```
logLik.bamlss <- function(object, ..., optimizer = FALSE, samples = FALSE)
{
  Call <- match.call()
  Call <- Call[!(names(Call) %in% c("optimizer", "samples"))]
  mn <- as.character(Call)[-1L]
  object <- list(object, ...)
  mstop <- object$mstop
  if(any(names(object) != "")) {
    i <- names(object) == ""
    object <- object[i]
    mn <- mn[i]
  }
  object <- object[mn != "mstop"]
```

Advanced Bayesian Methods: **Theory and Applications in R**

Big Data and Variable Selection

Nikolaus Umlauf
https://nikum.org/abm.html

Estimation

The main building block of regression model algorithms is the probability density function $d_y(\mathbf{y}|\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_K)$.

Estimation typically requires to evaluate the log-likelihood

$$\ell(\boldsymbol{\beta}; \mathbf{y}, \mathbf{X}) \;=\; \sum_{i=1}^{n} \log\, d_y(y_i; \theta_1(\mathbf{x}_i; \boldsymbol{\beta}_1),\; \ldots\;,\; \theta_K(\mathbf{x}_i; \boldsymbol{\beta}_K)),$$

with $\mathbf{X} = (\mathbf{X}_1, \ldots, \mathbf{X}_K)$.

The log-posterior (frequentist penalized log-likelihood)

$$\log\, \pi(\boldsymbol{\beta}, \boldsymbol{\tau}; \mathbf{y}, \mathbf{X}, \boldsymbol{\alpha}) \propto \ell(\boldsymbol{\beta}; \mathbf{y}, \mathbf{X}) + \sum_{k=1}^{K} \sum_{j=1}^{J_k} \left[\log\, p_{jk}(\boldsymbol{\beta}_{jk}; \boldsymbol{\tau}_{jk}, \boldsymbol{\alpha}_{jk})\right],$$

where $p_{jk}(\cdot)$ are priors, $\boldsymbol{\tau}_{jk}$ (smoothing) variances and $\boldsymbol{\alpha}_{jk}$ fixed hyper parameters.

Priors $p_{jk}(\cdot)$

For simple linear effects $\mathbf{X}_{jk}\boldsymbol{\beta}_{jk}$: $p_{jk}(\boldsymbol{\beta}_{jk}) \propto const.$

For the smooth terms:

$$p_{jk}(\boldsymbol{\beta}_{jk}; \boldsymbol{\tau}_{jk}, \boldsymbol{\alpha}_{jk}) \propto d_{\boldsymbol{\beta}_{jk}}(\boldsymbol{\beta}_{jk} | \boldsymbol{\tau}_{jk}; \boldsymbol{\alpha}_{\boldsymbol{\beta}_{jk}}) \cdot d_{\boldsymbol{\tau}_{jk}}(\boldsymbol{\tau}_{jk} | \boldsymbol{\alpha}_{\boldsymbol{\tau}_{jk}}).$$

Using a basis function approach a common choice is

$$d_{\boldsymbol{\beta}_{jk}}(\boldsymbol{\beta}_{jk} | \boldsymbol{\tau}_{jk}, \boldsymbol{\alpha}_{\boldsymbol{\beta}_{jk}}) \propto |\mathbf{P}_{jk}(\boldsymbol{\tau}_{jk})|^{\frac{1}{2}} \exp\left(-\frac{1}{2}\boldsymbol{\beta}_{jk}^{\top}\mathbf{P}_{jk}(\boldsymbol{\tau}_{jk})\boldsymbol{\beta}_{jk}\right).$$

Precision matrix $\mathbf{P}_{jk}(\boldsymbol{\tau}_{jk})$ derived from prespecified penalty matrices $\boldsymbol{\alpha}_{\boldsymbol{\beta}_{jk}} = \{\mathbf{K}_{1jk}, \ldots, \mathbf{K}_{Ljk}\}$.

The variances parameters $\boldsymbol{\tau}_{jk}$ are equivalent to the inverse smoothing parameters in a frequentist approach.

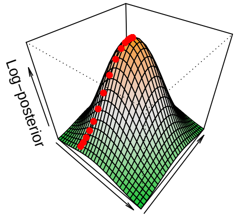
Estimation

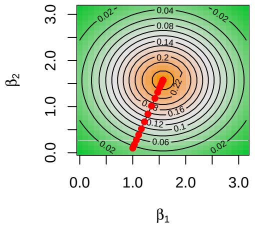
Bayesian point estimates of parameters are obtained by:

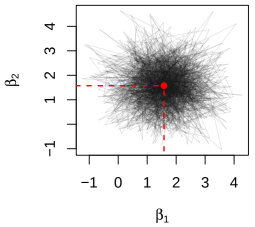
1. Maximization of the log-posterior for posterior mode estimation.
2. Solving high dimensional integrals, e.g., for posterior mean or median estimation.

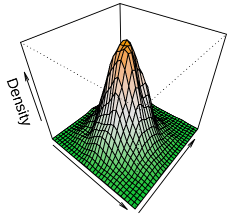
Problems 1 and 2 are commonly solved by computer intensive iterative algorithms of the following type:

$$(\boldsymbol{\beta}^{[t+1]}, \boldsymbol{\tau}^{[t+1]}) = \mathrm{U}(\boldsymbol{\beta}^{[t]}, \boldsymbol{\tau}^{[t]}; \mathbf{y}, \mathbf{X}, \boldsymbol{\alpha}).$$









Efficient Updating (1)

Typically the number of different observations $x_{(1)} < x_{(2)} < \cdots < x_{(m)}$ in a design matrix $\mathbf{X}$ is much smaller than the total number n of observations, i.e., $m \ll n$.
For **sorted** observations x_i:

- Index vector **ind** with $\mathbf{ind}[i] \in \{1, \ldots, m\}$, i.e., if $x_i = x_{(s)}$ then $\mathbf{ind}[i] = s$.
- Decompose the design matrix in $\mathbf{X} = \mathbf{D}\mathbf{P}\tilde{\mathbf{X}}$ where
- $\tilde{\mathbf{X}}$ is the $m \times L$ reduced design matrix for the different and sorted observations $x_{(1)}, \ldots, x_{(m)}$, i.e., $\tilde{\mathbf{X}}[s, l] = X_l(x_s)$, $s = 1, \ldots, m$, $l = 1, \ldots, L$,
- $\mathbf{P}$ is a $n \times L$ permutation matrix, which reverts the sorting, i.e., $\mathbf{P}[i, s] = I(\mathbf{ind}[i] = s)$.
- $\mathbf{D}$ is a diagonal matrix, e.g., for varying coefficient models or $\mathbf{D} = \mathbf{I}$ for simple additive terms.
- For the vector of function evaluations we obtain $\mathbf{f} = \mathbf{X}\beta = \mathbf{D}\mathbf{P}\tilde{\mathbf{X}}\beta$.

Efficient Updating (1)

Using the permutation, we get

$$\mathbf{X}_{jk}^\top \mathbf{W}_{kk} \mathbf{X}_{jk} = \tilde{\mathbf{X}}_{jk}^\top \mathbf{P}_{jk}^\top \mathbf{D}_{jk}^\top \mathbf{W}_{kk} \mathbf{D}_{jk} \mathbf{P}_{jk} \tilde{\mathbf{X}}_{jk} = \tilde{\mathbf{X}}_{jk}^\top \tilde{\mathbf{W}} \tilde{\mathbf{X}}_{jk},$$

where

$$\tilde{\mathbf{W}} = \mathbf{P}_{jk}^\top \mathbf{D}_{jk}^\top \mathbf{W}_{kk} \mathbf{D}_{jk} \mathbf{P}_{jk} = diag(\tilde{w}_1, \ldots, \tilde{w}_{m_{jk}})$$

and the “reduced” weights $\tilde{w}_s$, are given by

$$\tilde{w}_s = \sum_{i\,:\,\mathbf{ind}[i]=s} z_i^2 \mathbf{W}_{kk}[i,i].$$

The weights $\tilde{w}_s$ can be computed by first initializing $\tilde{w}_s = 0$ followed by a simple loop:

For $i = 1, \ldots, n$ add $z_i^2 \mathbf{W}_{kk}[i,i]$ to $\tilde{w}_{\mathbf{ind}[i]}$.

Efficient Updating (1)

For $\mathbf{X}_{jk}^\top \mathbf{W}_{kk}(\mathbf{z}_k - \boldsymbol{\eta}_{k,-j}^{(t)})$ we obtain

$$\mathbf{X}_{jk}^\top \mathbf{W}_{kk}\mathbf{r} = \tilde{\mathbf{X}}_{jk}^\top \mathbf{P}_{jk}^\top \mathbf{D}_{jk}^\top \mathbf{W}_{kk}\mathbf{r} = \tilde{\mathbf{X}}_{jk}^\top \tilde{\mathbf{r}},$$

with partial residuals $\mathbf{r} = \mathbf{z}_k - \boldsymbol{\eta}_{k,-j}^{(t)}$.

The “reduced” partial residuals yield a $m_{jk} \times 1$ vector $\tilde{\mathbf{r}} = (\tilde{r}_1, \dots, \tilde{r}_{m_{jk}})^\top$ given by

$$\tilde{r}_s = \sum_{i\,:\,\mathbf{ind}[i]=s} z_i\, \mathbf{W}_{kk}[i,i]\, r_i.$$

The $\tilde{r}_s$ are computed by first initializing $\tilde{r}_s = 0$ followed by the loop:

For $i = 1, \dots, n$ add $z_i \mathbf{W}_{kk}[i,i]\, r_i$ to $\tilde{r}_{\mathbf{ind}[i]}$.

Efficient Updating (1)

Example using the `IndiaNutrition` data set.

```
R> dim(IndiaNutrition)
```

```
[1] 25134    13
```

```
R> X <- smoothCon(s(mage, bs = "ps", k = 22),
+    IndiaNutrition, NULL)[[1]]$X
R> dim(X)
```

```
[1] 25134    22
```

```
R> i <- match.index(X)
R> tX <- X[i$nodups, ]
R> dim(tX)
```

```
[1] 26 22
```

```
R> print(object.size(X), units = "Mb")
```

```
4.2 Mb
```

```
R> print(object.size(tX), units = "Kb")
```

```
4.7 Kb
```

Sparsity

B-spline design matrix:

$$\mathbf{X}_{jk} = \begin{pmatrix} 0.496 & 0.504 & 0 & 0 & 0 & 0 \\ 0.153 & 0.747 & 0.1 & 0 & 0 & 0 \\ 0.006 & 0.597 & 0.397 & 0 & 0 & 0 \\ 0 & 0.221 & 0.723 & 0.056 & 0 & 0 \\ 0 & 0.025 & 0.673 & 0.303 & 0 & 0 \\ 0 & 0 & 0.303 & 0.673 & 0.025 & 0 \\ 0 & 0 & 0.056 & 0.723 & 0.221 & 0 \\ 0 & 0 & 0 & 0.397 & 0.597 & 0.006 \\ 0 & 0 & 0 & 0.1 & 0.747 & 0.153 \\ 0 & 0 & 0 & 0 & 0.504 & 0.496 \end{pmatrix}$$

Sparsity

B-spline penalty matrix:

$$\mathbf{K}_{jk} = \begin{pmatrix} 0.25 & -0.25 & 0 & 0 & 0 & 0 \\ -0.25 & 0.5 & -0.25 & 0 & 0 & 0 \\ 0 & -0.25 & 0.5 & -0.25 & 0 & 0 \\ 0 & 0 & -0.25 & 0.5 & -0.25 & 0 \\ 0 & 0 & 0 & -0.25 & 0.5 & -0.25 \\ 0 & 0 & 0 & 0 & -0.25 & 0.25 \end{pmatrix}$$

Sparsity

Markov random fields (MRF) design matrix:

$$\mathbf{X}_{jk} = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix}$$

Efficient Updating (2)

Products $\tilde{\mathbf{X}}_{jk}^\top \tilde{\mathbf{W}} \tilde{\mathbf{X}}_{jk}$ and $\tilde{\mathbf{X}}_{jk}^\top \tilde{\mathbf{r}}$ are stored in sparse matrix format.

Nonzero entries are stored in a vector $\mathbf{C}$ $(n_x \times 1)$. E.g., the t-th entry $\mathbf{C}[t]$ corresponds to

$$\mathbf{C}[t] = \sum_{s=1}^{m_{jk}} \tilde{w}_s \tilde{\mathbf{X}}_{jk}[s, r] \tilde{\mathbf{X}}_{jk}[s, l],$$

hence, most products are zero. Store the nonzero products in $\mathbf{h}_1$, the nonzero index s in $\mathbf{h}_2$ and the position of the first element in $\mathbf{h}_3$. Computation only requires

$$\mathbf{C}[t] = \sum_{s=\mathbf{h}_3[t]}^{\mathbf{h}_3[t+1]-1} \tilde{w}_{\mathbf{h}_2[s]} \mathbf{h}_1[s].$$

Similarly for $\tilde{\mathbf{X}}_{jk}^\top \tilde{\mathbf{r}}$, etc.

Efficient Updating (2)

Example using the `IndiaNutrition` data set.

```
R> H <- sparse.matrix.index(tX)
R> print(head(H))
     [,1] [,2] [,3] [,4]
[1,]    4    5    6    7
[2,]    3    4    5    6
[3,]    2    3    4    5
[4,]    9   10   11   12
[5,]    4    5    6    7
[6,]    5    6    7    8
R> print(nrow(X) * ncol(X))
[1] 552948
R> print(nrow(tX) * ncol(tX))
[1] 572
R> print(nrow(H) * ncol(H))
[1] 104
R> print(object.size(H), units = "Kb")
0.6 Kb
```

Scaleable Distributional Learning

Consider the following updating scheme

$$\beta_k^{[t+1]} = \mathtt{U}_k(\beta_k^{[t]}; \cdot) = \beta_k^{[t]} - \mathbf{H}_{kk}\left(\beta_k^{[t]}\right)^{-1}\mathbf{s}\left(\beta_k^{[t]}\right).$$

Assuming model terms that can be written as a matrix product of a design matrix and coefficients we obtain an iteratively weighted least squares scheme given by

$$\beta_{jk}^{[t+1]} = \mathtt{U}_{jk}(\beta_{jk}^{[t]}; \cdot) = (\mathbf{X}_{jk}^\top \mathbf{W}_{kk}\mathbf{X}_{jk} + \mathbf{G}_{jk}(\boldsymbol{\tau}_{jk}))^{-1}\mathbf{X}_{jk}^\top \mathbf{W}_{kk}(\mathbf{z}_k - \boldsymbol{\eta}_{k,-j}^{[t+1]}),$$

with working observations $\mathbf{z}_k = \boldsymbol{\eta}_k^{[t]} + \mathbf{W}_{kk}^{-1\,[t]}\mathbf{u}_k^{[t]}$, working weights $\mathbf{W}_{kk}^{-1\,[t]}$ and score vector $\mathbf{u}_k^{[t]}$.

Scaleable Distributional Learning

Instead of using all observations of the data, we only use a randomly chosen **subset** denoted by the subindex $[\mathbf{s}]$ in one updating step

$$\begin{aligned}\boldsymbol{\beta}_{jk}^{[t+1]} &= \nu \cdot (\mathbf{X}_{[\mathbf{s}],jk}^{\top}\mathbf{W}_{[\mathbf{s}],kk}\mathbf{X}_{[\mathbf{s}],jk} + \mathbf{G}_{jk}(\boldsymbol{\tau}_{jk}))^{-1}\mathbf{X}_{[\mathbf{s}],jk}^{\top}\mathbf{W}_{[\mathbf{s}],kk}(\mathbf{z}_{[\mathbf{s}],k} - \boldsymbol{\eta}_{[\mathbf{s}],k,-j}^{[t+1]}) + \\ &\quad (1-\nu)\cdot\boldsymbol{\beta}_{jk}^{[t]},\end{aligned}$$

where ν is a weight parameter which specifies how much the parameters at iteration $t+1$ are influenced by parameters of the previous iteration t.

Use **flat file** format for each $\mathbf{X}_{jk}$, i.e., only batch $[s]$ is in memory. This way, we can estimate models with **really** large datasets!

Scaleable Distributional Learning

Mimics a second order **stochastic gradient descent** (SGD) algorithm

$$\beta_{jk}^{[t+1]} = \beta_{jk}^{[t]} + \nu \cdot (\beta_{jk,[\mathbf{s}]} - \beta_{jk}^{[t]}) = \beta_{jk}^{[t]} + \nu \cdot \boldsymbol{\delta}_{jk}^{[t]},$$

and $\delta_{jk}^{[t]}$ is composed from first and second order derivative information with

$$\begin{aligned}
\boldsymbol{\delta}_{jk}^{[t]} &= \beta_{jk,[\mathbf{s}]} - \beta_{jk}^{[t]} \\
&= \left[\beta_{jk}^{[t]} - \mathbf{H}_{[\mathbf{s}],kk}\left(\beta_{jk}^{[t]}\right)^{-1} \mathbf{s}_{[\mathbf{s}]}\left(\beta_{jk}^{[t]}\right)\right] - \beta_{jk}^{[t]} \\
&= -\mathbf{H}_{[\mathbf{s}],kk}\left(\beta_{jk}^{[t]}\right)^{-1} \mathbf{s}_{[\mathbf{s}]}\left(\beta_{jk}^{[t]}\right)
\end{aligned}$$

Hence, the updating step length is adaptive.

Scaleable Distributional Learning

The idea is to select $\boldsymbol{\tau}_{jk}$ using a stepwise algorithm which is based on an **"out-of-sample" criterion**, i.e., the criterion $C(\cdot)$ is evaluated on another batch denoted by $[\tilde{\mathbf{s}}]$, $C_{[\tilde{\mathbf{s}}]}(\cdot)$ respectively, i.e.

$$\tau_{ljk}^{[t+1]} \leftarrow \underset{\tau_{ljk}^{\star} \in \mathcal{I}_{ljk}}{\arg\min}\, C_{[\tilde{\mathbf{s}}]}(U_{jk}(\beta_{jk}^{[t]}, \tau_{ljk}^{\star};\ \cdot)),$$

where $\mathcal{I}_{ljk}$ is a search interval for $\tau_{ljk}^{[t+1]}$, e.g.,

$$\mathcal{I}_{ljk} = [\tau_{ljk}^{[t]} \cdot 10^{-1}, \tau_{ljk}^{[t]} \cdot 10].$$

Scaleable Distributional Learning

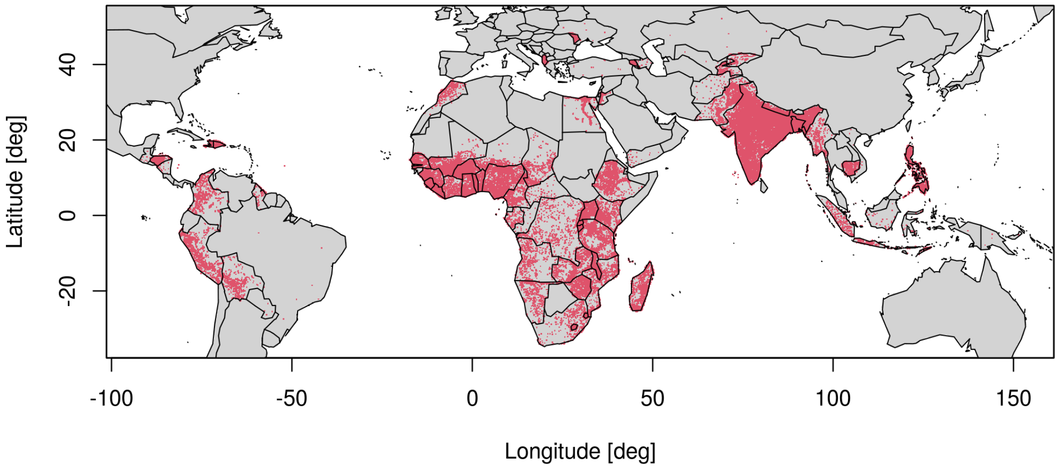
Some interesting features:

1. Set, e.g., $\nu = 0.1$, convergence after visiting m batches $[\mathbf{s}]$.
2. Only update if **"out-of-sample" log-likelihood** is **increased**.
3. **Boosting** for **variable selection**: Update only $f_{jk}(\cdot)$ with greatest contribution in "out-of-sample" log-likelihood.
4. **Bagging**: If $\nu = 1$, each update is so to say a **"sample"**. Convergence similar to MCMC algorithms, i.e., if $\beta_{jk}^{[t+1]}$ start fluctuating around a certain level.
5. **Slice sample** τ_{ljk} under $C_{[\tilde{\mathbf{s}}]}(\cdot)$, **much faster**!

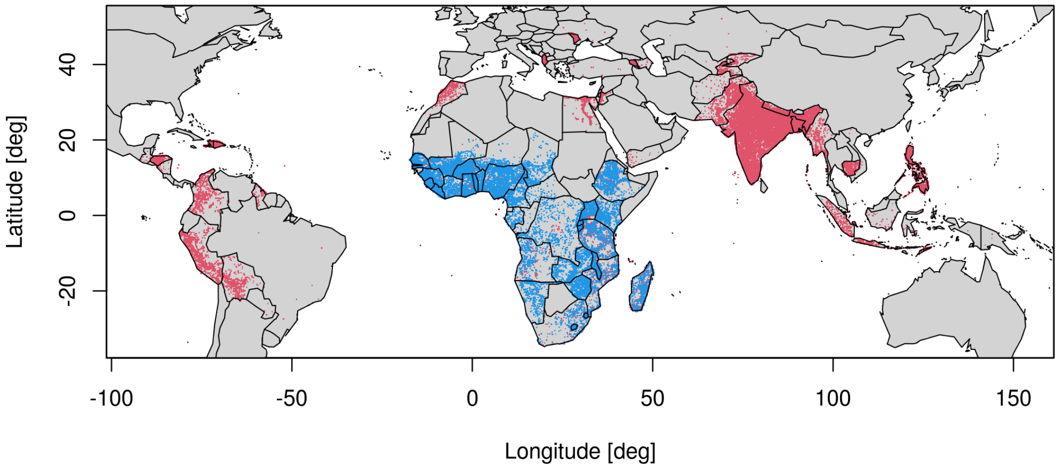
Application

- Project aiming to better explain the problems of childhood malnutrition in low- and middle-income countries.
- Contribute to monitoring of the Sustainable Development Goals (SGD).
- We compiled a brand new data set using DHS data.
- Data on global conflicts, topography and environmental data from satellite observations (NDVI), temperature and precipitation data from ERA5 (ECMWF).
- Data from 1990–2019 with $n > 3M$ observations.

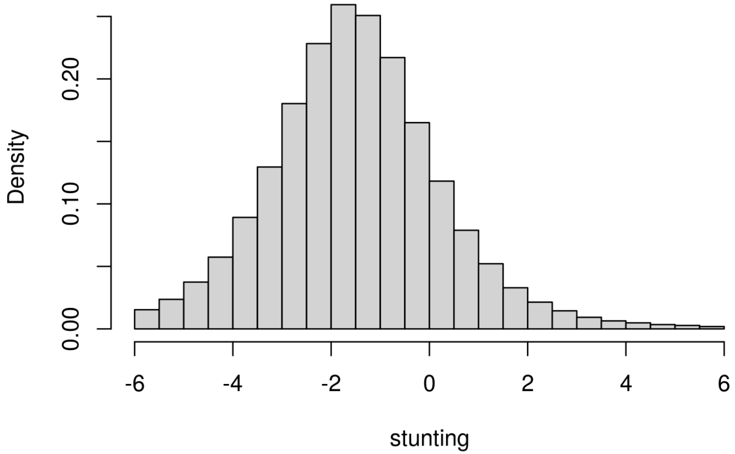
Application



Application



Application



Application

Example: Search distribution.

Define the batchsize.

```
R> bs <- 2000
```

Generate batches.

```
R> batch_ids <- lapply(1:200, function(...) {
+    sample(1:nrow(d), size = bs, replace = FALSE)
+  })
```

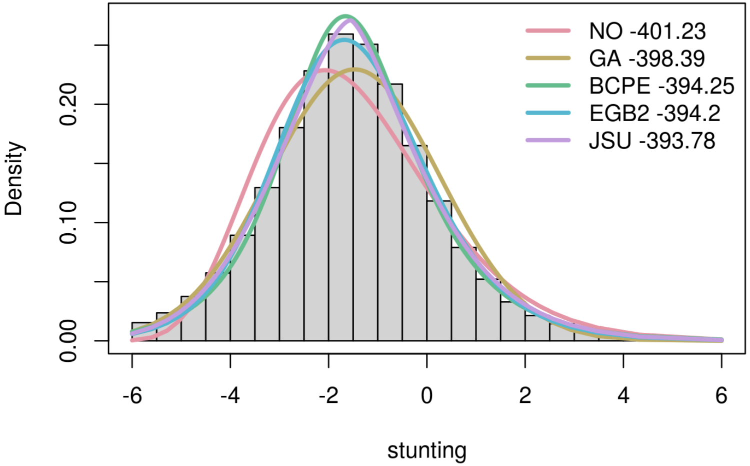
Estimate model.

```
R> b <- bamlss(y ~ 1, data = d, family = JSU,
+    sampler = FALSE, optimizer = opt_bbfitp, slice = 10, aic = TRUE, K = 2,
+    batch_ids = batch_ids)
```

Compute log-likelihood.

```
R> logLik(b, newdata = nd)
```

Application



Application

Example: Boosting flavour with *ff* data frame.

Set up a model formula.

```
R> f <- list(
+    stunting ~ s(cage) + s(bord) + s(hhs) + s(x, y) + ...,
+             ~ s(cage) + s(bord) + s(hhs) + s(x, y) + ...
+  )
```

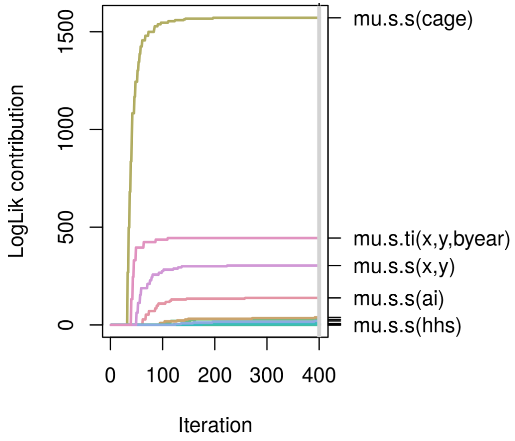
Estimate model.

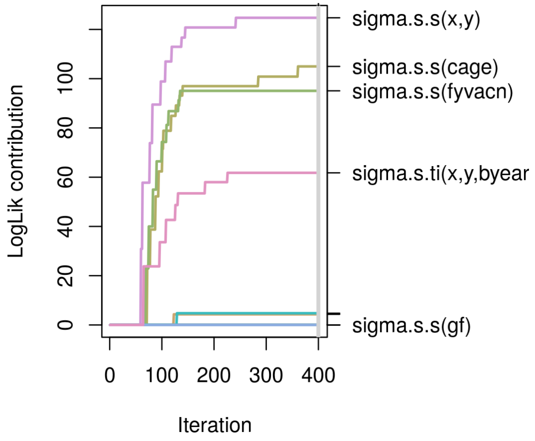
```
R> b <- bamlss(f, data = dff, family = JSU,
+    sampler = FALSE, optimizer = opt_bbfit,
+    batch_ids = batch_ids, select = TRUE, aic = TRUE, always = FALSE,
+    eps_loglik = 0.001, K = 2, overwrite = TRUE, delete = FALSE,
+    ff_name = ff_name)
```

Plot results.

```
R> contribplot(b)
```

Application





Application

Example: Bagging type flavour with slice sampling.

Extract formula.

```
R> nf <- new_formula(b)
```

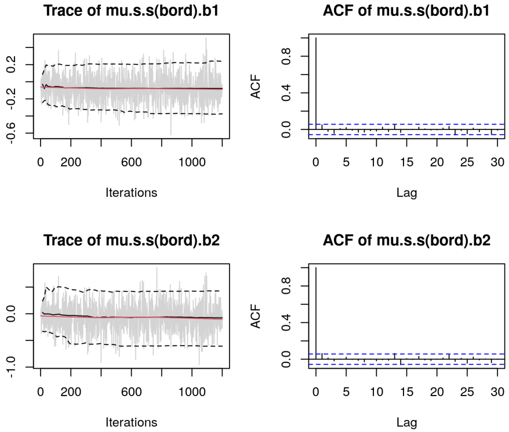
Estimate model using ff processed data.

```
R> m <- bamlss(nf, data = dff, family = JSU,
+    sampler = FALSE, optimizer = opt_bbfitp,
+    batch_ids = batch_ids, aic = TRUE, slice = TRUE,
+    ff_name = ff_name)
```

Afterwards, all extractor functions provided by *bamlss* can be used, e.g., `summary()`, `predict()`, `plot()`, etc.

Application

```
R> plot(m, which = "samples")
```


Trace of mu.s.s(bord).b1
ACF of mu.s.s(bord).b1
Trace of mu.s.s(bord).b2
ACF of mu.s.s(bord).b2
Iterations
Lag
ACF