ASP20 Boost

15 September 2020

ASP20 Boost

Term Paper
Submitted in partial fulfillment of the requirements for the master's Seminar
"Advanced Statistical Programming "
of the
Faculty the Economic Sciences
of the
Georg August University of Göttingen
2020

Johannes Stauß
Juan Sebastián Aristizábal Ortiz
Levin Wiebelt

Date of submission: 10.09.2020

Advisor: Prof. Dr. Thomas Kneib
Center for Statistics

List of Tables

List of Figures

Abstract

We describe version 1.0 of the R add-on package asp20boost. The package implements boosting *Note: Expand.*

Keywords: component-wise boosting, gradient descent, location and scale.

1. Introduction:

The R package asp20boost provides a boosting algorithm for estimation of location-scale regression models with gaussian response. It arose in the context of the course "Advanced Statistical Programming with R". It consists of a series of student projects that aim to estimate a location-scale regression model by implementing different methods each building upon an R6 class (Chang 2019) given to us as part of the `asp20model` repository (Riebl 2020).

In the following part 2 *model and methods* we first describe the nature of location-scale regression models as well as the concept of boosting. This offers context and understanding of how gradient boosting approaches location-scale regression problems. Next in part three *software implementation*, we explain how we implemented model and boosting algorithm in the `asp20boost` package and we offer a walkthrough regarding the basic usage of both the class and the function provided by the package. *Function or functions?* Furthermore in part 5 *simulation studies*, we present the results of our simulation studies and assess the performance and the limitations of our package. Finally, in section 5 *discussion*, we discuss the results and limitation of this implementations an offer outlook for future directions for further development.

2. Model and Methods

Location-Scale Regression Model

Location-scale regression models are linear models, where the expectation (location), as well as the standard deviation (scale) of the response are dependent on some regressor(s). Modeling only the relationship influencing the location while dismissing the influence upon the scale would lead to heteroskedasticity problems. Figure 1 below depicts the scatterplot of the sample model configuration of the asp20model (Riebl 2020). The heteroskedastic response y is plotted against the covariate x. In this case x is used to model both location and scale, which allows nice plotting. In a more general case scale may be modeled by a different set of predictors z.

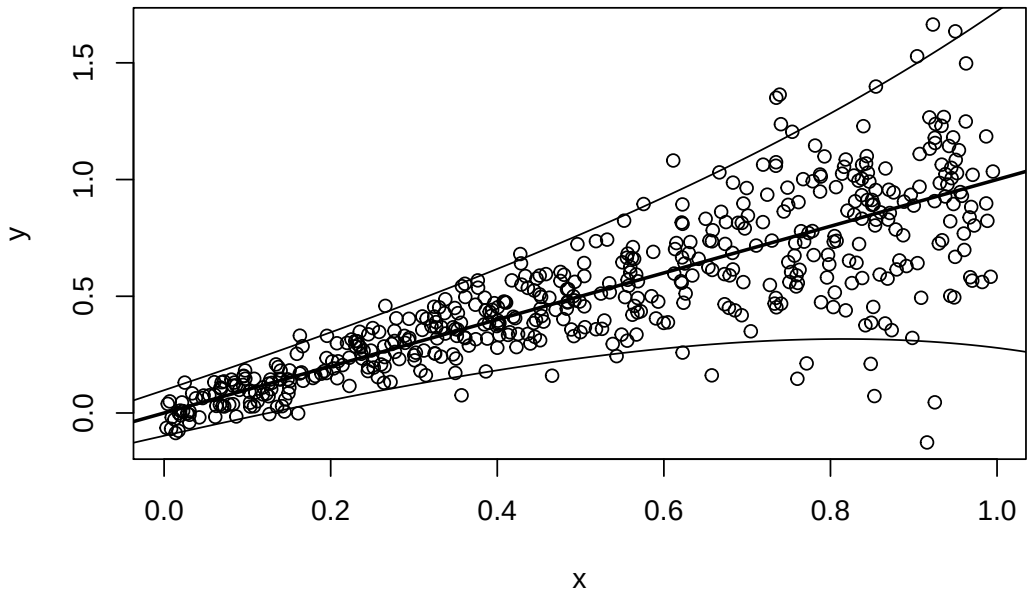


Figure 1: A visual example of Heteroskedasticity. Source: asp20model

In Figure 1 above, heteroskedasticity is clearly indicated by the funnel shape of the point cloud. Estimating such a model naively by means of OLS would lead to false variance estimates and hence invalid hypotheses tests and confidence intervals. Modeling the location part is done by a conventional linear model. The expectation of an observation y_i is modeled as a linear combination of a vector of predictors $\boldsymbol{x_i}$:

$$E(y_i) = \eta_{\mu_i} = \boldsymbol{x_i'}\boldsymbol{\beta} \tag{1}$$

Modeling the scale involves an additional concern: the modeled standard deviations $sd(y_i)$ should be non-negative. This is ensured by the response function $h(\eta) = \exp(\eta)$. Hence the standard deviations depend on transformed linear predictors:

$$sd(y_i) = \exp(\boldsymbol{z_i'}\boldsymbol{\gamma}) \tag{2}$$

This means that the unit-specific standard deviations $sd(y_i)$ are not modeled linearly. Transforming them using the link function $g(\sigma_i) = \log(\sigma_i)$, which is the inverse of the response function, allows for linearly regressing on the scale predictor

$$\log sd(y_i) = \eta_{\sigma_i} = \boldsymbol{z_i'}\boldsymbol{\gamma} \tag{3}$$

The goal of estimating a location-scale regression model is to attain estimates for $\boldsymbol{\beta}$ and $\boldsymbol{\gamma}$. Gradient boosting is able to address this heteroskedasticity yielding correct variance estimates i.e. unbiased $\boldsymbol{\gamma}$ estimates.

Boosting

We explain the concept of boosting in two steps. To get a good grasp of the idea behind boosting we first introduce what we call *residual boosting*. It is an intuitive and comprehensible algorithm, but only applicable to location estimation. Second, we explain *gradient boosting* which is a broader concept that allows for the estimation of the scale as well. Because the latter case is the problem we attempt to address, the `asp20boost` package implements exclusively gradient boosting as we describe it. Lastly, we describe *componentwise boosting*, an additional functionality of the asp20boost package. It is an extension of the boosting concept, which enables variable selection.

Residual Boosting

The general concept of boosting relies on the idea of iteratively estimating so called *base-learners* and aggregating them into an overall estimate. In residual boosting, the focus lies on the residuals. Initial parameter estimates $\hat{\boldsymbol{\beta}}$, which may be far from optimal, are needed to calculate residuals.

$$\begin{gathered}\boldsymbol{u} = \boldsymbol{y} - \boldsymbol{X}\hat{\boldsymbol{\beta}}^{(t-1)} \\ \text{residual calculation step}\end{gathered} \tag{4}$$

The effect of the location regressors on these residuals, summarized in the design matrix $\boldsymbol{X}$, is estimated using least squares resulting in the residual-estimate, or base-learner, $\hat{\boldsymbol{\beta}}$

$$\begin{gathered}\hat{\boldsymbol{b}} = (\boldsymbol{X'X})^{-1}\boldsymbol{X'u} \\ \text{residual estimation step}\end{gathered} \tag{5}$$

$\hat{\boldsymbol{b}}$ is added to the location-estimate $\hat{\boldsymbol{\beta}}$, adjusted by some learning rate ν.

$$\begin{gathered}\hat{\boldsymbol{\beta}}^{(t)} = \hat{\boldsymbol{\beta}}^{(t-1)} + \nu\hat{\boldsymbol{b}} \\ \text{parameter Updating Step}\end{gathered} \tag{6}$$

This yields an updated parameter estimate and the procedure is repeated thereafter. With each update, the fit of the model improves leading to smaller residuals and hence smaller base-learners $\hat{\boldsymbol{b}}$. This means than with each iteration the overall change in parameter $\hat{\boldsymbol{\beta}}$ decreases thus allowing the boosting algorithm to converge towards the OLS estimate which is, in fact, the true value of $\hat{\boldsymbol{\beta}}$.

Boosting seems to be rather cumbersome for estimating location parameters alone, as it yields no advantage over OLS. However, for models with varying scale, OLS is no reliable estimation method. This is where gradient boosting plays its role. It is able to address location-scale regression settings extending the idea of boosting by shifting the focus from residuals to gradients.

Gradient Boosting

Gradient boosting is no fundamentally different concept, but rather another view on the boosting algorithm. In this case the goal is to improve a certain loss function instead of estimating residuals. We will see that for location the gradients turn out to be (variance adjusted) residuals. A high degree of flexibility characterizes the choice of the loss function, as it allows even non-linearity. This enables gradient boosting to address various kinds of situations. For location-scale regression, the negative log likelihood of the observed response is convenient.

$$= \sum \log \sigma_i + \frac{1}{2} \sum \log 2\pi + \frac{1}{2} \sum \frac{y_i^2}{\sigma_i^2} + \frac{1}{2} \sum \frac{\mu_i^2}{\sigma_i^2} + \frac{1}{2} \sum \frac{-2 y_i \mu_i}{\sigma_i^2} \tag{7}$$

The unit-wise gradients of this loss function play a key role. These are different from the gradients supplied in the `asp20model package`. The relevant gradients for boosting are attained by partially deriving the loss function by the unit-specific linear predictors η_{μ_i} and by η_{σ_i} as in *insert the number of the equations* respectively:

$$\begin{aligned} \eta_{\mu_i} &= \boldsymbol{x_i'} \hat{\boldsymbol{\beta}} = E(y_i) \\ \eta_{\sigma_i} &= \boldsymbol{z_i'} \hat{\boldsymbol{\gamma}} = \log sd(y_i) \end{aligned} \tag{8}$$

Deriving the negative log likelihood of the gaussian distribution by η_{μ_i} and η_{σ_i} respectively yields the following gradients:

Here we see, as mentioned above, that location gradients are equivalent to the variance adjusted residuals. In location regression with constant variance it (*it is what?*) would be equivalent to residuals up to a multiplicative constant. *This sentence isn't clear* In this sense, gradient boosting is the broader concept, while residual boosting as described above is a special case (*of what?*). We obtain n μ gradients, as well as n σ gradients. *It is better to re-state the role specifically* The gradients play the role residuals did in the last part. They are estimated by regressing them on the model covariates x_i and z_i respectively.

$$\begin{aligned} \hat{\boldsymbol{b}} &= \boldsymbol{X}' \boldsymbol{X}^{-1} \boldsymbol{X}' \boldsymbol{u}_\mu \\ \hat{\boldsymbol{g}} &= \boldsymbol{Z}' \boldsymbol{Z}^{-1} \boldsymbol{Z}' \boldsymbol{u}_\sigma \end{aligned} \tag{9}$$

gradient estimation step

This results in two separate gradient estimates or base-learners. The base learning procedure in our case is again the least squares estimate. Another possible way of generalizing gradient boosting even further is to chose different kinds of base learning procedures, such as splines or regression trees. The gradient estimates $\hat{\boldsymbol{b}}$ and $\hat{\boldsymbol{g}}$ are used to update the overall parameter estimates $\boldsymbol{\beta}$ and γ, again adjusted by some learning rate ν.

$$\begin{aligned} \hat{\boldsymbol{\beta}}^{(t)} &= \hat{\boldsymbol{\beta}}^{(0)} + \nu\hat{\boldsymbol{b}} \\ \hat{\boldsymbol{\gamma}}^{(t)} &= \hat{\boldsymbol{\gamma}}^{(t-1)} + \nu\hat{\boldsymbol{g}} \\ &\text{parameter updating step} \end{aligned} \tag{10}$$

Updating the parameters leads to an improved fit i.e. a smaller loss function, which in turn leads to changed gradients. The next iteration may be performed *(for what/to achieve what / until what is achieved?)*. Parameter estimates in the gradient boosting algorithm also converge towards the true parameter values of the location-scale regression model.

Componentwise boosting

Moreover, asp20boost implements componentwise boosting, which is an extension of the boosting algorithm. It provides variable selection for designs where many explaining variables are available, including settings where the number of regressors p exceed the number of observations n. The idea is to fit multiple base learners in each step and choosing the one that improves the loss function the most. Again, three steps are performed in each iteration:

```
i. Gradient calculation
ii. Gradient estimation
iii. Parameter updating
```

Gradient calculation works equivalently as described above. Gradient estimation, in contrast, demands further calculations. The gradients are regressed on each covariate or component respectively, resulting in p gradient estimators.

$$\begin{aligned} \hat{b_j} &= (\boldsymbol{x}'\boldsymbol{x})^{-1}\boldsymbol{x}'\boldsymbol{u}_\mu \\ \hat{g_j} &= (\boldsymbol{x}'\boldsymbol{x})^{-1}\boldsymbol{x}'\boldsymbol{u}_\sigma \end{aligned} \tag{11}$$

*Is it really called minimum SSC?*Then, the best loss improvement j^* is determined by a minimum sum of squares criterion:

$$\begin{aligned} j^* &= \text{argmin} \sum (\boldsymbol{u}_\mu - x_{ij}\hat{\boldsymbol{b}}_j)^2 \\ k^* &= \text{argmin} \sum (\boldsymbol{u}_\sigma - z_{ij}\hat{\boldsymbol{g}}_j)^2 \end{aligned} \tag{12}$$

The parameter vectors are then updated as well in a componentwise manner. This means that only one component of each parameter vector $\boldsymbol{\beta}$ and $\boldsymbol{\gamma}$ changes, while $p-1$ components remain unaffected by the updating step:

$$\begin{aligned} \hat{\boldsymbol{\beta}}_{j^*}^{(t)} &= \hat{\boldsymbol{\beta}}_{j^*}^{(t-1)} + \nu\hat{b}_{j^*} \\ \hat{\boldsymbol{\beta}}_{j}^{(t)} &= \hat{\boldsymbol{\beta}}_{j}^{(t-1)},\ j \neq j^* \\ \hat{\boldsymbol{\gamma}}_{j^*}^{(t)} &= \hat{\boldsymbol{\gamma}}_{j^*}^{(t-1)} + \nu\hat{b}_{j^*} \\ \hat{\boldsymbol{\gamma}}_{j}^{(t)} &= \hat{\boldsymbol{\gamma}}_{j}^{(t-1)},\ j \neq j^* \end{aligned} \tag{13}$$

With this update restriction some components may become updated quite late in the progression of the algorithm. This is the case when the particular component does not yield high improvements of model fit, which in turn reflects that this component, or the predictive variable, has a rather small effect on the response. The algorithm is not executed until convergence

of all parameter components, but stopped after a predefined number of iterations `maxit`. Hence it could be the case, that covariates with small effect sizes do not become updated at all and remain zero after the componentwise gradient-boost algorithm runs completely. If this is the case, such variables will be dropped out of the model. In this way i.e. by having parameter update restrictions combined with early stopping implicit variable selection is achieved.

3. Software Implementation: the `asp20boost` package

The code of the `asp20boost` package consists of two major parts. The first is the R6 class `LocationScaleRegressionBoost`, which represents the location-scale regression model and includes some quantities (e.g. gradients) necessary for executing the boosting algorithm. The second part is `gradient_boost()`, a function specifically conceived to execute the gradient boost algorithm upon a `LocationScaleRegressionBoost` object.

It is important to emphasize that this package allows for the practical application of the concepts described in the „gradient boosting“and „componentwise boosting“ sections presented above. The concept of”residual boosting“ fulfilled an explanatory purpose and is not implemented in the package.

`gradient_boost()` function

It is easier to comprehend our code by considering first the `gradient_boost()` function. It uses the quantities of the model class in order to perform the boosting algorithm. The principal argument is the `model LocationScaleRegressionBoost` object followed by several optional specifiers. In each iteration of the algorithm, the function directly applies the third step of parameter updating by calling the public method update_parameters_*() of the model class. Steps one and two, i.e. gradient calculation and estimation, are performed by the model class object and explained in the next subsection.

This function performs a prespecified number of 1000 iterations unless another quantity was specified by means of the `maxit` argument in the `gradient_boost()` function. After iterating, a modified model object is returned instead of mere parameter values. This includes, for example, the model formula, the log likelihood value and the desired parameter configuration after the execution of the algorithm. In each iteration $\boldsymbol{\beta}$ gets updated first followed by $\boldsymbol{\gamma}$. If `componentwise` is set to `TRUE`, the algorithm employs the public method `update_parameters_compwise()`, which performs additional calculations that `update_parameters_conventional()` does not. The functioning of these methods is explained in the following subsection where the `LocationScaleRegressionBoost` class is described.

`LocationScaleRegressionBoost` R6-Class

The class extends the `LocationScaleRegression` class of the `asp20model` package (Riebl 2020). As already mentioned, this is the case due to the fact that `asp20boost` arose as a series of student projects aiming to solve a common problem building an implementation upon the `asp20model` package, specifically its `LocationScaleRegression` class.

We accomplish this by adding a total of eight additional methods:

- `gradients_loglik_mu`
- `gradients_loglik_sigma`
- `gradient_estimators_mu`
- `gradient_estimators_sigma`
- `gradient_estimators_mu_compwise`
- `gradient_estimators_sigma_compwise`
- `update_parameters_conventional`
- `update_parameters_compwise`

Two of the public methods are easy to comprehend: `gradients_loglik_mu()` and `gradients_loglik_sigma()` perform simply the first gradient-calculation step of each iteration by employing the gradient formulas derived theoretically in the last part.

LEVIN: Confess our mistakes!

These two functions use residuals obtained from the `resid()` method and the estimates of the standard deviation obtained from the public field `fitted_scale`. Both of these fields are inherited from the parental class `LocationScaleRegression`. As the calculated gradients may be possibly of interest, we allowed for external access by setting `gradients_loglik_mu()` and `gradients_loglik_sigma()`as a public methods.

The four gradient estimator functions `gradient_estimators_*()`, in contrast, are private and can not be accessed via class-external functions, such as `gradient_boost()`. However, they are used in the calculations performed by two public methods `update_parameters_*()`.

LEVIN: update!

The private functions are called `gradient_estimators_*()` and perform the second step of gradient estimation. The functions with the suffix *conventional* simply regress the location gradients on location covariates and the scale gradients on scale covariates respectively. The functions suffixed with *componentwise* return a list of gradient estimators obtained by regressing the gradients on one particular component $\boldsymbol{x_j}$ or $\boldsymbol{z_k}$ of the respective design matrix $\boldsymbol{X}$ or $\boldsymbol{Z}$.

The centerpiece of the boosting implementation are the update methods

`update_parameters_conventional`and `update_parameters_compwise`.

The *conventional* version merely adds the gradients estimators to the active fields $\boldsymbol{\beta}$ and $\boldsymbol{\gamma}$ multiplied by the defined stepsize for $\boldsymbol{\beta}$, and for $\boldsymbol{\gamma}$. The `stepsize` parameter is set by default to 0.01 for $\boldsymbol{\beta}$, and 0.1 for $\boldsymbol{\gamma}$. Additionally to this, the componentwise version further performs first the loss improvement calculation and then builds an update vector consisting of zeros with exception of the best fitting component. Finally, `update_parameters_compwise` performs the update. In both of its versions, the update method finally assigns new values to $\boldsymbol{\beta}$ and $\boldsymbol{\gamma}$.

These active fields are inherited from the `LocationScaleRegression` class of the `asp20model` package. Changing them induces a new model state of the `LocationScaleRegressionBoost` object. Hence, various fields become new values assigned. Among them are the fields `.resid`, `fitted_location` and `fitted_scale`, which are used to calculate residuals and finally gradients.

This was added by Johannes. Does it make sense Levin? As a consequence, the public methods `gradients_loglik_mu` and `gradient_estimators_mu` of our `LocationScaleRegressionBoost` class also yield different values if called *Begining from here this sentence does not make sense*as well as their counterparts for $\boldsymbol{\sigma}$ and componentwise estimation. This enables proceeding to the next iteration.

User Interface by example

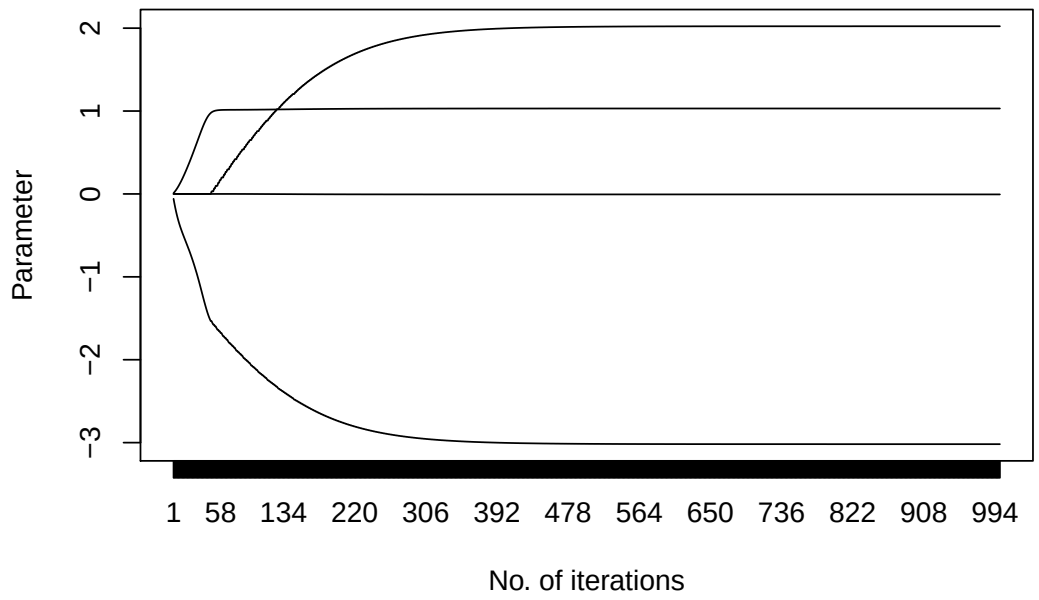
First, we define the model by generating sample data and defining explanatory variables for the standard deviation of the response variable. After the definition *what definition?*

`gradient_boost` will iterate over the model and update the parameters at each step, depending on the selected method. The function `update_parameter_componentwise` will instruct the model to use componentwise boosting if set to `TRUE`. Depending on the sample data, a custom step size for β and γ might be useful, which can be set by means of the `stepsize` parameter imputing a vector *what dimensions should this vector have?* containing the desired step size for β and γ respectively.

```
set.seed(12345)
n <- 500
x <- runif(n)
y <- x + rnorm(n, sd = exp(-3 + 2 * x))
model <- LocationScaleRegressionBoost$new(y ~ x, ~ x)
gradient_boost(model,
               stepsize = c(0.01, 0.1),
               maxit = 1000,
               componentwise = TRUE)
```

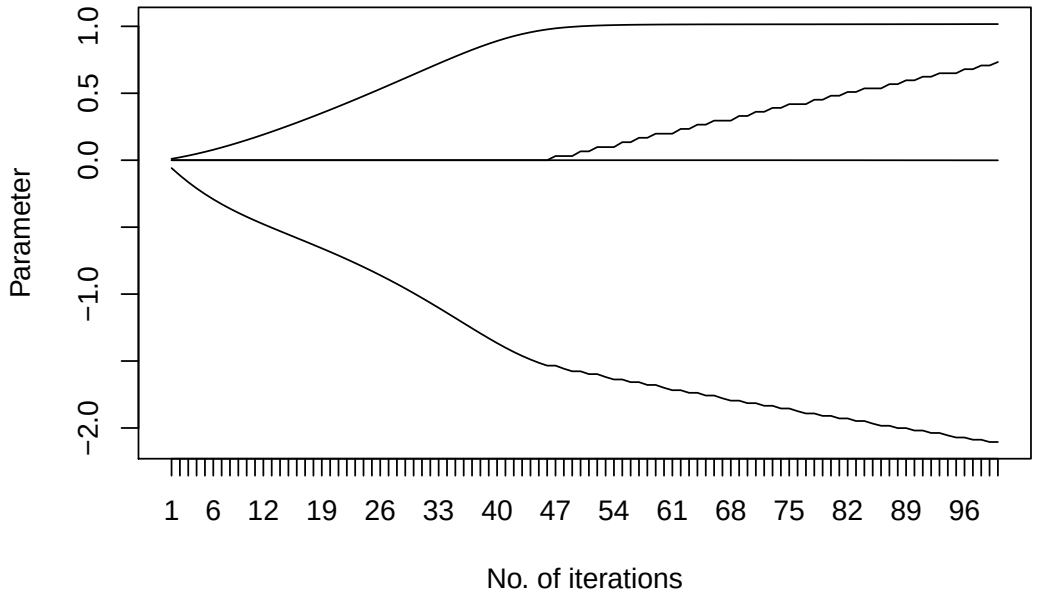
To evaluate the parameter changes throughout the iterations, plotting them can be enabled by setting `plot = TRUE`. See #1.

```
model <- LocationScaleRegressionBoost$new(y ~ x, ~ x)
gradient_boost(model,
               stepsize = c(0.01, 0.1),
               componentwise = TRUE, plot = TRUE) #1.
```



The number of iterations can be limited by specifying the desired value in the `maxit` parameter. For example, for 100 iterations (see #2):

```
model <- LocationScaleRegressionBoost$new(y ~ x, ~ x)
gradient_boost(model,
               stepsize = c(0.01, 0.1),
               componentwise = TRUE,
               maxit = 100, # 2.
               plot = TRUE)
```



Parameter vectors of $\boldsymbol{\beta}$ and $\boldsymbol{\gamma}$ can be obtained by accessing the their corresponding fields via `model$beta` and `model$gamma`as follows (see "#3").

```
model <- LocationScaleRegressionBoost$new(y ~ x, ~ x)
message("Initial Beta parameters:")
```

```
## Initial Beta parameters:
```

```
model$beta # 3.
```

```
## [1] 0 0
```

```
message("Initial Gamma parameters:")
```

```
## Initial Gamma parameters:
```

```
model$gamma # 3.
```

```
## [1] 0 0
```

```
gradient_boost(model,
               stepsize = c(0.01, 0.1),
               maxit = 1000,
               componentwise = TRUE, plot = FALSE)
message("Beta parameters:")
```

```
## Beta parameters:
```

```
model$beta # 3.
```

```
## [1] -0.005810434  1.030810459
```

```
message("Gamma parameters:")
```

```
## Gamma parameters:
```

```
model$gamma # 3.
```

```
## [1] -3.018521  2.022926
```

The verbose mode for further details on each iteration can be enabled by setting `verbose = TRUE`. See "#4".

```
model <- LocationScaleRegressionBoost$new(y ~ x, ~ x)
gradient_boost(model, maxit = 3, verbose = TRUE) # 4.
```

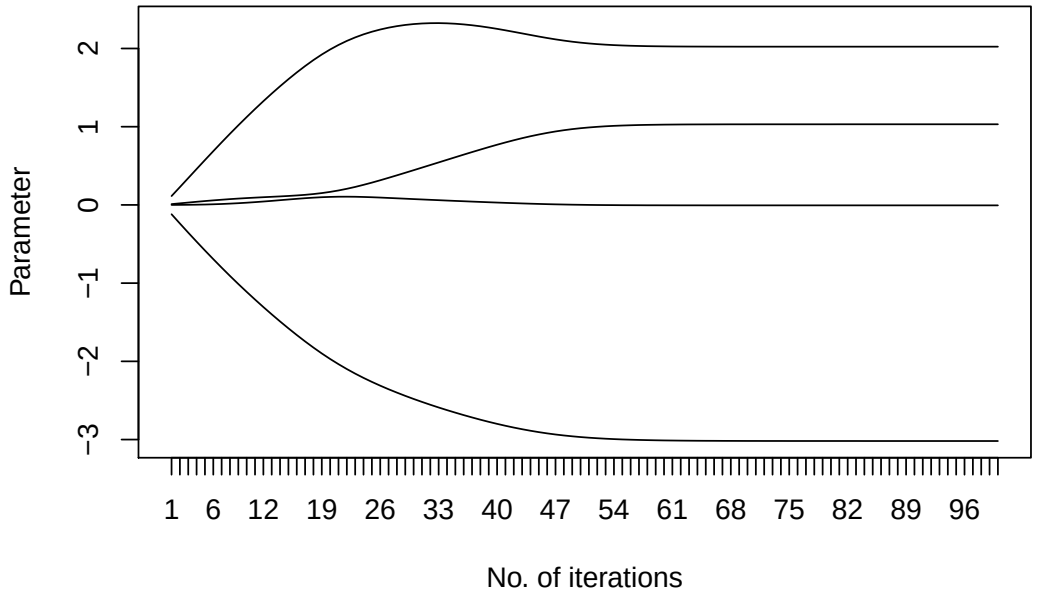
```
## Iteration:      1
## Parameters:     5.38e-06 1.01e-02 -1.19e-01 1.13e-01
## Squared Resid:  201.924794863565
## Log-likelihood: -537.653
## ===============

## Iteration:      2
## Parameters:     0.000442 0.020247 -0.237741 0.227707
## Squared Resid:  198.023884557381
## Log-likelihood: -513.8
## ===============

## Iteration:      3
## Parameters:     0.00139 0.03021 -0.35455 0.34310
## Squared Resid:  193.951674532815
## Log-likelihood: -490.702
## ===============
```

To execute the conventional boosting algorithm on the model, the user sets the `componentwise = FALSE`, which will result in an update of all parameters at once. See #5.

This was previously described in section 2.2.2.

Didn't we state above that we didn't implement conventional boosting but just explained it for the sake of building an argument?

```
model <- LocationScaleRegressionBoost$new(y ~ x, ~ x)
gradient_boost(model,
               stepsize = c(0.01, 0.1),
               componentwise = FALSE, # 5
               maxit  = 100, plot = TRUE)
```



4. Simulation Studies

In this part we design and perform a simulation study in order to demonstrate functionality of the implemented boosting algorithm. For this, we first employ models with two location and scale parameters each, and show basic functionality. Second, validity of the variable selection features is demonstrated by using location and scale parameters with up to 50 components. Lastly, we compare our algorithm with the one implemented in the `gamboostLSS` package. *really?*

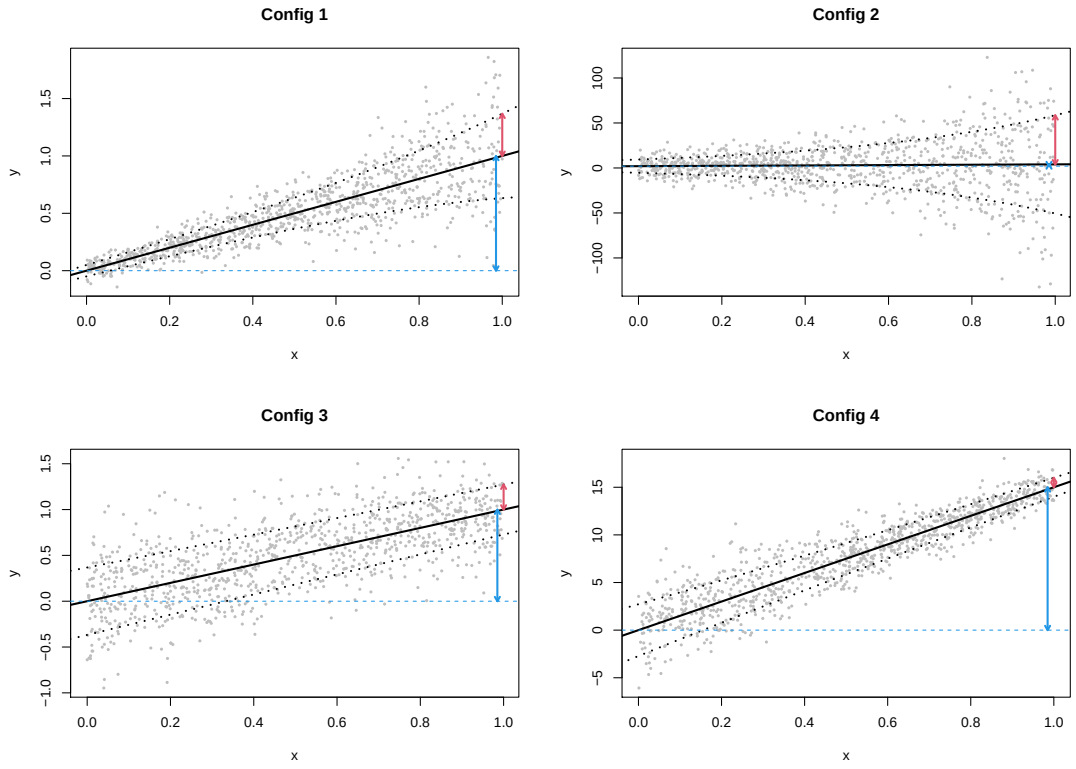
```
rm(list=ls())
options(scipen=999)
source("../simulation_studies/helper_plot_model.R")
source("../simulation_studies/helper_init_model.R")
source("../simulation_studies/helper_calc_mse.R")
source("../simulation_studies/helper_init_large_model.R")
library(asp20boost)
library(gamboostLSS)
library(microbenchmark)
```

Two-dimesional Case

The goal in this first part is to demonstrate that the algorithm finds correct estimates for sensible choices of $\boldsymbol{\beta}$ and $\boldsymbol{\gamma}$. We use the following 4 different parameter configurations to explore location-scale regression model characteristics.

```
set.seed(1337)
mod_1 <- init_model(beta_1, gamma_1, 1000)$model
gradient_boost(mod_1)
mod_2 <- init_model(beta_2, gamma_2, 1000)$model
gradient_boost(mod_2)
mod_3 <- init_model(beta_3, gamma_3, 1000)$model
gradient_boost(mod_3)
mod_4 <- init_model(beta_4, gamma_4, 1000)$model
gradient_boost(mod_4)
```

```
par(mfrow = c(2,2))
plot_model(beta_1, gamma_1, 1000, "Config 1")
plot_model(beta_2, gamma_2, 1000, "Config 2")
plot_model(beta_3, gamma_3, 1000, "Config 3")
plot_model(beta_4, gamma_4, 1000, "Config 4")
```




```
##   Config beta_true beta.est. gamma_true gamma.est.
## 1      1         0    -0.006        -3.0     -2.951
## 2      1         1     1.019         2.0      1.944
## 3      2         2       NaN         2.0        NaN
## 4      2         2       NaN         2.0        NaN
## 5      3         0    -0.020        -1.0     -0.947
## 6      3         1     1.037        -0.3     -0.475
## 7      4         0     1.270         1.0      0.838
## 8      4        15    -0.245        -1.0      1.944
```

Evaluation Configuration 1

Values: $\boldsymbol{\beta} = (0, 1)'$ and $\boldsymbol{\gamma} = (-3, 2)'$

This parameter configuration is equivalent to the base example case of the `asp20model` package. As already mentioned, heteroskedasticity is present, as indicated visually by the funnel shape of the scatter plot. The solid black line represents the true response mean, while the dashed lines represent the true upwards and downwards standard deviations from this mean. The red arrow indicates the standard deviation at the point $x = 1$ i.e. the upper limit of covariate values, since covariates are designed to be on the unit interval. For cases with increasing scale, the red arrow hence indicates maximum standard deviation of the response. The blue arrow depicts the slope of the location parameter vector $\boldsymbol{\beta_1}$. The ratio of those two quantities is essential for the estimability of the model. Here, the blue arrow is dominant indicating that the model may be well estimated. Estimates are displayed in the table above. We see that both $\boldsymbol{\beta hat}$ and $\hat{\boldsymbol{\gamma}}$ estimates are close to the true parameter vectors $\boldsymbol{\beta}$ and $\boldsymbol{\gamma}$.

Evaluation Configuration 2

Values: $\boldsymbol{\beta} = (2, 2)'$ and $\boldsymbol{\gamma} = (2, 2)'$

Configuration 2 is a somewhat naive parameter configuration where all parameters are set to value 2. Estimating this model does not yield valid results. The `gradient_boost()` function runs into numerical problems. To understand why this happens, we need to understand the model characteristics resulting from parameter configuration `2-2-2-2`. Here, the red arrow shows a large size, while the blue arrow is hardly visible. This happens because of the huge standard deviations present in response y that concealing the changes in the mean $\bar{y}$. In fact, the standard deviation at $x = 1$ is $exp(2 + 2) = 54.60$. In contrast, the true response mean values range from 0 to 2. The change in response location (the signal), indicated by the blue arrow, perishes *stirbt?* due to the high amount of noise caused by the high $\boldsymbol{\gamma}$ parameters. Thus, a signal-noise ratio results that is impossible to estimate. Values for parameters $\boldsymbol{\gamma}$ need to be chosen sensibly for simulating data. Nevertheless, this is no straight-forward task, since the effect of $\boldsymbol{\gamma}$ on the response standard deviation is not linear but exponential. To check this *check the exponentiality?* in the two-dimensional case, the arrows in the model plot may be of help.

Evaluation Configuration 3

Values: $\boldsymbol{\beta} = (0, 1)'$ and $\boldsymbol{\gamma} = (-1, -0.3)'$

Configuration 3, in turn, yields reasonable estimates. It contains a negative $\boldsymbol{\gamma}$ slope leading to decreasing response standard deviations. This can be observed by the funnel shape of the plot pointing in the opposite direction as before.

It may be of interest to check if estimates lie closer to the true parameters relative to e.g. configuration 1. To do this, we measure the gap between true values and estimates by the mean squared error (MSE). This enables comparison of configurations 1 and 3.

```
mse_1 <- calc_MSE(beta_1, gamma_1, mod_1$beta, mod_1$gamma)
mse_3 <- calc_MSE(beta_3, gamma_3, mod_3$beta, mod_3$gamma)
cbind("config 1" = mse_1,
      "config_3" = mse_3)
```

```
##       config 1     config_3
## beta  0.0003999613 0.001758186
## gamma 0.005490317  0.03329887
## total 0.005890278  0.03505706
```

Configuration 1 leads to smaller MSE *Don't we speak here of config 3? This conclusion is too lonely here.* This indicates a better performance in this configuration.

Evaluation Configuration 4

Values: $\boldsymbol{\beta} = (0, 15)'$ and $\boldsymbol{\gamma} = (1, -1)'$

In configuration 4, we again included both a negative slope for $\boldsymbol{\gamma}$ and a very high slope $\boldsymbol{\beta}$. The idea is to ensure a very good signal-noise ratio to guarantee a good model fit measured by a low MSE. Contrary to our expectations, the algorithm performs poorly for this parameter configuration.

```r
mse_4 <- calc_MSE(beta_4, gamma_4, mod_4$beta, mod_4$gamma)
cbind("config 1" = mse_1,
      "config_3" = mse_3,
      "config 4" = mse_4)
```

```
##       config 1     config_3    config 4
## beta  0.0003999613 0.001758186 234.0104
## gamma 0.005490317  0.03329887  8.694547
## total 0.005890278  0.03505706  242.7049
```

The bad model fit is due to the fact that the location slope is highly underestimated.

Conclusions

The observed two-dimensional cases let us conclude that not every arbitrary parameter configuration is estimable by the implemented `gradient_boost()` algorithm. A first insight is that signal-noise ratio needs to be sufficiently high in order to prevent that the location effect gets concealed by inflated standard deviations. Regarding the finding of configuration 4, we offer several explanations:

- Further considerations to simulating data sensibly need to be taken, which we were not yet able to work out.
- Our algorithm does not reliably yield robust results if negative γ slopes are included.
- Our algorithm runs into problems if the signal is too strong compared to noise.

Large Model - Variable Selection in the multidimensional case

In this part we demonstrate how the `gradient_boost()` function performs variable selection. We have a look at three large model configurations. Configuration 5 present 6-dimensional parameter setting. In configuration 6 β and γ are of length 11, while in configuration 7 they contain 51 components. The parameter vectors are created randomly. Then some components are set to 0. This results in the configurations displayed below. In order to ensure a good signal-noise ratio, to creation of random parameter is restricted, as explained below.

```
##      beta_true beta_est gamma_true gamma_est
## [1,]         8    8.821       -0.2     0.404
## [2,]         0    0.000        0.0     0.000
## [3,]         0    0.000        0.6     0.066
## [4,]         7    6.835        0.0     0.668
## [5,]         0    0.000        1.0     0.045
## [6,]         7    5.616        0.0     0.110
```

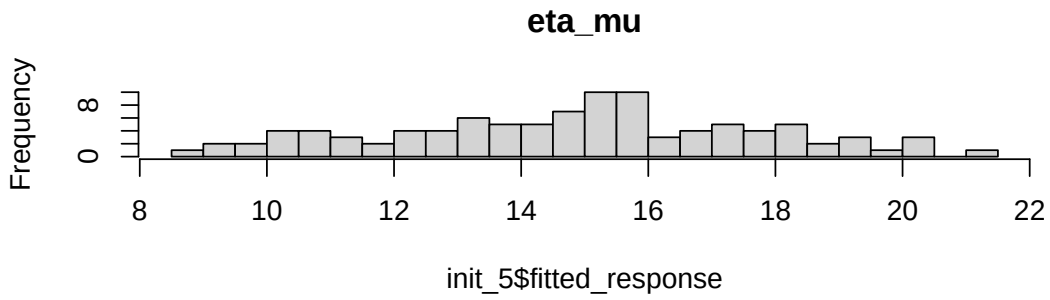
```
##       beta_true beta_est gamma_true gamma_est
##  [1,]         5    8.449       -0.2     1.462
##  [2,]         0    0.000        0.0    -0.124
##  [3,]         0    0.000        0.7     0.000
##  [4,]         3    2.415        0.0    -0.206
##  [5,]         0    0.000        0.3     0.000
```

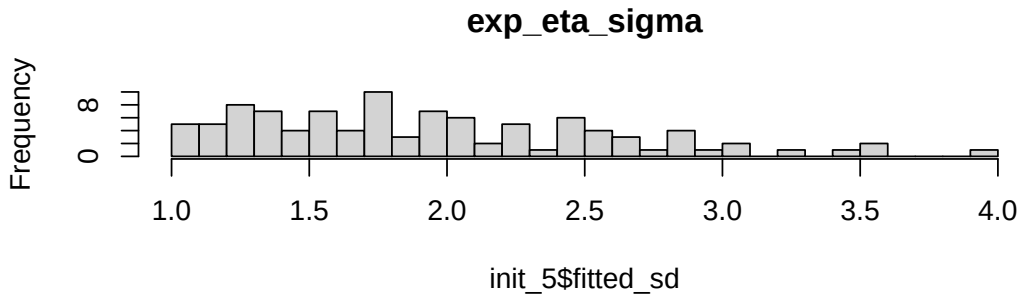
```
##  [6,]         4    0.000         0.0     0.045
##  [7,]         3    0.174         0.8     0.414
##  [8,]         0    0.000         0.0     0.000
##  [9,]         0    0.000         0.4     0.000
## [10,]         6    0.868         0.0     0.396
## [11,]         0    0.000         0.7     0.000
```

Configuration 5

As mentioned above, there are restrictions on parameter creation since not every possible parameter configuration leads to estimable models, as we have seen in the two-dimensional case. In the multidimensional case, the following problem remains: if $\boldsymbol{\gamma}$ parameters add up to high values, the resulting simulated response has huge standard deviation. This leads to the concealing of the location effects. Checking for well balanced signal-noise ratio remains to be done in the multidimensional case as well. Furthermore, plotting is hardly an option, as dimensions increase. Therefore we check using another method i.e. we look at the distributions of true response means and true response standard deviations.

```
par(mfrow=c(2,1))
hist(init_5$fitted_response, breaks = 25, main = "eta_mu")
hist(init_5$fitted_sd, breaks = 25, main = "exp_eta_sigma")
```





Location predictors take values between 8 and 22 with a maximum of 15. In contrast, the highest response- standard deviation is 4, while most values are at about 2.5 and lower. The resulting response variation should be both small enough to not conceal the location effects and big enough to be estimated.

In configuration 5, the algorithm correctly depicts which effects exists on the location response *location response oder response location?* i.e. it finds the non-zero elements of $\boldsymbol{\beta}$. For the scale

estimate $\gamma_h at$ too many parameters are chosen since two zero elements of the true parameter γ are assigned non-zero values by the estimate $\hat{\gamma}$. The estimated effect sizes differ considerably higher from the true values relative the two-dimensional case already presented. This results in a higher MSE.

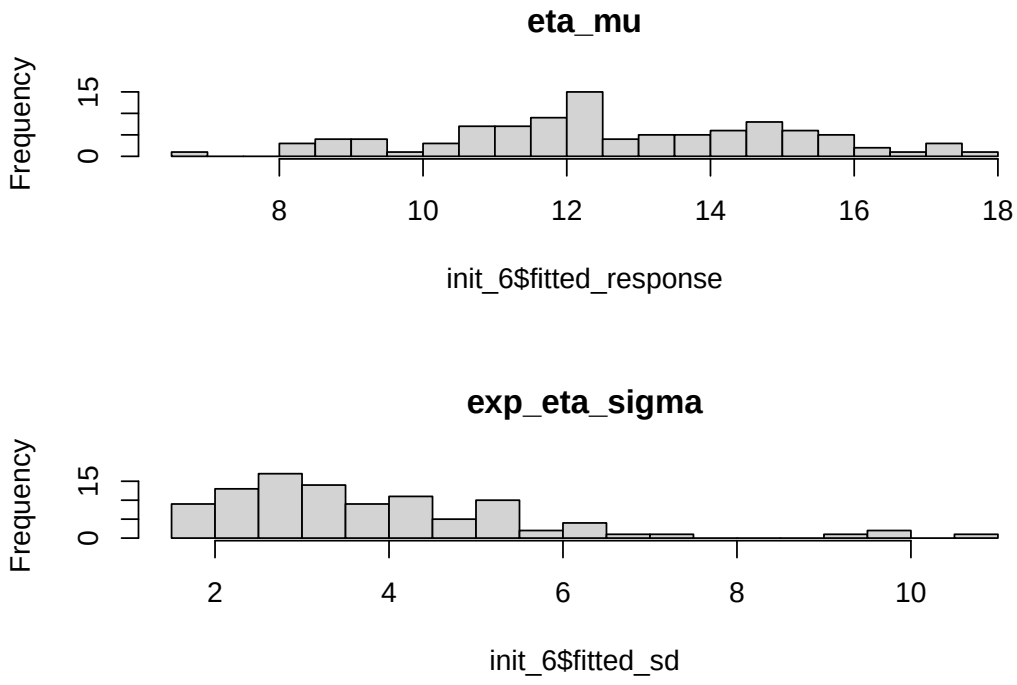
```
mse_5 <- calc_MSE(init_5$beta, init_5$gamma, mod_5$beta, mod_5$gamma)
cbind("config 1" = mse_1,
      "config_3" = mse_3,
      "config 4" = mse_4,
      "config_5" = mse_5)
```

```
##       config 1     config_3    config 4 config_5
## beta  0.0003999613 0.001758186 234.0104 0.6741996
## gamma 0.005490317  0.03329887  8.694547 0.3642567
## total 0.005890278  0.03505706  242.7049 1.038456
```

Configuration 6

In this case, we face a 10-dimensional configuration. We get the following distributions for the mean and the standard deviation respectively.

```
par(mfrow=c(2,1))
hist(init_6$fitted_response, breaks = 25, main = "eta_mu")
hist(init_6$fitted_sd, breaks = 25, main = "exp_eta_sigma")
```



For β the algorithm chose 1 *what?* too few. For γ entries 1-3 are chosen correctly. Then the algorithm yields errors. Entry 4 is chosen, while in fact being zero. Entry 5 is not chosen even when an effect in the true model exists.

Configuration 7

When scaling the parameter dimension up to 51 variables, the algorithm runs into a critical problem: all slope estimates are zero.

```
round(mod_7$beta, 3)
```

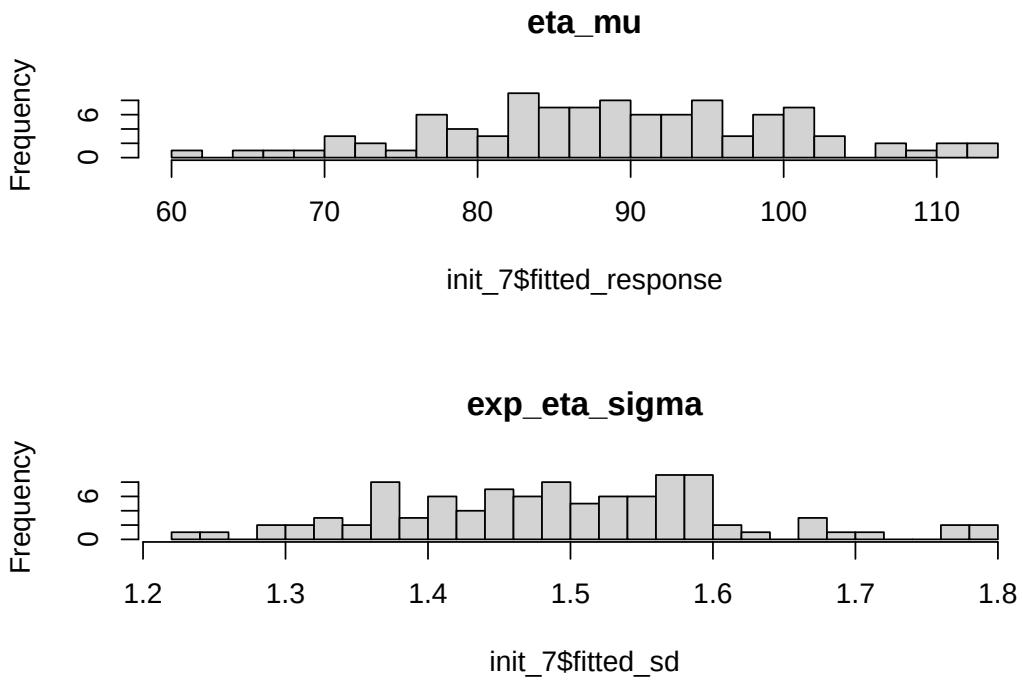
```
##  [1] 8.911 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
## [13] 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
## [25] 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
## [37] 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
## [49] 0.000 0.000 0.000
```

```
round(mod_7$gamma, 3)
```

```
##  [1] 70.581  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000
## [11]  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000
## [21]  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000
## [31]  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000
## [41]  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000  0.000
## [51]  0.000
```

Histograms indicate a well-balanced signal-noise ratio. The problem's cause lies thus somewhere else.

```
par(mfrow=c(2,1))
hist(init_7$fitted_response, breaks = 25, main = "eta_mu")
hist(init_7$fitted_sd, breaks = 25, main = "exp_eta_sigma")
```



One explanation for this problem is that the algorithm fits only intercepts in the first step resulting in very high values. The remaining iterations are used to lower intercepts *Um den intercepts to reducing? oder was meints du?*. This may be checked by setting `verbose = TRUE`. For this reason, no other components of the parameter estimates are updated. Possibly, a sensible choice of initial parameter values may address this problem. # 5. Discussion.

6. References

Chang, Winston. 2019. *R6: Encapsulated Classes with Reference Semantics*. https://CRAN.R-project.org/package=R6.

Riebl, Hannes. 2020. *Asp20model: An R6 Class for Location-Scale Regression Models*.

7. Appendix

$$
\begin{aligned}
\text{loss}(\boldsymbol{y}) &= -LL(\boldsymbol{y}) \\
&= -\log\left(\prod_{i=1}^{n} f\left(y_i|\boldsymbol{x_i}\right)\right) \\
\text{with } f\left(y_i|\boldsymbol{x_i}\right) &= \frac{1}{\sigma_i\sqrt{2\pi}}\exp\left\{-\frac{1}{2}\left(\frac{y_i-\mu_i}{\sigma_i}\right)^2\right\} \\
&= -\sum \log f\left(y_i|\boldsymbol{x_i}\right) \\
&= -\sum\left(\log\frac{1}{\sigma_i\sqrt{2\pi}} + \log\exp\left\{-\frac{1}{2}\left(\frac{y_i-\mu_i}{\sigma_i}\right)^2\right\}\right)
\end{aligned}
\tag{14}
$$

This is the loss function:

$$
= \sum \log\sigma_i + \frac{1}{2}\sum \log 2\pi + \frac{1}{2}\sum\frac{y_i^2}{\sigma_i^2} + \frac{1}{2}\sum\frac{\mu_i^2}{\sigma_i^2} + \frac{1}{2}\sum\frac{-2y_i\mu_i}{\sigma_i^2} \tag{15}
$$

For deriving the loss function we transform it to the following

$$
\sum \eta_{\sigma_i} + \frac{n}{2}\log 2\pi + \frac{1}{2}\sum y_i^2\exp(\eta_{\sigma_i})^{-2} + \frac{1}{2}\sum \eta_{\mu_i}^2\exp(\eta_{\sigma_i})^{-2} + \frac{1}{2}\sum -2\times y_i\times\eta_{\mu_i}\times\exp(\eta_{\sigma_i})^{-2} \tag{16}
$$

Then, we derive $\text{loss}(\boldsymbol{y})$ towards η_{μ_i}

$$
\begin{aligned}
\frac{\partial\text{loss}(\boldsymbol{y})}{\partial\eta_{\mu_i}} &= 0+0+0+\frac{1}{2}\exp(\eta_{\sigma_i})^{-2}\times 2\eta_{\mu_i} + \frac{1}{2}(-2)y_i\exp(\eta_{\sigma_i})^{-2}\times 1 \\
&= \mu_i\times\frac{1}{\sigma_i^2} - y_i\times\frac{1}{\sigma_i^2} = -\frac{y_i-\mu_i}{\sigma_i^2} = -\frac{\epsilon_i}{\sigma_i^2}
\end{aligned}
\tag{17}
$$

And then the derivative of the $\text{loss}(\boldsymbol{y})$ towards η_{σ_i}

$$
\frac{\partial\exp(\eta_{\sigma_i})^{-2}}{\partial\eta_{\sigma_i}} = -2\exp(\eta_{\sigma_i})^{-3}\times\exp(\eta_{\sigma_i}) = -2\exp\eta_{\sigma_i}^{-2} \tag{18}
$$

$$
\begin{aligned}
\frac{\partial\text{loss}(\boldsymbol{y})}{\partial\eta_{\sigma_i}} &= 1+0+\frac{1}{2}y_i^2(-2)\exp(\eta_{\sigma_i})^{-2} + \frac{1}{2}\eta_{\mu_i}^2(-2) + \frac{1}{2}(-2)y_i\eta_{\mu_i}(-2)\exp(\eta_{\sigma_i})^{-2} \\
&= 1 - \frac{y_i^2}{\sigma_i^2} - \frac{\mu_i^2}{\sigma_i^2} + \frac{2y_i\mu_i}{\sigma_i^2} \\
&= -\frac{(y_i-\mu_i)^2}{\sigma_i^1} + 1 \\
&= -\frac{\epsilon_i^2}{\sigma_i^2} + 1
\end{aligned}
\tag{19}
$$