Introduction to gradient boosting for extremes

Clement Dombry and Jasper Velthoen

02/14/2020

Extreme quantile regression via GPD modelling of exceedances above high threshold

By the Pickands-de Haan-Balkema theorem, if the rescaled distribution of exceedances converges to a non degenerate distribution

$$\lim_{u\uparrow y^*} \mathbb{P}\left(\frac{Y-u}{f(u)} > y \mid Y > u\right) = 1 - H(y),$$

the limit H is necessarily a GPD distribution

$$H_{\gamma,\sigma}(y) = 1 - \left(1 + \gamma\frac{y}{\sigma}\right)_+^{-1/\gamma}.$$

This happens if and only if Y is in the max-domain of attraction of the extreme value distribution with index γ. The probability to exceed the high threshold u is denoted by $p = \mathbb{P}(Y > u)$. The approximation

$$\mathcal{L}(Y - u \mid Y > u) \overset{d}{\approx} H_{\gamma,\tilde{\sigma}}, \quad \tilde{\sigma} = f(u)\sigma,$$

implies that the quantiles of high order $1-\alpha$ are approximated by

$$q(\alpha) \approx u + \tilde{\sigma}\frac{(\alpha/p)^{-\gamma} - 1}{\gamma}, \quad \alpha < p.$$

For a covariate dependent model, the high threshold $u(x)$ may depend on x and we assume for simplicity that the exceedances are exactly GPD. This yields the model

$$\begin{cases} p(x) = \mathbb{P}(Y > u(x) \mid X = x), \\ \mathcal{L}(Y - u(x) \mid Y > u(x), X = x) = H_{\gamma(x),\sigma(x)}. \end{cases}$$

The corresponding conditional quantile for Y given $X = x$ with order $1-\alpha$ is

$$q(\alpha|x) = u(x) + \sigma(x)\frac{(\alpha/p(x))^{-\gamma(x)} - 1}{\gamma(x)}, \quad \alpha < p(x).$$

The estimation of $q(\alpha|x)$ is then obtained by plugging estimations of $p(x)$, $\sigma(x)$ and $\gamma(x)$.

For GPD estimation, maximum likelihood estimation is standard and provides asymptotically normal estimators in the i.i.d. case with $\gamma > -1/2$. It is natural to introduce the rescaled exceedance

$$Z = \max(Y - u(x), 0),$$

where $Z = 0$ corresponds to no exceedance. The negative log-likelihood at $\theta = (p, \sigma, \gamma)$ writes

$$\ell_z(\theta) = \Big[\log(1-p))1_{\{z=0\}} - \log p 1_{\{z>0\}}\Big] + \Big[(1 + 1/\gamma)\log\Big(1 + \gamma\frac{z}{\sigma}\Big) + \log\sigma\Big]1_{z>0}.$$

The first term corresponds to the Bernoulli negative log-likelihood and is related to the classification problem $(z > 0)$ VS $(z = 0)$, that is exceedance VS no exceedance. The second term is the GPD negative log-likelihood and is related to the GP modeling for exceedances $z > 0$ only.

In a statistical learning framework, we need to learn the function

$$\theta(x) = (p(x), \sigma(x), \gamma(x))$$

from an i.i.d. sample of observations (x_i, z_i), $1 \leq i \leq n$. For simplicity, we assume a deterministic threshold $u(x)$. Our proposal is to use gradient boosting (Friedman) and generalized random forest (Athey, Wager, Tisbshirani), mostly from a methodological point of view because theoretical properties seem difficult to tackle. In the following, we consider gradient boosting that is more straightforward to write in our specific setting and easier to code.

Gradient boosting for extreme quantile regression

We refer to The Elements of Statistical Learning, section 10.10 (Numerical Optimization via Gradient Boosting). The following algorithm is a natural adaptation of Algorithm 10.3 therein. The main difference is that in our framework, the function to learn $\theta(x)$ has three components, so that we will learn three sequences of trees - a similar strategy is used in multiclass classification where several sequences of trees are trained to learn the probabilities of the different classes (see Algorithm 10.4 in ESL).

For $\theta = (p, \sigma, \gamma)$, we use the notation $\theta = (\theta^p, \theta^\sigma, \theta^\gamma)$ and one generic component is noted θ^δ with $\delta \in \{p, \sigma, \gamma\}$. The algorithm runs as follows:

- data set: (x_i, z_i), $1 \leq i \leq n$.
- parameters:
  - B number of trees (same for three sequences),
  - $\lambda = (\lambda^p, \lambda^\sigma, \lambda^\gamma)$ learning rates,
  - $J = (J^p, J^\sigma, J^\gamma)$ numbers of leaves in the trees.
- Procedure:
  1. Initialize at
  $$\theta_0(x) \equiv \arg\min_\theta \sum_{i=1}^n \ell_{z_i}(\theta).$$
  This is simply the maximum likelihood estimator when the x_i's are ignored and the z_i's are assumed i.i.d. with likelihood ℓ.
  2. For $b = 1$ to B:
     (a) (gradient) For $\delta \in \{p, \sigma, \gamma\}$, compute the partial derivatives
     $$r_{bi}^\delta = \frac{\partial \ell_{z_i}}{\partial \theta^\delta}(\theta_{b-1}(x_i)), \quad 1 \leq i \leq n.$$
     (b) (tree regions) For $\delta \in \{p, \sigma, \gamma\}$, fit a regression tree $r_{bi}^\delta \sim x_i$, yielding J^δ terminal regions R_{bj}^δ, $1 \leq j \leq J^\delta$.
     (c) (tree values with line search) For $\delta \in \{p, \sigma, \gamma\}$ and $1 \leq j \leq J^\delta$, compute
     $$\gamma_{bj}^\delta = \arg\min_\gamma \sum_{x_i \in R_{bj}^\delta} \ell_{z_i}(\theta_{b-1}(x_i) + \gamma).$$

where γ acts on the δ-component only; define the tree

$$T_b^\delta(x) = \sum_{j=1}^{J^\delta} \gamma_{bj}^\delta 1_{\{x \in R_{bj}^\delta\}}.$$

(d) (update) For $\delta \in \{p, \sigma, \gamma\}$, update

$$\theta_b^\delta(x) = \theta_{b-1}^\delta(x) + \lambda^\delta T_b^\delta(x).$$

- Output: $\hat{\theta}(x) = \theta_B(x)$. The δ-component is the sum

$$\hat{\theta}(x) = \theta_0^\delta + \lambda^\delta \sum_{b=1}^{B} T_b^\delta(x).$$

In practice, the line search 2(c) is computationnaly too demanding and we use only one Newton-Raphson step. That is, the objective function is approximated by its Taylor expansion of order 2, yielding

$$\begin{aligned}
\tilde{\gamma}_{bj}^\delta &= \arg\min_\gamma \sum_{x_i \in R_{bj}^\delta} \left(\ell_{z_i}(\theta_{b-1}(x_i)) + \gamma \frac{\partial \ell_{z_i}}{\partial \theta^\delta}(\theta_{b-1}(x_i)) + \frac{\gamma^2}{2} \frac{\partial^2 \ell_{z_i}}{\partial \theta^{\delta 2}}(\theta_{b-1}(x_i)) \right) \\
&= -\frac{\sum_{x_i \in R_{bj}^\delta} \frac{\partial \ell_{z_i}}{\partial \theta^\delta}(\theta_{b-1}(x_i))}{\sum_{x_i \in R_{bj}^\delta} \frac{\partial^2 \ell_{z_i}}{\partial \theta^{\delta 2}}(\theta_{b-1}(x_i))}
\end{aligned}$$

We observed that the algorithm may be unstable, especially when tuning parameters are ill specified. Gradient steps are sometimes rather large leading to large decreases in deviance and possibly negative σ parameter. Therefore, we bound the absolute value of the gradient step by the learning rate for that parameter. Additionally, we reparamaterize $\sigma = \exp(\beta)$ and rewrote all the algorithm in terms of the β parameter.

The `gbex` package

In the current implementation of the `gbex` package, we assume that the data follow a GPD distribution with parameters depending on the covariate x through functions $\sigma(x)$ and $\gamma(x)$ and we construct two sequence of trees two estimate those functions. For now, we use

$$\begin{aligned}
&X_1, \ldots, X_d \sim \text{Unif}(-1, 1), \\
&\overline{X} = \frac{1}{d} \sum_{i=1}^{d} X_i^2, \\
&\sigma(x) = \exp(\overline{X}), \\
&\gamma(x) = \frac{1}{3} + \frac{1}{10} \overline{X} n \\
&Y \sim GPD(\sigma(x), \gamma(x)).
\end{aligned}$$

First we can install and load the package,

```
rm(list=ls())
require(rpart)
```

```
## Loading required package: rpart
```

```
require(treeClust)
```

```
## Loading required package: treeClust

## Loading required package: cluster
```

```
# package_directory <- "/Users/jjvelthoen/Documents/GitHub/gbex"
package_directory <- "C:/Users/cd-admin/Dropbox/RECHERCHE/JUAN/GPD Boosting/gbex"
install.packages(package_directory,repos=NULL,type="source",quiet=T)
```

```
## Warning in install.packages(package_directory, repos = NULL, type = "source", :
## installation of package 'C:/Users/cd-admin/Dropbox/RECHERCHE/JUAN/GPD Boosting/
## gbex' had non-zero exit status
```

```
library(gbex)
```

The data can be generated with the `get_data` function. We should soon incorporate arguments `fun_sigma` and `fun_gamma` to allow the user to provide his own parameter functions.

```
set.seed(1234)

### Data generating process ###
n <- 1000
d <- 2
data <- get_data(n,d)
s <- data$s # the sigma parameter
g <- data$g # the gamma parameter
y <- data$y # the response vector
X <- data.frame(X=data$X) # the covariates
colnames(X) = paste0("X",1:ncol(X))
```

The model contains several tuning parameters. The choice of B is crucial: $B = 0$ corresponds to constant parameters so that small value of B lead to underfitting while too large values of B lead to overfitting. The learning rates, minimum leaf size and sampling fraction act as regularization parameters and we would like to provide good default for them. The tree depth is interpreted as the model complexity and provides the order of interaction between covariates: depth=0 corresponds to constant parameter, depth=1 corresponds to additive modeling (without interaction between variables), depth=2 allows for interaction of order 2 and so on.

```
B=250  # Number of trees
lambda=c(0.01,0.0025) # Learning rate for sigma and gamma
min_leaf_size=c(10,10) # Minimum_leaf_size of trees for sigma and gamma
sf=0.75 # Subsampling fraction to use for each tree
depth=c(2,2) # Maximum depth of trees for sigma and gamma
```

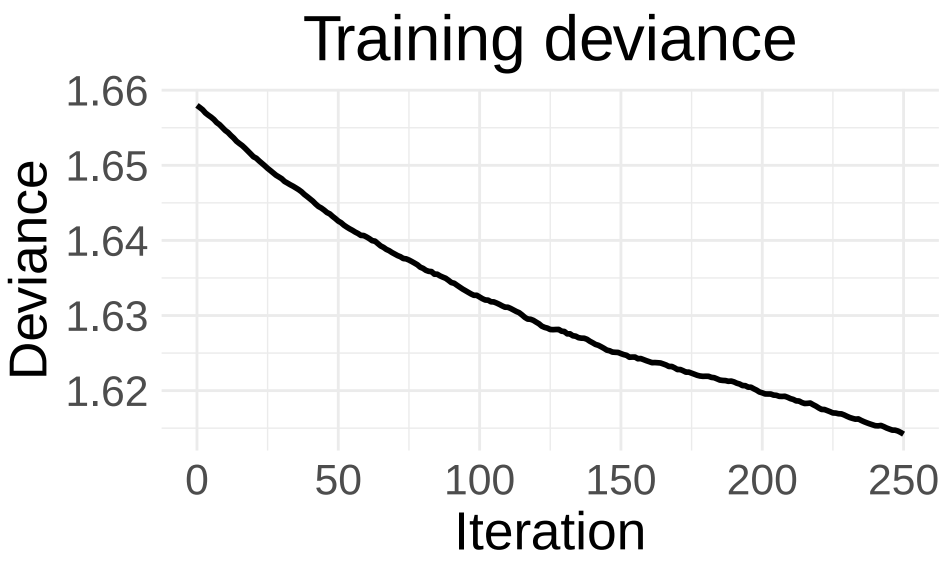
For fixed parameters, the `gbex` function is used to fit the boosting model. The model is fitted by gradient boosting using maximum likelihood when alpha = 0, and minimum power divergence when alpha>0.

```
fit <- gbex(y, X, B=B, lambda=lambda, depth=depth, min_leaf_size=min_leaf_size,
            sf=sf, alpha=0, silent=T)
print(fit)
```

```
## gbex(y = y, X = X, B = B, lambda = lambda, depth = depth, min_leaf_size = min_leaf_size,
##      sf = sf, alpha = 0, silent = T)
## A gradient boosting model for extremes fitted by likelihood.
## 250 trees are fitted.
## Training error was equal to 1.6142066602263.
```

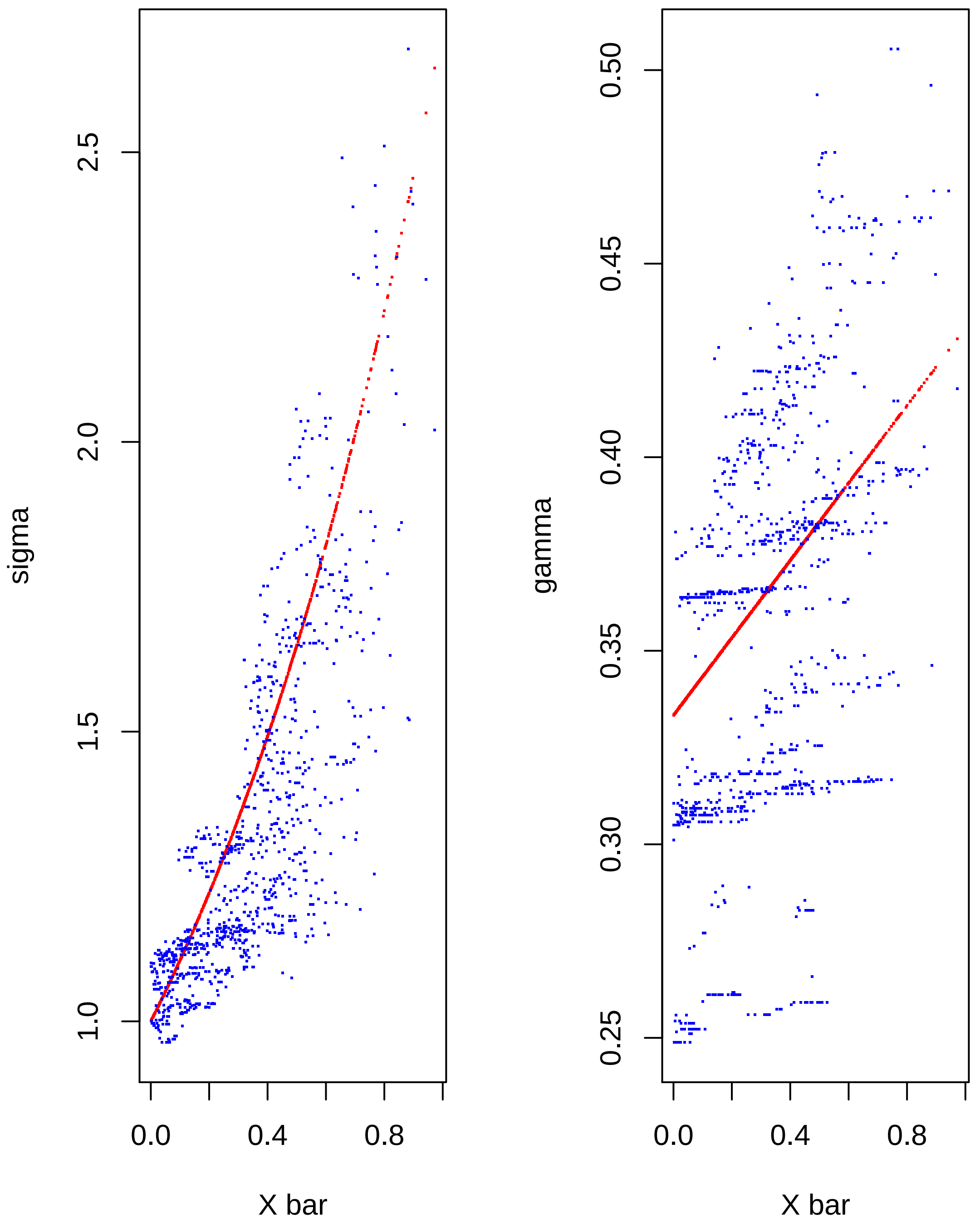
We can check that the deviance (negative log-likelihood) decreases when trees are added to the model, which is precisely the goal of gradient boosting. Here the training deviance is considered, that is deviance on the training set. This diagnostic does not allow to detect overfitting.

```r
plot(fit)
```



In this simulation study, the covariate are 2-dimensional but the parameters depend on the 1-dimensional index $\bar{X}$. As a graphical check, we plot in blue the estimated parameters and in red the true parameters against the index $\overline{X}$.

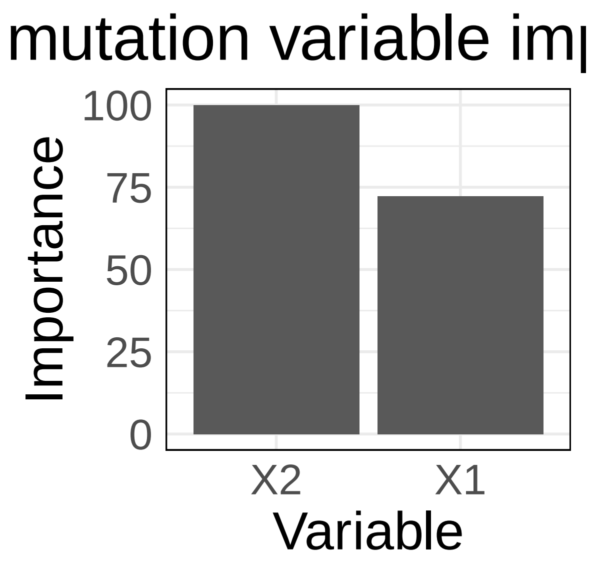
```r
x_bar=apply(X^2, 1,mean)
par(mfrow=c(1,2))
plot(x_bar,s,pch='.', col='red',ylim=c(range(c(s,exp(fit$theta$b)))),
     xlab="X bar",ylab="sigma")
points(x_bar,exp(fit$theta$b),pch='.',col='blue')
plot(x_bar,g,pch='.', col='red',ylim=c(range(c(g,fit$theta$g))),
     xlab="X bar",ylab="gamma")
points(x_bar,fit$theta$g,pch='.',col='blue')
```



We next describe two important diagnostic tools that are standard for random forest and boosting models: variable importance and partial dependentce plot.

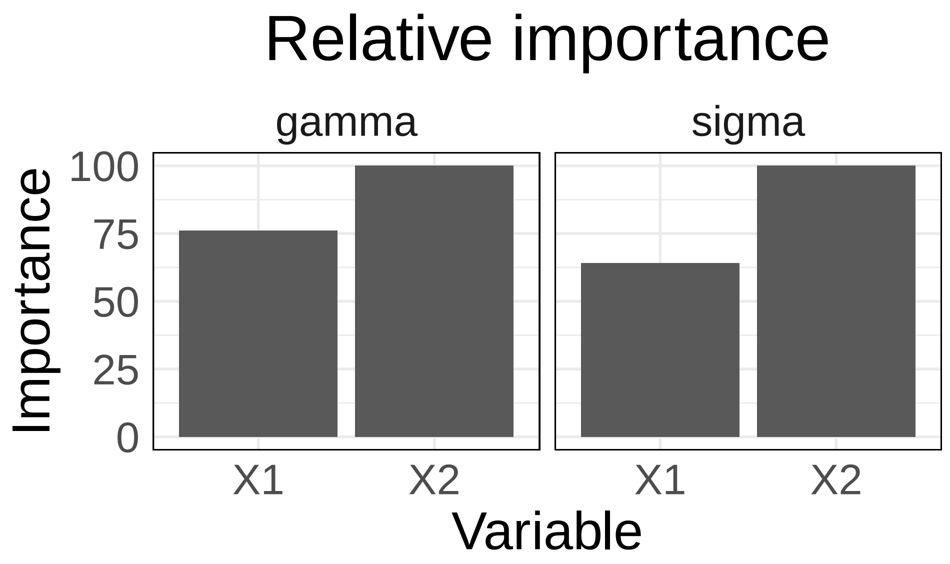
The permutation variable importance measures the increase in deviance when all values for a single covariate are permuted. The variable importance are rescaled so that the largest is equal to 100. It is computed thanks to the `variable_importance` function with `type="permutation"`.

```
variable_importance(fit,type="permutation")
```



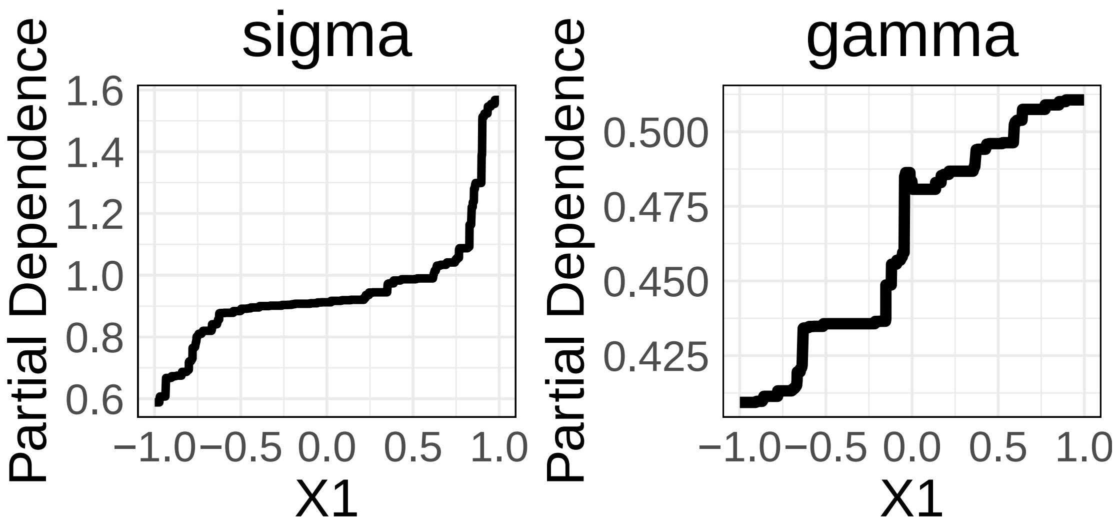
Alternatively, variable importance can be computed by inspecting all the trees involved in the boosting model and recording the deviance decrease due to the different covariates. Because two sequence of trees are built, we get variable importance relatively two both parameters. The option `type="relative"` is used for that purpose. Although our model is symmetric in X_1 and X_2, we can see that on this set of realizations, the variable X_2 has a slightly stronger contribution to the model.

```r
variable_importance(fit,type="relative")
```

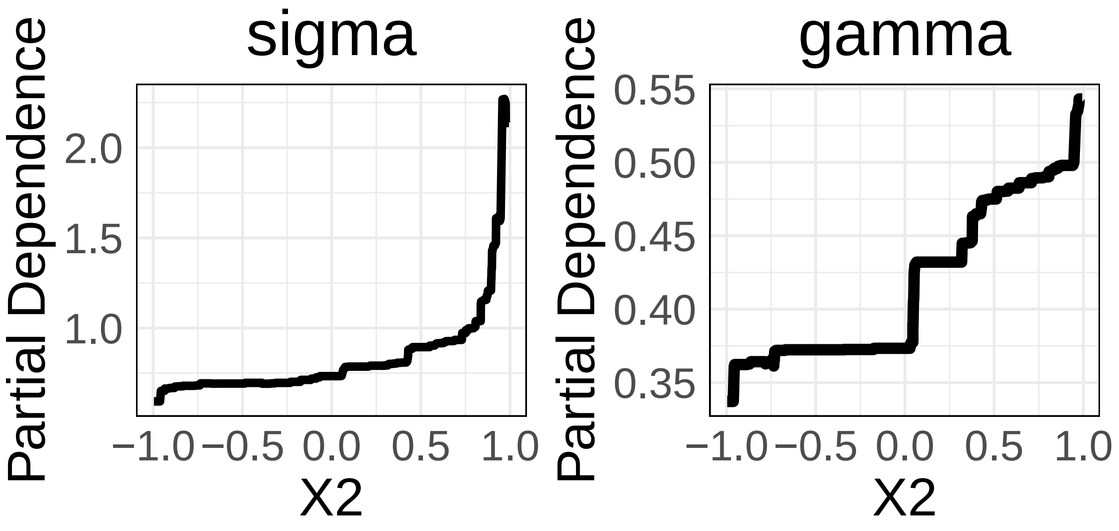


Finally, it is important to have an idea of the dependence of the parameters σ and γ with respect to the different covariates. The partial dependence plots are meant to provide such an information.

```r
partial_dependence(fit,variable = 1)
```



```
partial_dependence(fit,variable = 2)
```



On this simulation study, the results are sensible. The model parameters are estimated reasonably well. The deviance is decreasing and the dependence on the covariates approximates the true dependence.

Cross validation

Cross validation for the number of trees

One major issue is to tune the parameters and select a good model. Crucial is the choice of B and we want to avoid both underfitting and overfitting. For this, k-fold cross-validation is commonly use. We divide the sample into k-folds, leave out one fold for test and fit the model on the remaining $k-1$ folds. We monitor the deviance on the test fold when trees are added to model fitted on the training folds.

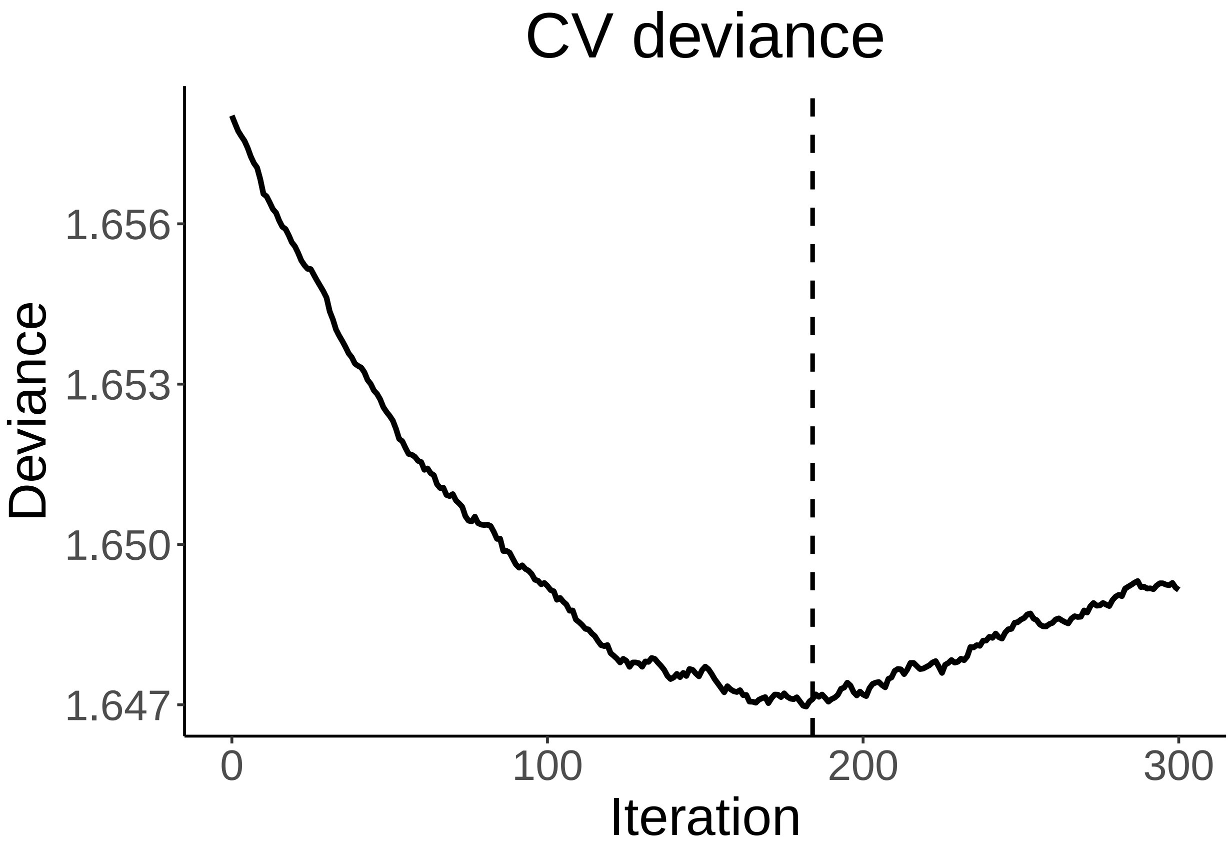
In the context of extreme value, it is knwon that extreme observations act as leverage points with a strong influence on maximum likelihood estimators (especially regarding the shape parameter γ). For this reason, we propose to use k-folds cross validation with stratification where the extreme values are evenly distributed into the different folds. The stratification is performed using the order statistics $Y_{(1)} \geq \dots Y_{(n)}$ and each group of k consecutive order statistics is randomly dispatched into the different folds. More precisely, the observations $(Y_{(k*j+1)}, Y_{(k*j+2)}, \dots, Y_{(k*j+k)})$ are randomly assigned to different folds.

The choice of B, the number of trees, is performed thanks to the `CV_gbex` function. The number of folds is indicated thanks to `num_folds=` and `stratified=` indicates wether stratification should be used for defining the k folds. The argument `par="B"` indicates that we perform cross-validation for B and then a maximum number `Bmax` of trees must be specified. It must be large enough to prevent underfitted models. The `plot` method then shows the mean deviance across the different folds and its minimum.

```
CV_fit = CV_gbex(y, X, num_folds=6, Bmax=300, stratified=T,
                 lambda=lambda, depth=depth, min_leaf_size=min_leaf_size, sf=sf,silent=T)
print(CV_fit)
```

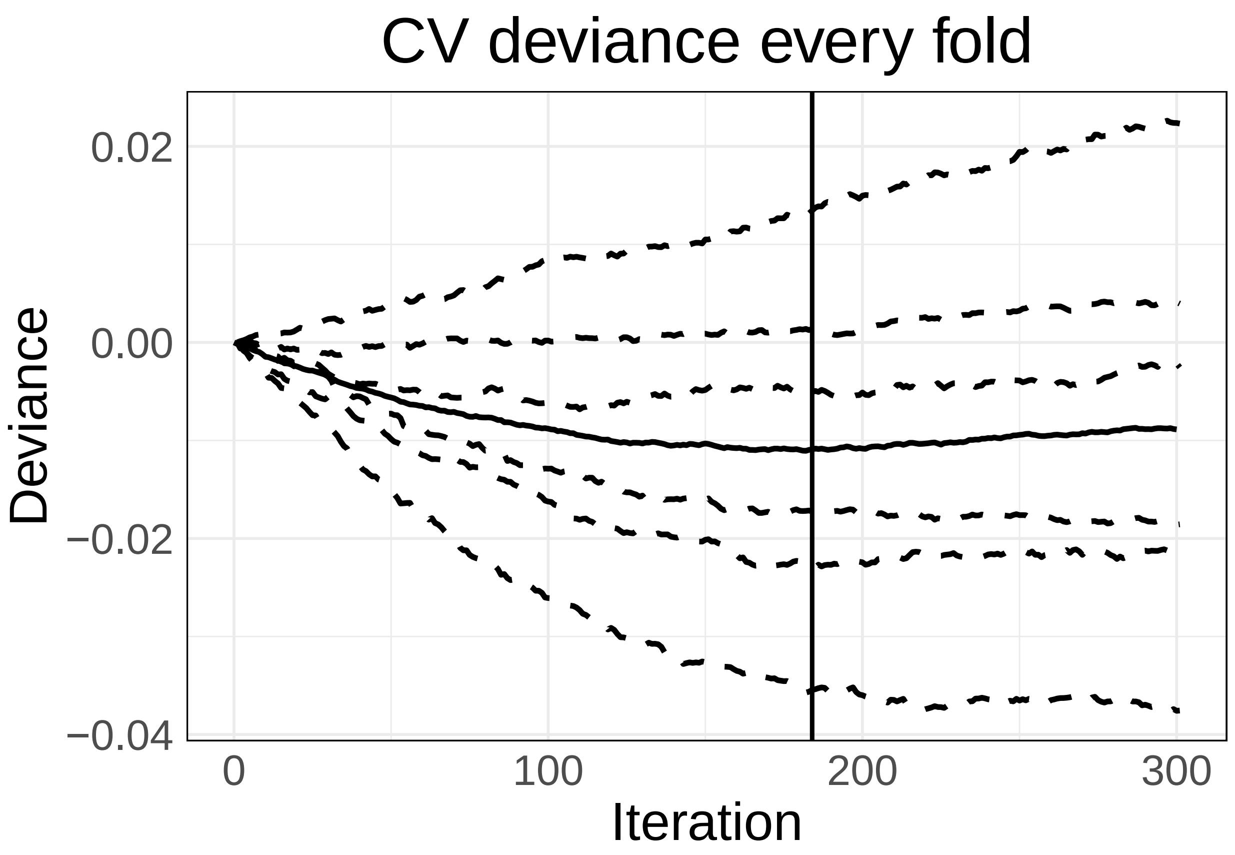
```
## CV_gbex(y = y, X = X, num_folds = 6, Bmax = 300, stratified = T,
##      lambda = lambda, depth = depth, min_leaf_size = min_leaf_size,
##      sf = sf, silent = T)
## A cross validation object for gbex for parameter B.
## The number of folds is 6 which are obtained by stratified sampling.
## The optimal parameter value is B = 182.
```

```
plot(CV_fit, what="general")
```



A more in-depth look on the cross validation procedure is obtained by plotting the deviance in the different folds. In order to compare the folds more easily, the decrease of deviance with respect to the unconditional model (B=0) is plotted. Values lower (higher) than zero indicate a better (worse) performance compared to the unconditional model.

```
plot(CV_fit, what="per_fold")
```

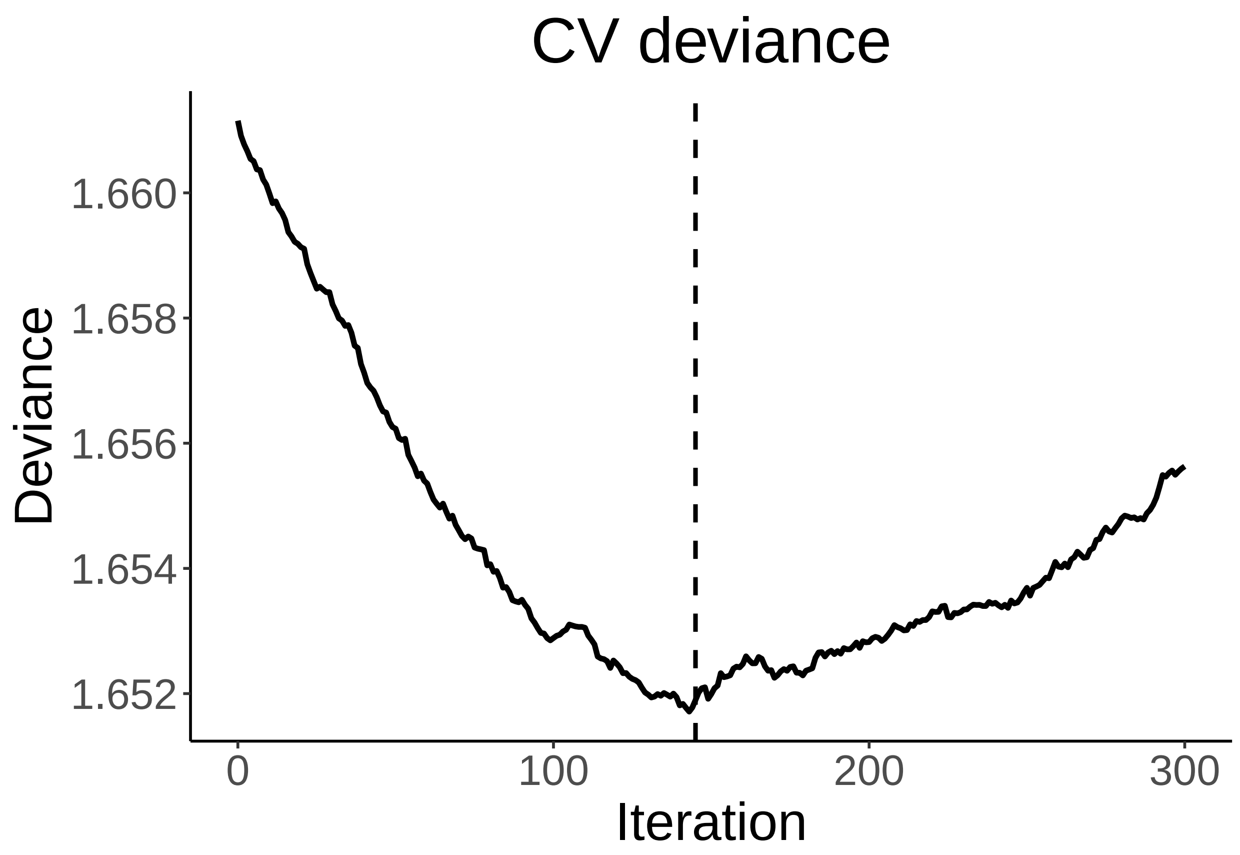


For the purpose of comparison, let us see what we get without stratification.

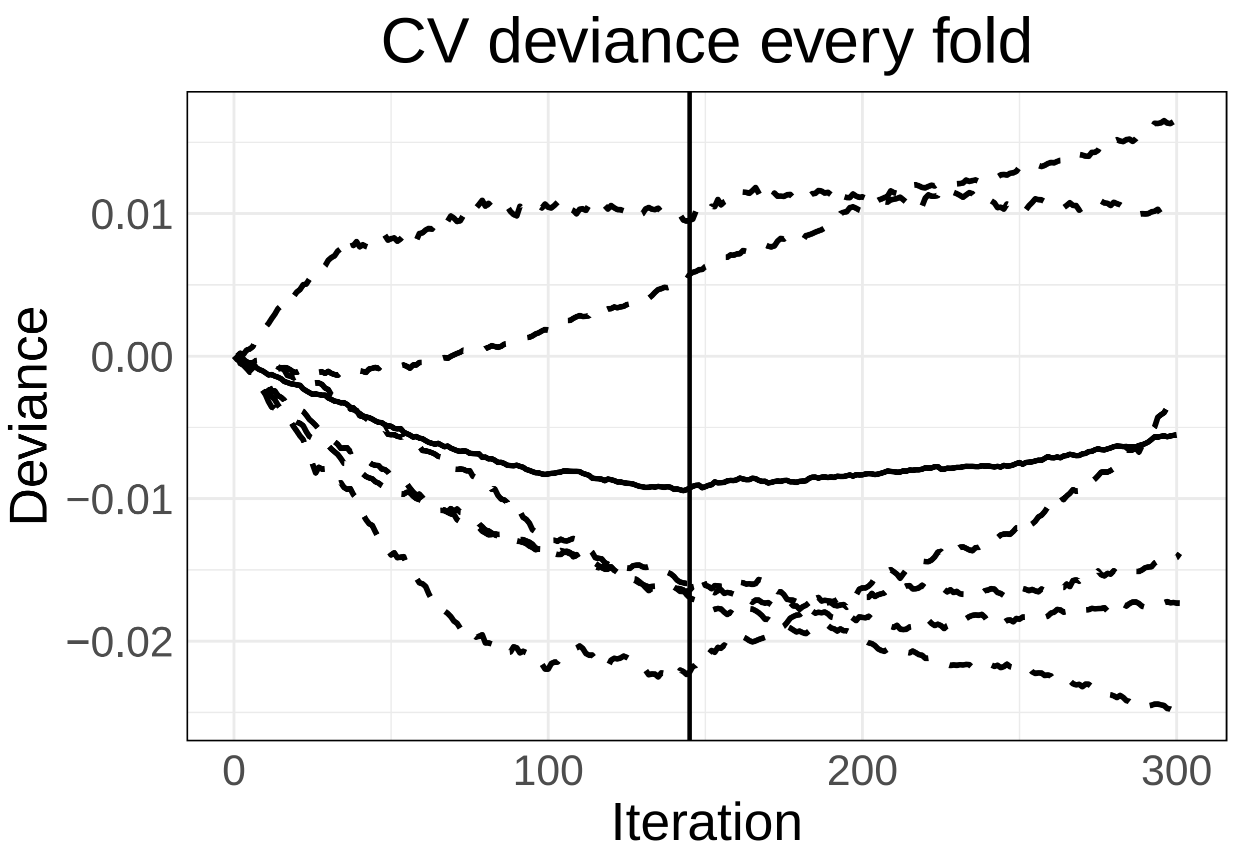
```
CV_fit = CV_gbex(y, X, num_folds=6, Bmax=300, stratified=F,
                 lambda=lambda, depth=depth, min_leaf_size=min_leaf_size, sf=sf,silent=T)
print(CV_fit)
```

```
## CV_gbex(y = y, X = X, num_folds = 6, Bmax = 300, stratified = F,
##      lambda = lambda, depth = depth, min_leaf_size = min_leaf_size,
##      sf = sf, silent = T)
## A cross validation object for gbex for parameter B.
## The number of folds is 6 which are obtained by random sampling.
## The optimal parameter value is B = 143.
```

```
plot(CV_fit, what="general")
```



```r
plot(CV_fit, what="per_fold")
```



Cross validation for other parameters

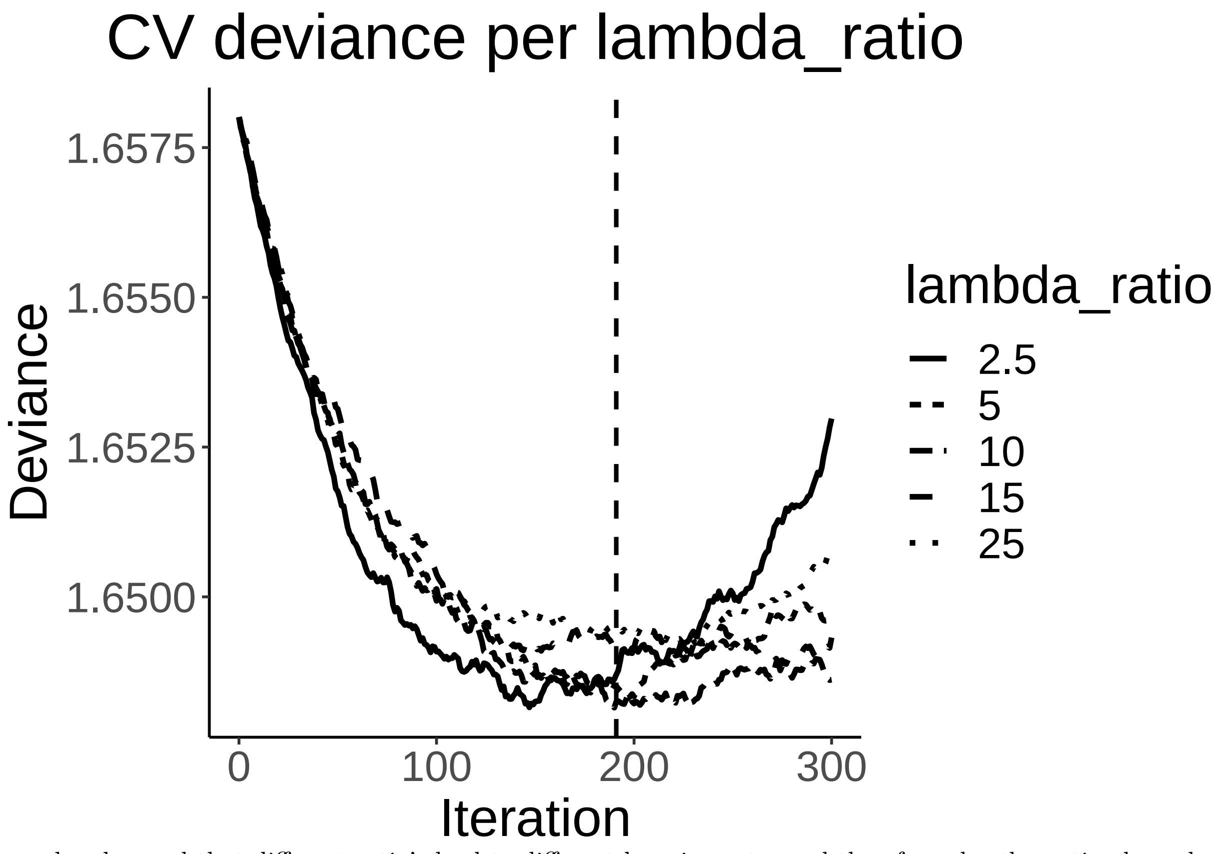
Many of the other parameters are hard to optimize, but the cross validation routine can also be used to try and optimize these parameters. One of the most important parameters is the *lambda* parameter. The σ and γ parameters depend in the optimization on each other. Taking too large steps with one will therefore negatively influence the other. For this reason we reperametrize λ by $\lambda_{ratio} = \frac{\lambda_{sigma}}{\lambda_{gamma}}$ and $\lambda_{scale} = \lambda_{sigma}$. By choosing λ_{scale} small enough we can use the cross validation function to find the optimal λ_{ratio}.

```
grid_lambda_ratio = c(2.5,5,10,15,25)
CV_lambda_ratio = CV_gbex(y,X,num_folds = 6,par_name="lambda_ratio", Bmax=300,par_grid = grid_lambda_ra
print(CV_lambda_ratio)
```

```
## CV_gbex(y = y, X = X, num_folds = 6, Bmax = 300, par_name = "lambda_ratio",
##      par_grid = grid_lambda_ratio, stratified = T, lambda_scale = 0.01,
##      depth = depth, min_leaf_size = min_leaf_size, sf = sf, silent = T)
## A cross validation object for gbex for parameter lambda_ratio.
## The number of folds is 6 which are obtained by stratified sampling.
## The optimal parameter value is lambda_ratio = 10.
```

Where we can again have a look at cross validation results for the different parameter values,

```
plot(CV_lambda_ratio,what="general")
```



It can be observed that different ratio's lead to different learning rates and therefore also the optimal number of trees is different. It is therefore important for each tuning parameter to allow for different values of B.