gamboostLSS: An R Package for Model Building and Variable Selection in the GAMLSS Framework

Benjamin Hofner
FAU Erlangen-Nürnberg

Andreas Mayr
FAU Erlangen-Nürnberg

Matthias Schmid
University of Bonn

Abstract

This vignette is a slightly modified version of **?** *which appeared in the* ***Journal of Statistical Software****. Please cite that article when using the package* ***gamboostLSS*** *in your work.*

Generalized additive models for location, scale and shape are a flexible class of regression models that allow to model multiple parameters of a distribution function, such as the mean and the standard deviation, simultaneously. With the R package **gamboostLSS**, we provide a boosting method to fit these models. Variable selection and model choice are naturally available within this regularized regression framework. To introduce and illustrate the R package **gamboostLSS** and its infrastructure, we use a data set on stunted growth in India. In addition to the specification and application of the model itself, we present a variety of convenience functions, including methods for tuning parameter selection, prediction and visualization of results. The package **gamboostLSS** is available from CRAN (Red`http://cran.r-project.org/package=gamboostLSS`).

Keywords: additive models, prediction intervals, high-dimensional data.

1. Introduction

Generalized additive models for location, scale and shape (GAMLSS) are a flexible statistical method to analyze the relationship between a response variable and a set of predictor variables. Introduced by **?**, GAMLSS are an extension of the classical GAM (generalized additive model) approach (**?**). The main difference between GAMs and GAMLSS is that GAMLSS do not only model the conditional mean of the outcome distribution (location) but *several* of its parameters, including scale and shape parameters (hence the extension "LSS"). In Gaussian regression, for example, the density of the outcome variable Y conditional on the predictors $\mathbf{X}$ may depend on the mean parameter μ, and an additional scale parameter σ, which corresponds to the standard deviation of $Y|\mathbf{X}$. Instead of assuming σ to be fixed, as in classical GAMs, the Gaussian GAMLSS regresses both parameters on the predictor variables,

$$\mu = \mathsf{E}(y \,|\, \mathbf{X}) \quad = \quad \eta_\mu = \beta_{\mu,0} + \sum_j f_{\mu,j}(x_j), \tag{1}$$

$$\log(\sigma) = \log(\sqrt{\mathsf{VAR}(y \,|\, \mathbf{X})}) \quad = \quad \eta_\sigma = \beta_{\sigma,0} + \sum_j f_{\sigma,j}(x_j), \tag{2}$$

where η_μ and η_σ are *additive predictors* with parameter specific intercepts $\beta_{\mu,0}$ and $\beta_{\sigma,0}$, and functions $f_{\mu,j}(x_j)$ and $f_{\sigma,j}(x_j)$, which represent the effects of predictor x_j on μ and σ,

respectively. In this notation, the functional terms $f(\cdot)$ can denote various types of effects (e.g., linear, smooth, random).

In our case study, we will analyze the prediction of stunted growth for children in India via a Gaussian GAMLSS. The response variable is a stunting score, which is commonly used to relate the growth of a child to a reference population in order to assess effects of malnutrition in early childhood. In our analysis, we model the expected value (μ) of this stunting score and also its variability (σ) via smooth effects for mother- or child-specific predictors, as well as a spatial effect to account for the region of India where the child is growing up. This way, we are able to construct point predictors (via η_μ) and additionally child-specific prediction intervals (via η_μ and η_σ) to evaluate the individual risk of stunted growth.

In recent years, due to their versatile nature, GAMLSS have been used to address research questions in a variety of fields. Applications involving GAMLSS range from the normalization of complementary DNA microarray data (**?**) and the analysis of flood frequencies (**?**) to the development of rainfall models (**?**) and stream-flow forecasting models (**?**). The most prominent application of GAMLSS is the estimation of centile curves, e.g., for reference growth charts (**???**). The use of GAMLSS in this context has been recommended by the World Health Organization (see **?**, and the references therein).

Classical estimation of a GAMLSS is based on backfitting-type Gauss-Newton algorithms with AIC-based selection of relevant predictors. This strategy is implemented in the R (**?**) package **gamlss** (**???**), which provides a great variety of functions for estimation, hyper-parameter selection, variable selection and hypothesis testing in the GAMLSS framework.

In this article we present the R package **gamboostLSS** (**?**), which is designed as an alternative to **gamlss** for high-dimensional data settings where variable selection is of major importance. Specifically, **gamboostLSS** implements the *gamboostLSS* algorithm, which is a new fitting method for GAMLSS that was recently introduced by **?**. The *gamboostLSS* algorithm uses the same optimization criterion as the Gauss-Newton type algorithms implemented in the package **gamlss** (namely, the log-likelihood of the model under consideration) and hence fits the same type of statistical model. In contrast to **gamlss**, however, the **gamboostLSS** package operates within the component-wise gradient boosting framework for model fitting and variable selection (**??**). As demonstrated in **?**, replacing Gauss-Newton optimization by boosting techniques leads to a considerable increase in flexibility: Apart from being able to fit basically any type of GAMLSS, **gamboostLSS** implements an efficient mechanism for variable selection and model choice. As a consequence, **gamboostLSS** is a convenient alternative to the AIC-based variable selection methods implemented in **gamlss**. The latter methods can be unstable, especially when it comes to selecting possibly different sets of variables for multiple distribution parameters. Furthermore, model fitting via *gamboostLSS* is also possible for high-dimensional data with more candidate variables than observations ($p > n$), where the classical fitting methods become unfeasible.

The **gamboostLSS** package is a comprehensive implementation of the most important issues and aspects related to the use of the *gamboostLSS* algorithm. The package is available on CRAN (`http://cran.r-project.org/package=gamboostLSS`). Current development versions are hosted on GitHub (`https://github.com/hofnerb/gamboostLSS`). As will be demonstrated in this paper, the package provides a large number of response distributions (e.g., distributions for continuous data, count data and survival data, including *all* distributions currently available in the **gamlss** framework; see **?**). Moreover, users of **gamboostLSS**

can choose among many different possibilities for modeling predictor effects. These include linear effects, smooth effects and trees, as well as spatial and random effects, and interaction terms.

After starting with a toy example (Section **??**) for illustration, we will provide a brief theoretical overview of GAMLSS and component-wise gradient boosting (Section **??**). In Section **??**, we will introduce the `india` data set, which is shipped with the R package **gamboostLSS**. We present the infrastructure of **gamboostLSS**, discuss model comparison methods and model tuning, and will show how the package can be used to build regression models in the GAMLSS framework (Section **??**). In particular, we will give a step by step introduction to **gamboostLSS** by fitting a flexible GAMLSS model to the `india` data. In addition, we will present a variety of convenience functions, including methods for the selection of tuning parameters, prediction and the visualization of results (Section **??**).

2. A toy example

Before we discuss the theoretical aspects of the *gamboostLSS* algorithm and the details of the implementation, we present a short, illustrative toy example. This highlights the ease of use of the **gamboostLSS** package in simple modeling situations. Before we start, we load the package

```
R> library("gamboostLSS")
```

Note that **gamboostLSS** 1.2-0 or newer is needed. We simulate data from a heteroscedastic normal distribution, i.e., both the mean and the variance depend on covariates:

```
R> set.seed(1907)
R> n <- 150
R> x1 <- rnorm(n)
R> x2 <- rnorm(n)
R> x3 <- rnorm(n)
R> toydata <- data.frame(x1 = x1, x2 = x2, x3 = x3)
R> toydata$y <- rnorm(n, mean = 1 + 2 * x1 - x2,
+                        sd = exp(0.5 - 0.25 * x1 + 0.5 * x3))
```

Next we fit a linear model for location, scale and shape to the simulated data

```
R> lmLSS <- glmboostLSS(y ~ x1 + x2 + x3, data = toydata)
```

and extract the coefficients using `coef(lmLSS)`. When we add the offset (i.e., the starting values of the fitting algorithm) to the intercept, we obtain

```
R> coef(lmLSS, off2int = TRUE)

$mu
(Intercept)          x1          x2
  0.8139756   1.6411143  -0.3905382
```

```
$sigma
(Intercept)          x1          x2          x3
 0.62351136 -0.22308703 -0.02128006  0.30850745
```

Usually, model fitting involves additional tuning steps, which are skipped here for the sake of simplicity (see Section **??** for details). Nevertheless, the coefficients coincide well with the true effects, which are $\beta_\mu = (1, 2, -1, 0)$ and $\beta_\sigma = (0.5, -0.25, 0, 0.5)$. To get a graphical display, we `plot` the resulting model

```
R> par(mfrow = c(1, 2), mar = c(4, 4, 2, 5))
R> plot(lmLSS, off2int = TRUE)
```

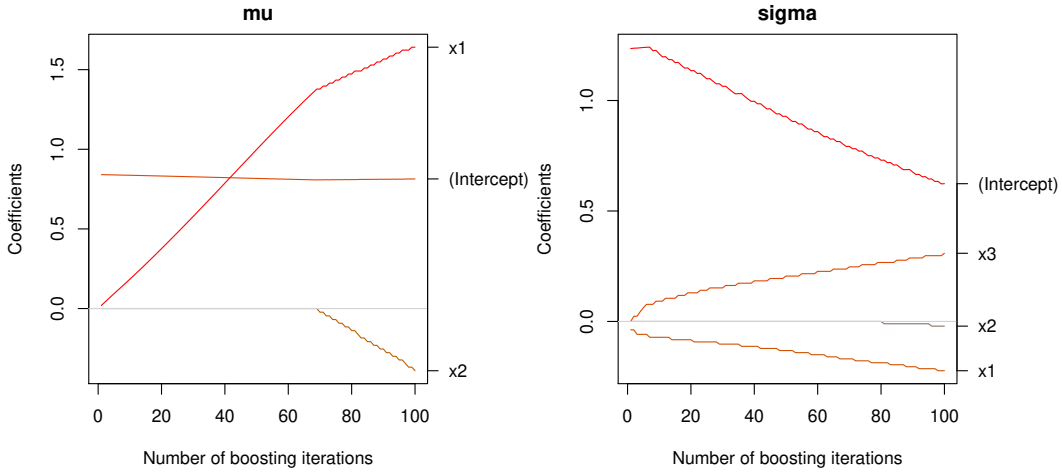

Figure 1: Coefficient paths for linear LSS models, which depict the change of the coefficients over the iterations of the algorithm.

To extract fitted values for the mean, we use the function `fitted(, parameter = "mu")`. The results are very similar to the true values:

```
R> muFit <- fitted(lmLSS, parameter = "mu")
R> rbind(muFit, truth = 1 + 2 * x1 - x2)[, 1:5]
```

```
              1          2         3         4         5
muFit -3.243757 -0.6727116 0.8922116 1.049360 0.8499387
truth -4.331456 -0.8519794 0.7208595 1.164517 1.1033806
```

The same can be done for the standard deviation, but we need to make sure that we apply the response function (here $\exp(\eta)$) to the fitted values by additionally using the option `type = "response"`:

```
R> sigmaFit <- fitted(lmLSS, parameter = "sigma", type = "response")[, 1]
R> rbind(sigmaFit, truth = exp(0.5 - 0.25 * x1 + 0.5 * x3))[, 1:5]
```

```
                1        2        3        4        5
sigmaFit 2.613536 1.469919 1.503953 2.225158 2.527370
truth    2.260658 1.017549 1.221171 2.261453 2.684958
```

For new observations stored in a data set `newData` we could use `predict(lmLSS, newdata = newData)` essentially in the same way. As presented in Section **??**, the complete distribution could also be depicted as marginal prediction intervals via the function `predint()`.

3. Boosting GAMLSS models

GamboostLSS is an algorithm to fit GAMLSS models via component-wise gradient boosting (**?**) adapting an earlier strategy by **?**. While the concept of boosting emerged from the field of supervised machine learning, boosting algorithms are nowadays often applied as flexible alternative to estimate and select predictor effects in statistical regression models (statistical boosting, **?**). The key idea of statistical boosting is to iteratively fit the different predictors with simple regression functions (base-learners) and combine the estimates to an additive predictor. In case of gradient boosting, the base-learners are fitted to the negative gradient of the loss function; this procedure can be described as gradient descent in function space (**?**). For GAMLSS, we use the negative log-likelihood as loss function. Hence, the negative gradient of the loss function equals the (positive) gradient of the log-likelihood. To avoid confusion we directly use the gradient of the log-likelihood in the remainder of the article.

To adapt the standard boosting algorithm to fit additive predictors for all distribution parameters of a GAMLSS we extended the component-wise fitting to multiple parameter dimensions: In each iteration, *gamboostLSS* calculates the partial derivatives of the log-likelihood function $l(y, \boldsymbol{\theta})$ with respect to each of the additive predictors η_{θ_k}, $k = 1, \ldots, K$. The predictors are related to the parameter vector $\boldsymbol{\theta} = (\theta_k)_{k=1,\ldots,K}^{\top}$ via parameter-specific link functions g_k, $\theta_k = g_k^{-1}(\eta_{\theta_k})$. Typically, we have at maximum $K = 4$ distribution parameters (**?**), but in principle more are possible. The predictors are updated successively in each iteration, while the current estimates of the other distribution parameters are used as offset values. A schematic representation of the updating process of *gamboostLSS* with four parameters in iteration $m + 1$ looks as follows:

$$
\begin{aligned}
\frac{\partial}{\partial \eta_\mu} l(y, \hat{\mu}^{[m]} &, \hat{\sigma}^{[m]} &, \hat{\nu}^{[m]} &, \hat{\tau}^{[m]}) &\xrightarrow{\text{update}}\ & \hat{\eta}_\mu^{[m+1]} \Longrightarrow \hat{\mu}^{[m+1]} ,\\
\frac{\partial}{\partial \eta_\sigma} l(y, \hat{\mu}^{[m+1]} &, \hat{\sigma}^{[m]} &, \hat{\nu}^{[m]} &, \hat{\tau}^{[m]}) &\xrightarrow{\text{update}}\ & \hat{\eta}_\sigma^{[m+1]} \Longrightarrow \hat{\sigma}^{[m+1]} ,\\
\frac{\partial}{\partial \eta_\nu} l(y, \hat{\mu}^{[m+1]} &, \hat{\sigma}^{[m+1]} &, \hat{\nu}^{[m]} &, \hat{\tau}^{[m]}) &\xrightarrow{\text{update}}\ & \hat{\eta}_\nu^{[m+1]} \Longrightarrow \hat{\nu}^{[m+1]} ,\\
\frac{\partial}{\partial \eta_\tau} l(y, \hat{\mu}^{[m+1]} &, \hat{\sigma}^{[m+1]} &, \hat{\nu}^{[m+1]} &, \hat{\tau}^{[m]}) &\xrightarrow{\text{update}}\ & \hat{\eta}_\tau^{[m+1]} \Longrightarrow \hat{\tau}^{[m+1]} .
\end{aligned}
$$

The algorithm hence circles through the different parameter dimensions: in every dimension, it carries out one boosting iteration, updates the corresponding additive predictor and includes the new prediction in the loss function for the next dimension.

As in classical statistical boosting, inside each boosting iteration only the best fitting base-learner is included in the update. Typically, each base-learner corresponds to one component

of $\mathbf{X}$, and in every boosting iteration only a small proportion (a typical value of the *step-length* is 0.1) of the fit of the selected base-learner is added to the current additive predictor $\eta_{\theta_k}^{[m]}$. This procedure effectively leads to data-driven variable selection which is controlled by the stopping iterations $\boldsymbol{m}_{\text{stop}} = (m_{\text{stop},1}, ..., m_{\text{stop},K})^\top$: Each additive predictor η_{θ_k} is updated until the corresponding stopping iterations $\boldsymbol{m}_{\text{stop},k}$ is reached. If m is greater than $m_{\text{stop},k}$, the kth distribution parameter dimension is no longer updated. Predictor variables that have never been selected up to iteration $m_{\text{stop},k}$ are effectively excluded from the resulting model. The vector $\boldsymbol{m}_{\text{stop}}$ is a tuning parameter that can, for example, be determined using multi-dimensional cross-validation (see Section **??** for details). A discussion of model comparison methods and diagnostic checks can be found in Section **??**. The complete *gamboostLSS* algorithm can be found in Appendix **??** and is described in detail in **?**.

Scalability of boosting algorithms One of the main advantages of boosting algorithms in practice, besides the automated variable selection, is their applicability in situations with more variables than observations ($p > n$). Despite the growing model complexity, the run time of boosting algorithms for GAMs increases only linearly with the number of base-learners **?**. An evaluation of computing times for up to $p = 10000$ predictors can be found in **?**. In case of boosting GAMLSS, the computational complexity additionally increases with the number of distribution parameters K. For an example on the performance of *gamboostLSS* in case of $p > n$ see the simulation studies provided in **?**. To speed up computations for the tuning of the algorithm via cross-validation or resampling, **gamboostLSS** incorporates parallel computing (see Section **??**).

4. Childhood malnutrition in India

Eradicating extreme poverty and hunger is one of the Millennium Development Goals that all 193 member states of the United Nations have agreed to achieve by the year 2015. Yet, even in democratic, fast-growing emerging countries like India, which is one of the biggest global economies, malnutrition of children is still a severe problem in some parts of the population. Childhood malnutrition in India, however, is not necessarily a consequence of extreme poverty but can also be linked to low educational levels of parents and cultural factors (**?**). Following a bulletin of the World Health Organization, growth assessment is the best available way to define the health and nutritional status of children (**?**). Stunted growth is defined as a reduced growth rate compared to a standard population and is considered as the first consequence of malnutrition of the mother during pregnancy, or malnutrition of the child during the first months after birth. Stunted growth is often measured via a Z score that compares the anthropometric measures of the child with a reference population:

$$Z_i = \frac{\text{AI}_i - \text{MAI}}{s}$$

In our case, the individual anthropometric indicator (AI_i) will be the height of the child i, while MAI and s are the median and the standard deviation of the height of children in a reference population. This Z score will be denoted as *stunting score* in the following. Negative values of the score indicate that the child's growth is below the expected growth of a child with normal nutrition.

The stunting score will be the outcome (response) variable in our application,: we analyze the relationship of the mother's and the child's body mass index (BMI) and age with stunted growth resulting from malnutrition in early childhood. Furthermore, we will investigate regional differences by including also the district of India in which the child is growing up. The aim of the analysis is both, to explain the underlying structure in the data as well as to develop a prediction model for children growing up in India. A prediction rule, based also on regional differences, could help to increase awareness for the individual risk of a child to suffer from stunted growth due to malnutrition. For an in-depth analysis on the multi-factorial nature of child stunting in India, based on boosted quantile regression, see **?**, and **?**.

The data set that we use in this analysis is based on the Standard Demographic and Health Survey, 1998-99, on malnutrition of children in India, which can be downloaded after registration from `http://www.measuredhs.com`. For illustrative purposes, we use a random subset of 4000 observations from the original data (approximately 12%) and only a (very small) subset of variables. For details on the data set and the data source see the help file of the `india` data set in the **gamboostLSS** package and **?**.

Case study: Childhood malnutrition in India First of all we load the data sets `india` and `india.bnd` into the workspace. The first data set includes the outcome and 5 explanatory variables. The latter data set consists of a special boundary file containing the neighborhood structure of the districts in India.

```
R> data("india")
R> data("india.bnd")
R> names(india)
```

```
[1] "stunting"   "cbmi"       "cage"       "mbmi"       "mage"
[6] "mcdist"     "mcdist_lab"
```

The outcome variable `stunting` is depicted with its spatial structure in Figure **??**. An overview of the data set can be found in Table **??**. One can clearly see a trend towards malnutrition in the data set as even the 75% quantile of the stunting score is below zero. ◆

		Min.	25% Qu.	Median	Mean	75% Qu.	Max.
Stunting	`stunting`	-5.99	-2.87	-1.76	-1.75	-0.65	5.64
BMI (child)	`cbmi`	10.03	14.23	15.36	15.52	16.60	25.95
Age (child; months)	`cage`	0.00	8.00	17.00	17.23	26.00	35.00
BMI (mother)	`mbmi`	13.14	17.85	19.36	19.81	21.21	39.81
Age (mother; years)	`mage`	13.00	21.00	24.00	24.41	27.00	49.00

Table 1: Overview of `india` data.

5. The package gamboostLSS

The *gamboostLSS* algorithm is implemented in the publicly available R add-on package **gamboostLSS** (**?**). The package makes use of the fitting algorithms and some of the infrastructure

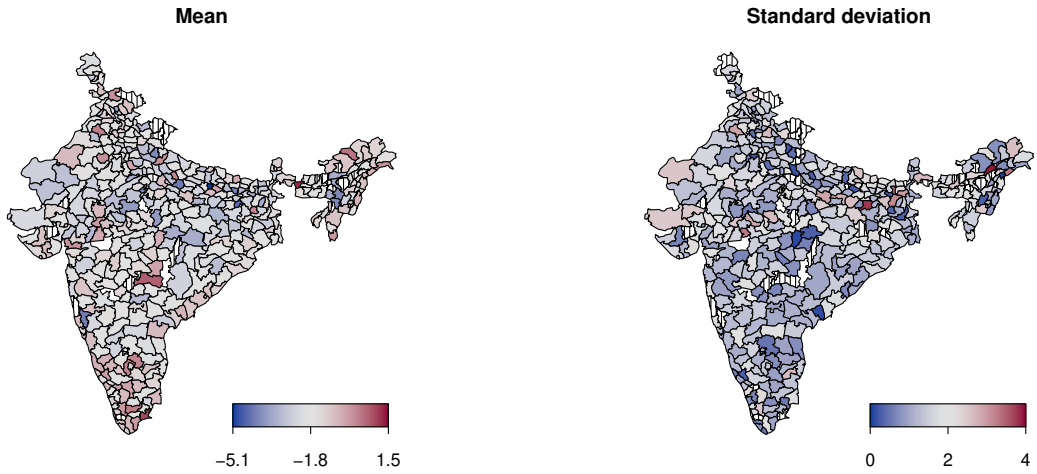

Figure 2: Spatial structure of stunting in India. The raw mean per district is given in the left figure, ranging from dark blue (low stunting score), to dark red (higher scores). The right figure depicts the standard deviation of the stunting score in the district, ranging from dark blue (no variation) to dark red (maximal variability). Dashed regions represent regions without data.

of **mboost** (**???**). Furthermore, many naming conventions and features are implemented in analogy to **mboost**. By relying on the **mboost** package, **gamboostLSS** incorporates a wide range of base-learners and hence offers a great flexibility when it comes to the types of predictor effects on the parameters of a GAMLSS distribution. In addition to making the infrastructure available for GAMLSS, **mboost** constitutes a well-tested, mature software package in the back end. For the users of **mboost**, **gamboostLSS** offers the advantage of providing a drastically increased number of possible distributions to be fitted by boosting.

As a consequence of this partial dependency on **mboost**, we recommend users of **gamboostLSS** to make themselves familiar with the former before using the latter package. To make this tutorial self-contained, we try to shortly explain all relevant features here as well. However, a dedicated hands-on tutorial is available for an applied introduction to **mboost** (**?**).

5.1. Model fitting

The models can be fitted using the function `glmboostLSS()` for linear models. For all kinds of structured additive models the function `gamboostLSS()` can be used. The function calls are as follows:

```
R> glmboostLSS(formula, data = list(), families = GaussianLSS(),
+              control = boost_control(), weights = NULL, ...)
R> gamboostLSS(formula, data = list(), families = GaussianLSS(),
+              control = boost_control(), weights = NULL, ...)
```

Note that here and in the remainder of the paper we sometimes focus on the most relevant (or most interesting) arguments of a function only. Further arguments might exist. Thus, for a complete list of arguments and their description we refer the reader to the respective help file.

The `formula` can consist of a single `formula` object, yielding the same candidate model for all distribution parameters. For example,

```
R> glmboostLSS(y ~ x1 + x2 + x3, data = toydata)
```

specifies linear models with predictors `x1` to `x3` for all GAMLSS parameters (here μ and σ of the Gaussian distribution). As an alternative, one can also use a named list to specify different candidate models for different parameters, e.g.,

```
R> glmboostLSS(list(mu = y ~ x1 + x2, sigma = y ~ x1 + x3), data = toydata)
```

fits a linear model with predictors `x1` and `x2` for the `mu` component and a linear model with predictors `x1` and `x3` for the `sigma` component. As for all R functions with a formula interface, one must specify the data set to be used (argument `data`). Additionally, `weights` can be specified for weighted regression. Instead of specifying the argument `family` as in **mboost** and other modeling packages, the user needs to specify the argument `families`, which basically consists of a list of sub-families, i.e., one family for each of the GAMLSS distribution parameters. These sub-families define the parameters of the GAMLSS distribution to be fitted. Details are given in the next section.

The initial number of boosting iterations as well as the step-lengths (ν_{sl}; see Appendix **??**) are specified via the function `boost_control()` with the same arguments as in **mboost**. However, in order to give the user the possibility to choose different values for each additive predictor (corresponding to the different parameters of a GAMLSS), they can be specified via a vector or list. Preferably a *named* vector or list should be used, where the names correspond to the names of the sub-families. For example, one can specify:

```
R> boost_control(mstop = c(mu = 100, sigma = 200),
+                nu = c(mu = 0.2, sigma = 0.01))
```

Specifying a single value for the stopping iteration `mstop` or the step-length `nu` results in equal values for all sub-families. The defaults is `mstop = 100` for the initial number of boosting iterations and `nu = 0.1` for the step-length. Additionally, the user can specify if status information should be printed by setting `trace = TRUE` in `boost_control`. Note that the argument `nu` can also refer to one of the GAMLSS distribution parameters in some families (and is also used in **gamlss** as the name of a distribution parameter). In `boost_control`, however, `nu` always represents the step-length ν_{sl}.

5.2. Distributions

Some GAMLSS distributions are directly implemented in the R add-on package **gamboostLSS** and can be specified via the `families` argument in the fitting function `gamboostLSS()` and `glmboostLSS()`. An overview of the implemented families is given in Table **??**. The parametrization of the negative binomial distribution, the log-logistic distribution and the t distribution in boosted GAMLSS models is given in **?**. The derivation of boosted beta regression, another special case of GAMLSS, can be found in **?**. In our case study we will use the default `GaussianLSS()` family to model childhood malnutrition in India. The resulting object of the family looks as follows:

```
R> str(GaussianLSS(), 1)

List of 2
 $ mu   :Formal class 'boost_family' [package "mboost"] with 10 slots
 $ sigma:Formal class 'boost_family' [package "mboost"] with 10 slots
 - attr(*, "class")= chr "families"
 - attr(*, "qfun")=function (p, mu = 0, sigma = 1, lower.tail = TRUE, log.p = FALSE)
 - attr(*, "name")= chr "Gaussian"
```

We obtain a list of class `"families"` with two sub-families, one for the μ parameter of the distribution and another one for the σ parameter. Each of the sub-families is of type `"boost_family"` from package **mboost**. Attributes specify the name and the quantile function (`"qfun"`) of the distribution.

In addition to the families implemented in the **gamboostLSS** package, there are many more possible GAMLSS distributions available in the **gamlss.dist** package (**?**). In order to make our boosting approach available for these distributions as well, we provide an interface to automatically convert available distributions of **gamlss.dist** to objects of class `"families"` to be usable in the boosting framework via the function `as.families()`. As input, a character string naming the `"gamlss.family"`, or the function itself is required. The function `as.families()` then automatically constructs a `"families"` object for the **gamboostLSS** package. To use for example the gamma family as parametrized in **gamlss.dist**, one can simply use `as.families("GA")` and plug this into the fitting algorithms of **gamboostLSS**:

```
R> gamboostLSS(y ~ x, families = as.families("GA"))
```

	Name	Response	μ	σ	ν	Note
Continuous response						
Gaussian	`GaussianLSS()`	cont.	id	log		
Student's t	`StudentTLSS()`	cont.	id	log	log	The 3rd parameter is denoted by `df` (degrees of freedom).
Continuous non-negative response						
Gamma	`GammaLSS()`	cont. > 0	log	log		
Fractions and bounded continuous response						
Beta	`BetaLSS()`	$\in (0, 1)$	logit	log		The 2nd parameter is denoted by `phi`.
Models for count data						
Negative binomial	`NBinomialLSS()`	count	log	log		For over-dispersed count data.
Zero inflated Poisson	`ZIPoLSS()`	count	log	logit		For zero-inflated count data; the 2nd parameter is the probability parameter of the zero mixture component.
Zero inflated neg. binomial	`ZINBLSS()`	count	log	log	logit	For over-dispersed and zero-inflated count data; the 3rd parameter is the probability parameter of the zero mixture component.
Survival models (accelerated failure time models; see, e.g., **?**)						
Log-normal	`LogNormalLSS()`	cont. > 0	id	log		All three families assume that the data are
Weibull	`WeibullLSS()`	cont. > 0	id	log		subject to right-censoring. Therefore the
Log-logistic	`LogLogLSS()`	cont. > 0	id	log		response must be a `Surv()` object.

Table 2: Overview of `"families"` that are implemented in **gamboostLSS**. For every distribution parameter the corresponding link-function is displayed (id = identity link).

With this interface, it is possible to apply boosting for any distribution implemented in **gamlss.dist** and for all new distributions that will be added in the future. Note that one can also fit censored or truncated distributions by using `gen.cens()` (from package **gamlss.cens**; see **?**) or `gen.trun()` (from package **gamlss.tr**; see **?**), respectively. An overview of common GAMLSS distributions is given in Appendix **??**. Minor differences in the model fit when applying a pre-specified distribution (e.g., `GaussianLSS()`) and the transformation of the corresponding distribution from **gamlss.dist** (e.g., `as.families("NO")`) can be explained by possibly different offset values.

5.3. Base-learners

For the base-learners, which carry out the fitting of the gradient vectors using the covariates, the **gamboostLSS** package completely depends on the infrastructure of **mboost**. Hence, every base-learner which is available in **mboost** can also be applied to fit GAMLSS distributions via **gamboostLSS**. The choice of base-learners is crucial for the application of the *gamboostLSS* algorithm, as they define the type(s) of effect(s) that covariates will have on the predictors of the GAMLSS distribution parameters. See **?** for details and application notes on the base-learners.

The available base-learners include simple linear models for *linear* effects and penalized regression splines (*P*-splines, **?**) for *non-linear* effects. *Spatial* or other *bivariate* effects can be incorporated by setting up a bivariate tensor product extension of P-splines for two continuous variables (**?**). Another way to include spatial effects is the adaptation of Markov random fields for modeling a neighborhood structure (**?**) or radial basis functions (**?**). *Constrained* effects such as monotonic or cyclic effects can be specified as well (**??**). *Random* effects can be taken into account by using ridge-penalized base-learners for fitting categorical grouping variables such as random intercepts or slopes (see supplementary material of **?**).

Case study (*cont'd*): Childhood malnutrition in India
First, we are going to set up and fit our model. Usually, one could use `bmrf(mcdist, bnd = india.bnd)` to specify the spatial base-learner using a Markov random field. However, as it is relatively time-consuming to compute the neighborhood matrix from the boundary file and as we need it several times, we pre-compute it once. Note that **R2BayesX** (**?**) needs to be loaded in order to use this function:

```
R> library("R2BayesX")
R> neighborhood <- bnd2gra(india.bnd)
```

The other effects can be directly specified without further care. We use smooth effects for the age (`mage`) and BMI (`mbmi`) of the mother and smooth effects for the age (`cage`) and BMI (`cbmi`) of the child. Finally, we specify the spatial effect for the district in India where mother and child live (`mcdist`).

We set the options

```
R> ctrl <- boost_control(trace = TRUE, mstop = c(mu = 1269, sigma = 84))
```

and fit the boosting model

```
R> mod_nonstab <- gamboostLSS(stunting ~ bbs(mage) + bbs(mbmi) +
+                               bbs(cage) + bbs(cbmi) +
+                               bmrf(mcdist, bnd = neighborhood),
+                             data = india,
+                             families = GaussianLSS(),
+                             control = ctrl)

[    1] ..................................... -- risk: 7351.327
[   39] ..................................... -- risk: 7256.697

(...)

[1'217] ..................................... -- risk: 7082.747
[1'255] ..............
Final risk: 7082.266
```

We specified the initial number of boosting iterations as `mstop = c(mu = 1269, sigma = 84)`, i.e., we used 1269 boosting iterations for the μ parameter and only 84 for the σ parameter. This means that we cycle between the μ and σ parameter until we have computed 84 update steps in both sub-models. Subsequently, we update only the μ model and leave the σ model unchanged. The selection of these tuning parameters will be discussed in the next section. ◆

Instead of optimizing the gradients per GAMLSS parameter in each boosting iteration, one can potentially stabilize the estimation further by standardizing the gradients in each step. Details and an explanation are given in Appendix **??**.

Case study (*cont'd*): Childhood malnutrition in India We now refit the model with the built-in median absolute deviation (MAD) stabilization by setting `stabilization = "MAD"` in the definition of the families:

```
R> mod <- gamboostLSS(stunting ~ bbs(mage) + bbs(mbmi) +
+                       bbs(cage) + bbs(cbmi) +
+                       bmrf(mcdist, bnd = neighborhood),
+                     data = india,
+                     families = GaussianLSS(stabilization = "MAD"),
+                     control = ctrl)

[    1] ..................................... -- risk: 7231.517
[   39] ..................................... -- risk: 7148.868

(...)

[1'217] ..................................... -- risk: 7003.024
[1'255] ..............
Final risk: 7002.32
```

One can clearly see that the stabilization changes the model and reduces the intermediate and final risks. ◆

5.4. Model complexity and diagnostic checks

Measuring the complexity of a GAMLSS is a crucial issue for model building and parameter tuning, especially with regard to the determination of optimal stopping iterations for gradient boosting (see next section). In the GAMLSS framework, valid measures of the complexity of a fitted model are even more important than in classical regression, since variable selection and model choice have to be carried out in *several* additive predictors within the same model.

In the original work by **?**, the authors suggested to evaluate AIC-type information criteria to measure the complexity of a GAMLSS. Regarding the complexity of a classical boosting fit with one predictor, AIC-type measures are available for a limited number of distributions (see **?**). Still, there is no commonly accepted approach to measure the degrees of freedom of a boosting fit, even in the classical framework with only one additive predictor. This is mostly due to the algorithmic nature of gradient boosting, which results in regularized model fits for which complexity is difficult to evaluate **?**. As a consequence, the problem of deriving valid (and easy-to-compute) complexity measures for boosting remains largely unsolved (**?**, Sec. 4).

In view of these considerations, and because it is not possible to use the original information criteria specified for GAMLSS in the *gamboostLSS* framework, **?** suggested to use cross-validated estimates of the empirical risk (i.e., of the predicted log-likelihood) to measure the complexity of *gamboostLSS* fits. Although this strategy is computationally expensive and might be affected by the properties of the used cross-validation technique, it is universally applicable to all **gamboostLSS** families and does not rely on possibly biased estimators of the effective degrees of freedom. We therefore decided to implement various resampling procedures in the function `cvrisk()` to estimate model complexity of a `gamboostLSS` fit via cross-validated empirical risks (see next section).

A related problem is to derive valid diagnostic checks to compare different families or link functions. For the original GAMLSS method, **?** proposed to base diagnostic checks on normalized quantile residuals. In the boosting framework, however, residual checks are generally difficult to derive because boosting algorithms result in regularized fits that reflect the trade-off between bias and variance of the effect estimators. As a consequence, residuals obtained from boosting fits usually contain a part of the remaining structure of the predictor effects, rendering an unbiased comparison of competing model families via residual checks a highly difficult issue. While it is of course possible to compute residuals from `gamboostLSS` models, valid comparisons of competing models are more conveniently obtained by considering estimates of the predictive risk.

Case study (*cont'd*): Childhood malnutrition in India To extract the empirical risk in the last boosting iteration (i.e., in the last step) of the model which was fitted with stabilization (see Page **??**) one can use

```
R> emp_risk <- risk(mod, merge = TRUE)
R> tail(emp_risk, n = 1)

     mu
7002.32
```

and compare it to the risk of the non-stabilized model

```
R> emp_risk_nonstab <- risk(mod_nonstab, merge = TRUE)
R> tail(emp_risk_nonstab, n = 1)

      mu
7082.266
```

In this case, the stabilized model has a lower (in-bag) risk than the non-stabilized model. Note that usually both models should be tuned before the empirical risk is compared. Here it merely shows that the risk of the stabilized model decreases quicker.

To compare the risk on new data sets, i.e., the predictive risk, one could combine all data in one data set and use weights that equal zero for the new data. Let us fit the model only on a random subset of 2000 observations. To extract the risk for observations with zero weights, we need to additionally set `risk = "oobag"`.

```
R> weights <- sample(c(rep(1, 2000), rep(0, 2000)))
R> mod_subset <- update(mod, weights = weights, risk = "oobag")
```

Note that we could also specify the model anew via

```
R> mod_subset <- gamboostLSS(stunting ~ bbs(mage) + bbs(mbmi) +
+                        bbs(cage) + bbs(cbmi) +
+                        bmrf(mcdist, bnd = neighborhood),
+                      data = india,
+                      weights = weights,
+                      families = GaussianLSS(),
+                      control = boost_control(mstop = c(mu = 1269, sigma = 84),
+                                              risk = "oobag"))
```

To refit the non-stabilized model we use

```
R> mod_nonstab_subset <- update(mod_nonstab,
+                               weights = weights, risk = "oobag")
```

Now we extract the predictive risks which are now computed on the 2000 "new" observations:

```
R> tail(risk(mod_subset, merge = TRUE), 1)

      mu
3605.222

R> tail(risk(mod_nonstab_subset, merge = TRUE), 1)

      mu
3609.056
```

Again, the stabilized model has a lower predictive risk. ◆

5.5. Model tuning: Early stopping to prevent overfitting

As for other component-wise boosting algorithms, the most important tuning parameter of the *gamboostLSS* algorithm is the stopping iteration $\boldsymbol{m}_{\text{stop}}$ (here a K-dimensional vector). In some low-dimensional settings it might be convenient to let the algorithm run until convergence (i.e., use a large number of iterations for each of the K distribution parameters). In these cases, as they are optimizing the same likelihood, boosting should converge to the same model as **gamlss** – at least when the same penalties are used for smooth effects.

However, in most settings, where the application of boosting is favorable, it is crucial that the algorithm is not run until convergence but some sort of early stopping is applied (**?**). Early stopping results in shrunken effect estimates, which has the advantage that predictions become more stable since the variance of the estimates is reduced. Another advantage of early stopping is that *gamboostLSS* has an intrinsic mechanism for data-driven variable selection, since only the best-fitting base-learner is updated in each boosting iteration. Hence, the stopping iteration $m_{\text{stop},k}$ does not only control the amount of shrinkage applied to the effect estimates but also the complexity of the models for the distribution parameter θ_k.

To find the optimal complexity, the resulting model should be evaluated regarding the predictive risk on a large grid of stopping values by cross-validation or resampling methods, using the function `cvrisk()`. In case of *gamboostLSS*, the predictive risk is computed as the negative log likelihood of the out-of-bag sample. The search for the optimal $\boldsymbol{m}_{\text{stop}}$ based on resampling is far more complex than for standard boosting algorithms. Different stopping iterations can be chosen for the parameters, thus allowing for different levels of complexity in each sub-model (*multi-dimensional* early stopping). In the package **gamboostLSS** a multi-dimensional grid can be easily created utilizing the function `make.grid()`.

In most of the cases the μ parameter is of greatest interest in a GAMLSS model and thus more care should be taken to accurately model this parameter. **?**, the inventors of GAMLSS, stated on the help page for the function `gamlss()`: "Respect the parameter hierarchy when you are fitting a model. For example a good model for μ should be fitted before a model for σ is fitted.". Consequently, we provide an option `dense_mu_grid` in the `make.grid()` function that allows to have a finer grid for (a subset of) the μ parameter. Thus, we can better tune the complexity of the model for μ which helps to avoid over- or underfitting of the mean without relying to much on the grid. Details and explanations are given in the following paragraphs.

Case study (*cont'd*): Childhood malnutrition in India We first set up a grid for `mstop` values starting at 20 and going in 10 equidistant steps on a logarithmic scale to 500:

```
R> grid <- make.grid(max = c(mu = 500, sigma = 500), min = 20,
+                    length.out = 10, dense_mu_grid = FALSE)
```

Additionally, we can use the `dense_mu_grid` option to create a dense grid for μ. This means that we compute the risk for all iterations $m_{\text{stop},\mu}$, if $m_{\text{stop},\mu} \geq m_{\text{stop},\sigma}$ and do not use the values on the sparse grid only:

```
R> densegrid <- make.grid(max = c(mu = 500, sigma = 500), min = 20,
+                         length.out = 10, dense_mu_grid = TRUE)
```

```
R> plot(densegrid, pch = 20, cex = 0.2,
+       xlab = "Number of boosting iterations (mu)",
+       ylab = "Number of boosting iterations (sigma)")
R> abline(0,1)
R> points(grid, pch = 20, col = "red")
```

A comparison and an illustration of the sparse and the dense grids can be found in Figure **??** (left). Red dots refer to all possible combinations of $m_{\text{stop},\mu}$ and $m_{\text{stop},\sigma}$ on the sparse grid, whereas the black lines refer to the additional combinations when a dense grid is used. For a given $m_{\text{stop},\sigma}$, all iterations $m_{\text{stop},\mu} \geq m_{\text{stop},\sigma}$ (i.e., below the bisecting line) can be computed without additional computing time. For example, if we fit a model with `mstop = c(mu = 30, sigma = 15)`, all m_{stop} combinations on the red path (Figure **??**, right) are computed. Until the point where $m_{\text{stop},\mu} = m_{\text{stop},\sigma}$, we move along the bisecting line. Then we stop increasing $m_{\text{stop},\sigma}$ and increase $m_{\text{stop},\mu}$ only, i.e., we start moving along a horizontal line. Thus, all iterations on this horizontal line are computed anyway. Note that it is quite expensive to move from the computed model to one with `mstop = c(mu = 30, sigma = 16)`. One cannot simply increase $m_{\text{stop},\sigma}$ by 1 but needs to go along the black dotted path. As the dense grid does not increase the run time (or only marginally), we recommend to always use this option, which is also the default.

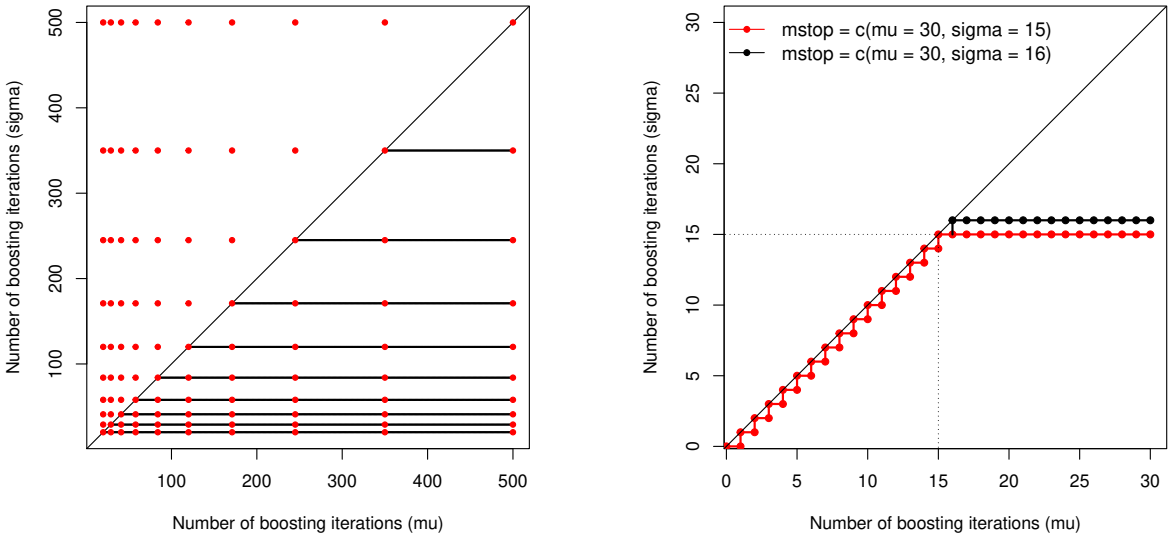

Figure 3: *Left:* Comparison between sparse grid (red) and dense μ grid (black horizontal lines in addition to the sparse red grid). *Right:* Example of the path of the iteration counts.

The `dense_mu_grid` option also works for asymmetric grids (e.g., `make.grid(max = c(mu = 100, sigma = 200))`) and for more than two parameters (e.g., `make.grid(max = c(mu = 100, sigma = 200, nu = 20))`). For an example in the latter case see the help file of `make.grid()`.

Now we use the dense grid for cross-validation (or subsampling to be more precise). The computation of the cross-validated risk using `cvrisk()` takes more than one hour on a 64-bit Ubuntu machine using 2 cores.

```
R> cores <- ifelse(grepl("linux|apple", R.Version()$platform), 2, 1)
```

```
R> if (!file.exists("cvrisk/cvr_india.Rda")) {
+      set.seed(1907)
+      folds <- cv(model.weights(mod), type = "subsampling")
+      densegrid <- make.grid(max = c(mu = 5000, sigma = 500), min = 20,
+                             length.out = 10, dense_mu_grid = TRUE)
+      cvr <- cvrisk(mod, grid = densegrid, folds = folds, mc.cores = cores)
+      save("cvr", file = "cvrisk/cvr_india.Rda", compress = "xz")
+  }
```

By using more computing cores or a larger computer cluster the speed can be easily increased. The usage of `cvrisk()` is practically identical to that of `cvrisk()` from package **mboost**. See **?** for details on parallelization and grid computing. As Windows does not support addressing multiple cores from R, on Windows we use only one core whereas on Unix-based systems two cores are used. We then load the pre-computed results of the cross-validated risk:

```
R> load("cvrisk/cvr_india.Rda")
```

◆

5.6. Methods to extract and display results

In order to work with the results, methods to extract information both from boosting models and the corresponding cross-validation results have been implemented. Fitted **gamboostLSS** models (i.e., objects of type `"mboostLSS"`) are lists of `"mboost"` objects. The most important distinction from the methods implemented in **mboost** is the widespread occurrence of the additional argument `parameter`, which enables the user to apply the function on all parameters of a fitted GAMLSS model or only on one (or more) specific parameters.

Most importantly, one can extract the coefficients of a fitted model (`coef()`) or plot the effects (`plot()`). Different versions of both functions are available for linear GAMLSS models (i.e., models of class `"glmboostLSS"`) and for non-linear GAMLSS models (e.g., models with P-splines). Additionally, the user can extract the risk for all iterations using the function `risk()`. Selected base-learners can be extracted using `selected()`. Fitted values and predictions can be obtained by `fitted()` and `predict()`. For details and examples, see the corresponding help files and **?**. Furthermore, a special function for marginal prediction intervals is available (`predint()`) together with a dedicated plot function (`plot.predint()`).

For cross-validation results (objects of class `"cvriskLSS"`), there exists a function to extract the estimated optimal number of boosting iterations (`mstop()`). The results can also be plotted using a special `plot()` function. Hence, convergence and overfitting behavior can be visually inspected.

In order to increase or reduce the number of boosting steps to the appropriate number (as e.g., obtained by cross-validation techniques) one can use the function `mstop`. If we want to reduce our model, for example, to 10 boosting steps for the `mu` parameter and 20 steps for the `sigma` parameter we can use

```
R> mstop(mod) <- c(10, 20)
```

This directly alters the object `mod`. Instead of specifying a vector with separate values for each sub-family one can also use a single value, which then is used for each sub-family (see Section **??**).

Case study (*cont'd*): Childhood malnutrition in India We first inspect the cross-validation results (see Figure **??**):

```
R> plot(cvr)
```

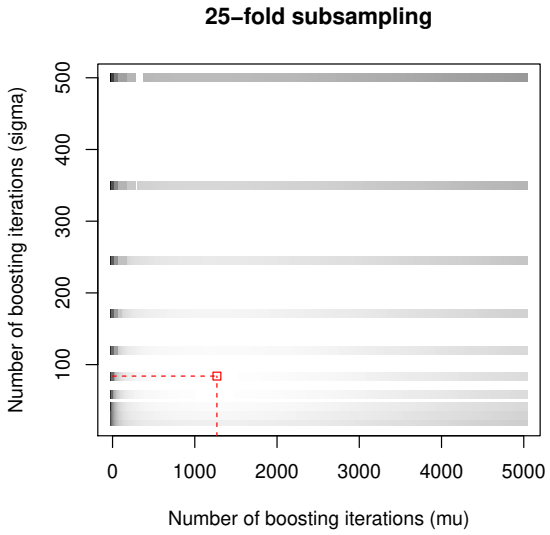

Figure 4: Cross-validated risk. Darker color represents higher predictive risks. The optimal combination of stopping iterations is indicated by dashed red lines.

If the optimal stopping iteration is close to the boundary of the grid one should re-run the cross-validation procedure with different `max` values for the grid and/or more grid points. This is not the case here (Figure **??**). To extract the optimal stopping iteration one can now use

```
R> mstop(cvr)
   mu sigma
 1269    84
```

To use the optimal model, i.e., the model with the iteration number from the cross-validation, we set the model to these values:

```
R> mstop(mod) <- mstop(cvr)
```

In the next step, the `plot()` function can be used to plot the partial effects. A partial effect is the effect of a certain predictor only, i.e., all other model components are ignored for the plot. Thus, the reference level of the plot is arbitrary and even the actual size of the effect might not be interpretable; only changes and hence the functional form are meaningful. If no further arguments are specified, all *selected* base-learners are plotted:

```
R> par(mfrow = c(2, 5))
R> plot(mod)
```

Special base-learners can be plotted using the argument `which` (to specify the base-learner) and the argument `parameter` (to specify the parameter, e.g., `"mu"`). Partial matching is used for `which`, i.e., one can specify a sub-string of the base-learners' names. Consequently, all matching base-learners are selected. Alternatively, one can specify an integer which indicates the number of the effect in the model formula. Thus

```
R> par(mfrow = c(2, 4), mar = c(5.1, 4.5, 4.1, 1.1))
R> plot(mod, which = "bbs", type = "l")
```

plots *all* P-spline base-learners irrespective if they where selected or not. The partial effects in Figure **??** can be interpreted as follows: The age of the mother seems to have a minor impact on stunting for both the mean effect and the effect on the standard deviation. With increasing BMI of the mother, the stunting score increases, i.e., the child is better nourished. At the same time the variability increases until a BMI of roughly 25 and then decreases again. The age of the child has a negative effect until the age of approximately 1.5 years (18 months). The variability increases over the complete range of age. The BMI of the child has a negative effect on stunting, with lowest variability for an BMI of approximately 16. While all other effects can be interpreted quite easily, this effect is more difficult to interpret. Usually, one would expect that a child that suffers from malnutrition also has a small BMI. However, the height of the child enters the calculation of the BMI in the denominator, which means that a lower stunting score (i.e., small height) should lead on average to higher BMI values if the weight of a child is fixed.

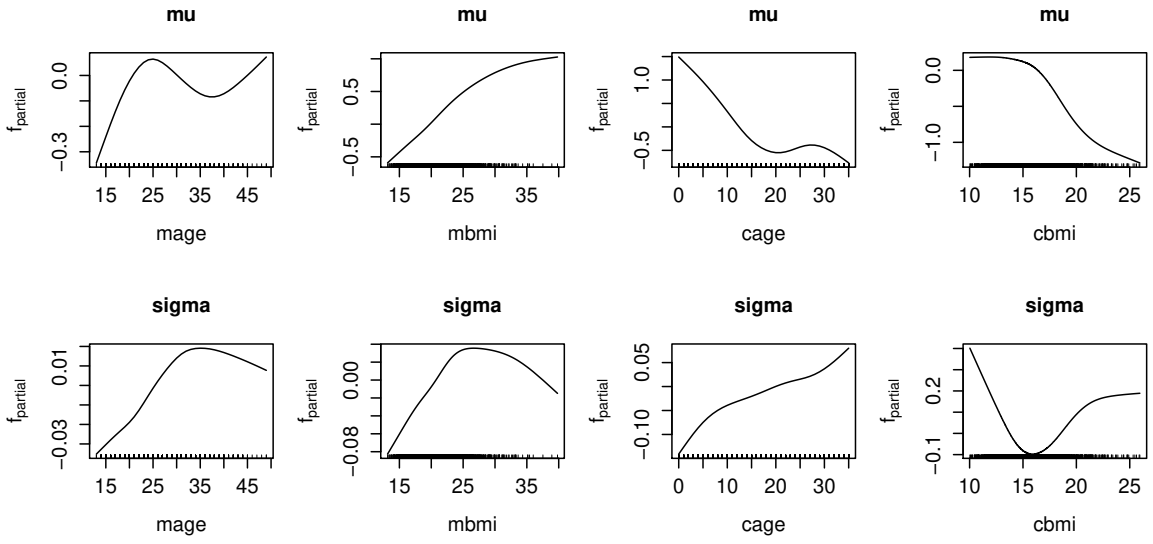

Figure 5: Smooth partial effects of the estimated model with the rescaled outcome. The effects for `sigma` are estimated and plotted on the log-scale (see Equation **??**), i.e., we plot the predictor against $\log(\hat{\sigma})$.

If we want to plot the effects of all P-spline base-learners for the μ parameter, we can use

```
R> plot(mod, which = "bbs", parameter = "mu")
```

Instead of specifying (sub-)strings for the two arguments one could use integer values in both cases. For example,

```
R> plot(mod, which = 1:4, parameter = 1)
```

results in the same plots.

Prediction intervals for new observations can be easily constructed by computing the quantiles of the conditional GAMLSS distribution. This is done by plugging the estimates of the distribution parameters (e.g., $\hat{\mu}(x_{\text{new}}), \hat{\sigma}(x_{\text{new}})$ for a new observation x_{new}) into the quantile function (**?**).

Marginal prediction intervals, which reflect the effect of a single predictor on the quantiles (keeping all other variables fixed), can be used to illustrate the combined effect of this variable on various distribution parameters and hence the shape of the distribution. For illustration purposes we plot the influence of the children's BMI via `predint()`. To obtain marginal prediction intervals, the function uses a grid for the variable of interest, while fixing all others at their mean (continuous variables) or modus (categorical variables).

```
R> plot(predint(mod, pi = c(0.8, 0.9), which = "cbmi"),
+       lty = 1:3, lwd = 3, xlab =  "BMI (child)",
+       ylab = "Stunting score")
```

To additionally highlight observations from Greater Mumbai, we use

```
R> points(stunting ~ cbmi, data = india, pch = 20,
+         col = rgb(1, 0, 0, 0.5), subset = mcdist == "381")
```

The resulting marginal prediction intervals are displayed in Figure **??**. For the interpretation and evaluation of prediction intervals, see **?**.

For the spatial `bmrf()` base-learner we need some extra work to plot the effect(s). We need to obtain the (partial) predicted values per region using either `fitted()` or `predict()`:

```
R> fitted_mu <- fitted(mod, parameter = "mu", which = "mcdist",
+                      type = "response")
R> fitted_sigma <- fitted(mod, parameter = "sigma", which = "mcdist",
+                         type = "response")
```

In case of `bmrf()` base-learners we then need to aggregate the data for multiple observations in one region before we can plot the data. Here, one could also plot the coefficients, which constitute the effect estimates per region. Note that this interpretation is not possible for for other bivariate or spatial base-learners such as `bspatial()` or `brad()`:

```
R> fitted_mu <- tapply(fitted_mu, india$mcdist, FUN = mean)
R> fitted_sigma <- tapply(fitted_sigma, india$mcdist, FUN = mean)
R> plotdata <- data.frame(region = names(fitted_mu),
+                         mu = fitted_mu, sigma = fitted_sigma)
R> par(mfrow = c(1, 2), mar = c(1, 0, 2, 0))
```

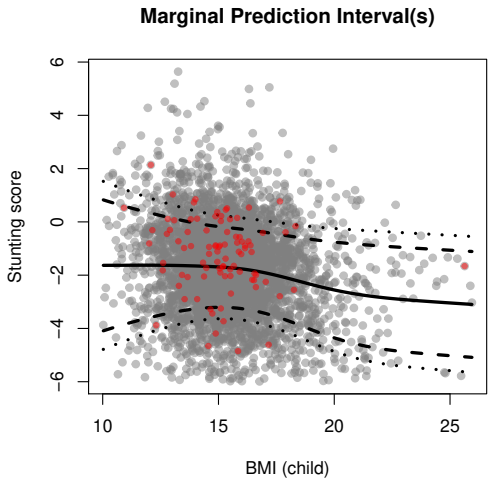

Figure 6: 80% (dashed) and 90% (dotted) marginal prediction intervals for the BMI of the children in the district of Greater Mumbai (which is the region with the most observations). For all other variables we used average values (i.e., a child with average age, and a mother with average age and BMI). The solid line corresponds to the median prediction (which equals the mean for symmetric distributions such as the Gaussian distribution). Observations from Greater Mumbai are highlighted in red.

```
R> plotmap(india.bnd, plotdata[, c(1, 2)], range = c(-0.62, 0.82),
+          main = "Mean", pos = "bottomright", mar.min = NULL)
R> plotmap(india.bnd, plotdata[, c(1, 3)], range = c(0.75, 1.1),
+          main = "Standard deviation", pos = "bottomright", mar.min = NULL)
```

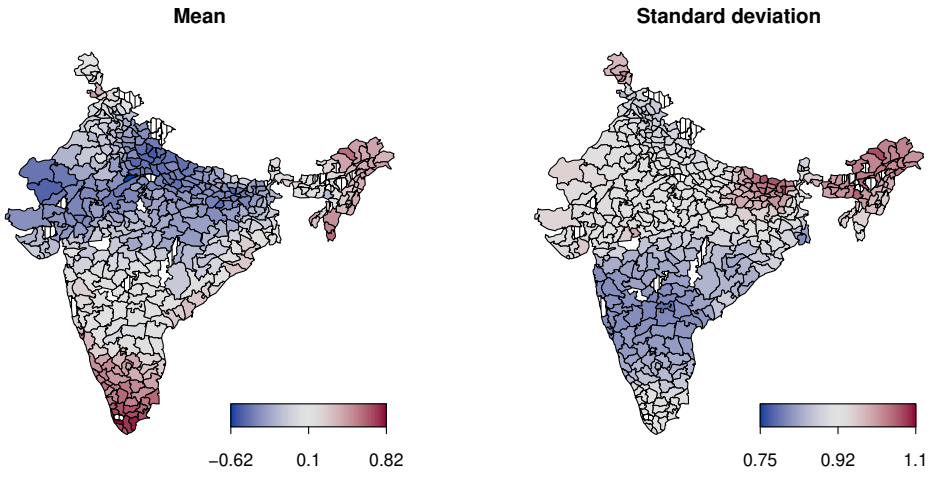

Figure 7: Spatial partial effects of the estimated model. Dashed regions represent regions without data. Note that effect estimates for these regions exist and could be extracted.

Figure **??** (left) shows a clear spatial pattern of stunting. While children in the southern regions like Tamil Nadu and Kerala as well as in the north-eastern regions around Assam

and Arunachal Pradesh seem to have a smaller risk for stunted growth, the central regions in the north of India, especially Bihar, Uttar Pradesh and Rajasthan seem to be the most problematic in terms of stunting due to malnutrition. Since we have also modeled the scale of the distribution, we can gain much richer information concerning the regional distribution of stunting: the regions in the south which seem to be less affected by stunting do also have a lower partial effect with respect to the expected standard deviation (Figure **??**, right), i.e., a reduced standard deviation compared to the average region. This means that not only the expected stunting score is smaller on average, but that the distribution in this region is also narrower. This leads to a smaller size of prediction intervals for children living in that area. In contrast, the regions around Bihar in the central north, where India shares border with Nepal, do not only seem to have larger problems with stunted growth but have a positive partial effect with respect the scale parameter of the conditional distribution as well. This leads to larger prediction intervals, which could imply a greater risk for very small values of the stunting score for an individual child in that region. On the other hand, the larger size of the interval also offers the chance for higher values and could reflect higher differences between different parts of the population. ◆

6. Summary

The GAMLSS model class has developed into one of the most flexible tools in statistical modeling, as it can tackle nearly any regression setting of practical relevance. Boosting algorithms, on the other hand, are one of the most flexible estimation and prediction tools in the toolbox of a modern statistician (**?**).

In this paper, we have presented the R package **gamboostLSS**, which provides the first implementation of a boosting algorithm for GAMLSS. Hence, beeing a combination of boosting and GAMLSS, **gamboostLSS** combines a powerful machine learning tool with the world of statistical modeling (**?**), offering the advantage of intrinsic model choice and variable selection in potentially high-dimensional data situations. The package also combines the advantages of both **mboost** (with a well-established, well-tested modular structure in the back-end) and **gamlss** (which implements a large amount of families that are available via conversion with the `as.families()` function).

While the implementation in the R package **gamlss** (provided by the inventors of GAMLSS) must be seen as the gold standard for fitting GAMLSS, the **gamboostLSS** package offers a flexible alternative, which can be advantageous, amongst others, in following data settings: (i) models with a large number of coefficients, where classical estimation approaches become unfeasible; (ii) data situations where variable selection is of great interest; (iii) models where a greater flexibility regarding the effect types is needed, e.g., when spatial, smooth, random, or constrained effects should be included and selected at the same time.

Acknowledgments

We thank the editors and the two anonymous referees for their valuable comments that helped to greatly improve the manuscript. We gratefully acknowledge the help of Nora Fenske and Thomas Kneib, who provided code to prepare the data and also gave valuable input on the package **gamboostLSS**. We thank Mikis Stasinopoulos for his support in implementing

`as.families` and Thorsten Hothorn for his great work on **mboost**. The work of Matthias Schmid and Andreas Mayr was supported by the Deutsche Forschungsgemeinschaft (DFG), grant SCHM-2966/1-1, and the Interdisciplinary Center for Clinical Research (IZKF) of the Friedrich-Alexander University Erlangen-Nürnberg, project J49.

A. The *gamboostLSS* algorithm

Let $\boldsymbol{\theta} = (\theta_k)_{k=1,\ldots,K}$ be the vector of distribution parameters of a GAMLSS, where $\theta_k = g_k^{-1}(\eta_{\theta_k})$ with parameter-specific link functions g_k and additive predictor η_{θ_k}. The *gamboostLSS* algorithm (**?**) circles between the different distribution parameters θ_k, $k = 1, \ldots, K$, and fits all base-learners $h(\cdot)$ separately to the negative partial derivatives of the loss function, i.e., in the GAMLSS context to the partial derivatives of the log-likelihood with respect to the additive predictors η_{θ_k}, i.e., $\frac{\partial}{\partial \eta_{\theta_k}} l(\boldsymbol{y}, \boldsymbol{\theta})$.

Initialize

(1) Set the iteration counter $m := 0$. Initialize the additive predictors $\hat{\eta}_{\theta_{k,i}}^{[m]}$, $k = 1, \ldots, K$, $i = 1, \ldots, n$, with offset values, e.g., $\hat{\eta}_{\theta_{k,i}}^{[0]} \equiv \underset{c}{\operatorname{argmax}} \sum_{i=1}^{n} l(y_i, \theta_{k,i} = c)$.

(2) For each distribution parameter θ_k, $k = 1, \ldots, K$, specify a set of base-learners: i.e., for parameter θ_k by $h_{k,1}(\cdot), \ldots, h_{k,p_k}(\cdot)$, where p_k is the cardinality of the set of base-learners specified for θ_k.

Boosting in multiple dimensions

(3) **Start** a new boosting iteration: increase m by 1 and set $k := 0$.

(4) (a) Increase k by 1.
If $m > m_{\text{stop},k}$ proceed to step 4(e).
Else compute the partial derivative $\frac{\partial}{\partial \eta_{\theta_k}} l(y, \boldsymbol{\theta})$ and plug in the current estimates $\hat{\boldsymbol{\theta}}_i^{[m-1]} = \left(\hat{\theta}_{1,i}^{[m-1]}, \ldots, \hat{\theta}_{K,i}^{[m-1]}\right) = \left(g_1^{-1}(\hat{\eta}_{\theta_{1,i}}^{[m-1]}), \ldots, g_K^{-1}(\hat{\eta}_{\theta_{K,i}}^{[m-1]})\right)$:

$$u_{k,i}^{[m-1]} = \frac{\partial}{\partial \eta_{\theta_k}} l(y_i, \boldsymbol{\theta}) \Big|_{\boldsymbol{\theta} = \hat{\boldsymbol{\theta}}_i^{[m-1]}}, \; i = 1, \ldots, n.$$

(b) **Fit** each of the base-learners contained in the set of base-learners specified for the parameter θ_k in step (2) to the gradient vector $\boldsymbol{u}_k^{[m-1]}$.

(c) **Select** the base-learner j^* that best fits the partial-derivative vector according to the least-squares criterion, i.e., select the base-learner h_{k,j^*} defined by

$$j^* = \underset{1 \le j \le p_k}{\operatorname{argmin}} \sum_{i=1}^{n} (u_{k,i}^{[m-1]} - h_{k,j}(\cdot))^2 \; .$$

(d) **Update** the additive predictor η_{θ_k} as follows:

$$\hat{\eta}_{\theta_k}^{[m-1]} := \hat{\eta}_{\theta_k}^{[m-1]} + \nu_{\text{sl}} \cdot h_{k,j^*}(\cdot) \; ,$$

where ν_{sl} is a small step-length ($0 < \nu_{\text{sl}} \ll 1$).

(e) Set $\hat{\eta}_{\theta_k}^{[m]} := \hat{\eta}_{\theta_k}^{[m-1]}$.

(f) **Iterate** steps 4(a) to 4(e) for $k = 2, \ldots, K$.

Iterate

(5) Iterate steps 3 and 4 until $m > m_{\text{stop},k}$ for all $k = 1, \ldots, K$.

B. Data pre-processing and stabilization of gradients

As the *gamboostLSS* algorithm updates the parameter estimates in turn by optimizing the gradients, it is important that these are comparable for all GAMLSS parameters. Consider for example the standard Gaussian distribution where the gradients of the log-likelihood with respect to η_μ and η_σ are

$$\frac{\partial}{\partial \eta_\mu} l(y_i, g_\mu^{-1}(\eta_\mu), \hat{\sigma}) = \frac{y_i - \eta_{\mu i}}{\hat{\sigma}_i^2},$$

with identity link, i.e., $g_\mu^{-1}(\eta_\mu) = \eta_\mu$, and

$$\frac{\partial}{\partial \eta_\sigma} l(y_i, \hat{\mu}, g_\sigma^{-1}(\eta_\sigma)) = -1 + \frac{(y_i - \hat{\mu}_i)^2}{\exp(2\eta_{\sigma i})},$$

with log link, i.e., $g_\sigma^{-1}(\eta_\sigma) = \exp(\eta_\sigma)$.

For small values of $\hat{\sigma}_i$, the gradient vector for μ will hence inevitably become huge, while for large variances it will become very small. As the base-learners are directly fitted to this gradient vector, this will have a dramatic effect on convergence speed. Due to imbalances regarding the range of $\frac{\partial}{\partial \eta_\mu} l(y_i, \mu, \sigma)$ and $\frac{\partial}{\partial \eta_\sigma} l(y_i, \mu, \sigma)$, a potential bias might be induced when the algorithm becomes so unstable that it does not converge to the optimal solution (or converges very slowly).

Consequently, one can use standardized gradients, where in **each step** the gradient is divided by its median absolute deviation, i.e., it is divided by

$$\text{MAD} = \text{median}_i(|u_{k,i} - \text{median}_j(u_{k,j})|), \tag{3}$$

where $u_{k,i}$ is the gradient of the kth GAMLSS parameter in the current boosting step i. If weights are specified (explicitly or implicitly as for cross-validation) a weighted median is used. MAD-stabilization can be activated by setting the argument `stabilization` to `"MAD"` in the fitting families (see example on p. **??**). Using `stabilization = "none"` explicitly switches off the stabilization. As this is the current default, this is only needed for clarity.

Another way to improve convergence speed might be to standardize the response variable (and/or to use a larger step size ν_{sl}). This is especially useful if the range of the response differs strongly from the range of the negative gradients. Both, the built in stabilization and the standardization of the response are not always advised but need to be carefully considered given the data at hand. If convergence speed is slow or if the negative gradient even starts to become unstable, one should consider one or both options to stabilize the fit. To judge the impact of these methods one can run the `gamboostLSS` algorithm using different options and compare the results via cross-validated predictive risks (see Sections **??** and **??**).

C. Additional Families

Table **??** gives an overview of common, additional GAMLSS distributions and GAMLSS distributions with a different parametrization than in **gamboostLSS**. For a comprehensive overview see the distribution tables available at `www.gamlss.org` and the documentation of the **gamlss.dist** package (**?**). Note that **gamboostLSS** works only for more-parametric distributions, while in **gamlss.dist** also a few one-parametric distributions are implemented. In this case the `as.families()` function will construct a corresponding `"boost_family"` which one can use as `family` in **mboost** (a corresponding advice is given in a warning message).

	Name	Response	μ	σ	ν	τ	Note
Continuous response							
Generalized t	`GT`	cont.	id	log	log	log	
Box-Cox t	`BCT`	cont.	id	log	id	log	
Gumbel	`GU`	cont.	id	log			For moderately skewed data.
Reverse Gumbel	`RG`	cont.	id	log			Extreme value distribution.
Continuous non-negative response (without censoring)							
Gamma	`GA`	cont. > 0	log	log			Also implemented as `GammaLSS()` [a,b].
Inverse Gamma	`IGAMMA`	cont. > 0	log	log			
Zero-adjusted Gamma	`ZAGA`	cont. ≥ 0	log	log	logit		Gamma, additionally allowing for zeros.
Inverse Gaussian	`IG`	cont. > 0	log	log			
Log-normal	`LOGNO`	cont. > 0	log	log			For positively skewed data.
Box-Cox Cole and Green	`BCCG`	cont. > 0	id	log	id		For positively and negatively skewed data.
Pareto	`PARETO2`	cont. > 0	log	log			
Box-Cox power exponential	`BCPE`	cont. > 0	id	log	id	log	Recommended for child growth centiles.
Fractions and bounded continuous response							
Beta	`BE`	$\in (0, 1)$	logit	logit			Also implemented as `BetaLSS()` [a,c].
Beta inflated	`BEINF`	$\in [0, 1]$	logit	logit	log	log	Beta, additionally allowing for zeros and ones.
Models for count data							
Beta binomial	`BB`	count	logit	log			
Negative binomial	`NBI`	count	log	log			For over-dispersed count data; also implemented as `NBinomialLSS()` [a,d].

Table 3: Overview of common, additional GAMLSS distributions that can be used via `as.families()` in **gamboostLSS**. For every modeled distribution parameter, the corresponding link-function is displayed. [a] The parametrizations of the distribution functions in **gamboostLSS** and **gamlss.dist** differ with respect to the variance. [b] `GammaLSS(mu, sigma)` has $\mathsf{VAR}(y|x) = \texttt{mu}^2/\texttt{sigma}$, and `as.families(GA)(mu, sigma)` has $\mathsf{VAR}(y|x) = \texttt{sigma}^2 \cdot \texttt{mu}^2$. [c] `BetaLSS(mu, phi)` has $\mathsf{VAR}(y|x) = \texttt{mu} \cdot (1 - \texttt{mu}) \cdot (1 + \texttt{phi})^{-1}$, and `as.families(BE)(mu, sigma)` has $\mathsf{VAR}(y|x) = \texttt{mu} \cdot (1 - \texttt{mu}) \cdot \texttt{sigma}^2$. [d] `NBinomialLSS(mu, sigma)` has $\mathsf{VAR}(y|x) = \texttt{mu} + 1/\texttt{sigma} \cdot \texttt{mu}^2$, and `as.families(NBI)(mu, sigma)` has $\mathsf{VAR}(y|x) = \texttt{mu} + \texttt{sigma} \cdot \texttt{mu}^2$.

Affiliation:

Benjamin Hofner & Andreas Mayr
Department of Medical Informatics, Biometry and Epidemiology
Friedrich-Alexander-Universität Erlangen-Nürnberg
Waldstraße 6
91054 Erlangen, Germany
E-mail: benjamin.hofner@fau.de,
andreas.mayr@fau.de
URL: http://www.imbe.med.uni-erlangen.de/cms/benjamin_hofner.html,
http://www.imbe.med.uni-erlangen.de/ma/A.Mayr/

Matthias Schmid
Department of Medical Biometry, Informatics and Epidemiology
University of Bonn
Sigmund-Freud-Straße 25
53105 Bonn
E-mail: matthias.schmid@imbie.uni-bonn.de
URL: http://www.imbie.uni-bonn.de