_fit <- lgspline(response = y, predictors = t)
model_fit <- lgspline(data = as.data.frame(dat), formula = y ~ .)

# This is not valid: lgspline(y ~ ., t)
# This is not valid: lgspline(y, data = as.data.frame(dat))
# Do not put operations in formulas, not valid: lgspline(y ~ log(t) + spl(t))
```

```
## Basic Functionality
predict(model_fit, new_predictors = rnorm(1)) # make prediction on new data
head(leave_one_out(model_fit)) # leave-one-out cross-validated predictions
coef(model_fit) # extract coefficients
summary(model_fit) # model information and Wald inference
generate_posterior(model_fit) # generate draws of parameters from posterior distribution
find_extremum(model_fit, minimize = TRUE) # find the minimum of the fitted function

## Incorporate range constraints, custom knots, keep penalization identical
# across partitions
model_fit <- lgspline(y ~ spl(t),
                      unique_penalty_per_partition = FALSE,
                      custom_knots = cbind(c(-2, -1, 0, 1, 2)),
                      data = data.frame(t = t, y = y),
                      qp_range_lower = -150,
                      qp_range_upper = 150)

## Plotting the constraints and knots
plot(model_fit,
     custom_title = 'Fitted Function Constrained to Lie Between (-150, 150)',
     cex.main = 0.75)
# knot locations
abline(v = model_fit$knots)
# lower bound from quadratic program
abline(h = -150, lty = 2)
# upper bound from quadratic program
abline(h = 150, lty = 2)
# observed data
points(t, y, cex = 0.24)

## Enforce monotonic increasing constraints on fitted values
# K = 4 => 5 partitions
t <- seq(-10, 10, length.out = 100)
y <- 5*sin(t) + t + 2*rnorm(length(t))
model_fit <- lgspline(t,
                      y,
                      K = 4,
                      qp_monotonic_increase = TRUE)
plot(t, y, main = 'Monotonic Increasing Function with Respect to Fitted Values')
plot(model_fit,
     add = TRUE,
     show_formulas = TRUE,
     legend_pos = 'bottomright',
     custom_predictor_lab = 't',
     custom_response_lab = 'y')

## ## ## ## 2D Example using Volcano Dataset ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ##
## Prep
data('volcano')
volcano_long <-
  Reduce('rbind', lapply(1:nrow(volcano), function(i){
    t(sapply(1:ncol(volcano), function(j){
      c(i, j, volcano[i,j])
```

```
    }))
  }))
colnames(volcano_long) <- c('Length', 'Width', 'Height')

## Fit, with 50 partitions
# When fitting with > 1 predictor and large K, including quartic terms
# is highly recommended, and/or dropping the second-derivative constraint.
# Otherwise, the constraints can impose all partitions to be equal, with one
# cubic function fit for all (there isn't enough degrees of freedom to fit
# unique cubic functions due to the massive amount of constraints).
# Below, quartic terms are included and the constraint of second-derivative
# smoothness at knots is ignored.
model_fit <- lgspline(volcano_long[,c(1, 2)],
                      volcano_long[,3],
                      include_quadratic_interactions = TRUE,
                      K = 49,
                      opt = FALSE,
                      return_U = FALSE,
                      return_varcov = FALSE,
                      estimate_variance = TRUE,
                      return_Ghalf = FALSE,
                      return_G = FALSE,
                      include_constrain_second_deriv = FALSE,
                      unique_penalty_per_predictor = FALSE,
                      unique_penalty_per_partition = FALSE,
                      wiggle_penalty = 1e-5, # the fixed wiggle penalty
                      flat_ridge_penalty = 1e-4) # the ridge penalty

## Plotting on new data with interactive visual + formulas
new_input <- expand.grid(seq(min(volcano_long[,1]),
                             max(volcano_long[,1]),
                             length.out = 250),
                         seq(min(volcano_long[,2]),
                             max(volcano_long[,2]),
                             length.out = 250))
model_fit$plot(new_predictors = new_input,
               show_formulas = TRUE,
               custom_response_lab = "Height",
               custom_title = 'Volcano 3-D Map',
               digits = 2)

## ## ## ## Advanced Techniques using Trees Dataset ## ## ## ## ## ## ## ## ## ## ## ## ## ##
## Goal here is to introduce how lgspline works with non-canonical GLMs and
# demonstrate some custom features
data('trees')

## L1-regularization constraint function on standardized coefficients
# Bound all coefficients to be less than a certain value (l1_bound) in absolute
# magnitude such that | B^{(j)}_k | < lambda for all j = 1....p coefficients,
# and k = 1...K+1 partitions.
l1_constraint_matrix <- function(p, K) {
  ## Total number of coefficients
  P <- p * (K + 1)
```

```
  ## Create diagonal matrices for L1 constraint
  # First matrix: lamdba > -bound
  # Second matrix: -lambda > -bound
  first_diag <- diag(P)
  second_diag <- -diag(P)

  ## Combine matrices
  l1_Amat <- cbind(first_diag, second_diag)

  return(l1_Amat)
}

## Bounds absolute value of coefficients to be < l1_bound
l1_bound_vector <- function(qp_Amat,
                            scales,
                            l1_bound) {

  ## Combine matrices
  l1_bvec <- rep(-l1_bound, ncol(qp_Amat)) * c(1, scales)

  return(l1_bvec)
}

## Fit model, using predictor-response formulation, assuming
# Gamma-distributed response, and custom quadratic-programming constraints,
# with qp_score_function/glm_weight_function updated for non-canonical GLMs
# as well as quartic terms, keeping the effect of height constant across
# partitions, and 3 partitions in total. Hence, this is an advanced-usage
# case.
# You can modify this code for performing l1-regularization in general.
# For canonical GLMs, the default qp_score_function/glm_weight_function are
# correct and do not need to be changed
# (custom functionality is not needed for canonical GLMs).
model_fit <- lgspline(
  Volume ~ spl(Girth) + Height*Girth,
  data = with(trees, cbind(Girth, Height, Volume)),
  family = Gamma(link = 'log'),
  keep_weighted_Lambda = TRUE,
  glm_weight_function = function(
    mu,
    y,
    order_indices,
    family,
    dispersion,
    observation_weights,
   ...){
     rep(1/dispersion, length(y))
   },
   dispersion_function = function(
     mu,
     y,
     order_indices,
```

```
      family,
      observation_weights,
    ...){
     mean(
       mu^2/((y-mu)^2)
     )
  }, # = biased estimate of 1/shape parameter
  need_dispersion_for_estimation = TRUE,
  unbias_dispersion = TRUE, # multiply dispersion by N/(N-trace(XUGX^{T}))
  K = 2, # 3 partitions
  opt = FALSE, # keep penalties fixed
  unique_penalty_per_partition = FALSE,
  unique_penalty_per_predictor = FALSE,
  flat_ridge_penalty = 1e-64,
  wiggle_penalty = 1e-64,
  qp_score_function = function(X, y, mu, order_list, dispersion, VhalfInv,
    observation_weights, ...){
   t(X) %**% diag(c(1/mu * 1/dispersion)) %**% cbind(y - mu)
  }, # updated score for gamma regression with log link
  qp_Amat_fxn = function(N, p, K, X, colnm, scales, deriv_fxn, ...) {
    l1_constraint_matrix(p, K)
  },
  qp_bvec_fxn = function(qp_Amat, N, p, K, X, colnm, scales, deriv_fxn, ...) {
    l1_bound_vector(qp_Amat, scales, 25)
  },
  qp_meq_fxn = function(qp_Amat, N, p, K, X, colnm, scales, deriv_fxn, ...) 0
)

## Notice, interaction effect is constant across partitions as is the effect
# of Height alone
Reduce('cbind', coef(model_fit))
print(summary(model_fit))

## Plot results
plot(model_fit, custom_predictor_lab1 = 'Girth',
     custom_predictor_lab2 = 'Height',
     custom_response_lab = 'Volume',
     custom_title = 'Girth and Height Predicting Volume of Trees',
     show_formulas = TRUE)

## Verify magnitude of unstandardized coefficients does not exceed bound (25)
print(max(abs(unlist(model_fit$B))))

## Find height and girth where tree volume is closest to 42
# Uses custom objective that minimizes MSE discrepancy between predicted
# value and 42.
# The vanilla find_extremum function can be thought of as
# using "function(mu)mu" aka the identity function as the
# objective, where mu = "f(t)", our estimated function. The derivative is then
# d_mu = "f'(t)" with respect to predictors t.
# But with more creative objectives, and since we have machinery for
# f'(t) already available, we can compute gradients for (and optimize)
# arbitrary differentiable functions of our predictors too.
```

```
# For any objective, differentiate w.r.t. to mu, then multiply by d_mu to
# satisfy chain rule.
# Here, we have objective function: 0.5*(mu-42)^2
# and gradient                    : (mu-42)*d_mu
# and L-BFGS-B will be used to find the height and girth that most closely
# yields a prediction of 42 within the bounds of the observed data.
# The d_mu also takes into account link function transforms automatically
# for most common link functions, and will return warning + instructions
# on how to program the link-function derivatives otherwise.

## Custom acquisition functions for Bayesian optimization could be coded here.
find_extremum(
  model_fit,
  minimize = TRUE,
  custom_objective_function = function(mu, sigma, ybest, ...){
    0.5*(mu - 42)^2
  },
  custom_objective_derivative = function(mu, sigma, ybest, d_mu, ...){
    (mu - 42) * d_mu
  }
)

## ## ## ## How to Use Formulas in lgspline ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ##
## Demonstrates splines with multiple mixed predictors and interactions

## Generate data
n <- 2500
x <- rnorm(n)
y <- rnorm(n)
z <- sin(x)*mean(abs(y))/2

## Categorical predictors
cat1 <- rbinom(n, 1, 0.5)
cat2 <- rbinom(n, 1, 0.5)
cat3 <- rbinom(n, 1, 0.5)

## Response with mix of effects
response <- y + z + 0.1*(2*cat1 - 1)

## Continuous predictors re-named
continuous1 <- x
continuous2 <- z

## Combine data
dat <- data.frame(
  response = response,
  continuous1 = continuous1,
  continuous2 = continuous2,
  cat1 = cat1,
  cat2 = cat2,
  cat3 = cat3
)
```

```
## Example 1: Basic Model with Default Terms, No Intercept
# standardize_response = FALSE often needed when constraining intercepts to 0
fit1 <- lgspline(
  formula = response ~ 0 + spl(continuous1, continuous2) +
    cat1*cat2*continuous1 + cat3,
  K = 2,
  standardize_response = FALSE,
  data = dat
)
## Examine coefficients included
rownames(fit1$B$partition1)
## Verify intercept term is near 0 up to some numeric tolerance
abs(fit1$B[[1]][1]) < 1e-8

## Example 2: Similar Model with Intercept, Other Terms Excluded
fit2 <- lgspline(
  formula = response ~ spl(continuous1, continuous2) +
    cat1*cat2*continuous1 + cat3,
  K = 1,
  standardize_response = FALSE,
  include_cubic_terms = FALSE,
  exclude_these_expansions = c( # Not all need to actually be present
    '_batman_x_robin_',
    '_3_x_4_', # no cat1 x cat2 interaction, coded using column indices
    'continuous1xcontinuous2', # no continuous1 x continuous2 interaction
    'thejoker'
  ),
  data = dat
)
## Examine coefficients included
rownames(Reduce('cbind',coef(fit2)))
# Intercept will probably be present and non-0 now
abs(fit2$B[[1]][1]) < 1e-8

## ## ## ## Compare Inference to survreg for Weibull AFT Model Validation ##
# Only linear predictors, no knots, no penalties, using Weibull AFT Model
# The goal here is to ensure that for the special case of no spline effects
# and no knots, this implementation will be consistent with other model
# implementations.
# Also note, that when using models (like Weibull AFT) where dispersion is
# being estimated and is required for estimating beta coefficients,
# we use a shur complement correction function to adjust (or "correct") our
# variance-covariance matrix for both estimation and inference to account for
# uncertainty in estimating the dispersion.
# Typically the shur_correction_function would return a negative-definite
# matrix, as it's output is elementwise added to the information matrix prior
# to inversion.
require(survival)
df <- data.frame(na.omit(
  pbc[,c('time','trt','stage','hepato','bili','age','status')]
))

## Weibull AFT using lgspline, showing how some custom options can be used to
```

```
# fit more complicated models
model_fit <- lgspline(time ~ trt + stage + hepato + bili + age,
                      df,
                      family = weibull_family(),
                      need_dispersion_for_estimation = TRUE,
                      dispersion_function = weibull_dispersion_function,
                      glm_weight_function = weibull_glm_weight_function,
                      shur_correction_function = weibull_shur_correction,
                      unconstrained_fit_fxn = unconstrained_fit_weibull,
                      opt = FALSE,
                      wiggle_penalty = 0,
                      flat_ridge_penalty = 0,
                      K = 0,
                      status = pbc$status!=0)
print(summary(model_fit))

## Survreg results match closely on estimates and inference for coefficients
survreg_fit <- survreg(Surv(time, status!=0) ~ trt + stage + hepato + bili + age,
                       df)
print(summary(survreg_fit))

## sigmasq_tilde = scale^2 of survreg
print(c(sqrt(model_fit$sigmasq_tilde), survreg_fit$scale))

## ## ## ## Modelling Correlation Structures ## ## ## ## ## ## ## ## ## ## ## ## ## ## ##
## Setup
n_blocks <- 150 # Number of correlation_ids (subjects)
block_size <- 5 # Size of each correlation_ids (number of repeated measures per subj.)
N <- n_blocks * block_size # total sample size (balanced here)
rho_true <- 0.25  # True correlation

## Generate predictors and mean structure
t <- seq(-9, 9, length.out = N)
true_mean <- sin(t)

## Create block compound symmetric errors = I(1-p) + Jp
errors <- Reduce('rbind',
                 lapply(1:n_blocks,
                        function(i){
                          sigma <- diag(block_size) + rho_true *
                            (matrix(1, block_size, block_size) -
                               diag(block_size))
                          matsqrt(sigma) %*% rnorm(block_size)
                        }))

## Generate response with correlated errors
y <- true_mean + errors * 0.5

## Fit model with correlation structure
# include_warnings = FALSE is a good idea here, since many proposed
# correlations won't work
model_fit <- lgspline(t,
                      y,
```

```
                      K = 4,
                      correlation_id = rep(1:n_blocks, each = block_size),
                      correlation_structure = 'exchangeable',
                      include_warnings = FALSE
)

## Assess overall fit
plot(t, y, main = 'Sinosudial Fit Under Correlation Structure')
plot(model_fit, add = TRUE, show_formulas = TRUE, custom_predictor_lab = 't')

## Compare estimated vs true correlation
rho_est <- tanh(model_fit$VhalfInv_params_estimates)
print(c("True correlation:" = rho_true,
        "Estimated correlation:" = rho_est))

## Quantify uncertainty in correlation estimate with 95% confidence interval
se <- c(sqrt(diag(model_fit$VhalfInv_params_vcov))) / sqrt(model_fit$N)
ci <- tanh(model_fit$VhalfInv_params_estimates + c(-1.96, 1.96)*se)
print("95% CI for correlation:")
print(ci)

## Also check SD (should be close to 0.5)
print(sqrt(model_fit$sigmasq_tilde))

## Toeplitz Simulation Setup, with demonstration of custom functions
# and boilerplate. Toep is not implemented by default, because it makes
# strong assumptions on the study design and missingness that are rarely met,
# with non-obvious workarounds.
# If a GLM was to-be-fit, you'd also submit a function "Vhalf_fxn" analogous
# to VhalfInv_fxn with same argument (par) and an output of an N x N matrix
# that yields the inverse of VhalfInv_fxn output.
n_blocks <- 150   # Number of correlation_ids
block_size <- 4   # Observations per correlation_id
N <- n_blocks * block_size # total sample size
rho_true <- 0.5    # True correlation within correlation_ids
true_intercept <- 2     # True intercept
true_slope <- 0.5       # True slope for covariate

## Create design matrix with meaningful predictors
Tmat <- matrix(0, N, 2)
Tmat[,1] <- 1  # Intercept
Tmat[,2] <- cos(rnorm(N))  # Continuous predictor

## True coefficients
beta <- c(true_intercept, true_slope)

## Create time and correlation_id variables
time_var <- rep(1:block_size, n_blocks)
correlation_id_var <- rep(1:n_blocks, each = block_size)

## Create block compound symmetric errors
errors <- Reduce('rbind',
                 lapply(1:n_blocks, function(i) {
```

```
                    sigma <- diag(block_size) + rho_true *
                      (matrix(1, block_size, block_size) -
                         diag(block_size))
                    matsqrt(sigma) %*% rnorm(block_size)
                  }))

## Generate response with correlated errors and covariate effect
y <- Tmat %*% beta + errors * 2

## Toeplitz correlation function
VhalfInv_fxn <- function(par) {
  # Initialize correlation matrix
  corr <- matrix(0, block_size, block_size)

  # Construct Toeplitz matrix with correlation by lag
  for(i in 1:block_size) {
    for(j in 1:block_size) {
      lag <- abs(time_var[i] - time_var[j])
      if(lag == 0) {
        corr[i,j] <- 1
      } else if(lag <= length(par)) {
        # Use tanh to bound correlations between -1 and 1
        corr[i,j] <- tanh(par[lag])
      }
    }
  }

  ## Matrix square root inverse
  corr_inv_sqrt <- matinvsqrt(corr)

  ## Expand to full matrix using Kronecker product
  kronecker(diag(n_blocks), corr_inv_sqrt)
}

## Determinant function (for efficiency)
# This avoids taking determinant of N by N matrix
VhalfInv_logdet <- function(par) {
  # Initialize correlation matrix
  corr <- matrix(0, block_size, block_size)

  # Construct Toeplitz matrix
  for(i in 1:block_size) {
    for(j in 1:block_size) {
      lag <- abs(time_var[i] - time_var[j])
      if(lag == 0) {
        corr[i,j] <- 1
      } else if(lag <= length(par)) {
        corr[i,j] <- tanh(par[lag])
      }
    }
  }

  # Compute log determinant
```

```
  log_det_invsqrt_corr <- -0.5 * determinant(corr, logarithm=TRUE)$modulus[1]
  return(n_blocks * log_det_invsqrt_corr)
}

## REML gradient function
REML_grad <- function(par, model_fit, ...) {
  ## Initialize gradient vector
  n_par <- length(par)
  gradient <- numeric(n_par)

  ## Get dimensions and organize data
  nr <- nrow(model_fit$X[[1]])

  ## Process derivatives one parameter at a time
  for(p in 1:n_par) {
    ## Initialize derivative matrix
    dV <- matrix(0, nrow(model_fit$VhalfInv), ncol(model_fit$VhalfInv))
    V <- matrix(0, nrow(model_fit$VhalfInv), ncol(model_fit$VhalfInv))

    ## Compute full correlation matrix and its derivative for parameter p
    for(clust in unique(correlation_id_var)) {
      inds <- which(correlation_id_var == clust)
      block_size <- length(inds)

      ## Initialize block matrices
      V_block <- matrix(0, block_size, block_size)
      dV_block <- matrix(0, block_size, block_size)

      ## Construct Toeplitz matrix and its derivative
      for(i in 1:block_size) {
        for(j in 1:block_size) {
          ## Compute lag between observations
          lag <- abs(time_var[i] - time_var[j])

          ## Diagonal is always 1
          if(i == j) {
            V_block[i,j] <- 1
            dV_block[i,j] <- 0
          } else {
            ## Correlation for off-diagonal depends on lag
            if(lag <= length(par)) {
              ## Correlation via tanh parameterization
              V_block[i,j] <- tanh(par[lag])

              ## Derivative for the relevant parameter
              if(lag == p) {
                ## Chain rule for tanh: d/dx tanh(x) = 1 - tanh^2(x)
                dV_block[i,j] <- 1 - tanh(par[p])^2
              }
            }
          }
        }
      }
```

```
  ## Assign blocks to full matrices
  V[inds, inds] <- V_block
  dV[inds, inds] <- dV_block
}

## GLM Weights based on current model fit (all 1s for normal)
glm_weights <- rep(1, model_fit$N)

## Quadratic form contribution
resid <- model_fit$y - model_fit$ytilde
VinvResid <- model_fit$VhalfInv %**% cbind(resid) / glm_weights
quad_term <- -0.5 * ((t(VinvResid) %**% dV) %**% VinvResid) /
  model_fit$sigmasq_tilde

## Log|V| contribution - trace term
trace_term <- 0.5 * sum(diag(model_fit$VhalfInv %**%
                               model_fit$VhalfInv %**%
                               dV))

## Information matrix contribution
U <- t(t(model_fit$U) * rep(c(1, model_fit$expansion_scales),
                            model_fit$K + 1)) /
  model_fit$sd_y
VhalfInvX <- model_fit$VhalfInv %**%
  collapse_block_diagonal(model_fit$X)[unlist(
    model_fit$og_order
  ),] %**%
  U

## Lambda computation for GLMs
if(length(model_fit$penalties$L_partition_list) != (model_fit$K + 1)){
  model_fit$penalties$L_partition_list <- lapply(
    1:(model_fit$K + 1), function(k)0
  )
}
Lambda <- U %**% collapse_block_diagonal(
  lapply(1:(model_fit$K + 1),
         function(k)
           c(1, model_fit$expansion_scales) * (
             model_fit$penalties$L_partition_list[[k]] +
               model_fit$penalties$Lambda) %**%
           diag(c(1, model_fit$expansion_scales)) /
           model_fit$sd_y^2
  )
) %**% t(U)

XVinvX_inv <- invert(gramMatrix(t(t(VhalfInvX)*c(glm_weights))) +
                       Lambda)
VInvX <- model_fit$VhalfInv %**% VhalfInvX
sc <- sqrt(norm(VInvX, '2'))
VInvX <- VInvX/sc
dXVinvX <-
```

```
      (XVinvX_inv %**% t(VInvX)) %**%
      (dV %**% VInvX)
    XVinvX_term <- -0.5 * colSums(cbind(c(diag(dXVinvX) * sc))) * sc

    ## Store gradient component (all three terms)
    gradient[p] <- as.numeric(quad_term + trace_term + XVinvX_term)
  }

  ## Return normalized gradient
  return(gradient / model_fit$N)
}

## Visualization
plot(time_var, y, col = correlation_id_var,
     main = "Simulated Data with Toeplitz Correlation")

## Fit model with custom Toeplitz (takes ~ 5-10 minutes on my laptop)
# Note, for GLMs, efficiency would be improved by supplying a Vhalf_fxn
# although strictly only VhalfInv_fxn and VhalfInv_par_init are needed
model_fit <- lgspline(
  response = y,
  predictors = Tmat[,2],
  K = 4,
  VhalfInv_fxn = VhalfInv_fxn,
  VhalfInv_logdet = VhalfInv_logdet,
  REML_grad = REML_grad,
  VhalfInv_par_init = c(0, 0, 0),
  include_warnings = FALSE
)

## Print comparison of true and estimated correlations
rho_true <- rep(0.5, 3)
rho_est <- tanh(model_fit$VhalfInv_params_estimates)
cat("Correlation Estimates:\n")
print(data.frame(
  "True Correlation" = rho_true,
  "Estimated Correlation" = rho_est
))

## Should be ~ 2
print(sqrt(model_fit$sigmasq_tilde))

## ## ## ## Parallelism ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ##
## Data generating function
a <- runif(500000, -9, 9)
b <- runif(500000, -9, 9)
c <- rnorm(500000)
d <- rpois(500000, 1)
y <- sin(a) + cos(b) - 0.2*sqrt(a^2 + b^2) +
  abs(a) + b +
  0.5*(a^2 + b^2) +
  (1/6)*(a^3 + b^3) +
  a*b*c -
```

```
  c +
  d +
  rnorm(500000, 0, 5)

## Set up cores
cl <- parallel::makeCluster(1)
on.exit(parallel::stopCluster(cl))

## This example shows some options for what operations can be parallelized
# By default, only parallel_eigen and parallel_unconstrained are TRUE
# G, G^{-1/2}, and G^{1/2} are computed in parallel across each of the
# K+1 partitions.
# However, parallel_unconstrained only affects GLMs without corr. components
# - it does not affect fitting here
system.time({
  parfit <- lgspline(y ~ spl(a, b) + a*b*c + d,
                     data = data.frame(y = y,
                                       a = a,
                                       b = b,
                                       c = c,
                                       d = d),
                     cl = cl,
                     K = 1,
                     parallel_eigen = TRUE,
                     parallel_unconstrained = TRUE,
                     parallel_aga = FALSE,
                     parallel_find_neighbors = FALSE,
                     parallel_trace = FALSE,
                     parallel_matmult = FALSE,
                     parallel_make_constraint = FALSE,
                     parallel_penalty = FALSE)
})
parallel::stopCluster(cl)
print(summary(parfit))
```

`loglik_weibull` *Compute Log-Likelihood for Weibull Accelerated Failure Time Model*

Description

Calculates the log-likelihood for a Weibull accelerated failure time (AFT) survival model, supporting right-censored survival data.

Usage

```
loglik_weibull(log_y, log_mu, status, scale, weights = 1)
```

Arguments

log_y	Numeric vector of logarithmic response/survival times
log_mu	Numeric vector of logarithmic predicted survival times
status	Numeric vector of censoring indicators (1 = event, 0 = censored) Indicates whether an event of interest occurred (1) or the observation was right-censored (0). In survival analysis, right-censoring occurs when the full survival time is unknown, typically because the study ended or the subject was lost to follow-up before the event of interest occurred.
scale	Numeric scalar representing the Weibull scale parameter
weights	Optional numeric vector of observation weights (default = 1)

Details

The function computes log-likelihood contributions for a Weibull AFT model, explicitly accounting for right-censored observations. It supports optional observation weighting to accommodate complex sampling designs.

This both provides a tool for actually fitting Weibull AFT models, and boilerplate code for users who wish to incorporate Lagrangian multiplier smoothing splines into their own custom models.

Value

A numeric scalar representing the total log-likelihood of the model

Examples

```
## Minimal example of fitting a Weibull Accelerated Failure Time model
# Simulating survival data with right-censoring
set.seed(1234)
x1 <- rnorm(1000)
x2 <- rbinom(1000, 1, 0.5)
yraw <- rexp(exp(0.01*x1 + 0.01*x2))
# status: 1 = event occurred, 0 = right-censored
status <- rbinom(1000, 1, 0.25)
yobs <- ifelse(status, runif(1, 0, yraw), yraw)
df <- data.frame(
  y = yobs,
  x1 = x1,
  x2 = x2
)

## Fit model using lgspline with Weibull AFT specifics
model_fit <- lgspline(y ~ spl(x1) + x2,
                      df,
                      unconstrained_fit_fxn = unconstrained_fit_weibull,
                      family = weibull_family(),
                      need_dispersion_for_estimation = TRUE,
                      dispersion_function = weibull_dispersion_function,
                      glm_weight_function = weibull_glm_weight_function,
                      shur_correction_function = weibull_shur_correction,
```

```
                    status = status,
                    opt = FALSE,
                    K = 1)

loglik_weibull(log(model_fit$y), log(model_fit$ytilde), status,
  sqrt(model_fit$sigmasq_tilde))
```

`matinvsqrt` *Calculate Matrix Inverse Square Root*

Description

Calculate Matrix Inverse Square Root

Usage

```
matinvsqrt(mat)
```

Arguments

`mat`	A symmetric, positive-definite matrix $\mathbf{M}$

Details

For matrix $\mathbf{M}$, computes $\mathbf{B}$ where $\mathbf{BB} = \mathbf{M}^{-1}$ using eigenvalue decomposition:

1. Compute eigendecomposition $\mathbf{M} = \mathbf{VDV}^T$
2. Set eigenvalues below `sqrt(.Machine$double.eps)` to 0
3. Take elementwise reciprocal square root: $\mathbf{D}^{-1/2}$
4. Reconstruct as $\mathbf{B} = \mathbf{VD}^{-1/2}\mathbf{V}^T$

For nearly singular matrices, eigenvalues below the numerical threshold are set to 0, and their reciprocals in $\mathbf{D}^{-1/2}$ are also set to 0.

This implementation is particularly useful for whitening procedures in GLMs with correlation structures and for computing variance-covariance matrices under constraints.

You can use this to help construct a custom `VhalfInv_fxn` for fitting correlation structures, see `lgspline`.

Value

A matrix $\mathbf{B}$ such that $\mathbf{BB} = \mathbf{M}^{-1}$

Examples

```
## Identity matrix
m1 <- diag(2)
matinvsqrt(m1)  # Returns identity matrix

## Compound symmetry correlation matrix
rho <- 0.5
m2 <- matrix(rho, 3, 3) + diag(1-rho, 3)
B <- matinvsqrt(m2)
# Verify: B %**% B approximately equals solve(m2)
all.equal(B %**% B, solve(m2))

## Example for GLM correlation structure
n_blocks <- 2  # Number of subjects
block_size <- 3  # Measurements per subject
rho <- 0.7  # Within-subject correlation
# Correlation matrix for one subject
R <- matrix(rho, block_size, block_size) +
     diag(1-rho, block_size)
## Full correlation matrix for all subjects
V <- kronecker(diag(n_blocks), R)
## Create whitening matrix
VhalfInv <- matinvsqrt(V)

# Example construction of VhalfInv_fxn for lgspline
VhalfInv_fxn <- function(par) {
  rho <- tanh(par)  # Transform parameter to (-1, 1)
  R <- matrix(rho, block_size, block_size) +
       diag(1-rho, block_size)
  kronecker(diag(n_blocks), matinvsqrt(R))
}
```

`matsqrt` *Calculate Matrix Square Root*

Description

Calculate Matrix Square Root

Usage

```
matsqrt(mat)
```

Arguments

`mat`	A symmetric, positive-definite matrix **M**

Details

For matrix $\mathbf{M}$, computes $\mathbf{B}$ where $\mathbf{BB} = \mathbf{M}$ using eigenvalue decomposition:

1. Compute eigendecomposition $\mathbf{M} = \mathbf{VDV}^T$
2. Set eigenvalues below `sqrt(.Machine$double.eps)` to 0 for stability
3. Take elementwise square root of eigenvalues: $\mathbf{D}^{1/2}$
4. Reconstruct as $\mathbf{B} = \mathbf{VD}^{1/2}\mathbf{V}^T$

This provides the unique symmetric positive-definite square root.

You can use this to help construct a custom `Vhalf_fxn` for fitting correlation structures, see `lgspline`.

Value

A matrix $\mathbf{B}$ such that $\mathbf{BB} = \mathbf{M}$

Examples

```
## Identity matrix
m1 <- diag(2)
matsqrt(m1)  # Returns identity matrix

## Compound symmetry correlation matrix
rho <- 0.5
m2 <- matrix(rho, 3, 3) + diag(1-rho, 3)
B <- matsqrt(m2)
# Verify: B %**% B approximately equals m2
all.equal(B %**% B, m2)

## Example for correlation structure
n_blocks <- 2  # Number of subjects
block_size <- 3  # Measurements per subject
rho <- 0.7  # Within-subject correlation
# Correlation matrix for one subject
R <- matrix(rho, block_size, block_size) +
     diag(1-rho, block_size)
# Full correlation matrix for all subjects
V <- kronecker(diag(n_blocks), R)
Vhalf <- matsqrt(V)
```

`plot.lgspline` *Plot Method for Lagrangian Multiplier Smoothing Spline Models*

Description

Creates visualizations of fitted spline models, supporting both 1D line plots and 2D surface plots with optional formula annotations and customizable aesthetics. (Wrapper for internal plot method)

Usage

```
## S3 method for class 'lgspline'
plot(
  x,
  show_formulas = FALSE,
  digits = 4,
  legend_pos = "topright",
  custom_response_lab = "y",
  custom_predictor_lab = NULL,
  custom_predictor_lab1 = NULL,
  custom_predictor_lab2 = NULL,
  custom_formula_lab = NULL,
  custom_title = "Fitted Function",
  text_size_formula = NULL,
  legend_args = list(),
  new_predictors = NULL,
  xlim = NULL,
  ylim = NULL,
  color_function = NULL,
  add = FALSE,
  vars = c(),
  ...
)
```

Arguments

x	A fitted lgspline model object containing the model fit to be plotted
show_formulas	Logical; whether to display analytical formulas for each partition. Default FALSE
digits	Integer; Number of decimal places for coefficient display in formulas. Default 4
legend_pos	Character; Position of legend for 1D plots ("top", "bottom", "left", "right", "topleft", etc.). Default "topright"
custom_response_lab	Character; Label for response variable axis. Default "y"
custom_predictor_lab	Character; Label for predictor axis in 1D plots. If NULL (default), uses predictor column name
custom_predictor_lab1	Character; Label for first predictor axis (x1) in 2D plots. If NULL (default), uses first predictor column name
custom_predictor_lab2	Character; Label for second predictor axis (x2) in 2D plots. If NULL (default), uses second predictor column name
custom_formula_lab	Character; Label for fitted response on link function scale. If NULL (default), uses "link(E[custom_response_lab])" for non-Gaussian models with non-identity link, otherwise uses custom_response_lab

`custom_title`	Character; Main plot title. Default "Fitted Function"
`text_size_formula`	Numeric; Text size for formula display. Passed to cex in legend() for 1D plots and hover font size for 2D plots. If NULL (default), uses 0.8 for 1D and 8 for 2D
`legend_args`	List; Additional arguments passed to legend() for 1D plots
`new_predictors`	Matrix; Optional new predictor values for prediction. If NULL (default), uses original fitting data
`xlim`	Numeric vector; Optional x-axis limits for 1D plots. Default NULL
`ylim`	Numeric vector; Optional y-axis limits for 1D plots. Default NULL
`color_function`	Function; Returns colors for plotting by partition, must return K+1 vector of valid colors. Defaults to NULL, in which case `grDevices::rainbow(K+1)` is used for 1D and `grDevices::colorRampPalette(RColorBrewer::brewer.pal(8, "Spectral"))(K+1)` used in multiple.
`add`	Logical; If TRUE, adds to existing plot (1D only). Similar to add in `hist`. Default FALSE
`vars`	Numeric or character vector; Optional indices for selecting variables to plot. Can either be numeric (the column indices of "predictors" or "data") or character (the column names, if available from "predictors" or "data")
`...`	Additional arguments passed to underlying plot functions: • 1D: Passed to `plot` • 2D: Passed to `plot_ly`

Details

Produces different visualizations based on model dimensionality:

- 1D models: Line plot showing fitted function across partitions, with optional data points and formula annotations
- 2D models: Interactive 3D surface plot using plotly, with hover text showing predicted values and optional formula display

Partition boundaries are indicated by color changes in both 1D and 2D plots.

When plotting using "select_vars" option, it is recommended to use the "new_predictors" argument to set all terms not involved with plotting to 0 to avoid non-sensical results. But for some cases, it may be useful to set other predictors fixed at certain values. By default, observed values in the data set are used.

The function relies on linear expansions being present - if (for example) a user includes the argument "_1_" or "_2_" in "exclude_these_expansions", then this function will not be able to extract the predictors needed for plotting.

For this case, try constraining the effects of these terms to 0 instead using "constraint_vectors" and "constraint_values" argument, so they are kept in the expansions but their corresponding coefficients will be 0.

Value

Returns

1D Invisibly returns NULL (base R plot is drawn to device).

2D Plotly object showing interactive surface plot.

See Also

`lgspline` for model fitting, `plot` for additional 1D plot parameters, `plot_ly` for additional 2D plot parameters

Examples

```
## Generate example data
set.seed(1234)
t_data <- runif(1000, -10, 10)
y_data <- 2*sin(t_data) + -0.06*t_data^2 + rnorm(length(t_data))

## Fit model with 10 partitions
model_fit <- lgspline(t_data, y_data, K = 9)

## Basic plot
plot(model_fit)

## Customized plot with formulas
plot(model_fit,
     show_formulas = TRUE,         # Show partition formulas
     custom_response_lab = 'Price',  # Custom axis labels
     custom_predictor_lab = 'Size',
     custom_title = 'Price vs Size', # Custom title
     digits = 2,                    # Round coefficients
     legend_pos = 'bottom',         # Move legend
     text_size_formula = 0.375,     # Adjust formula text size
     pch = 16,                      # Point style
     cex.main = 1.25)               # Title size
```

---

predict.lgspline *Predict Method for Fitted Lagrangian Multiplier Smoothing Spline*

Description

Generates predictions, derivatives, and basis expansions from a fitted lgspline model. Supports both in-sample and out-of-sample prediction with optional parallel processing. (Wrapper for internal predict method)

Usage

```
## S3 method for class 'lgspline'
predict(
  object,
  newdata = NULL,
  parallel = FALSE,
  cl = NULL,
  chunk_size = NULL,
  num_chunks = NULL,
  rem_chunks = NULL,
  B_predict = NULL,
  take_first_derivatives = FALSE,
  take_second_derivatives = FALSE,
  expansions_only = FALSE,
  new_predictors = NULL,
  ...
)
```

Arguments

`object`	A fitted lgspline model object containing model parameters and fit
`newdata`	Matrix or data.frame; New predictor values for out-of-sample prediction. If NULL (default), uses training data
`parallel`	Logical; whether to use parallel processing for prediction computations. Experimental feature - use with caution. Default FALSE
`cl`	Optional cluster object for parallel processing. Required if parallel=TRUE. Default NULL
`chunk_size`	Integer; Size of computational chunks for parallel processing. Default NULL
`num_chunks`	Integer; Number of chunks for parallel processing. Default NULL
`rem_chunks`	Integer; Number of remainder chunks for parallel processing. Default NULL
`B_predict`	Matrix; Optional custom coefficient matrix for prediction. Default NULL (uses object$B internally).
`take_first_derivatives`	Logical; whether to compute first derivatives of the fitted function. Default FALSE
`take_second_derivatives`	Logical; whether to compute second derivatives of the fitted function. Default FALSE
`expansions_only`	Logical; whether to return only basis expansions without computing predictions. Default FALSE
`new_predictors`	Matrix or data frame; overrides 'newdata' if provided.
`...`	Additional arguments passed to internal prediction methods.

Details

Implements multiple prediction capabilities:

- Standard prediction: Returns fitted values for new data points
- Derivative computation: Calculates first and/or second derivatives
- Basis expansion: Returns design matrix of basis functions
- Correlation structures: Supports non-Gaussian GLM correlation via variance-covariance matrices

If newdata and new_predictor are left NULL, default input used for model fitting will be used. Priority will be awarded to new_predictor over newdata when both are not NULL.

To obtain fitted values, users may also call model_fit$predict() or model_fit$ytilde for an lgspline object "model_fit".

The parallel processing feature is experimental and should be used with caution. When enabled, computations are split across chunks and processed in parallel, which may improve performance for large datasets.

Value

Depending on the options selected, returns the following:

predictions Numeric vector of predicted values (default case, or if derivatives requested).

first_deriv Numeric vector of first derivatives (if take_first_derivatives = TRUE).

second_deriv Numeric vector of second derivatives (if take_second_derivatives = TRUE).

expansions List of basis expansions (if expansions_only = TRUE).

With derivatives included, output is in the form of a list with elements "preds", "first_deriv", and "second_deriv" for the vector of predictions, first derivatives, and second derivatives respectively.

See Also

`lgspline` for model fitting, `plot.lgspline` for visualizing predictions

Examples

```
## Generate example data
set.seed(1234)
t <- runif(1000, -10, 10)
y <- 2*sin(t) + -0.06*t^2 + rnorm(length(t))

## Fit model
model_fit <- lgspline(t, y)

## Generate predictions for new data
newdata <- matrix(sort(rnorm(10000)), ncol = 1) # Ensure matrix format
preds <- predict(model_fit, newdata)

## Compute derivative
deriv1_res <- predict(model_fit, newdata,
```

```
                      take_first_derivatives = TRUE)
deriv2_res <- predict(model_fit, newdata,
                      take_second_derivatives = TRUE)

## Visualize results
oldpar <- par(no.readonly = TRUE) # Save current par settings
layout(matrix(c(1,1,2,2,3,3), byrow = TRUE, ncol = 2))

## Plot function
plot(newdata[,1], preds,
       main = 'Fitted Function',
       xlab = 't',
       ylab = "f(t)", type = 'l')

## Plot first derivative
plot(newdata[,1],
       deriv1_res$first_deriv,
       main = 'First Derivative',
       xlab = 't',
       ylab = "f'(t)", type = 'l')

## Plot second derivative
plot(newdata[,1],
       deriv2_res$second_deriv,
       main = 'Second Derivative',
       xlab = 't',
       ylab = "f''(t)", type = 'l')

par(oldpar) # Reset to original par settings
```

`print.lgspline` *Print Method for lgspline Objects*

Description

Provides a standard print method for lgspline model objects to display key model characteristics.

Usage

```
## S3 method for class 'lgspline'
print(x, ...)
```

Arguments

`x`	An lgspline model object
`...`	Additional arguments (not used)

Value

Invisibly returns the original `lgspline` object `x`. This function is called for printing a concise summary of the fitted model's key characteristics (family, link, N, predictors, partitions, basis functions) to the console.

`print.summary.lgspline` *Print Method for lgspline Object Summaries*

Description

Print Method for lgspline Object Summaries

Usage

```
## S3 method for class 'summary.lgspline'
print(x, ...)
```

Arguments

`x`	A summary.lgspline object, the result of calling `summary()` on an `lgspline` object.
`...`	Not used.

Value

Invisibly returns the original `summary.lgspline` object `x`. Like other print methods, this function is called to display a formatted summary of the fitted `lgspline` model to the console. This includes model dimensions, family information, dispersion estimate, effective degrees of freedom, and a coefficient table for univariate inference (if available) analogous to output from `summary.glm`.

`prior_loglik` *Log-Prior Distribution Evaluation for lgspline Models*

Description

Evaluates the log-prior distribution on beta coefficients conditional upon dispersion and penalaties,

Usage

```
prior_loglik(model_fit, sigmasq = NULL)
```

Arguments

model_fit	An lgspline model object
sigmasq	A scalar numeric representing the dispersion parameter. By default it is NULL, and the sigmasq_tilde associated with model_fit will be used. Otherwise, custom values can be supplied.

Details

Returns the quadratic form of B^T(Lambda)B evaluated at the tuned or fixed penalties, scaled by negative one-half inverse dispersion.

Assuming fixed penalties, the prior distribution of β is given as follows:

$$\beta|\sigma^2 \sim \mathcal{N}(\mathbf{0}, \frac{1}{\sigma^2}\Lambda)$$

The log-likelihood obtained from this can be shown to be equivalent to the following, with C a constant with respect to β.

$$\implies \log P(\beta|\sigma^2) = C - \frac{1}{2\sigma^2}\beta^T\Lambda\beta$$

This is useful for computing joint log-likelihoods and performing valid likelihood ratio tests between nested lgspline models.

Value

A numeric scalar for the prior-loglikelihood (the penalty on beta coefficients actually computed)

See Also

lgspline

Examples

```
## Data
t <- sort(runif(100, -5, 5))
y <- sin(t) - 0.1*t^2 + rnorm(100)

## Model keeping penalties fixed
model_fit <- lgspline(t, y, opt = FALSE)

## Full joint log-likelihood, conditional upon known sigma^2 = 1
jntloglik <- sum(dnorm(model_fit$y,
                       model_fit$ytilde,
                       1,
                       log = TRUE)) +
          prior_loglik(model_fit, sigmasq = 1)
print(jntloglik)
```

`summary.lgspline` *Summary method for lgspline Objects*

Description

Summary method for lgspline Objects

Usage

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

Arguments

`object`	An lgspline model object
`...`	Not used.

Value

An object of class `summary.lgspline`. This object is a list containing detailed information from `lgspline` fit, prepared for display. Its main components are:

model_family The `family` object or custom list specifying the distribution and link.

observations The number of observations (N) used in the fit.

predictors The number of original predictor variables (q) supplied.

knots The number of partitions (K+1) minus 1.

basis_functions The number of basis functions (coefficients) estimated per partition (p).

estimate_dispersion A character string ("Yes" or "No") indicating if the dispersion parameter was estimated.

cv The critical value (`critical_value` from the fit) used by the `print.summary.lgspline` method for confidence intervals.

coefficients A matrix summarizing univariate inference results. Columns typically include 'Estimate', 'Std. Error', test statistic ('t value' or 'z value'), 'Pr(>|t|)' or 'Pr(>|z|)', and confidence interval bounds ('CI LB', 'CI UB'). This table is fully populated only if `return_varcovmat=TRUE` was set in the original `lgspline` call. Otherwise, it defaults to a single column of estimates.

sigmasq_tilde The estimated (or fixed) dispersion parameter, $\tilde{\sigma}^2$.

trace_XUGX The calculated trace term trace$(\mathbf{XUGX}^T)$, related to effective degrees of freedom.

N Number of observations (N), re-included for convenience and printing.

`wald_univariate` *Univariate Wald Tests and Confidence Intervals for Lagrangian Multiplier Smoothing Splines*

Description

Performs coefficient-specific Wald tests and constructs confidence intervals for fitted lgspline models. (Wrapper for internal wald_univariate method). For Gaussian family with identity-link, a t-distribution replaces a normal distribution (and t-intervals, t-tests etc.) over Wald when mentioned.

Usage

```
wald_univariate(object, scale_vcovmat_by = 1, cv, ...)
```

Arguments

`object`	A fitted lgspline model object containing coefficient estimates and variance-covariance matrix (requires return_varcovmat = TRUE in fitting).
`scale_vcovmat_by`	Numeric; Scaling factor for variance-covariance matrix. Adjusts standard errors and test statistics. Default 1.
`cv`	Numeric; Critical value for confidence interval construction. If missing, defaults to value specified in lgspline() fit (‘object$critical_value‘) or ‘qnorm(0.975)‘ as a fallback. Common choices: • qnorm(0.975) for normal-based 95 • qt(0.975, df) for t-based 95
`...`	Additional arguments passed to the internal ‘wald_univariate‘ method.

Details

For each coefficient, provides:

- Point estimates
- Standard errors from the model's variance-covariance matrix
- Two-sided test statistics and p-values
- Confidence intervals using specified critical values

Value

A data frame with rows for each coefficient (across all partitions) and columns:

estimate Numeric; Coefficient estimate.

std_error Numeric; Standard error.

statistic Numeric; Wald or t-statistic (estimate/std_error).

p_value Numeric; Two-sided p-value based on normal or t-distribution.

lower_ci Numeric; Lower confidence bound (estimate - cv*std_error).

upper_ci Numeric; Upper confidence bound (estimate + cv*std_error).

See Also

lgspline

Examples

```
## Simulate some data and fit using default settings
set.seed(1234)
t <- runif(1000, -10, 10)
y <- 2*sin(t) + -0.06*t^2 + rnorm(length(t))
# Ensure varcovmat is returned for Wald tests
model_fit <- lgspline(t, y, return_varcovmat = TRUE)

## Use default critical value (likely qnorm(0.975) if not set in fit)
wald_default <- wald_univariate(model_fit)
print(wald_default)

## Specify t-distribution critical value
eff_df <- NA
if(!is.null(model_fit$N) && !is.null(model_fit$trace_XUGX)) {
   eff_df <- model_fit$N - model_fit$trace_XUGX
}
if (!is.na(eff_df) && eff_df > 0) {
  wald_t <- wald_univariate(
    model_fit,
    cv = stats::qt(0.975, eff_df)
  )
  print(wald_t)
} else {
  warning("Effective degrees of freedom invalid.")
}
```

---

weibull_dispersion_function

Estimate Weibull Dispersion for Accelerated Failure Time Model

---

Description

Computes the scale parameter for a Weibull accelerated failure time (AFT) model, supporting right-censored survival data.

This both provides a tool for actually fitting Weibull AFT Models, and boilerplate code for users who wish to incorporate Lagrangian multiplier smoothing splines into their own custom models.

Usage

```
weibull_dispersion_function(
  mu,
```

```
  y,
  order_indices,
  family,
  observation_weights,
  status
)
```

Arguments

mu	Predicted survival times
y	Observed response/survival times
order_indices	Indices to align status with response
family	Weibull AFT model family specification
observation_weights	Optional observation weights
status	Censoring indicator (1 = event, 0 = censored) Indicates whether an event of interest occurred (1) or the observation was right-censored (0). In survival analysis, right-censoring occurs when the full survival time is unknown, typically because the study ended or the subject was lost to follow-up before the event of interest occurred.

Value

Squared scale estimate for the Weibull AFT model (dispersion)

See Also

weibull_scale for the underlying scale estimation function

Examples

```
## Simulate survival data with covariates
set.seed(1234)
n <- 1000
t1 <- rnorm(n)
t2 <- rbinom(n, 1, 0.5)

## Generate survival times with Weibull-like structure
lambda <- exp(0.5 * t1 + 0.3 * t2)
yraw <- rexp(n, rate = 1/lambda)

## Introduce right-censoring
status <- rbinom(n, 1, 0.75)
y <- ifelse(status, yraw, runif(1, 0, yraw))

## Example of using dispersion function
mu <- mean(y)
order_indices <- seq_along(y)
weights <- rep(1, n)
```

```
## Estimate dispersion
dispersion_est <- weibull_dispersion_function(
  mu = mu,
  y = y,
  order_indices = order_indices,
  family = weibull_family(),
  observation_weights = weights,
  status = status
)

print(dispersion_est)
```

`weibull_family` *Weibull Family for Survival Model Specification*

Description

Creates a compatible family object for Weibull accelerated failure time (AFT) models with customizable tuning options.

This both provides a tool for actually fitting Weibull AFT Models, and boilerplate code for users who wish to incorporate Lagrangian multiplier smoothing splines into their own custom models.

Usage

```
weibull_family()
```

Details

Provides a comprehensive family specification for Weibull AFT models, including Family name, link function, inverse link function, and custom loss function for model tuning

Supports right-censored survival data with flexible parameter estimation.

Value

A list containing family-specific components for survival model estimation

Examples

```
## Simulate survival data with covariates
set.seed(1234)
n <- 1000
t1 <- rnorm(n)
t2 <- rbinom(n, 1, 0.5)

## Generate survival times with Weibull-like structure
lambda <- exp(0.5 * t1 + 0.3 * t2)
yraw <- rexp(n, rate = 1/lambda)
```

```
## Introduce right-censoring
status <- rbinom(n, 1, 0.75)
y <- ifelse(status, yraw, runif(1, 0, yraw))

## Prepare data
df <- data.frame(y = y, t1 = t1, t2 = t2, status = status)

## Fit model using custom Weibull family
model_fit <- lgspline(y ~ spl(t1) + t2,
                      df,
                      unconstrained_fit_fxn = unconstrained_fit_weibull,
                      family = weibull_family(),
                      need_dispersion_for_estimation = TRUE,
                      dispersion_function = weibull_dispersion_function,
                      glm_weight_function = weibull_glm_weight_function,
                      shur_correction_function = weibull_shur_correction,
                      status = status,
                      opt = FALSE,
                      K = 1)

summary(model_fit)
```

`weibull_glm_weight_function` *Weibull GLM Weight Function for Constructing Information Matrix*

Description

Computes diagonal weight matrix $\mathbf{W}$ for the information matrix $\mathbf{G} = (\mathbf{X}^T\mathbf{W}\mathbf{X} + \mathbf{L})^{-1}$ in Weibull accelerated failure time (AFT) models.

Usage

```
weibull_glm_weight_function(
  mu,
  y,
  order_indices,
  family,
  dispersion,
  observation_weights,
  status
)
```

Arguments

`mu`	Predicted survival times
`y`	Observed response/survival times

order_indices	Order of observations when partitioned to match "status" to "response"
family	Weibull AFT family
dispersion	Estimated dispersion parameter (s^2)
observation_weights	Weights of observations submitted to function
status	Censoring indicator (1 = event, 0 = censored) Indicates whether an event of interest occurred (1) or the observation was right-censored (0). In survival analysis, right-censoring occurs when the full survival time is unknown, typically because the study ended or the subject was lost to follow-up before the event of interest occurred.

Details

This function generates weights used in constructing the information matrix after unconstrained estimates have been found. Specifically, it is used in the construction of the **U** and **G** matrices following initial unconstrained parameter estimation.

These weights are analogous to the variance terms in generalized linear models (GLMs). Like logistic regression uses $\mu(1-\mu)$, Poisson regression uses e^{μ}, and Linear regression uses constant weights, Weibull AFT models use $\exp((\log y - \log \mu)/s)$ where s is the scale (= $\sqrt{\text{dispersion}}$) parameter.

Value

Vector of weights for constructing the diagonal weight matrix **W** in the information matrix $\mathbf{G} = (\mathbf{X}^T\mathbf{W}\mathbf{X} + \mathbf{L})^{-1}$.

Examples

```
## Demonstration of glm weight function in constrained model estimation
set.seed(1234)
n <- 1000
t1 <- rnorm(n)
t2 <- rbinom(n, 1, 0.5)

## Generate survival times
lambda <- exp(0.5 * t1 + 0.3 * t2)
yraw <- rexp(n, rate = 1/lambda)

## Introduce right-censoring
status <- rbinom(n, 1, 0.75)
y <- ifelse(status, yraw, runif(1, 0, yraw))

## Fit model demonstrating use of custom glm weight function
model_fit <- lgspline(y ~ spl(t1) + t2,
                      data.frame(y = y, t1 = t1, t2 = t2),
                      unconstrained_fit_fxn = unconstrained_fit_weibull,
                      family = weibull_family(),
                      need_dispersion_for_estimation = TRUE,
                      dispersion_function = weibull_dispersion_function,
                      glm_weight_function = weibull_glm_weight_function,
```

```
                      shur_correction_function = weibull_shur_correction,
                      status = status,
                      opt = FALSE,
                      K = 1)

print(summary(model_fit))
```

`weibull_qp_score_function` *Compute gradient of log-likelihood of Weibull accelerated failure model without penalization*

Description

Calculates the gradient of log-likelihood for a Weibull accelerated failure time (AFT) survival model, supporting right-censored survival data.

Usage

```
weibull_qp_score_function(
  X,
  y,
  mu,
  order_list,
  dispersion,
  VhalfInv,
  observation_weights,
  status
)
```

Arguments

`X`	Design matrix
`y`	Response vector
`mu`	Predicted mean vector
`order_list`	List of observation indices per partition
`dispersion`	Dispersion parameter (scale^2)
`VhalfInv`	Inverse square root of correlation matrix (if applicable)
`observation_weights`	Observation weights
`status`	Censoring indicator (1 = event, 0 = censored)

Details

Needed if using "blockfit", correlation structures, or quadratic programming with Weibull AFT models.

Value

A numeric vector representing the gradient with respect to coefficients.

Examples

```
set.seed(1234)
t1 <- rnorm(1000)
t2 <- rbinom(1000, 1, 0.5)
yraw <- rexp(exp(0.01*t1 + 0.01*t2))
status <- rbinom(1000, 1, 0.25)
yobs <- ifelse(status, runif(1, 0, yraw), yraw)
df <- data.frame(
  y = yobs,
  t1 = t1,
  t2 = t2
)

## Example using blockfit for t2 as a linear term - output does not look
# different, but internal methods used for fitting change
model_fit <- lgspline(y ~ spl(t1) + t2,
                      df,
                      unconstrained_fit_fxn = unconstrained_fit_weibull,
                      family = weibull_family(),
                      need_dispersion_for_estimation = TRUE,
                      qp_score_function = weibull_qp_score_function,
                      dispersion_function = weibull_dispersion_function,
                      glm_weight_function = weibull_glm_weight_function,
                      shur_correction_function = weibull_shur_correction,
                      K = 1,
                      blockfit = TRUE,
                      opt = FALSE,
                      status = status,
                      verbose = TRUE)

print(summary(model_fit))
```

---

weibull_scale *Estimate Scale for Weibull Accelerated Failure Time Model*

Description

Computes maximum log-likelihood scale estimate of Weibull accelerated failure time (AFT) survival model.

This both provides a tool for actually fitting Weibull AFT Models, and boilerplate code for users who wish to incorporate Lagrangian multiplier smoothing splines into their own custom models.

Usage

```
weibull_scale(log_y, log_mu, status, weights = 1)
```

Arguments

log_y	Logarithm of response/survival times
log_mu	Logarithm of predicted survival times
status	Censoring indicator (1 = event, 0 = censored) Indicates whether an event of interest occurred (1) or the observation was right-censored (0). In survival analysis, right-censoring occurs when the full survival time is unknown, typically because the study ended or the subject was lost to follow-up before the event of interest occurred.
weights	Optional observation weights (default = 1)

Details

Calculates maximum log-likelihood estimate of scale for Weibull AFT model accounting for right-censored observations using Brent's method for optimization.

Value

Scalar representing the estimated scale

Examples

```
## Simulate exponential data with censoring
set.seed(1234)
mu <- 2  # mean of exponential distribution
n <- 500
y <- rexp(n, rate = 1/mu)

## Introduce censoring (25% of observations)
status <- rbinom(n, 1, 0.75)
y_obs <- ifelse(status, y, NA)

## Compute scale estimate
scale_est <- weibull_scale(
  log_y = log(y_obs[!is.na(y_obs)]),
  log_mu = log(mu),
  status = status[!is.na(y_obs)]
)

print(scale_est)
```

`weibull_shur_correction` *Correction for the Variance-Covariance Matrix for Uncertainty in Scale*

Description

Computes the shur complement $\mathbf{S}$ such that $\mathbf{G}^* = (\mathbf{G}^{-1} + \mathbf{S})^{-1}$ properly accounts for uncertainty in estimating dispersion when estimating variance-covariance. Otherwise, the variance-covariance matrix is optimistic and assumes the scale is known, when it was in fact estimated. Note that the parameterization adds the output of this function elementwise (not subtract) so for most cases, the output of this function will be negative or a negative definite/semi-definite matrix.

Usage

```
weibull_shur_correction(
  X,
  y,
  B,
  dispersion,
  order_list,
  K,
  family,
  observation_weights,
  status
)
```

Arguments

`X`	Block-diagonal matrices of spline expansions
`y`	Block-vector of response
`B`	Block-vector of coefficient estimates
`dispersion`	Scalar, estimate of dispersion, = Weibull scale^2
`order_list`	List of partition orders
`K`	Number of partitions minus 1 (K)
`family`	Distribution family
`observation_weights`	Optional observation weights (default = 1)
`status`	Censoring indicator (1 = event, 0 = censored) Indicates whether an event of interest occurred (1) or the observation was right-censored (0). In survival analysis, right-censoring occurs when the full survival time is unknown, typically because the study ended or the subject was lost to follow-up before the event of interest occurred.

Details

Adjusts the variance-covariance matrix unscaled for coefficients to account for uncertainty in estimating the Weibull scale parameter, that otherwise would be lost if simply using $\mathbf{G} = (\mathbf{X}^T\mathbf{W}\mathbf{X} + \mathbf{L})^{-1}$. This is accomplished using a correction based on the Shur complement so we avoid having to construct the entire variance-covariance matrix, or modifying the procedure for lgspline substantially. For any model with nuisance parameters that must have uncertainty accounted for, this tool will be helpful.

This both provides a tool for actually fitting Weibull accelerated failure time (AFT) models, and boilerplate code for users who wish to incorporate Lagrangian multiplier smoothing splines into their own custom models.

Value

List of $p \times p$ matrices representing the shur-complement corrections $\mathbf{S}_k$ to be elementwise added to each block of the information matrix, before inversion.

Examples

```
## Minimal example of fitting a Weibull Accelerated Failure Time model
# Simulating survival data with right-censoring
set.seed(1234)
t1 <- rnorm(1000)
t2 <- rbinom(1000, 1, 0.5)
yraw <- rexp(exp(0.01*t1 + 0.01*t2))
# status: 1 = event occurred, 0 = right-censored
status <- rbinom(1000, 1, 0.25)
yobs <- ifelse(status, runif(1, 0, yraw), yraw)
df <- data.frame(
  y = yobs,
  t1 = t1,
  t2 = t2
)

## Fit model using lgspline with Weibull shur correction
model_fit <- lgspline(y ~ spl(t1) + t2,
                      df,
                      unconstrained_fit_fxn = unconstrained_fit_weibull,
                      family = weibull_family(),
                      need_dispersion_for_estimation = TRUE,
                      dispersion_function = weibull_dispersion_function,
                      glm_weight_function = weibull_glm_weight_function,
                      shur_correction_function = weibull_shur_correction,
                      status = status,
                      opt = FALSE,
                      K = 1)

print(summary(model_fit))

## Fit model using lgspline without Weibull shur correction
naive_fit <- lgspline(y ~ spl(t1) + t2,
                      df,
```

```
                        unconstrained_fit_fxn = unconstrained_fit_weibull,
                        family = weibull_family(),
                        need_dispersion_for_estimation = TRUE,
                        dispersion_function = weibull_dispersion_function,
                        glm_weight_function = weibull_glm_weight_function,
                        status = status,
                        opt = FALSE,
                        K = 1)

print(summary(naive_fit))
```

`%**%` *Efficient Matrix Multiplication Operator*

Description

Operator wrapper around C++ `efficient_matrix_mult()` for matrix multiplication syntax.

This is an internal function meant to provide improvement over base R's operator for certain large matrix operations, at a cost of potential slight slowdown for smaller problems.

Usage

```
x %**% y
```

Arguments

x	Left matrix
y	Right matrix

Value

Matrix product of x and y

Examples

```
M1 <- matrix(1:4, 2, 2)
M2 <- matrix(5:8, 2, 2)
M1 %**% M2
```

Index