Journal of Statistical Software

August 2020, Volume 94, Issue 10. doi: 10.18637/jss.v094.i10

Boosting Functional Regression Models with FDboost

Sarah Brockhaus
Ludwig-Maximilans-
Universität München

David Rügamer
Ludwig-Maximilans-
Universität München

Sonja Greven
Ludwig-Maximilans-
Universität München

Abstract

The R add-on package **FDboost** is a flexible toolbox for the estimation of functional regression models by model-based boosting. It provides the possibility to fit regression models for scalar and functional response with effects of scalar as well as functional covariates, i.e., scalar-on-function, function-on-scalar and function-on-function regression models. In addition to mean regression, quantile regression models as well as generalized additive models for location scale and shape can be fitted with **FDboost**. Furthermore, boosting can be used in high-dimensional data settings with more covariates than observations. We provide a hands-on tutorial on model fitting and tuning, including the visualization of results. The methods for scalar-on-function regression are illustrated with spectrometric data of fossil fuels and those for functional response regression with a data set including bioelectrical signals for emotional episodes.

Keywords: functional data analysis, function-on-function regression, function-on-scalar regression, gradient boosting, model-based boosting, scalar-on-function regression.

1. Introduction

With the progress of technology today, we have the ability to observe more and more data of a functional nature, such as curves, trajectories or images (Ramsay and Silverman 2005). Functional data can be found in many scientific fields like demography, biology, medicine, meteorology and economics (see, e.g., Ullah and Finch 2013). In practice, the functions are observed on finite grids. In this paper, we deal with one-dimensional functional data that are observed over a real valued interval. Examples for such data are growth curves over time, acoustic signals, temperature curves and spectrometric measurements in a certain range of wavelengths. Regression models are a versatile tool for data analysis and various models have been proposed for regression with functional variables; see Morris (2015) and Greven and

Scheipl (2017) for recent reviews of functional regression models. One can distinguish between three different types of functional regression models: scalar-on-function regression, a regression with scalar response and functional covariates, function-on-scalar regression referring to models with functional response and scalar covariates and function-on-function regression, which is used when both response and covariates are functional. Models for scalar-on-function regression are sometimes also called signal regression.

Greven and Scheipl (2017) lay out a generic framework for functional regression models including the three mentioned model types. Many types of covariate effects are discussed including linear and non-linear effects of scalar covariates as well as linear effects of functional covariates and interaction terms. They describe that estimation can be based on a mixed models framework (Scheipl, Staicu, and Greven 2015; Scheipl, Gertheiss, and Greven 2016) or on component-wise gradient boosting (Brockhaus, Scheipl, Hothorn, and Greven 2015; Brockhaus, Melcher, Leisch, and Greven 2017; Brockhaus, Fuest, Mayr, and Greven 2018; Rügamer, Brockhaus, Gentsch, Scherer, and Greven 2018). In this paper, we describe the latter approach and provide a hands-on tutorial for its implementation in R (R Core Team 2020) in the comprehensive R package **FDboost** (Brockhaus, Rügamer, and Stöcker 2020) which is available from the Comprehensive R Archive Network (CRAN) at `https://CRAN.R-project.org/package=FDboost`.

Boosting estimates the model by iteratively combining simple models and can be seen as a method that conducts gradient descent (Bühlmann and Hothorn 2007). Boosting is capable of estimating models in high-dimensional data settings and implicitly does variable selection. The modeled features of the conditional response distribution can be chosen quite flexibly by minimizing different loss functions. The framework includes linear models (LMs), generalized linear models (GLMs) as well as quantile and expectile regression. Furthermore, generalized additive models for location, scale and shape (GAMLSS; Rigby and Stasinopoulos 2005) can be fitted (Mayr, Fenske, Hofner, Kneib, and Schmid 2012). GAMLSS model all distribution parameters of the conditional response distribution simultaneously depending on potentially different covariates. Brockhaus *et al.* (2018) discuss GAMLSS with scalar response and functional covariates. Stöcker, Brockhaus, Schaffer, von Bronk, Opitz, and Greven (2018) introduce GAMLSS for functional response. Due to variable selection and shrinkage of the coefficient estimates, no classical inference concepts are available for the boosted models. However, it is possible to quantify uncertainty by bootstrap (Efron 1979) and stability selection (Meinshausen and Bühlmann 2010). The main advantages of the boosting approach are the possibility to fit models in high-dimensional data settings with variable selection and to estimate not only mean regression models but also GAMLSS and quantile regression models. The main disadvantage is the lack of formal inference.

Other frameworks for flexible regression models with functional response exist. Morris and Carroll (2006) and Meyer, Coull, Versace, Cinciripini, and Morris (2015) use a basis transformations approach and Bayesian inference to model functional variables. Usually, loss-less transformations like a wavelet transformation are used. See Morris (2017) for a detailed comparison of the two frameworks.

In this tutorial, we present the R package **FDboost** (Brockhaus *et al.* 2020), which is designed to fit a great variety of functional regression models by boosting. **FDboost** builds on the R package **mboost** (Hothorn, Bühlmann, Kneib, Schmid, and Hofner 2020) for statistical model-based boosting. Thus, in the back-end we rely on a well-tested implementation. **FDboost** provides a comprehensive implementation of the most important methods for boost-

ing functional regression models. In particular, the package can be used to conveniently fit models with functional response. For effects of scalar covariates on functional responses, we provide base learners with suitable identifiability constraints. In addition, base learners that model effects of functional covariates are implemented. The package also contains functions for model tuning and for visualizing results.

As a case study for scalar-on-function regression, we use a data set on fossil fuels, which was analyzed in Fuchs, Scheipl, and Greven (2015) and Brockhaus *et al.* (2015) and is part of the **FDboost** package. In this application, the heat value of fossil fuels should be predicted based on spectral data. As a case study for function-on-scalar and function-on-function regression, we use the emotion components data set, which is analyzed in Rügamer *et al.* (2018) in the context of factor-specific historical effect estimation and which is provided in an aggregated version in **FDboost**. Note that we use both data sets as a running example to illustrate the capabilities of the package. We give a more complex example with a stronger focus on answering the underlying research question in Appendix E.

The remainder of the paper is structured as follows. We shortly review the generic functional regression model (Section 2) for scalar and for functional response. Then the boosting algorithm used for model fitting is introduced in Section 3. In Section 4, we give details on the infrastructure of the package **FDboost**. Scalar-on-function regression with **FDboost** is described in Section 5. Regression models for functional response with scalar and/or functional covariates are described in Section 6. We present possible covariate effects as well as discuss model tuning and show how to extract and display results. In Section 7, we discuss regression models that model other characteristics of the response distribution than the mean, in particular median regression and GAMLSS. In Section 8, we shortly comment on stability selection in combination with boosting. In Section 9 we comment on the computational burden of fitting models with **FDboost**. We conclude with a discussion in Section 10. The paper is structured such that the sections on functional response can be skipped if one is only interested in scalar-on-function regression.

2. Functional regression models

In Section 2.1 we first introduce a generic model for scalar response with functional and scalar covariates. Afterwards, we deal with models with functional response in Section 2.2.

2.1. Scalar response and functional covariates

Let the random variable Y be the scalar response with realization $y \in \mathbb{R}$. The covariate set $\boldsymbol{X}$ can include both scalar and functional variables. We denote a generic scalar covariate by Z and a generic functional covariate by $X(s)$, with $s \in \mathcal{S} = [S_1, S_2]$ and $S_1 < S_2$, $S_1, S_2 \in \mathbb{R}$. We assume that we observe $i = 1, \ldots, N$ data pairs $(y_i, \boldsymbol{x}_i)$, where $\boldsymbol{x}_i$ comprises the realizations z_i of scalar covariates as well as the realizations $x_i(s)$ of $X_i(s)$. In practice, $x_i(s)$ is observed on a grid of evaluation points $s_1, \ldots, s_R$, such that each curve is observed as a vector $(x_i(s_1), \ldots, x_i(s_R))^\top$. While different functional covariates may be observed on different grid points over different intervals, which is supported by **FDboost** as also the following example will show, we do no introduce additional indices here for ease of notation.

Covariate(s)	Type of effect	$h_j(x)$
Functional covariate $x(s)$	Linear functional effect	$\int_{\mathcal{S}} x(s)\beta(s)\,ds$
Scalar and functional covariate, z and $x(s)$	Linear interaction	$z\int_{\mathcal{S}} x(s)\beta(s)\,ds$
	Smooth interaction	$\int_{\mathcal{S}} x(s)\beta(z,s)\,ds$

Table 1: Overview of possible covariate effects of functional covariates, including interaction effects with scalar covariates.

We model the expectation of the response by an additive regression model

$$\mathbb{E}(Y_i|\boldsymbol{X}_i = \boldsymbol{x}_i) = h(\boldsymbol{x}_i) = \sum_{j=1}^{J} h_j(\boldsymbol{x}_i), \tag{1}$$

where $h(\boldsymbol{x}_i)$ is the additive predictor containing the additive effects $h_j(\boldsymbol{x}_i)$. Each effect $h_j(\boldsymbol{x}_i)$ can depend on one or more covariates in $\boldsymbol{x}_i$. Possible effects include linear, non-linear and interaction effects of scalar covariates as well as linear effects of functional covariates. Moreover, group-specific effects and interaction effects between scalar and functional variables are possible. To give an idea of possible effects $h_j(\boldsymbol{x})$, Table 1 lists effects of functional covariates that are currently implemented in **FDboost**. A scalar-on-function model with only one functional covariate would be $\mathbb{E}(Y_i|\boldsymbol{X}_i = \boldsymbol{x}_i) = \beta_0 + \int_{\mathcal{S}} x_i(s)\beta(s)\,ds$, see Section 5 for concrete examples of scalar-on-function models for the fossil fuel data set.

The effects $h_j(\boldsymbol{x}_i)$ are linearized using a basis representation:

$$h_j(\boldsymbol{x}_i) = \boldsymbol{b}_j(\boldsymbol{x}_i)^\top \boldsymbol{\theta}_j, \quad j = 1, \ldots, J, \tag{2}$$

with basis vector $\boldsymbol{b}_j(\boldsymbol{x}_i) \in \mathbb{R}^{K_j}$ and coefficient vector $\boldsymbol{\theta}_j \in \mathbb{R}^{K_j}$ that has to be estimated. The $N \times K_j$ design matrix for the jth effect consists of rows $\boldsymbol{b}_j(\boldsymbol{x}_i)^\top$ for all observations $i = 1, \ldots, N$. A ridge-type penalty term $\lambda_j \boldsymbol{\theta}_j^\top \boldsymbol{P}_j \boldsymbol{\theta}_j$ is used for regularization, where $\boldsymbol{P}_j$ is a suitable penalty matrix for $\boldsymbol{b}_j$ and λ_j is a non-negative smoothing parameter. The smoothing parameter controls the degrees of freedom of the effect.

Consider, for example, a linear effect of a functional covariate $\int_{\mathcal{S}} x_i(s)\beta(s)\,ds$. Using $\boldsymbol{\theta}_j = (\theta_{j1}, \ldots, \theta_{jK_j})^\top$, this effect is computed as

$$\begin{aligned}
\int_{\mathcal{S}} x_i(s)\beta(s)\,ds &\approx \int_{\mathcal{S}} x_i(s) \underbrace{\sum_{k=1}^{K_j} \phi_k(s)\theta_{jk}}_{\approx \beta(s)}\, ds \\
&\approx \sum_{r=1}^{R} \Big(\Delta(s_r) x_i(s_r) \sum_{k=1}^{K_j} \phi_k(s_r)\theta_{jk} \Big) \\
&= \sum_{k=1}^{K_j} \Big(\underbrace{\sum_{r=1}^{R} \Delta(s_r) x_i(s_r) \phi_k(s_r)}_{\text{entries in } \boldsymbol{b}_j(\boldsymbol{x}_i)}\, \theta_{jk} \Big) \\
&= \boldsymbol{b}_j(\boldsymbol{x}_i)^\top \boldsymbol{\theta}_j,
\end{aligned}$$

where first, the smooth effect $\beta(s)$ is expanded in basis functions, second, the integration is approximated by a weighted sum and, third, the terms are rearranged such that they fit into the scheme $\boldsymbol{b}_j(\boldsymbol{x}_i)^\top \boldsymbol{\theta}_j$. The basis $\boldsymbol{b}_j(\boldsymbol{x}_i)$ is thus computed as

$$\begin{aligned} \boldsymbol{b}_j(\boldsymbol{x}_i)^\top &= \left[\sum_{r=1}^{R} \Delta(s_r) x_i(s_r) \phi_1(s_r) \;\cdots\; \sum_{r=1}^{R} \Delta(s_r) x_i(s_r) \phi_{K_j}(s_r)\right] \\ &\approx \left[\int_{\mathcal{S}} x_i(s) \phi_1(s)\, ds \;\cdots\; \int_{\mathcal{S}} x_i(s) \phi_{K_j}(s)\, ds\right], \end{aligned} \tag{3}$$

with spline functions ϕ_k, $k = 1, \ldots, K_j$, for the expansion of the smooth effect $\beta(s)$ in s direction and integration weights $\Delta(s_r)$ for numerical computation of the integral. The penalty matrix $\boldsymbol{P}_j$ is chosen such that it is suitable to regularize the splines ϕ_k. In the current implementation only P-splines are readily available to estimate smooth effects. To set up a P-spline basis (Eilers and Marx 1996) for the smooth effect, ϕ_k in Equation 3 are B-splines and the penalty $\boldsymbol{P}_j$ is a squared difference matrix.

Case study: Heat value of fossil fuels

The aim of this application is to predict the heat value y of fossil fuels using spectral data (Fuchs *et al.* 2015, Siemens AG). For $N = 129$ samples, the data set contains the heat value, the percentage of humidity z_{H2O} and two spectral measurements, which can be thought of as functional variables $x_{\text{NIR}}(s_{\text{NIR}})$ observed over $\mathcal{S}_{\text{NIR}} = [250.4, 876.8]$ and $x_{\text{UV}}(s_{\text{UV}})$ observed over $\mathcal{S}_{\text{UV}} = [800.4, 2761.0]$. One spectrum is ultraviolet-visible (UVVIS), the other a near infrared spectrum (NIR). For both spectra, the observation points are not equidistant. The data set is contained in the R package **FDboost**.

```
R> library("FDboost")
R> data("fuelSubset", package = "FDboost")
R> str(fuelSubset)

List of 7
 $ heatan      : num [1:129] 26.8 27.5 23.8 18.2 17.5 ...
 $ h2o         : num [1:129] 2.3 3 2 1.85 2.39 ...
 $ nir.lambda  : num [1:231] 800 803 805 808 810 ...
 $ NIR         : num [1:129, 1:231] 0.2818 0.2916 -0.0042 -0.034 -0.1804 ...
 $ uvvis.lambda: num [1:134] 250 256 261 267 273 ...
 $ UVVIS       : num [1:129, 1:134] 0.145 -1.584 -0.814 -1.311 -1.373 ...
 $ h2o.fit     : num [1:129] 2.58 3.43 1.83 2.03 3.07 ...
```

Figure 1 shows the two spectral measurements colored according to the heat value. Predictive models for the heat values, discussed in the next sections, will include scalar-on-function terms to accommodate the spectral covariates.

2.2. Functional response

We denote the functional response by $Y(t)$, where t is the evaluation point at which the function is observed. We assume that $t \in \mathcal{T}$, where $\mathcal{T}$ is a real-valued interval $[T_1, T_2]$, for example a time-interval. All response curves can be observed on one common grid or on

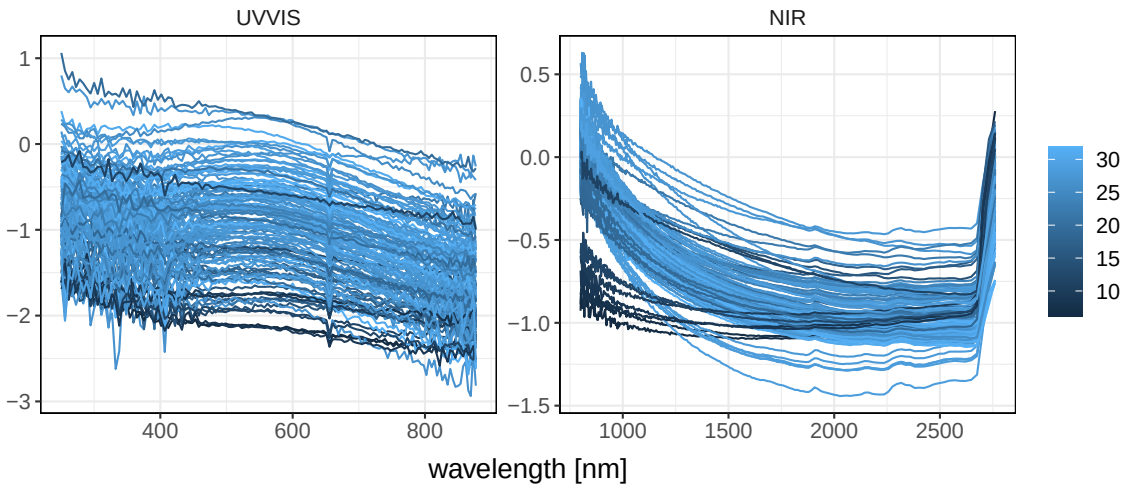

Figure 1: Spectral data of fossil fuels. Coloring of the spectral data depicts the corresponding heat value.

curve-specific grids. For responses observed on one common grid, we write $y_i(t_g)$ for the observations, with $t_g \in \{t_1, \ldots, t_G\}$ denoting the grid of evaluation points. For curve-specific evaluation points, the observations are denoted by $y_i(t_{ig})$, with $t_{ig} \in \{t_{i1}, \ldots, t_{iG_i}\}$. As above, the covariate set $\boldsymbol{X}$ can contain both scalar and functional variables.

As in model (1), we model the conditional expectation of the response. In this case, the expectation is modeled for each point $t \in \mathcal{T}$:

$$\mathbb{E}(Y_i(t)|\boldsymbol{X}_i = \boldsymbol{x}_i) = h(\boldsymbol{x}_i, t) = \sum_{j=1}^{J} h_j(\boldsymbol{x}_i, t). \tag{4}$$

As the response $Y_i(t)$ is a function of t, the linear predictor $h(\boldsymbol{x}_i, t)$ as well as the additive effects $h_j(\boldsymbol{x}_i, t)$ are functions of t. Each effect $h_j(\boldsymbol{x}_i, t)$ can depend on one or more covariates in $\boldsymbol{x}_i$ as well as on t. To give an idea of possible effects $h_j(\boldsymbol{x}_i, t)$, Table 2 lists some effects that are currently implemented. A function-on-function model with only one functional covariate would be $\mathbb{E}(Y_i|\boldsymbol{X}_i = \boldsymbol{x}_i) = \beta_0(t) + \int_S x_i(s)\beta(s,t)\,ds$. In Section 6, we give several examples for concrete models with functional response.

All effects mentioned in Table 2 are varying over t but can also be modeled as constant in t. The upper part of the table contains linear, smooth and interaction effects for scalar covariates. The middle part of the table gives possible effects of functional covariates and interaction effects between scalar and functional covariates. The lower part of the table in addition shows some group-specific effects.

In practice, all effects $h_j(\boldsymbol{x}_i, t_{ig})$ are linearized using a basis representation (Brockhaus *et al.* 2017):

$$h_j(\boldsymbol{x}_i, t_{ig}) = \boldsymbol{b}_{jY}(\boldsymbol{x}_i, t_{ig})^\top \boldsymbol{\theta}_j, \quad j = 1, \ldots, J, \tag{5}$$

where the basis vector $\boldsymbol{b}_{jY}(\boldsymbol{x}_i, t_{ig}) \in \mathbb{R}^{K_{jY}}$ depends on covariates $\boldsymbol{x}_i$ and the observation-point of the response t_{ig}. The corresponding coefficient vector $\boldsymbol{\theta}_j \in \mathbb{R}^{K_{jY}}$ has to be estimated. The design matrix for the jth effect consists of rows $\boldsymbol{b}_{jY}(\boldsymbol{x}_i, t_{ig})^\top$ for all observations $i = 1, \ldots, N$ and all time points t_{ig}, $g = 1, \ldots, G_i$.

Covariate(s)	Type of effect	$h_j(x, t)$
(None)	Smooth intercept	$\beta_0(t)$
Scalar covariate z	Linear effect	$z\beta(t)$
	Smooth effect	$f(z, t)$
Two scalars z_1, z_2	Linear interaction	$z_1 z_2 \beta(t)$
	Functional varying coefficient	$z_1 f(z_2, t)$
	Smooth interaction	$f(z_1, z_2, t)$
Functional covariate $x(s)$	Linear functional effect	$\int_{\mathcal{S}} x(s)\beta(s, t)\, ds$
Scalar z and functional $x(s)$	Linear interaction	$z \int_{\mathcal{S}} x(s)\beta(s, t)\, ds$
	Smooth interaction	$\int_{\mathcal{S}} x(s)\beta(z, s, t)\, ds$
Functional covariate $x(s)$,	Concurrent effect	$x(t)\beta(t)$
with $\mathcal{S} = \mathcal{T} = [T_1, T_2]$	Historical effect	$\int_{T_1}^{t} x(s)\beta(s, t)\, ds$
	Lag effect, with lag $\delta > 0$	$\int_{t-\delta}^{t} x(s)\beta(s, t)\, ds$
	Lead effect, with lead $\delta > 0$	$\int_{T_1}^{t-\delta} x(s)\beta(s, t)\, ds$
	Effect with t-specific integration limits $[l(t), u(t)]$	$\int_{l(t)}^{u(t)} x(s)\beta(s, t)\, ds$
Grouping variable g	Group-specific smooth intercepts	$\beta_g(t)$
Grouping variable g and scalar z	Group-specific linear effects	$z\beta_g(t)$
Curve indicator i	Curve-specific smooth residuals	$e_i(t)$

Table 2: Overview of some possible covariate effects that can be represented within the framework of functional regression.

In the following, we will use a modularization of the basis into a first part depending on covariates and a second part that only depends on t. This modular structure reduces the problem of specifying the basis $\boldsymbol{b}_{jY}(\boldsymbol{x}_i, t_{ig})$ to that of creating two suitable marginal bases. For many effects, the marginal bases are easy to define as they are known from regression with scalar response.

First, we focus on responses observed on one common grid $(t_1, \ldots, t_G)^\top$ which does not depend on i. In this case, we represent the effects using the Kronecker product $\otimes$ of two marginal bases (Brockhaus *et al.* 2015)

$$h_j(\boldsymbol{x}_i, t_g) = (\boldsymbol{b}_j(\boldsymbol{x}_i)^\top \otimes \boldsymbol{b}_Y(t_g)^\top)\boldsymbol{\theta}_j, \tag{6}$$

where the marginal basis vector $\boldsymbol{b}_j(\boldsymbol{x}_i) \in \mathbb{R}^{K_j}$, $i = 1, \ldots, N$, depends on covariates in $\boldsymbol{x}_i$ and the marginal basis vector $\boldsymbol{b}_Y(t_g) \in \mathbb{R}^{K_Y}$, $g = 1, \ldots, G$, depends on the grid point t_g. The $NG \times K_j K_Y$ design matrix is computed as the Kronecker product of the two marginal design matrices, which have dimensions $N \times K_j$ and $G \times K_Y$. If the effect can be represented as in Equation 6 it fits into the framework of linear array models (Currie, Durban, and Eilers 2006). The representation as array model has computational advantages, saving time and

memory. Brockhaus *et al.* (2015) discuss array models in the context of functional regression.

Note that the representation in Equation 6 is only possible for responses observed on one common grid, as otherwise $\boldsymbol{b}_Y(t_{ig})$ depends on the curve-specific grid points t_{ig}. In this case, the marginal bases are combined by the row-wise tensor product (Scheipl *et al.* 2015; Brockhaus *et al.* 2017). This is a rather technical detail and is thoroughly explained in Brockhaus *et al.* (2017), also for the case where the basis for the covariates depends on t_{ig} such as for historical effects.

We regularize the effects by a ridge-type penalty term $\boldsymbol{\theta}_j^\top \boldsymbol{P}_{jY} \boldsymbol{\theta}_j$. The penalty matrix for the composed basis can be constructed as (Wood 2017, Section 4.1.8)

$$\boldsymbol{P}_{jY} = \lambda_j(\boldsymbol{P}_j \otimes \boldsymbol{I}_{K_Y}) + \lambda_Y(\boldsymbol{I}_{K_j} \otimes \boldsymbol{P}_Y), \tag{7}$$

where $\boldsymbol{P}_j = [p_{j,\varkappa,\varsigma}]_{\varkappa,\varsigma \in \{1,\ldots,K_s\}}$ is a suitable penalty for $\boldsymbol{b}_j$ and $\boldsymbol{P}_Y$ is a suitable penalty for $\boldsymbol{b}_Y$. The non-negative smoothing parameters λ_j and λ_Y determine the degree of smoothing in each direction. To illustrate the resulting penalty matrix, we explicitly compute the Kronecker products in Equation 7:

$$\boldsymbol{P}_{jY} = \lambda_j \begin{bmatrix} p_{j,1,1} \cdot \boldsymbol{I}_{K_y} & \cdots & p_{j,1,K_s} \cdot \boldsymbol{I}_{K_y} \\ \vdots & \ddots & \vdots \\ p_{j,K_s,1} \cdot \boldsymbol{I}_{K_y} & \cdots & p_{j,K_s,K_s} \cdot \boldsymbol{I}_{K_y} \end{bmatrix} + \lambda_Y \begin{bmatrix} \boldsymbol{P}_Y & & \boldsymbol{0} & & \\ & \ddots & & & \\ \boldsymbol{0} & & \boldsymbol{P}_Y & & \\ & 0 & & 0 & p_{YK_tK_t} \end{bmatrix}.$$

This shows the block structure of the penalty matrix and how the two marginal penalty matrices are combined. The anisotropic penalty in Equation 7 can be simplified in the case of an isotropic penalty depending on only one smoothing parameter $\lambda_j \geq 0$:

$$\boldsymbol{P}_{jY} = \lambda_j(\boldsymbol{P}_j \otimes \boldsymbol{I}_{K_Y} + \boldsymbol{I}_{K_j} \otimes \boldsymbol{P}_Y). \tag{8}$$

In this simplified case only one instead of two smoothing parameters has to be estimated. If $\boldsymbol{P}_j = \boldsymbol{0}$ in Equation 8, this results in a penalty that only penalizes the marginal basis in t direction:

$$\boldsymbol{P}_{jY} = \lambda_j(\boldsymbol{I}_{K_j} \otimes \boldsymbol{P}_Y). \tag{9}$$

Consider, for example, a linear effect of a functional covariate $\int_{\mathcal{S}} x_i(s)\beta(s,t)\,ds$. The basis vector $\boldsymbol{b}_j(\boldsymbol{x}_i)$ and the penalty $\boldsymbol{P}_j$ are the same as in Equation 3. For the basis in t direction, we use a spline representation

$$\boldsymbol{b}_Y(t_g)^\top = [\phi_1(t_g) \ \cdots \ \phi_{K_Y}(t_g)] \tag{10}$$

with spline functions ϕ_k, $k = 1, \ldots, K_Y$ and the penalty matrix $\boldsymbol{P}_Y$ has to be chosen such that it is suitable for the chosen spline basis. Using P-splines again, ϕ_k are B-splines and $\boldsymbol{P}_Y$ is a squared difference matrix (Eilers and Marx 1996). The complete basis is

$$\boldsymbol{b}_j(\boldsymbol{x}_i)^\top \otimes \boldsymbol{b}_Y(t_g)^\top = \left[\int_{\mathcal{S}} x_i(s)\phi_1(s)\,ds \ \cdots \ \int_{\mathcal{S}} x_i(s)\phi_{K_j}(s)\,ds\right] \otimes [\phi_1(t_g) \ \cdots \ \phi_{K_Y}(t_g)].$$

This choice expands $\beta(s,t)$ in a tensor-product spline basis and approximates the integral using numerical integration.

For this effect, the penalty matrix from Equation 7 ensures smoothness of $\beta(s,t)$ in s and in t direction.

Case study: Emotion components data with EEG and EMG

The emotion components data set is based on a study of Gentsch, Grandjean, and Scherer (2014), in which brain activity (EEG) as well as facial muscle activity (EMG) was simultaneously recorded during a computerized game. As the facial muscle activity should be traceable to the brain activity for a certain game situation, Rügamer *et al.* (2018) analyzed the synchronization of EEG and EMG signal using function-on-function regression models with factor-specific historical effects. During the gambling rounds, three binary game conditions were varied, resulting in a total of 8 different study settings:

- the goal conduciveness (`game_outcome`) corresponding to the monetary outcome (`gain` or `loss`) at the end of each game round,
- the `power` setting, which determined whether the player was able or not able to change the final outcome in her favor (`high` or `low`, respectively) and,
- the `control` setting, which was manipulated to change the participant's subjective feeling about her ability to cope with the game outcome. The player was told to frequently have high power in rounds with `high` control and have frequently low power in `low` control situations.

We focus on the EMG of the frontalis muscle, which is used to raise the eyebrow. The EMG signal is a functional response $Y(t)$, with $t \in \mathcal{T} = [0, 1560]$ ms, which is measured at a frequency of 256 Hz resulting in 384 equidistant observed time points given by the vector `t`. The experimental conditions are scalar covariates. The EEG signal $x_{\text{EEG}}(s)$ is observed over the same time interval as the EMG signal. We use the EEG signal from the Fz electrode, which is in the center front of the head.

In the following, we consider an aggregated version of the data, in which the EEG and EMG signals are aggregated per subject and game condition. One participant is excluded, yielding $N = 23$ subjects.

```
R> data("emotion", package = "FDboost")
R> str(emotion)

List of 8
 $ power       : Factor w/ 2 levels "high","low": 1 1 2 2 1 1 2 2 1 1 ...
 $ game_outcome: Factor w/ 2 levels "gain","loss": 1 2 1 2 1 2 1 2 1 2 ...
 $ control     : Factor w/ 2 levels "high","low": 1 1 1 1 2 2 2 2 1 1 ...
 $ subject     : Factor w/ 23 levels "1","2","3","4",..: 1 1 1 1 1 1 1 1 ...
 $ EEG         : num [1:184, 1:384] -0.14 0.303 -0.715 0.7 0.11 ...
 $ EMG         : num [1:184, 1:384] -2.56 -4.06 -1.15 4.11 8.09 ...
 $ s           : int [1:384] 1 2 3 4 5 6 7 8 9 10 ...
 $ t           : int [1:384] 1 2 3 4 5 6 7 8 9 10 ...
```

In order to fit simple and meaningful models for function-on-function regression, we define a subset of the data that contains only the observations for a certain game condition. We use the game condition with high control, gain and low power:

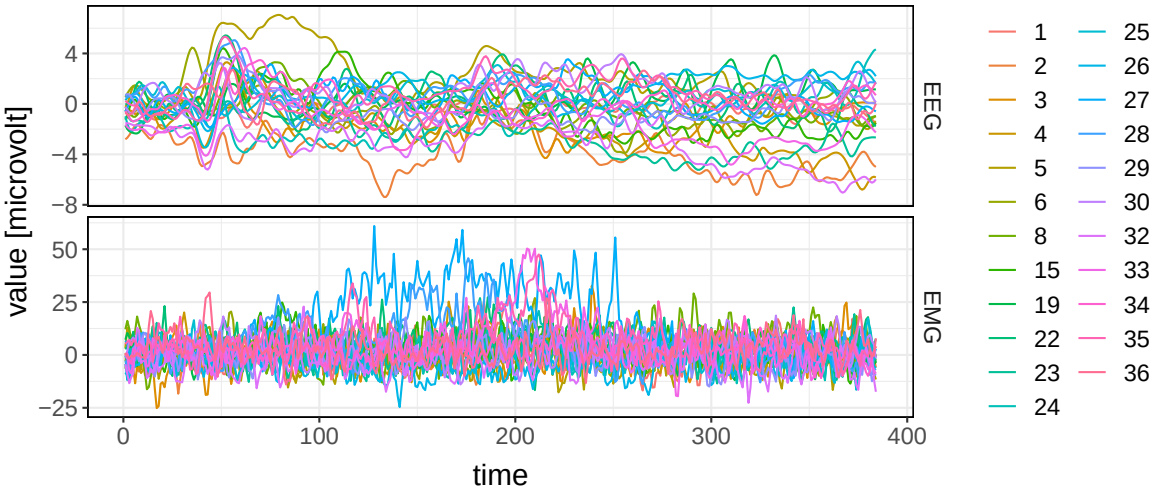

Figure 2: EEG signal (Fz electrode) and EMG signal (frontalis muscle) for each of the 23 participants (line colors) and the chosen game condition.

```
R> subset <- with(emotion, control == "high" & game_outcome == "gain" &
+    power == "low")
R> emotionHGL <- list()
R> emotionHGL$subject <- emotion$subject[subset]
R> emotionHGL$EMG <- emotion$EMG[subset, ]
R> emotionHGL$EEG <- emotion$EEG[subset, ]
R> emotionHGL$s <- emotionHGL$t <- emotion$t
```

In Figure 2 the EEG and EMG signal is depicted for each of the 23 participants and the 384 observation points.

3. Estimation by gradient boosting

Initially, boosting was proposed as a technique to iteratively improve the predictive performance of simple models or *base learners* (Ridgeway 1999). Boosting was soon recognized as a model fitting technique for statistical applications. Based on the idea of Friedman (2001), Bühlmann and Hothorn (2007) proposed the model-based boosting framework, which allows for a component-wise fitting of additive terms in the linear predictor and can handle complex additive effects. Many boosting algorithms, which are purely used for prediction, fit a rather simple model using all covariates. In contrast, in model-based boosting it is possible to define the effects of each covariate separately in different base learners. By iteratively selecting only one base learner at a time, model-based boosting performs variable selection as base learners that are never selected for the model update are excluded from the model. This framework is implemented in the **mboost** package. In contrast to other implementations of gradient boosting, such as **gbm** (Greenwell, Boehmke, Cunningham, and GBM Developers 2019), the focus of model-based boosting lies in estimating an interpretable additive structure rather than aiming at optimal predictive performance.

Component-wise gradient boosting minimizes the expected loss (risk) via gradient descent

in a stepwise procedure. In each boosting step, each base learner is fitted separately to the negative gradient and only the best fitting base learner is selected for the model update; hence the term "component-wise". To fit a model for the expectation, like the models in Equations 1 and 4, the squared error loss (L_2 loss) is minimized. In this case, the negative gradient corresponds to the residuals.

Resulting estimation and prediction performance of boosting depends on different tuning parameters, namely the *number of boosting iterations* m_{stop}, the *step-length* ν, and the specification of the base learners, e.g., whether a continuous covariate has a linear or smooth effect and the set-up of spline functions and penalties for smooth effects. We will give guidance on the choice of these parameters in the following by briefly describing the functionality of the algorithm.

The most important tuning parameter of boosting is the number of boosting iterations, as the algorithm is usually stopped before convergence. This so-called early stopping leads to regularized effect estimates and therefore yields more stable predictions. Since some of the base learners are never selected in the course of all iterations, boosting also performs variable selection. The optimal stopping iteration can be determined by methods like cross-validation, sub-sampling or bootstrap. For each fold, the empirical out-of-bag risk is computed and the stopping iteration that yields the lowest empirical risk is chosen. As resampling must be conducted on the level of independent observations, this is done on the level of curves for functional response.

In order to avoid overshooting the minimum of the loss function in each iteration, only a small step in the chosen direction is made. The length of the update is determined by the step-length ν. Some boosting frameworks adapt the choice of the step-length in each iteration. Bühlmann and Hothorn (2007) show that the estimation performance is barely affected by setting ν to a fixed and sufficiently small value for all iterations. They there propose to use a fixed step-length in the range 0.01 to 0.1. The appropriate size of the step-length depends on the loss that is minimized. In practice, the default value $\nu = 0.1$ works well for most applications when the model is specified using the L_2 loss. A smaller step-length than 0.01 is sometimes needed for loss functions, which result in discontinuous gradients, such as the check-function for quantile regression (Fenske, Kneib, and Hothorn 2011) or for loss functions, which can result in infinite pseudo-residuals (gradients), such as the Poisson likelihood loss. Since base-learner-specific tuning parameters are fixed for all iterations, the model fit is determined by the number of iterations for a given step-length.

By representing all base learners as linear effects of covariates (if necessary, by using a basis representation for non-linear effects), base learners also define the covariate effects in the sense of additive regression models and can be associated with a specific hat matrix as well as a certain number of degrees of freedom.

The degrees of freedom for each base learner and other base-learner-specific tuning parameters have an influence on the prediction and estimation performance. The degrees of freedom df_j for each base learner $j = 1, \ldots, J$ – not to be confused with the *effective degrees of freedom* for each model term in the final model – determine the flexibility of each base learner prior to the model fit. In the model-based boosting framework each base learner is fitted to the pseudo-residuals using a (penalized) least squares fit with fixed smoothing parameter λ_j, which is determined via the pre-specified degrees of freedom. Whereas defining a fixed smoothness for each model term prior to the model fit might seem restrictive at first sight, the final

smoothness of each model term is in fact determined through the number of iterations in which the respective base learner is chosen. The *effective degrees of freedom* for each smooth component after the model fit are accumulated over the iterations where the model term is selected and typically differ from the initially specified df_j. The model fit can thus adapt even to relatively complex functions by repeatedly selecting and updating a particular model term (cf. Brockhaus *et al.* 2015). Determining the smoothness through the number of iterations works well in practice and allows for a closed-form solution of the penalized least squares fit in each update. As boosting chooses base learners in a greedy manner, selection in each step is biased towards more flexible base learners with higher degrees of freedom, if base learners exhibit different degrees of freedom. This is due to the fact that these base learners more likely yield larger improvements of the fit in each iteration (see Hofner, Hothorn, Kneib, and Schmid 2011, for details). For parameter estimation quality, it is essential to facilitate a fair base learner selection in each step (Hofner *et al.* 2011). It is recommended to set df_j to an equal and rather small number for all base learners $j = 1, \ldots, J$ (Kneib, Hothorn, and Tutz 2009; Hofner *et al.* 2011). In the case of scalar-on-function regression, fulfilling this constraint is not straightforward as functional covariates must usually be incorporated with more than one degree of freedom whereas scalar linear effects are restricted to have one degree of freedom. In order to maintain a fair base learner selection, more complex effects can be orthogonalized such that they represent deviations from less complex effects. For example, a smooth effect can be centered around its linear effect, thereby allowing both terms to have one degree of freedom. In Section 4.3 as well as in Appendix E different examples demonstrate how to facilitate a fair selection in this respect.

Due to the nature of the algorithm, other base-learner-specific tuning parameters are also defined prior to the model fit and kept fixed over the iterations. The number of knots is of primary interest for functional or smooth predictors and should be chosen considering a trade-off between computing time and flexibility of each base learner. Per default, 10 knots are used, which can be rather large for some applications, but allows for a large flexibility of the estimated effects. The number of knots can be decreased if computing time is a concern. Moreover, due to the smoothness penalty, with the default penalizing deviations from linearity for smooth functions, users need not to be concerned about overfitting when increasing the number of knots.

Functional response

To adapt boosting for a functional response, we compute the loss at each point t and integrate it over the domain of the response $\mathcal{T}$ (Brockhaus *et al.* 2015).

For the L_2 loss the optimization problem for functional response aims at minimizing

$$\sum_{i=1}^{N} \int \left[y_i(t) - h(\boldsymbol{x}_i, t) \right]^2 dt, \tag{11}$$

which is approximated by numerical integration. To obtain identifiable models, suitable identifiability constraints for the base learners are necessary and implemented. **FDboost** also contains base learners that model the effects of functional covariates. For a discussion of both points, please see Brockhaus *et al.* (2015).

4. The package FDboost

Fitting functional regression models via boosting is implemented in the R package **FDboost**. The package uses the fitting algorithm and other infrastructure from the R package **mboost** (Hothorn *et al.* 2020). All base learners and distribution families that are implemented in **mboost** can be used within **FDboost**. Many naming conventions and methods in **FDboost** are implemented in analogy to **mboost**. A tutorial for **mboost** can be found in Hofner, Mayr, Robinzonov, and Schmid (2014). We will mention all features of **mboost** that are important when working with **FDboost** in the following.

4.1. Main fitting function and its arguments

The main fitting function to estimate functional regression models, like the models in Equations 1 and 4, is called `FDboost()`. The interface of `FDboost()` is as follows:[1]

```
FDboost(formula, timeformula, id = NULL, numInt = "equal", data,
  offset = NULL, ...)
```

First, we focus on the arguments that are necessary for regression models both with scalar and with functional response. `formula` specifies the base learners for the covariate effects $\boldsymbol{b}_j$ and `timeformula` specifies $\boldsymbol{b}_Y$, which is the basis along t. Per default, this basis $\boldsymbol{b}_Y$ is the same for all effects $j = 1, \ldots, J$. To specify different base learners along t, it is necessary to set up the Kronecker product of two base learners explicitly in `formula`. For a detailed explanation, we refer to Appendix C. The data is provided in the `data` argument as a '`data.frame`' or a named '`list`'. The `data` object has to contain the response, all covariates and the evaluation points of functional variables. Prior to the model fit, an offset is subtracted from the response to center it. This corresponds to initializing the fit with this offset, e.g., an overall average, and leads to faster convergence and better stability of the boosting algorithm. For mean regression, the default `offset = NULL` implies that the offset is the smoothed pointwise mean of the response over time without taking into account covariates. This offset is part of the intercept and corresponds to an initial estimate that is then updated. In the dots argument, `...`, further arguments passed to `mboost()` and `mboost_fit()` can be specified. The most important argument is `family` determining the loss- and link-function for the model fit. The default is `family = Gaussian()`, which minimizes the squared error loss and uses the identity as link function. Thus, per default a mean regression model for continuous response is fitted. For the duality of loss-function and the `family` argument, we refer to Section 7. Further important arguments are `control`, which determines the number of boosting iterations and the step-length ν of the boosting algorithm specified by `nu`. The argument `control` must be supplied as a call to the function `boost_control()`. For example, `control = boost_control(mstop = 100, nu = 0.1)` implies 100 boosting iterations and step-length $\nu = 0.1$, which also corresponds to the default settings. Note that while 100 iterations are the default chosen to avoid a computationally expensive default, this might not be sufficient and should be chosen appropriately for the given application.

FDboost allows for (tensor product) spline or functional principle component bases, but user-specified base learners allow for possible extensions (see, e.g., Hofner *et al.* 2014). Although

[1]Note that for the presentation of functions we restrict ourselves to the most important function arguments. For the full list of arguments, we refer to the corresponding help pages.

the package only provides base learners with ridge- or L_2-type penalization, model selection as facilitated by an L_1-penalty is achieved by early stopping of the algorithm. Dependent functions can be modeled by including regularized cluster-specific functional intercepts or smooth temporal / spatial effects.

4.2. Specification for scalar response

For scalar response, we set `timeformula = NULL` as no expansion of the effects in t direction is necessary. `formula` specifies the base learners for the covariates effects $\boldsymbol{b}_j$ as in Equation 2. The arguments `id` and `numInt` are only needed for functional responses. For scalar response, `offset = NULL` results in a default offset, as, for example, the overall mean for mean regression or the overall median for median regression.

4.3. Arguments needed for functional response

For functional response, the set-up of the covariate effects generally follows Equation 6 by separating the effects into two marginal parts. The marginal effects $\boldsymbol{b}_j$, $j = 1, \ldots, J$, are represented in the `formula` as `y ~ b_1 + b_2 + ...+ b_J`. The marginal effect $\boldsymbol{b}_Y$ is represented in the `timeformula`, which has the form `~ b_Y`. The base learners for the marginal effects also contain suitable penalty matrices. Internally, the base learners specified in `formula` are combined with the base learner specified in `timeformula` as in Equation 6 and a suitable penalty matrix is constructed according to Equation 8. Per default, the response is expected to be a matrix. In this case `id = NULL`. The matrix representation is not possible for a response which is observed on curve-specific grids. In this case the response is provided as vector in long format and `id` specifies which position in the vector is attributed to which curve; see Section 6 for details. The argument `numInt` provides the numerical integration scheme for computing the integral of the loss over $\mathcal{T}$ in Equation 11. Per default, `numInt = "equal"`, and thus all integration weights are set to one; for `numInt = "Riemann"` Riemann sums are used. For functional response, `offset = NULL` induces a smooth offset varying over t. The offset is estimated by adaptive splines using R package **mgcv** (Wood 2011). The argument `offset_control = o_control()` allows to control the smoothness of the estimated offset by changing the dimension of its basis representation. It is important that the offset is not too wiggly as an overfitted offset will lead to an overfitted intercept in the model. It is less problematic if the offset is underfitting the given data as the intercept in the model can be updated by the corresponding base learner to account for potential structure in the data. A visual inspection of the estimated offset can thus be helpful to determine the amount of smoothness for the given data. Alternatively, a suitable value for `k_min` can be found by comparing the performance of models with different offset specification on a validation data set or based on their out-of-bag risk. For `offset = "scalar"`, a scalar offset is computed. For functional response, this corresponds to an offset that is constant along t. Instead of fitting the offset within **FDboost**, it is also possible to provide the offset as a vector via the argument `offset`.

For more details and the full list of arguments, see the manual of `FDboost()`.

5. Scalar response and functional covariates

In this section, we give details on models with scalar response and functional covariates like the model in Equation 1. Such models are called scalar-on-function regression models. As case study the data on fossil fuels is used.

Additive predictor $h(\boldsymbol{x}) = \sum_j h_j(\boldsymbol{x})$	Call
$\beta_0 + \int_{\mathcal{S}} x(s)\beta_1(s)\,ds$	`y ~ 1 + bsignal(x, s = s)`
	`y ~ 1 + bfpc(x, s = s)`
$\beta_0 + z\beta_1 + \int_{\mathcal{S}} x(s)\beta_2(s)\,ds$	`y ~ 1 + bolsc(z) + bsignal(x, s = s)`
$+z \int_{\mathcal{S}} x(s)\beta_3(s)\,ds$	`+ bsignal(x, s = s) %X% bolsc(z)`

Table 3: Additive predictors for scalar-on-function regression models.

5.1. Potential covariate effects: Base learners

In order to fit a scalar-on-function model as in Equation 1, the `timeformula` is set to `NULL` and potential covariate effects $h_j(\boldsymbol{x}_i)$ are specified in the `formula` argument. The effects of scalar covariates can be linear or non-linear. A linear effect $z\beta$ for the covariate z is obtained using the base learner `bols(z)`, which is also suitable for factor variables, in which case dummy variables are constructed for each factor level (Hofner *et al.* 2014). Per default, `bols()` contains an intercept. If the specified degrees of freedom are less than the number of columns in the design matrix, `bols()` penalizes the linear effect by a ridge penalty with the identity matrix as penalty matrix. The base learner `brandom()` for factor variables sets up an effect, which is centered around zero and is penalized by a ridge penalty, having similar properties to a random effect, but no underlying distributional assumption. It is not possible to estimate random effects in the classical sense such that they are estimated using variance parameters. See the web appendix of Kneib *et al.* (2009) for a discussion on `brandom()`. The ridge penalized effects, however, have a similar interpretation as random effects as a quadratic penalty is mathematically equivalent to a Gaussian prior. Note that this also allows for other types of random effects such as cluster-specific random effect functions. A non-linear effect expanded by P-splines is obtained by the base learner `bbs()`. Within `bbs()`, the argument `knots` determines the number of knots of the P-spline basis, `degree` specifies the degree of the spline basis and `differences` the order of the differences in the penalty matrix. Per default, cubic B-splines on 20 knots with a second order difference penalty are used. Those settings imply rather smooth effects. In order to capture more irregular features in functional observations such as spikes or drops, it is recommended to increase the number of knots and/or only use first-order differences in the penalty matrix. Using no penalty at all (by setting `lambda = 0`) also facilitates to recover features in even more spiky data, but may be difficult in terms of unbiased base learner selection (see, e.g., Section 6.4). Note that increasing the number of knots will increase the run-time and memory consumption (see Section 9 for more details on computational burden). For more details on base learners with scalar covariates, we refer to Hofner *et al.* (2014).

Potential base learners for functional covariates can be seen in Table 3. In this table exemplary linear predictors are listed in the left column. In the right column, the corresponding call to `formula` is given. Because of the scalar response, the call to `timeformula` is set to `NULL`. For simplicity, only one possible parameterization which leads to simple interpretations and one corresponding model call are shown, although **FDboost** allows to specify several parameterizations.

For a linear effect of a functional covariate $\int_{\mathcal{S}} x(s)\beta_1(s)\,ds$, two base learners exist that use

different basis expansions. Assuming $\beta_1(s)$ to be smooth, `bsignal()` uses a P-spline representation for the expansion of $\beta_1(s)$. In this case, the observations $x(s)$ are used directly without any basis representation. Assuming that the main modes of variation in the functional covariate are the important directions for the coefficient function $\beta_1(s)$, a representation with functional principal components is suitable (Ramsay and Silverman 2005). In the base learner `bfpc()`, the coefficient function $\beta_1(s)$ and the functional covariate $x(s)$ are both represented by an expansion in the estimated functional principal components of $x(s)$. As penalty matrix, the identity matrix is used. In Appendix B, technical details on the representation of functional effects are given.

The specification of a model with an interaction term between a scalar and a functional covariate is given at the end of Table 3. The interaction term is centered around the main effect of the functional covariate using `bolsc` for the scalar covariate (as is the linear effect of the scalar covariate around the intercept). Thus, the main effect of the functional covariate has to be included in the model. For more details on interaction effects, we refer to Brockhaus *et al.* (2015) and Rügamer *et al.* (2018). The interaction is formed using the operator `%X%` that builds the row-wise tensor product of the two marginal bases, see Appendix C.

As explained in Section 3, all base learners in a model should have equal and rather low degrees of freedom. The number of degrees of freedom that can be given to a base learner is restricted. On the one hand, the maximum number is bounded by the number of columns of the design matrix (more precisely by the rank of the design matrix). On the other hand, for rank-deficient penalties, the minimum number of degrees of freedom is given by the rank of the null space of the penalty matrix.

The interface of `bsignal()` is as follows:

```
bsignal(x, s, knots = 10, degree = 3, differences = 1, df = 4,
  lambda = NULL, check.ident = FALSE)
```

The arguments `x` and `s` specify the name of the functional covariate and the name of its argument. `knots` gives the number of inner knots for the P-spline basis, `degree` the degree of the B-splines and `differences` the order of the differences that are used for the penalty. Thus, per default, 14 cubic P-splines with first order difference penalty are used. The argument `df` specifies the number of degrees of freedom for the effect and `lambda` the smoothing parameter. Only one of those two arguments can be supplied. If `check.ident = TRUE` identifiability checks proposed by Scheipl and Greven (2016) for functional linear effects are additionally performed.

The interface of `bfpc()` is:

```
bfpc(x, s, df = 4, lambda = NULL, pve = 0.99, npc = NULL)
```

The arguments `x, s, df` and `lambda` have the same meaning as in `bsignal()`. The two other arguments allow to control how many functional principal components are used as basis. Per default the number of functional principal components is chosen such that the proportion of the explained variance is 99%. This proportion can be changed using the argument `pve` (proportion variance explained). Alternatively, the number of components can be set to a specific value using `npc` (number principal components).

The interface of `bolsc()` is very similar to that of `bols()`, which is laid out in detail in

Hofner *et al.* (2014). In contrast to `bols()`, `bolsc()` centers the design matrix such that the resulting linear effect is centered around zero. More details on `bolsc()` are given in Section 6.

```
bolsc(..., df = NULL, lambda = 0, K = NULL)
```

In the dots argument, `...`, one or more covariates can be specified. For factor variables `bolsc()` sets up a design matrix in dummy-coding. The arguments `df` and `lambda` have the same meaning as above. If `lambda` > 0 or `df` $<$ the number of columns of the design matrix a ridge penalty is applied. Per default, `K = NULL`, the penalty matrix is the identity matrix. Setting the argument `K` to another matrix allows for customized penalty matrices.

Case study (continued): Fossil fuel data

For the heat values Y_i, $i = 1, \ldots, 129$, we fit the model

$$\mathbb{E}(Y|\boldsymbol{x}) = \beta_0 + f(z_{\text{H2O}}) + \int_{\mathcal{S}_{\text{NIR}}} x_{\text{NIR}}(s_{\text{NIR}})\beta_{\text{NIR}}(s_{\text{NIR}})\, ds_{\text{NIR}} + \int_{\mathcal{S}_{\text{UV}}} x_{\text{UV}}(s_{\text{UV}})\beta_{\text{UV}}(s_{\text{UV}})\, ds_{\text{UV}}, \quad (12)$$

with water content z_{H2O} and centered spectral curves x_{NIR} and x_{UV}, which are observed over the wavelengths $s_{\text{NIR}} \in \mathcal{S}_{\text{NIR}}$ and $s_{\text{UV}} \in \mathcal{S}_{\text{UV}}$. We center the NIR and the UVVIS measurement per wavelength such that $\sum_{i=1}^{N} x_{\text{NIR},i}(s_{\text{NIR}}) = 0 \, \forall s_{\text{NIR}}$ and analogously for UVVIS. Thus, the functional effects have mean zero, $\sum_{i=1}^{N} \int_{\mathcal{S}_{\text{NIR}}} x_{\text{NIR},i}(s_{\text{NIR}})\beta(s_{\text{NIR}})\, ds_{\text{NIR}} = 0$ and analogously for UVVIS. This does not affect the interpretation of $\beta_{\text{NIR}}(s_{\text{NIR}})$ and $\beta_{\text{UV}}(s_{\text{UV}})$, it only changes the interpretation of the intercept of the regression model. If all effects are centered, the intercept can be interpreted as overall mean and the other effects as deviations from the overall mean.

Note that the functional covariates have to be supplied as <number of curves> by <number of evaluation points> matrices. The non-linear effect of the scalar variable H2O is specified using the `bbs()` base learner. For the linear functional effect of NIR and UVVIS, we use the base learner `bsignal()`. The degrees of freedom are set to 4 for each base learner. For the functional effects, we use a P-spline basis with 20 inner knots. Because of the scalar response `timeformula = NULL`.

```
R> fuelSubset$UVVIS <- scale(fuelSubset$UVVIS, scale = FALSE)
R> fuelSubset$NIR <- scale(fuelSubset$NIR, scale = FALSE)
R> sof <- FDboost(heatan ~ bbs(h2o, df = 4) +
+    bsignal(UVVIS, s = uvvis.lambda, knots = 20, df = 4) +
+    bsignal(NIR, s = nir.lambda, knots = 20, df = 4),
+    timeformula = NULL, data = fuelSubset)
```

5.2. Model tuning and early stopping

Boosting iteratively selects base learners to update the additive predictor. Fixing the base learners and the step-length, the model complexity is controlled by the number of boosting iterations. With more boosting iterations the model becomes more complex (Bühlmann and Yu 2003). The step-length ν is chosen sufficiently small in the interval $(0, 1]$, usually as $\nu = 0.1$, which is also the default. For smaller step-length, more boosting iterations are required and vice versa (Friedman 2001). Note that the default number of boosting iterations is 100. This is arbitrary and in most cases not adequate. The number of boosting iterations and the

step-length of the algorithm can be specified in the argument `control`. This argument must be supplied as a call to `boost_control()`. For example, `control = boost_control(mstop = 50, nu = 0.2)` implies 50 boosting iterations and step-length $\nu = 0.2$.

The most important tuning parameter is the number of boosting iterations. For regression with scalar response, the `cvrisk()` method for '`FDboost`' objects can be used to determine the optimal stopping iteration. This function directly calls the `cvrisk()` method for '`mboost`' objects from the **mboost** package, which performs an empirical risk estimation using a specified resampling method. The interface of the `cvrisk()` method for '`FDboost`' objects is:

```
cvrisk(object,
  folds = cvLong(id = object$id, weights = model.weights(object)),
  grid = 1:mstop(object))
```

In the argument `object`, the fitted model object is specified. `grid` defines the grid on which the optimal stopping iteration is searched. Per default the grid from 1 to the current stopping iteration of the model object is used as search grid. But it is also possible to specify a larger grid, e.g., `1:5000`. The argument `folds` expects an integer weight matrix with dimension $N \times k$ (<number of observations> times <number of folds>). Depending on the range of values in the weight matrix, different types of resampling are performed. For example, if the weights sum to N for each column but also have values larger than one, the resampling scheme corresponds to bootstrap while a k-fold cross-validation is employed by using an incidence matrix, for which the rows sum to $k - 1$. If not manually specified, **mboost** and **FDboost** provide convenience functions – `cv()` and `cvLong()` – that construct such matrices on the basis of the given model object. The function `cvLong()` is suited for functional response and treats scalar response as the special case with one observation per curve. For scalar response, the function `cv()` from package **mboost** can be used, which has a simpler interface.

```
cv(weights, type = c("bootstrap", "kfold", "subsampling"),
  B = ifelse(type == "kfold", 10, 25))
```

The argument `weights` is used to specify the weights of the original model, which can be extracted using `model.weights(object)`. Usually all model weights are one. Via argument `type` the resampling scheme is defined: `"bootstrap"` for non-parametric bootstrap, `"kfold"` for cross-validation and `"subsampling"` for resampling half of all observations for each fold. The number of folds is defined by `B`. Per default, 10 folds are used for cross-validation and 25 folds for bootstrap as well as for subsampling.

The function `cvLong()` is especially suited for functional response and has the additional argument `id`, which is used to specify which observations belong to the same response curve. For scalar response, `id = 1:N`.

Case study (continued): Fossil fuel data

To tune the scalar-on-function regression model (12), we search the optimal stopping iteration by 10-fold bootstrapping. First, the bootstrap folds are created using the function `cv()`. Second, for each bootstrap fold, the out-of-bag risk is computed for models with 1 to 1000 boosting iterations using the `cvrisk` function. The choice of the grid is independent of the number of boosting iterations of the fitted model object.

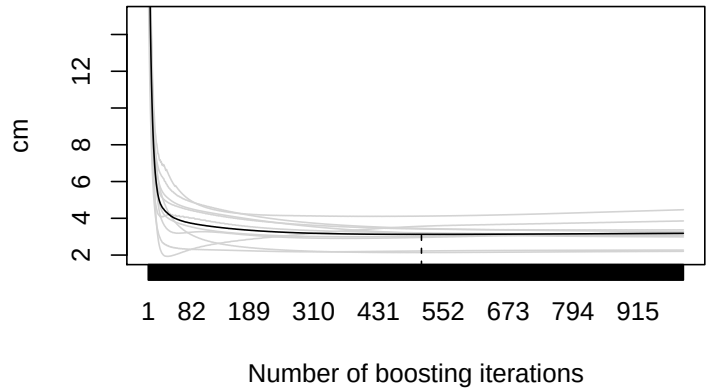

Figure 3: Bootstrapped out-of-bag risk for the model of the fossil fuels. For each fold, the out-of-bag risk is displayed as a gray line. The mean out-of-bag risk is visualized by a black line. The optimal number of boosting iterations is marked by a dashed vertical line.

```
R> set.seed(123)
R> folds_sof <- cv(weights = model.weights(sof), type = "bootstrap", B = 10)
R> cvm_sof <- cvrisk(sof, folds = folds_sof, grid = 1:1000)
```

The object `cvm_sof` contains the out-of-bag risk of each fold for all 1000 iterations.

5.3. Methods to extract and visualize results from the resampling object

For a '`cvrisk`' object as created by `cvrisk()`, the method `mstop()` extracts the estimated optimal number of boosting iterations, which corresponds to the number of boosting iterations yielding the minimal mean out-of-bag risk. `plot()` generates a plot of the estimated out-of-bag risk per stopping iteration in each fold. In addition, the mean out-of-bag risk per stopping iteration is displayed. The estimated optimal stopping iteration is marked by a dashed vertical line. In such a plot, the convergence behavior can be graphically examined.

Case study (continued): Fossil fuel data

We generate a plot that displays for each fold the estimated out-of-bag risk per stopping iteration for each fold; see Figure 3.

```
R> plot(cvm_sof, ylim = c(2, 15))
```

For small numbers of boosting iterations, the out-of-bag risk declines sharply with a growing number of boosting iterations. With more and more iterations the model gets more complex and the out-of-bag risk starts to slowly increase. The dashed vertical line marks the estimated optimal stopping iteration of 511, which can be accessed using the function `mstop()`:

```
R> mstop(cvm_sof)

[1] 511
```

5.4. Methods to extract and display results from the model object

Fitted 'FDboost' objects inherit methods from class 'mboost'. Thus, all methods available for 'mboost' objects can also be applied to models fitted by `FDboost()`. The design and penalty matrices that are constructed by the base learners can be extracted using the `extract()` function. For example, `extract(object, which = 1)` returns the design matrix of the first base learner and `extract(object, which = 1, what = "penalty")` the corresponding penalty matrix. The number of boosting iterations for an 'FDboost' object can be changed afterwards using the subset operator; e.g., `object[50]` sets the number of boosting iterations for `object` to 50. Note that the subset operator directly changes `object`, and hence no assignment is necessary.

One can access the estimated coefficients by the `coef()` function. The function takes a fitted `object` produced by `FDboost()` and returns estimated coefficient functions such as $\hat{\beta}(s)$, $\hat{\beta}(s,t)$, $\hat{g}(x)$ or other estimated effects. For smooth effects, `coef()` returns the smooth estimated effects evaluated on a regular grid. The resolution of the grid can be specified by the arguments `n1, n2` and `n3` for 1-, 2- and 3-dimensional smooth terms, respectively, which define the number of equidistantly spaced grid points over the range of the covariate. The resulting object is a list containing an element for the offset and a named list with one entry for each further model term. The value of the offset for each observation can be accessed with `coef(object)$offset$value`. List entries for model terms in `coef(object)$smterms` are, in turn, lists with different entries, in particular, including `$x` (`$y, $z`) representing unique grid-points used to evaluate the coefficient function and `$value` representing a vector, matrix or list of matrices with the coefficient values. The estimated spline-coefficients $\hat{\boldsymbol{\theta}}_j$ of smooth effects can be obtained by `object$coef()`, which is equal to setting the argument `raw` to `TRUE` in the `coef` function.

The estimated effects can be graphically displayed by the `plot()` function. The coefficient plots can be customized by various arguments. For example, coefficient surfaces can be displayed as image plots, setting `pers = FALSE`, or as perspective plots, setting `pers = TRUE`. To plot only some of the base learners, the argument `which` can be used. For instance, `plot(object, which = c(1, 3))` plots the estimated effects of the first and the third base learner. The fitted values and predictions for new data can be obtained by the methods `fitted()` and `predict()`, respectively.

Case study (continued): Fossil fuel data

To better understand the penalization used in the `sof` model, we can exemplarily extract the marginal penalty matrix for UVVIS as follows:

```
R> marg_pen <- extract(sof, "penalty", which = 2)
R> marg_pen[[1]][1:5, 1:5]

     [,1] [,2] [,3] [,4] [,5]
[1,]    1   -1    0    0    0
[2,]   -1    2   -1    0    0
[3,]    0   -1    2   -1    0
[4,]    0    0   -1    2   -1
[5,]    0    0    0   -1    2
```

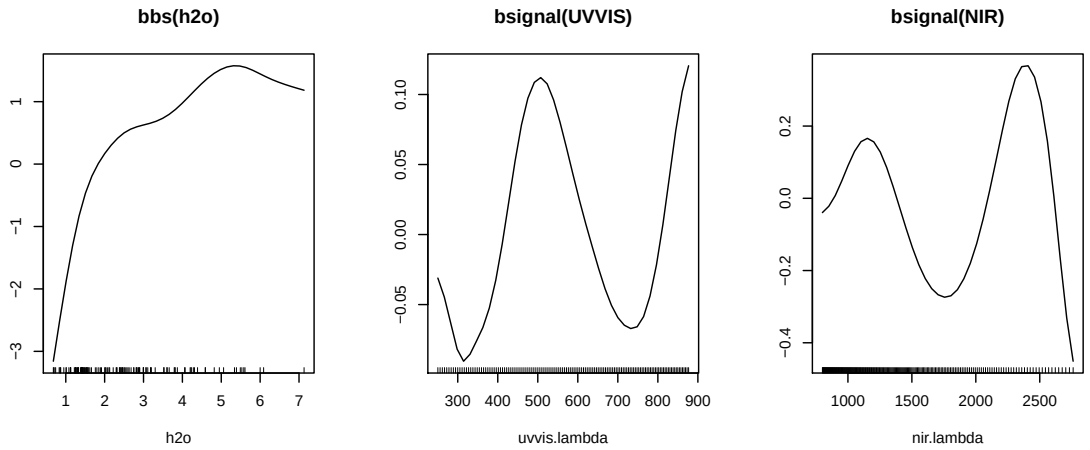

Figure 4: Coefficient estimates of the model for the heat value of the fossil fuels with optimal number of boosting iterations. The smooth effect of the water content (left), the linear effect of the UVVIS spectrum (center) and the NIR spectrum (right) are displayed.

In order to continue working with the optimal model, we set the number of boosting iterations to the estimated optimal value.

```
R> sof <- sof[mstop(cvm_sof)]
```

We can access estimated coefficients using `coef()`, e.g., by extracting the estimated coefficient function $\hat{\beta}_{\text{NIR}}(s_{\text{NIR}})$ contained in `$value` evaluated at grid points `$x`.

```
R> coef_sof <- coef(sof)
R> str(coef_sof$smterms$`bsignal(NIR)`)
```

To display the estimated effects, `plot()` can be called on the fitted '`FDboost`' object. Per default, `plot()` only displays effects of base learners that were selected at least once. See Figure 4 for the resulting plots.

```
R> par(mfrow = c(1, 3))
R> plot(sof, ask = FALSE, ylab = "")
```

The mean heat value is estimated to be higher for higher water content and lower for lower water content (see Figure 4 left). High values of the UVVIS spectrum at a wavelength of around 500 and 850 nm are associated with higher heat values. Higher values of the UVVIS spectrum at wavelength around 300 and 750 nm are associated with lower heat values (see Figure 4 middle). The effect of the NIR spectrum can be interpreted analogously.

5.5. Bootstrapped coefficient estimates

In order to get a measure for the uncertainty associated with the estimated coefficient functions, one can employ nested bootstrap. The optimal number of boosting iterations in each bootstrap fold, in turn, is estimated by an inner resampling procedure. The bootstrapped coefficients are shrunken towards zero as boosting shrinks coefficients towards zero due to

early stopping. Thus, the resulting bootstrap "confidence" interval is biased towards zero but still captures the variability of the coefficient estimates. While they do not have proper coverage properties due to shrinkage bias, these bootstrap intervals capture all the sources of uncertainty (induced by the resampling, the model selection as well as the actual uncertainty of coefficients). They may be used to check, e.g., for the existence of certain effects by examining whether the resulting intervals contain the value zero, which was found to work well in Rügamer *et al.* (2018). Having no formal inference procedure clearly is a limitation of the model-based boosting framework in general and users who want to formally test prespecified hypotheses are referred to alternative software packages such as **refund** (Goldsmith *et al.* 2019) for cases where these are applicable and the particular strengths of model-based boosting (high-dimensional data and models, model selection, general loss-functions) are not needed. In **FDboost** the function `bootstrapCI()` can be used to conveniently compute bootstrapped coefficients:

```
bootstrapCI(object, B_outer = 100, B_inner = 25, ...)
```

The argument `object` is the fitted model object. The maximal number of boosting iterations for each bootstrap fold is the number of boosting iterations of the model object. Per default bootstrap is used with `B_outer = 100` outer folds and `B_inner = 25` inner folds. The dots argument, `...`, can be used to pass further arguments to `applyFolds()`, which is used for the outer bootstrap. In particular, setting the argument `mc.cores` to an integer greater 1 will run the outer bootstrap in parallel on the number of cores that are specified via `mc.cores` (this does not work under Windows, as the parallelization is based on the function `mclapply()`). As for the resampling scheme, which determines the number of iterations, the bootstrap which is done to quantify uncertainty of coefficient estimates should be conducted on the level of independent observations. This is particularly relevant for functional responses, where both resampling procedures should be done on the level of curves. Additional dependence in the data, such as observations sampled from clusters or in a longitudinal fashion, should also be taken into account for scalar-on-function models. To this end, observations should be sampled on the levels of clusters, subjects, or in nested designs, by a nested sampling for each of the levels. This yields a limitation of our method in cases, in which observations cannot be separated into independent units (e.g., for spatially correlated observations with a strong dependence among all observations). However, customized solutions such as a block-wise bootstrap (cf. Brockhaus *et al.* 2018) for time-series data can be employed as in the scalar case.

Case study (continued): Fossil fuel data

We recompute the model on 100 bootstrap samples to compute bootstrapped coefficient estimates. In each bootstrap fold the optimal number of boosting iterations is estimated by an inner bootstrap with 10 folds. In contrast to other methods and analytic inference concepts, employing bootstrap for coefficient uncertainty is much more time consuming but can be easily parallelized. See the help page of `bootstrapCI()` for example code. The resulting estimated coefficients can be seen in Figure 5.

```
R> set.seed(123)
R> sof_bootstrapCI <- bootstrapCI(sof[100], B_outer = 5, B_inner = 3,
+    mc.cores = 1)
```

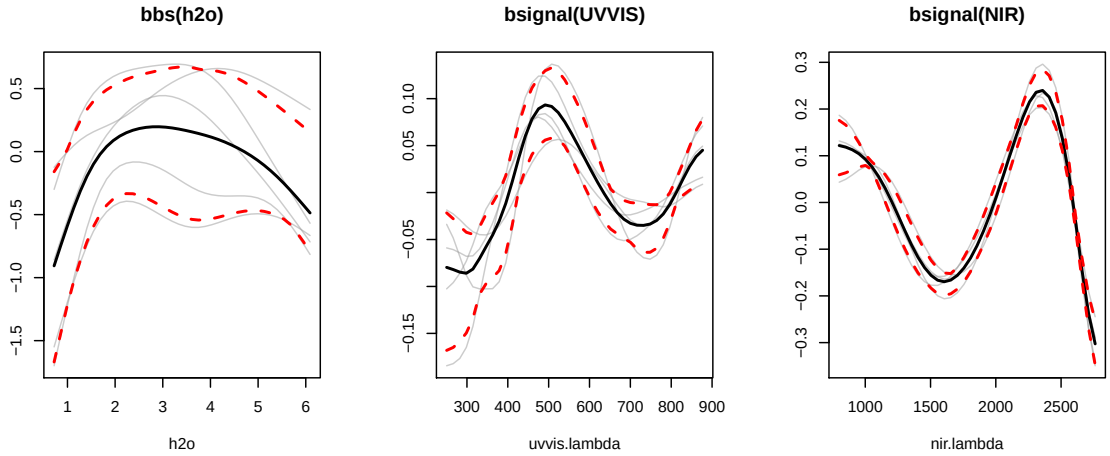

Figure 5: Bootstrapped coefficient estimates of the model for the heat value of the fossil fuels. The coefficient estimates in the bootstrap samples for the smooth effect of the water content (left), the linear effect of the UVVIS spectrum (middle) and the NIR spectrum (right) are displayed. The pointwise 5% and 95% quantiles are marked with dashed red lines. The pointwise 50% quantile is marked by a black line.

```
R> par(mfrow = c(1, 3))
R> plot(sof_bootstrapCI, ask = FALSE, commonRange = FALSE, ylab = "")
```

6. Functional response

In this section, we explain how to fit models with functional response like model (4). Models with scalar and functional covariates are treated, thus covering function-on-scalar and function-on-function regression models.

6.1. Specification of functional response

If a functional variable is observed on one common grid, its observations can be represented by a matrix. In **FDboost**, such functional variables have to be supplied as <number of curves> by <number of evaluation points> matrices. That is, a functional response $y_i(t_g)$, with $i = 1, \ldots, N$ curves and $g = 1, \ldots, G$ evaluation points, is stored in an $N \times G$ matrix with cases in rows and evaluation points in columns. This corresponds to a data representation in wide format. The t variable must be given as vector $(t_1, \ldots, t_G)^\top$.

For the functional response, curve-specific observation grids are possible, i.e., the ith response curve is observed at evaluation points $(t_{ig}, \ldots, t_{iG_i})^\top$ specific for each curve i. In this case, three pieces of information must be supplied: the values of the response, the evaluation points and the curve to which each of the observations belongs. The response is supplied as the vector $(y_1(t_{11}), \ldots, y_N(t_{NG_N}))^\top$. This vector has length $n = \sum_{i=1}^{N} G_i$. The `t` variable contains all evaluation points $(t_{11}, \ldots, t_{NG_N})^\top$. The argument `id` contains the information on which observation corresponds to which response curve. The argument `id` must be supplied as a right-sided formula `id = ~ idvariable`.

Case study (continued): Emotion components data

In the following, we give an example for a model fit with a functional response. In the first model fit, the response is stored in the matrix `EMG`, in the second in the vector `EMG_long`. We fit an intercept model by defining the `formula` as `y ~ 1` and the `timeformula` as `~ bbs(t)`.

```
R> fos_intercept <- FDboost(EMG ~ 1, timeformula = ~ bbs(t, df = 6),
+   data = emotionHGL)
```

The corresponding mathematical formula is

$$\mathbb{E}(Y_{\text{EMG}}(t)) = \beta_0(t),$$

i.e., we simply estimate the mean curve $\beta_0(t)$ of the functional EMG signal. Recall that the intercept in the model $\beta_0(t)$ is estimated as the sum of the offset and the intercept base learner. Per default (`offset = NULL`) the model is fitted using a smooth offset. For the given application the resulting intercept is almost equal to the offset, as the effect of the intercept base learner is almost zero. To get a less smooth offset, one could specify `offset_control = o_control(k_min = 30)`, which increases the basis dimension in the estimation of the offset. A smoother offset can be achieved by setting `offset_control = o_control(k_min = 10)`. The argument `k_min` is the dimension of the basis that is used for estimating the offset and defaults to 20. See Figure 6 for a comparison of the estimated intercepts depending on the smoothness of the offset.

```
R> fos_intercept_wiggly <- FDboost(EMG ~ 1, timeformula = ~ bbs(t, df = 6),
+    data = emotionHGL, offset_control = o_control(k_min = 30))
R> fos_intercept_smooth <- FDboost(EMG ~ 1, timeformula = ~ bbs(t, df = 6),
+    data = emotionHGL, offset_control = o_control(k_min = 10))
R> par(mfrow = c(1, 3))
R> plot(fos_intercept_smooth, ask = FALSE)
R> plot(fos_intercept, ask = FALSE)
R> plot(fos_intercept_wiggly, ask = FALSE)
```

To fit a model with response in long format, we first have to convert the data into the corresponding format. We therefore construct a data set `data_emotion_long` that contains the response in long format. Usually, the long format specification is only necessary for responses that are observed on curve-specific grids. We here provide this version for illustrative purposes, but in this example the following model specification is equivalent to the previous model fit `fos_intercept`.

```
R> emotion_long <- emotionHGL
R> emotion_long$EMG_long <- as.vector(emotion_long$EMG)
R> emotion_long$time_long <- rep(emotionHGL$t, each = nrow(emotionHGL$EMG))
R> emotion_long$curveid <- rep(1:nrow(emotionHGL$EMG), ncol(emotionHGL$EMG))
R> fos_intercept_long <- FDboost(EMG_long ~ 1,
+    timeformula = ~ bbs(time_long, df = 3), id = ~ curveid,
+    data = emotion_long)
```

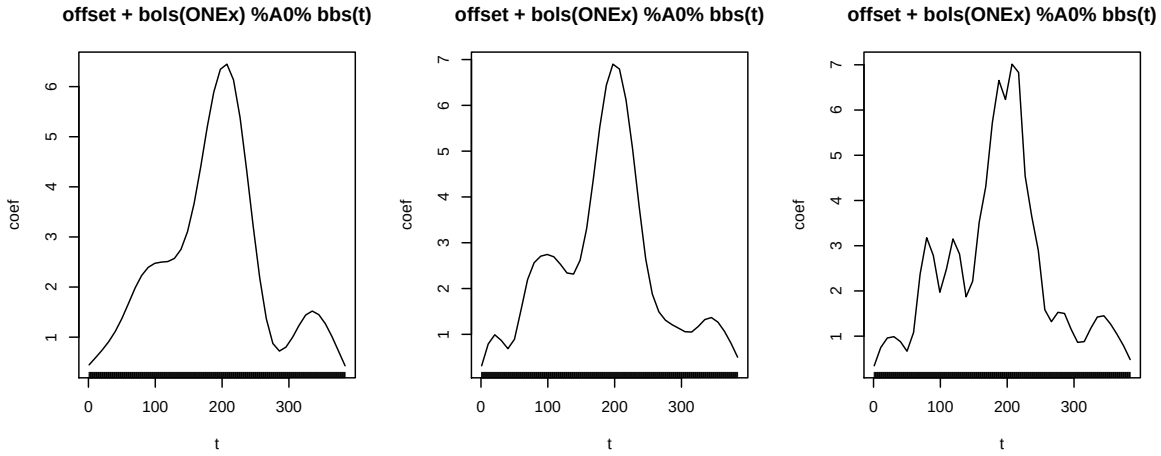

Figure 6: Estimated intercepts for different offset specifications. From left to right we use a basis representation with 10, 20 or 30 splines for estimating the offset.

6.2. Effects in the `formula` that are combined with the `timeformula`

Many covariate effects can be represented by the Kronecker product of two marginal bases as in Equation 6. The response and the bases in covariate direction $\boldsymbol{b}_j(x)$ are specified in `formula` as `Y ~ b_1 + ...+ b_J`. The base learner for the expansion along t is specified in `timeformula` as `~ b_Y`. Each base learner in `formula` is combined with the base-leaner in `timeformula` using the operator `%O%`. This operator implements the Kronecker product of two basis vectors as in Equation 6. Consider, for example, `formula = Y ~ b_1 + b_2`. If, `b_1` is defined by `bols(z)` with covariate z and a scalar response is given, using `timeformula = NULL` specifies a model with linear effect $z\beta$. In the case of a functional response, we usually want the effect $z\beta$ to vary for each time point $t \in \mathcal{T}$ of the response, i.e., $z\beta(t)$. This can be done by defining `timeformula = ~ b_Y`, where the base learner `b_Y` defines the form of variation in t direction. Assuming a linear effect in t, `b_Y` is set to `bols(t)`. The combination of `timeformula` and `formula` yields `Y ~ b_1 %O% b_Y + b_2 %O% b_Y`. For the particular example, `b_1 %O% b_Y` is equal to `bols(z) %O% bols(t)` yielding $z\beta(t)$.

If marginal base learners are specified with a penalty, the Kronecker product of the two basis vectors is defined with an isotropic penalty matrix as in Equation 8. If the effect should only be penalized in t direction, the operator `%A0%` can be used as it sets up the penalty as in Equation 9. If `formula` contains base learners that are composed of two base learners by `%O%` or `%A0%`, those effects are not expanded with `timeformula`, allowing for model specifications with different effects in t direction. This can be used, for example, to model some effects linearly and others non-linearly in t or to construct effects using `%A0%`. For further details on these operators and their use, we refer to Appendix C.

We start with base learners for the `timeformula`. Theoretically, it is possible to use any base learner which models the effect of a continuous variable. Usually, the effects are assumed to be smooth along t. In this case, the base learner `bbs()` can be used, which represents the smooth effect by P-splines (Schmid and Hothorn 2008a). Thus, `bbs()` uses a B-spline representation for the design matrix and a squared difference matrix as penalty matrix. Using the `bbs()` base-leaner in the `timeformula` corresponds to using a marginal basis $\boldsymbol{b}_Y$ as described in Equation 10.

Additive predictor $h(\boldsymbol{x}, t) = \sum_j h_j(\boldsymbol{x}, t)$	Call
$\beta_0(t)$	`y ~ 1`
$\beta_0(t) + z_1\beta_1(t)$	`y ~ 1 + bolsc(z1)`
$\beta_0(t) + f_1(z_1, t)$	`y ~ 1 + bbsc(z1)`
$\beta_0(t) + z_1\beta_1(t) + z_2\beta_2(t) + z_1z_2\beta_3(t)$	`y ~ 1 + bolsc(z1) + bolsc(z2) + bols(z1) %Xc% bols(z2)`
$\beta_0(t) + z_1\beta_1(t) + f_2(z_2, t) + z_1f_3(z_2, t)$	`y ~ 1 + bolsc(z1) + bbsc(z2) + bols(z1) %Xc% bbs(z2)`
$\beta_0(t) + f_1(z_1, t) + f_2(z_2, t) + f_3(z_1, z_2, t)$	`y ~ 1 + bbsc(z1) + bbsc(z2) + bbs(z1) %Xc% bbs(z2)`
$\beta_0(t) + \int_{\mathcal{S}} x(s)\beta_1(s, t)\, ds$	`y ~ 1 + bsignal(x, s = s)`
	`y ~ 1 + bfpc(x, s = s)`
$\beta_0(t) + z\beta_1(t) + \int_{\mathcal{S}} x(s)\beta_2(s, t)\, ds$	`y ~ 1 + bolsc(z) + bsignal(x, s = s) +`
$+z \int_{\mathcal{S}} x(s)\beta_3(s, t)\, ds$	`bsignal(x, s = s) %X% bolsc(z)`

Table 4: Additive predictors that can be represented within the array framework.

Base learners that can be used in `formula` are listed in Table 4. In this table, a selection of additive predictors that can be represented within the array framework are listed in the left column. In the right column, the corresponding `formula` is given. The `timeformula` is set to `~ bbs(t)` to model all effects as smooth effects in t. Thus, the specified effects in `formula` are combined with `timeformula` using the Kronecker product.

For `offset = NULL`, the model contains a smooth offset $\beta_0^*(t)$. The smooth offset is computed prior to the model fit as smoothed population minimizer of the loss. For mean regression, the smooth offset is the smoothed mean over t. The specification `offset = "scalar"` yields a constant offset β_0^*. The resulting intercept in the final model is the sum of the offset and the smooth intercept $\tilde{\beta}_0(t)$ specified in the `formula` as `1`, i.e., $\beta_0(t) = \beta_0^*(t) + \tilde{\beta}_0(t)$.

The upper part of Table 4 gives examples for linear predictors with scalar covariates. A linear effect of a scalar covariate is specified using the base learner `bolsc()`. This base learner works for continuous and for factor variables. A smooth effect of a continuous covariate is obtained by using the base learner `bbsc()`. The base learners `bolsc()` and `bbsc()` are similar to the base learners `bols()` and `bbs()` from the **mboost** package, but enforce pointwise sum-to-zero constraints to ensure identifiability for models with functional response (the suffix 'c' refers to 'constrained'). Since, for example, the effect $f_1(z_1, t)$ contains a smooth intercept as special case, the model would not be identifiable without constraints, see Appendix A for more details. We use the constraint $\sum_{i=1}^{N} h_j(\boldsymbol{x}_i, t) = 0$ for all t, which centers each effect for each point t (Scheipl *et al.* 2015). This implies that effects varying over t can be interpreted as deviations from the smooth intercept and that the intercept can be interpreted as global

mean if all effects are centered in this way. It is possible to check whether all covariate effects sum to zero for all points t by setting `check0 = TRUE` in the `FDboost()` call. To specify interaction effects of two scalar covariates, the base learners for each of the covariates are combined using the operator `%Xc%` that applies the sum-to-zero constraint to the interaction effect.

The lower part of Table 4 gives examples for linear predictors with functional covariates. In analogy to models with scalar response, the linear effect $\int_{\mathcal{S}} x(s)\beta(s,t)\,ds$ can be fitted by `bsignal()` or `bfpc()` and the interaction effect is formed using the operator `%X%` (see the explanations for Table 3).

Case study (continued): Emotion components data

For the emotion components data with the EMG signal as functional response, $Y_{\text{EMG}}(t)$, $t \in [0, 1560]$ ms, we fit models with scalar and functional covariate effects in the following.

Function-on-scalar regression

We specify a model for the conditional expectation of the EMG signal using a random intercept curve for each subject and a linear effect for the study setting `power`:

$$\mathbb{E}(Y_{\text{EMG}}(t)|\boldsymbol{x}) = \beta_0(t) + \sum_{k=1}^{23} I(x_{\text{subject}} = k)\beta_{\text{subject},k}(t) + x_{\text{power}}\beta_{\text{power}}(t), \tag{13}$$

with `subject` having values 1 to 23 for the participants of the study, and x_{power} taking values $\{-1, 1\}$ for low and high power. Both covariate effects in the model are specified by using a centered base learner. The linear effect of the factor variable `subject` and the effect of `power` are both specified using the `bolsc()` base learner. Therefore, the effects sum up to zero for each time point t over all observations $i = 1, \ldots, N = 184$, i.e., $\sum_{i=1}^{N}\sum_{k=1}^{23} I(x_{\text{subject},i} = k)\beta_{\text{subject},k}(t) = 0$ for all t.

```
R> fos_random_power <- FDboost(EMG ~ 1 + bolsc(subject, df = 2) +
+    bolsc(power, df = 1) %A0% bbs(t, df = 6),
+    timeformula = ~ bbs(t, df = 3), data = emotion)
```

As described in Section 3, it is important that all base learners have the same number of degrees of freedom. In this model the degrees of freedom for each base learner are $2 \cdot 3 = 6$. By specifying the `bolsc`-base learner with `df = 2` for `subject`, the subject effect is estimated with a ridge penalty similar to a random effect, whereas the `power` effect is estimated unpenalized due to the use of the `%A0%`-operator. If the effects are assumed to have more (spiky) features over time, the smoothness of the effect curves can be decreased by increasing the number of `knots`, decreasing the order of `differences` in the penalty or decreasing the `degree` of the B-spline basis in the `bbs`-base learners. Another way to change the smoothness is to increase the degrees of freedom, `df`. For models with more than one base learner this, however, requires also changing the degrees of freedom for other base learners to facilitate unbiased base learner selection as explained in Section 3.

Analogously, a model with response in long format as in `fos_intercept_long` could be specified by changing the formula to the formula of `fos_random_power`.

Function-on-function regression

For the data subset for one specific game condition, we use the effect of the EEG signal to model the EMG signal:

$$\mathbb{E}(Y_{\text{EMG}}(t)|\boldsymbol{x}) = \beta_0(t) + \int_{\mathcal{S}} x_{\text{EEG}}(s)\beta_{\text{EEG}}(s,t)\,ds. \tag{14}$$

In this model each time point of the covariate $x_{\text{EEG}}(s)$ potentially influences each time point of the response $Y_{\text{EMG}}(t)$. We center the EEG signal per time point such that $\sum_{i=1}^{N} x_{\text{EEG},i}(s) = 0$ for each s to center its effect per time point.

```
R> emotionHGL$EEG <- scale(emotionHGL$EEG, scale = FALSE)
R> fof_signal <- FDboost(EMG ~ 1 + bsignal(EEG, s = s, df = 2),
+    timeformula = ~ bbs(t, df = 3), data = emotionHGL)
```

We will show and interpret plots of the estimated coefficients later on. If the brain activity (measured via the EEG) triggers the muscle activity (measured via the EMG), it is reasonable to assume that EMG signals are only influenced by past EEG signals. Such a relationship can be represented using a historical effect $\int_{T_1}^{t} x(s)\beta(s,t)\,ds$, which will be discussed in the next paragraph.

6.3. Effects in the `formula` in both covariate and t direction

If the covariate varies with t, the effect cannot be separated into a marginal basis depending on the covariate and a marginal basis depending only on t. In this case the effects are represented as in Equation 5. Examples for such effects are historical and concurrent functional effects, as discussed in Brockhaus *et al.* (2017). In Table 5 we give an overview of possible additive predictors containing such effects.

The concurrent effect $\beta(t)x(t)$ is only meaningful if the functional response and the functional covariate are observed over the same domain. Models with concurrent effects can be seen as varying-coefficient models (Hastie and Tibshirani 1993), where the effect varies over t. The base learner `bconcurrent()` expands the smooth concurrent effect $\beta(t)$ in P-splines. The historical effect $\int_{T_1}^{t} x(s)\beta(s,t)\,ds$ uses only covariate information up to the current observation point of the response. The base learner `bhist()` expands the coefficient surface $\beta(s,t)$ in s and in t direction using P-splines to fit the historical effect. In Appendix B, details on the representation of functional effects are given.

The interface of `bhist()` is:

```
bhist(x, s, time, limits = "s<=t", knots = 10, degree = 3, differences = 1,
  df = 4, lambda = NULL, check.ident = FALSE)
```

Most arguments of `bhist()` are analogous to those of `bsignal()`. `bhist()` has the additional argument `time` to specify the observation points of the response. Via the argument `limits` in `bhist()` the user can specify integration limits depending on t. Per default a historical effect with limits $s \leq t$ is used. Other integration limits can be specified by using a function with arguments `s` and `t`, which returns `TRUE` for combinations of `s` and `t` that lie within the integration interval and `FALSE` otherwise. In the following, we give three examples for functions that can be used for `limits` resulting in a classical historical effect, a lag effect or a lead effect, respectively:

Additive predictor $h(x,t) = \sum_j h_j(x,t)$	Call
$\beta_0(t) + x(t)\beta(t)$	`y ~ 1 + bconcurrent(x, s = s, time = t)`
$\beta_0(t) + \int_{T_1}^{t} x(s)\beta(s,t)\,ds$	`y ~ 1 + bhist(x, s = s, time = t)`
$\beta_0(t) + \int_{t-\delta}^{t} x(s)\beta(s,t)\,ds$	`y ~ 1 + bhist(x, s = s, time = t, limits = limitsLag)`
$\beta_0(t) + \int_{T_1}^{t-\delta} x(s)\beta(s,t)\,ds$	`y ~ 1 + bhist(x, s = s, time = t, limits = limitsLead)`
$\int_{l(t)}^{u(t)} x(s)\beta(s,t)\,ds$	`y ~ 1 + bhist(x, s = s, time = t, limits = mylimits)`
$\beta_0(t) + z\beta_1(t) + \int_{T_1}^{t} x(s)\beta_2(s,t)\,ds +$ $z\int_{T_1}^{t} x(s)\beta_3(s,t)\,ds$	`y ~ 1 + bolsc(z) + bhist(x, s = s, time = t) + bhistx(x) %X% bolsc(z)`

Table 5: Additive predictors that contain effects that cannot be separated into an effect in covariate direction and an effect in t direction. These effects in `formula` are not expanded by `timeformula`. We give examples for general limit functions `mylimits` in this section. In `bhistx()`, the variable `x` has to be of class '`hmatrix`', please see the help page of `bhistx()` for details.

```
R> limitsHist <- function(s, t) {
+    s <= t
+  }
R> limitsLag <- function(s, t, delta = 5) {
+    s >= t - delta &  s <= t
+  }
R> limitsLead <- function(s, t, delta = 5) {
+    s <= t - delta
+  }
```

The base learner `bhistx()` is especially suited to form interaction effects such as factor-specific historical effects (Rügamer *et al.* 2018), as `bhist()` cannot be used in combination with the row-wise tensor product operator `%X%` to form interaction effects. `bhistx()` requires the data to be supplied as an object of type '`hmatrix`'; see the help page of `bhistx()` for its set-up.

Case study (continued): Emotion components data

Again, we use the subset of the data for one specific game condition. We start with a simple function-on-function regression model by specifying a concurrent effect of the EEG signal on the EMG signal:

$$\mathbb{E}(Y_{\text{EMG}}(t)|\boldsymbol{x}) = \beta_0(t) + x_{\text{EEG}}(t)\beta(t).$$

A concurrent effect is obtained by the base learner `bconcurrent()`, which is not expanded

by the base learner in `timeformula`. In this model, `timeformula` is only used to expand the smooth intercept.

```
R> fof_concurrent <- FDboost(EMG ~ 1 +
+    bconcurrent(EEG, s = s, time = t, df = 6),
+    timeformula = ~ bbs(t, df = 6), data = emotionHGL,
+    control = boost_control(mstop = 300))
```

Assuming that the activity in the muscle can be completely traced back to previous activity in the brain, a more appropriate model seems to be a historical model including a historical effect

$$\mathbb{E}(Y_{\text{EMG}}(t)|\boldsymbol{x}) = \beta_0(t) + \int_{l(t)}^{u(t)} x_{\text{EEG}}(s)\beta_{\text{EEG}}(s,t)\,ds. \tag{15}$$

From a neuro-anatomy perspective, the signal from the brain requires time to reach the muscle. We therefore set $l(t) = 0$ and $u(t) = t - 3$, which is in line with Rügamer *et al.* (2018).

```
R> fof_historical <- FDboost(EMG ~ 1 + bhist(EEG, s = s, time = t,
+    limits = function(s, t) s <= t - 3, df = 6),
+    timeformula = ~ bbs(t, df = 6), data = emotionHGL,
+    control = boost_control(mstop = 300))
```

More complex historical models are discussed in Rügamer *et al.* (2018). In particular, a model containing random effects for the participants, effects for the game conditions and game condition- as well as subject-specific historical effects of the EEG signal.

It is also possible to combine effects listed in Tables 4 and 5 to form more complex models. In particular, base learners with and without array structure can be combined within one model. As in the component-wise boosting procedure each base learner is evaluated separately, the array structure of the Kronecker product base learners can still be exploited in such hybrid models.

6.4. Model tuning and early stopping

For a fair selection of base learners, additional care is needed for functional responses as only some of the base learners in the `formula` are expanded by the base learner in `timeformula`. In particular, all base learners listed in Table 4 are expanded by `timeformula`, whereas base learners given in Table 5 are not expanded by `timeformula`. For the row-wise tensor product and the Kronecker product of two base learners, the degrees of freedom for the combined base learner is computed as product of the two marginally specified degrees of freedom. For instance, `formula = y ~ bbsc(z, df = 3) + bhist(x, s = s, df = 12)` and `timeformula = ~ bbs(t, df = 4)` implies $3 \cdot 4 = 12$ degrees of freedom for the first combined base learner and 12 degrees of freedom for the second base learner. The call `extract(object, "df")` displays the degrees of freedom for each base learner in an '`FDboost`' object. For other tuning options such as the number of iterations and the specification of the step-length see Section 5.

To find the optimal number of boosting iterations for a model fit with functional response, **FDboost** provides two resampling functions. Depending on the specified model, some parameters are computed from the data prior to the model fit: By default a smooth functional

offset $\beta_0^*(t)$ is computed (`offset = NULL` in `FDboost()`) and for linear and smooth effects of scalar variables, defined by `bolsc()` and `bbsc()`, transformation matrices for the sum-to-zero constraints are computed. The `cvrisk()` method for 'FDboost' objects uses the smooth functional offset and the transformation matrices from the original model fit in all folds. Thus, these parameters are treated as fixed and the uncertainty induced by their estimation is not considered in the resampling. On the other hand, `applyFolds()` recomputes the whole model in each fold. The two resampling methods are equal if no smooth offset is used and if the model does not contain any base learner with a sum-to-zero constraint (i.e., neither `bolsc()` nor `bbsc()`). In general, we recommend to use the function `applyFolds()` to determine the optimal number of boosting iterations for a model with functional response. The interface of `applyFolds()` is:

```
applyFolds(object, folds = cv(rep(1, length(unique(object$id))),
  type = "bootstrap"), grid = 1:mstop(object))
```

The interface is in analogy to the interface of `cvrisk()`. In the argument `object`, the fitted model object is specified. `grid` defines the grid on which the optimal stopping iteration is searched. Via the argument `folds` the resampling folds are defined by suitable weights. The function `applyFolds()` expects resampling weights that are defined on the level of curves, $i = 1, \ldots, N$. That means that the folds must contain weights w_i, $i = 1, \ldots, N$, which can be done easily using the function `cv()`.

6.5. Methods to extract and display results

Methods to extract and visualize results are the same irrespective of scalar or functional response. Thus, we refer to the corresponding paragraphs at the end of Section 5.

Case study (continued): Emotion components data

Exemplarily, the penalty matrix for the historical effect can be extracted as follows:

```
R> kron_pen <- extract(fof_historical, "penalty")
R> as.matrix(kron_pen[[1]][1:5, 1:5])
```

This is equal to the kronecker sum of two marginal B-spline penalties with isotropic penalization (as defined by Equation 7 with $\lambda_j = \lambda_Y$):

```
R> margPen <- extract(with(emotionHGL,
+    bbs(s, knots = 10, differences = 1)), "penalty")
R> (kronecker(margPen, diag(ncol(margPen))) +
+    kronecker(diag(ncol(margPen)), margPen))[1:5, 1:5]

     [,1] [,2] [,3] [,4] [,5]
[1,]    2   -1    0    0    0
[2,]   -1    3   -1    0    0
[3,]    0   -1    3   -1    0
[4,]    0    0   -1    3   -1
[5,]    0    0    0   -1    3
```

As for scalar response, the `plot`-function can be used to access the estimated effects in a function-on-function regression. In the following, we compare the three basic types of functional covariate effects, which can be used in conjunction with a functional response. We first determine the optimal number of stopping iterations for all three presented models.

```
R> set.seed(123)
R> folds_bs <- cv(weights = rep(1, fof_signal$ydim[1]),
+    type = "kfold", B = 5)
R> cvm_concurrent <- applyFolds(fof_concurrent, folds = folds_bs,
+    grid = 1:300)
R> ms_conc <- mstop(cvm_concurrent)
R> fof_concurrent <- fof_concurrent[ms_conc]
R> cvm_signal <- applyFolds(fof_signal, folds = folds_bs, grid = 1:300)
R> ms_signal <- mstop(cvm_signal)
R> fof_signal <- fof_signal[ms_signal]
R> cvm_historical <- applyFolds(fof_historical, folds = folds_bs,
+    grid = 1:300)
R> ms_hist <- mstop(cvm_historical)
R> fof_historical <- fof_historical[ms_hist]
```

Then, we plot the estimated effects into one figure:

```
R> par(mfrow = c(1, 3))
R> plot(fof_concurrent, which = 2, main = "Concurrent EEG effect")
R> plot(fof_signal, which = 2, main = "Signal EEG effect",
+    n1 = 80, n2 = 80, zlim = c(-0.02, 0.025),
+    col = hcl.colors(20, "YlGnBu"))
R> plot(fof_historical, which = 2, main = "Historical EEG effect",
+    n1 = 80, n2 = 80, zlim = c(-0.02, 0.025),
+    col = hcl.colors(20, "YlGnBu"))
```

The concurrent effect corresponds to the diagonal of the other two surfaces in Figure 7 and assumes that off-diagonal time points have no association. Due to the temporal lag between EEG and EMG discussed for model (15), there is no meaningful interpretation for this model and the effect is only shown for demonstrative purposes. The historical effect corresponds to the assumption that the upper triangle in the signal EEG effects should be zero, as future brain activity should not influence the present muscle activity. The results in Figure 7 (right panel) can be interpreted in the same manner as results of a scalar-on-function regression when keeping a certain time point t fixed. For the time point $t = 350$ of the EMG signal, for example, time points $s = 0$ to $s \approx 150$ of the EEG signal do not show an effect, but for $s > 150$ the estimated effect on the expected EMG signal is positive. For a detailed description of the interpretation of historical effect surfaces as shown in Figure 7, we refer to the online appendix of Rügamer *et al.* (2018).

Careful interpretation has to take into account that this data set has a rather small signal-to-noise ratio due to the oscillating nature of both signals. In such cases, it is recommended to check the uncertainty of estimated effects via bootstrap, e.g., by using the `bootstrapCI()` function as exemplarily shown in Figure 8. As the 5%-quantile surface exhibits exclusively

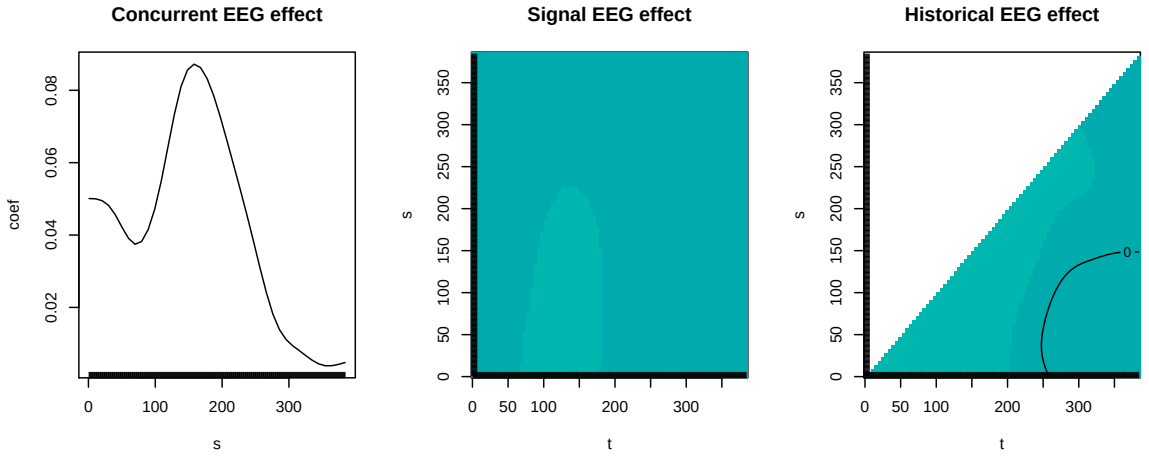

Figure 7: Visualization of estimated concurrent EEG effect (left panel), signal EEG effect (center panel) and historical EEG effect (right panel).

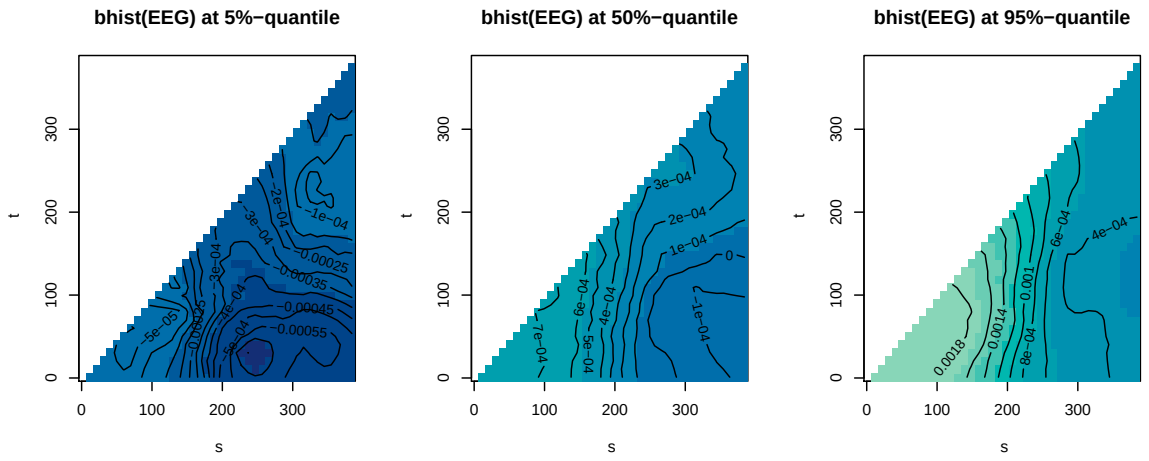

Figure 8: Visualization of three bootstrap quantiles for the historical EEG effect based on 100 bootstrap samples and a 10-fold cross-validation to optimize the stopping iteration for each bootstrap sample.

negative coefficients and the 95%-quantile surface exclusively positive coefficients in this case, the bootstrap surface suggests that the historical EEG may have no effect on the EMG-signal in the given model.

```
R> fof_historical_bci <- bootstrapCI(fof_historical, mc.cores = 2,
+    B_inner = 10, type_inner = "kfold")
R> par(mfrow = c(1, 3))
R> plot(fof_historical_bci, which = 2, ask = FALSE, pers = FALSE,
+    col = hcl.colors(20, "YlGnBu"), probs = c(0.05, 0.5, 0.95))
```

7. Functional regression models beyond the mean

Using boosting for model estimation it is possible to optimize other loss functions than the squared error loss. This allows to fit, e.g., generalized linear models (GLMs) and quantile regression models (Koenker 2005). It is also possible to fit models for several parameters of the conditional response distribution in the framework of generalized additive models for location, scale and shape (GAMLSS; Rigby and Stasinopoulos 2005).

For the estimation of these more general models, a suitable loss function in accordance with the modeled characteristic of the response distribution is defined and optimized. The absolute error loss (L_1 loss), for instance, implies median regression, and minimizing the L_2 loss yields mean regression.

In `FDboost()`, the regression type is specified by the `family` argument. The `family` argument expects an object of class '`Family`', which implements the respective loss function with its corresponding negative gradient and link function. The default is `family = Gaussian()` which yields L_2-boosting (Bühlmann and Yu 2003). This means that the mean squared error loss is minimized, which is equivalent to maximizing the log-likelihood of the normal distribution. Table 6 lists some loss functions currently implemented in **mboost**, which can be directly used in **FDboost** (see Hofner *et al.* 2014, for more families). Hofner *et al.* (2014) also give an example on how to implement new families via the function `Family()`. See also the help page `?Family` for more details on all families.

For a continuous response, several model types are available (Bühlmann and Hothorn 2007): L_2-boosting yields mean regression; a more robust alternative is median regression, which

Response type	Regression type	Loss	Call
Continuous	Mean regression	L_2 loss	`Gaussian()`
	Median regression	L_1 loss	`Laplace()`
	Quantile regression	Check function	`QuantReg()`
	Expectile regression	Asymmetric L_2	`ExpectReg()`
	Robust regression	Huber loss	`Huber()`
Non-negative	Gamma regression	$-l_{\text{gamma}}$	`GammaReg()`
Binary	Logistic regression	$-l_{\text{Bernoulli}}$	`Binomial()`
	AdaBoost classification	Exponential loss	`AdaExp()`
Count	Poisson model	$-l_{\text{Poisson}}$	`Poisson()`
	Neg. binomial model	$-l_{\text{neg. binomial}}$	`NBinomial()`
Scalar ordinal	Proportional odds model	$-l_{\text{proportional odds model}}$	`ProppOdds()`
Scalar categorical	Multinomial model	$-l_{\text{multinomial}}$	`Multinomial()`
Scalar survival time	Cox model	$-l_{\text{cox}}$	`CoxPH()`

Table 6: Overview of some families that are implemented in **mboost**. $-l_F$ denotes the negative log-likelihood of the distribution or model F.

optimizes the absolute error loss; the Huber loss is a combination of L_1 and L_2 loss (Huber 1964); quantile regression can be used to model a certain quantile of the conditional response distribution (Fenske *et al.* 2011); and expectile regression for modeling an expectile (Newey and Powell 1987; Sobotka and Kneib 2012). For a non-negative continuous response, models assuming the gamma distribution can be useful. A binary response can be modeled in a GLM framework with a logit model or by minimizing the exponential loss, which corresponds to the first boosting algorithm "AdaBoost" (Friedman 2001; Bühlmann and Hothorn 2007). Count data can be modeled assuming a Poisson or negative binomial distribution (Schmid, Potapov, Pfahlberg, and Hothorn 2010).

For functional response, we compute the loss pointwise and integrate over the domain of the response.

The following models can only be applied for scalar and not for functional response. For ordinal response, a proportional odds model can be used (Schmid, Hothorn, Maloney, Weller, and Potapov 2011). For categorical response, the multinomial logit model is available. For survival models, boosting Cox proportional hazard models and accelerated failure time models have been introduced by Schmid and Hothorn (2008b).

Case study (continued): Emotion components data

So far, we fitted a model for the conditional mean of the response. As a more robust alternative, we consider median regression by setting `family = QuantReg(tau = 0.5)` which is equal to `family = Laplace()`. We use the `update` function, to update the functional model with the new family.

```
R> fof_signal_med <- update(fof_signal, family = QuantReg(tau = 0.5))
```

For median regression, the smooth intercept is the estimated median at each time point and the effects are deviations from the median.

Similarly, if a certain quantile of the functional response is of interest, for example the 90% quantile, the model can be updated as follows:

```
R> fof_historical_q90 <- update(fof_historical, family = QuantReg(tau = 0.9))
```

which is equivalent to the following initial model specification:

```
R> fof_historical_q90 <- FDboost(EMG ~ 1 + bhist(EEG, s = s, time = t,
+    limits = function(s, t) s <= t - 3, df = 6),
+    timeformula = ~ bbs(t, df = 3), data = emotionHGL,
+    control = boost_control(mstop = 300), family = QuantReg(tau = 0.9))
```

To illustrate an example for scalar-on-function regression with binary response, consider the case, in which the goal is to predict the `game_outcome` in the case study for the emotions component data using only the muscle activity measured via the EMG. Consider the model

$$g(\mathbb{P}(Y_{i,j}|\boldsymbol{x}_{i,j})) = \beta_0 + \gamma_j + \int_{\mathcal{S}} x_{\text{EMG},i,j}(s)\beta_{\text{EMG}}(s)ds + \int_{\mathcal{S}} x_{\text{EMG},i,j}(s)\gamma_{\text{EMG},j}(s)ds,$$

for observation $i = 1, \ldots, 8$ of subject $j = 1, \ldots, 23$, where g is the inverse of the logit function, $Y_{i,j} \in \{0, 1\}$ determines the game outcome (*gain* and *loss*, respectively) for participant j in

game i, γ_j is a subject effect and the EMG is modeled using a global EMG effect β_{EMG} as well as a subject-specific EMG effect $\gamma_{\text{EMG},j}$. We first center the EMG-signal as it is now used as covariate:

```
R> emotion$EMG <- scale(emotion$EMG, center = TRUE, scale = FALSE)
```

and specify the model in `FDboost` as follows:

```
R> sof_binary <- FDboost(game_outcome ~ 1 + brandom(subject, df = 4) +
+   bsignal(EMG, s = s, df = 4) + brandom(subject, df = 2) %X%
+   bsignal(EMG, s = s, df = 2), data = emotion, family = Binomial(),
+   control = boost_control(mstop = 5000), timeformula = NULL)
```

Note that the row-wise tensor product operator `%X%` in this case is used to specify a subject-specific functional effect of the EMG-signal and the resulting degrees of freedom of this base learner are determined as the product of the `df`s of both base learners. To get a measure of the performance of this model, we could, e.g., compute predictions and look at the confusion matrix when simply rounding the predictions:

```
R> predictions <- predict(sof_binary, type = "response")
R> round_preds <- round(predictions)
R> table(round_preds, as.numeric(emotion$game_outcome))
```

```
round_preds  1  2
          0 76 12
          1 16 80
```

The combination of GAMLSS with functional variables is discussed in Brockhaus *et al.* (2018) and Stöcker *et al.* (2018). For GAMLSS models, **FDboost** builds on the package **gamboostLSS** (Hofner, Mayr, Fenske, Thomas, and Schmid 2020), in which families are implemented to fit GAMLSS. For details on the boosting algorithm to fit GAMLSS, see Mayr *et al.* (2012) and Thomas, Mayr, Bischl, Schmid, Smith, and Hofner (2018). The families in **gamboostLSS** need to model at least two distribution parameters. For an overview of currently implemented response distributions for GAMLSS, we refer to Hofner, Mayr, and Schmid (2016). In **FDboost**, the function `FDboostLSS()` implements GAMLSS with functional data. The interface of `FDboostLSS()` is:

```
FDboostLSS(formula, timeformula, data = list(), families = GaussianLSS(),
  ...)
```

In `formula` a named list of formulas is supplied. Each list entry in the `formula` specifies the potential covariate effects for one of the distribution parameters. The names of the list are the names of the distribution parameters. The argument `families` is used to specify the assumed response distribution with its modeled distribution parameters. The default `families = GaussianLSS()` yields a Gaussian location scale model. In the dots argument, `...`, further arguments passed to `FDboost()` can be supplied. The model object which is fitted by `FDboostLSS()` is a list of '`FDboost`' model objects. It is not possible to automatically fit a

smooth offset within `FDboostLSS()`. Per default, a scalar offset value is used for each distribution parameter. For functional response, it can thus be useful to center the response prior to the model fit. All integration weights for the loss function are set to one, corresponding to the negative log-likelihood of the observation points under a working independent assumption (conditional on all model terms).

For model objects fitted by `FDboostLSS()`, methods to estimate the optimal stopping iterations, as well as methods for plotting and prediction exist. For more details on boosting GAMLSS models, we refer to Hofner *et al.* (2016), which is a tutorial for the package **gamboostLSS**.

Case study (continued): Fossil fuel data

We fit a Gaussian location scale model for the heat value. Such a model is obtained by setting `families = GaussianLSS()`, where the expectation is modeled using the identity link and the standard deviation by a log-link. Mean and standard deviation of the heat value are modeled by different covariates:

$$
\begin{aligned}
&Y_i|\boldsymbol{x}_i \sim N(\mu_i, \sigma_i^2), \\
&\quad \mu_i = \beta_0 + f(z_{\text{H2O},i}) + \int_{\mathcal{S}_{\text{NIR}}} x_{\text{NIR},i}(s_{\text{NIR}})\beta_{\text{NIR}}(s_{\text{NIR}})\, ds_{\text{NIR}} + \int_{\mathcal{S}_{\text{UV}}} x_{\text{UV},i}(s_{\text{UV}})\beta_{\text{UV}}(s_{\text{UV}})\, ds_{\text{UV}}, \\
&\log \sigma_i = \alpha_0 + \alpha_1 z_{\text{H2O},i}.
\end{aligned}
$$

The mean is modeled depending on the water content as well as depending on the NIR and the UVVIS spectrum. The standard deviation is modeled using a log-link and a linear predictor based on the water content. The `formula` has to be specified as a list of two formulas with names `mu` and `sigma` for mean and standard deviation of the normal distribution. We use the noncyclic fitting method that is introduced by Thomas *et al.* (2018).

```
R> fuelSubset$h2o_center <- fuelSubset$h2o - mean(fuelSubset$h2o)
R> library("gamboostLSS")
R> sof_ls <- FDboostLSS(list(mu = heatan ~ bbs(h2o, df = 4) +
+      bsignal(UVVIS, uvvis.lambda, knots = 40, df = 4) +
+      bsignal(NIR, nir.lambda, knots = 40, df = 4),
+    sigma = heatan ~ 1 + bols(h2o_center, df = 2)), timeformula = NULL,
+    data = fuelSubset, families = GaussianLSS(), method = "noncyclic")
R> names(sof_ls)

[1] "mu"    "sigma"
```

The optimal number of boosting iterations is searched on a grid of 1 to 2000 boosting iterations. The algorithm updates in each boosting iteration the base learner that best fits the negative gradient. Thus, in each iteration the additive predictor for only one of the distribution parameters is updated.

```
R> set.seed(123)
R> cvm_sof_ls <- cvrisk(sof_ls, folds = cv(model.weights(sof_ls[[1]]),
+    B = 5), grid = 1:2000, trace = FALSE)
```

The estimated coefficients for the expectation are similar to the effects resulting from the pure mean model. The water content has a negative effect on the standard deviation, with higher water content being associated with lower variability.

8. Variable selection by stability selection

Variable selection can be refined using stability selection (Meinshausen and Bühlmann 2010; Shah and Samworth 2013). Stability selection is a procedure to select influential variables while controlling false discovery rates and maximal model complexity. For component-wise gradient boosting, it is implemented in **mboost** in the function `stabsel()` (Hofner, Boccuto, and Göker 2015), which can also be used for model objects fitted by `FDboost()`. Brockhaus *et al.* (2017) compute function-on-function regression models with more functional covariates than observations and perform variable selection by stability selection. Thomas *et al.* (2018) discuss stability selection for GAMLSS estimated by boosting.

9. Computational characteristics and costs

In order to give rough estimates on how **FDboost** scales up with increasing number of observations N, observation points per response curve G, number of base learners J as well as other data and run-time related set-ups, this section provides some further insights into the algorithm and bottlenecks to bear in mind.

Estimating the run-time of **FDboost** is not straightforward as it depends on the number of boosting iterations, the size of the data set, the number and complexity of base learners, as well as the type and parallelization of resampling. Different loss-functions, i.e., different types of regression should not change the run-time directly, but may require a smaller step-length as explained before which in turn induces a higher number of boosting iterations. In the following simulation study, we use the default value $\nu = 0.1$. **FDboost** scales linearly in the number of iterations, which is why we use a fixed number $m_{\text{stop}} = 50$ in the following. However, note that the initialization of the model can get computationally very expensive, if very complex base learners are defined (see, e.g., Rügamer *et al.* 2018). This is due to a singular-value decomposition of the design matrix of each base learner, which is needed to compute the smoothing parameter corresponding to the pre-defined degrees of freedom and which has cubic run-time in the number of columns of the design matrix. For smooth effects, the number of columns of the design matrix of a base learner is defined by the number of knots. For the simulation study, we use 20 knots for a historical or unrestricted functional effect base learner for function-on-function and scalar-on-function models, respectively. This corresponds to the number of knots used in the `fuelSubset` data and yields rather flexible estimates of functions. For applications where less flexibility is needed, this simulation study can be seen as a worst-case scenario estimate of run-times.

Furthermore, we define the number of observations to be $N \in \{10, 100, 1000\}$, the number of time points to be $G \in \{1, 10, 100, 1000\}$ and the number of base learners to be $J \in \{5, 10, 15\}$. For $G = 1$ scalar-on-function regression is performed, the other settings correspond to function-on-function regression. Due to computational burden, we exclude settings, in which $N = 1000$ and $G = 1000$ at the same time. The simulation was conducted on a Linux server with *Intel Xeon CPU E5-4620 0* with *2.20GHz*, *64 cores* and *512 GB RAM*.

We do not consider resampling or validation here as resampling on k-folds should approximately yield a k-multiple of the original run-time if not parallelized, i.e., run-times scale linearly in the number of folds. With parallelization the run-time can be reduced to the run-time of a single model fit.

The results of the simulation study are visualized in the following, indicating a roughly linear increase in run-time and total allocation of memory by the number of observations (note that both are plotted against $\log_{10}(N)$), a linear increase by the number of observed time points per curve G as well as by the number of base learners J. The $m_{\text{stop}} = 50$ iterations play a comparatively minor role in time and memory consumption after the model has been initialized. Although the hat matrix for each base learner is available after the initialization of the model and therefore the model iterations boil down to simple matrix multiplications, it is noteworthy that for larger problems (in N and/or G) and/or more complex models, these simple operations can add up and the actual model fitting then takes much longer than the model initialization. For some models, in particular quantile regression, several 1,000s of iterations may be necessary, which then also increases computing time considerably. In addition, if it is not possible to parallelize resampling for finding the optimal number of stopping iterations, the run-time might be a k-multiple of this protracted process. We therefore recommend to initially fit the model using only a few iterations m_0 and then update the model as described in Section 5.3 to some more additional iterations m_{add} in order to assess the approximate run-time. By observing the time t_{add} required to update the model from m_0 to m_{add} iterations, an estimate for the final run-time of m_{stop} iterations can be easily obtained by linear extrapolation: $m_{\text{stop}} \cdot \frac{t_{\text{add}}}{m_{\text{add}} - m_0} + m_0$. Since '`FDboost`' objects can be updated repeatedly, the initialized model can be re-used for both the run-time assessment as well as for the final model fit. Note that the total amount of allocated memory can only be interpreted in relative terms for model comparisons, but does not correspond to the maximum amount of consumed memory at one time point, which is considerably smaller.

10. Discussion

The R add-on package **FDboost** provides a comprehensive implementation to fit functional regression models by gradient boosting. The implementation allows to fit regression models with scalar or functional response depending on many covariate effects. The framework includes mean, mean with link function, median and quantile regression models as well as GAMLSS. Various covariate effects are implemented including linear and smooth effects of scalar covariates, linear effects of functional covariates and interaction effects, also between scalar and functional covariates (Rügamer *et al.* 2018). The linear functional effects can have flexible integration limits, for example, to form historical or lag effects (Brockhaus *et al.* 2017). Whenever possible, the effects are represented in the structure of linear array models (Currie *et al.* 2006) to increase computational efficiency (Brockhaus *et al.* 2015). Component-wise gradient boosting allows to fit models in high-dimensional data situations and performs data-driven variable selection. **FDboost** builds on the well tested and modular implementation of **mboost** (Hothorn *et al.* 2020). This facilitates the implementation of further base learners in order to fit new covariate effects and that of families modeling other characteristics of the conditional response distribution.

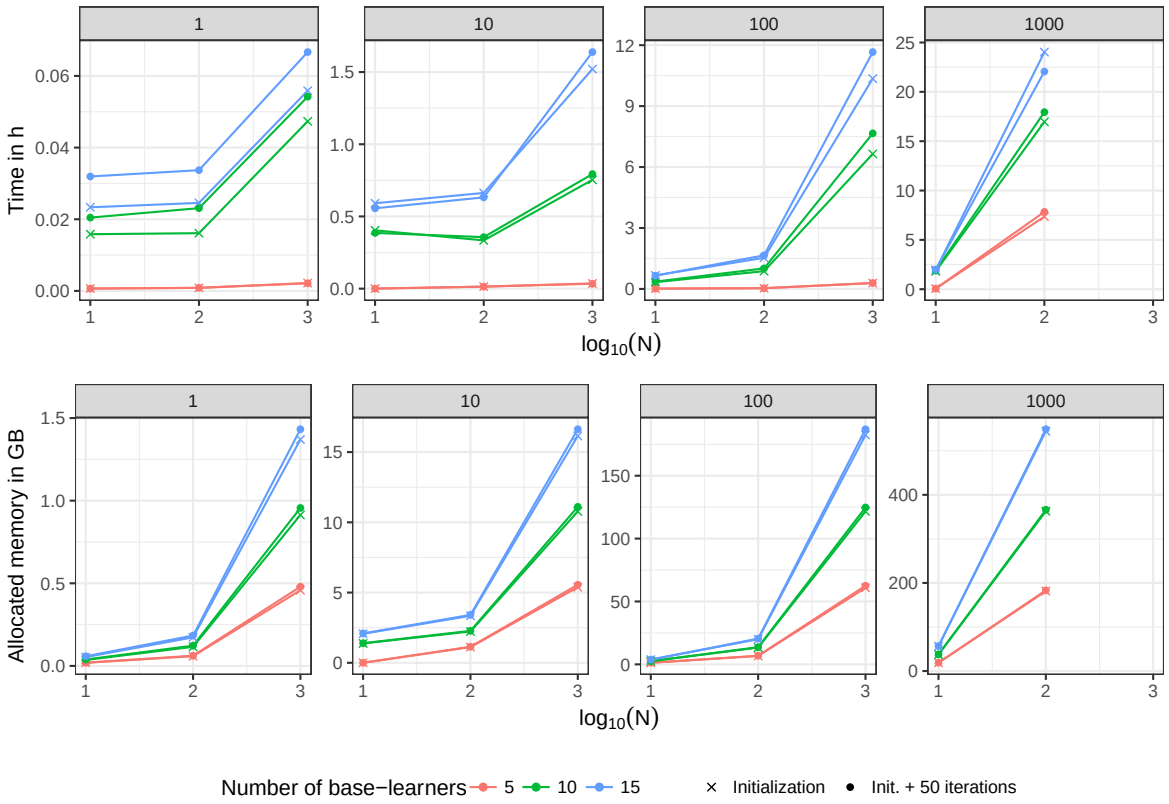

Figure 9: Estimated computational costs of **FDboost** in the simulation study. Different columns correspond to different numbers of observed time points per curve (G) and the number of base learners (J) is visualized by different colors.

Acknowledgments

We acknowledge funding by Emmy Noether grant GR 3793/1-1 from the German Research Foundation. We also thank the two anonymous reviewers whose comments and suggestions helped improve and clarify this manuscript.

References

Brockhaus S, Fuest A, Mayr A, Greven S (2018). "Signal Regression Models for Location, Scale and Shape with an Application to Stock Returns." *Journal of the Royal Statistical Society C*, **67**(3), 665–686. doi:10.1111/rssc.12252.

Brockhaus S, Melcher M, Leisch F, Greven S (2017). "Boosting Flexible Functional Regression Models with a High Number of Functional Historical Effects." *Statistics and Computing*, **27**(4), 913–926. doi:10.1007/s11222-016-9662-1.

Brockhaus S, Rügamer D, Stöcker A (2020). ***FDboost**: Boosting Functional Regression Models*. R package version 1.0-0, URL https://CRAN.R-project.org/package=FDboost.

Brockhaus S, Scheipl F, Hothorn T, Greven S (2015). "The Functional Linear Array Model." *Statistical Modelling*, **15**(3), 279–300. doi:10.1177/1471082x14566913.

Bühlmann P, Hothorn T (2007). "Boosting Algorithms: Regularization, Prediction and Model Fitting." *Statistical Science*, **22**(4), 477–505. doi:10.1214/07-sts242.

Bühlmann P, Yu B (2003). "Boosting with the L_2 Loss: Regression and Classification." *Journal of the American Statistical Association*, **98**(462), 324–339. doi:10.1198/016214503000125.

Currie ID, Durban M, Eilers PHC (2006). "Generalized Linear Array Models with Applications to Multidimensional Smoothing." *Journal of the Royal Statistical Society B*, **68**(2), 259–280. doi:10.1111/j.1467-9868.2006.00543.x.

Efron B (1979). "Bootstrap Methods: Another Look at the Jackknife." *The Annals of Statistics*, **7**(1), 1–26. doi:10.1214/aos/1176344552.

Eilers PHC, Marx BD (1996). "Flexible Smoothing with B-Splines and Penalties." *Statistical Science*, **11**(2), 89–121. doi:10.1214/ss/1038425655.

Fenske N, Kneib T, Hothorn T (2011). "Identifying Risk Factors for Severe Childhood Malnutrition by Boosting Additive Quantile Regression." *Journal of the American Statistical Association*, **106**(494), 494–510. doi:10.1198/jasa.2011.ap09272.

Friedman JH (2001). "Greedy Function Approximation: A Gradient Boosting Machine." *The Annals of Statistics*, **29**(5), 1189–1232. doi:10.1214/aos/1013203451.

Fuchs K, Scheipl F, Greven S (2015). "Penalized Scalar-on-Functions Regression with Interaction Term." *Computational Statistics & Data Analysis*, **81**, 38–51. doi:10.1016/j.csda.2014.07.001.

Gentsch K, Grandjean D, Scherer KR (2014). "Coherence Explored Between Emotion Components: Evidence from Event-Related Potentials and Facial Electromyography." *Biological Psychology*, **98**, 70–81. doi:10.1016/j.biopsycho.2013.11.007.

Goldsmith J, Scheipl F, Huang L, Wrobel J, Di C, Gellar J, Harezlak J, McLean MW, Swihart B, Xiao L, Crainiceanu C, Reiss PT (2019). ***refund**: Regression with Functional Data*. R package version 0.1-21, URL https://CRAN.R-project.org/package=refund.

Greenwell B, Boehmke B, Cunningham J, GBM Developers (2019). ***gbm**: Generalized Boosted Regression Models*. R package version 2.1.5, URL https://CRAN.R-project.org/package=gbm.

Greven S, Scheipl F (2017). "A General Framework for Functional Regression Modelling." *Statistical Modelling*, **17**(1–2), 1–35. doi:10.1177/1471082x16681317.

Hastie TJ, Tibshirani RJ (1993). "Varying-Coefficient Models." *Journal of the Royal Statistical Society B*, **55**(4), 757–796. doi:10.1111/j.2517-6161.1993.tb01939.x.

Hofner B, Boccuto L, Göker M (2015). "Controlling False Discoveries in High-Dimensional Situations: Boosting with Stability Selection." *BMC Bioinformatics*, **16**(1), 1–17. doi:10.1186/s12859-015-0575-3.

Hofner B, Hothorn T, Kneib T, Schmid M (2011). "A Framework for Unbiased Model Selection Based on Boosting." *Journal of Computational and Graphical Statistics*, **20**(4), 956–971. doi:10.1198/jcgs.2011.09220.

Hofner B, Mayr A, Fenske N, Thomas J, Schmid M (2020). ***gamboostLSS**: Boosting Methods for GAMLSS*. R package version 2.0-1.1, URL https://CRAN.R-project.org/package=gamboostLSS.

Hofner B, Mayr A, Robinzonov N, Schmid M (2014). "Model-Based Boosting in R: A Hands-on Tutorial Using the R Package **mboost**." *Computational Statistics*, **29**(1), 3–35. doi:10.1007/s00180-012-0382-5.

Hofner B, Mayr A, Schmid M (2016). "**gamboostLSS**: An R Package for Model Building and Variable Selection in the GAMLSS Framework." *Journal of Statistical Software*, **74**(1), 1–31. doi:10.18637/jss.v074.i01.

Hothorn T, Bühlmann P, Kneib T, Schmid M, Hofner B (2020). ***mboost**: Model-Based Boosting*. R package version 2.9-2, URL https://CRAN.R-project.org/package=mboost.

Hothorn T, Kneib T, Bühlmann P (2013). "Conditional Transformation Models." *Journal of the Royal Statistical Society B*, **76**(1), 3–27. doi:10.1111/rssb.12017.

Huber PJ (1964). "Robust Estimation of a Location Parameter." *The Annals of Mathematical Statistics*, **35**(1), 73–101. doi:10.1214/aoms/1177703732.

Kneib T, Hothorn T, Tutz G (2009). "Variable Selection and Model Choice in Geoadditive Regression Models." *Biometrics*, **65**(2), 626–634. doi:10.1111/j.1541-0420.2008.01112.x.

Koenker R (2005). *Quantile Regression*. Cambridge University Press, Cambridge.

Mayr A, Fenske N, Hofner B, Kneib T, Schmid M (2012). "Generalized Additive Models for Location, Scale and Shape for High Dimensional Data – A Flexible Approach Based on Boosting." *Journal of the Royal Statistical Society C*, **61**(3), 403–427. doi:10.1111/j.1467-9876.2011.01033.x.

Meinshausen N, Bühlmann P (2010). "Stability Selection." *Journal of the Royal Statistical Society B*, **72**(4), 417–473. doi:10.1111/j.1467-9868.2010.00740.x.

Meyer MJ, Coull BA, Versace F, Cinciripini P, Morris JS (2015). "Bayesian Function-on-Function Regression for Multilevel Functional Data." *Biometrics*, **71**(3), 563–574. doi:10.1111/biom.12299.

Morris JS (2015). "Functional Regression." *Annual Review of Statistics and Its Application*, **2**(1), 321–359. doi:10.1146/annurev-statistics-010814-020413.

Morris JS (2017). "Comparison and Contrast of Two General Functional Regression Modelling Frameworks." *Statistical Modelling*, **17**(1–2), 59–85. doi:10.1177/1471082x16681875.

Morris JS, Carroll RJ (2006). "Wavelet-Based Functional Mixed Models." *Journal of the Royal Statistical Society B*, **68**(2), 179–199. doi:10.1111/j.1467-9868.2006.00539.x.

Newey WK, Powell JL (1987). "Asymmetric Least Squares Estimation and Testing." *Econometrica*, **55**(4), 819–847. doi:10.2307/1911031.

Ramsay JO, Silverman BW (2005). *Functional Data Analysis*. Springer-Verlag. doi:10.1007/b98888.

R Core Team (2020). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/.

Ridgeway G (1999). "The State of Boosting." *Computing Science and Statistics*, **31**, 172–181.

Rigby RA, Stasinopoulos DM (2005). "Generalized Additive Models for Location, Scale and Shape." *Journal of the Royal Statistical Society C*, **54**(3), 507–554. doi:10.1111/j.1467-9876.2005.00510.x.

Rügamer D, Brockhaus S, Gentsch K, Scherer K, Greven S (2018). "Boosting Factor-Specific Functional Historical Models for the Detection of Synchronization in Bioelectrical Signals." *Journal of the Royal Statistical Society C*, **67**(3), 621–642. doi:10.1111/rssc.12241.

Ruppert D, Wand MP, Carroll RJ (2003). *Semiparametric Regression*. Cambridge University Press.

Scheipl F, Gertheiss J, Greven S (2016). "Generalized Functional Additive Mixed Models." *Electronic Journal of Statistics*, **10**(1), 1455–1492. doi:10.1214/16-ejs1145.

Scheipl F, Greven S (2016). "Identifiability in Penalized Function-on-Function Regression Models." *Electronic Journal of Statistics*, **10**(1), 495–526. doi:10.1214/16-ejs1123.

Scheipl F, Staicu AM, Greven S (2015). "Functional Additive Mixed Models." *Journal of Computational and Graphical Statistics*, **24**(2), 477–501. doi:10.1080/10618600.2014.901914.

Schmid M, Hothorn T (2008a). "Boosting Additive Models Using Component-Wise P-Splines." *Computational Statistics & Data Analysis*, **53**(2), 298–311. doi:10.1016/j.csda.2008.09.009.

Schmid M, Hothorn T (2008b). "Flexible Boosting of Accelerated Failure Time Models." *BMC Bioinformatics*, **9**(1), 1–13. doi:10.1186/1471-2105-9-269.

Schmid M, Hothorn T, Maloney KO, Weller DE, Potapov S (2011). "Geoadditive Regression Modeling of Stream Biological Condition." *Environmental and Ecological Statistics*, **18**(4), 709–733. doi:10.1007/s10651-010-0158-4.

Schmid M, Potapov S, Pfahlberg A, Hothorn T (2010). "Estimation and Regularization Techniques for Regression Models with Multidimensional Prediction Functions." *Statistics and Computing*, **20**(2), 139–150. doi:10.1007/s11222-009-9162-7.

Shah RD, Samworth RJ (2013). "Variable Selection with Error Control: Another Look at Stability Selection." *Journal of the Royal Statistical Society B*, **75**(1), 55–80. doi:10.1111/j.1467-9868.2011.01034.x.

Sobotka F, Kneib T (2012). "Geoadditive Expectile Regression." *Computational Statistics & Data Analysis*, **56**(4), 755–767. doi:10.1016/j.csda.2010.11.015.

Stöcker A, Brockhaus S, Schaffer S, von Bronk B, Opitz M, Greven S (2018). "Boosting Functional Response Models for Location, Scale and Shape with an Application to Bacterial Competition." arXiv:1809.09881 [stat.ME], URL http://arxiv.org/abs/1809.09881.

Thomas J, Mayr A, Bischl B, Schmid M, Smith A, Hofner B (2018). "Gradient Boosting for Distributional Regression: Faster Tuning and Improved Variable Selection via Noncyclical Updates." *Statistics and Computing*, **28**(3), 673–687. doi:10.1007/s11222-017-9754-6.

Ullah S, Finch CF (2013). "Applications of Functional Data Analysis: A Systematic Review." *BMC Medical Research Methodology*, **13**(43), 1–12. doi:10.1186/1471-2288-13-43.

Wood SN (2011). "Fast Stable Restricted Maximum Likelihood and Marginal Likelihood Estimation of Semiparametric Generalized Linear Models." *Journal of the Royal Statistical Society B*, **73**(1), 3–36. doi:10.1111/j.1467-9868.2010.00749.x.

Wood SN (2017). *Generalized Additive Models: An Introduction with R*. 2nd edition. Chapman & Hal/CRC, Boca Raton, Florida. doi:10.1201/9781315370279.

A. Constraints for effects of scalar covariates

Consider a model for functional response with smooth intercept and an effect that contains a smooth intercept as special case, $\mathbb{E}(Y_i(t)) = \beta_0(t) + h_j(\boldsymbol{x}_i, t)$, and define the mean effect at each point t as $\bar{h}_j(\boldsymbol{x}, t) = \mathbb{E}_X(h_j(\boldsymbol{X}, t))$. This model can be parametrized in different ways, e.g., as

$$\begin{aligned}\mathbb{E}(Y_i(t)) &= \beta_0(t) + h_j(\boldsymbol{x}_i, t) \\ &= \left[\beta_0(t) + \bar{h}_j(\boldsymbol{x}, t)\right] + \left[h_j(\boldsymbol{x}_i, t) - \bar{h}_j(\boldsymbol{x}, t)\right] \\ &= \tilde{\beta}_0(t) + \tilde{h}_j(\boldsymbol{x}, t).\end{aligned}$$

The problem arises as $\bar{h}_j(\boldsymbol{x}, t)$ (or any other smooth function in t) can be shifted between the intercept and the covariate effect. At the level of the design matrices of these effects, this can be explained by the fact that the columns of the design matrix $\boldsymbol{B}_{jY}$ and the columns of the design matrix of the functional intercept are linearly dependent. To obtain identifiable effects, Scheipl *et al.* (2015) propose to center such effects $h_j(\boldsymbol{x}, t)$ at each point t. The centering is achieved by setting the pointwise expectation over the covariate effects to zero on $\mathcal{T}$, i.e., $\mathbb{E}_X(h_j(\boldsymbol{X}, t)) = 0$ for all t, approximated by the sum-to-zero constraint $\sum_{i=1}^{N} h_j(\boldsymbol{x}_i, t) = 0$ for all t. How to enforce such constraints is described in Appendix A of Brockhaus *et al.* (2015). Other constraints to obtain identifiable models are possible. However, this sum-to-zero constraint for each point t yields an intuitive interpretation: The intercept can be interpreted as global mean and the covariate effects can be interpreted as deviations from the smooth intercept.

The constraint is enforced by a basis transformation of the design and penalty matrix. As shown in Brockhaus *et al.* (2015), it is sufficient to apply the constraint on the covariate part of the design and the penalty matrix. Thus, it is not necessary to transform the basis in t direction.

B. Base learners for functional covariates

The base learner `bsignal()` sets up a linear effect of a functional variable $\int_{\mathcal{S}} x_j(s)\beta_j(s)\,ds \approx \boldsymbol{b}_j(\boldsymbol{x})^\top \boldsymbol{\theta}_j$ using P-splines. We approximate the integral numerically as a weighted sum using integration weights $\Delta(s)$ (Wood 2011), see Equation 3:

$$\begin{aligned}\boldsymbol{b}_j(\boldsymbol{x}_i)^\top &= \left[\sum_{r=1}^{R} \Delta(s_r) x_i(s_r)\phi_1(s_r) \;\cdots\; \sum_{r=1}^{R} \Delta(s_r) x_i(s_r)\phi_{K_j}(s_r)\right] \\ &\approx \left[\int_{\mathcal{S}} x_i(s)\phi_1(s)\,ds \;\cdots\; \int_{\mathcal{S}} x_i(s)\phi_{K_j}(s)\,ds\right],\end{aligned}$$

where $\phi_k(s_r)$, $k = 1, \ldots, K_j$ are B-splines evaluated at s_r. The corresponding penalty matrix $\boldsymbol{P}_j$ is a squared difference matrix and thus, the smooth effect $\beta_j(s)$ in s is represented by P-splines.

Using the base learner `bfpc()` the linear functional effect $\int_{\mathcal{S}} x_j(s)\beta_j(s)\,ds$ is specified using an FPC basis. The functional covariate $x_j(s)$ and the coefficient $\beta_j(s)$ are both represented in the basis that is spanned by the functional principal components (FPCs, see, e.g., Ramsay and Silverman 2005, Chapters 8 and 9) of $x_j(s)$. Let $X_j(s)$ be a zero-mean stochastic process

in the space of all square-integrable functions $L^2(\mathcal{S})$. Let $x_{ij}(s)$ be the observations of the copies $X_{ij}(s)$ of this process. We denote the eigenvalues of the auto-covariance of $X_j(s)$ as $\zeta_1 \geq \zeta_2 \geq \cdots \geq 0$ and the corresponding eigenfunctions as $e_k(s)$, $k \in \mathbb{N}$. The eigenfunctions $\{e_k(s), k \in \mathbb{N}\}$ form an orthonormal basis for the $L^2(\mathcal{S})$. Using the Karhunen-Loève theorem, the functional covariate can be represented as weighted sum

$$X_{ij}(s) = \sum_{k=1}^{\infty} Z_{ik} e_k(s),$$

where Z_{ik} are uncorrelated mean zero random variables with variance ζ_k and realizations z_{ik}. In practice, the infinite sum is truncated at a certain value K_j. Representing the functional covariate and the coefficient function by this truncated basis with weights θ_l and z_{ik}, respectively, the effect simplifies to

$$\int_{\mathcal{S}} x_{ij}(s)\beta_j(s)\, ds \approx \sum_{k,l=1}^{K_j} \int_{\mathcal{S}} z_{ik} e_k(s) e_l(s) \theta_l \, ds = \sum_{k=1}^{K_j} z_{ik}\theta_k,$$

as the eigenfunctions $e_k(s)$ are orthonormal. Thus, this approach is equivalent to using the (estimated) first K_j FPC scores z_{ik} as linear covariates. The number of eigenfunctions is usually chosen such that the truncated basis explains a fixed proportion of the total variability of the covariate, for example 99% (cf. Morris 2015). This truncation achieves regularized effects, as the effect can only lie in the space spanned by the first K_j eigenfunctions. For the penalty matrix $\boldsymbol{P}_j$ the identity matrix is used in `bfpc()`.

For scalar response, the base learners `bsignal()` and `bfpc()` yield the effect $\int_{\mathcal{S}} x_j(s)\beta_j(s)\, ds$. Combining them with a smooth effect in t using `bbs()`, they can be used to fit effects for function-on-function regression $\int_{\mathcal{S}} x_j(s)\beta_j(s,t)\, ds$.

The base learner `bhist()` allows to specify functional linear effects with integration limits depending on t, $\int_{l(t)}^{u(t)} x(s)\beta(s,t)\, ds$. Per default, a historical effects with limits $[l(t), u(t)] = [T_1, t]$ is fitted. The integral is approximated by a numerical integration scheme (Scheipl *et al.* 2015). We transform the observations of the functional covariate $x_j(s_r)$ such that they contain the integration limits and the weights for numerical integration. We define $\tilde{x}_j(s_r, t) = I\left(l(t) \leq s_r \leq u(t)\right) \Delta(s_r) x_j(s_r)$, with indicator function $I(\cdot)$ and integration weights $\Delta(s_r)$. The marginal basis over the covariates $\boldsymbol{x}$, which in this case also depends on t, is:

$$\begin{aligned}
\boldsymbol{b}_{jY}(\boldsymbol{x}_i, t)^\top &= \left[\sum_{r=1}^{R} \tilde{x}_j(s_r, t)\phi_1(s_r) \;\cdots\; \sum_{r=1}^{R} \tilde{x}_j(s_r, t)\phi_{K_j}(s_r)\right] \otimes \left[\phi_1(t_g) \;\cdots\; \phi_{K_Y}(t_g)\right] \\
&\approx \left[\int_{l(t)}^{u(t)} x_i(s)\phi_1(s)\, ds \;\cdots\; \int_{l(t)}^{u(t)} x_i(s)\phi_{K_j}(s)\, ds\right] \otimes \left[\phi_1(t_g) \;\cdots\; \phi_{K_Y}(t_g)\right].
\end{aligned}$$

The isotropic penalty in Equation 8 is used with squared difference matrices as marginal penalties to form P-splines bases for the s and t direction of $\beta(s,t)$.

For a concurrent effect $x(t)\beta(t)$, the base learner `bconcurrent()` can be used. The smooth effect $\beta(t)$ in t is expanded by P-splines.

C. Row tensor product and Kronecker product bases

In the R package **mboost** (Hothorn *et al.* 2020), the Kronecker product of two base learners is implemented as `%O%`. The row-wise tensor product of two base learners is implemented in the operator `%X%`. The row-wise tensor product of two marginal design matrices, $\boldsymbol{B}_j \in \mathbb{R}^{n \times K_j}$ and $\boldsymbol{B}_Y \in \mathbb{R}^{n \times K_Y}$, is defined as $n \times K_j K_Y$ matrix

$$\boldsymbol{B}_j \odot \boldsymbol{B}_Y = (\boldsymbol{B}_j \otimes \mathbf{1}_{K_Y}^\top) \cdot (\mathbf{1}_{K_j}^\top \otimes \boldsymbol{B}_Y),$$

where $\cdot$ denotes entry-wise multiplication and $\mathbf{1}_K$ is the K-dimensional vector of ones. The operators `%X%` and `%O%` use the Kronecker product or the row-wise tensor product to compute the design matrix. The penalty is computed according to Equation 7. When `%X%` or `%O%` is called with specified argument `df` in both marginal base learners, the degrees of freedom of the composed effect are computed as the product of the two specified degrees of freedom. Then, only one smoothing parameter is computed for an isotropic penalty like in Equation 8. Consider, for example, the composed base learner `bols(z1, df = df1) %O% bbs(t, df = df2)`. The base learner `bols()` specifies a linear effect. The base learner `bbs()` specifies a smooth effect represented by P-splines. Thus, the composed base learner yields the effect $z_1 \beta_j(t)$, which is linear in z_1 and smooth in t. The global degrees of freedom for the composed base learner are computed as $df_j =$ `df1 * df2`. The corresponding smoothing parameter λ_j is computed by Demmler-Reinsch orthogonalization (Ruppert, Wand, and Carroll 2003, Appendix B.1.1).

For array models, `FDboost()` connects the effects of `formula` and `timeformula` by the operator `%O%`, yielding `b_1 %O% b_Y + ...+ b_J %O% b_Y`. The operator `%O%` uses the array framework of Currie *et al.* (2006) to efficiently implement such effects in boosting (Hothorn, Kneib, and Bühlmann 2013). If it is not possible to use the array framework, e.g., if the response is observed on curve-specific grids or for historical effects, the design matrix is computed as row-wise tensor product basis, i.e., using the operator `%X%`. Within the function `FDboost()` the appropriate operator is used automatically. When the marginal base learners are supplied with specified degrees of freedom (argument `df`), `%O%` and `%X%` use the isotropic penalty (8).

The anisotropic penalty (7) is obtained if the smoothing parameter is specified in both marginal base learners; for instance, as `bols(z1, lambda = lambda1) %O% bbs(t, lambda = lambda2)`. However, it is hard to control the degrees of freedom in this case such that each base learner in the model has the same number of degrees of freedom. Thus, specifying the smoothing parameter λ in both marginal base learners is hardly applicable in practice.

In some cases, one only wants to penalize the basis in t direction. In this case, the penalty in Equation 9 can be used. Such a penalty is obtained using the operators `%A0%` or `%Xa0%`, for the Kronecker and the row-wise tensor product basis, respectively. When `%A0%` or `%Xa0%` are used to form an effect with penalty (9), the number of degrees of freedom in the first base learner has to be equal to the number of its columns. Consider, `bols(z1, df = 1, intercept = FALSE) %A0% bbs(t, df = df2)`, with a metric variable `z1`. This specification implies $\boldsymbol{b}_j(\boldsymbol{x}_i) = z_{i1}$ and $\boldsymbol{P}_j = \mathbf{0}$ for the `bols()` base learner. The `bbs()` base learner sets up a design matrix of B-spline evaluations in t and a squared difference matrix as penalty matrix.

Linking `formula` and `timeformula` in `FDboost()` to representation (6), the J base learners in `formula` correspond to the J marginal bases $\boldsymbol{b}_j$ and the base learners in `timeformula` corresponds to the marginal basis $\boldsymbol{b}_Y$. If it is possible to represent the effects as Kronecker

product, the base learners are combined by `%O%`. Otherwise, the row-wise tensor product `%X%` is used to combine the marginal bases.

Consider, for example, `formula = Y ~ b_1 + b_2 + ...+ b_J`, and `timeformula = ~ b_Y`. For an array model, this yields `Y ~ b_1 %O% b_Y + b_2 %O% b_Y + ... + b_J %O% b_Y`. If `formula` contains base learners that are composed of two base learners by `%O%` or `%A0%`, those effects are not expanded with `timeformula`, allowing for model specifications with different effects in t direction. For example, `formula = Y ~ b_1 + b_2 %A0% b_Y0`, and `timeformula = ~ b_Y`, with non-linear base learner `b_Y` and linear base learner `b_Y0`, yield `Y ~ b_1 %O% b_Y + b_2 %A0% b_Y0`.

D. Example code for resampling with repeated measurements

In the following, we search the optimal stopping iteration for model (13), which contains a linear effect for the game condition power and a person-specific effect.

We search the optimal stopping iteration by a 5-fold cross-validation. The resampling is done on the level of curves, assuming that the observations per subject are independent conditional on the subject-specific effects. We use the function `applyFolds()` for the resampling.

```
R> set.seed(123)
R> folds_bs <- cv(weights = rep(1, fos_random_power$ydim[1]),
+    type = "kfold", B = 5)
R> cvm <- applyFolds(fos_random_power, folds = folds_bs, grid = 1:200)
```

The optimal stopping iteration is estimated to be 200, which is the upper limit of the searched grid. Thus, the resampling has to be rerun with a higher maximal number of boosting iterations.

To resample the observations on the level of independent observation units, the folds can be set up on the level of subjects. The corresponding folds for a leave-one-subject-out cross-validation, which are then passed to `applyFolds()`, could be constructed as follows:

```
R> set.seed(123)
R> folds_bs_long_subject <- sapply(levels(emotion$subject),
+    function(x) as.numeric(x != emotion$subject))
```

E. Fitting factor-specific historical models

In this section we provide code to fit a more complex and realistic model to the emotion component data. As the EMG signal might depend on all three study settings (`power`, `game_outcome`, `control`) as well as their interactions, and the influence of the EEG signal might also be specific for each setting as well as for each subject, we assume the following

model (cf. Rügamer *et al.* 2018):

$$\begin{aligned}\mathbb{E}(Y_{\text{EMG},i,j}(t)|\boldsymbol{x}_{i,j}) = \beta_0(t) &+ \gamma_{\text{subject},j}(t) \\ &+ I(x_{\text{power},i,j}=1)\beta_{\text{power}}(t) \\ &+ I(x_{\text{outcome},i,j}=1)\beta_{\text{outcome}}(t) \\ &+ I(x_{\text{control},i,j}=1)\beta_{\text{control}}(t) \\ &+ I(x_{\text{power},i,j}=1, x_{\text{outcome},i,j}=1)\beta_{\text{power,outcome}}(t) \\ &+ I(x_{\text{outcome},i,j}=1, x_{\text{control},i,j}=1)\beta_{\text{outcome,control}}(t) \\ &+ I(x_{\text{power},i,j}=1, x_{\text{control},i,j}=1)\beta_{\text{power,control}}(t) \\ &+ I(x_{\text{power},i,j}=1, x_{\text{outcome},i,j}=1, x_{\text{control},i,j}=1)\cdot \\ &\quad \beta_{\text{power,outcome,control},i}(t) \\ &+ \int_0^{t-3} x_{\text{EMG},i,j}(s)\beta_{\text{EMG}}(s,t)ds \\ &+ \int_0^{t-3} x_{\text{EMG},i,j}(s)\gamma_{\text{EMG},i}(s,t)ds \\ &+ \int_0^{t-3} x_{\text{EMG},i,j}(s)\zeta_{\text{EMG},j}(s,t)ds + \varepsilon_{i,j}(t)\end{aligned} \tag{16}$$

for observation $i = 1, \ldots, 8$ corresponding to the 8 different game conditions of subject $j = 1, \ldots, 23$. The model was proposed in Rügamer *et al.* (2018), which extended historical models by allowing for factor-specific historical effects. To our knowledge, **FDboost** so far is the only software capable of fitting such effects.

To this end, we have to define the 3 two-way interactions `power.outcome`, `outcome.control`, `power.control`, 1 three-way interaction `gamecondition` and an '`hmatrix`' object `X1h`. The object is needed for the function `bhistx`, which in turn allows to combine historical effects with factor variables using the row-wise tensor product operator `%X%`. To construct a '`hmatrix`' object, the time and an identifier for each curve in long format must be supplied along with the original response. The corresponding model fit in R takes around 75 minutes to fit the model with 5000 iterations and needs approximately a maximum of 15GB RAM at once. We further allow for an anisotropic penalty for all factor effects that are time-dependent, which is achieved by using the `%A%`-operator.

This example also demonstrates how the degrees of freedom can be defined to be equal across all base learners (in this case $df_j = 20$), which is explained in Appendix C.

```
R> N <- nrow(emotion$EEG)
R> G <- ncol(emotion$EEG)
R> emotion$id_repeated <- rep(1:N, G)
R> emotion$EEG <- scale(emotion$EEG, center = TRUE, scale = FALSE)
R> X1h <- hmatrix(time = rep(emotion$t, each = N),
+    id = emotion$id_repeated, x = emotion$EEG)
R> emotion$power.outcome <- interaction(emotion$power, emotion$game_outcome)
R> emotion$outcome.control <- interaction(emotion$game_outcome,
+    emotion$control)
R> emotion$power.control <- interaction(emotion$power, emotion$control)
R> emotion$gamecondition <- interaction(emotion$power, emotion$game_outcome,
```

```
+    emotion$control)
R> emotion$X1h <- I(X1h)
R> mod <- FDboost(EMG ~ 1 + brandomc(subject, df = 5) %A% bbs(t, df = 4) +
+    bolsc(power, df = 2, intercept = TRUE) %A% bbs(t, df = 10) +
+    bolsc(game_outcome, df = 2, intercept = TRUE) %A% bbs(t, df = 10) +
+    bolsc(control, df = 2, intercept = TRUE)  %A% bbs(t, df = 10)+
+    bolsc(power.outcome, intercept = TRUE, df = 2) %A% bbs(t, df = 10) +
+    bolsc(outcome.control, intercept = TRUE, df = 2) %A% bbs(t, df = 10) +
+    bolsc(power.control, intercept = TRUE, df = 2) %A% bbs(t, df = 10) +
+    bolsc(gamecondition, intercept = TRUE, df = 2) %A% bbs(t, df = 10) +
+    bhistx(X1h, limits = function(s, t) { s < t - 3 }, df = 20, knots = 10,
+      differences = 2, standard = "length") +
+    bhistx(X1h, limits = function(s, t) { s < t - 3 }, df = 5, knots = 10,
+      differences = 2, standard = "length") %X%
+    bolsc(gamecondition, df = 4, intercept = TRUE, index = id_repeated) +
+    bhistx(X1h, limits = function(s, t) { s < t - 3 }, df = 5, knots = 10,
+      differences = 2, standard = "length") %X%
+    brandomc(subject, df = 4, index = id_repeated),
+    control = boost_control(mstop = 5000, trace = TRUE),
+    timeformula = ~ bbs(t), data = emotion)
```


Affiliation:

Sarah Brockhaus, David Rügamer, Sonja Greven
Ludwig-Maximilans-Universität München
E-mail: sarah.brockhaus@stat.uni-muenchen.de,
david.ruegamer@stat.uni-muenchen.de,
sonja.greven@stat.uni-muenchen.de

Journal of Statistical Software http://www.jstatsoft.org/
published by the Foundation for Open Access Statistics http://www.foastat.org/
August 2020, Volume 94, Issue 10 *Submitted:* 2017-05-30
doi:10.18637/jss.v094.i10 *Accepted:* 2018-11-21