Package 'bamlss'

May 7, 2026

Version 1.2-5

Date 2024-10-11

Title Bayesian Additive Models for Location, Scale, and Shape (and Beyond)

Description Infrastructure for estimating probabilistic distributional regression models in a Bayesian framework.
The distribution parameters may capture location, scale, shape, etc. and every parameter may depend
on complex additive terms (fixed, random, smooth, spatial, etc.) similar to a generalized additive model.
The conceptual and computational framework is introduced in Umlauf, Klein, Zeileis (2019) <doi:10.1080/10618600.2017.1407325> and the R package in Umlauf, Klein, Simon, Zeileis (2021)
<doi:10.18637/jss.v100.i04>.

Depends R (>= 3.5.0), coda, colorspace, distributions3 (>= 0.2.1), mgcv

Imports Formula, MBA, mvtnorm, sp, Matrix, survival, methods, parallel

Suggests bit, ff, fields, gamlss, gamlss.dist, interp, rjags, BayesX, mapdata, maps, sf, nnet, spatstat, spdep, zoo, keras, splines2, sdPrior, statmod, glogis, glmnet, scoringRules, knitr, rmarkdown, MASS, tensorflow

License GPL-2 | GPL-3

LazyLoad yes

RoxygenNote 7.1.1

URL http://www.bamlss.org/

VignetteBuilder knitr

NeedsCompilation yes

Author Nikolaus Umlauf [aut, cre] (ORCID: <https://orcid.org/0000-0003-2160-9803>), Nadja Klein [aut] (ORCID: <https://orcid.org/0000-0002-5072-5347>), Achim Zeileis [aut] (ORCID: <https://orcid.org/0000-0003-0918-3766>), Meike Koehler [ctb],

Thorsten Simon [aut] (ORCID: <https://orcid.org/0000-0002-3778-7738>),
Stanislaus Stadlmann [ctb],
Alexander Volkmann [ctb] (ORCID:
<https://orcid.org/0000-0001-5028-8098>)

Maintainer Nikolaus Umlauf <Nikolaus.Umlauf@uibk.ac.at>

Repository CRAN

Date/Publication 2024-10-11 15:10:02 UTC

Contents

`bamlss-package` *Bayesian Additive Models for Location Scale and Shape (and Beyond)*

Description

The **bamlss** package is a general tool for complex Bayesian regression modeling with structured additive predictors based on Markov chain Monte Carlo simulation. The design of this package substantially focuses on maximum flexibility and easy integration of new code and/or standalone systems. The package makes heavy use of **mgcv** infrastructures to build up all necessary model matrices and information from which it is relatively easy for the user to construct estimation algorithms or interfaces to existing software packages. The package can also be seen as an harmonized framework for regression modeling since it does not restrict to any type of problem. The main function in this package is `bamlss`, which is a wrapper function that calls optimizer and/or sampling functions for fitting Bayesian additive models for location scale and shape (and beyond). These model fitting functions can be exchanged by the user. Moreover, the package contains numerous functions for creating post-estimation results like summary statistics and effect plots etc.

Author(s)

Nikolaus Umlauf, Nadja Klein, Achim Zeileis.

References

Umlauf N, Klein N, Zeileis A (2019). BAMLSS: Bayesian Additive Models for Location, Scale and Shape (and Beyond). *Journal of Computational and Graphical Statistics*, **27**(3), 612–627. doi:10.1080/10618600.2017.1407325

Umlauf N, Klein N, Simon T, Zeileis A (2021). bamlss: A Lego Toolbox for Flexible Bayesian Regression (and Beyond). *Journal of Statistical Software*, **100**(4), 1–53. doi:10.18637/jss.v100.i04

See Also

`bamlss`, `bamlss.frame`

`BAMLSS` *Create* **distributions3** *Object*

Description

A single class and corresponding methods encompassing all `bamlss.family` distributions (from the **bamlss** package) using the workflow from the **distributions3** package.

Usage

```
BAMLSS(family, ...)
```

Arguments

family	object. BAMLSS family specifications recognized by bamlss.family, including family.bamlss objects, family-generating functions (e.g., gaussian_bamlss), or characters with family names (e.g., "gaussian" or "binomial").
...	further arguments passed as parameters to the BAMLSS family. Can be scalars or vectors.

Details

The constructor function BAMLSS sets up a distribution object, representing a distribution from the BAMLSS (Bayesian additive model of location, scale, and shape) framework by the corresponding parameters plus a family attribute, e.g., gaussian_bamlss for the normal distribution or binomial_bamlss for the binomial distribution. The parameters employed by the family vary across the families but typically capture different distributional properties (like location, scale, shape, etc.).

All parameters can also be vectors, so that it is possible to define a vector of BAMLSS distributions from the same family with potentially different parameters. All parameters need to have the same length or must be scalars (i.e., of length 1) which are then recycled to the length of the other parameters.

For the BAMLSS distribution objects there is a wide range of standard methods available to the generics provided in the **distributions3** package: pdf and log_pdf for the (log-)density (PDF), cdf for the probability from the cumulative distribution function (CDF), quantile for quantiles, random for simulating random variables, and support for the support interval (minimum and maximum). Internally, these methods rely on the usual d/p/q/r functions provided in **bamlss**, see the manual pages of the individual families. The methods is_discrete and is_continuous can be used to query whether the distributions are discrete on the entire support or continuous on the entire support, respectively.

See the examples below for an illustration of the workflow for the class and methods.

Value

A BAMLSS distribution object.

See Also

bamlss.family

Examples

```
## package and random seed
library("distributions3")
set.seed(6020)

## three Weibull distributions
```

```
X <- BAMLSS("weibull", lambda = c(1, 1, 2), alpha = c(1, 2, 2))
X

## moments (FIXME: mean and variance not provided by weibull_bamlss)
## mean(X)
## variance(X)

## support interval (minimum and maximum)
support(X)
is_discrete(X)
is_continuous(X)

## simulate random variables
random(X, 5)

## histograms of 1,000 simulated observations
x <- random(X, 1000)
hist(x[1, ], main = "Weibull(1,1)")
hist(x[2, ], main = "Weibull(1,2)")
hist(x[3, ], main = "Weibull(2,2)")

## probability density function (PDF) and log-density (or log-likelihood)
x <- c(2, 2, 1)
pdf(X, x)
pdf(X, x, log = TRUE)
log_pdf(X, x)

## cumulative distribution function (CDF)
cdf(X, x)

## quantiles
quantile(X, 0.5)

## cdf() and quantile() are inverses
cdf(X, quantile(X, 0.5))
quantile(X, cdf(X, 1))

## all methods above can either be applied elementwise or for
## all combinations of X and x, if length(X) = length(x),
## also the result can be assured to be a matrix via drop = FALSE
p <- c(0.05, 0.5, 0.95)
quantile(X, p, elementwise = FALSE)
quantile(X, p, elementwise = TRUE)
quantile(X, p, elementwise = TRUE, drop = FALSE)

## compare theoretical and empirical mean from 1,000 simulated observations
## (FIXME: mean not provided by weibull_bamlss)
## cbind(
##   "theoretical" = mean(X),
##   "empirical" = rowMeans(random(X, 1000))
## )
```

bamlss *Fit Bayesian Additive Models for Location Scale and Shape (and Beyond)*

Description

This is the main model fitting function of the package. Function bamlss() is a wrapper function that parses the data and the model formula, or extended bamlss.formula, as well as the bamlss.family into a bamlss.frame. The bamlss.frame then holds all model matrices and information that is needed for setting up estimation engines. The model matrices are based on **mgcv** infrastructures, i.e., smooth terms are constructed using smooth.construct and smoothCon. Therefore, all **mgcv** model term constructors like s, te, t2 and ti can be used. Identifiability conditions are imposed using function gam.side.

After the bamlss.frame is set up function bamlss() applies optimizer and/or sampling functions. These functions can also be provided by the user. See the details below on how to create new engines to be used with function bamlss().

Finally, the estimated parameters and/or samples are used to create model output results like summary statistics or effect plots. The computation of results may also be controlled by the user.

Usage

```
bamlss(formula, family = "gaussian", data = NULL,
  start = NULL, knots = NULL, weights = NULL,
  subset = NULL, offset = NULL, na.action = na.omit,
  contrasts = NULL, reference = NULL, transform = NULL,
  optimizer = NULL, sampler = NULL, samplestats = NULL,
  results = NULL, cores = NULL, sleep = NULL,
  combine = TRUE, model = TRUE, x = TRUE,
  light = FALSE, ...)
```

Arguments

formula	A formula or extended formula, i.e., the formula can be a list of formulas where each list entry specifies the details of one parameter of the modeled response distribution, see bamlss.formula. For incorporating smooth terms, all model term constructors implemented in **mgcv** such as s, te and ti can be used, amongst others.
family	A bamlss.family object, specifying the details of the modeled distribution such as the parameter names, the density function, link functions, etc. Can be a character without the "_bamlss" extension of the bamlss.family name.
data	A data.frame or list containing the model response variable(s) and covariates specified in the formula. By default the variables are taken from environment(formula): typically the environment from which bamlss is called.
start	A named numeric vector containing starting values to be send to the optimizer and/or sampler function. For a possible naming convention for the parameters see function parameters, but this is not restrictive and engine specific.

knots — An optional list containing user specified knots, see the documentation of function gam.

weights — Prior weights on the data.

subset — An optional vector specifying a subset of observations to be used in the fitting process.

offset — Can be used to supply model offsets for use in fitting.

na.action — A function which indicates what should happen when the data contain NA's. The default is set by the na.action setting of options, and is na.omit if set to NULL.

contrasts — An optional list. See the contrasts.arg of model.matrix.default.

reference — A character specifying a reference category, e.g., when fitting a multinomial model.

transform — A transformer function that is applied on the bamlss.frame. See, e.g., function randomize and bamlss.engine.setup.

optimizer — An optimizer function that returns, e.g., posterior mode estimates of the parameters as a named numeric vector. The default optimizer function is opt_bfit. If set to FALSE, no optimizer function will be used.

sampler — A sampler function that returns a matrix of samples, the columns represent the parameters, the rows the iterations. The returned matrix must be coerced to an object of class "mcmc", see as.mcmc. The default sampler function is sam_GMCMC. If set to FALSE, no sampler function will be used.

samplestats — A function computing statistics from samples, per default function samplestats is used. If set to FALSE, no samplestats function will be used. Note that this option is crucial for very large datasets, as computing statistics from samples this way may be very time consuming!

results — A function computing results from the parameters and/or samples, e.g., for creating effect plots, see function link{results.bamlss.default}. If set FALSE no results function will be used.

cores — An integer specifying the number of cores that should be used for the sampler function. This is based on function mclapply of the **parallel** package.

sleep — Time the system should sleep before the next core is started.

combine — If samples are computed on multiple cores, should the samples be combined into one mcmc matrix?

model — If set to FALSE the model frame used for modeling is not part of the return value.

x — If set to FALSE the model matrices are not part of the return value.

light — Should the returned object be lighter, i.e., if light = TRUE the returned object will not contain the model.frame and design and penalty matrices are deleted.

... — Arguments passed to the transformer, optimizer, sampler, results and samplestats function.

Details

The main idea of this function is to provide infrastructures that make it relatively easy to create estimation engines for new problems, or write interfaces to existing software packages.

The steps that are performed within the function are:

- First, the function parses the `data`, the `formula` or the extended `bamlss.formula` as well as the `bamlss.family` into a model frame like object, the `bamlss.frame`. This object holds all necessary model matrices and information that is needed for subsequent model fitting engines. Per default, all package **mgcv** smooth term constructor functions like `s`, `te`, `t2` and `ti` can be used (see also function `smooth.construct`), however, even special user defined constructors can be included, see the manual of `bamlss.frame`.
- In a second step, the `bamlss.frame` can be transformed, e.g., if a mixed model representation of smooth terms is needed, see function `randomize`.
- Then an optimizer function is started, e.g., a function that finds posterior mode estimates of the parameters. A convention for model fitting engines is that such functions should have the following arguments:

  `optimizer(x, y, family, start, weights, offset, ...)`

  Internally, function `bamlss()` will send the `x` object that holds all model matrices, the response `y` object, the `family` object, `start`ing values for the parameters, possible `weights` and `offset`s of the created `bamlss.frame` to the optimizer function (see the manual of `bamlss.frame` for more details on the `x`, `y` and other objects). The job of the optimizer is to return a named numeric vector of optimum parameters. The names of the parameters should be such that they can be uniquely mapped to the corresponding model matrices in `x`. See function `parameters` for more details on parameter names. The default optimizer function is `opt_bfit`. The optimizer can return more information than only the optimum parameters. It is possible to return a list, the convention here is that an element named `"parameters"` then holds the named vector of estimated parameters. Possible other return values could be fitted values, the Hessian matrix, information criteria or information about convergence of the algorithm, etc. Note that the parameters are simply added to the `bamlss.frame` in an (list) entry named `parameters`.
- After the optimization step, a `sampler` function is started. The arguments of such sampler functions are the same as for the `optimizer` functions

  `sampler(x, y, family, start, weights, offset, ...)`

  Sampler functions must return a matrix of samples, each row represents one iteration and the matrix can be coerced to `mcmc` objects. The function may return a list of samples, e.g., if multiple chains are returned each list entry then holds one sample matrix of one chain. The column names of the sample matrix should be the same as the names of estimated parameters. For a possible naming convention see function `parameters`, which ensures unique mapping of samples with the model matrices in the `x` object of the `bamlss.frame`. The samples are added to the `bamlss.frame` in an (list) entry named `samples`.
- Next, the `samplestats` function is applied. This function can compute any quantity from the samples and the `x` object, the arguments of such functions are

  `samplestats(samples, x, y, family, ...)`

  where argument `samples` are the samples returned from the `sampler` function, and `x`, `y` and `family` are the same objects as passed to the optimizer and or sampler functions. For example, the default function in `bamlss()` for this task is also called `samplestats` and returns the mean of the log-likelihood and the log-posterior computed of all samples, as well as the DIC.

- The last step is to compute more complex information about the model using the `results` function. The arguments of such `results` functions are

  `results(bamlss.frame, ...)`

  here, the full `bamlss.frame` including possible `parameters` and `samples` is passed to the function within `bamlss()`. The default function for this task is `results.bamlss.default` which returns an object of class `"bamlss.results"` for which generic plotting functions are and a `summary` function is provided. Hence, the user can control the output of the model, the plotting and summary statistics, too.

Note that function `transform()`, `optimizer()`, `sampler()`, `samplestats()` and `results()` can be provided from the `bamlss.family` object, e.g., if a `bamlss.family` object has an element named `"optimizer"`, which represents a valid optimizer function such as `opt_bfit`, exactly this optimizer function will be used as a default when using the family.

Value

An object of class `"bamlss"`. The object is in principle only a slight extension of a `bamlss.frame`, i.e., if an `optimizer` is applied it will hold the estimated parameters in an additional element named `"parameters"`. If a sampler function is applied it will additionally hold the samples in an element named `"samples"`. The same mechanism is used for `results` function.

If the `optimizer` function computes additional output next to the parameters, this will be saved in an element named `"model.stats"`. If a `samplestats` function is applied, the output will also be saved in the `"model.stats"` element.

Additionally, all functions that are called are saved as attribute `"functions"` in the returned object.

Author(s)

Nikolaus Umlauf, Nadja Klein, Achim Zeileis.

References

Umlauf N, Klein N, Zeileis A (2019). BAMLSS: Bayesian Additive Models for Location, Scale and Shape (and Beyond). *Journal of Computational and Graphical Statistics*, **27**(3), 612–627. doi:10.1080/10618600.2017.1407325

Umlauf N, Klein N, Simon T, Zeileis A (2021). bamlss: A Lego Toolbox for Flexible Bayesian Regression (and Beyond). *Journal of Statistical Software*, **100**(4), 1–53. doi:10.18637/jss.v100.i04

See Also

`bamlss.frame`, `family.bamlss`, `bamlss.formula`, `randomize`, `bamlss.engine.setup`, `opt_bfit`, `sam_GMCMC`, `continue`, `coef.bamlss`, `parameters`, `predict.bamlss`, `plot.bamlss`

Examples

```
## Not run: ## Simulated data example.
d <- GAMart()
f <- num ~ s(x1) + s(x2) + s(x3) + te(lon, lat)
b <- bamlss(f, data = d)
summary(b)
```

```
plot(b)
plot(b, which = 3:4)
plot(b, which = "samples")

## Use of optimizer and sampler functions:
## * first run optimizer,
b1 <- bamlss(f, data = d, optimizer = opt_bfit, sampler = FALSE)
print(b1)
summary(b1)

## * afterwards, start sampler with staring values,
b2 <- bamlss(f, data = d, start = coef(b1), optimizer = FALSE, sampler = sam_GMCMC)
print(b2)
summary(b2)

## Continue sampling.
b3 <- continue(b2, n.iter = 12000, burnin = 0, thin = 10)
plot(b3, which = "samples")
plot(b3, which = "max-acf")
plot(b3, which = "max-acf", burnin = 500, thin = 4)

## End(Not run)
```

`bamlss.engine.helpers` *BAMLSS Engine Helper Functions*

Description

These functions can be useful when setting up new model fitting engines that are based on the setup function `bamlss.engine.setup`. See the examples.

Usage

```
## Functions to extract parameter states.
get.par(x, what = NULL)
get.state(x, what = NULL)
set.par(x, replacement, what)

## Function for setting starting values.
set.starting.values(x, start)
```

Arguments

`x`	For function `get.par()` and `set.par()` argument `x` is a named numeric vector. For function `get.state()` argument `x` is an object of the `smooth.construct` list that is processed by function `bamlss.engine.setup`, i.e., which has a `"state"` object. For function `set.starting.values()` argument `x` is the `x` list, as returned from function `bamlss.frame`.
`what`	The name of the parameter(s) that should be extracted or replaced.

replacement	The value(s) that should be used for replacement.
start	The named numeric vector of starting values. The name convention is based on function parameters.

See Also

bamlss.engine.setup

Examples

```
## Create a bamlss.frame.
d <- GAMart()
bf <- bamlss.frame(num ~ s(x1) + s(x2) + te(lon,lat), data = d, family = "gaussian")
names(bf$x$mu$smooth.construct)

## Use the setup function for
## adding state elements.
bf$x <- bamlss.engine.setup(bf$x, df = c("s(x1)" = 1, "s(x2)" = 3))
names(bf$x$mu$smooth.construct)

## Extract regression coefficients.
get.state(bf$x$mu$smooth.construct[["te(lon,lat)"]], "b")

## Extract smoothing variances.
get.state(bf$x$mu$smooth.construct[["te(lon,lat)"]], "tau2")

## More examples.
state <- bf$x$mu$smooth.construct[["te(lon,lat)"]]$state
get.par(state$parameters, "b")
get.par(state$parameters, "tau2")

state$parameters <- set.par(state$parameters, c(0.1, 0.5), "tau2")
get.par(state$parameters, "tau2")

## Setting starting values.
start <- c("mu.s.s(x1).b" = 1:9, "mu.s.s(x1).tau2" = 0.1)
bf$x <- set.starting.values(bf$x, start = start)
get.state(bf$x$mu$smooth.construct[["s(x1)"]], "b")
get.state(bf$x$mu$smooth.construct[["s(x1)"]], "tau2")
```

---

bamlss.engine.setup *BAMLSS Engine Setup Function*

Description

This function takes the x object of a bamlss.frame and adds additional objects that are useful for model fitting engines. This is applied only for 'regular' terms, e.g., as created by s and te. For special model terms the corresponding smooth.construct method is in charge of this (see also the examples for function bfit).

Usage

```
bamlss.engine.setup(x, update = "iwls", propose = "iwlsC_gp",
  do.optim = NULL, df = NULL, parametric2smooth = TRUE, ...)
```

Arguments

x	The x list, as returned from function bamlss.frame, holding all model matrices and other information that is used for fitting the model.
update	Sets the updating function for model terms, see function bfit.
propose	Sets the propose function for model terms, see function GMCMC.
do.optim	Adds list element "do.optim" in the "state" element, see the details below.
df	The initial degrees of freedom that should be assigned to a smooth model term, based on the trace of the smoother matrix. Note that df can be a named numeric vector. If the names match the labels of the model terms, the corresponding df are used, e.g., df = c("s(x1)" = 1, "s(x2)" = 2) sets different dfs for each term.
parametric2smooth	Should parametric model terms be transformed into an artificial smooth model term and be added to the "smooth.construct" object within the x list? This feature is handy, since algorithms can then cycle over the "smooth.construct" object, only.
...	Currently not used.

Details

For each model term in the "smooth.construct" object of the x list (as returned from bamlss.frame), this function adds a named list called "state" with the following entries:

- "parameters": A numeric vector. Regression coefficients are named with "b", smooth variances are named with "tau2".
- "fitted.values": Given the "parameters", the actual fitted values of the model term.
- "edf": Given the smoothing variances, the actual equivalent degrees of freedom (edf) of the model term.
- "do.optim": Should an optimizer function try to find optimum smoothing variances?

The state will be changed in each iteration and can be passed outside an updating function.

Additionally, if missing in the xt argument of a model term (see, e.g., function s for xt) the function adds the corresponding log-prior and its first and second order derivatives w.r.t. regression coefficients in functions grad() and hess().

Also, objects named "lower" and "upper" are added to each model term. These indicate the lower and upper boundaries of the parameter space.

Value

A transformed x list, as returned from function bamlss.frame.

See Also

bamlss.frame, bfit, GMCMC, get.par, set.par, get.state

Examples

```
d <- GAMart()
bf <- bamlss.frame(num ~ s(x1) + s(x2), data = d, family = "gaussian")
names(bf$x$mu$smooth.construct)
bf$x <- bamlss.engine.setup(bf$x, df = c("s(x1)" = 1, "s(x2)" = 3))
names(bf$x$mu$smooth.construct)
names(bf$x$mu$smooth.construct[["s(x1)"]])
names(bf$x$mu$smooth.construct[["s(x1)"]]$state)
sapply(bf$x$mu$smooth.construct, function(x) {
  c(x$state$edf, get.state(x, "tau2"))
})
```

---

bamlss.formula *Formulae for BAMLSS*

Description

This function creates an extended BAMLSS formula. In combination with a bamlss.family object, each parameter of the response distribution is linked to a single formula. If no formula is supplied for a parameter, a simple intercept only model is created. Moreover, the function identifies hierarchical structures, see the examples. This function is useful for creating complex model.frames for (hierarchical) multi-parameter models and is used by function bamlss.frame.

Usage

```
bamlss.formula(formula, family = NULL, specials = NULL, env = NULL, ...)
```

Arguments

formula	A simple formula, or a list of simple formulae, or an extended Formula. For formula lists or extended Formulae, each single formula represents one model for the respective parameter as specified in the family object.
family	A bamlss.family object.
specials	A character vector specifying special functions to be used within formulae, see terms.object.
env	The environment that should be assigned to the formula.
...	Arguments passed to the family.

Value

A named list of class "bamlss.formula". Each list entry specifies a model, e.g., for one parameter of a provided bamlss.family object. Each entry (parameter model) then holds:

formula	A simple formula.
fake.formula	A formula with all function calls being dropped, e.g., the formula y ~ s(x1) + s(x2) is represented in the fake.formula entry as y ~ x1 + x2. The fake.formula is useful for creating model frames.

See Also

bamlss, bamlss.frame, bamlss.family

Examples

```
## Simple formula without family object.
f <- bamlss.formula(y ~ x1 + s(x2))
print(f)
print(str(f))

## Complex formula with list of formulae.
f <- list(
  y1 ~ x1 + s(x2),
  y2 ~ x3 + te(lon, lat),
  u ~ x4 + x1
)

f <- bamlss.formula(f)
print(f)
print(names(f))

## Same formula but using extended formulae
## of package Formula.
f <- y1|y2|u ~ x1 + s(x2)|x3 + te(lon,lat)|x4 + x1
f <- bamlss.formula(f)
print(f)
print(names(f))

## Using a bamlss family object, e.g., gaussian_bamlss().
## The family has two parameters, mu and sigma, for
## each parameter one formula is returned. If no
## formula is specified an intercept only model is
## generated for the respective parameter.
f <- bamlss.formula(y ~ x1 + s(x2), family = gaussian_bamlss)

## Note, same as:
f <- bamlss.formula(y ~ x1 + s(x2), family = "gaussian")
print(f)

## Specify model for parameter sigma, too
f <- list(
```

```
  y ~ x1 + s(x2),
  sigma ~ x2 + te(lon, lat)
)
f <- bamlss.formula(f, family = "gaussian")
print(f)

## With complex hierarchical structures,
## each parameter is another list of formulae,
## indicated by the h1,...,hk, names.
f <- list(
  y ~ x1 + s(x2) + id1,
  sigma ~ x2 + te(lon, lat) + id2,
  id1 ~ s(x3) + x4 + s(id3),
  id3 ~ x5 + s(x5, x6),
  id2 ~ x7
)
f <- bamlss.formula(f, family = "gaussian")
print(f)
```

---

bamlss.frame *Create a Model Frame for BAMLSS*

Description

This function parses the data and the model formula, or extended bamlss.formula, as well as the bamlss.family into a bamlss.frame object. The bamlss.frame then holds all model matrices and information that is needed for setting up estimation engines.

Usage

```
bamlss.frame(formula, data = NULL, family = "gaussian",
  weights = NULL, subset = NULL, offset = NULL,
  na.action = na.omit, contrasts = NULL,
  knots = NULL, specials = NULL, reference = NULL,
  model.matrix = TRUE, smooth.construct = TRUE,
  ytype = c("matrix", "vector", "integer"),
  scale.x = FALSE, scale.d = FALSE, ...)
```

Arguments

formula	A formula or extended formula, i.e., the formula can be a list of formulas where each list entry specifies the details of one parameter of the modeled response distribution, see bamlss.formula. For incorporating smooth terms, all model term constructors implemented in **mgcv** such as s, te and ti can be used, amongst others.
data	A data.frame or list containing the model response variable(s) and covariates specified in the formula. By default the variables are taken from environment(formula): typically the environment from which bamlss is called.

family	A bamlss.family object, specifying the details of the modeled distribution such as the parameter names, the density function, link functions, etc.
weights	Prior weights on the data.
subset	An optional vector specifying a subset of observations to be used in the fitting process.
offset	Can be used to supply model offsets for use in fitting.
na.action	A function which indicates what should happen when the data contain NA's. The default is set by the na.action setting of options, and is na.omit if set to NULL.
contrasts	An optional list. See the contrasts.arg of model.matrix.default.
knots	An optional list containing user specified knots, see the documentation of function gam.
specials	Specify new special terms here to be used with the bamlss.formula, see also terms.object.
reference	A character specifying a reference category, e.g., when fitting a multinomial model.
model.matrix	Logical, should model matrices for linear parts be returned?
smooth.construct	Logical, should model matrices, e.g., as returned from smooth.construct and smoothCon be part of returned bamlss.frame?.
ytype	For categorical responses, should the response be a vector or matrix. If ytype == "matrix" bamlss.frame() uses function model.matrix to construct the response matrix from levels. If the response is a factor ytype == "integer" will create an integer response.
scale.x	Logical, should the model matrices of the linear parts be scaled?
scale.d	Logical, should the numeric variables in the model frame be scaled?
...	Arguments passed to function smooth.construct.bamlss.frame.

Details

The function parses the data, the formula or the extended bamlss.formula as well as the bamlss.family into a model frame like object, the bamlss.frame. This object holds all necessary model matrices and information that is needed for model fitting engines. Per default, all package **mgcv** smooth term constructor functions like s, te, t2 and ti can be used (see also function smooth.construct), however, even special user defined constructors can be included, see the examples below.

Function bamlss.frame() uses function model.matrix.bamlss.frame to compute all design matrices for simple linear parts, all smooth terms are parsed with function smooth.construct.bamlss.frame.

It is also possible to create a "bamlss.frame" using hierarchical formulae, see the example below.

Value

An list of class "bamlss.frame" with the following elements:

call	The initial call.

model.frame	The model.frame used to compute all design matrices.
formula	The bamlss.formula.
family	The bamlss.family object.
terms	The terms.bamlss object.
x	A named list, the elements correspond to the parameters that are specified within the bamlss.family object. For each parameter the corresponding formula, a fake.formula only holding the covariate names, a terms object, a model.matrix for the linear part and a list smooth.construct holding all information for smooth terms as returned from function link{smooth.construct.bamlss.frame} is created.
y	The response data.

See Also

bamlss, bamlss.formula, bamlss.family, smooth.construct.bamlss.frame, model.matrix.bamlss.frame

Examples

```
## Create a 'bamlss.frame'.
d <- GAMart()
f <- list(
  num ~ fac + s(x1) + s(x2) + te(lon, lat),
  sigma ~ id + s(x2) + s(x3)
)
bf <- bamlss.frame(f, data = d, family = "gaussian")

## Show parts of the 'bamlss.frame'.
print(bf)

## Categorical responses.
f <- list(
  cat ~ fac + s(x1) + s(x2)
)

bf <- bamlss.frame(f, data = d, family = "multinomial", reference = "low")
print(bf)

## The response is a matrix per default.
head(bf$y)

## 0/1 responses.
d <- cbind(d, model.matrix(~ -1 + cat, data = d))

f <- list(
  catnone ~ fac + s(x1),
  catlow ~ s(x2),
  catmedium ~ s(x3)
)

bf <- bamlss.frame(f, data = d, family = "multinomial")
```

```
print(bf)

## Hierarchical structures.
f <- list(
  num ~ s(x1) + s(x2) + id,
  id ~ te(lon, lat),
  sigma ~ s(x1) + fac
)

bf <- bamlss.frame(f, data = d, family = "gaussian")
print(bf)

## Special model term constructors,
## set up "new" constructor function and eval
## with bamlss.frame().
s77 <- function(...) {
  sm <- s(...)
  sm$label <- paste("s77(", paste(sm$term, collapse = ","), ")", sep = "")
  sm
}

f <- list(
  num ~ s77(x1) + s(x2) + id,
  sigma ~ s77(x1)
)

bf <- bamlss.frame(f, data = d, family = "gaussian", specials = "s77")
print(bf)
names(bf$x$mu$smooth.construct)
```

---

bboost *Bootstrap Boosting*

Description

Wrapper function for applying bootstrap estimation using gradient boosting.

Usage

```
## Bootstrap boosting.
bboost(..., data, type = 1, cores = 1,
  n = 2, prob = 0.623, fmstop = NULL,
  trace = TRUE, drop = FALSE, replace = FALSE)

## Plotting function.
bboost_plot(object, col = NULL)

## Predict method.
## S3 method for class 'bboost'
predict(object, newdata, ..., cores = 1, pfun = NULL)
```

Arguments

...	Arguments passed to bamlss and predict.bamlss.
data	The data frame to be used for modeling.
type	Type of algorithm, type = 1 uses all observations and samples with replacement, type = 2 uses only a fraction specified in prob and samples with replacement.
cores	The number of cores to be used.
n	The number of bootstrap iterations.
prob	The fraction that should be used to fit the model in each bootstrap iteration.
fmstop	The function that should return the optimum stopping iteration. The function must have two arguments: (1) the model end (2) the data. The function must return a list with two named arguments: (1) "mstop" the optimum stopping iteration and (2) a vector of the objective criterion that should be evaluated by the hold out sample data during each bootstrap iteration. See the examples.
trace	Prints out the current state of the bootstrap algorithm.
drop	Should only the best set of parameters be saved?
replace	Sampling with replacement, or sampling ceiling(nobs * prob) rows of the data for fitting the n models.
object	The "bboost" object used for prediction and plotting.
col	The color that should be used for plotting.
newdata	The data frame predictions should be made for.
pfun	The prediction function that should be used, for example predictn could be used, too. Note that this is experimental.

Value

A list of bamlss objects.

See Also

bamlss, boost, lasso, BayesX

Examples

```
## Not run: ## Simulate data.
set.seed(123)
d <- GAMart()

## Estimate model.
f <- num ~ s(x1) + s(x2) + s(x3) + s(lon,lat)

## Function for evaluation of hold out sample
## criterion to find the optimum mstop.
fmstop <- function(model, data) {
  p <- predict(model, newdata = data, model = "mu")
  mse <- NULL
  for(i in 1:nrow(model$parameters))
```

```
    mse <- c(mse, mean((data$num - p[, i])^2))
  list("MSE" = mse, "mstop" = which.min(mse))
}

## Bootstrap boosted models.
b <- bboost(f, data = d, n = 50, cores = 3, fmstop = fmstop)

## Plot hold out sample MSE.
bboost_plot(b)

## Predict for each bootstrap sample.
nd <- data.frame("x2" = seq(0, 1, length = 100))
p <- predict(b, newdata = nd, model = "mu", term = "x2")
plot2d(p ~ x2, data = nd)

## End(Not run)
```

---

boost2 *Some Shortcuts*

Description

Some simple shortcuts to model fitting engines.

Usage

```
## BayesX.
bayesx2(...)

## Gradient boosting.
boost2(...)

## Lasso.
lasso2(...)
```

Arguments

...	Arguments passed to bamlss and predict.bamlss.

Value

A bamlss object.

See Also

bamlss, boost, lasso, BayesX

Examples

```
## Not run: ## Simulate data.
set.seed(123)
d <- GAMart()

## Estimate model.
f <- num ~ s(x1) + s(x2) + s(x3) + s(lon,lat)

## Boosted model.
b <- boost2(f, data = d)

## Plot estimated effects.
plot(b)

## End(Not run)
```

---

c95 *Compute 95% Credible Interval and Mean*

Description

Small helper function that computes the 2.5% and 97.5% quantiles and the mean of a vector. Useful for example when using function predict.bamlss.

Usage

```
c95(x)
```

Arguments

x	A numeric vector.

See Also

predict.bamlss, coef.bamlss

Examples

```
x <- rnorm(100)
c95(x)

## Not run: ## Example computing predictions.
set.seed(123)
d <- data.frame("x" = seq(-3, 3, length = 30))
d$y <- sin(d$x) + rnorm(30, sd = 0.3)

## Estimate model and compute predictions.
## with c95().
b <- bamlss(y ~ s(x), data = d)
```

```
p <- predict(b, model = "mu", FUN = c95)
plot(d)
matplot(d$x, p, type = "l", lty = c(2, 1, 2),
  col = "black", add = TRUE)

## Example extracting coefficients.
coef(b, FUN = c95)

## End(Not run)
```

---

coef.bamlss *Extract BAMLSS Coefficients*

Description

Methods to extract coefficients of fitted bamlss objects, either coefficients returned from optimizer functions, or samples from a sampler functions.

Method confint.bamlss() produces credible intervals or parameter samples using quantiles.

Usage

```
## S3 method for class 'bamlss'
coef(object, model = NULL, term = NULL,
  FUN = NULL, parameters = NULL,
  pterms = TRUE, sterms = TRUE,
  hyper.parameters = TRUE, list = FALSE,
  full.names = TRUE, rescale = FALSE, ...)

## S3 method for class 'bamlss'
confint(object, parm, level = 0.95,
  model = NULL, pterms = TRUE, sterms = FALSE,
  full.names = FALSE, hyper.parameters = FALSE, ...)
```

Arguments

object	An object of class "bamlss"
model	Character or integer. For which model should coefficients be extracted?
term	Character or integer. For which term should coefficients be extracted?
FUN	A function that is applied on the parameter samples.
parameters	If is set to TRUE, additionally adds estimated parameters returned from an optimizer function (if available).
pterms	Should coefficients of parametric terms be included?
sterms	Should coefficients of smooths terms be included?
hyper.parameters	For smooth terms, should hyper parameters such as smoothing variances be included?

list	Should the returned object have a list structure for each distribution parameter?
full.names	Should full names be assigned, indicating whether a term is parametric "p" or smooth "s".
rescale	Should parameters of the linear parts be rescaled if the scale.d argument in bamlss.frame is set to TRUE.
parm	Character or integer. For which term should coefficients intervals be extracted?
level	The credible level which defines the lower and upper quantiles that should be computed from the samples.
...	Arguments to be passed to FUN and function samples.

Value

Depending on argument list and the number of distributional parameters, either a list or vector/matrix of model coefficients.

See Also

bamlss.

Examples

```
## Not run: ## Simulate data.
d <- GAMart()

## Model formula.
f <- list(
  num ~ s(x1) + s(x2) + s(x3),
  sigma ~ s(x1) + s(x2) + s(x3)
)

## Estimate model.
b <- bamlss(f, data = d)

## Extract coefficients based on MCMC samples.
coef(b)

## Now only the mean.
coef(b, FUN = mean)

## As list without the full names.
coef(b, FUN = mean, list = TRUE, full.names = FALSE)

## Coefficients only for "mu".
coef(b, model = "mu")

## And "s(x2)".
coef(b, model = "mu", term = "s(x2)")

## With optimizer parameters.
coef(b, model = "mu", term = "s(x2)", parameters = TRUE)
```

```
## Only parameteric part.
coef(b, sterms = FALSE, hyper.parameters = FALSE)

## For sigma.
coef(b, model = "sigma", sterms = FALSE,
  hyper.parameters = FALSE)

## 95 perc. credible interval based on samples.
confint(b)

## End(Not run)
```

`colorlegend` *Plot a Color Legend*

Description

Function to generate a color legend, the legend may be added to an existing plot or drawn in a separate plotting window.

Usage

```
colorlegend(color = NULL, ncol = NULL, x = NULL, breaks = NULL,
  pos = "center", shift = 0.02, side.legend = 1L, side.ticks = 1L,
  range = NULL, lrange = NULL, width = 0.25, height = 0.05,
  scale = TRUE, xlim = NULL, ylim = NULL, plot = NULL, full = FALSE,
  add = FALSE, col.border = "black", lty.border = 1L, lwd.border = 1L,
  ticks = TRUE, at = NULL, col.ticks = "black", lwd.ticks = 1L,
  lty.ticks = 1L, length.ticks = 0.3, labels = NULL,
  distance.labels = 0, col.labels = "black", cex.labels = 1L,
  digits = 2L, swap = FALSE, symmetric = TRUE, xpd = NULL,
  title = NULL, side.title = 2, shift.title = c(0, 0),
  cex.title = 1, ...)
```

Arguments

`color`	Character, integer. The colors for the legend, may also be a function, e.g. `colors = heat.colors`.
`ncol`	Integer, the number of different colors that should be generated if `color` is a function.
`x`	Numeric, values for which the color legend should be drawn.
`breaks`	Numeric, a set of breakpoints for the colors: must give one more breakpoint than `ncol`.
`pos`	Character, numeric. The position of the legend. Either a numeric vector, e.g. `pos = c(0.1, 0.2)` will add the legend at the 10% point in the x-direction and at the 20% point in the y-direction of the plotting window, may also be negative, or one

	of the following: "bottomleft", "topleft", "topright", "bottomright", "left", "right", "top", "bottom" and "center".
shift	Numeric, if argument pos is a character, shift determines the distance of the legend from the plotting box.
side.legend	Integer, if set to 2 the legend will be flipped by 90 degrees.
side.ticks	Integer, if set to 2, the ticks and labels will be on the opposite site of the legend.
range	Numeric, specifies a range for x values for which the legend should be drawn.
lrange	Numeric, specifies the range of legend.
width	Numeric, the width of the legend, if scale = TRUE the width is proportional to the x-limits of the plotting window.
height	Numeric, the height of the legend, if scale = TRUE the height is proportional to the y-limits of the plotting window.
scale	Logical, if set to TRUE, the width and height of the legend will be calculated proportional to the x- and y-limits of the plotting window.
xlim	Numeric, the x-limits of the plotting window the legend should be added for, numeric vector, e.g., returned from function range.
ylim	Numeric, the y-limits of the plotting window the legend should be added for, numeric vector, e.g., returned from function range.
plot	Logical, if set to TRUE, the legend will be drawn in a separate plotting window.
full	Logical, if set to TRUE, the legend will be drawn using the full window range.
add	Logical, if set to TRUE, the legend will be added to an existing plot.
col.border	The color of the surrounding border line of the legend.
lty.border	The line type of the surrounding border line of the legend.
lwd.border	The line width of the surrounding border line of the legend.
ticks	Logical, if set to TRUE, ticks will be added to the legend.
at	Numeric, specifies at which locations ticks and labels should be added.
col.ticks	The colors of the ticks.
lwd.ticks	The line width of the ticks.
lty.ticks	The line type of the ticks.
length.ticks	Numeric, the length of the ticks as percentage of the height or width of the colorlegend.
labels	Character, specifies labels that should be added to the ticks.
distance.labels	Numeric, the distance of the labels to the ticks, proportional to the length of the ticks.
col.labels	The colors of the labels.
cex.labels	Text size of the labels.
digits	Integer, the decimal places if labels are numerical.
swap	Logical, if set to TRUE colors will be represented in reverse order.

symmetric	Logical, if set to TRUE, a symmetric legend will be drawn corresponding to the +- max(abs(x)) value.
xpd	Sets the xpd parameter in function par.
title	Character, a title for the legend.
side.title	Integer, 1 or 2. Specifies where the legend is placed, either on top if side.title = 1 or at the bottom if side.title = 2.
shift.title	Numeric vector of length 2. Specifies a possible shift of the title in either x- or y-direction.
cex.title	Text size for the title.
...	Other graphical parameters to be passed to function text.

Value

A named list with the colors generated, the breaks and the function map, which may be used for mapping of x values to the colors specified in argument colors, please see the examples below.

Examples

```
## Play with colorlegend.
colorlegend()
colorlegend(side.legend = 2)
colorlegend(side.legend = 2, side.ticks = 2)
colorlegend(height = 2)
colorlegend(width = 1, height = 0.8, scale = FALSE,
  pos = c(0, 0.2), length.ticks = 0.5)
colorlegend(color = heat.colors, ncol = 9)
colorlegend(color = heat.colors, ncol = 9, swap = TRUE)
colorlegend(pos = "bottomleft")
colorlegend(pos = "topleft")
colorlegend(pos = "topright")
colorlegend(pos = "bottomright")


## Take x values for the color legend.
x <- runif(100, -2, 2)
colorlegend(color = diverge_hcl, x = x)
colorlegend(color = diverge_hcl, x = x, at = c(-1.5, 0, 1.5))
colorlegend(color = diverge_hcl, x = x, at = c(-1.5, 0, 1.5),
  labels = c("low", "middle", "high"))
colorlegend(color = rainbow_hcl, x = x, at = c(-1.5, 0, 1.5),
  labels = c("low", "middle", "high"), length.ticks = 1.5)
colorlegend(color = heat_hcl, x = x, at = c(-1.5, 0, 1.5),
  labels = c("low", "middle", "high"), length.ticks = 1.5,
  lwd.border = 2, lwd.ticks = 2, cex.labels = 1.5, font = 2)
colorlegend(color = topo.colors, x = x, at = c(-1.5, 0, 1.5),
  labels = c("low", "middle", "high"), length.ticks = 1.5,
  lwd.border = 2, lwd.ticks = 2, cex.labels = 1.5, font = 2,
  col.border = "green3", col.ticks = c(2, 5, 2),
  col.labels = c(6, 4, 3))
colorlegend(color = diverge_hsv, x = x, at = c(-1.5, 0, 1.5),
```

```
  labels = c("low", "middle", "high"), length.ticks = 1.5,
  lwd.border = 2, lwd.ticks = 2, cex.labels = 1.5, font = 2,
  col.border = "green3", col.ticks = c(2, 5, 2),
  col.labels = c(6, 4, 3), lty.border = 2, lty.ticks = c(2, 3, 2))
colorlegend(color = diverge_hsv, x = x, at = c(-1.5, 0, 1.5),
  labels = c("low", "middle", "high"), length.ticks = 1.5,
  lwd.border = 2, lwd.ticks = 2, cex.labels = 1.5, font = 2,
  col.border = "green3", col.ticks = c(2, 5, 2),
  col.labels = c(6, 4, 3), lty.border = 2, lty.ticks = c(2, 3, 2),
  ncol = 3)
colorlegend(color = c("red", "white", "red"), x = x, at = c(-1.5, 0, 1.5),
  labels = c("low", "middle", "high"), length.ticks = 1.5,
  lwd.border = 2, lwd.ticks = 2, cex.labels = 1.5, font = 2,
  col.border = "green3", col.ticks = c(2, 5, 2),
  col.labels = c(6, 4, 3), lty.border = 2, lty.ticks = c(2, 3, 2),
  ncol = 3, breaks = c(-2, -1, 1, 2))
colorlegend(color = diverge_hcl, x = x, range = c(-3, 3))
colorlegend(color = diverge_hcl, x = x, range = c(-3, 3), lrange = c(-6, 6))


## Combine plot with color legend.
n <- 100
x <- y <- seq(-3, 3, length.out = n)
z <- outer(sin(x), cos(x))
pal <- colorlegend(color = diverge_hcl, x = z, plot = FALSE)
op <- par(no.readonly = TRUE)
par(mar = c(4.1, 4.1, 1.1, 1.1))
layout(matrix(c(1, 2), nrow = 1), widths = c(1, 0.3))
image(x = x, y = y, z = z, col = pal$colors, breaks = pal$breaks)
par(mar = c(4.1, 0.1, 1.1, 3.1))
colorlegend(color = diverge_hcl, x = z, plot = TRUE, full = TRUE,
  side.legend = 2, side.ticks = 2)
par(op)


## Another example with different plot.
n <- 50
x <- sin(seq(-3, 3, length.out = n))
pal <- colorlegend(color = diverge_hcl, x = x, plot = FALSE)
op <- par(no.readonly = TRUE)
par(mar = c(7.1, 4.1, 1.1, 1.1))
barplot(x, border = "transparent", col = pal$map(x))
colorlegend(color = diverge_hcl, x = x, plot = FALSE, add = TRUE,
  xlim = c(0, 60), ylim = c(-1, 1), pos = c(0, -0.15), xpd = TRUE,
  scale = FALSE, width = 60, height = 0.15,
  at = seq(min(x), max(x), length.out = 9))
par(op)
```

`continue` *Continue Sampling*

Description

This function takes a bamlss object which was created using a sampler function and continues sampling from the last state of the MCMC chain.

Usage

```
continue(object, cores = NULL, combine = TRUE,
  sleep = NULL, results = TRUE, ...)
```

Arguments

object	A bamlss object which contains samples.
cores	An integer specifying the number of cores that should be used for the sampler function. This is based on function mclapply of the **parallel** package.
combine	Should the new samples be combined with the old samples into one mcmc matrix? Note that if combine = FALSE and the number of iterations differ from one sampling step to the other there will be an error since the start and end points in the samples mcmc objects are different!
sleep	Time the system should sleep before the next core is started.
results	If a results function was used to create the bamlss object, should the results function be applied using the new samples?
...	Arguments passed to the sampler function.

Value

A bamlss object.

See Also

bamlss

Examples

```
## Not run: ## Simulate and run model with MCMC.
set.seed(123)
d <- GAMart()
b <- bamlss(num ~ s(x1) + s(x2) + s(x3) + te(lon,lat), data = d)

## Continue sampling.
a <- continue(b)

## Plot all samples.
## plot(a, which = "samples")

## End(Not run)
```

cox_predict *Cox Model Prediction*

Description

This function takes a fitted Cox model, i.e., a model estimated by opt_Cox or sam_Cox and computes predictions given a new data set or the original values. Survival probabilities are computed using numerical integration, therefore, computation may take some time. To avoid problems with computer memory, the prediction of survival probabilities can be split into chunks and computed parallel on different cores.

Usage

```
cox_predict(object, newdata,
  type = c("link", "parameter", "probabilities"),
  FUN = function(x) { mean(x, na.rm = TRUE) },
  time = NULL, subdivisions = 100, cores = NULL,
  chunks = 1, verbose = FALSE, ...)
```

Arguments

object	A "bamlss" object as returned from function bamlss using the optimizer opt_Cox or sampler function sam_Cox.
newdata	A data frame or list containing the values of the model covariates at which predictions are required. If missing newdata is the model.frame of the provided model.
type	Specifies the type of predictions that should be computed.
FUN	A function that should be applied on each row of the samples of the additive predictor, parameter or probabilities. Per default the function computes means of samples, however, other functions like quantile can be supplied.
time	numeric, specifies the time for which survival probabilities should be computed if type = "probabilities". Note that this overwrites survival times that are supplied in argument newdata.
subdivisions	How many time points should be created for each individual.
cores	Specifies the number of cores that should be used for prediction. The problem is split into core chunks, each chunk is then processed by one core.
chunks	The number of chunks that should be processed sequentially on one core. This way memory problems can be avoided when computing survival times for large problems.
verbose	Print progress information.
...	Arguments passed to predict.bamlss.

Value

Depending on the type of function provided in argument FUN, a numeric vector or matrix.

See Also

sam_Cox, cox_bamlss, surv_transform, simSurv, bamlss, predict.bamlss

Examples

```
## Not run: library("survival")
set.seed(123)

## Simulate survival data.
d <- simSurv(n = 500)

## Formula of the survival model, note
## that the baseline is given in the first formula by s(time).
f <- list(
  Surv(time, event) ~ s(time) + s(time, by = x3),
  gamma ~ s(x1) + s(x2)
)

## Cox model with continuous time.
## Note the the family object cox_bamlss() sets
## the default optimizer and sampler function!
## First, posterior mode estimates are computed
## using function opt_Cox(), afterwards the
## sampler sam_Cox() is started.
b <- bamlss(f, family = "cox", data = d)

## Predict survival probabilities P(T > t).
p <- predict(b, type = "probabilities",
  time = 3, subdivisions = 100, FUN = c95)

## End(Not run)
```

Crazy *Crazy simulated data*

Description

This function creates simulated data based on a crazy function.

Usage

```
Crazy(n = 1000)
```

Arguments

n	The number of observations to be simulated.

Value

A data frame with response y and covariate x.

See Also

GAMart

Examples

```
d <- Crazy(1000)
head(d)
plot(d)
```

CRPS *Continuous Rank Probability Score*

Description

The function computes the continuous rank probability score (CRPS). Note that the function uses numerical integration, for highly efficient computation please see the **scoringRules** package.

Usage

```
CRPS(object, newdata = NULL,
  interval = c(-Inf, Inf), FUN = mean,
  term = NULL, ...)
```

Arguments

object	An object returned from bamlss.
newdata	Optional new data that should be used for calculation.
interval	The interval that should be used for numerical integration
FUN	Function to be applied on the CRPS scores.
term	If required, specify the model terms that should be used within the predict.bamlss function.
...	Arguments passed to function FUN.

References

Gneiting T, Raftery AE (2007). Strictly Proper Scoring Rules, Prediction, and Estimation." Journal of the American Statistical Association, 102(477), 359–378. doi:10.1198/016214506000001437cd ..

Gneiting T, Balabdaoui F, Raftery AE (2007). Probabilistic Forecasts, Calibration and Sharpness. Journal of the Royal Statistical Society B, 69(2), 243–268. doi:10.1111/j.14679868.2007.00587.x

Examples

```
## Not run: ## Simulate data.
d <- GAMart()

## Model only including covariate x1.
b1 <- bamlss(num ~ s(x1), data = d)

## Now, also including x2 and x2.
b2 <- bamlss(num ~ s(x1) + s(x2) + s(x3), data = d)

## Compare using the CRPS score.
CRPS(b1)
CRPS(b2)

## End(Not run)
```

---

ddnn *Deep Distributional Neural Network*

Description

This function interfaces **keras** infrastructures for high-level neural networks. The function can be used as a standalone model fitting engine such as bamlss or as an on top model engine to capture special features in the data that could not be captures by other model fitting engines.

Usage

```
## Deep distributional neural net.
ddnn(object, optimizer = "adam",
  learning_rate = 0.01,
  epochs = 100, batch_size = NULL,
  nlayers = 2, units = 100, activation = "relu",
  l1 = NULL, l2 = NULL,
  validation_split = 0.2, early_stopping = TRUE, patience = 50,
  verbose = TRUE, ...)

## Predict method.
## S3 method for class 'ddnn'
predict(object, newdata,
  model = NULL, type = c("link", "parameter"),
  drop = TRUE, ...)

## CV method for optimizing
## the number of epochs using
## the CRPS.
cv_ddnn(formula, data, folds = 10,
  min_epochs = 300, max_epochs = 400,
  interval = c(-Inf, Inf), ...)
```

Arguments

object	An object of class "bamlss" or a bamlss.formula.
optimizer	Character or call to optimizer functions to be used within fit. For character, options are: "adam" "sgd", "rmsprop", "adagrad", "adadelta", "adamax", "adam". The default is optimizer_rmsprop with learning rate set to 1e-04.
learning_rate	The learning rate of the optimizer.
epochs	Number of times to iterate over the training data arrays, see fit.
batch_size	Number of samples per gradient update, see fit.
nlayers	Number of hidden layers.
units	Number of nodes per hidden layer, can be a vector.
activation	Activation functions used for the hidden layers, can be a vector.
l1	Shrinkage parameter for L1 penalty.
l2	Shrinkage parameter for L2 penalty.
validation_split	Proportion of data that should be used for validation.
early_stopping	Logical, should early stopping of the optimizer be applied?
patience	Integer, number of iterations the optimizer waits until early stopping is applied after changes get small in validation data set.
verbose	Print information during runtime of the algorithm.
newdata	A list or data.frame that should be used for prediction.
model	Character or integer specifying for which distributional parameter predictions should be computed.
type	If type = "link" the predictor of the corresponding model is returned. If type = "parameter" predictions on the distributional parameter scale are returned.
drop	If predictions for only one model are returned, the list structure is dropped.
formula	The model formula.
data	The data used for estimation.
folds	The number of folds that should be generated.
min_epochs, max_epochs	Defines the minimum and maximum epochs thet should be used.
interval	Response interval, see function CRPS.
...	Arguments passed to bamlss.frame.

Details

The default **keras** model is a sequential model with two hidden layers with "relu" activation function and 100 units in each layer. Between each layer is a dropout layer with 0.1 dropout rate.

Value

For function ddnn() an object of class "ddnn". Note that extractor functions fitted and residuals.bamlss can be applied. For function predict.ddnn() a list or vector of predicted values.

WARNINGS

The deep learning infrastructure is experimental!

See Also

bamlss.frame, bamlss

Examples

```
## Not run: ## Simulate data.
set.seed(123)
n <- 300
x <- runif(n, -3, 3)
fsigma <- -2 + cos(x)
y <- sin(x) + rnorm(n, sd = exp(fsigma))

## Setup model formula.
f <- list(
  y ~ x,
  sigma ~ x
)

## Fit neural network.
library("keras")
b <- ddnn(f, epochs = 2000)

## Plot estimated functions.
par(mfrow = c(1, 2))
plot(x, y)
plot2d(fitted(b)$mu ~ x, add = TRUE)
plot2d(fitted(b)$sigma ~ x,
  ylim = range(c(fitted(b)$sigma, fsigma)))
plot2d(fsigma ~ x, add = TRUE, col.lines = "red")

## Predict with newdata.
nd <- data.frame(x = seq(-6, 6, length = 100))
nd$p <- predict(b, newdata = nd, type = "link")

par(mfrow = c(1, 2))
plot(x, y, xlim = c(-6, 6), ylim = range(c(nd$p$mu, y)))
plot2d(p$mu ~ x, data = nd, add = TRUE)
plot2d(p$sigma ~ x, data = nd,
  ylim = range(c(nd$p$sigma, fsigma)))
plot2d(fsigma ~ x, add = TRUE, col.lines = "red")

## Plot quantile residuals.
e <- residuals(b)
plot(e)

## End(Not run)
```

DIC *Deviance Information Criterion*

Description

Generic function returning the deviance information criterion (DIC) of a fitted model object.

Usage

```
DIC(object, ...)
```

Arguments

object	A fitted model object for which there exists a DIC method.
...	Optionally more fitted model objects.

Examples

```
## Not run: d <- GAMart()
b1 <- bamlss(num ~ s(x1), data = d)
b2 <- bamlss(num ~ s(x1) + s(x2), data = d)
DIC(b1, b2)

## End(Not run)
```

dist_mvnchol *Cholesky MVN (disttree)*

Description

disttree Families for MVN with Cholesky Parameterization

Usage

```
dist_mvnchol(k, r = k - 1L, type = c("basic", "modified", "chol"), ...)
```

Arguments

k	integer. The dimension of the multivariate distribution.
r	Integer, the number of off-diagonals to model (AD-r covariance).
type	character. Choose "basic" Cholesky decomposition or "modified" Cholesky decomposition. (For back compatibility "chol" is identical to "basic".)
...	not used.

Details

NOTE: These functions are under development!! disttree families that models a multivariate Normal (Gaussian) distribution by (modified) Cholesky decomposition of the covariance matrix.

Value

a bamlss family.

---

engines *Show Available Engines for a Family Object*

Description

The function shows available optimizer and sampling engines for a given family object.

Usage

```
engines(family, ...)
```

Arguments

family	A family object or the name of the family.
...	Further family objects or names.

Examples

```
engines(gaussian_bamlss, "gamma", cox_bamlss)
```

---

family.bamlss *Distribution Families in* **bamlss**

Description

Family objects in **bamlss** specify the information that is needed for using (different) model fitting engines, e.g., the parameter names and corresponding link functions, the density function, derivatives of the log-likelihood w.r.t. the predictors, and so forth. The optimizer or sampler functions that are called by bamlss must know how much information is needed to interpret the model since the family objects are simply passed through. Family objects are also used for computing post-modeling statistics, e.g., for residual diagnostics or random number generation. See the details and examples.

Usage

```
## Family objects in bamlss:
ALD_bamlss(..., tau = 0.5, eps = 0.01)
beta_bamlss(...)
binomial_bamlss(link = "logit", ...)
cnorm_bamlss(...)
cox_bamlss(...)
dw_bamlss(...)
DGP_bamlss(...)
dirichlet_bamlss(...)
ELF_bamlss(..., tau = 0.5)
gaussian_bamlss(...)
gaussian2_bamlss(...)
Gaussian_bamlss(...)
gamma_bamlss(...)
logNN_bamlss(...)
multinomial_bamlss(...)
mvnorm_bamlss(k = 2, ...)
mvnormAR1_bamlss(k = 2, ...)
poisson_bamlss(...)
gpareto_bamlss(...)
glogis_bamlss(...)
AR1_bamlss(...)
beta1_bamlss(ar.start, ...)
nbinom_bamlss(...)
ztnbinom_bamlss(...)
lognormal_bamlss(...)
weibull_bamlss(...)
Sichel_bamlss(...)
GEV_bamlss(...)
gumbel_bamlss(...)
mix_bamlss(f1, f2, ...)
ZANBI_bamlss(...)

## Extractor functions:
## S3 method for class 'bamlss'
family(object, ...)
## S3 method for class 'bamlss.frame'
family(object, ...)
```

Arguments

object	An object of class "bamlss" or "bamlss.frame", see function bamlss and bamlss.frame.
k	The dimension of the multivariate normal. Note, if k = 1 function gaussian_bamlss() is called.
ar.start	Logical vector of length equal to the number of rows of the full data set used for modeling. Must hold entries TRUE indicating the start of a time series of a

	section. If `ar.start = NULL` lagged residuals are computed by simple shifting. See also `bam`.
`link`	Possible link functions.
`tau`	The quantile the should be fitted.
`eps`	Constant to be used for the approximation of the absolute function.
`f1, f2`	A family of class `"gamlss.family"`, see package **gamlss.dist**.
`...`	Arguments passed to functions that are called within the family object.

Details

The following lists the minimum requirements on a **bamlss** family object to be used with `bamlss` and `bamlss.frame`:

- The family object must return a `list` of class `"family.bamlss"`.
- The object must contain the family name as a character string.
- The object must contain the names of the parameters as a character string, as well as the corresponding link functions as character string.

For most optimizer and sampling functions at least the density function, including a `log` argument, should be provided. When using generic model fitting engines like `opt_bfit` or `sam_GMCMC`, as well as for computing post-modeling statistics with function `samplestats`, and others, it is assumed that the density function in a family object has the following arguments:

`d(y, par, log = FALSE, ...)`

where argument `y` is the response (possibly a matrix) and `par` is a named list holding the evaluated parameters of the distribution, e.g., using a normal distribution `par` has two elements, one for the mean `par$mu` and one for the standard deviation `par$sigma`. The dots argument is for passing special internally used objects, depending on the type of model this feature is usually not needed.

Similarly, for derivative based algorithms, e.g. using iteratively weighted least squares (IWLS, see function `opt_bfit`, the family object holds derivative functions evaluating derivatives of the log-likelihood w.r.t. the predictors (or expectations of derivatives). For each parameter, these functions also hold the following arguments:

`score(y, par, ...)`

for computing the first derivative of the log-likelihood w.r.t. a predictor and

`hess(y, par, ...)`

for computing the negative second derivatives. Within the family object these functions are organized in a named list, see the examples below.

In addition, for the cumulative distribution function (`p(y, par, ...)`), for the quantile function (`q(y, par, ...)`) or for creating random numbers (`r(n, par, ...)`) the same structure is assumed. See, e.g., the code of function `gaussian.bamlss()`.

Some model fitting engines can initialize the distributional parameters which oftentimes leads to much faster convergence. The initialize functions are again organized within a named list, one entry for each parameter, similar to the `score` and `hess` functions, e.g., see the code of family object `gaussian.bamlss`.

Using function bamlss, residuals.bamlss and predict.bamlss the family objects may also specify the transform()er, optimizer(), sampler(), samplestats(), results(), residuals() and predict() function that should be used with this family. See for example the setup of cox_bamlss.

For using specialized estimation engines like sam_JAGS it is recommended to supply any extra arguments needed by those engines with an additional list entry within the family object, e.g., using gaussian_bamlss() with sam_JAGS the family objects holds special details in an element named "bugs".

The examples below illustrate this setup. See also the code of the **bamlss** family functions.

See Also

bamlss, bamlss.frame

Examples

```
## New family object for the normal distribution,
## can be used by function opt_bfit() and sam_GMCMC().
normal_bamlss <- function(...) {
  f <- list(
    "family" = "normal",
    "names" = c("mu", "sigma"),
    "links" = c("identity", "log"),
    "d" = function(y, par, log = FALSE) {
      dnorm(y, mean = par$mu, sd = par$sigma, log = log)
    },
    "score" = list(
      "mu" = function(y, par, ...) {
        drop((y - par$mu) / (par$sigma^2))
      },
      "sigma" = function(y, par, ...) {
        drop(-1 + (y - par$mu)^2 / (par$sigma^2))
      }
    ),
    "hess" = list(
      "mu" = function(y, par, ...) {
        drop(1 / (par$sigma^2))
      },
      "sigma" = function(y, par, ...) {
        rep(2, length(y))
      }
    )
  )
  class(f) <- "family.bamlss"
  return(f)
}

## Not run: ## Test on simulated data.
d <- GAMart()
b <- bamlss(num ~ s(x1) + s(x2) + s(x3),
  data = d, family = "normal")
plot(b)
```

```
## Compute the log-likelihood using the family object.
f <- family(b)
sum(f$d(y = d$num, par = f$map2par(fitted(b)), log = TRUE))

## For using JAGS() more details are needed.
norm4JAGS_bamlss <- function(...) {
  f <- normal_bamlss()
  f$bugs <- list(
    "dist" = "dnorm",
    "eta" = BUGSeta,
    "model" = BUGSmodel,
    "reparam" = c(sigma = "1 / sqrt(sigma)")
  )
  return(f)
}

## Now with opt_bfit() and sam_JAGS().
b <- bamlss(num ~ s(x1) + s(x2) + s(x3), data = d,
  optimizer = opt_bfit, sampler = sam_JAGS, family = "norm4JAGS")
plot(b)

## End(Not run)
```

`fatalities` *Weekly Number of Fatalities in Austria*

Description

This data set includes weekly fatalities in Austria from 2000 to 46 weeks in 2020. The data is taken from the Eurostat data base.

Usage

```
data("fatalities")
```

Format

The `fatalities` data contains the following variables:

num: Integer, the number of fatalities.

year: Integer, the corresponding year fatalities are recorded.

week: Integer, the corresponding week fatalities are recorded..

References

Eurostat Database (2020). *Population and social conditions, demography and migration, mortality, weekly deaths, deaths by week and NUTS 3 region, Austria* https://ec.europa.eu/eurostat/

Examples

```
data("fatalities")
plot(num ~ week, data = fatalities)
```

`fitted.bamlss` *BAMLSS Fitted Values*

Description

Function to compute fitted values for `bamlss` models. The function calls `predict.bamlss` to compute fitted values from samples.

Usage

```
## S3 method for class 'bamlss'
fitted(object, model = NULL, term = NULL,
  type = c("link", "parameter"), samples = TRUE,
  FUN = c95, nsamps = NULL, ...)
```

Arguments

`object`	An object of class `"bamlss"`
`model`	Character or integer, specifies the model for which fitted values should be computed.
`term`	Character or integer, specifies the model terms for which fitted values are required. Note that if `samples = TRUE`, e.g., `term = c("s(x1)", "x2")` will compute the combined fitted values `s(x1) + x2`.
`type`	If `type = "link"` the predictor of the corresponding `model` is returned. If `type = "parameter"` fitted values on the distributional parameter scale are returned.
`samples`	Should fitted values be computed using samples of parameters or estimated parameters as returned from optimizer functions (e.g., function `bfit` returns `"fitted.values"`). The former results in a call to `predict.bamlss`, the latter simply extracts the `"fitted.values"` of the `bamlss` object and is not model term specific.
`FUN`	A function that should be applied on the samples of predictors or parameters, depending on argument `type`.
`nsamps`	If the fitted `bamlss` object contains samples of parameters, computing fitted values may take quite some time. Therefore, to get a first feeling it can be useful to compute fitted values only based on `nsamps` samples, i.e., `nsamps` specifies the number of samples which are extracted on equidistant intervals.
`...`	Arguments passed to function `predict.bamlss`.

Value

Depending on arguments `model`, `FUN` and the structure of the `bamlss` model, a list of fitted values or simple vectors or matrices of fitted values.

See Also

bamlss, predict.bamlss.

Examples

```
## Not run: ## Generate some data.
d <- GAMart()

## Model formula.
f <- list(
  num ~ s(x1) + s(x2) + s(x3) + te(lon,lat),
  sigma ~ s(x1) + s(x2) + s(x3) + te(lon,lat)
)

## Estimate model.
b <- bamlss(f, data = d)

## Fitted values returned from optimizer.
f1 <- fitted(b, model = "mu", samples = FALSE)

## Fitted values returned from sampler.
f2 <- fitted(b, model = "mu", samples = TRUE, FUN = mean)

plot(f1, f2)

## End(Not run)
```

GAMart *GAM Artificial Data Set*

Description

This function creates artificial GAM-type data.frames. The function is mainly used for testing purposes.

Usage

```
GAMart(n = 500, sd = 0.1, seed = FALSE,
  ti = c("none", "vcm", "main", "both"))
```

Arguments

n	The number of observations.
sd	Standard deviation of the normal errors.
seed	Sets the seed to 111.
ti	For tensor product interaction term, the type of interaction.

Examples

```
d <- GAMart()
head(d)

## Not run: b <- bamlss(num ~ s(x1) + s(x2) + s(x3) + te(lon,lat), data = d)
plot(b)

## End(Not run)
```

`gamlss_distributions` *Extract Distribution families of the* **gamlss.dist** *Package*

Description

The functions searches in the **gamlss.dist** namespace for available distributions. It returns a named list of faily functions which can be used with `bamlss`.

Usage

```
gamlss_distributions(type = c("continuous", "discrete"))
```

Arguments

`type`	Character specifying the type of distribution to be extracted.

See Also

`bamlss`

Examples

```
## Not run:
dists <- gamlss_distributions(type = "continuous")
print(dists)

## End(Not run)
```

gF *Get a BAMLSS Family*

Description

Function to get a family.bamlss object to be used for fitting. The main purpose of this function is to ease the handling of extra arguments to the family object.

Usage

```
gF(x, ...)
```

Arguments

x	The name of the family.bamlss without the ".bamlss" extension.
...	Arguments passed to the family object.

Value

A family.bamlss object.

See Also

family.bamlss.

Examples

```
f <- gF(gaussian)
print(f)
```

Golf *Prices of Used Cars Data*

Description

This dataset is taken from the Regression Book and is about prices of used VW Golf cars.

Usage

```
data("Golf")
```

Format

The Golf data contains the following variables:

price: Numeric, sale price in 1000 Euro.

age: Numeric, age of the car in month.

kilometer: Numeric, kilometer reading in 1000 kilometers.

tia: Numeric, month until the next TIA appointment (German TUEV).

abs: Factor, does the car have abs?

sunroof: Factor, does the car have a sunroof?

References

Fahrmeir, L., Kneib, T., Lang, S. and Marx, B. (2013). Regression - Models, Methods and Applications, Springer. https://www.uni-goettingen.de/de/551357.html.

Examples

```
data("Golf")
plot(price ~ age, data = Golf)
plot(price ~ kilometer, data = Golf)
```

---

homstart_data *HOMSTART Precipitation Data*

Description

This function downloads and compiles the HOMSTART-project data set. The data is downloaded from the Zentralanstalt fuer Meteorologie und Geodynamik (ZAMG, http://www.zamg.ac.at) and funded by the Austrian Climate Research Programme (ACRP) and is free for research purposes.

Usage

```
homstart_data(dir = NULL, load = TRUE, tdir = NULL)
```

Arguments

dir	The directory where the homstart.rda file should be stored.
load	Should the homstart data be loaded?
tdir	An optional temporary directory where all downloaded files are processed.

Value

A data frame containing the following variables:

`raw`	The daily precipitation observations.
`cens`	Precipitation observations censored at 0.
`bin`	Factor with levels `"yes"` or `"no"` indicating precipitation.
`cat`	Factor with levels `"none"`, `"low"`, `"medium"` and `"high"` indicating the amount of precipitation.
`trend`	A numeric time trend
`month`	Month of of the observation.
`year`	Year of the observation.
`day`	Day of the year.
`lon`	The longitude coordinate of the corresponding meteorological station.
`lat`	The latitude coordinate of the corresponding meteorological station.
`id`	Factor, meteorological station identifier.
`cos1`, `cos2`, `sin1`, `sin2`	Transformed time trend for harmonic regression.
`weekend`	Factor, with levels `"yes"` and `"no"` indication if the observation was measured on a weekend.
`elevation`	Numeric, the elevation of the meteorological station.

References

Nemec J, Gruber C, Chimani B, Auer I (2012). Trends in extreme temperature indices in Austria based on a new homogenised dataset. *International Journal of Climatology*. DOI 10.1002/joc.3532.

Nemec J, Chimani B, Gruber C, Auer I (2011). Ein neuer Datensatz homogenisierter Tagesdaten. *OEGM Bulletin*, **1**, 19–20. https://www.meteorologie.at/docs/OEGM_bulletin_2011_1.pdf

Umlauf N, Mayr G, Messner J, Zeileis A (2012). Why does it always rain on me? A spatio-temporal analysis of precipitation in Austria. *Austrian Journal of Statistics*, **41**(1), 81–92. doi:10.17713/ajs.v41i1.190

Examples

```
## Not run: homstart_data(load = TRUE)
head(homstart)

## End(Not run)
```

jm_bamlss *Fit Flexible Additive Joint Models*

Description

Family object to fit a flexible additive joint model for longitudinal and survival data under a Bayesian approach as presented in Koehler et al. (2017a, b). All parts of the joint model can be specified as structured additive predictors. See the details and examples.

Usage

```
## JM family object.
jm_bamlss(...)

## "bamlss.frame" transformer function
## to set up joint models.
jm_transform(x, y, data, terms, knots, formula, family, subdivisions = 25,
  timedependent = c("lambda", "mu", "alpha", "dalpha"), timevar = NULL,
  idvar = NULL, alpha = .Machine$double.eps, mu = NULL, sigma = NULL,
  sparse = TRUE, nonlinear = FALSE, edf_alt = FALSE, start_mu = NULL,
  k_mu = 6, ...)

## Posterior mode optimizing engine.
opt_JM(x, y, start = NULL, weights = NULL, offset = NULL,
  criterion = c("AICc", "BIC", "AIC"), maxit = c(100, 1),
  nu = c("lambda" = 0.1, "gamma" = 0.1, "mu" = 1, "sigma" = 1,
    "alpha" = 1, "dalpha" = 1),
  update.nu = FALSE, eps = 0.0001, alpha.eps = 0.001, ic.eps = 1e-08,
  nback = 40, verbose = TRUE, digits = 4, ...)

jm_mode(x, y, start = NULL, weights = NULL, offset = NULL,
  criterion = c("AICc", "BIC", "AIC"), maxit = c(100, 1),
  nu = c("lambda" = 0.1, "gamma" = 0.1, "mu" = 1, "sigma" = 1,
    "alpha" = 1, "dalpha" = 1),
  update.nu = FALSE, eps = 0.0001, alpha.eps = 0.001, ic.eps = 1e-08,
  nback = 40, verbose = TRUE, digits = 4, ...)

## Sampler function.
sam_JM(x, y, family, start = NULL, weights = NULL, offset = NULL,
  n.iter = 1200, burnin = 200, thin = 1, verbose = TRUE, digits = 4,
  step = 20, ...)

jm_mcmc(x, y, family, start = NULL, weights = NULL, offset = NULL,
  n.iter = 1200, burnin = 200, thin = 1, verbose = TRUE, digits = 4,
  step = 20, ...)

## Predict function, set to default in jm_bamlss().
```

```
jm_predict(object, newdata,
  type = c("link", "parameter", "probabilities", "cumhaz", "loglik"),
  dt, steps, id, FUN = function(x) { mean(x, na.rm = TRUE) },
  subdivisions = 100, cores = NULL, chunks = 1,
  verbose = FALSE,  ...)

## Survival plot.
jm_survplot(object, id = 1, dt = NULL, steps = 10,
  points = TRUE, rug = !points)
```

Arguments

x	The x list, as returned from function bamlss.frame (and transformed by function jm_transform()), holding all model matrices and other information that is used for fitting the model.
y	The model response, as returned from function bamlss.frame.
data	A data.frame or list containing the model response variable(s) and covariates specified in the formula in long format. By default the variables are taken from environment(formula): typically the environment from which bamlss is called.
terms	The corresponding terms.bamlss object needed for processing.
knots	An optional list containing user specified knots, see the documentation of function gam.
formula	The corresponding bamlss.formula.
family	The bamlss.family object.
subdivisions	How many time points should be created for each individual.
timedependent	A character vector specifying the names of parameters in x that are time-dependent. Time grid design matrices are only computed for these parameters.
timevar	A character specifying the name of the survival time variable in the data set.
idvar	Depending on the type of data set, this is the name of the variable specifying identifier of individuals.
alpha	Numeric, a starting value for the intercept of the association parameter alpha.
mu	Numeric, a starting value for the intercept of the mu parameter.
sigma	Numeric, a starting value for the intercept of the sigma parameter.
sparse	Logical, indicating if sparse matrix structures are used for updating and sampling of mu parameter model terms.
nonlinear	Logical, indicating if association is nonlinear in mu. See Details on the different model specifications.
edf_alt	Logical, indicating if an alternative computation of estimated degrees of freedom for penalized model terms should be used.
start_mu	Starting values for the computation of mu. For estimating associations which are nonlinear in mu, knot placement is based on these starting values which can improve stability.

k_mu	Number of knots for spline basis of association nonlinear in mu. Reducing this number improves stability of the estimation.
start	A named numeric vector containing possible starting values, the names are based on function parameters.
weights	Currently not supported.
offset	Currently not supported.
criterion	Information criterion to be used, e.g., for smoothing variance selection. Options are the corrected AIC "AICc" (see Details), the "BIC" and "AIC". Defaults to "AICc"?
maxit	Vector containing the maximum number of iterations for the backfitting algorithm with maxit[1] defining the iterations for the full model and maxit[2] the iterations within each predictor. maxit[2] defaults to 1 if only one value is specified.
nu	Vector of step lengths for parameter updates of one Newton-Raphson update for each predictor of the joint model.
update.nu	Should the updating step length be optimized in each iteration of the backfitting algorithm? Uses nu as starting value if set to TRUE.
eps	The relative convergence tolerance of the backfitting algorithm.
alpha.eps	The relative convergence tolerance of the backfitting algorithm for predictor alpha.
ic.eps	The relative convergence tolerance of the information criterion used, e.g., for smoothing variance selection.
nback	For computing ic.eps, how many iterations back should be included when computing relative convergence tolerance of the information criterion.
verbose	Print information during runtime of the algorithm.
digits	Set the digits for printing when verbose = TRUE.
n.iter	the number of MCMC iterations.
burnin	the burn-in phase of the sampler, i.e., the number of starting samples that should be removed.
thin	the thinning parameter for MCMC simulation. E.g., thin = 10 means, that only every 10th sampled parameter will be stored.
step	How many times should algorithm runtime information be printed, divides n.iter.
object	A "bamlss" object processed with the JM optimizer function opt_JM() ans/or sampler function sam_JM() for which the survival plot should be created.
newdata	Dataset for which to create predictions. Not needed for conditional survival probabilities.
type	Character string indicating which type of predictions to compute. link returns estimates for all predictors with the respective link functions applied, "parameter" returns the estimates for all pedictors, "probabilities" returns the survival probabilities conditional on the survival up to the last longitudinal measurement, and "cumhaz" return the cumulative hazard up to the survival time or for a time window after the last longitudinal measurement. If type is set to "loglik", the log-likelihood of the joint model is returned.

id	Integer or character, that specifies the individual for which the plot should be created.
dt	The time window after the last observed measurement for which predictions should be computed. The default is 0.4 * max(obstime) and obstime are the individual's longitudinal measurement times.
steps	Integer, the number of steps for which to evaluate the conditional survival probability up to dt.
FUN	A function that should be applied on the samples of predictors or parameters, depending on argument type.
cores	Specifies the number of cores that should be used for prediction. Note that this functionality is based on the parallel package.
chunks	Should computations be split into chunks? Prediction is then processed sequentially.
points	Should longitudinal observations be added to the plot.
rug	Should longitudinal observed time points be added on the x-axis to the plot.
...	Currently not used.

Details

We refer to the papers of Koehler et al. (2017a, b) for details on the flexible additive joint model. In short, we model the hazard of subject i an event at time t as

$$h_i(t) = \exp[\eta_{\lambda i}(t) + \eta_{\gamma i} + \eta_{\alpha i}(\eta_{\mu i}(t), t)]$$

with predictor η_λ for all survival covariates that are time-varying or have a time-varying coefficient (including the log baseline hazard), predictor η_γ for baseline survival covariates, predictor η_α representing the potentially time-varying or nonlinear association between the longitudinal marker η_μ and the hazard. The longitudinal response y_{ij} at time points t_{ij} is modeled as

$$y_{ij} = \eta_{\mu i}(t_{ij}) + e_{ij}$$

with independent normal errors $N(0, \exp[\eta_{\sigma i}(t_{ij})]^2)$.

Each predictor η_{ki} is a structured additive predictor, i.e. a sum of functions of covariates $\eta_{ki} = \sum_{m=1}^{M_k} f_{km}(\boldsymbol{x}_{ki})$. Each of these functions can be modeled parametrically or using basis function evaluations from the smooth constructors in **mgcv** such as s, te and ti and can include smooth time-varying, random or spatial effects. For the Bayesian estimation of these effects we specify corresponding priors: For linear or parametric terms we use vague normal priors, smooth and random effect terms are regularized by placing generic multivariate normal priors on the coefficients and for anisotropic smooths, when multiple smoothing variance parameters are involved, more complex prior are in place (cf. Koehler et al., 2017a). We use inverse Gamma hyper-priors, i.e. IG(0.001, 0.001) to obtain an inverse Gamma full conditional for the variance parameters. We estimate the posterior mode by maximizing the log-posterior of the model using a Newton-Raphson procedure, the posterior mean is obtained via derivative-based Metropolis-Hastings sampling. We recommend to use posterior mode estimates for a quick model assessment. In order to draw correct inferences from the model, posterior mean estimates should be computed. We approximate integration in the survival part of the likelihood using trapezoidal rule. For posterior mode estimation.

A variety specifications of the association $\eta_{\alpha i}(\eta_{\mu i}(t), t)$ are possible with an important distinction between associations which are nonlinear in η_μ for nonlinear = TRUE (Koehler et al. 2017b) or linear where $\eta_{\alpha i}(\eta_{\mu i}(t), t) = \eta_{\alpha i}(t)\eta_{\mu i}(t)$ for nonlinear = FALSE (Koehler et al. 2017a).

Note

The indicator `nonlinear` for associations with are linear or nonlinear in η_μ was named `interaction]` in earlier versions stages of the development.

References

Koehler M, Umlauf N, Beyerlein, A., Winkler, C. Ziegler, A.-G., Greven S (2017). Flexible Bayesian Additive Joint Models with an Application to Type 1 Diabetes Research. *Biometrical Journal*. doi:10.1002/bimj.201600224

Meike Koehler, Nikolaus Umlauf, and Sonja Greven (2018). Nonlinear association structures in flexible Bayesian additive joint models. *Statistics in Medicine*. doi:10.1002/sim.7967

See Also

`bamlss`, `bamlss.frame`.

Examples

```
## Not run:

set.seed(123)
## Simulate survival data
## with random intercepts/slopes and a linear effect of time,
## constant association alpha and no effect of the derivative
d <- simJM(nsub = 200, long_setting = "linear",
  alpha_setting = "constant",
  dalpha_setting = "zero", full = FALSE)

## Formula of the according joint model
f <- list(
  Surv2(survtime, event, obs = y) ~ s(survtime, bs = "ps"),
  gamma ~ s(x1, bs = "ps"),
  mu ~ obstime + s(id, bs = "re") +
    s(id, obstime, bs = "re"),
  sigma ~ 1,
  alpha ~ 1,
  dalpha ~ -1
)

## Joint model estimation
## jm_bamlss() sets the default optimizer and sampler function.
## First, posterior mode estimates are computed using function
## opt_JM(), afterwards the sampler sam_JM() is started.
b <- bamlss(f, data = d, family = "jm",
           timevar = "obstime", idvar = "id")

## Plot estimated effects.
plot(b)

## Predict event probabilities for two individuals
## at 12 time units after their last longitudinal measurement.
```

```
## The event probability is conditional on their survival
## up to their last observed measurement.
p <- predict(b, type = "probabilities", id = c(1, 2), dt = 12, FUN = c95)
print(p)

## Plot of survival probabilities and
## corresponding longitudinal effects
## for individual id.
jm_survplot(b, id = 3)
jm_survplot(b, id = 30)


## Simulate survival data
## with functional random intercepts and a nonlinear effect
## of time, time-varying association alpha and no effect
## of the derivative.
## Note: This specification is the simJM default.
d <- simJM(nsub = 200, full = FALSE)

## Formula of the according joint model
## number of knots for the smooth nonlinear effect of time
klong <- 8
f <- list(
  Surv2(survtime, event, obs = y) ~ s(survtime, bs = "ps"),
  gamma ~ s(x1, bs = "ps"),
  mu ~ ti(id, bs = "re") +
    ti(obstime, bs = "ps", k = klong) +
    ti(id, obstime, bs = c("re", "ps"),
      k = c(nlevels(d$id), klong)) +
    s(x2, bs = "ps"),
  sigma ~ 1,
  alpha ~ s(survtime, bs = "ps"),
  dalpha ~ -1
)

## Estimating posterior mode only using opt_JM()
b_mode <- bamlss(f, data = d, family = "jm",
                 timevar = "obstime", idvar = "id",
                 sampler = FALSE)

## Estimating posterior means using sam_JM()
##  with starting values generated from posterior mode
b_mean <- bamlss(f, data = d, family = "jm",
                 timevar = "obstime", idvar = "id", optimizer = FALSE,
                 start = parameters(b_mode), results = FALSE)

## Plot effects.
plot(b_mean, model = "alpha")



## Simulate survival data
## with functional random intercepts and an association nonlinear in mu
```

```
set.seed(234)
d <- simJM(nsub = 300, long_setting = "functional", alpha_setting = "nonlinear",
           nonlinear = TRUE, full = FALSE, probmiss = 0.9)

## Calculate longitudinal model to obtain starting values for mu
long_df <- 7
f_start <- y ~ ti(id, bs = "re") + ti(obstime, bs = "ps", k = long_df) +
               ti(id, obstime, bs = c("re", "ps"), k = c(nlevels(d$id), long_df)) +
               s(x2, bs = "ps")
b_start <- bamlss(f_start, data = d, sampler = FALSE)
mu <- predict(b_start)$mu

## Fit joint model with nonlinear association (nonlinear = TRUE)
f <- list(
  Surv2(survtime, event, obs = y) ~ s(survtime, bs = "ps"),
  gamma ~ x1,
  mu ~ ti(id, bs = "re") + ti(obstime, bs = "ps", k = long_df) +
       ti(id, obstime, bs = c("re", "ps"), k = c(nlevels(d$id), long_df)) +
       s(x2, bs = "ps"),
  sigma ~ 1,
  alpha ~ 1,
  dalpha ~ -1
)
b <- bamlss(f, data = d, family = "jm", timevar = "obstime", idvar = "id",
            nonlinear = TRUE, start_mu = mu,
            n.iter = 6000, burnin = 2000, thin = 2)
plot(b)
samplestats(b$samples)

## End(Not run)
```

---

la *Lasso Smooth Constructor*

Description

Smooth constructors and optimizer for Lasso penalization with bamlss. The penalization is based on a Taylor series approximation of the Lasso penalty.

Usage

```
## Smooth constructor function.
la(formula, type = c("single", "multiple"), ...)

## Single Lasso smoothing parameter optimizer.
opt_lasso(x, y, start = NULL, adaptive = TRUE, lower = 0.001, upper = 1000,
  nlambda = 100, lambda = NULL,  multiple = FALSE, verbose = TRUE,
  digits = 4, flush = TRUE, nu = NULL, stop.nu = NULL,
  ridge = .Machine$double.eps^0.5, zeromodel = NULL, ...)
```

```
lasso(x, y, start = NULL, adaptive = TRUE, lower = 0.001, upper = 1000,
  nlambda = 100, lambda = NULL,  multiple = FALSE, verbose = TRUE,
  digits = 4, flush = TRUE, nu = NULL, stop.nu = NULL,
  ridge = .Machine$double.eps^0.5, zeromodel = NULL, ...)

## Lasso transformation function to set
## adaptive weights from an unpenalized model.
lasso_transform(x, zeromodel, nobs = NULL, ...)

## Plotting function for opt_lasso() optimizer.
lasso_plot(x, which = c("criterion", "parameters"),
  spar = TRUE, model = NULL, name = NULL, mstop = NULL,
  retrans = FALSE, color = NULL, show.lambda = TRUE,
  labels = NULL, digits = 2, ...)

## Extract optimum stopping iteration for opt_lasso() optimizer.
## Based on the minimum of the information criterion.
lasso_stop(x)

## Extract retransformed Lasso coefficients.
lasso_coef(x, ...)
```

Arguments

formula	A formula like ~ x1 + x2 + ... + xk of variables which should be penalized with Lasso.
type	Should one single penalty parameter be used or multiple parameters, one for each covariate in formula.
x	For function opt_lasso() and lasso_transform() the x list, as returned from function bamlss.frame, holding all model matrices and other information that is used for fitting the model. For the plotting function and lasso_stop()/lasso_coef() the corresponding bamlss object fitted with the opt_lasso() optimizer.
y	The model response, as returned from function bamlss.frame.
start	A vector of starting values. Note, Lasso smoothing parameters will be dropped.
adaptive	Should adaptive weights be used for fused Lasso terms?
lower	Numeric. The minimum lambda value.
upper	Numeric. The maximum lambda value.
nlambda	Integer. The number of smoothing parameters for which coefficients should be estimated, i.e., the vector of smoothing parameters is build up as a sequence from lower to upper with length nlambda.
lambda	Numeric. A sequence/vector of lambda parameters that should be used.
multiple	Logical. Should the lambda grid be exapnded to search for multiple lambdas, one for each distributional parameter.
verbose	Print information during runtime of the algorithm.
digits	Set the digits for printing when verbose = TRUE. If the optimum lambda value is plotted, the number of decimal decimal places to be used within lasso_plot().

flush	use flush.console for displaying the current output in the console.
nu	Numeric or logical. Defines the step length for parameter updating of a model term, useful when the algorithm encounters convergence problems. If nu = TRUE the step length parameter is optimized for each model term in each iteration of the backfitting algorithm.
stop.nu	Integer. Should step length reduction be stopped after stop.nu iterations of the Lasso algorithm?
ridge	A ridge penalty parameter that should be used when finding adaptive weights, i.e., parameters from an unpenalized model. The ridge penalty is used to stabilize the estimation in complex models.
zeromodel	A model containing the unpenalized parameters, e.g., for each la() terms one can place a simple ridge penalty with la(x, ridge = TRUE, sp = 0.1). This way it is possible to find the unpenalized parameters that can be used as adaptive weights for fusion penalties.
nobs	Integer, number of observations of the data used for modeling. If not supplied nobs is taken from the number of rows from the model term design matrices.
which	Which of the two provided plots should be created, character or integer 1 and 2.
spar	Should graphical parameters be set by the plotting function?
model	Character selecting for which model the plot shpuld be created.
name	Character, the name of the coefficient group that should be plotted. Note that the string provided in name will be removed from the labels on the 4th axis.
mstop	Integer vector, defines the path length to be plotted.
retrans	Logical, should coefficients be re-transformed before plotting?
color	Colors or color function that creates colors for the group paths.
show.lambda	Logical. Should the optimum value of the penalty parameter lambda be shown?
labels	A character string of labels that should be used on the 4 axis.
...	Arguments passed to the subsequent smooth constructor function. lambda controls the starting value of the penalty parameter, const the constant that is added within the penalty approximation. Moreover, fuse = 1 enforces nominal fusion of categorical variables and fuse = 2 ordered fusion within la() Note that la() terms with and without fusion should not be mixed when using the opt_lasso() optimizer function. For the optimizer opt_lasso() arguments passed to function bfit.

Value

For function la(), similar to function s a simple smooth specification object.

For function opt_lasso() a list containing the following objects:

fitted.values	A named list of the fitted values based on the last lasso iteration of the modeled parameters of the selected distribution.
parameters	A matrix, each row corresponds to the parameter values of one boosting iteration.
lasso.stats	A matrix containing information about the log-likelihood, log-posterior and the information criterion for each lambda.

References

Andreas Groll, Julien Hambuckers, Thomas Kneib, and Nikolaus Umlauf (2019). Lasso-type penalization in the framework of generalized additive models for location, scale and shape. *Computational Statistics & Data Analysis*. doi:10.1016/j.csda.2019.06.005

Oelker Margreth-Ruth and Tutz Gerhard (2015). A uniform framework for combination of penalties in generalized structured models. *Adv Data Anal Classif*. doi:10.1007/s116340150205y

See Also

s, smooth.construct

Examples

```
## Not run: ## Simulated fusion Lasso example.
bmu <- c(0,0,0,2,2,2,4,4,4)
bsigma <- c(0,0,0,-2,-2,-2,-1,-1,-1)
id <- factor(sort(rep(1:length(bmu), length.out = 300)))

## Response.
set.seed(123)
y <- bmu[id] + rnorm(length(id), sd = exp(bsigma[id]))

## Estimate model:
## fuse=1 -> nominal fusion,
## fuse=2 -> ordinal fusion,
## first, unpenalized model to be used for adaptive fusion weights.
f <- list(y ~ la(id,fuse=2,fx=TRUE), sigma ~ la(id,fuse=1,fx=TRUE))
b0 <- bamlss(f, sampler = FALSE)

## Model with single lambda parameter.
f <- list(y ~ la(id,fuse=2), sigma ~ la(id,fuse=1))
b1 <- bamlss(f, sampler = FALSE, optimizer = opt_lasso,
  criterion = "BIC", zeromodel = b0)

## Plot information criterion and coefficient paths.
lasso_plot(b1, which = 1)
lasso_plot(b1, which = 2)
lasso_plot(b1, which = 2, model = "mu", name = "mu.s.la(id).id")
lasso_plot(b1, which = 2, model = "sigma", name = "sigma.s.la(id).id")

## Extract coefficients for optimum Lasso parameter.
coef(b1, mstop = lasso_stop(b1))

## Predict with optimum Lasso parameter.
p1 <- predict(b1, mstop = lasso_stop(b1))

## Full MCMC, needs lasso_transform() to assign the
## adaptive weights from unpenalized model b0.
b2 <- bamlss(f, optimizer = FALSE, transform = lasso_transform,
  zeromodel = b0, nobs = length(y), start = coef(b1, mstop = lasso_stop(b1)),
  n.iter = 4000, burnin = 1000)
```

```
summary(b2)
plot(b2)

ci <- confint(b2, model = "mu", pterms = FALSE, sterms = TRUE)
lasso_plot(b1, which = 2, model = "mu", name = "mu.s.la(id).id", spar = FALSE)
for(i in 1:8) {
  abline(h = ci[i, 1], lty = 2, col = "red")
  abline(h = ci[i, 2], lty = 2, col = "red")
}

## End(Not run)
```

---

lin *Linear Effects for BAMLSS*

Description

This smooth constructor implements simple linear effects. The columns of the design matrix are automatically scaled. The main advantage of this constructor is speed when used in the BAMLSS boosting algorithm boost. Optionally, a ridge penalty can be added, please see the example.

Usage

```
## Linear smooth constructor.
lin(...)

## For mgcv.
## S3 method for class 'linear.smooth.spec'
smooth.construct(object, data, knots, ...)
```

Arguments

...	For function lin() a formula of the type ~x1+x2+x3 that specifies the covariates that should be modeled.
object, data, knots	See smooth.construct.

Value

Function lin(), similar to function s a simple smooth specification object.

See Also

bamlss, predict.bamlss, bfit, boost

Examples

```
## Not run: ## Simulate data.
set.seed(123)
d <- GAMart()

## Estimate model.
f <- num ~ lin(~x1+x2+x3+fac,ridge=TRUE)

b <- bamlss(f, data = d)

summary(b)

## End(Not run)
```

LondonFire *London Fire Data*

Description

Provides the compiled dwelling fire data of London in 2015. The data is provided as a SpatialPointsDataFrame in object LondonFire. In addition the boundary and borough information is provided in objects LondonBoundaries and LondonBoroughs. Locations of all fire stations in London of 2015 are provided in object LondonFStations.

Usage

```
data("LondonFire")
```

Format

The LondonFire data contains the following variables:

arrivaltime: Numeric, the time after the emergency call until the first fire engine arrived.

daytime: Numeric, The time of day at which the emergency call was received.

fsintens: Numeric, the fire station intensity at the location of the fire scene. The intensity is measured using a kernel density estimate of the LondonFStations using the **spatstat** package.

References

London Fire (2015). London Fire Brigade Incident Records. *London Data Store*, UK Open Government Licence (OGL v2). https://data.london.gov.uk/dataset/london-fire-brigade-incident-records

London Boroughs/Boundaries (2015). Statistical GIS Boundary Files for London. *London Data Store*, UK Open Government Licence (OGL v2). https://data.london.gov.uk/dataset/statistical-gis-boundary-

London Fire Stations (2015). *London Fire Brigade*, https://www.london-fire.gov.uk/. (old url http://www.london-fire.gov.uk/A-ZFireStations.asp)

Taylor BM (2016). Spatial Modelling of Emergency Service Response Times. *Journal of the Royal Statistical Society*: Series A (Statistics in Society), **180**(2), 433–453. doi:10.1111/rssa.12192

Examples

```
data("LondonFire")
plot(LondonFire, col = "red")
plot(LondonFStations, col = "blue", cex = 1.5, pch = 16, add = TRUE)
plot(LondonBoroughs, lwd = 0.5, add = TRUE)
plot(LondonBoundaries, add = TRUE, lwd = 1.5)
head(LondonFire@data)
```

`make_formula` *Formula Generator*

Description

Generate a formula for a MVN Cholesky model.

Usage

```
make_formula(formula, type = "basic")
```

Arguments

`formula`	formula.
`type`	character. Type of Cholesky decomposition.

Details

This is a helper to generate a formula for a bamlss model with k-dimensional multi-variate normal distribution and Cholesky decomposed variance-covariance matrix. It is helpful if one formula should be used for means, another for all diagonal entries of the Cholesky factor, and a third one for all lower triangular entries of the Cholesky factor. The left hand side has k elements separated by `|`. The right hand side has one to three elements separated by `|` specifying the formulas used for all means, diagonal entries of the Cholesky factor and lower triangular entries of the Cholesky factor, respectively.

See Also

`mvnchol_bamlss`

Examples

```
f <- O | C | E | A | N ~ s(x1) + s(x2) | s(y) | z
f2 <- make_formula(f)
f2
```

`model.frame.bamlss` *BAMLSS Model Frame*

Description

Extracts the model frame of a `bamlss` or `bamlss.frame` object.

Usage

```
## S3 method for class 'bamlss'
model.frame(formula, ...)

## S3 method for class 'bamlss.frame'
model.frame(formula, ...)

## Internal helper function for creating
## the model frame.
bamlss.model.frame(formula, data, family = gaussian_bamlss(),
  weights = NULL, subset = NULL, offset = NULL,
  na.action = na.omit, specials = NULL, contrasts.arg = NULL,
  drop.unused.levels = TRUE, ...)
```

Arguments

`formula`	An object of class "`bamlss`" or "`bamlss.frame`".
`data`	A `data.frame` or `list` containing the model response variable(s) and covariates specified in the `formula`. By default the variables are taken from `environment(formula)`.
`family`	A `bamlss.family` object, see function `bamlss`.
`weights`	Prior weights on the data.
`subset`	An optional vector specifying a subset of observations to be used in the fitting process.
`offset`	Can be used to supply model offsets for use in fitting.
`na.action`	A function which indicates what should happen when the data contain `NA`'s. The default is set by the `na.action` setting of `options`, and is `na.omit` if set to `NULL`.
`specials`	Special function in formulas, see `terms.object`.
`contrasts.arg`	An optional list. See the `contrasts.arg` of `model.matrix.default`.
`drop.unused.levels`	Should factors have unused levels dropped?
`...`	Arguments to be passed to `bamlss.model.frame()` and others.

Value

The `data.frame` containing the variables used for modeling with function `bamlss`.

See Also

bamlss, bamlss.frame, model.matrix.bamlss.frame.

Examples

```
## Not run: ## Generate some data.
d <- GAMart()

## Model formula.
f <- list(
  log(pnum) ~ s(x1) + log(x2),
  sigma ~ s(x3)
)

## Estimate model.
b <- bamlss(f, data = d)

## Extract the model frame.
head(model.frame(b))

## End(Not run)
```

---

model.matrix.bamlss.frame

Construct/Extract BAMLSS Design Matrices

Description

The function creates design (or model) matrices for BAMLSS, i.e., for each parameter of a bamlss.family object.

Usage

```
## S3 method for class 'bamlss.frame'
model.matrix(object, data = NULL, model = NULL,
  drop = TRUE, scale.x = FALSE, ...)

## S3 method for class 'bamlss.formula'
model.matrix(object, data = NULL, model = NULL,
  drop = TRUE, scale.x = FALSE, ...)

## S3 method for class 'bamlss.terms'
model.matrix(object, data = NULL, model = NULL,
  drop = TRUE, scale.x = FALSE, ...)
```

Arguments

object	A bamlss.frame, bamlss.formula or terms.bamlss object.
data	A data frame or list.
model	Character or integer, specifies the model for which design matrices should be returned.
drop	If model matrices for only one model are returned, the list structure is dropped.
scale.x	Logical, should the model matrices of the linear parts be scaled?
...	Not used.

Value

Depending on the type of model a named list of model matrices or a single model matrix.

See Also

model.matrix, bamlss.frame, bamlss.formula, terms.bamlss.

Examples

```
## Generate some data.
d <- GAMart()

## Model formula.
f <- list(
  num ~ x1 + x2 + id,
  sigma ~ x3 + fac + lon + lat
)

## Create a "bamlss.frame".
bf <- bamlss.frame(f, data = d)

## Get the model matrices.
X <- model.matrix(bf)
head(X$sigma)

## Same with "bamlss.formula".
X <- model.matrix(bamlss.formula(f), data = d)
head(X$sigma)
```

---

mvnchol_bamlss *Cholesky MVN*

Description

BAMLSS Families for MVN with Cholesky Parameterization

Usage

```
mvnchol_bamlss(k, type = c("basic", "modified", "chol"), ...)
```

Arguments

k	integer. The dimension of the multivariate distribution.
type	character. Choose "basic" Cholesky decomposition or "modified" Cholesky decomposition. (For back compatibility "chol" is identical to "basic".)
...	not used.

Details

BAMLSS families that models a multivariate Normal (Gaussian) distribution by (modified) Cholesky decomposition of the covariance matrix.

For examples see TempIbk.

Value

a bamlss family.

See Also

simdata, TempIbk

---

mvn_chol *Cholesky MVN*

Description

BAMLSS Family for MVN with Cholesky Parameterization

Usage

```
mvn_chol(k = 2L, ...)
```

Arguments

k	integer. The dimension of the multivariate distribution.
...	not used.

Details

This is a prototype implementation of a BAMLSS family that models a multivariate Normal (Gaussian) distribution by a Cholesky decomposition of the covariance matrix.

Value

a bamlss family.

See Also

mvnchol_bamlss

`mvn_modchol` *Modified Cholesky MVN*

Description

BAMLSS Family for MVN with Modified Cholesky Parameterization

Usage

```
mvn_modchol(k = 2L, ...)
```

Arguments

k	integer. The dimension of the multivariate distribution.
...	not used.

Details

This is a prototype implementation of a BAMLSS family that models a multivariate Normal (Gaussian) distribution by a modified Cholesky decomposition of the covariance matrix.

Value

a bamlss family.

See Also

mvnchol_bamlss

n *Neural Networks for BAMLSS*

Description

This smooth constructor implements single hidden layer neural networks.

Usage

```
## The neural network smooth constructor.
n(..., k = 10, type = 2)

## Initialize weights.
n.weights(nodes, k, r = NULL, s = NULL,
  type = c("sigmoid", "gauss", "softplus", "cos", "sin"),
  x = NULL, ...)

## Second weights initializer, internally calls n.weights.
make_weights(object, data, dropout = 0.2)

## Boosted neural net predictions.
predictn(object, newdata, model = NULL,
  mstop = NULL, type = c("link", "parameter"))
```

Arguments

`...`	For function `n()` a formula of the type `~x1+x2+x3` that specifies the covariates that should be modeled by the neural network. For function `predictn()`, arguments to be passed to `predict.bamlss`.
`k`	For function `n()`, the number of hidden nodes of the network. Note that one can set an argument `split = TRUE` to split up the neural network into, e.g., `nsplit = 5` parts with `k` nodes each. For function `n.weights()`, argument `k` is the number of input variables of the network (number of covariates).
`type`	Integer. Type 1 fits a complete network in each boosting iteration, `type = 2` selects the best fitting node in each boosting iteration. for function `n.weights()`, the type of activation function that should be used. For function `predictn()`, the type of prediction that should be computed.
`nodes`	Number of nodes for each layer, i.e., can also be a vector.
`r, s`	Parameters controlling the shape of the activation functions.
`x`	A scaled covariate matrix, the data will be used to identify the range of the weights.
`object, data`	See `smooth.construct`. For function `predictn()`, a boosted `"bamlss"` object.
`dropout`	The fraction of inner weights that should be set to zero.
`newdata`	The data frame that should be used for prediction.

model	For which parameter of the distribution predictions should be computed.
mstop	The stopping iteration for which predictions should be computed. The default is to return a matrix of predictions, each column represents the prediction of one boosting iteration.

Value

Function n(), similar to function s a simple smooth specification object.

See Also

bamlss, predict.bamlss, bfit, boost

Examples

```
## ... coming soon ...!
```

---

neighbormatrix *Compute a Neighborhood Matrix from Spatial Polygons*

Description

The function takes a SpatialPolygonsDataFrame and computes the neighbor penalty matrix that can be used to fit a Markov random field, e.g., using the smooth constructor smooth.construct.mrf.smooth.spec.

Usage

```
## Compute the neighborhood matrix.
neighbormatrix(x, type = c("boundary", "dist", "delaunay", "knear"),
  k = 1, id = NULL, nb = FALSE, names = NULL, ...)

## Plot neighborhood structure.
plotneighbors(x, add = FALSE, ...)
```

Arguments

x	An object of class "SpatialPolygons" or "SpatialPolygonsDataFrame".
type	Which type of neighborhood structure should be used, "boundary" uses poly2nb, "dist" uses function dnearneigh, "delaunay" uses function tri2nb and option "knear" applies function knn2nb.
k	For type = "knear", specifies number of nearest neighbors.
id	An identifier variable for which the penalty matrix should be computed.
nb	Should only the neighborhood object be returned.
names	Specifies the column where the regions names are provided in the data slot in the "SpatialPolygonsDataFrame"
add	Should the neighborhood structure be added to an existing plot?
...	Arguments to be passed to function poly2nb, dnearneigh, tri2nb or knn2nb.

See Also

smooth.construct.mrf.smooth.spec, dnearneigh, tri2nb, knn2nb.

Examples

```
## Not run: data("LondonFire")

## Compute polygon boundary based
## neighborhood matrix.
nm <- neighbormatrix(LondonBoroughs)
print(nm)

## Plot neighborhood structures.
plotneighbors(LondonBoroughs)
plotneighbors(LondonBoroughs, type = "delaunay")
plotneighbors(LondonBoroughs, type = "dist", d1 = 0, d2 = 0.15)

## End(Not run)
```

---

opt_bbfit *Batchwise Backfitting*

Description

Batchwise backfitting estimation engine for GAMLSS using very large data sets.

Usage

```
## Batchwise backfitting engine.
opt_bbfit(x, y, family, shuffle = TRUE, start = NULL, offset = NULL,
  epochs = 1, nbatch = 10, verbose = TRUE, ...)

bbfit(x, y, family, shuffle = TRUE, start = NULL, offset = NULL,
  epochs = 1, nbatch = 10, verbose = TRUE, ...)

## Parallel version.
opt_bbfitp(x, y, family, mc.cores = 1, ...)

## Loglik contribution plot.
contribplot(x, ...)
```

Arguments

x	For function bfit() the x list, as returned from function bamlss.frame, holding all model matrices and other information that is used for fitting the model. For the updating functions an object as returned from function smooth.construct or smoothCon. For function contribplot(), a "bamlss" object using bbfit() with argument select = TRUE.

y	The model response, as returned from function bamlss.frame.
family	A **bamlss** family object, see family.bamlss.
shuffle	Should observations be shuffled?
start	A named numeric vector containing possible starting values, the names are based on function parameters.
offset	Can be used to supply model offsets for use in fitting, returned from function bamlss.frame.
epochs	For how many epochs should the algorithm run?
nbatch	Number of batches. Can also be a number between 0 and 1, i.e., determining the fraction of observations that should be used for fitting.
verbose	Print information during runtime of the algorithm.
mc.cores	On how many cores should estimation be started?
...	For bbfitp() all arguments to be passed to bbfit().

Details

The algorithm uses batch-wise estimation of regression coefficients and smoothing variances. The smoothing variances are estimated on an hold-out batch. This way, models for very large data sets can be estimated. Note, the algorithm can work in combination with the **ff** and **ffbase** package, i.e., the entire data is never in the computer RAM. Therefore, the data can either to be stored as comma separated file on disc or provided as "ffdf" data frame, see also the examples.

The optimizer functions use additional arguments:

- batch_ids. This argument can either be a list of indices specifying the batches that should be used for estimation, or a vector of length 2, where the first element specifies the number of observations that should be sampled for each batch and the second argument specifies the number of batches, see the example.
- nu, the step length control parameter. Defaults to nu = 0.05. If argument slice = TRUE then nu = 1.
- loglik, defaults to loglik = FALSE. If set to loglik = TRUE the "out-of-sample" log-likelihood is used for smoothing variance estimation.
- aic, defaults to aic = FALSE, If set to aic = TRUE the "out-of-sample" AIC is used for smoothing variance estimation.
- eps_loglik, defaults to eps_loglik = 0.01. This argument specifies the relative change in the "out-of-sample" log-likelihood that is needed such that a model term gets updated.
- select, defaults to select = FALSE. If set to select = TRUE, the algorithm only selects the model term with the largest contribution in the "out-of-sample" log-likelihood for updating in each iteration/batch.
- always, defaults to always = FALSE. If set to always = TRUE no log-likelihood contribution checks will be used and model terms are always updated.
- K, defaults to K = 2. This argument controls the penalty on the degrees of freedom in the computation of the AIC.

- slice, defaults to slice = FALSE. If set to slice = TRUE, slice sampling using the "out-of-sample" log-likelihood or AIC is used for smoothing variance estimation. Moreover, always = TRUE, eps_loglik = -Inf and nu = 1. If slice is an integer n, slice sampling is started after n iterations, before smoothing variances are optimized.

When using function opt_bbfitp, the parameter updates are stored as "mcmc" objects. In this case the traceplots can be visualized using plot.bamlss.

Value

For function opt_bbfit() a list containing the following objects:

fitted.values	A named list of the fitted values of the modeled parameters of the selected distribution.
parameters	The estimated set regression coefficients and smoothing variances.
shuffle	Logical
runtime	The runtime of the algorithm.

See Also

bamlss, bfit

Examples

```
## Not run: ## Simulate data.
set.seed(123)
d <- GAMart(n = 27000, sd = -1)

## Write data to disc.
tf <- tempdir()
write.table(d, file.path(tf, "d.raw"), quote = FALSE, row.names = FALSE, sep = ",")

## Model formula.
f <- list(
  y ~ s(x1,k=40) + s(x2,k=40) + s(x3,k=40) + te(lon,lat,k=10),
  sigma ~ s(x1,k=40) + s(x2,k=40) + s(x3,k=40) + te(lon,lat,k=10)
)

## Specify 50 batches with 1000 observations.
batch_ids <- c("nobs" = 1000, "nbatch" = 50)

## Note, can also be a list of indices, e.g.
## batch_ids <- lapply(1:50, function(i) { sample(1:nrow(d), size = 1000) })

## Different flavors:
## (1) Using "out-of-sample" aic for smoothing
##     variance estimation. Update is only accepted
##     if the "out-of-sample" log-likelihood is
##     increased. If data is a filepath, the data set is
##     read into R using package ff and model and
##     design matrices are processed with ff. This may
```

```
##     take some time depending on the size of the data.
set.seed(1)
b1 <- bamlss(f, data = file.path(tf, "d.raw"),
  sampler = FALSE, optimizer = opt_bbfit,
  batch_ids = batch_ids, nu = 0.1, aic = TRUE, eps_loglik = -Inf,
  always = FALSE)

## Plot estimated effects.
## plot(b1)

## Plot coefficient paths for x3 in mu.
## pathplot(b1, name = "mu.s.s(x3).b")

## (2) Same but always update, this mimics the classic SGD.
##     Note, for prediction only the last iteration is
##     used in this case. To use more iterations use opt_bbfitp(),
##     Then iterations are stored as "mcmc" object and we can
##     predict using the burnin argment, e.g.,
##     p <- predict(b2, model = "mu", burnin = 35)
set.seed(2)
b2 <- bamlss(f, data = file.path(tf, "d.raw"),
  sampler = FALSE, optimizer = opt_bbfit,
  batch_ids = batch_ids, nu = 0.1, aic = TRUE, eps_loglik = -Inf,
  always = TRUE)

## Plot coefficient paths for x3 in mu.
## pathplot(b2, name = "mu.s.s(x3).b")

## (3) Boosting type flavor, only update model term with
##     the largest contribution in the "out-of-sample"
##     log-likelihood. In this case, if edf = 0 during
##     runtime of the algorithm, no model has an additional
##     contribution and the algorithm converges. This
##     behavior is controlled by argument eps_loglik, the
##     higher eps_loglik, the more restrictive is the
##     updating step.

## Initialize intercepts.
set.seed(0)

batch_ids <- lapply(1:400, function(i) { sample(1:nrow(d), size = 1000) })

b0 <- bamlss(y ~ 1, data = d, sampler = FALSE, optimizer = opt_bbfitp,
  batch_ids = batch_ids)

## Compute starting values, remove the first
## 10 iterates and compute the mean of the
## remaining iterates.
start <- coef(b0, FUN = mean, burnin = 200)

## Start boosting, only update if change in
## "out-of-sample" log-likelihood is 0.1
## eps_loglik = 0.001.
```

```
b3 <- bamlss(f, data = d, sampler = FALSE, optimizer = opt_bbfit,
    batch_ids = batch_ids, nu = 0.1, aic = TRUE, eps_loglik = 0.001,
    select = TRUE, always = FALSE, start = start)

## Plot log-likelihood contributions.
## contribplot(b3)
## In this case, the algorithm did not converge,
## we need more iterations/batches.

## Note, prediction uses last iterate.
p3 <- predict(b3, model = "mu")

## (4) Use slice sampling under the "out-of-sample"
##     log likelihood for estimation of smoothing
##     variances. In this case model terms are always
##     updated ad parameter paths behave like a MCMC
##     chain. Therefore, use opt_bbfitp(), which stores
##     parameter paths as "mcmc" objects and we can
##     inspect using traceplots. Note nu = 1 if
##     slice = TRUE.
set.seed(4)
b4 <- bamlss(f, data = d, sampler = FALSE, optimizer = opt_bbfitp,
  batch_ids = batch_ids, aic = TRUE, slice = TRUE)

## plot(b4)

## Plot parameter updates/samples.
## plot(b4, which = "samples")

## Predict with burnin and compute mean
## prediction of the last 20 iterates.
p4 <- predict(b4, model = "mu", burnin = 30, FUN = mean)

## End(Not run)
```

`opt_bfit` *Fit BAMLSS with Backfitting*

Description

This optimizer function is a generic tool for fitting BAMLSS using a backfitting algorithm. The backfitting procedure is based on iteratively weighted least squares (IWLS) for finding posterior mode estimates, however, the updating methods for model terms can be more general, see the details section. In addition, the default IWLS updating scheme implements optimum smoothing variance selection based on information criteria using a stepwise approach.

Usage

```
## Optimizer functions:
opt_bfit(x, y, family, start = NULL, weights = NULL, offset = NULL,
```

```
  update = "iwls", criterion = c("AICc", "BIC", "AIC"),
  eps = .Machine$double.eps^0.25, maxit = 400,
  outer = NULL, inner = FALSE, mgcv = FALSE,
  verbose = TRUE, digits = 4, flush = TRUE,
  nu = TRUE, stop.nu = NULL, ...)

bfit(x, y, family, start = NULL, weights = NULL, offset = NULL,
  update = "iwls", criterion = c("AICc", "BIC", "AIC"),
  eps = .Machine$double.eps^0.25, maxit = 400,
  outer = NULL, inner = FALSE, mgcv = FALSE,
  verbose = TRUE, digits = 4, flush = TRUE,
  nu = TRUE, stop.nu = NULL, ...)

## Model term updating functions:
bfit_iwls(x, family, y, eta, id, weights, criterion, ...)
bfit_iwls_Matrix(x, family, y, eta, id, weights, criterion, ...)
bfit_lm(x, family, y, eta, id, weights, criterion, ...)
bfit_optim(x, family, y, eta, id, weights, criterion, ...)
bfit_glmnet(x, family, y, eta, id, weights, criterion, ...)
```

Arguments

x	For function opt_bfit() the x list, as returned from function bamlss.frame, holding all model matrices and other information that is used for fitting the model. For the updating functions an object as returned from function smooth.construct or smoothCon.
y	The model response, as returned from function bamlss.frame.
family	A **bamlss** family object, see family.bamlss.
start	A named numeric vector containing possible starting values, the names are based on function parameters.
weights	Prior weights on the data, as returned from function bamlss.frame.
offset	Can be used to supply model offsets for use in fitting, returned from function bamlss.frame.
update	Sets the updating function for model terms, e.g. for a term s(x) in the model formula. Per default this is set to "iwls", a character pointing to the set of updating functions, see above. Other options are "optim" and "lm" etc., however, this is more experimental and should not be set by the user. Another option is to pass a full updating function which should be used for each model term, the structure of updating functions is described in the details below. Model terms may also have different updating functions, see the example section implementing a new model term constructor for Gompertz growth curves using this feature.
criterion	Set the information criterion that should be used, e.g., for smoothing variance selection. Options are the corrected AIC "AICc", the "BIC" and "AIC".
eps	The relative convergence tolerance of the backfitting algorithm.
maxit	The maximum number of iterations for the backfitting algorithm

`outer` Should the current working observations and weights be computed in one outer iteration, otherwise the working observations are computed anew for each model term updating step. The default will run one outer iteration first, afterwards model weights are computed for each term anew.

`inner` Should the model terms for one parameter of the modeled distribution be fully updated until convergence in an inner iteration, i.e., the algorithm waits until coefficients for the current distribution parameter do not change anymore before updating the next parameter.

`mgcv` Should the **mgcv** `gam` function be used for computing updates in an `inner` iteration with working observations provided in an `outer` iteration.

`verbose` Print information during runtime of the algorithm.

`digits` Set the digits for printing when `verbose = TRUE`.

`flush` use `flush.console` for displaying the current output in the console.

`nu` Logical, numeric or `NULL`. Function `opt_bfit()` uses step length optimization of parameters when updating a model term, useful to encounter convergence problems of the algorithm. If `nu = TRUE` the step length parameter is optimized for each model term in each iteration of the backfitting algorithm. If `nu` is numeric, e.g. `nu = 1`, then `nu` is halfed until an improvement in the log-posterior is obtained or nu is smaller than `.Machine$double.eps`. If `nu = NULL`, no step length optimization is performed. Note, using very large data sets it is usually better to switch of step length optimization.

`stop.nu` Integer. Should step length reduction be stopped after `stop.nu` iterations of the backfitting algorithm?

`eta` The current value of the predictors, provided as a named list, one list entry for each parameter. The names correspond to the parameter names in the family object, see `family.bamlss`. E.g., when using the `gaussian_bamlss` family object, the current values for the mean can be extracted by `eta\$mu` and for the standard deviation by `eta\$sigma`.

`id` Character, the name of the current parameter for which the model term should be updated.

`...` For function `opt_bfit()`, arguments passed to function `bamlss.engine.setup`. For updating functions, within the dots argument the actual `iteration` number of the backfitting algorithm, the actual total number of equivalent degrees of freedom `edf` and vectors `z` and `hess` only if argument `outer = TRUE` are provided.

Details

This algorithm is based on iteratively weighted least squares (IWLS) for BAMLSS, i.e., a Newton-Raphson or Fisher scoring algorithm is applied, similar to Rigby~and~Stasinopoulos~(2005). The algorithm utilizes the chain rule for computing derivatives of the log-posterior w.r.t. regression coefficients, therefore, to compute the working observations and weights only the derivatives of the log-likelihood w.r.t. the predictors are required.

It is assumed that the provided `family` object holds functions for computing the first and second order derivatives of the log-likelihood w.r.t. the predictors. These Functions are provided

within the named lists "score" and "hess" within the family object. See the documentation of family.bamlss and the code of the provided families, e.g. gaussian_bamlss, for examples of the required structure.

The algorithm either updates each model term over all distributional parameters sequentially, or does a full update until convergence for model terms for one distributional parameter before updating the next parameter, see argument inner. Additionally, working observations and weights can be computed only once in an outer iteration.

Starting values of regression coefficients and smoothing variances can be supplied, moreover, if a family object holds functions for initializing the distributional parameters, see also family.bamlss, starting values are based on the initialize functions.

The default updating function for model terms is based on IWLS, which is assigned by function bamlss.engine.setup, however, special updating functions can be used. This is achieved by providing an updating function to argument update, which should be used for all model terms. Another option is to set the updating function within the xt argument of the **mgcv** smooth term constructor functions, see e.g. function s. If the xt list then holds an element named "update", which is a valid updating function, this updating function is used for the corresponding model term. This way it is possible to call different (special) updating functions for specific terms, e.g., that do not fit in the IWLS scheme. See the examples below. Note that this does not work if mgcv = TRUE, since the gam function assumes a strict linear representation of smooth terms.

A model term updating function has the following arguments:

```
update(x, family, y, eta, id, weights, criterion, ...)
```

Here x is an object as returned from function smooth.construct or smoothCon. The x object is preprocessed by function bamlss.engine.setup, i.e., an element called "state" is assigned. The state element represents the current state of the model term holding the current values of the parameters with corresponding fitted values, as well as equivalent degrees of freedom, see also the values that are returned by such functions below. The backfitting algorithm uses the state of a model term for generating updates of the parameters. Note that for special model terms the state list should already be provided within the call to the corresponding smooth constructor function, see the growth curve example below.

In addition, for special model terms the fitted values may not be computed by a linear combination of the design matrix and the coefficients. Therefore, the x object should hold an element named "fit.fun" which is a function for computing the fitted values. See also smooth.construct.bamlss.frame and predict.bamlss that use this setup. The arguments of fitting functions are

```
fit.fun(X, b, ...)
```

where X is the design matrix and b is the vector of coefficients. Hence, for usual IWLS updating the fitted values are computed by X %*% b. For special terms like nonlinear growth curves this may not be the case, see the example below. The fitting functions are assigned by bamlss.engine.setup, unless the function is already provided after calling the constructor function smooth.construct or smoothCon. Note that the dots argument is usually not needed by the user.

The default updating function is bfit_iwls(). Function bfit_iwls_Matrix() uses the sparse matrix infrastructures of package **Matrix**. The **Matrix** package and bfit_iwls_Matrix() is used for model terms where the maximum number of non-zero entries in the design matrix is less than half of the total number of columns, if an additional argument force.Matrix is set to TRUE in the opt_bfit() call.

The IWLS updating functions find optimum smoothing variances according to an information criterion using a stepwise approach, i.e., in each iteration and for each model term update the updating

functions try to find a better smoothing variance to control the trade-off between over-smoothing and nonlinear functional estimation. The search interval is centered around the current state of the smoothing variances, hence, in each iteration only a slight improvement is achieved. This algorithm is based on Belitz~and~Lang~(2008) and can also be viewed as a boosting approach for optimization.

Value

For function `opt_bfit()` a list containing the following objects:

`fitted.values`	A named list of the fitted values of the modeled parameters of the selected distribution.
`parameters`	The estimated set regression coefficients and smoothing variances.
`edf`	The equivalent degrees of freedom used to fit the model.
`logLik`	The value of the log-likelihood.
`logPost`	The value of the log-posterior.
`IC`	The value of the information criterion.
`converged`	Logical, indicating convergence of the backfitting algorithm.

For updating functions a list providing the current state

`fitted.values`	The resulting fitted values after updating.
`parameters`	The resulting named numeric vector of updated model term parameters. Coefficients should be named with `"b1"`, ..., `"bk"`, where `k` is the total number of coefficients. Smoothing variances should be named with `"tau21"`, ..., `"tau2m"`, where `m` is the total number of smoothing variances assigned to the model term.
`edf`	The equivalent degrees of freedom used to produce the fitted values.
`hessian`	Optional, the coefficient Hessian information
`log.prior`	Optional, the value of the log-prior of the model term.

References

Belitz C, Lang S (2008). Simultaneous Selection of Variables and Smoothing Parameters in Structured Additive Regression Models. *Computational Statistics & Data Analysis*, **53**, pp 61-81.

Umlauf N, Klein N, Zeileis A (2016). Bayesian Additive Models for Location Scale and Shape (and Beyond). *(to appear)*

Rigby, R. A. and Stasinopoulos D. M. (2005). Generalized additive models for location, scale and shape, (with discussion), *Appl. Statist.*, **54**, part 3, pp 507-554.

See Also

`bamlss`, `bamlss.frame`, `bamlss.engine.setup`, `set.starting.values`, `s2`

Examples

```
## Not run: ## Simulated data example illustrating
## how to call the optimizer function.
## This is done internally within
## the setup of function bamlss().
d <- GAMart(n = 200)
f <- num ~ s(x1) + s(x2) + s(x3)
bf <- bamlss.frame(f, data = d, family = "gaussian")
opt <- with(bf, opt_bfit(x, y, family))
print(str(opt))

## Same with bamlss().
b <- bamlss(f, data = d, family = "gaussian", sampler = FALSE)
plot(b)
summary(b)

## Use of different updating function.
b <- bamlss(f, data = d, family = "gaussian",
  sampler = FALSE, update = bfit_lm)
plot(b)

## Use mgcv gam() function for updating.
b <- bamlss(f, data = d, family = "gaussian",
  sampler = FALSE, mgcv = TRUE)
plot(b)

## Special smooth constructor including updating/sampler
## function for nonlinear Gompertz curves.
## Note: element special.npar is needed here since this
##       function has 3 parameters but the design matrix only
##       one column!
smooth.construct.gc.smooth.spec <- function(object, data, knots)
{
  object$X <- matrix(as.numeric(data[[object$term]]), ncol = 1)
  center <- if(!is.null(object$xt$center)) {
    object$xt$center
  } else TRUE
  object$by.done <- TRUE
  if(object$by != "NA")
    stop("by variables not supported!")
  object$fit.fun <- function(X, b, ...) {
    f <- b[1] * exp(-b[2] * exp(-b[3] * drop(X)))
    if(center)
      f <- f - mean(f)
    f
  }
  object$update <- bfit_optim
  object$propose <- GMCMC_slice
  object$prior <- function(b) { sum(dnorm(b, sd = 1000, log = TRUE)) }
  object$fixed <- TRUE
  object$state$parameters <- c("b1" = 0, "b2" = 0.5, "b3" = 0.1)
  object$state$fitted.values <- rep(0, length(object$X))
```

```
  object$state$edf <- 3
  object$special.npar <- 3 ## Important!
  class(object) <- c("gc.smooth", "no.mgcv", "special")
  object
}

## Work around for the "prediction matrix" of a growth curve.
Predict.matrix.gc.smooth <- function(object, data, knots)
{
  X <- matrix(as.numeric(data[[object$term]]), ncol = 1)
  X
}

## Heteroscedastic growth curve data example.
set.seed(111)

d <- data.frame("time" = 1:30)
d$y <- 2 + 1 / (1 + exp(0.5 * (15 - d$time))) +
  rnorm(30, sd = exp(-3 + 2 * cos(d$time/30 * 6 - 3)))

## Special model terms must be called with s2()!
f <- list(
  y ~ s2(time, bs = "gc"),
  sigma ~ s(time)
)

## Fit model with special model term.
b <- bamlss(f, data = d,
  optimizer = opt_bfit, sampler = sam_GMCMC)

## Plot the fitted curves.
plot(b)

## Predict with special model term.
nd <- data.frame("time" = seq(1, 30, length = 100))
p <- predict(b, newdata = nd, model = "mu", FUN = c95)
plot(d, ylim = range(c(d$y, p)))
matplot(nd$time, p, type = "l",
  lty = c(2, 1, 2), col = "black", add = TRUE)

## End(Not run)
```

---

opt_boost *Boosting BAMLSS*

Description

Optimizer functions for gradient and likelihood boosting with bamlss. In each boosting iteration the function selects the model term with the largest contribution to the log-likelihood, AIC or BIC.

Usage

```
## Gradient boosting optimizer.
opt_boost(x, y, family, weights = NULL,
  offset = NULL, nu = 0.1, nu.adapt = TRUE, df = 4, maxit = 400,
  mstop = NULL, maxq = NULL, qsel.splitfactor = FALSE,
  verbose = TRUE, digits = 4, flush = TRUE,
  eps = .Machine$double.eps^0.25,
  nback = NULL, plot = TRUE, initialize = TRUE,
  stop.criterion = NULL, select.type = 1,
  force.stop = TRUE, hatmatrix = !is.null(stop.criterion),
  reverse.edf = FALSE, approx.edf = FALSE,
  always = FALSE, ...)

boost(x, y, family, weights = NULL,
  offset = NULL, nu = 0.1, nu.adapt = TRUE, df = 4, maxit = 400,
  mstop = NULL, maxq = NULL, qsel.splitfactor = FALSE,
  verbose = TRUE, digits = 4, flush = TRUE,
  eps = .Machine$double.eps^0.25,
  nback = NULL, plot = TRUE, initialize = TRUE,
  stop.criterion = NULL, select.type = 1,
  force.stop = TRUE, hatmatrix = !is.null(stop.criterion),
  reverse.edf = FALSE, approx.edf = FALSE,
  always = FALSE, ...)

## Modified likelihood based boosting.
opt_boostm(x, y, family, offset = NULL,
  nu = 0.1, df = 3, maxit = 400, mstop = NULL,
  verbose = TRUE, digits = 4, flush = TRUE,
  eps = .Machine$double.eps^0.25, plot = TRUE,
  initialize = TRUE, stop.criterion = "BIC",
  force.stop = !is.null(stop.criterion),
  do.optim = TRUE, always = FALSE, ...)

boostm(x, y, family, offset = NULL,
  nu = 0.1, df = 3, maxit = 400, mstop = NULL,
  verbose = TRUE, digits = 4, flush = TRUE,
  eps = .Machine$double.eps^0.25, plot = TRUE,
  initialize = TRUE, stop.criterion = "BIC",
  force.stop = !is.null(stop.criterion),
  do.optim = TRUE, always = FALSE, ...)

## Boosting summary extractor.
boost_summary(object, ...)

## Plot all boosting paths.
boost_plot(x, which = c("loglik", "loglik.contrib", "parameters",
  "aic", "bic", "user"), intercept = TRUE, spar = TRUE, mstop = NULL,
  name = NULL, drop = NULL, labels = NULL, color = NULL, ...)
```

```
## Boosting summary printing and plotting.
## S3 method for class 'boost_summary'
print(x, summary = TRUE, plot = TRUE,
  which = c("loglik", "loglik.contrib"), intercept = TRUE,
  spar = TRUE, ...)
## S3 method for class 'boost_summary'
plot(x, ...)

## Model frame for out-of-sample selection.
boost_frame(formula, train, test, family = "gaussian", ...)
```

Arguments

x	For function opt_boost() the x list, as returned from function bamlss.frame, holding all model matrices and other information that is used for fitting the model. For the plotting function the corresponding bamlss object fitted with the opt_boost() optimizer.
y	The model response, as returned from function bamlss.frame.
family	A **bamlss** family object, see family.bamlss.
weights	Prior weights on the data, as returned from function bamlss.frame.
offset	Can be used to supply model offsets for use in fitting, returned from function bamlss.frame.
nu	Numeric, between [0, 1], controls the step size, i.e., the amount that should be added to model term parameters.
nu.adapt	Logical. If set to TRUE (default) step size nu is divided by 2, if current boosting iteration did not improve the loglikelihood.
df	Integer, defines the initial degrees of freedom that should be assigned to each smooth model term. May also be a named vector, the names must match the model term labels, e.g., as provided in summary.bamlss.
maxit	Integer, the maximum number of boosting iterations.
mstop	For convenience, overwrites maxit.
maxq	Integer, defines the maximum number of selected base-learners. The algorithm stops if this numer is exceeded.
qsel.splitfactor	Logical, if set to TRUE dummy variables of categorical predictors are counted individually.
name	Character, the name of the coefficient (group) that should be plotted. Note that the string provided in name will be removed from the labels on the 4th axis.
drop	Character, the name of the coefficient (group) that should not be plotted.
labels	A character string of labels that should be used on the 4 axis.
color	Colors or color function that creates colors for the (group) paths.
verbose	Print information during runtime of the algorithm.
digits	Set the digits for printing when verbose = TRUE.

flush	use flush.console for displaying the current output in the console.
eps	The tolerance used as stopping mechanism, see argument nback.
nback	Integer. If nback is not NULL, then the algorithm stops if the the change in the log-likelihood of the last nback iterations is smaller or equal to eps. If maxit = NULL the maximum number of iterations is set to 10000.
plot	Should the boosting summary be printed and plotted?
initialize	Logical, should intercepts be initialized?
stop.criterion	Character, selects the information criterion that should be used to determine the optimum number of boosting iterations. Either "AIC" or "BIC" is possible. Note that this feature requires to compute hat-matrices for each distributional parameter, therefore, the routine may be slow and computer storage intensive.
select.type	Should model terms be selected by the log-likelihood contribution, select.type = 1, or by the corresponding stop.criterion, select.type = 2.
force.stop	Logical, should the algorithm stop if the information criterion increases?
do.optim	Logical. Should smoothing parameters be optimized in each boosting iteration?
hatmatrix	Logical, if set to TRUE the hat-matrices for each distributional parameter will be computed. The hat-matrices are used to determine the effective (equivalent) degrees of freedom in each boosting iteration, i.e., it is possible to compute information criteria like the AIC or BIC for selecting the optimum number of boosting iterations.
reverse.edf	Logical. Instead of computing degrees of freedom with hat-matrices, the actual smoothing parameters are reverse engineered to compute the corresponding actual smoother matrix. Note that this option is still experimental.
approx.edf	Logical. Another experimental and fast approximation of the degrees of freedom.
always	Logical or character. Should the intercepts forced to be updated in each boosting iteration? If always = TRUE each intercept of each distributional parameter is updated, if always = "best" only the intercept corresponding to the distributional of the best fitting model term is updated.
object	A bamlss object that was fitted using opt_boost().
summary	Should the summary be printed?
which	Which of the three provided plots should be created?
intercept	Should the coefficient paths of intercepts be dropped in the plot?
spar	Should graphical parmeters be set with par?
formula	See bamlss.frame.
train, test	Data frames used for training and testing the model..
...	For function opt_boost(), arguments passed to bamlss.engine.setup. for function boost_summary() arguments passed to function print.boost_summary().

Value

For function boost_summary() a list containing information on selection frequencies etc. For function opt_boost() and opt_boostm() a list containing the following objects:

fitted.values	A named list of the fitted values based on the last boosting iteration of the modeled parameters of the selected distribution.
parameters	A matrix, each row corresponds to the parameter values of one boosting iteration.
boost_summary	The boosting summary which can be printed and plotted.

WARNINGS

The function does not take care of variable scaling for the linear parts! This must be done by the user, e.g., one option is to use argument scale.d in function bamlss.frame, which uses scale.

Function opt_boost() does not select the optimum stopping iteration! The modified likelihood based algorithm implemented in function opt_boostm() is still experimental!

See Also

bamlss.frame, bamlss

Examples

```
## Not run: ## Simulate data.
set.seed(123)
d <- GAMart()

## Estimate model.
f <- num ~ x1 + x2 + x3 + lon + lat +
  s(x1) + s(x2) + s(x3) + s(lon) + s(lat) + te(lon,lat)

b <- bamlss(f, data = d, optimizer = opt_boost,
  sampler = FALSE, scale.d = TRUE, nu = 0.01,
  maxit = 1000, plot = FALSE)

## Plot estimated effects.
## plot(b)

## Print and plot the boosting summary.
boost_summary(b, plot = FALSE)
## boost_plot(b, which = 1)
## boost_plot(b, which = 2)
## boost_plot(b, which = 3, name = "mu.s.te(lon,lat).")

## Extract estimated parameters for certain
## boosting iterations.
parameters(b, mstop = 1)
parameters(b, mstop = 100)

## Also works with predict().
```

```
head(do.call("cbind", predict(b, mstop = 1)))
head(do.call("cbind", predict(b, mstop = 100)))

## Another example using the modified likelihood
## boosting algorithm.
f <- list(
  num ~ x1 + x2 + x3 + lon + lat +
    s(x1) + s(x2) + s(x3) + s(lon) + s(lat) + te(lon,lat),
  sigma ~ x1 + x2 + x3 + lon + lat +
    s(x1) + s(x2) + s(x3) + s(lon) + s(lat) + te(lon,lat)
)

b <- bamlss(f, data = d, optimizer = opt_boostm,
  sampler = FALSE, scale.d = TRUE, nu = 0.05,
  maxit = 400, stop.criterion = "AIC", force.stop = FALSE)

## Plot estimated effects.
## plot(b)

## Plot AIC and log-lik contributions.
## boost_plot(b, "AIC")
## boost_plot(b, "loglik.contrib")

## Out-of-sample selection of model terms.
set.seed(123)
d <- GAMart(n = 5000)

## Split data into training and testing
i <- sample(1:2, size = nrow(d), replace = TRUE)
dtest <- subset(d, i == 1)
dtrain <- subset(d, i == 2)

## Model formula
f <- list(
  num ~ s(x1) + s(x2) + s(x3),
  sigma ~ s(x1) + s(x2) + s(x3)
)

## Create model frame for out-of-sample selection.
sm <- boost_frame(f, train = dtrain, test = dtest, family = "gaussian")

## Out-of-sample selection function.
sfun <- function(parameters) {
  sm$parameters <- parameters
  p <- predict(sm, type = "parameter")
  -1 * sum(sm$family$d(dtest$num, p, log = TRUE))
}

## Start boosting with out-of-sample negative
## log-likelihood selection of model terms.
b <- bamlss(f, data = dtrain, sampler = FALSE, optimizer = opt_boost,
  selectfun = sfun, always = "best")
```

```
## Plot curve of negative out-of-sample log-likelihood.
## boost_plot(b, which = "user")

## End(Not run)
```

---

opt_Cox *Cox Model Posterior Mode Estimation*

Description

This function computes posterior mode estimates of the parameters of a flexible Cox model with structured additive predictors using a Newton-Raphson algorithm. Integrals are solved numerically. Moreover, optimum smoothing variances are computed using a stepwise optimization, see also the details section of function bfit.

Usage

```
opt_Cox(x, y, start, weights, offset,
  criterion = c("AICc", "BIC", "AIC"),
  nu = 0.1, update.nu = TRUE,
  eps = .Machine$double.eps^0.25, maxit = 400,
  verbose = TRUE, digits = 4, ...)

cox_mode(x, y, start, weights, offset,
  criterion = c("AICc", "BIC", "AIC"),
  nu = 0.1, update.nu = TRUE,
  eps = .Machine$double.eps^0.25, maxit = 400,
  verbose = TRUE, digits = 4, ...)
```

Arguments

x	The x list, as returned from function bamlss.frame and transformed by function surv_transform, holding all model matrices and other information that is used for fitting the model.
y	The model response, as returned from function bamlss.frame.
start	A named numeric vector containing possible starting values, the names are based on function parameters.
weights	Prior weights on the data, as returned from function bamlss.frame.
offset	Can be used to supply model offsets for use in fitting, returned from function bamlss.frame.
criterion	Set the information criterion that should be used, e.g., for smoothing variance selection. Options are the corrected AIC "AICc", the "BIC" and "AIC".
nu	Calibrates the step length of parameter updates of one Newton-Raphson update.
update.nu	Should the updating step length be optimized in each iteration of the backfitting algorithm.

eps	The relative convergence tolerance of the backfitting algorithm.
maxit	The maximum number of iterations for the backfitting algorithm
verbose	Print information during runtime of the algorithm.
digits	Set the digits for printing when verbose = TRUE.
...	Currently not used.

Value

A list containing the following objects:

fitted.values	A named list of the fitted values of the modeled parameters of the selected distribution.
parameters	The estimated set regression coefficients and smoothing variances.
edf	The equivalent degrees of freedom used to fit the model.
logLik	The value of the log-likelihood.
logPost	The value of the log-posterior.
hessian	The Hessian matrix evaluated at the posterior mode.
converged	Logical, indicating convergence of the backfitting algorithm.
time	The runtime of the algorithm.

References

Umlauf N, Klein N, Zeileis A (2016). Bayesian Additive Models for Location Scale and Shape (and Beyond). *(to appear)*

See Also

sam_Cox, cox_bamlss, surv_transform, simSurv, bamlss

Examples

```
## Please see the examples of function sam_Cox()!
```

---

opt_isgd *Implicit Stochastic Gradient Descent Optimizer*

Description

This optimizer performs an implicit stochastic gradient descent algorithm. It is mainly used within a bamlss call.

Usage

```
opt_isgd(x, y, family, weights = NULL, offset = NULL,
     gammaFun = function(i) 1/(1 + i), shuffle = TRUE,
     CFun = function(beta) diag(length(beta)),
     start = NULL, i.state = 0)
```

Arguments

x	For function boost() the x list, as returned from function bamlss.frame, holding all model matrices and other information that is used for fitting the model.
y	The model response, as returned from function bamlss.frame.
family	A **bamlss** family object, see family.bamlss.
weights	Prior weights on the data, as returned from function bamlss.frame.
offset	Can be used to supply model offsets for use in fitting, returned from function bamlss.frame.
gammaFun	Function specifying the step length.
shuffle	Should the data be shuffled?
CFun	Hessian approximating function.
start	Vector of starting values.
i.state	Added to gammaFUN().

Details

tpf

Value

For function opt_isgd() a list containing the following objects:

fitted.values	A named list of the fitted values based on the last iteration of the modeled parameters of the selected distribution.
parameters	A matrix, each row corresponds to the parameter values of one iteration.
sgd.summary	The summary of the stochastic gradient descent algorithm which can be printed and plotted.

Warning

CAUTION: Arguments weights and offset are not implemented yet!

Note

Motivated by the lecture 'Regression modelling with large data sets' given by Ioannis Kosmidis in Innsbruck, January 2017.

Author(s)

Thorsten Simon

References

Toulis, P and Airoldi, EM (2015): Scalable estimation strategies based on stochastic approximations: Classical results and new insights. Statistics and Computing, 25, no. 4, 781–795. doi: 10.1007/s11222-015-9560-y

See Also

bamlss.frame, bamlss

Examples

```
## Not run:
set.seed(111)
d <- GAMart(n = 10000)
f <- num ~ s(x1) + s(x2) + s(x3) + te(lon, lat)
b <- bamlss(f, data = d, optimizer = opt_isgd, sampler = FALSE)
plot(b, ask = F)

## loop over observations a 2nd time
b <- bamlss(f, data = d, optimizer = opt_isgd, sampler = FALSE, start = parameters(b),
            i.state = b$model.stats$optimizer$sgd.summary$i.state)
plot(b, ask = F)

## try differeent gammaFuns, e.g.,
# gammaFun <- function(i) .3/sqrt((1+i)) + 0.001

## testing some families
f2 <- bin ~ s(x1) + s(x2) + s(x3) + te(lon, lat)
b2 <- bamlss(f2, data = d, optimizer = opt_isgd, sampler = FALSE, family = "binomial")

f3 <- cens ~ s(x1) + s(x2) + s(x3) + te(lon, lat)
b3 <- bamlss(f3, data = d, optimizer = opt_isgd, sampler = FALSE, family = "cnorm")

## End(Not run)
```

---

parameters *Extract or Initialize Parameters for BAMLSS*

Description

The function either sets up a list of all parameters of a bamlss.frame, which can be used for setting up models, or extracts the estimated parameters of a bamlss object.

Usage

```
parameters(x, model = NULL, start = NULL,
  fill = c(0, 1e-04), list = FALSE,
  simple.list = FALSE, extract = FALSE,
  ...)
```

Arguments

x	A bamlss.frame or bamlss object.
model	The model name for which parameters should be initialized or extracted.

start	A named numeric vector which should be used when creating the parameter list. See also function link{set.starting.values}
fill	Numeric, when setting up a parameter list, the values the should be used for regression coefficients (first element of fill) and for smoothing variances (second element of fill).
list	Should the function return a list of all parameters?
simple.list	Should the names of parameter vectors be dropped?
extract	Should parameters of a bamlss.frame be extracted or initialized?
...	Currently not used.

Details

Parameters for BAMLSS are used for optimizer functions in function bamlss. The function is useful for initializing all parameters given a bamlss.frame (which is done internally in function bamlss), but also for extracting all estimated parameters of some optimizer.

The naming convention of the parameter list is used by a couple of functions in this package. For each parameter of the modeled distribution, e.g., gaussian_bamlss has parameters "mu" and "sigma", a list element is created. These elements the contain the list of all model term parameters. Parametric model terms are indicated with "p" and smooth model terms with "s". If the design matrix of a model term in the x list of a bamlss.frame does not contain any columns names, then the parameters are named with a leading "b", otherwise the column names of the design matrix are used. Smoothing variances parameter vectors are named with a leading "tau2".

The naming convention is useful when setting up new model fitting engines for bamlss and is used, e.g., by bfit and GMCMC, which are based on parameter state list objects as provided by function bamlss.engine.setup.

Value

A named list of all parameters of a bamlss.frame or bamlss object.

See Also

bamlss.frame, bamlss, opt_bfit, sam_GMCMC, get.par, set.par

Examples

```
## Create a "bamlss.frame"
set.seed(123)
d <- GAMart()
bf <- bamlss.frame(num ~ s(x1) + te(lon,lat), data = d)

## Create list of all parameters from "bamlss.frame".
p <- parameters(bf, list = TRUE)
str(p)

## Not run: ## Estimate model.
f <- list(num ~ s(x1) + te(lon,lat), sigma ~ s(x1))
b <- bamlss(f, data = d, sampler = FALSE)
```

```
## Extract estimated parameters.
parameters(b)
parameters(b, list = TRUE)

## End(Not run)
```

`pathplot` *Plot Coefficients Paths*

Description

This is a simple wrapper function to plot coefficients paths obtained from the boosting optimizer function `boost` and the LASSO optimizer `lasso`.

Usage

```
pathplot(object, ...)
```

Arguments

`object`	An object of class `"bamlss"`.
`...`	Arguments passed to `boost_plot` or `lasso_plot`.

See Also

`boost_plot`, `lasso_plot`

`plot.bamlss` *Plotting BAMLSS*

Description

Plotting methods for objects of class `"bamlss"` and `"bamlss.results"`, which can be used for producing effect plots of model terms, trace plots of samples or residual plots. Note that effect plots of model terms with more than two covariates are not supported, for this purpose use function `predict.bamlss`.

Usage

```
## S3 method for class 'bamlss'
plot(x, model = NULL, term = NULL,
  which = "effects", parameters = FALSE,
  ask = dev.interactive(), spar = TRUE, ...)

## S3 method for class 'bamlss.results'
plot(x, model = NULL, term = NULL,
  ask = dev.interactive(), scale = 1, spar = TRUE, ...)
```

Arguments

x	An object of class "bamlss" or "bamlss.results".
model	Character or integer. For which model should the plots be created?
term	Character or integer. For which model term should a plot be created?
which	Character or integer, selects the type of plot: "effects" produces effect plots of smooth model terms, "samples" shows trace plots of samples, "hist-resid" shows a histogram of residuals (see also residuals.bamlss for the different available types), "qq-resid" shows a quantile-quantile plot of residuals, "scatter-resid" shows a scatter plot of residuals with fitted values for the distribution mean (if available in the family object), "max-acf" shows an acf plot of the maximum autocorrelation of all parameter samples.
parameters	For trace plots of parameters, should corresponding parameter values as returned from an optimizer function (e.g., opt_bfit) be added as horizontal lines?
ask	For multiple plots, the user is asked to show the next plot.
scale	If set to 1, effect plots all have the same scale on the y-axis. If set to 0 each effect plot has its own scale for the y-axis.
spar	Should graphical parameters be set?
...	Arguments to be passed to plot2d, plot3d, sliceplot, plotblock, plotmap and residuals.bamlss.

See Also

bamlss, results.bamlss.default, residuals.bamlss.

Examples

```
## Not run: ## Generate some data.
d <- GAMart()

## Model formula.
f <- list(
  num ~ s(x1) + s(x2) + s(x3) + te(lon,lat),
  sigma ~ s(x2) + te(lon,lat)
)

## Estimate model.
b <- bamlss(f, data = d)

## Effect plots
plot(b, ask = FALSE)
plot(b, model = "mu")
plot(b, model = "sigma", term = "te(lon,lat)")

## Trace plots.
plot(b, which = "samples")

## Residual plots.
plot(b, which = 3:4)
```

```
## End(Not run)
```

---

plot2d *Plot 2D Effects*

Description

Function to plot simple 2D graphics for univariate effects/functions.

Usage

```
plot2d(x, residuals = FALSE, rug = FALSE, jitter = TRUE,
  col.residuals = NULL, col.lines = NULL, col.polygons = NULL,
  col.rug = NULL, c.select = NULL, fill.select = NULL,
  data = NULL, sep = "", month = NULL, year = NULL,
  step = 12, shift = NULL, trans = NULL,
  scheme = 2, s2.col = NULL, grid = 50, ...)
```

Arguments

x	A matrix or data frame, containing the covariate for which the effect should be plotted in the first column and at least a second column containing the effect. Another possibility is to specify the plot via a formula, e.g. y ~ x, see the examples. x may also be a character file path to the data to be used for plotting.
residuals	If set to TRUE, residuals may also be plotted if available. Residuals must be supplied as an attribute named "residuals", which is a matrix or data frame where the first column is the covariate and the second column the residuals.
rug	Add a rug to the plot.
jitter	If set to TRUE a jittered rug plot is added.
col.residuals	The color of the partial residuals.
col.lines	The color of the lines.
col.polygons	Specify the background color of polygons, if x has at least 3 columns, i.e. column 2 and 3 can form one polygon.
col.rug	Specify the color of the rug representation.
c.select	Integer vector of maximum length of columns of x, selects the columns of the resulting data matrix that should be used for plotting. E.g. if x has 5 columns, then c.select = c(1, 2, 5) will select column 1, 2 and 5 for plotting. Note that first element of c.select should always be the column that holds the variable for the x-axis.
fill.select	Integer vector, select pairwise the columns of the resulting data matrix that should form one polygon with a certain background color specified in argument col. E.g. x has three columns, or is specified with formula f1 + f2 ~ x, then setting fill.select = c(0, 1, 1) will draw a polygon with f1 and f2 as

	boundaries. If x has five columns or the formula is e.g. f1 + f2 + f3 + f4 ~ x, then setting fill.select = c(0, 1, 1, 2, 2), the pairs f1, f2 and f3, f4 are selected to form two polygons.
data	If x is a formula, a data.frame or list. By default the variables are taken from environment(x): typically the environment from which plot2d is called. Note that data may also be a character file path to the data.
sep	The field separator character when x or data is a character, see function read.table.
month, year, step	Provide specific annotation for plotting estimation results for temporal variables. month and year define the minimum time point whereas step specifies the type of temporal data with step = 4, step = 2 and step = 1 corresponding to quarterly, half yearly and yearly data.
shift	Numeric constant to be added to the smooth before plotting.
trans	Function to be applied to the smooth before plotting, e.g., to transform the plot to the response scale.
scheme	Sets the plotting scheme for polygons, possible values are 1 and 2.
s2.col	The color for the second plotting scheme.
grid	Integer, specifies the number of polygons for the second plotting scheme.
...	Other graphical parameters, please see the details.

Details

For 2D plots the following graphical parameters may be specified additionally:

- cex: Specify the size of partial residuals,
- lty: The line type for each column that is plotted, e.g. lty = c(1, 2),
- lwd: The line width for each column that is plotted, e.g. lwd = c(1, 2),
- poly.lty: The line type to be used for the polygons,
- poly.lwd: The line width to be used for the polygons,
- density angle, border: See polygon,
- ...: Other graphical parameters, see function plot.

See Also

plot3d, plotmap, plotblock, sliceplot.

Examples

```
## Generate some data.
set.seed(111)
n <- 500
## Regressor.
d <- data.frame(x = runif(n, -3, 3))

## Response.
d$y <- with(d, 10 + sin(x) + rnorm(n, sd = 0.6))
```

```
## Not run: ## Estimate model.
b <- bamlss(y ~ s(x), data = d)
summary(b)

## Plot estimated effect.
plot(b)
plot(b, rug = FALSE)

## Extract fitted values.
f <- fitted(b, model = "mu", term = "s(x)")
f <- cbind(d["x"], f)

## Now use plot2d.
plot2d(f)
plot2d(f, fill.select = c(0, 1, 0, 1))
plot2d(f, fill.select = c(0, 1, 0, 1), lty = c(2, 1, 2))
plot2d(f, fill.select = c(0, 1, 0, 1), lty = c(2, 1, 2),
  scheme = 2)

## Variations.
plot2d(sin(x) ~ x, data = d)
d$f <- with(d, sin(d$x))
plot2d(f ~ x, data = d)
d$f1 <- with(d, f + 0.1)
d$f2 <- with(d, f - 0.1)
plot2d(f1 + f2 ~ x, data = d)
plot2d(f1 + f2 ~ x, data = d, fill.select = c(0, 1, 1), lty = 0)
plot2d(f1 + f2 ~ x, data = d, fill.select = c(0, 1, 1), lty = 0,
  density = 20, poly.lty = 1, poly.lwd = 2)
plot2d(f1 + f + f2 ~ x, data = d, fill.select = c(0, 1, 0, 1),
  lty = c(0, 1, 0), density = 20, poly.lty = 1, poly.lwd = 2)

## End(Not run)
```

---

plot3d *Plot 3D Effects*

Description

Function to plot 3D graphics or image and/or contour plots for bivariate effects/functions.

Usage

```
plot3d(x, residuals = FALSE, col.surface = NULL,
  ncol = 99L, swap = FALSE, col.residuals = NULL, col.contour = NULL,
  c.select = NULL, grid = 30L, image = FALSE, contour = FALSE,
  legend = TRUE, cex.legend = 1, breaks = NULL, range = NULL,
  digits = 2L, d.persp = 1L, r.persp = sqrt(3), outscale = 0,
  data = NULL, sep = "", shift = NULL, trans = NULL,
```

```
type = "mba", linear = FALSE, extrap = FALSE,
k = 40, ...)
```

Arguments

x	A matrix or data frame, containing the covariates for which the effect should be plotted in the first and second column and at least a third column containing the effect. Another possibility is to specify the plot via a formula, e.g. for simple plotting of bivariate surfaces z ~ x + y, see the examples. x may also be a character file path to the data to be used for plotting.
residuals	If set to TRUE, residuals may also be plotted if available. Residuals must be supplied as an attribute named "residuals", which is a matrix or data frame where the first two columns are covariates and the third column the residuals.
col.surface	The color of the surface, may also be a function, e.g. col.surface = heat.colors.
ncol	the number of different colors that should be generated, if col.surface is a function.
swap	If set to TRUE colors will be represented in reverse order.
col.residuals	The color of the partial residuals, or if contour = TRUE the color of the contour lines.
col.contour	The color of the contour lines.
c.select	Integer vector of maximum length of columns of x, selects the columns of the resulting data matrix that should be used for plotting. E.g. if x has 5 columns, then c.select = c(1, 2, 5) will select column 1, 2 and 5 for plotting. If c.select = 95 or c.select = 80, function plot3d will search for the corresponding columns to plot a 95% or 80% confidence surfaces respectively. Note that if e.g. c.select = c(1, 2), plot3d will use columns 1 + 2 and 2 + 2 for plotting.
grid	The grid size of the surface(s).
image	If set to TRUE, an image.plot is drawn.
contour	If set to TRUE, a contour plot is drawn.
legend	If image = TRUE an additional legend may be added to the plot.
cex.legend	The expansion factor for the legend text, see text.
breaks	A set of breakpoints for the colors: must give one more breakpoint than ncol.
range	Specifies a certain range values should be plotted for.
digits	Specifies the legend decimal places.
d.persp	See argument d in function persp.
r.persp	See argument r in function persp.
outscale	Scales the outer ranges of x and z limits used for interpolation.
data	If x is a formula, a data.frame or list. By default the variables are taken from environment(x): typically the environment from which plot3d is called. Note that data may also be a character file path to the data.
sep	The field separator character when x or data is a character, see function read.table.

shift	Numeric constant to be added to the smooth before plotting.
trans	Function to be applied to the smooth before plotting, e.g., to transform the plot to the response scale.
type	Character, which type of interpolation method should be used. The default is type = "akima", see function interp. The two other options are type = "mba", which calls function mba.surf of package **MBA**, or type = "mgcv", which uses a spatial smoother withing package **mgcv** for interpolation. The last option is definitely the slowest, since a full regression model needs to be estimated.
linear	Logical, should linear interpolation be used withing function interp?
extrap	Logical, should interpolations be computed outside the observation area (i.e., extrapolated)?
k	Integer, the number of basis functions to be used to compute the interpolated surface when type = "mgcv".
...	Parameters passed to colorlegend if an image plot with legend is drawn, also other graphical parameters, please see the details.

Details

For 3D plots the following graphical parameters may be specified additionally:

- cex: Specify the size of partial residuals,
- col: It is possible to specify the color for the surfaces if se > 0, then e.g. col = c("green", "black", "red"),
- pch: The plotting character of the partial residuals,
- ...: Other graphical parameters passed functions persp, image.plot and contour.

Note

Function plot3d can use the **akima** package to construct smooth interpolated surfaces, therefore, package **akima** needs to be installed. The **akima** package has an ACM license that restricts applications to non-commercial usage, see

https://www.acm.org/publications/policies/software-copyright-notice

Function plot3d prints a note referring to the ACM license. This note can be suppressed by setting

options("use.akima" = TRUE)

See Also

colorlegend, plot2d, plotmap, plotblock, sliceplot.

Examples

```
## Generate some data.
set.seed(111)
n <- 500

## Regressors.
```

```
d <- data.frame(z = runif(n, -3, 3), w = runif(n, 0, 6))

## Response.
d$y <- with(d, 1.5 + cos(z) * sin(w) + rnorm(n, sd = 0.6))

## Not run: ## Estimate model.
b <- bamlss(y ~ s(z,w), data = d)
summary(b)

## Plot estimated effect.
plot(b, model = "mu", term = "s(z,w)")

## Extract fitted values.
f <- fitted(b, model = "mu", term = "s(z,w)", intercept = FALSE)
f <- cbind(d[, c("z", "w")], f)

## Now use plot3d().
plot3d(f)
plot3d(f, swap = TRUE)
plot3d(f, grid = 100, border = NA)

## Only works if columns are named with
## '2.5
plot3d(f, c.select = 95, border = c("red", NA, "green"),
  col.surface = c(1, NA, 1), resid = TRUE, cex.resid = 0.2)

## Now some image and contour.
# plot3d(f, image = TRUE, legend = FALSE)
# plot3d(f, image = TRUE, legend = TRUE)
# plot3d(f, image = TRUE, contour = TRUE)
# plot3d(f, image = TRUE, contour = TRUE, swap = TRUE)
# plot3d(f, image = TRUE, contour = TRUE, col.contour = "white")
# plot3d(f, contour = TRUE)
# plot3d(f, image = TRUE, contour = TRUE, c.select = 3)
# plot3d(f, image = TRUE, contour = TRUE, c.select = "Mean")
# plot3d(f, image = TRUE, contour = TRUE, c.select = "97.5

## End(Not run)

## Variations.
d$f1 <- with(d, sin(z) * cos(w))
with(d, plot3d(cbind(z, w, f1)))

## Same with formula.
plot3d(sin(z) * cos(w) ~ z + w, zlab = "f(z,w)", data = d)
plot3d(sin(z) * cos(w) ~ z + w, zlab = "f(z,w)", data = d,
  ticktype = "detailed")

## Play with palettes.
plot3d(sin(z) * cos(w) ~ z + w, col.surface = heat.colors, data = d)
plot3d(sin(z) * cos(w) ~ z + w, col.surface = topo.colors, data = d)
plot3d(sin(z) * cos(w) ~ z + w, col.surface = cm.colors, data = d)
plot3d(sin(z) * cos(w) ~ z + w, col.surface = rainbow, data = d)
```

```
plot3d(sin(z) * cos(w) ~ z + w, col.surface = terrain.colors, data = d)

plot3d(sin(z) * cos(w) ~ z + w, col.surface = rainbow_hcl, data = d)
plot3d(sin(z) * cos(w) ~ z + w, col.surface = diverge_hcl, data = d)
plot3d(sin(z) * cos(w) ~ z + w, col.surface = sequential_hcl, data = d)

plot3d(sin(z) * cos(w) ~ z + w,
  col.surface = rainbow_hcl(n = 99, c = 300, l = 80, start = 0, end = 100),
  data = d)
# plot3d(sin(z) * cos(w) ~ z + w,
#  col.surface = rainbow_hcl(n = 99, c = 300, l = 80, start = 0, end = 100),
#  image = TRUE, grid = 200, data = d)
```

---

plotblock *Factor Variable and Random Effects Plots*

Description

Function to plot effects for model terms including factor, or group variables for random effects.

Usage

```
plotblock(x, residuals = FALSE, range = c(0.3, 0.3),
  col.residuals = "black", col.lines = "black", c.select = NULL,
  fill.select = NULL , col.polygons = NULL, data = NULL,
  shift = NULL, trans = NULL, labels = NULL, ...)
```

Arguments

x	Either a `list` of length of the unique factors, where each `list` element contains the estimated effects for one factor as a matrix, or one data matrix with first column as the group or factor variable. Also formulas are accepted, e.g it is possible to specify the plot with `f ~ x` or `f1 + f2 ~ x`. By convention, the covariate for which effects should be plotted, is always in the first column in the resulting data matrix, that is used for plotting, i.e. in the second formula example, the data matrix is `cbind(x, f1, f2)`, also see argument `c.select` and `fill.select`.
residuals	If set to `TRUE`, residuals will be plotted if available. Residuals may be set as an attribute of `x` named `"residuals"`, where the residuals must be a matrix with first column specifying the covariate, and second column the residuals that should be plotted.
range	Numeric vector, specifying the left and right bound of the block.
col.residuals	The color of the partial residuals.
col.lines	Vector of maximum length of columns of `x` minus 1, specifying the color of the lines.

c.select	Integer vector of maximum length of columns of x, selects the columns of the resulting data matrix that should be used for plotting. E.g. if x has 5 columns, then c.select = c(1, 2, 5) will select column 1, 2 and 5 for plotting. Note that first element of c.select should always be 1, since this is the column of the covariate the effect is plotted for.
fill.select	Integer vector, select pairwise the columns of the resulting data matrix that should form one polygon with a certain background color specified in argument col. E.g. x has three columns, or is specified with formula f1 + f2 ~ x, then setting fill.select = c(0, 1, 1) will draw a polygon with f1 and f2 as boundaries. If x has five columns or the formula is e.g. f1 + f2 + f3 + f4 ~ x, then setting fill.select = c(0, 1, 1, 2, 2), the pairs f1, f2 and f3, f4 are selected to form two polygons.
col.polygons	Specify the background color for the upper and lower confidence bands, e.g. col = c("green", "red").
data	If x is a formula, a data.frame or list. By default the variables are taken from environment(x): typically the environment from which plotblock is called.
shift	Numeric constant to be added to the smooth before plotting.
trans	Function to be applied to the smooth before plotting, e.g., to transform the plot to the response scale.
labels	Character, labels for the factor levels.
...	Graphical parameters, please see the details.

Details

Function plotblock draws for every factor or group the effect as a "block" in one graphic, i.e., similar to boxplots, estimated fitted effects, e.g., containing quantiles of MCMC samples, are drawn as one block, where the upper lines represent upper quantiles, the middle line the mean or median, and lower lines lower quantiles, also see the examples. The following graphical parameters may be supplied additionally:

- cex: Specify the size of partial residuals,
- lty: The line type for each column that is plotted, e.g. lty = c(1, 2),
- lwd: The line width for each column that is plotted, e.g. lwd = c(1, 2),
- poly.lty: The line type to be used for the polygons,
- poly.lwd: The line width to be used for the polygons,
- density angle, border: See polygon,
- ...: Other graphical parameters, see function plot.

See Also

plot2d, plot3d, plotmap, sliceplot.

Examples

```
## Generate some data.
set.seed(111)
n <- 500

## Regressors.
d <- data.frame(fac = factor(rep(1:10, n/10)))

## Response.
d$y <- with(d, 1.5 + rnorm(10, sd = 0.6)[fac] +
  rnorm(n, sd = 0.6))

## Not run: ## Estimate model.
b <- bamlss(y ~ s(fac,bs="re"), data = d)
summary(b)

## Plot random effects.
plot(b)

## Extract fitted values.
f <- fitted(b, model = "mu", term = "fac")
f <- cbind(d["fac"], f)

## Now use plotblock.
plotblock(f)

## Variations.
plotblock(f, fill.select = c(0, 1, 0, 1), col.poly = "red")
plotblock(f, fill.select = c(0, 1, 0, 1), col.poly = "lightgray",
  lty = c(2, 1, 2), lwd = c(2, 1, 2))

## End(Not run)

## More examples.
plotblock(y ~ fac, data = d, range = c(0.45, 0.45))

d <- data.frame(fac = factor(rep(1:10, n/10)))
d$y <- with(d, c(2.67, 5, 6, 3, 4, 2, 6, 7, 9, 7.5)[fac])
plotblock(y ~ fac, data = d)
plotblock(cbind(y - 0.1, y + 0.1) ~ fac, data = d)
```

---

plotmap *Plot Maps*

Description

The function takes a list polygons and draws the corresponding map. Different colors for each polygon can be used.

Usage

```
plotmap(map, x = NA, id = NULL, select = NULL,
  legend = TRUE, names = FALSE, values = FALSE, ...)
```

Arguments

map	The map to be plotted, usually an object that inherits from SpatialPolygons, but may also be a list of polygons, i.e., each list entry is a matrix with x- and y-coordinates.
x	A vector, data.frame or matrix. In the latter case x should have two columns, one column that is the region identifier and one that contains the values to be plotted. In case x is a data.frame, the function searches for "character" or "factor" columns for the region identifier. If x is a matrix, the second column of x is supposed to be the region identifier. If x = NA and map is an object of class "SpatialPolygonsDataFrame" only the polygons will be plotted without using the data.
id	If argument x is a vector, argument id should contain the region identifier vector.
select	Select the column of the data in x which should be used for plotting, may be an integer or character with the corresponding column name.
legend	Should a color legend be added to the plot, see also function colorlegend.
names	If set to TRUE the name for each polygon will also be plotted at the centroids of the corresponding polygons.
values	If set to TRUE the corresponding values for each polygon will also be plotted at the centroids of the polygons.
...	Arguments to be passed to colorlegend and others, e.g. change the border of the polygons and density (mdensity for missing regions in id), see polygon.

See Also

plot2d, plot3d, sliceplot, plotblock.

Examples

```
## Example from mgcv ?mrf.
## Load Columbus Ohio crime data (see ?columbus for details and credits).
data("columb", package = "mgcv")
data("columb.polys", package = "mgcv")

## Plot the map.
plotmap(columb.polys)

## Plot aggregated data.
a <- with(columb, aggregate(crime,
  by = list("district" = district), FUN = mean))

plotmap(columb.polys, x = a$x, id = a$district)
plotmap(columb.polys, x = a$x, id = a$district,
  pos = "topleft")
```

```
plotmap(columb.polys, x = a$x, id = a$district,
  pos = "topleft", side.legend = 2)
plotmap(columb.polys, x = a$x, id = a$district,
  pos = "topleft", side.legend = 2, side.ticks = 2)
plotmap(columb.polys, x = a$x, id = a$district,
  pos = "topleft", side.legend = 2, side.ticks = 2,
  col = heat_hcl, swap = TRUE)
plotmap(columb.polys, x = a$x, id = a$district,
  pos = "topleft", side.legend = 2, side.ticks = 2,
  col = heat_hcl, swap = TRUE, range = c(10, 50))
plotmap(columb.polys, x = a$x, id = a$district,
  pos = "topleft", side.legend = 2, side.ticks = 2,
  col = heat_hcl(5), swap = TRUE, range = c(10, 50),
  lrange = c(0, 60))
```

`predict.bamlss` *BAMLSS Prediction*

Description

Takes a fitted `bamlss` object and computes predictions. Predictions can be based on estimated parameters of optimizer functions or on samples returned from sampler functions. It is possible to compute predictions on multiple cores using the `parallel` and to chunk predictions to save computation time and memory storage. Predictions can be computed for full distributional parameters or specific model terms. If a `link{bamlss}` model was fitted on multiple cores, i.e., the samples are provided as `link{mcmc.list}` where each list entry represents samples from one core, function `predict.bamlss()` computes combined predictions based on samples of all cores.

Usage

```
## S3 method for class 'bamlss'
predict(object, newdata, model = NULL, term = NULL,
  match.names = TRUE, intercept = TRUE, type = c("link", "parameter"),
  FUN = function(x) { mean(x, na.rm = TRUE) }, trans = NULL,
  what = c("samples", "parameters"), nsamps = NULL,
  verbose = FALSE, drop = TRUE,
  cores = NULL, chunks = 1, ...)
```

Arguments

`object`	An object of class `"bamlss"`
`newdata`	A data frame or list containing the values of the model covariates at which predictions are required. Note that depending on argument `term`, only covariates that are needed by the corresponding model terms need to be supplied.
`model`	Character or integer, specifies the model for which predictions should be computed.

term	Character or integer, specifies the model terms for which predictions are required. Note that, e.g., term = c("s(x1)", "x2") will compute the combined prediction s(x1) + x2.
match.names	Should partial string matching be used to select the terms for prediction. Note that, e.g., term = "x1" will select all terms including "x1" if match.names = TRUE.
intercept	Should the intercept be included?
type	If type = "link" the predictor of the corresponding model is returned. If type = "parameter" predictions on the distributional parameter scale are returned.
FUN	A function that should be applied on the samples of predictors or parameters, depending on argument type.
trans	A transformer function or named list of transformer functions that computes transformed predictions. If trans is a list, the list names must match the names of the parameters of the bamlss.family.
what	Predictions can be computed from samples or estimated parameters of optimizer functions. If no samples are available the default is to use estimated parameters.
nsamps	If the fitted bamlss object contains samples of parameters, computing predictions may take quite some time. Therefore, to get a first feeling it can be useful to compute predictions only based on nsamps samples, i.e., nsamps specifies the number of samples which are extracted on equidistant intervals.
verbose	If predictions are chunked, information on the prediction process can be printed.
drop	If predictions for only one model are returned, the list structure is dropped.
cores	Specifies the number of cores that should be used for prediction. Note that this functionality is based on the parallel package.
chunks	Should computations be split into chunks? Prediction is then processed sequentially.
...	Arguments passed to prediction functions that are part of a bamlss.family object, i.e., the objects has a $predict() function that should be used instead.

Value

Depending on arguments model, FUN and the structure of the bamlss model, a list of predictions or simple vectors or matrices of predictions.

See Also

link{bamlss}, fitted.bamlss.

Examples

```
## Not run: ## Generate some data.
d <- GAMart()

## Model formula.
f <- list(
  num ~ s(x1) + s(x2) + s(x3) + te(lon,lat),
```

```
  sigma ~ s(x1) + s(x2) + s(x3) + te(lon,lat)
)

## Estimate model.
b <- bamlss(f, data = d)

## Predictions.
p <- predict(b)
str(b)

## Prediction for "mu" model and term "s(x2)".
p <- predict(b, model = "mu", term = "s(x2)")

## Plot effect
plot2d(p ~ x2, data = d)

## Same for "sigma" model.
p <- predict(b, model = "sigma", term = "s(x2)")
plot2d(p ~ x2, data = d)

## Prediction for "mu" model and term "s(x1)" + "s(x2)"
## without intercept.
p <- predict(b, model = "mu", term = c("s(x1)", "s(x2)"),
  intercept = FALSE)

## Prediction based on quantiles.
p <- predict(b, model = "mu", term = "s(x2)", FUN = c95)
plot2d(p ~ x2, data = d)

## Extract samples of predictor for "s(x2)".
p <- predict(b, model = "mu", term = "s(x2)",
  intercept = FALSE, FUN = function(x) { x })
print(dim(p))
plot2d(p ~ x2, data = d, col.lines = rgb(0.1, 0.1, 0.1, alpha = 0.1))

## Or using specific combinations of terms.
p <- predict(b, model = "mu", term = c("s(x2)", "te(lon,lat)"),
  intercept = FALSE, FUN = function(x) { x })
head(p)

## Prediction using new data.
## Only need x3 data when predicting
## for s(x3).
nd <- data.frame("x3" = seq(0, 1, length = 100))
nd <- cbind(nd, predict(b, newdata = nd, term = "s(x3)"))
print(head(nd))
plot2d(mu ~ x3, data = nd)
plot2d(sigma ~ x3, data = nd)

## End(Not run)
```

randomize *Transform Smooth Constructs to Random Effects*

Description

The transformer function takes a bamlss.frame object and transforms all smooth.constructs into a random effects representation. Note that this is only possible for smooth terms with a single smoothing variance. The function is based on function smooth2random.

Usage

```
trans_random(x)
randomize(x)
```

Arguments

x	Object returned from function bamlss.frame.

Details

The decomposition is achieved by a spectral decomposition of the penalty and design matrix by finding a basis of the null space of the penalty matrix. This feature is used, e.g., for the JAGS sampler function. For more details see also jagam.

Value

A transformed bamlss.frame. To each smooth.construct model term an element named "Xf", the fixed effects design matrix, and an element "rand$Xr", the random effects design matrix, is added. In addition, for re-transforming parameters elements "trans.U" and "trans.D" are supplied. See also function smooth2random.

References

Fahrmeir L, Kneib T, Lang S, Marx B (2013). Regression - Models, Methods and Applications. Springer-Verlag, Berlin. ISBN 978-3-642-34332-2.

Wood S.N. (2006). Generalized Additive Models: An Introduction with R. Chapman and Hall/CRC.

See Also

bamlss.frame, bamlss, smooth2random.

Examples

```
## Simulate data.
d <- GAMart()

## Create a "bamlss.frame".
bf <- bamlss.frame(num ~ s(x1) + s(x2) + s(x3) + s(lon,lat), data = d)
```

```
## Structure of the "s(x1)" smooth.construct.
str(bf$x$mu$smooth.construct[["s(x1)"]])

## Transform.
bf <- randomize(bf)

## New structure adding fixed
## and random effect matrices.
str(bf$x$mu$smooth.construct[["s(x1)"]])
```

---

rb *Random Bits for BAMLSS*

Description

This smooth constructor implements random bits model terms. Note that this is experimental.

Usage

```
## Linear smooth constructor.
rb(..., k = 50)

## For mgcv.
## S3 method for class 'randombits.smooth.spec'
smooth.construct(object, data, knots, ...)
```

Arguments

`...`	For function `rb()` a formula of the type `~x1+x2+x3` that specifies the covariates that should be modeled.
`k`	Integer, number of radnom bit columns in the design matrix.
`object`, `data`, `knots`	See `smooth.construct`.

Value

Function `rb()`, similar to function `s` a simple smooth specification object.

See Also

`bamlss`, `predict.bamlss`, `bfit`, `boost`

Examples

```
## Not run: ## Simulate data.
set.seed(123)
d <- GAMart()

## Estimate model.
f <- num ~ rb(x1) + rb(x2) + rb(x3) + rb(~lon+lat)

b <- bamlss(f, data = d)

plot(b)

## End(Not run)
```

`residuals.bamlss` *Compute BAMLSS Residuals*

Description

Function to compute quantile and response residuals.

Usage

```
## S3 method for class 'bamlss'
residuals(object, type = c("quantile", "response"),
  nsamps = NULL, ...)

## S3 method for class 'bamlss.residuals'
plot(x, which = c("hist-resid", "qq-resid", "wp"),
  spar = TRUE, ...)
```

Arguments

`object`	An object of class `"bamlss"`.
`type`	The type of residuals wanted, possible types are `"quantile"` residuals and `"response"` residuals.
`nsamps`	If the fitted `bamlss` object contains samples of parameters, computing residuals may take quite some time. Therefore, to get a first feeling it can be useful to compute residuals only based on `nsamps` samples, i.e., `nsamps` specifies the number of samples which are extracted on equidistant intervals.
`x`	Object returned from function `residuals.bamlss()`.
`which`	Should a histogram with kernel density estimates be plotted, a qq-plot or a worm plot?
`spar`	Should graphical parameters be set by the plotting function?
`...`	For function `residuals.bamlss()` arguments passed to possible `$residuals()` functions that may be part of a `bamlss.family`. For function `plot.bamlss.residuals()` arguments passed to function `hist.default` and `qqnorm.default`.

Details

Response residuals are the raw residuals, i.e., the response data minus the fitted distributional mean. If the `bamlss.family` object contains a function `$mu(par, ...)`, then raw residuals are computed with `y - mu(par)` where `par` is the named list of fitted values of distributional parameters. If `$mu(par, ...)` is missing, then the fitted values of the first distributional parameter are used.

Randomized quantile residuals are based on the cumulative distribution function of the `bamlss.family` object, i.e., the `$p(y, par, ...)` function.

Value

A vector of residuals.

References

Dunn P. K., and Smyth G. K. (1996). Randomized Quantile Residuals. *Journal of Computational and Graphical Statistics* **5**, 236–244.

van Buuren S., and Fredriks M. (2001) Worm Plot: Simple Diagnostic Device for Modelling Growth Reference Curves. *Statistics in Medicine*, **20**, 1259–1277

See Also

`bamlss`, `predict.bamlss`, `fitted.bamlss`.

Examples

```
## Not run: ## Generate data.
d <- GAMart()

## Estimate models.
b1 <- bamlss(num ~ s(x1), data = d)
b2 <- bamlss(num ~ s(x1) + s(x2) + s(x3), data = d)

## Extract quantile residuals.
e1 <- residuals(b1, type = "quantile")
e2 <- residuals(b2, type = "quantile")

## Plots.
plot(e1)
plot(e2)

## End(Not run)
```

response_name *Extract the reponse name of a* bamlss.frame *object.*

Description

This is a small helper function to quickly extract the response name(s) of an object of class "bamlss.frame" or "bamlss".

Usage

```
response_name(object, ...)
```

Arguments

object	An object of class "bamlss.frame" or "bamlss".
...	Not used.

See Also

bamlss, bamlss.frame

Examples

```
## Simulate some data.
d <- GAMart()

## Create a bamlss.frame.
bf <- bamlss.frame(num ~ s(x1) + s(x2) + s(x3), data = d)

## Extract the response name.
response_name(bf)
```

results.bamlss.default *Compute BAMLSS Results for Plotting and Summaries*

Description

The results function combines estimated parameters and/or samples with the bamlss.frame and computes the data that can be used, e.g., for creating effect plots or summary statistics. The function is usually used internally within bamlss. The object returned is of class "bamlss.results", which has a plotting method, see plot.bamlss.results.

Usage

```
results.bamlss.default(x, what = c("samples", "parameters"),
  grid = -1, nsamps = NULL, burnin = NULL, thin = NULL, ...)
```

Arguments

x	A bamlss.frame which has estimated parameters or samples. See also bfit and GMCMC.
what	Should the results data be prepared using estimated parameters or samples?
grid	Integer, sets the number of grid points for univariate functions to be used for creating results data, e.g., for plotting. This is more efficient when using data sets with a large number of unique covariate values. If negative suitable defaults are chosen.
nsamps	Integer, if results are computed using parameter samples, this argument controls the number of samples that should be used, e.g., if nsamps = 100 only 100 samples with equidistant intervals are selected. Basically similar to argument thin.
burnin	Integer, sets the number of samples that should be dropped from the beginning of the MCMC chain when creating results.
thin	Integer, should the MCMC chain be thinned additionally?
...	Currently not used.

Value

An object of class "bamlss.results".

See Also

plot.bamlss.results, bamlss.

Examples

```
## Not run: ## Simulate data.
d <- GAMart()

## Estimate model with no results.
b <- bamlss(num ~ s(x1) + s(x2) + s(x3),
  data = d, results = FALSE)

## Compute model results
a <- results.bamlss.default(b)

## Plot results for smooth terms.
plot(a)

## End(Not run)
```

rmf *Remove Special Characters*

Description

A simple helper function that removes special characters from a character string.

Usage

```
rmf(x)
```

Arguments

x	A character string.

Value

A character string with special characters removed.

Examples

```
rmf("ba*&m^l$$:s.s")
```

s2 *Special Smooths in BAMLSS Formulae*

Description

This is a simple wrapper function to define special smooth terms in BAMLSS formulae. The function calls the smooth term constructor function s. The return value of s is only slightly modified, such that function bamlss.frame identifies this term as a special term and uses the appropriate (internal) infrastructures.

This structure is useful when the model term structure is, e.g., not a linear combination of a design matrix and coefficients. See the example section of function bfit on how to use this setup.

Usage

```
s2(...)
```

Arguments

...	Arguments passed to function s.

Value

Slightly modified return value of function s.

See Also

bamlss, bamlss.frame, bamlss.formula, opt_bfit

Examples

```
print(names(s(x)))
print(names(s2(x)))
```

---

samples *Extract Samples*

Description

Generic function to extract samples from objects.

Usage

```
## Generic.
samples(object, ...)

## Method for "bamlss" objects.
## S3 method for class 'bamlss'
samples(object, model = NULL, term = NULL,
  combine = TRUE, drop = TRUE, burnin = NULL,
  thin = NULL, coef.only = FALSE, ...)
```

Arguments

object	An object for which samples should be extracted.
model	Character or integer, specifies the model for which samples should be extracted.
term	Character or integer, specifies the term for which samples should be extracted.
combine	Samples stored as a mcmc.list, e.g., when a model is estimated on multiple cores, can be combined into one large sample matrix.
drop	If there is only one model for which samples should be extracted, should the list structure be dropped?
burnin	Integer, specifies the number of samples that should be withdrawn as a burn-in phase.
thin	Integer, specifies the step length of samples that should be extracted, e.g., thin = 10 mean that only every 10th sample is returned.
coef.only	Logical, should only samples of model coefficients be returned?
...	Other arguments.

See Also

bamlss.

Examples

```
## Not run: ## Generate data.
d <- GAMart()

## Estimate model.
b <- bamlss(num ~ s(x1) + s(x2) + s(x3), data = d)

## Extract samples for "s(x2)".
sa <- samples(b, term = "s(x2)")
head(sa)

## Trace plot.
plot(sa)

## End(Not run)
```

`samplestats` *Sampling Statistics*

Description

The function computes the average the log-likelihood, log-posterior, the deviance information criterion and estimated degrees of freedom from samples of, e.g., a `bamlss` object.

Usage

```
samplestats(samples, x = NULL, y = NULL,
  family = NULL, logLik = FALSE, ...)
```

Arguments

`samples`	An object of class `"mcmc.list"` or `"bamlss"` which contains MCMC samples.
`x`	The `x` list as returned by function `bamlss.frame`.
`y`	The model response, as returned by function `bamlss.frame`
`family`	A `bamlss.family` object.
`logLik`	Logical, should the log-likelihood be computed, may take some time!
`...`	Currently not used.

Details

If the log-likelihood is not available in the samples, the function tries to compute the information. Depending on the complexity of the model, this may take some time. Computations are based on the `$d()` or `$loglik()` function of the `bamlss.family` object.

If a `bamlss.family` object contains a function `$p2d()` or `$p2loglik()`, which computes the log-likelihood from parameters, these functions are used for computation.

Value

A list with the following entries (if available):

logLik	The average log-likelihood.
logPost	The average log-posterior.
DIC	The deviance information criterion.
pd	The estimated degrees of freedom.

See Also

bamlss

Examples

```
## Not run: ## Generate some data.
d <- GAMart()

## Estimate model without sampling statistics
b <- bamlss(num ~ s(x1) + s(x2) + s(x3) + te(lon,lat),
  data = d, samplestats = FALSE)

## Note: needs the $d() or $loglik() function in the family!
names(family(b))

## Compute sampling statistics.
samplestats(b)

## End(Not run)
```

---

sam_BayesX *Markov Chain Monte Carlo for BAMLSS using* **BayesX**

Description

This sampler function for BAMLSS is an interface to the **BayesX** (https://www.uni-goettingen.de/de/bayesx/550513.
command-line binary from R. The sampler is based on the command line version and functions provided in the **BayesXsrc** package, which can be installed using function get_BayesXsrc().

Usage

```
## Sampler functions:
sam_BayesX(x, y, family, start = NULL, weights = NULL, offset = NULL,
  data = NULL, control = BayesX.control(...), ...)

BayesX(x, y, family, start = NULL, weights = NULL, offset = NULL,
  data = NULL, control = BayesX.control(...), ...)
```

```
## Sampler control:
BayesX.control(n.iter = 1200, thin = 1, burnin = 200,
  seed = NULL, predict = "light", model.name = "bamlss",
  data.name = "d", prg.name = NULL, dir = NULL,
  verbose = FALSE, show.prg = TRUE, modeonly = FALSE, ...)

## Special BayesX smooth term constructor.
sx(x, z = NULL, bs = "ps", by = NA, ...)

## Special BayesX tensor product smooth term constructor.
tx(..., bs = "ps", k = -1,
  ctr = c("center", "main", "both", "both1", "both2",
    "none", "meanf", "meanfd", "meansimple", "nullspace"),
  xt = NULL, special = TRUE)
tx2(...)
tx3(..., bs = "ps", k = c(10, 5),
  ctr = c("main", "center"),
  xt = NULL, special = TRUE)
tx4(..., ctr = c("center", "main", "both", "both1", "both2"))

## Smooth constructors and predict matrix.
## S3 method for class 'tensorX.smooth.spec'
smooth.construct(object, data, knots, ...)
## S3 method for class 'tensorX.smooth'
Predict.matrix(object, data)
## S3 method for class 'tensorX3.smooth.spec'
smooth.construct(object, data, knots, ...)
## S3 method for class 'tensorX3.smooth'
Predict.matrix(object, data)

## Family object for quantile regression with BayesX.
quant_bamlss(prob = 0.5)

## Download the newest version of BayesXsrc.
get_BayesXsrc(dir = NULL, install = TRUE)
```

Arguments

x	For function BayesX() the x list, as returned from function bamlss.frame, holding all model matrices and other information that is used for fitting the model. For function sx() arguments x and z specify the variables the smooth should be a function of.
y	The model response, as returned from function bamlss.frame.
z	Second variable in a sx() term.
family	A **bamlss** family object, see family.bamlss.
start	A named numeric vector containing possible starting values, the names are based on function parameters.

weights	Prior weights on the data, as returned from function bamlss.frame.
offset	Can be used to supply model offsets for use in fitting, returned from function bamlss.frame.
data	The model frame that should be used for modeling. Note that argument data needs not to be specified when the BayesX() sampler function is used with bamlss. For the smooth constructor for tx() terms, see function smooth.construct.
control	List of control arguments to be send to **BayesX**. See below.
n.iter	Sets the number of MCMC iterations.
thin	Defines the thinning parameter for MCMC simulation. E.g., thin = 10 means, that only every 10th sampled parameter will be stored.
burnin	Sets the burn-in phase of the sampler, i.e., the number of starting samples that should be removed.
seed	Sets the seed.
predict	Not supported at the moment, do not modify!
model.name	The name that should be used for the model when calling **BayesX**.
data.name	The name that should be used for the data set when calling **BayesX**.
prg.name	The name that should be used for the .prg file that is send to **BayesX**.
dir	Specifies the directory where **BayesX** should store all output files. For function get_BayesXsrc(), the directory where **BayesXsrc** should be stored.
verbose	Print information during runtime of the algorithm.
show.prg	Show the **BayesX** .prg file.
modeonly	Should only the posterior mode be compute, note that this is done using fixed smoothing parameters/variances.
bs	A character string, specifying the basis/type which is used for this model term.
by	A by variable for varying coefficient model terms.
k	The dimension(s) of the bases used to represent the tx() smooth term.
...	Not used in BayesX.control. For function sx() any extra arguments that should be passed to **BayesX** for this model term can be specified here. For function tx(), all variables the smooth should be a function of are specified here. For function sam_BayesX() all arguments that should be passed to BayesX.control.
ctr	Specifies the type of constraints that should be applied. "main", both main effects should be removed; "both", both main effects and varying effects should be removed; "none", no constraint should be applied.
xt	A list of extra arguments to be passed to **BayesX**.
special	Should the tx() model term be treated as a special smooth. This must be set to TRUE if using the sam_BayesX sampler and should be set to FALSE, e.g., when using the sam_GMCMC sampler.
object, knots	See, function smooth.construct.
prob	Numeric, specifies the quantile to be modeled, see the examples.
install	Should package **BayesXsrc** be installed?

Details

Function sam_BayesX() writes a **BayesX** .prg file and processes the data. Then, the function call the **BayesX** binary via function run.bayesx() of the **BayesXsrc** package. After the **BayesX** sampler has finished, the function reads back in all the parameter samples that can then be used for further processing within bamlss, i.a.

The smooth term constructor functions s and te can be used with the sam_BayesX() sampler. When using te note that only one smoothing variance is estimated by **BayesX**.

For anisotropic penalties use function tx() and tx3(), the former currently supports smooth functions of two variables, while tx3() is supposed to model space-time interactions. Note that in tx3() the first variable represents time and the 2nd and 3rd variable the coordinates in space.

Value

Function sam_BayesX() returns samples of parameters. The samples are provided as a mcmc matrix.

Function BayesX.control() returns a list with control arguments for **BayesX**.

Function sx() a list of class "xx.smooth.spec" and "no.mgcv", where "xx" is a basis/type identifying code given by the bs argument.

Function tx() and tx2() a list of class tensorX.smooth.spec.

Note

Note that this interface is still experimental and needs the newest version of the **BayesX** source code, which is not yet part of the **BayesXsrc** package on CRAN. The newest version can be installed with function get_BayesXsrc. Note that the function assumes that sh, subversion (svn) and R can be run from the command line!

Note that for setting up a new family object to be used with sam_BayesX() additional information needs to be supplied. The extra information must be placed within the family object in an named list element named "bayesx". For each parameter of the distribution a character string with the corresponding **BayesX** family name and the equationtype must be supplied. See, e.g., the R code of gaussian_bamlss how the setup works.

For function sx() the following basis types are currently supported:

- "ps": P-spline with second order difference penalty.
- "mrf": Markov random fields: Defines a Markov random field prior for a spatial covariate, where geographical information is provided by a map object in boundary or graph file format (see function read.bnd, read.gra and shp2bnd), as an additional argument named map.
- "re": Gaussian i.i.d. Random effects of a unit or cluster identification covariate.

Function tx() currently supports smooth terms with two variables.

See Also

bamlss, bamlss.frame

Examples

```
## Get newest version of BayesXsrc.
## Note: needs sh, svn and R build tools!
## get_BayesXsrc()
## Not run: if(require("BayesXsrc")) {
  ## Simulate some data
  set.seed(123)
  d <- GAMart()

  ## Estimate model with BayesX. Note
  ## that BayesX computes starting values, so
  ## these are not required by some optimizer function
  ## in bamlss()
  b1 <- bamlss(num ~ s(x1) + s(x2) + s(x3) + s(lon,lat),
    data = d, optimizer = FALSE, sampler = sam_BayesX)

  plot(b1)

  ## Same model with anisotropic penalty.
  b2 <- bamlss(num ~ s(x1) + s(x2) + s(x3) + tx(lon,lat),
    data = d, optimizer = FALSE, sampler = sam_BayesX)

  plot(b2)

  ## Quantile regression.
  b3_0.1 <- bamlss(num ~ s(x1) + s(x2) + s(x3) + tx(lon,lat),
    data = d, optimizer = FALSE, sampler = sam_BayesX,
    family = gF("quant", prob = 0.1))

  b3_0.9 <- bamlss(num ~ s(x1) + s(x2) + s(x3) + tx(lon,lat),
    data = d, optimizer = FALSE, sampler = sam_BayesX,
    family = gF("quant", prob = 0.9))

  ## Predict quantiles.
  p_0.1 <- predict(b3_0.1, term = "s(x2)")
  p_0.9 <- predict(b3_0.9, term = "s(x2)")

  ## Plot.
  plot2d(p_0.1 + p_0.9 ~ x2, data = d)
}

## End(Not run)
```

---

sam_Cox *Cox Model Markov Chain Monte Carlo*

Description

This sampler function implements a derivative based MCMC algorithm for flexible Cox models with structured additive predictors.

Usage

```
sam_Cox(x, y, family, start, weights, offset,
  n.iter = 1200, burnin = 200, thin = 1,
  verbose = TRUE, digits = 4, step = 20, ...)

cox_mcmc(x, y, family, start, weights, offset,
  n.iter = 1200, burnin = 200, thin = 1,
  verbose = TRUE, digits = 4, step = 20, ...)
```

Arguments

x	The x list, as returned from function bamlss.frame and transformed by function surv_transform, holding all model matrices and other information that is used for fitting the model.
y	The model response, as returned from function bamlss.frame.
family	A **bamlss** family object, see family.bamlss. In this case this is the cox_bamlss family object.
start	A named numeric vector containing possible starting values, the names are based on function parameters.
weights	Prior weights on the data, as returned from function bamlss.frame.
offset	Can be used to supply model offsets for use in fitting, returned from function bamlss.frame.
n.iter	Sets the number of MCMC iterations.
burnin	Sets the burn-in phase of the sampler, i.e., the number of starting samples that should be removed.
thin	Defines the thinning parameter for MCMC simulation. E.g., thin = 10 means, that only every 10th sampled parameter will be stored.
verbose	Print information during runtime of the algorithm.
digits	Set the digits for printing when verbose = TRUE.
step	How many times should algorithm runtime information be printed, divides n.iter.
...	Currently not used.

Details

The sampler uses derivative based proposal functions to create samples of parameters. For time-dependent functions the proposals are based on one Newton-Raphson iteration centered at the last state, while for the time-constant functions proposals can be based on iteratively reweighted least squares (IWLS), see also function GMCMC. The integrals that are part of the time-dependent function updates are solved numerically. In addition, smoothing variances are sampled using slice sampling.

Value

The function returns samples of parameters. The samples are provided as a mcmc matrix.

References

Umlauf N, Klein N, Zeileis A (2016). Bayesian Additive Models for Location Scale and Shape (and Beyond). *(to appear)*

See Also

opt_Cox, cox_bamlss, surv_transform, simSurv, bamlss

Examples

```
## Not run: library("survival")
set.seed(123)

## Simulate survival data.
d <- simSurv(n = 500)

## Formula of the survival model, note
## that the baseline is given in the first formula by s(time).
f <- list(
  Surv(time, event) ~ s(time) + s(time, by = x3),
  gamma ~ s(x1) + s(x2)
)

## Cox model with continuous time.
## Note the the family object cox_bamlss() sets
## the default optimizer and sampler function!
## First, posterior mode estimates are computed
## using function opt_Cox(), afterwards the
## sampler sam_Cox() is started.
b <- bamlss(f, family = "cox", data = d)

## Plot estimated effects.
plot(b)

## End(Not run)
```

---

sam_GMCMC *General Markov Chain Monte Carlo for BAMLSS*

Description

These functions provide a quite general infrastructure for sampling BAMLSS. The default proposal function is based on iteratively weighted least squares (IWLS), however, each model term may have a different updating function, see the details.

Usage

```
## Sampler functions:
sam_GMCMC(x, y, family, start = NULL, weights = NULL, offset = NULL,
  n.iter = 1200, burnin = 200, thin = 1, verbose = TRUE,
  step = 20, propose = "iwlsC_gp", chains = NULL, ...)

GMCMC(x, y, family, start = NULL, weights = NULL, offset = NULL,
  n.iter = 1200, burnin = 200, thin = 1, verbose = TRUE,
  step = 20, propose = "iwlsC_gp", chains = NULL, ...)

## Propose functions:
GMCMC_iwls(family, theta, id, eta, y, data,
  weights = NULL, offset = NULL, ...)
GMCMC_iwlsC(family, theta, id, eta, y, data,
  weights = NULL, offset = NULL, zworking, resids, rho, ...)
GMCMC_iwlsC_gp(family, theta, id, eta, y, data,
  weights = NULL, offset = NULL, zworking, resids, rho, ...)
GMCMC_slice(family, theta, id, eta, y, data, ...)
```

Arguments

x	For function bfit() the x list, as returned from function bamlss.frame, holding all model matrices and other information that is used for fitting the model. For the updating functions an object as returned from function smooth.construct or smoothCon.
y	The model response, as returned from function bamlss.frame.
family	A **bamlss** family object, see family.bamlss.
start	A named numeric vector containing possible starting values, the names are based on function parameters.
weights	Prior weights on the data, as returned from function bamlss.frame.
offset	Can be used to supply model offsets for use in fitting, returned from function bamlss.frame.
n.iter	Sets the number of MCMC iterations.
burnin	Sets the burn-in phase of the sampler, i.e., the number of starting samples that should be removed.
thin	Defines the thinning parameter for MCMC simulation. E.g., thin = 10 means, that only every 10th sampled parameter will be stored.
verbose	Print information during runtime of the algorithm.
step	How many times should algorithm runtime information be printed, divides n.iter.
propose	Sets the propose function for model terms, e.g. for a term s(x) in the model formula. Per default this is set to "iwlsC", a character pointing to the set of propose functions, see above. Other options are "iwls" and "slice", however, this is more experimental and should not be set by the user. Another option is to pass a full propose function which should be used for each model term, the structure of propose functions is described in the details below. Model terms may also have different propose functions, see the example section.

chains	How many chains should be started? Chains a sampled sequentially!
theta	The current state of parameters, provided as a named list. The first level represents the parameters of the distribution, the second level the parameters of the model terms. E.g., using the gaussian_bamlss family object theta[["mu"]][["s(x)"]] extracts the current state of a model term "s(x)" of the "mu" parameter. Extraction is done with the id argument.
id	The parameter identifier, a character vector of length 2. The first character specifies the current distributional parameter, the second the current model term.
eta	The current value of the predictors, provided as a named list, one list entry for each parameter. The names correspond to the parameter names in the family object, see family.bamlss. E.g., when using the gaussian_bamlss family object, the current values for the mean can be extracted by eta\$mu and for the standard deviation by eta\$sigma.
data	An object as returned from function smooth.construct or smoothCon. The object is preprocessed by function bamlss.engine.setup.
zworking	Preinitialized numeric vector of length(y), only for internal usage.
resids	Preinitialized numeric vector of length(y), only for internal usage.
rho	An environment, only for internal usage.
...	Arguments passed to function bamlss.engine.setup and to the propose functions.

Details

The sampler function sam_GMCMC() cycles through all distributional parameters and corresponding model terms in each iteration of the MCMC chain. Samples of the parameters of a model term (e.g., s(x)) are generated by proposal functions, e.g. GMCMC_iwls().

The default proposal function that should be used for all model terms is set with argument propose. For smooth terms, e.g. terms created with function s, if a valid propose function is supplied within the extra xt list, this propose function will be used. This way each model term may have its own propose function for creating samples of the parameters. See the example section.

The default proposal function GMCMC_iwlsC_gp allows for general priors for the smoothing variances and general penalty functions. Samples of smoothing variances are computed using slice sampling. Function GMCMC_iwlsC samples smoothing variances of univariate terms assuming an inverse gamma prior. Terms of higher dimensions use again slice sampling for creating samples of smoothing variances.

Function GMCMC_iwls is similar to function GMCMC_iwlsC but uses plain R code.

Function GMCMC_slice applies slice sampling also for the regression coefficients and is therefore relatively slow.

Value

The function returns samples of parameters, depending on the return value of the propose functions other quantities can be returned. The samples are provided as a mcmc matrix. If chains > 1, the samples are provided as a mcmc.list.

References

Umlauf N, Klein N, Zeileis A (2016). Bayesian Additive Models for Location Scale and Shape (and Beyond). *(to appear)*

See Also

bamlss, bamlss.frame, bamlss.engine.setup, set.starting.values, s2

Examples

```
## Not run: ## Simulated data example illustrating
## how to call the sampler function.
## This is done internally within
## the setup of function bamlss().
d <- GAMart()
f <- num ~ s(x1, bs = "ps")
bf <- bamlss.frame(f, data = d, family = "gaussian")

## First, find starting values with optimizer.
opt <- with(bf, bfit(x, y, family))

## Sample.
samps <- with(bf, sam_GMCMC(x, y, family, start = opt$parameters))
plot(samps)

## End(Not run)
```

---

sam_JAGS *Markov Chain Monte Carlo for BAMLSS using JAGS*

Description

This sampler function for BAMLSS is an interface to the JAGS library using package rjags. The function basically interprets the bamlss.frame into BUGS code, similar to the jagam function of package mgcv. I.e., the function uses the random effects representation of smooth terms, see the transformer function randomize to generate the BUGS code.

Note that estimating BAMLSS with JAGS is not very efficient. Also note that this function is more experimental and support is only provided for a small number of bamlss.family objects.

Function BUGSeta() therefore computes the code and data for one parameter of the modeled distribution. Function BUGSmodel() then collects all parameter model code and data, which can be send to JAGS.

Usage

```
## Sampler functions:
sam_JAGS(x, y, family, start = NULL,
  tdir = NULL, n.chains = 1, n.adapt = 100,
```

```
  n.iter = 4000, thin = 2, burnin = 1000,
  seed = NULL, verbose = TRUE, set.inits = TRUE,
  save.all = FALSE, modules = NULL, ...)

JAGS(x, y, family, start = NULL,
  tdir = NULL, n.chains = 1, n.adapt = 100,
  n.iter = 4000, thin = 2, burnin = 1000,
  seed = NULL, verbose = TRUE, set.inits = TRUE,
  save.all = FALSE, modules = NULL, ...)

## Function to interpret an additive predictor into BUGS code:
BUGSeta(x, id = NULL, ...)

## Function to interpret the full BAMLSS:
BUGSmodel(x, family, is.stan = FALSE, reference = NULL, ...)
```

Arguments

x	For function sam_JAGS() and BUGSmodel() the x list, as returned from function bamlss.frame, holding all model matrices and other information that is used for fitting the model. For function BUGSeta() argument x is one element of the x object, i.e., one parameter.
y	The model response, as returned from function bamlss.frame.
family	A **bamlss** family object, see family.bamlss.
start	A named numeric vector containing possible starting values, the names are based on function parameters.
tdir	The path to the temporary directory that should be used.
n.chains	Specifies the number of sequential MCMC chains that should be run with JAGS.
n.adapt	Specifies the number of iterations that should be used as an initial adaptive phase.
n.iter	Sets the number of MCMC iterations.
thin	Defines the thinning parameter for MCMC simulation. E.g., thin = 10 means, that only every 10th sampled parameter will be stored.
burnin	Sets the burn-in phase of the sampler, i.e., the number of starting samples that should be removed.
seed	Sets the seed.
verbose	Print information during runtime of the algorithm.
set.inits	Should initial values of BAMLSS parameters be provided to JAGS, if available. Set in argument start.
save.all	Should all JAGS files be saved in tdir.
modules	Specify additional modules that should be loaded, see function load.module.
id	Character, the current parameter name for which the BUGS code should be produced.

is.stan	Should the BUGS code be translated to STAN code. Note that this is only experimental.
reference	A character specifying a reference category, e.g., when fitting a multinomial model.
...	Currently not used.

Value

Function sam_JAGS() returns samples of parameters. The samples are provided as a mcmc matrix. If n.chains > 1, the samples are provided as a mcmc.list.

Function BUGSeta() returns the BUGS model code and preprocessed data for one additive predictor. Function BUGSmodel() then combines all single BUGS code chunks and the data and creates the final BUGS model code that can be send to JAGS.

Note

Note that for setting up a new family object to be used with sam_JAGS() additional information needs to be supplied. The extra information must be placed within the family object in an element named "bugs". The following entries should be supplied within the ..$bugs list:

- "dist". The name of the distribution in BUGS/JAGS model language.
- "eta". The function that computes the BUGS code for one structured additive predictor. Function BUGSeta() is used per default.
- "model". The function that merges all single predictor BUGS model code and data. The default function is BUGSmodel().
- "reparam". A named vector of character strings that specify a re-parametrization.

See also the example code of family.bamlss.

See Also

bamlss, bamlss.frame, bamlss.engine.setup, set.starting.values, bfit, GMCMC

Examples

```
## Not run: ## Simulated data example illustrating
## how to call the sampler function.
## This is done internally within
## the setup of function bamlss().
d <- GAMart()
f <- num ~ s(x1, bs = "ps")
bf <- bamlss.frame(f, data = d, family = "gaussian")

## First, find starting values with optimizer.
opt <- with(bf, opt_bfit(x, y, family))

## Sample with JAGS.
if(require("rjags")) {
  samps <- with(bf, sam_JAGS(x, y, family, start = opt$parameters))
```

```
  plot(samps)

  b <- bamlss(f, data = d, family = "gaussian", sampler = sam_JAGS)
  plot(b)
}

## End(Not run)
```

---

sam_MVNORM *Create Samples for BAMLSS by Multivariate Normal Approximation*

Description

This sampler function for BAMLSS uses estimated parameters and the Hessian information to create samples from a multivariate normal distribution. Note that smoothing variance uncertainty is not accounted for, therefore, the resulting credible intervals are most likely too narrow.

Usage

```
sam_MVNORM(x, y = NULL, family = NULL, start = NULL,
  n.samples = 500, hessian = NULL, ...)

MVNORM(x, y = NULL, family = NULL, start = NULL,
  n.samples = 500, hessian = NULL, ...)
```

Arguments

x	The x list, as returned from function bamlss.frame, holding all model matrices and other information that is used for fitting the model. Or an object returned from function bamlss.
y	The model response, as returned from function bamlss.frame.
family	A **bamlss** family object, see family.bamlss.
start	A named numeric vector containing possible starting values, the names are based on function parameters.
n.samples	Sets the number of samples that should be generated.
hessian	The Hessian matrix that should be used. Note that the row and column names must be the same as the names of the parameters. If hessian = NULL the function uses optim to compute the Hessian if it is not provided within x.
...	Arguments passed to function optim.

Value

Function MVNORM() returns samples of parameters. The samples are provided as a mcmc matrix.

See Also

bamlss, bamlss.frame, bamlss.engine.setup, set.starting.values, opt_bfit, sam_GMCMC

Examples

```
## Simulated data example illustrating
## how to call the sampler function.
## This is done internally within
## the setup of function bamlss().
d <- GAMart()
f <- num ~ s(x1, bs = "ps")
bf <- bamlss.frame(f, data = d, family = "gaussian")

## First, find starting values with optimizer.
o <- with(bf, opt_bfit(x, y, family))

## Sample.
samps <- with(bf, sam_MVNORM(x, y, family, start = o$parameters))
plot(samps)
```

`scale2` *Scaling Vectors and Matrices*

Description

The function scales numeric objects to specific ranges.

Usage

```
scale2(x, lower = -1.5, upper = 1.5)
```

Arguments

`x`	Numeric, vector or matrix.
`lower`	The upper range.
`upper`	The lower range.

Value

A scaled numeric vector or matrix, scaled to the range provided in `lower` and `upper`.

Examples

```
set.seed(123)
x <- runif(5)
scale2(x, -1, 1)
scale2(x, 0, 10)
```

simdata *Reference data.*

Description

Simulated data to test the implementation of the bamlss families.

Usage

```
data("simdata")
```

Format

An object of class `list` of length 3.

See Also

mvnchol_bamlss

Examples

```
## Not run: ## Reproducing code.
set.seed(111)
n <- 2000

## build orthogonal rotation matrix
thetax <- pi/4
thetay <- pi/4
thetaz <- pi/4
Rx <- matrix( c(1,0,0, 0,cos(thetax),sin(thetax), 0,-sin(thetax),cos(thetax) ), 3, 3 )
Ry <- matrix( c(cos(thetay),0,-sin(thetay), 0,1,0,  sin(thetay),0,cos(thetay) ), 3, 3 )
Rz <- matrix( c(cos(thetaz),sin(thetaz),0, -sin(thetaz),cos(thetaz),0, 0,0,1 ), 3, 3 )
R <- Rx %*% Ry %*% Rz

## non-linear functions
f1 <- function(x) (sin(pi * x))^2
f2 <- function(x) (cos(pi * x))^2

## random derivitates
x <- runif(n)

## eigenvalues
val1 <- f1(x)
val2 <- f2(x)
val3 <- rep(0, n)

## initialize vectors for parameter lists
p12 <- NULL
p13 <- NULL
p23 <- NULL
```

```
sig <- matrix(0, n, 3)

lamdiag <- matrix(0, n, 3)
lambda <- matrix(0, n, 3)

y <- matrix(0, n, 3)
log_dens_ref <- rep(0, n)

tau <- .1  ## offset on diagonal
l <- 0     ## count occasions with invertible cv
dens1 <- NULL
for ( ii in seq(n) ) {
    mu <- rep(0, 3)

    val <- diag( c(val1[ii], val2[ii], val3[ii]) ) + diag(tau, 3)
    ## compute covariance matrix from rotation matrix and eigenvalues
    cv <- R %*% val %*% t(R)

    ## compute parameters for parameter list
    sig[ii,] <- sqrt(diag(cv))
    p12[ii] <- cv[1,2]
    p13[ii] <- cv[1,3]
    p23[ii] <- cv[2,3]

    ## compute paramters for Cholesky family
    chol_cv <- solve(chol(cv))  # lambdas come from L^-1 not L
    lamdiag[ii,] <- diag(chol_cv)
    lambda[ii,] <- chol_cv[upper.tri(chol_cv)]

    ## Check if cv is invertible
    if ( !is.matrix(try(chol(cv))) ) l <- l + 1

    y[ii,] <- mvtnorm::rmvnorm(1, mu, cv)

    log_dens_ref[ii] <- mvtnorm::dmvnorm(y[ii,], mu, cv, log = TRUE)
}
print(l)

## Data
d <- as.data.frame(y)
names(d) <- paste0("y", 1:3)
d$x <- x

## make parameter list for mvn chol family
par <- list()
par[["mu1"]] <- rep(0,n)
par[["mu2"]] <- rep(0,n)
par[["mu3"]] <- rep(0,n)
par[["lamdiag1"]] <- lamdiag[,1]
par[["lamdiag2"]] <- lamdiag[,2]
par[["lamdiag3"]] <- lamdiag[,3]
par[["lambda12"]] <- lambda[,1]
par[["lambda13"]] <- lambda[,2]
```

```
par[["lambda23"]] <- lambda[,3]

simdata <- list(
    d   = d,
    par = par,
    y   = y
)

## save(simdata, file = "simdata.rda")
## End of simulation

## End(Not run)
```

---

simJM *Simulate longitudinal and survival data for joint models*

Description

Simulates longitudinal data with normal error and (Cox-type) survival times using the inversion method. The function `simJM()` is a wrapper specifying all predictors and the resulting data sets. The wrapper calls `rJM()` to sample the survival times, a modified version of `rSurvtime()` from the R package **CoxFlexBoost**.

Usage

```
simJM(nsub = 300, times = seq(0, 120, 1), probmiss = 0.75,
  long_setting = "functional",
  alpha_setting = if(nonlinear) "linear" else "nonlinear",
  dalpha_setting = "zero", sigma = 0.3, long_df = 6, tmax = NULL,
  seed = NULL, full = FALSE, file = NULL, nonlinear = FALSE,
  fac = FALSE)

rJM(hazard, censoring, x, r,
  subdivisions = 1000, tmin = 0, tmax,
  file = NULL, ...)
```

Arguments

`nsub`	number of individuals for which longitudinal data and survival times should be simulated.
`times`	vector of time points at which longitudinal measurements are "sampled".
`probmiss`	proportion of longitudinal measurements to be set to missing. Used to induce sparsity in the longitudinal measurements.
`long_setting`	Specification of the longitudinal trajectories of the sampled subjects. Preset specifications are `"linear"`, `"nonlinear"` and `"functional"`. See Details.
`alpha_setting`	specification of the association between survival and longitudinal. Preset specifications are `"simple"`, `"linear"`, `"nonlinear"` and `"nonlinear2"`. See Details.

dalpha_setting	specification of the association between survival and the derivative of the longitudinal. Work in progress.
sigma	standard deviation of the normal error around the true longitudinal measurements.
long_df	number of basis functions from which functional random intercepts are sampled.
tmax	For function simJM(), longest possible survival time, observations are censored after that timepoint. Defaults to max(times) and should not be specified longer than max(times) for longitudinal setting "functional". For function rJM(), latest time point to sample a survival time.
seed	numeric scalar setting the random seed.
full	logical indicating if only the longitudinal data set should be returned (FALSE) or additionally also the data for the survival part evaluated on a regular time grid and the longitudinal data set without longitudinal missings (TRUE).
file	name of the data file the generated data set should be stored into (e.g., "simdata.RData") or NULL if the dataset should directly be returned in R.
nonlinear	If set to TRUE, a nonlinear interaction between alpha and mu is simulated.
fac	If set to TRUE, a smooth interaction that varies by a factor is simulated.
hazard	complete hazard function to specify the joint model. Time must be the first argument.
censoring	function to compute (random) censoring.
x	matrix of sampled covariate values.
r	matrix of sampled random coefficients.
subdivisions	the maximum number of subintervals for the integration.
tmin	earliest time point to sample a survival time.
...	further arguments to be passed to hazard or censoring.

Details

The function simulates longitudinal data basing on the given specification at given times. The full hazard is built from all joint model predictors η_μ, η_σ, η_λ, η_γ, η_α as presented in Koehler, Umlauf, and Greven (2016), see also jm_bamlss. Survival times are sampled using the inversion method (cf. Bender, Augustin, & Blettner, 2005). Additional censoring and missingness is introduced. The longitudinal information is censored according to the survival information. The user can also specify own predictors and use only rJM to simulate survival times accordingly.

Pre-specified functions for η_μ in long_setting are for linear

$$\eta_{\mu i}(t) = 1.25 + r_{1i} + 0.6\sin(x_{2i}) + (-0.01)t + 0.02 r_{2i} t$$

, for nonlinear

$$\eta_{\mu i}(t) = 0.5 + r_{1i} + 0.6\sin(x_{2i}) + 0.1(t+1)\exp(-0.075t)$$

and for functional

$$\eta_{\mu i}(t) = 0.5 + r_{1i} + 0.6\sin(x_{2i}) + 0.1(t+1)\exp(-0.075t) + \sum_k \beta_{ki} B(t)$$

, where $B(.)$ denotes a B-spline basis function and β_{ki} are the sampled penalized coefficients from gen_b per person.

Prespecified functions for η_α in alpha_setting are for constant

$$\eta_\alpha(t) = 1$$

, for linear

$$\eta_\alpha(t) = 1 - 0.015t$$

, for nonlinear

$$\eta_\alpha(t) = \cos((time - 20)/20)$$

, and for nonlinear

$$\eta_\alpha(t) = \cos((time - 33)/33)$$

.

Additionally the fixed functions for $\eta_\lambda = 0.1(t+2)\exp(-0.075t)$ and $\eta_\lambda = 0.1(t+2)\exp(-0.075t)$ are employed.

Value

For full = TRUE a list of the three data.frames is returned:

data	Simulated dataset in long format including all longitudinal and survival covariates.
data_grid	Dataset of the time-varying survival predictors which are not subject specific, evaluated at a grid of fixed time points.
data_full	Simulated data set prior to generating longitudinal missings. Useful to assess the longitudinal fit.

For full = FALSE only the first dataset is returned.

Covariates within these datasets include a subject identifier id, the sampled survival times survtime, the event indicator event, the time points of longitudinally "observed" measurements obstime, the longitudinal response y, the cumulative hazard at the survival time cumhaz, as well as covariates x1, x2, random effects r1, r2, b1, ..., and the true predictors alpha, lambda, gamma, mu, sigma.

References

Hofner, B (2016). **CoxFlexBoost**: Boosting Flexible Cox Models (with Time-Varying Effects). R package version 0.7-0.

Bender, R., Augustin, T., and Blettner, M. (2005). Generating Survival Times to Simulate Cox Proportional Hazards Models. *Statistics in Medicine*, **24**, 1713-1723.

Koehler N, Umlauf N, Beyerlein, A., Winkler, C., Ziegler, A., and Greven S (2016). Flexible Bayesian Additive Joint Models with an Application to Type 1 Diabetes Research. *(submitted)*

See Also

jm_bamlss, opt_JM, sam_JM, bamlss.

Examples

```
## Not run: ## Simulate survival data
## with functional random intercepts and a nonlinear effect
## of time, time-varying association alpha.
d <- simJM(nsub = 300)
head(d)

## Simulate survival data
## with random intercepts/slopes and a linear effect of time,
## constant association alpha.
d <- simJM(nsub = 200, long_setting = "linear",
  alpha_setting = "constant")
head(d)

## End(Not run)
```

`simSurv` *Simulate Survival Times*

Description

Function `simSurv()` and `rSurvtime2()` simulate arbitrary (Cox-type) survival times using the inversion method. Function `simSurv()` is a simple wrapper that calls `rSurvtime2()`. The functions are based on the R package **CoxFlexBoost** implementation `rSurvtime()` and only slightly modify the code.

Usage

```
## Simulate a pre-specified survival times data set.
simSurv(n = 300)

## Simulate arbitrary survival times.
rSurvTime2(lambda, x, cens_fct, upper = 1000, ...,
  file = NULL, subdivisions = 1000)
```

Arguments

`n`	The number of individuals for which survival times should be simulated.
`lambda`	function. Baseline hazard $\lambda(t, x)$ where time must be first argument.
`x`	matrix. (Sampled) values for covariates (without time).
`cens_fct`	function. Function to compute (random) censoring.
`upper`	upper boundary of the interval the random survival times fall into.
`...`	further arguments to be passed to `lambda` or `cens_fct`.
`file`	character. name of the data file the generated data set should be stored into (e.g., "survtimes.RData") or NULL if the dataset should directly be returned in R.
`subdivisions`	The maximum number of subintervals for the integration.

Details

This is basically a slight modification according the computation of the integral, see the manual page of function `rSurvtime()` of package `CoxFlexBoost` for details.

Value

A `data.frame` consisting of the observed survival time (`time`), the non-censoring indicator (`event`) and further covariates `x` is returned. If `file` is specified, the data.frame is additionally stored on the disc.

References

Benjamin Hofner (2016). **CoxFlexBoost**: Boosting Flexible Cox Models (with Time-Varying Effects). R package version 0.7-0.

Ralph Bender and Thomas Augustin and Maria Blettner (2005), Generating Survival Times to Simulate Cox Proportional Hazards Models. *Statistics in Medicine*, **24**, 1713-1723.

See Also

`cox_bamlss`, `opt_Cox`, `sam_Cox`, `bamlss`

Examples

```
## The following shows the code of the
## wrapper function simSurv().
set.seed(111)
n <- 100
X <- matrix(NA, nrow = n, ncol = 3)
X[, 1] <- runif(n, -1, 1)
X[, 2] <- runif(n, -3, 3)
X[, 3] <- runif(n, -1, 1)

## Specify censoring function.
cens_fct <- function(time, mean_cens) {
  ## Censoring times are independent exponentially distributed.
  censor_time <- rexp(n = length(time), rate = 1 / mean_cens)
  event <- (time <= censor_time)
  t_obs <- apply(cbind(time, censor_time), 1, min)
  ## Return matrix of observed survival times and event indicator.
  return(cbind(t_obs, event))
}

## log(time) is the baseline hazard.
lambda <-  function(time, x) {
  exp(log(time) + 0.7 * x[1] + sin(x[2]) + sin(time * 2) * x[3])
}

## Simulate data with lambda() and cens_fct().
d <- rSurvTime2(lambda, X, cens_fct, mean_cens = 5)
```

sliceplot *Plot Slices of Bivariate Functions*

Description

This function plots slices from user defined values of bivariate surfaces.

Usage

```
sliceplot(x, y = NULL, z = NULL, view = 1, c.select = NULL,
  values = NULL, probs = c(0.1, 0.5, 0.9), grid = 100,
  legend = TRUE, pos = "topright", digits = 2, data = NULL,
  rawdata = FALSE, type = "mba", linear = FALSE,
  extrap = FALSE, k = 40, rug = TRUE, rug.col = NULL,
  jitter = TRUE, ...)
```

Arguments

x	A matrix or data frame, containing the covariates for which the effect should be plotted in the first and second column and at least a third column containing the effect. Another possibility is to specify the plot via a formula, e.g., for simple plotting of bivariate surfaces z ~ x + y, see the examples.
y	If x is a vector the argument y and z must also be supplied as vectors.
z	If x is a vector the argument y and z must also be supplied as vectors, z defines the surface given by $z = f(x, y)$.
view	Which variable should be used for the x-axis of the plot, the other variable will be used to compute the slices. May also be a character with the name of the corresponding variable.
c.select	Integer, selects the column that is used in the resulting matrix to be used as the z argument.
values	The values of the x or y variable that should be used for computing the slices, if set to NULL, slices will be constructed according to the quantiles, see also argument probs.
probs	Numeric vector of probabilities with values in [0,1] to be used within function quantile to compute the values for plotting the slices.
grid	The grid size of the surface where the slices are generated from.
legend	If set to TRUE, a legend with the values that where used for slicing will be added.
pos	The position of the legend, see also function legend.
digits	The decimal place the legend values should be rounded.
data	If x is a formula, a data.frame or list. By default the variables are taken from environment(x): typically the environment from which plot3d is called.
rawdata	If set to TRUE, the data will not be interpolated, only raw data will be used. This is useful when displaying data on a regular grid.

type	Character, which type of interpolation method should be used. The default is type = "akima", see function interp. The two other options are type = "mba", which calls function mba.surf of package **MBA**, or type = "mgcv", which uses a spatial smoother withing package **mgcv** for interpolation. The last option is definitely the slowest, since a full regression model needs to be estimated.
linear	Logical, should linear interpolation be used withing function interp?
extrap	Logical, should interpolations be computed outside the observation area (i.e., extrapolated)?
k	Integer, the number of basis functions to be used to compute the interpolated surface when type = "mgcv".
rug	Add a rug to the plot.
jitter	If set to TRUE a jittered rug plot is added.
rug.col	Specify the color of the rug representation.
...	Parameters passed to matplot and legend.

Details

Similar to function plot3d, this function first applies bivariate interpolation on a regular grid, afterwards the slices are computed from the resulting surface.

Note

Function sliceplot can use the **akima** package to construct smooth interpolated surfaces, therefore, package **akima** needs to be installed. The **akima** package has an ACM license that restricts applications to non-commercial usage, see

https://www.acm.org/publications/policies/software-copyright-notice

Function sliceplot prints a note referring to the ACM license. This note can be suppressed by setting

```
options("use.akima" = TRUE)
```

See Also

plot2d, plot3d, plotmap, plotblock.

Examples

```
## Generate some data.
set.seed(111)
n <- 500

## Regressors.
d <- data.frame(z = runif(n, -3, 3), w = runif(n, 0, 6))

## Response.
d$y <- with(d, 1.5 + cos(z) * sin(w) + rnorm(n, sd = 0.6))

## Not run: ## Estimate model.
```

```
b <- bamlss(y ~ te(z, w), data = d)
summary(b)

## Plot estimated effect.
plot(b, term = "te(z,w)", sliceplot = TRUE)
plot(b, term = "te(z,w)", sliceplot = TRUE, view = 2)
plot(b, term = "te(z,w)", sliceplot = TRUE, view = "w")
plot(b, term = "te(z,w)", sliceplot = TRUE, probs = seq(0, 1, length = 10))

## End(Not run)

## Variations.
d$f1 <- with(d, sin(z) * cos(w))
sliceplot(cbind(z = d$z, w = d$w, f1 = d$f1))

## Same with formula.
sliceplot(sin(z) * cos(w) ~ z + w, ylab = "f(z)", data = d)

## Compare with plot3d().
plot3d(sin(z) * 1.5 * w ~ z + w, zlab = "f(z,w)", data = d)
sliceplot(sin(z) * 1.5 * w ~ z + w, ylab = "f(z)", data = d)
sliceplot(sin(z) * 1.5 * w ~ z + w, view = 2, ylab = "f(z)", data = d)
```

`smooth.construct` *Constructor Functions for Smooth Terms in BAMLSS*

Description

The generic function is only a copy of `smooth.construct` adding a `...` argument. For objects of class `"bamlss.frame"` and `"bamlss"` the method extracts all smooth model terms, see function `bamlss.frame` for details on the setup of BAMLSS.

Usage

```
## Function as in package mgcv
## but with additional dots argument.
smooth.construct(object, data, knots, ...)

## For 'bamlss.frame's.
## S3 method for class 'bamlss.frame'
smooth.construct(object, data = NULL, knots = NULL,
  model = NULL, drop = TRUE, ...)

## S3 method for class 'bamlss.formula'
smooth.construct(object, data = NULL, knots = NULL,
  model = NULL, drop = TRUE, ...)

## S3 method for class 'bamlss.terms'
smooth.construct(object, data = NULL, knots = NULL,
  model = NULL, drop = TRUE, ...)
```

Arguments

object	Either a smooth specification object, or object of class "bamlss", "bamlss.frame", "bamlss.formula" or "bamlss.terms". For smooth specification objects, see function smooth.construct.
data	A data frame or list, see also see function smooth.construct.
knots	See function smooth.construct.
model	Character, specifies for which model parameter the smooth constructs should be created.
drop	If there is only one model parameter the returned named list is simplified.
...	Arguments passed to the smooth term constructor functions.

Value

For smooth specification objects see function see smooth.construct. For objects of class "bamlss.frame" or "bamlss" the list of smooth constructs, see function bamlss.frame for more details.

See Also

bamlss.frame, bamlss.formula, bamlss, smooth.construct.

Examples

```
## Generate some data.
d <- GAMart()

## Create a "bamlss.frame".
bf <- bamlss.frame(num ~ s(x1) + s(x2), data = d)

## Extract the smooth construct.
sc <- smooth.construct(bf)
str(sc)

## Also possible with formulas.
f <- bamlss.formula(list(
  num ~ s(x1) + te(lon,lat),
  sigma ~ s(x2)
), family = "gaussian")

sc <- smooth.construct(f, data = d)
str(sc)
```

`smooth.construct.kr.smooth.spec` *Kriging Smooth Constructor*

Description

This smooth constructor implements a kriging based model term.

Usage

```
## S3 method for class 'kr.smooth.spec'
smooth.construct(object, data, knots, ...)
## S3 method for class 'kriging.smooth'
Predict.matrix(object, data)
```

Arguments

`object`, `data`, `knots`	See `smooth.construct`.
`...`	Currently not used.

Details

This smooth constructor implements univariate and bivariate Kriging terms. The basis functions are based on the Matern covariance function. For finding knots, a space filling algorithm is used, see `cover.design`.

Value

A smooth specification object, see also `smooth.construct`.

References

Fahrmeir, L., Kneib, T. , Lang, S., Marx, B. (2013): Regression. Models, Methods and Applications, Springer Verlag. `https://www.uni-goettingen.de/de/551357.html`

See Also

`bamlss`, `smooth.construct`

Examples

```
## Not run: ## Simulate data.
set.seed(123)
d <- GAMart()

## Estimate model.
f <- num ~ s(x1,bs="kr") + s(x2,bs="kr") + s(x3,bs="kr") + s(lon,lat,bs="kr",k=30)
```

```
## Set the seed, estimate model.
set.seed(111)
b <- bamlss(f, data = d)

## Plot estimated effects.
plot(b)

## End(Not run)
```

`smooth.construct.ms.smooth.spec` *Smooth constructor for monotonic P-splines*

Description

The function sets up a smooth term for shape constraint estimation of P-spline model terms. Note that this currently only works using boosting and backfitting.

Usage

```
## S3 method for class 'ms.smooth.spec'
smooth.construct(object, data, knots, ...)
```

Arguments

`object`	Either a smooth specification object, or object of class `"bamlss"`, `"bamlss.frame"`, `"bamlss.formula"` or `"bamlss.terms"`. For smooth specification objects, see function `smooth.construct`.
`data`	A data frame or list, see also see function `smooth.construct`.
`knots`	See function `smooth.construct`.
`...`	Arguments passed to the smooth term constructor functions.

Value

See function see `smooth.construct`.

See Also

`bamlss.frame`, `bamlss.formula`, `bamlss`, `smooth.construct`.

Examples

```
## Not run: ## Generate some data.
set.seed(123)

n <- 300
x <- runif(n, -2, 3)
y <- sin(x) + rnorm(n, sd = 0.1)

d <- data.frame("y" = y, "x" = x)

## Increasing: constr = 1.
## Decreasing: constr = 2.
b <- bamlss(y ~ s2(x,bs="ms",xt=list(constr=1)),
  data = d, optimizer = opt_bfit, sampler = sam_MVNORM)

## Predict and plot.
p <- predict(b, model = "mu", FUN = c95)
plot(y ~ x)
plot2d(p ~ x, add = TRUE, col.lines = 4, lwd = 2)

## End(Not run)
```

---

smooth.construct.sr.smooth.spec

Random Effects P-Spline

Description

This smooth constructor implements the random effects representation of a P-spline.

Usage

```
## S3 method for class 'sr.smooth.spec'
smooth.construct(object, data, knots, ...)
```

Arguments

object, data, knots	See smooth.construct.
...	Currently not used.

Value

See smooth.construct

See Also

bamlss, predict.bamlss, opt_bfit, opt_boost

Examples

```
## Not run: ## Simulate data.
set.seed(123)
d <- GAMart()

## Estimate model.
f <- num ~ x1 + x2 + x3 + s2(x1,bs="sr") + s2(x2,bs="sr") + s2(x3,bs="sr")

b <- bamlss(f, data = d, optimizer = boost, sampler = FALSE)

plot(b)

## End(Not run)
```

---

smooth_check *MCMC Based Simple Significance Check for Smooth Terms*

Description

For each smooth term estimated with MCMC, the function computes 95 intervals and simply computes the fraction of the cases where the interval does not contain zero.

Usage

```
smooth_check(object, newdata = NULL, model = NULL, term = NULL, ...)
```

Arguments

object	A fitted model object which contains MCMC samples.
newdata	Optionally, use new data for computing the check.
model	Character, for which model should the check be computed?
term	Character, for which term should the check be computed?
...	Arguments passed to predict.bamlss.

Examples

```
## Not run: ## Simulate some data.
d <- GAMart()

## Model formula.
f <- list(
  num ~ s(x1) + s(x2) + s(x3),
  sigma ~ s(x1) + s(x2) + s(x3)
)

## Estimate model with MCMC.
b <- bamlss(f, data = d)
```

```
## Run the check, note that all variables
## for sigma should have no effect.
smooth_check(b)

## End(Not run)
```

`stabsel` *Stability selection.*

Description

Performs stability selection based on gradient boosting.

Usage

```
stabsel(formula, data, family = "gaussian",
  q, maxit, B = 100, thr = .9, fraction = 0.5, seed = NULL, ...)

## Plot selection frequencies.
## S3 method for class 'stabsel'
plot(x, show = NULL,
  pal = function(n) gray.colors(n, start = 0.9, end = 0.3), ...)
```

Arguments

`formula`	A formula or extended formula.
`data`	A `data.frame`.
`family`	A `bamlss.family` object.
`q`	An integer specifying how many terms to select in each boosting run.
`maxit`	An integer specifying the maximum number of boosting iterations. See `opt_boost`. Either choose `q` or `maxit` as hyper-parameter for regularization.
`B`	An integer. The boosting is run B times.
`thr`	Cut-off threshold of relative frequencies (between 0 and 1) for selection.
`fraction`	Numeric between 0 and 1. The fraction of data to be used in each boosting run.
`seed`	A seed to be set before the stability selection.
`x`	A object of class stabsel.
`show`	Number of terms to be shown.
`pal`	Color palette for different model terms.
`...`	Not used yet in `stabsel`.

Details

stabsel performs stability selection based on gradient boosting (opt_boost): The boosting algorithm is run B times on a randomly drawn fraction of the data. Each boosting run is stopped either when q terms have been selected, or when maxit iterations have been performed, i.e. either q or maxit can be used to tune the regularization of the boosting. After the boosting the relative selection frequencies are evaluated. Terms with a relative selection frequency larger then thr are suggested for a final regression model.

If neither q nor maxit has been specified, q will be set to the square root of the number of columns in data.

Gradient boosting does not depend on random numbers. Thus, the individual boosting runs differ only in the subset of data which is used.

Value

A object of class stabsel.

Author(s)

Thorsten Simon

Examples

```
## Not run: ## Simulate some data.
set.seed(111)
d <- GAMart()
n <- nrow(d)

## Add some noise variables.
for(i in 4:9)
  d[[paste0("x",i)]] <- rnorm(n)

f <- paste0("~ ", paste("s(x", 1:9, ")", collapse = "+", sep = ""))
f <- paste(f, "+ te(lon,lat)")
f <- as.formula(f)
f <- list(update(f, num ~ .), f)

## Run stability selection.
sel <- stabsel(f, data = d, q = 6, B = 10)
plot(sel)

## Estimate selected model.
nf <- formula(sel)
b <- bamlss(nf, data = d)
plot(b)

## End(Not run)
```

summary.bamlss *Summary for BAMLSS*

Description

The function takes an object of class "bamlss" and produces summaries of optimizer and sampler function outputs.

Usage

```
## S3 method for class 'bamlss'
summary(object, model = NULL,
  FUN = NULL, parameters = TRUE, ...)

## S3 method for class 'summary.bamlss'
print(x, digits = max(3, getOption("digits") - 3), ...)
```

Arguments

object	An object of class "bamlss".
x	An oject of class "summary.bamlss".
model	Character or integer, specifies the model for which a summary should be computed.
FUN	Function that should be applied on samples, see also function coef.bamlss.
parameters	If an optimizer function is applied within the bamlss call, should the values of the estimated parameters be part of the summary?
digits	Controls number of digits printed in output.
...	Other arguments.

Details

If the fitted model contains samples, summaries according to the supplied function can be computed, e.g., different quantiles of samples. See also function coef.bamlss that extracts the coefficient summaries.

If an optimizer function was used within the bamlss call, estimated parameters will be included per default into the summary.

Note that summaries not based on samples can be user defined, e.g., as returned from function samplestats or the return values of optimizer function, e.g., see function opt_bfit.

Value

summary.bamlss produces the following summary:

call	The initial bamlss call.
family	The family that is used for modeling.

`formula`	The model formula.
`model.matrix`	Summary of parameteric terms.
`smooth.construct`	Summary of smooth terms.
`model.stats`	Other model statistics, e.g., as returned from optimizer functions and/or produces by function `samplestats`.

See Also

`bamlss`

Examples

```
## Not run: ## Generate some data.
d <- GAMart()

## Model formula.
f <- list(
  num ~ s(x1) + s(x2),
  sigma ~ s(x3) + te(lon,lat)
)

## Estimate model.
b <- bamlss(f, data = d)

## Print the summary.
print(summary(b))

## End(Not run)
```

`Surv2` *Create a Survival Object for Joint Models*

Description

This function is only a slight extension of `Surv` for joint models.

Usage

```
Surv2(..., obs = NULL)
```

Arguments

`...`	Arguments passed to function `Surv`.
`obs`	The observed longitudinal response.

Value

An object of class `"Surv2"` and `"matrix"`.

See Also

opt_JM, sam_JM, bamlss

Examples

```
## Surv2(time, event, obs = y)
## See the examples of opt_JM() and sam_JM()!
```

---

surv_transform *Survival Model Transformer Function*

Description

This function takes a bamlss.frame and computes design matrices of model terms based on a time grid for time-dependent structured additive predictors in a survival context. Note that this transformer function is usually used internally by function bamlss and is the default transformer function using the cox_bamlss family object.

The time grid design matrices can be used to construct the full structured additive predictor for each time point. This way it is possible to solve the integrals that are part of, e.g., a Newton-Raphson updating scheme, numerically.

See the example section on how to extract the time grid design matrices.

Usage

```
surv_transform(x, y, data, family,
  subdivisions = 100, timedependent = "lambda",
  timevar = NULL, idvar = NULL, is.cox = FALSE,
  alpha = 0.1, ...)
```

Arguments

x	The x list, as returned from function bamlss.frame and transformed by function surv_transform, holding all model matrices and other information that is used for fitting the model.
y	The model response, as returned from function bamlss.frame.
data	The data.frame that should be used for setting up all matrices.
family	A **bamlss** family object, see family.bamlss. In this case this is the cox_bamlss family object.
subdivisions	How many time points should be created for each individual.
timedependent	A character vector specifying the names of parameters in x that are time-dependent. Time grid design matrices are only computed for these parameters.
timevar	A character specifying the name of the survival time variable in the data set.
idvar	Depending on the type of data set, this is the name of the variable specifying identifier of individuals.

is.cox	Should the bamlss.frame be set up for a Cox type survival model.
alpha	A value for the intercept of a parameter names alpha. Typically the association parameter of a longitudinal and survival process in a joint model.
...	Arguments passed to function bamlss.engine.setup.

Value

A bamlss.frame including the time grid design matrices.

See Also

cox_bamlss, opt_Cox, sam_Cox, simSurv, bamlss

Examples

```
library("survival")
set.seed(111)

## Simulate survival data.
d <- simSurv(n = 20)

## Formula of the survival model, note
## that the baseline is given in the first formula by s(time).
f <- list(
  Surv(time, event) ~ s(time) + s(time, by = x3),
  gamma ~ s(x1) + s(x2)
)

## Create the bamlss.frame.
bf <- bamlss.frame(f, family = "cox", data = d)

## Lambda is the time-dependent parameter.
print(bf)

## Apply the transformer.
bf <- with(bf, surv_transform(x, y, data = model.frame,
  family = family, is.cox = TRUE, subdivisions = 25))

## Extract the time grid design matrix for term s(time).
X <- bf$x$lambda$smooth.construct[["s(time)"]]$fit.fun_timegrid(NULL)
dim(X)

## Compute fitted values for each time point.
grid <- attr(bf$y[[1]], "grid")
gdim <- c(length(grid), length(grid[[1]]))
b <- runif(ncol(X))
fit <- X %*% b
fit <- matrix(fit, nrow = gdim[1], ncol = gdim[2], byrow = TRUE)

plot(as.vector(fit) ~ unlist(grid), type = "n",
  xlab = "Survival time", ylab = "Effect")
for(j in seq_along(grid)) {
```

```
  lines(fit[j, ] ~ grid[[j]], lwd = 2, col = rgb(0.1, 0.1, 0.1, alpha = 0.3))
  points(grid[[j]][gdim[2]], fit[j, gdim[2]], col = "red")
}
```

TempIbk *Temperature data.*

Description

Temperature Data for Innsbruck Airport

Usage

```
data("TempIbk")
```

Format

An object of class `data.frame` with 1798 rows and 17 columns.

Details

Numerical weather predictions (NWP) and observations of 2 meter temperature at Innsbruck Airport. The observations from the SYNOP station 11120 cover 5 years from 2015-01-01 to 2019-31-12. The NWP data are derived from GEFS reforecasts (Hamill et al. 2013). The data contain following variables:

- `init`: Time of initialization of the NWP model.
- `obs_*`: Observations for lead time `*`.
- `mean_ens_*`: NWP ensemble mean for lead time `*`.
- `logsd_ens_*`: NWP logarithm of ensemble standard deviation for lead time `*`.
- `yday`: Yearday.

References

Hamill TM, Bates GT, Whitaker JS, Murray DR, Fiorino M, Galarneau Jr TJ, Zhu Y, Lapenta W (2013). NOAA's Second-Generation Global Medium-Range Ensemble Reforecast Data Set. *Bulletin of the American Meteorological Society*, 94(10), 1553-1565.

See Also

`mvnchol_bamlss`

Examples

```
## Not run: ## Innsbruck temperature data.
data("TempIbk", package = "bamlss")

## Five lead times.
lead <- seq(192, 216, by = 6)

## Set up formulas.
f <- c(
  ## mu equations
  sprintf('obs_%s ~ s(yday, bs = "cc") + s(yday, bs = "cc", by = mean_ens_%s)', lead, lead),

  ## lambda diag equations
  sprintf('lamdiag%s ~ s(yday, bs = "cc") + s(yday, bs = "cc", by = logsd_ens_%s)', 1:5, lead),

  ## lambda off-diag equations
  sprintf('lambda%s ~ s(yday, bs = "cc")', apply(combn(1:5, 2), 2, paste, collapse = ""))
)
f <- lapply(f, as.formula)

## Multivariate normal family with basic Cholesky parameterization.
fam <- mvnchol_bamlss(k = 5, type = "basic")

## Fit model.
set.seed(123)
b <- bamlss(f, family = fam, data = TempIbk, optimizer = opt_boost, maxit = 1000)

## Show estimated effects.
par(mfrow = c(2, 2))
plot(b, model = "mu1", scale = 0, spar = FALSE)
plot(b, model = "lamdiag2", term = "s(yday)", spar = FALSE)
plot(b, model = "lambda12")

## Predict sample case.
nd <- subset(TempIbk, format(init, "%Y-%m-%d") %in% c("2015-01-03", "2015-10-10"))
fit <- predict(b, newdata = nd, type = "parameter")

## Plot correlation matrix for GEFS initialization 2015-10-10.
plot_cor <- function(i) {
    image(lead, lead, fam$correlation(fit)[[i]][5:1, ], zlim = c(0, 1),
      col = hcl.colors(10, "Blues 3", rev = TRUE), axes = FALSE,
      xlab = "lead time in hours", ylab = "lead time in hours",
      main = sprintf("Correlation matrix fitted for %s", nd[i, "init"]))
    axis(1, lead)
    axis(2, lead, rev(lead))
    box()
}
par(mfrow = c(1, 2))
plot_cor(1)
plot_cor(2)

## Plot means and standard deviations.
```

```
plot_ms <- function(i) {
stdev <- fam$stdev(fit)[[i]]
means <- fam$means(fit)[[i]]
lower <- means - stdev
upper <- means + stdev

plot(lead, means, type = 'b', cex = 2, lwd = 1, lty = 2, axes = FALSE,
 ylim = c(-6, 16), # c(min(lower), max(upper)),
 ylab = expression("Temperature in " * degree * "C"),
 xlab = "lead time in hours",
 main = sprintf("Means +/- one st. dev. for %s", nd[i, "init"]))
segments(lead, y0 = lower, y1 = upper)
axis(1, lead)
axis(2)
box()
}
par(mfrow = c(1, 2))
plot_ms(1)
plot_ms(2)

## End(Not run)
```

---

terms.bamlss *BAMLSS Model Terms*

Description

Extract terms.objects for BAMLSS.

Usage

```
## S3 method for class 'bamlss'
terms(x, specials = NULL, data = NULL,
  model = NULL, pterms = TRUE, sterms = TRUE,
  drop = TRUE, ...)

## S3 method for class 'bamlss.frame'
terms(x, specials = NULL, data = NULL,
  model = NULL, pterms = TRUE, sterms = TRUE,
  drop = TRUE, ...)

## S3 method for class 'bamlss.formula'
terms(x, specials = NULL, data = NULL,
  model = NULL, pterms = TRUE, sterms = TRUE,
  drop = TRUE, ...)
```

Arguments

x	An link{bamlss}, bamlss.frame or bamlss.formula object.
specials	See terms.object.
data	Data passed to terms.formula.
model	Character or integer, specifies the model for which terms should be returned.
pterms	Should parametric terms be part of the object?
sterms	Should smooth terms be part of the object?
drop	If terms for only one model are returned, the list structure is dropped.
...	Arguments passed to bamlss.formula.

Value

Object of class "bamlss.terms", a list of terms.objects, depending on the structure of the bamlss.formula object.

See Also

bamlss, bamlss.frame, bamlss.formula.

Examples

```
## Model formula.
f <- list(
  num ~ x1 + x2 + id,
  sigma ~ x3 + fac + lon + lat
)

## Create the terms object.
terms(bamlss.formula(f))
```

---

trans_AR1 *AR1 Transformer Function*

Description

The transformer function takes a bamlss.frame object and transforms the response and the design matrices to account for lag 1 autocorrelation. The method is also known as Prais-Winsten estimation.

Usage

```
trans_AR1(rho = 0.1)
AR1(rho = 0.1)
```

Arguments

rho	Specifies the correlation parameter at lag 1.

Value

A transformer function which can be used in the bamlss call.

References

Johnston, John (1972). Econometric Methods (2nd ed.). New York: McGraw-Hill. pp. 259–265.

See Also

bamlss.frame, bamlss, smooth2random.

Examples

```
## Not run: ## Simulate AR1 data.
set.seed(111)

n <- 240
d <- data.frame("t" = 1:n)

## Nonlinear function.
f <- function(x) {
  2 + sin(x / n * 2 * pi - pi)
}

## Correlated errors.
rho <- 0.8
e <- rnorm(n, sd = 0.1)
u <- c(e[1], rep(NA, n - 1))
for(i in 2:n){
  u[i] <- rho * u[i - 1] + e[i]
}

## Response.
d$y <- f(d$t) + u

## Plot time-series data.
plot(d, type = "l")

## Estimate models without and with AR1 transformation.
b0 <- bamlss(y ~ s(t,k=20), data = d, criterion = "BIC")
b1 <- bamlss(y ~ s(t,k=20), data = d, criterion = "BIC",
  transform = AR1(rho = 0.8))

## Estimate full AR1 model.
b2 <- bamlss(y ~ s(t,k=20), data = d, criterion = "BIC",
  family = "AR1")
rho <- predict(b2, model = "rho", type = "parameter")
print(range(rho))

## Estimated standard deviations.
sd0 <- predict(b0, model = "sigma", type = "parameter")
sd1 <- predict(b1, model = "sigma", type = "parameter")
```

```
sd2 <- predict(b2, model = "sigma", type = "parameter")
print(round(c(sd0[1], sd1[1], sd2[1]), 2))

## Plot fitted trends.
p0 <- predict(b0, model = "mu")
p1 <- predict(b1, model = "mu")
p2 <- predict(b2, model = "mu")

plot(d, type = "l")
lines(f(d$t) ~ d$t, col = 2, lwd = 2)
lines(p0 ~ d$t, col = 4, lwd = 2)
lines(p1 ~ d$t, col = 3, lwd = 3)
lines(p2 ~ d$t, col = 5, lwd = 3)
legend("topleft",
  c("no trans", "with trans", "AR1 model", "truth"),
  lwd = 2, col = c(4, 3, 5, 2), bty = "n")

## End(Not run)
```

`Volcano` *Artificial Data Set based on Auckland's Maunga Whau Volcano*

Description

This function creates a data set based on the `volcano` data by adding normal errors to the topographic information.

Usage

```
Volcano(sd = 0.3)
```

Arguments

`sd`	The standard deviation of the normal errors.

Value

A data frame with coordinates and noisy elevation.

See Also

`volcano`

Examples

```
d <- Volcano()
head(d)

## Not run: b <- bamlss(y ~ te(lon,lat,k=10), data = d)
plot(b, theta = -130)
```

```
## End(Not run)
```

WAIC *Watanabe-Akaike Information Criterion (WAIC)*

Description

Function returning the Watanabe-Akaike Information Criterion (WAIC) of a fitted model object.

Usage

```
WAIC(object, ..., newdata = NULL)
```

Arguments

object	A fitted model object which contains MCMC samples.
...	Optionally more fitted model objects.
newdata	Optionally, use new data for computing the WAIC.

Value

A data frame containing the WAIC and estimated number of parameters.

References

Watanabe S. (2010). Asymptotic Equivalence of Bayes Cross Validation and Widely Applicable Information Criterion in Singular Learning Theory. *The Journal of Machine Learning Research*, **11**, 3571–3594. https://jmlr.org/papers/v11/watanabe10a.html

Examples

```
## Not run: d <- GAMart()
b1 <- bamlss(num ~ s(x1), data = d)
b2 <- bamlss(num ~ s(x1) + s(x2), data = d)
WAIC(b1, b2)

## End(Not run)
```

Index