# User's Guide to **lqa**

Jan Ulbricht

April 17, 2010

## 1 Introduction

Ulbricht (2010b) has introduced the quite generic LQA algorithm to estimate Generalized Linear Models (GLMs) based on penalized likelihood inference. Here we do not want to repeat details on how to specify or estimate GLMs. For a nice overview on these topics see, for example, McCullagh and Nelder (1989), Fahrmeir and Tutz (2001) or Fahrmeir et al. (2009). Instead we provide you in a nutshell with the basic aspects of the LQA algorithm.
Consider the vector $\mathbf{b} = (\beta_0, \boldsymbol{\beta}^\top)^\top$ of unknown parameters in the predictor of a GLM. The term $\beta_0$ denotes the intercept and $\boldsymbol{\beta}$ contains the coefficients of the $p$ regressors. We want to solve the penalized regression problem

$$\min_{\mathbf{b}} -\ell(\mathbf{b}) + P_{\lambda}(\boldsymbol{\beta}), \tag{1}$$

where $\ell(\mathbf{b})$ is the log-likelihood of the underlying GLM and the penalty term has structure

$$P_{\lambda}(\boldsymbol{\beta}) = \sum_{j=1}^{J} p_{\lambda,j}(|\mathbf{a}_j^\top \boldsymbol{\beta}|), \tag{2}$$

with known vectors of constants $\mathbf{a}_j$. The subscript $\boldsymbol{\lambda}$ illustrates the dependency on a vector of tuning parameters. We assume the following properties to hold for all $J$ penalty terms

(i) $p_{\lambda,j} : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ with $p_{\lambda,j}(0) = 0$,

(ii) $p_{\lambda,j}$ is continuous and monotone in $|\mathbf{a}_j^\top \boldsymbol{\beta}|$,

(iii) $p_{\lambda,j}$ is continuously differentiable for all $\mathbf{a}_j^\top \boldsymbol{\beta} \neq 0$, so that $dp_{\lambda,j}(|\mathbf{a}_j^\top \boldsymbol{\beta}|)/d|\mathbf{a}_j^\top \boldsymbol{\beta}| \geq 0$ for all $\mathbf{a}_j^\top \boldsymbol{\beta} > 0$.

The first two assumptions are necessary to make sure that $P_{\lambda}$ is indeed a penalty. By construction the penalty is symmetric in $\mathbf{a}_j^\top \boldsymbol{\beta}$ around the origin.
The algebraic form of (2) covers a wider range of penalties, such as polytopes and quadratic penalties. If the penalty region is a polytope, then all penalty functions $p_{\lambda,j}$

must be linear. Note that the sum of all $J$ penalty functions determines the penalty term. The penalty level $\lambda$ must not be identical for all $J$ functions but to keep the notation simple we omit a further subindex $j$. Furthermore, the penalty function type must not be identical for all $j$. The number $J$ of penalty functions is not necessarily equal to $p$ the number of dimensions. For example, the fused lasso penalty (Tibshirani et al., 2005)

$$P_\lambda^{fl}(\boldsymbol{\beta}) = \lambda_1 \sum_{j=1}^{p} |\beta_j| + \lambda_2 \sum_{k=2}^{p} |\beta_k - \beta_{k-1}|$$

can be written as

$$P_\lambda^{fl}(\boldsymbol{\beta}) = \sum_{j=1}^{2p-1} p_{\lambda,j}(|\mathbf{a}_j^\top \boldsymbol{\beta}|), \tag{3}$$

where

$$p_{\lambda,j}(\cdot) = \lambda_1 |\mathbf{a}_j^\top \boldsymbol{\beta}|, \quad j = 1, \dots, p$$

and $\mathbf{a}_j = (0, \dots, 0, 1, 0, \dots, 0)^\top$ with a one at the $j$-th position, and

$$p_{\lambda,j}(\cdot) = \lambda_2 |\mathbf{a}_j^\top \boldsymbol{\beta}|, \quad j = p+1, \dots, 2p-1$$

where $\mathbf{a}_j = (0, \dots, 0, -1, 1, 0, \dots, 0)^\top$ with a one at the $(j-p+1)$-th position and a minus one at the $(j-p)$-th position. The parameters $\lambda_1$ and $\lambda_2$ correspond to $s_1$ and $s_2$ in the constrained regression problem formulation, see Tibshirani et al. (2005).

Ulbricht (2010b) has shown that if the penalty term has structure (2) then the penalized regression problem (1) can be solved by a Newton-type algorithm based on local quadratic approximations of the penalty term. That is, the unknown coefficients $\mathbf{b}$ can be estimated iteratively with the estimation equation

$$\mathbf{b}_{(k+1)} = \mathbf{b}_{(k)} - \gamma \left( \mathbf{F}(\mathbf{b}_{(k)}) + \mathbf{A}_\lambda^* \right)^{-1} \{ -\mathbf{s}(\mathbf{b}_{(k)}) + \mathbf{A}_\lambda^* \mathbf{b}_{(k)} \}, \tag{4}$$

where $\mathbf{s}(\mathbf{b})$ and $\mathbf{F}(\mathbf{b})$ denote score vector and Fisher information matrix of the underlying log-likelihood, respectively, and

$$\mathbf{A}_\lambda^* = \begin{bmatrix} 0 & \mathbf{0}_p^\top \\ \mathbf{0}_p & \mathbf{A}_\lambda \end{bmatrix},$$

where $\mathbf{0}_p$ is the $p$-dimensional null vector. The (approximated) penalty matrix $\mathbf{A}_\lambda$ is defined as

$$\mathbf{A}_\lambda = \sum_{j=1}^{J} \frac{p'_{\lambda,j}(|\mathbf{a}_j^\top \boldsymbol{\beta}_{(k)}|)}{\sqrt{\left(\mathbf{a}_j^\top \boldsymbol{\beta}_{(k)}\right)^2 + c}} \mathbf{a}_j \mathbf{a}_j^\top, \tag{5}$$

where $p'_{\lambda,j}(|\mathbf{a}_j^\top \boldsymbol{\beta}_{(k)}|) = dp_{\lambda,j}(|\mathbf{a}_j^\top \boldsymbol{\beta}|)/d|\mathbf{a}_j^\top \boldsymbol{\beta}|$ denotes the first derivative with $p'_{\lambda,j}(0) \equiv 0$, and $c > 0$ is a small positive real number. In the **lqa** package we will use $c = 1e^{-6}$ as default value. To our experience, this value works quite well.

The Newton update (4) is repeated until convergence. We apply the rule

$$\frac{\|\mathbf{b}_{(k+1)} - \mathbf{b}_{(k)}\|}{\|\mathbf{b}_{(k)}\|} \le \epsilon, \quad \epsilon > 0$$

to terminate the algorithm. Under this criterion, the algorithm is stopped if the relative distance moved during the $k$-th iteration is less or equal to $\epsilon$. We use an additional step length parameter $0 < \gamma \leq 1$ to enhance convergence of the algorithm. This parameter is usually treated as fixed constant. In our experience a value $\gamma < 1$ becomes especially appropriate if the penalty term includes an $L_\iota$-norm with $\iota$ 'large', e.g. $\iota \geq 10$. Examples where this can happen are the bridge (Frank and Friedman, 1993) or the OSCAR (Bondell and Reich, 2008) penalty.

In this article we will describe the R (R Development Core Team, 2009) add-on package **lqa**. The **lqa** package has been originally designed to apply the LQA algorithm (see Ulbricht, 2010b, for details) to compute the constrained MLEs of GLMs with specific penalties. In the further development the package has been extended to deal also with boosting algorithms such as componentwise boosting, GBlockBoost and ForwardBoost (Ulbricht, 2010b) which in turn are primarily intended to be used with quadratic penalties. The LQA algorithm can be viewed as an extension of the P-IRLS algorithm (see, e.g., Wood, 2006), so that the latter is also (indirectly) included.

As we will see later on we use a hierarchical structure of R functions for the package that provides preparation methods (standardization, extraction of formula components etc.), the computation of important statistics (degrees of freedom, AIC, penalized Fisher information matrix etc.) and generic functions (summary, plot etc.). A convenient way to specify the details of the penalty terms is crucial for the computational application of the LQA algorithm and the boosting methods mentioned above. Therefore we start with the introduction of the `penalty` class in the next section. The basic structure of the **lqa** package will be explained in Section 3, some examples for applications are illustrated in Section 4.

## 2 The `penalty` class

Since the **lqa** package focuses on shrinkage and boosting methods there will always be a penalty included in the objective function or plays a role during the inference procedure. Furthermore, we need to compute different terms corresponding to a specified penalty, such as the penalty level, or the coefficients or its gradients for the LQA algorithm. A concrete penalty consists of the penalty family and chosen tuning parameters. It will be reasonable to specify a penalty family and the tuning parameters separately. This is especially necessary for cross-validation procedures and plotting of coefficient build-ups. So the R source code for specifying e.g. the ridge penalty with tuning parameter $\lambda = 0.7$ should look like this

```
R> penalty <- ridge (lambda = 0.7)
```

The entities R operates on are technically known as *objects*. Hence we require a useful R object to represent penalties. Many programming languages, including R and S, use concepts from object-oriented programming (OOP). As given in Leisch (2008), the two main of those concepts as used in R are

- *classes* to define how objects of a certain type look like, and
- *methods* to define special functions operating on objects of a certain class.

For dealing with several penalties, also user-defined ones, we introduce a class of R objects called `penalty`. There are two approaches in R for dealing with classes, that is the 'older' S3 approach and the S4 approach as introduced in Chambers (1998). Most classes in R are based on the classical S3 concept. The more advanced S4 concept requires some more code but delivers more flawless definitions. However, both concepts have pros and cons. Some more details and examples are e.g. given in Venables and Ripley (2000).

In S4 classes the structure of an object is clearly specified in the term of *slots*. This leads to a huge amount of formality and precision. The structure of an S3 object can only be defined implicitly using the function `structure()`. As a direct consequence, checks on data structures or arguments can be swapped out to the initialization (or prototype) function. This possibility is only of limited merit when dealing with penalties due to a huge amount of heterogeneity among different penalties. Reasons for it are e.g. a different number of tuning parameters, or that the computation of the penalty matrix can depend on the regressor matrix $\mathbf{X}$ as for the correlation-based penalty (Tutz and Ulbricht, 2009) or just on dimension $p$ and information whether or not there is an intercept included in the model (ridge penalty in linear models). Consequently, checks on correct specification must also be customized and hence included in the several penalty objects.

| Method | Type | Input arguments | Output |
|---|---|---|---|
| `penalty` | character | | a character containing the penalty name |
| `lambda` | numeric vector | | a numeric vector containing the tuning parameters |
| `getpenmat()` | function | `beta = NULL, ...` | $p \times p$ penalty matrix evaluated at `beta` if necessary |
| `first.derivative()` | function | `beta = NULL, ...` | $J$ dimensional vector containing $p'_{\lambda,1}, \dots, p'_{\lambda,J}$ |
| `a.coefs()` | function | `beta = NULL, ...` | $p \times J$ matrix containing the coefficients $\mathbf{a}_1, \dots, \mathbf{a}_J$. |

Table 1: Methods operating on objects of class `penalty`. The dot operator `...` allows for some further arguments.

Furthermore, the clear specification of S4 classes is another drawback concerning our applications. When dealing with quadratic penalties

$$P_\lambda(\boldsymbol{\beta}) = \frac{1}{2}\boldsymbol{\beta}^\top \mathbf{M}_\lambda \boldsymbol{\beta}, \tag{6}$$

where $\mathbf{M}_\lambda$ is a positive definite matrix, we would just need the class character `penalty$penalty` (to indicate the name of the penalty) and the function `penalty$getpenmat()`. The latter returns the evaluated $(p \times p)$-dimensional penalty matrix $\mathbf{M}_\lambda$ based on the argument `beta` if necessary. Contrary, polytopes as penalties might require some more advanced functions such as `penalty$first.derivative()` or `penalty$a.coefs()` (see Table 1 below) to evaluate the penalty for a given $\boldsymbol{\beta}$ vector of coefficients. If a user wants to implement a new quadratic penalty then these further functions are actually not needed. However, when using S4 methods this would result in an error message or alternatively we would need two classes (`quad.penalty` and `penalty`) instead of just one where `quad.penalty` is a subclass of `penalty`. For these reasons, we use the S3 methods concept and put up with the in some way less comfortable implementation of new penalty instances. The penalties already implemented as `penalty` objects are listed in Table 2.

In the following we describe what kinds of methods are required for specific penalty families. The five basic methods for `penalty` objects are summarized in Table 1. The methods `penalty` and `lambda` are mandatory. They are necessary to identify the penalty family and, respectively, the tuning parameter vector in the R functions of **lqa**. But as we will see later on, they just appear as list elements in the `structure()` environment. The function `getpenmat()` and the functions `first.derivative()` and `a.coefs()` are mutually exclusive. Whether we need the first one or the last two depends on the nature of the penalty. Hence we have to distinguish two cases (see also Table 2):

(i) The use of a function `getpenmat()` is more efficient (in a numerical sense) if

- the penalty matrix $\mathbf{A}_\lambda$ as given in (5) is a diagonal matrix, e.g. if $J = p$ and $\mathbf{a}_j$, $j = 1, \ldots, J$ just contains one non-zero element, or
- the penalty is quadratic.

Then the (approximate) penalty matrix $\mathbf{A}_\lambda$ can be computed directly. Most implemented penalties are of those types, e.g. ridge, lasso, SCAD and correlation-based penalty.

(ii) The combination of the functions `first.derivative()` and `a.coefs()` is necessary in all other cases. The fused lasso penalty is an example for it.

We illustrate the basic structure of `penalty` objects with the `lasso` and the `fused.lasso` object. For the lasso penalty (Tibshirani, 1996)

$$P_\lambda^l(\boldsymbol{\beta}) = \sum_{j=1}^{p} p_{\lambda,j}(|\mathbf{a}_j^\top \boldsymbol{\beta}|),$$

with $\mathbf{a}_j = (0, \ldots, 0, 1, 0, \ldots, 0)^\top$, where the one is at the $j$-th position, and $p_{\lambda,j}(|\xi_j|) = \lambda|\xi_j|$, $\xi_j = \mathbf{a}_j^\top \boldsymbol{\beta}$ it is straightforward to show that the approximate penalty matrix is

$$\mathbf{A}_\lambda^l = \lambda \operatorname{diag}\left\{1_{\{\beta_1 \neq 0\}}(\beta_1^2 + c)^{-1/2}, \ldots, 1_{\{\beta_p \neq 0\}}(\beta_p^2 + c)^{-1/2}\right\} \tag{7}$$

| Name | Description | `getpenmat()` | `first.derivative()` + `a.coefs()` |
|---|---|---|---|
| `ridge` | ridge penalty | x | |
| `penalreg` | correlation-based penalty | x | |
| `lasso` | lasso penalty | x | |
| `adaptive.lasso` | adaptive lasso | x | |
| `fused.lasso` | fused lasso | | x |
| `oscar` | oscar | | x |
| `scad` | scad | x | |
| `weighted.fusion` | weighted fusion | x | |
| `bridge` | bridge | x | |
| `enet` | Elastic net | x | |
| `genet` | Generalized elastic net | x | |
| `icb` | Improved correlation-based penalty | x | |
| `licb` | $L_1$-norm improved correlation-based penalty | | x |
| `ao` | Approximated octagon penalty | x | |

Table 2: Implemented penalties of class `penalty`. Whether they consist of `getpenmat()` or of `first.derivative()` and `a.coefs()` is tagged with an ‘x’.

for some small $c > 0$, where $1_{\{\beta_j \neq 0\}} = 1$ if $\beta_j \neq 0$ and $1_{\{\beta_j \neq 0\}} = 0$ otherwise. Thus we might use the `getpenmat()` function.

The source code of the complete implementation of the lasso penalty is given in the following:

```
R> lasso <- function (lambda = NULL, ...) {
+   lambda.check (lambda)
+
+ ## Check on dimensionality of lambda
+   if (length (lambda) != 1)
+     stop ("lambda must be a scalar \n")
+
+   names (lambda) <- "lambda"
+
+   getpenmat <- function (beta = NULL, c1 = lqa.control()$c1, ...) {
+      if (is.null (beta))
+        stop ("'beta' must be the current coefficient vector \n")
+
+      if (c1 < 0)
+        stop ("'c1' must be non-negative \n")
+
+      penmat <- lambda * diag (1 / (sqrt (beta^2 + c1)))
+                * as.integer (beta != 0)
+      penmat }
+
+  structure (list (penalty = "lasso", lambda = lambda, getpenmat =
+    getpenmat), class = "penalty")}
```

Here you see the basic structure of a `penalty` object that might be kept all the time. If you want to implement a new `penalty` object then you should respect the following annotations. The object is initialized by calling the function of its name with `lambda` as argument. If your tuning parameter is a vector, then `lambda` must be a vector, too. You could provide some additional arguments if necessary, such as control parameters for numerical precision. The **lqa** control parameters such as the return of a call to the function `lqa.control()` are *not* meant by this. They are incorporated, if necessary, in the particular penalty methods such as the `c1` argument that corresponds with $c$ in (7), in the `getpenmat()` function. If your penalty depends on the regressor matrix or its correlation matrix, as e.g. the correlation-based penalty, then it is more reasonable to use the corresponding argument, e.g. the `x` argument, just in the particular functions, such as `getpenmat()`. This assures to relate to the right subset of regressors when indicated. Otherwise `x` would be initialized as a global argument of your penalty and hence can cause trouble if you want to apply methods with explicit variable selection such as componentwise boosting.

But now we come back to the basic structure. In a first step, you should call `lambda.check()`. The internal function `lambda.check()` just checks the `lambda` argument on existence (`!is.null()`) and element-wise nonnegativity. In a second step you should check the right dimensionality of the `lambda` argument. Afterwards you might name the elements of `lambda`. This helps to identify the tuning parameters of different methods when you want to compare their performances. In a next step, the `getpenmat()` method is implemented. To compute (7) we need the arguments `beta` that indicates the (current) $\boldsymbol{\beta}$ vector and `c1` that corresponds to $c$. According to the approximation (5) and the arguments made about this we must make sure that

$$p'_{\lambda,j}(0) \equiv 0.$$

Therefore we use `as.integer (beta != 0)` as a multiplier in the computation of `penmat` which is in fact the R representation of the indicator function. Finally, the object is initialized using the function `structure()`, where all its elements are listed. Note that the membership of the class `penalty` must also be given here.

For the fused lasso penalty, things are a little bit more complicated. Since $\mathbf{a}_j$, $j = p+1,\dots,2p-1$ consists of two non-zero elements we cannot apply `getpenmat()`. The first derivatives of the penalty terms are

$$p'_{\lambda,j}(|\xi_j|) = \begin{cases} \lambda_1 1_{\{\xi_j \neq 0\}}, & j = 1,\dots,p, \\ \lambda_2 1_{\{\xi_j \neq 0\}}, & j = p+1,\dots,2p-1. \end{cases}$$

This will be returned by the `first.derivative()` function, see below. A summarized version of the coefficients is

$$\begin{array}{cccccccc} \mathbf{a}_1 & \mathbf{a}_2 & \dots & \mathbf{a}_p & \mathbf{a}_{p+1} & \mathbf{a}_{p+2} & \dots & \mathbf{a}_{2p-1} \\ \end{array}$$
$$\left[\begin{array}{cccccccc} 1 & 0 & \dots & 0 & -1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 & 1 & -1 & \dots & 0 \\ 0 & 0 & \dots & 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 0 & 0 & 0 & \dots & 0 \\ 0 & 0 & \dots & 0 & 0 & 0 & \dots & -1 \\ 0 & 0 & \dots & 1 & 0 & 0 & \dots & 1 \end{array}\right].$$

This $(p \times (2p-1))$-dimensional matrix will be returned by `a.coefs()`. So the complete source code of the `fused.lasso` object is:

```
R> fused.lasso <- function (lambda = NULL, ...){
+   lambda.check (lambda)
+   if (length (lambda) != 2)
+     stop ("The fused.lasso penalty must consist on two parameters! \n")
+   names (lambda) <- c ("lambda1", "lambda2")
+
+  first.derivative <- function (beta = NULL, ...){
+    if (is.null (beta))
+      stop ("'beta' must be the current coefficient vector \n")
+    p <- length (beta)
+    if (p < 2)
+      stop ("There must be at least two regressors! \n")
+    vec1 <- c (rep (lambda[1], p), rep (lambda[2], p - 1))
+    return (vec1 * as.integer (drop (t (a.coefs (beta)) %*% beta) != 0))}
+
+  a.coefs <- function (beta = NULL, ...){
+    if (is.null (beta))
+      stop ("'beta' must be the current coefficient vector \n")
+    p <- length (beta)
+    if (p < 2)
+      stop ("There must be at least two regressors! \n")
+    if (p > 2){
+      h1 <- cbind (-diag (p-1), 0)
+      h2 <- cbind (0, diag (p-1))
+      mat1 <- h1 + h2
+      mat2 <- diag (p)
+      a.coefs.mat <- cbind (mat2, t (mat1))}
+    else
+      a.coefs.mat <- cbind (diag (2), c(-1,1))
+    return (a.coefs.mat)}
+
+  structure (list (penalty = "fused.lasso", lambda = lambda,
+      first.derivative = first.derivative, a.coefs = a.coefs),
+      class = "penalty")}
```

Note that the fused lasso penalty only works by definition for at least two regressors. Actually we do not need the `beta` argument in the `first.derivative()` and `a.coefs()` functions directly. We only use it to determine the dimension parameter `p`.

We conclude this section with two remarks on `penalty` objects. Note that the penalty objects only consider the $p$ regressors. Their methods do not take into consideration whether or not there is an intercept included in the model. An adjustment of the (approximated) penalty matrix $\mathbf{A}_\lambda$ to

$$\mathbf{A}_\lambda^* = \begin{bmatrix} 0 & \mathbf{0}^\top \\ \mathbf{0} & \mathbf{A}_\lambda \end{bmatrix}$$

will be done automatically by the function `get.Amat()` from the **lqa** package. This function is the conjunction between `penalty` objects and the algorithms that require $\mathbf{A}_\lambda$. From a numerical point of view a direct computation of the approximated penalty matrix via

$$\mathbf{A}_\lambda = \sum_{j=1}^{J} \frac{p'_{\lambda,j}(|\mathbf{a}_j^\top \boldsymbol{\beta}|)}{\sqrt{\left(\mathbf{a}_j^\top \boldsymbol{\beta}\right)^2 + c}} \mathbf{a}_j \mathbf{a}_j^\top \tag{8}$$

is often not efficient. As we have seen above, in some cases we can compute $\mathbf{A}_\lambda$ directly using the `getpenmat()` function. If we apply formula (8) directly, we would need to compute $J$ outer products based on the $\mathbf{a}_j$, that is we had to store $J$ matrices each of dimension $p \times p$. If the coefficients $\mathbf{a}_j$ are sparse as for the fused lasso penalty, those matrices would contain just a few non-zero components (to be concrete: e.g. four non-zero elements for the fused lasso penalty) no matter how much regressors are in the model. With increasing $p$ this becomes more inefficient, yet cumbersome. A numerically more efficient method would extract the non-zero elements of each $\mathbf{a}_j$, compute the outer products of those subvectors and record the resulting submatrices at the corresponding positions of a $(p \times p)$-dimensional working penalty matrix. If some positions overlap between the $J$ penalty terms then the recording is additive. Consequently, the required memory will be reduced and the speed of computation increases. This improving principle is used by the `get.Amat()` function.

## 3 Basic structure of the lqa package

In this section we describe the basic structure of the **lqa** package. As illustrated in Table 3, there is an elemental hierarchy of four methods (or classes of methods). As for the implementation of the penalties and their related methods we use the R concept of S3 classes. At the lowest level there is the generic function `lqa()`. Its task is to determine the class of its arguments and to use this information to select an appropriate method. Therefore they are also called dispatching methods. The function calls of the methods on the next two levels are

```
## S3 method for class 'formula':
    lqa(formula, data = list (), weights = rep (1, nobs), subset,
            na.action, start = NULL, etastart, mustart, offset, ...)
```

at level II and

```
## Default S3 method:
    lqa(x, y, family = gaussian (), penalty = NULL, method = "lqa.update2",
            weights = rep (1, nobs), start = NULL,
            etastart = NULL, mustart = NULL, offset = rep (0, nobs),
            control = lqa.control (), intercept = TRUE,
            standardize = TRUE, ...)
```

at level III.

As you can see, the S3 method for class `formula` requires less arguments than the default S3 methods at the next level. This is due to the extraction and 'translation' of information from the `formula` environment in the `lqa.formula()` function before `lqa.default()` is called. As a conclusion, for a user it is more comfortable to enter at the lowest possible level. Nevertheless, the big advantage from using S3 methods is that the user simply calls `lqa` with either one of the possible representations as the argument. The internal dispatching method will find the class of the object and apply the right method. Note that just looking at the syntax of the function call for the class `formula` is misleading for applications. As the function `lqa.formula()` is primarily meant to provide the data included in the `formula` environment, there are no input arguments necessary concerning the penalty or the fitting method. But those are to be specified for higher level methods. Therefore please use the examples at the next section as a guide to your applications to the `lqa()` function.

As you can see in Table 3 there are two main functions involved in the parameter estimation. The first is the default function (`lqa.default()`). This function can be interpreted

| Level | Method | Tasks |
|---|---|---|
| I | `lqa()` | • dispatching |
| II | `lqa.formula()` | • extract the data from the `formula` environment<br>• call the default method |
| III | `lqa.default()` | • check for the exponential family and link function in the `family` argument<br>• check the `penalty` argument<br>• check for existence of the fitting method<br>• check for model consistency (e.g. a column of ones must be included in the `x` argument if an intercept is present in the model)<br>• standardizes the data<br>• call the fitting method<br>• transform the estimated coefficients back<br>• compute important statistics (deviance, AIC, BIC, residuals, estimated predictors $\hat{\boldsymbol{\eta}}$ and responses $\hat{\boldsymbol{\mu}}$, etc.) |
| IV | fitting method | • computation of estimates |

Table 3: Implemented basic methods and their tasks in the **lqa** package.

as a principal or advisor of the estimation process. Its task is to check whether the input arguments are in order or not and to standardize the data if required. Afterwards it calls the fitting method and transforms the returned estimated coefficients back to the original scales of the regressors. Finally it computes some important statistics such as the deviance, the information criteria AIC and BIC, the estimated predictors $\hat{\boldsymbol{\eta}}$ and responses $\hat{\boldsymbol{\mu}}$. At last the default function assigns the object to be returned to the `lqa` class. Furthermore it inherits the affiliation to the `glm` and `lm` classes. Thereby we could use generic functions of those classes such as `coef()` to extract the estimated coefficients from an `lqa` object.

The other function or more precicely the other set of functions mainly involved in parameter estimation are the fitting methods. They can be interpreted as the workhorses or the agents of the package. Their task is to compute the estimates and some other statistics such as the trace of the hat matrix, see below. Up to now the following fitting methods are implemented:

- `lqa.update2` to compute the LQA algorithm. The '2' at the end of its name has just some historical reasons.

- `ForwardBoost` to compute the ForwardBoost algorithm,
- `GBlockBoost` to compute the GBlockBoost algorithm,
- `GBlockBoost` combined with the argument `componentwise = TRUE` computes componentwise boosting.

If you want to implement your own fitting method then you should remind of the following aspects. The basic call of a fitting method with name `method.name` will be

```
method.name (x, y, family = NULL, penalty = NULL, intercept = TRUE,
             control, ...)
```

where the input arguments are

| | |
|---|---|
| `x` | the standardized design matrix. This will usually include a column of ones if an intercept should be included in the model. |
| `y` | the vector of observed response values. |
| `family` | a description of the error distribution and link function to be used in the model. This can be a character string naming a family function, a family function or the result of a call to a family function. |
| `penalty` | a description of the penalty to be used in the fitting procedure. |
| `intercept` | a logical object whether the model should include an intercept (this is recommended) or not. The default value is `TRUE`. |
| `control` | a list of parameters for controlling the fitting process. |
| `...` | further arguments. |

The fitting methods have access to the environment of `lqa.default()`. So in principle, you can pass additional arguments from all lower level `lqa` function calls.

To be in line with the other functions in the package the fitting method should always return a list that contains at least the following arguments

| | |
|---|---|
| `coefficients` | the vector of the standardized estimated coefficients. |
| `tr.H` | the trace of the hat matrix. |
| `Amat` | the penalty matrix from the last iteration. |
| `converged` | a logical variable. This should be `TRUE` if the algorithm indeed converged. |
| `stop.at` | the number of iterations until convergence. |
| `m.stop` | the number of iterations until AIC reaches its minimum. |

The last argument `m.stop` is only necessary for fitting algorithms where the iteration of optimal stopping is less than the number of iterations until convergence. This is especially related to boosting methods.

The R package **mboost** (Hothorn et al., 2009) provides a framework for the application of functional gradient descent algorithms (boosting) for the optimization of general loss functions. The package is especially designed to utilize componentwise least squares as base learners. An alternative to directly implement ForwardBoost and GBlockBoost in the scope of the **lqa** package has been to incorporate the infrastructure of **mboost** through a kind of interface. In this case ForwardBoost and GBlockBoost must have been implemented as new **mboost** functions such as the already existing ones `gamboost()` or `glmboost()`. See Hothorn et al. (2009) for details. Furthermore, in order to fulfill the requirements of the output arguments of fitting methods for **lqa** (as described above) we would need to write another function in this case, that e.g. computes the trace of the hat matrix. Since we could also use some source code from the P-IRLS algorithm for the implementation of the boosting methods, it was obvious not to take **mboost** into consideration here.

In principle, the **lqa** package can be used to boost penalized GLMs based on the LQA algorithm. However, in such a case it is not advisable to use just one iteration of P-IRLS since the approximated penalty matrix $\mathbf{A}_\lambda$ will then strongly depend on the control parameter $c$. Consequently, the boosting methods must be adjusted then. We will not follow this idea but leave it for further investigations.

## 4 Some Applications of the lqa package

There are three main important functions in the **lqa** package. The function `lqa()` can be used to fit a penalized GLM for an already specified tuning parameter. The function `cv.lqa()` finds an optimal tuning parameter in up to three dimensions. The `plot.lqa()` function can be applied to visualize the solution path via coefficient build-ups. For `lqa` and `cv.lqa` objects, as returned by the corresponding functions, there exist `summary()` functions that give a short and compact overview on the computed results. But now we are going into details on how to apply these functions.

### 4.1 Using the `lqa()` function

When the tuning parameter is already specified, we can use the function `lqa()` to fit a penalized GLM by the LQA algorithm or some boosting algorithms with variable selection schemes. The main working function `lqa.update2` computes the LQA updates using a Cholesky decomposition of $\mathbf{X}^\top\mathbf{W}\mathbf{X}+\mathbf{A}^*_\lambda$. With $n$ observations and $\tilde{p} = p+1$ coefficients, including $p$ regressors and one intercept, the Cholesky decomposition requires $\tilde{p}^3 + n\tilde{p}^2/2$ operations. The Cholesky decomposition is usually fast, depending on the relative size of $n$ and $\tilde{p}$, but it can be less numerically stable (Lawson and Hanson, 1974).

Due to the arguments of `lqa.formula()` and `lqa.default()`, the syntax of a call to `lqa` should look like this

```
R> obj <- lqa (formula, family, penalty, data, method, standardize, ...)
```

with input parameters

| | |
|---|---|
| `formula` | a symbolic description of the model to be fit. A typical formula has the form `response` $\sim$ `terms`, where `response` is the (numeric) response vector and `terms` is a series of terms which specifies a linear predictor for `response`. Per default an intercept is included in the model. If it should be removed then use formulae of the form `response` $\sim$ `0 + terms` or `response` $\sim$ `terms - 1`. |
| `family` | a description of the error distribution and link function to be used in the model. This can be a character string naming a family function, a family function or the result of a call to a family function. (See the R-function `family()` for details of family functions.) |
| `penalty` | a description of the penalty to be used in the fitting procedure. This must be an object of the class `penalty`. |
| `data` | an optional data frame containing the variables in the model. If not found in `data`, the variables are taken from `environment(formula)`, typically the environment from which the function `lqa` is called. |
| `method` | a character indicating the fitting method. The default value `method = "lqa.update2"` applies the LQA algorithm. |
| `standardize` | a Boolean variable, whether the regressors should be standardized (this is recommended) or not. The default value is `standardize = TRUE`. |
| `...` | further arguments passed to or from other methods. |

This call returns an object `obj` of the class `lqa` which moreover inherits the class attributes `glm` and `lm`. As a consequence, we could apply to `obj` the typical methods as for instances of the class `lm`. Some of them are modified in order to meet the special needs of **lqa** instances, see below.

For illustration, consider the following small example. We simulate a GLM

$$\mu_i = h(\eta_i), \quad i = 1, \dots, n,$$

with $n = 100$ observations and $p = 5$ covariates, where the predictor $\eta_i = \beta_0 + \mathbf{x}_i^\top \boldsymbol{\beta}$ consists of the true parameters $\beta_0 = 0$ and

$$\boldsymbol{\beta} = (1, 2, 0, 0, -1)^\top.$$

The original design matrix (not regarding the intercept) consists of five univariate standard normally distributed covariates $\mathbf{x}_{(1)}, \dots, \mathbf{x}_{(5)}$. In order to get a stochastic dependency structure, we replace $\mathbf{x}_{(2)}$ and $\mathbf{x}_{(3)}$ by $\mathbf{x}_{(1)}$ where some additional noise is applied to obtain correlations near one but not exactly equal to one. As a result, we get a simulation setting with multicollinearity where the application of shrinkage methods indeed makes sense.

Now we specify the response. The GLM will be a logit model, that is $y_i \sim B(1, p_i)$ and $E(y_i|\mathbf{x}_i) = p_i = h(\eta_i)$ with $h(\eta_i) = 1/(1 + \exp\{-\eta_i\})$. We will denote the corresponding simulated response vector as $\mathbf{y}$. The R code for this data generation is given below, where we fixed the seed of the random number generator for illustration purpose.

```
R>  n <- 100
R>  p <- 5
R>
R>  set.seed (1234)
R>  x <- matrix (rnorm (n * p), ncol = p)
R>  x[,2] <- x[,1] + rnorm (n, sd = 0.01)
R>  x[,3] <- x[,1] + rnorm (n, sd = 0.1)
R>  beta <- c (1, 2, 0, 0, -1)
R>  prob1 <- 1 / (1 + exp (drop (-x %*% beta)))
R>  y <- sapply (prob1, function (prob1) {rbinom (1, 1, prob1)})
```

We want to fit a logistic regression model with `lqa`, using the lasso penalty with tuning parameter $\lambda = 1.5$ and fitting method `lqa.update2`. Since the latter is the default value of `method` we do not need to state it explicitly. This results in

```
R>   lqa.obj <- lqa (y ~ x, family = binomial (), penalty = lasso (0.9))
R>   lqa.obj

Call:  lqa.formula(formula = y ~ x, family = binomial(),
       penalty = lasso(0.9))

Coefficients:
(Intercept)          x1          x2          x3          x4          x5
 -7.940e-01   9.202e-02   1.208e+00   7.612e-03   8.299e-07  -5.213e-03

Degrees of Freedom: 99 Total (i.e. Null);  98.3658 Residual
Null Deviance:      123.8
Residual Deviance: 71.8         AIC: 75.07
```

Some additional information can be gained by the call `summary (lqa.obj)`.

## 4.2 Basic principles of using the `cv.lqa()` function

Normally we do not know the optimal tuning parameter a priori. It is more common to apply model selection via a set of tuning parameter candidates. This will be done by cross validation. In the **lqa** package the function `cv.lqa()` is designed for this purpose. The usage of it is

```
cv.lqa(y.train, x.train, intercept = TRUE, y.vali = NULL,
            x.vali = NULL, lambda.candidates, family, penalty.family,
            standardize = TRUE, n.fold, cv.folds,
            loss.func = aic.loss, control = lqa.control(), ...)
```

with input parameters

| | |
|---|---|
| `y.train` | the vector of response training data. |
| `x.train` | the design matrix of training data. If `intercept = TRUE` then it does not matter whether a column of ones is already included in `x.train` or not. The function adjusts it if necessary. |
| `intercept` | logical. If `intercept = TRUE` then an intercept is included in the model (this is recommended). |
| `y.vali` | an additional vector of response validation data. If given the validation data are used for evaluating the loss function. |
| `x.vali` | an additional design matrix of validation data. If given the validation data are used for evaluating the loss function. If `intercept = TRUE` then it does not matter whether a column of ones is already included in `x.train` or not. The function adjusts it if necessary. |
| `lambda.candidates` | a list containing the tuning parameter candidates. The number of list elements must be correspond to the dimension of the tuning parameter. See details below. |
| `family` | identifies the exponential family of the response and the link function of the model. See the description of the R function `family()` for further details. |
| `penalty.family` | a function or character argument identifying the penalty family. See details below. |
| `standardize` | logical. If `standardize = TRUE` the data are standardized (this is recommended). |
| `n.fold` | number of folds in cross-validation. This can be omitted if a validation set is used. |
| `cv.folds` | optional list containing the observation indices used in the particular cross-validation folds. This can be omitted if a validation set is used. |
| `loss.func` | a character indicating the loss function to be used in evaluating the model performance for the tuning parameter candidates. If `loss.func = NULL` the `aic.loss()` function will be used. See details below. |
| `control` | a list of parameters for controlling the fitting process. See the documentation of `lqa.control()` for details. |

...  Further arguments.

This function can be used for evaluating model performance for different tuning parameter candidates. If you just give training data a cross validation will be applied. If you additionally provide validation data then those data will be used for measuring the performance and the training data are completely used for model fitting.

You must specify a penalty family. This can be done by giving its name as a character (e.g. `penalty.family = "lasso"`) or as a function call (e.g. `penalty.family = lasso`).

The tuning parameter candidates are given in the argument `lambda.candidates`. Usually one should a priori generate a sequence of equidistant points and then use this as exponent to Euler's number, such as

```
R> lambdaseq <- exp (seq (-4, 4, length = 11))
R> lambdaseq
 [1]  0.0183156  0.0407622  0.0907179  0.2018965  0.4493289  1.0000000
 [7]  2.2255409  4.9530324 11.0231763 24.5325302 54.5981500
```

Note that `lambda.candidates` must be a list in order to cope with different numbers of candidates, e.g.

```
lambda.candidates = list (lambdaseq).
```

For evaluation you must specify a loss function. The default value is `aic.loss()` e.g. the AIC will be used to find an optimal tuning parameter. Other already implemented loss functions are `bic.loss()`, `gcv.loss()`, `squared.loss()` (quadratic loss function), `dev.loss()` (deviance as loss function). In the following, we explain the basic principles of the application of loss functions in the **lqa** package.

Before we start we introduce some notation to describe the cross-validation procedure in general. Let $K$ denote the number of cross-validation folds and $\kappa : \{1, \ldots, n\} \to \{1, \ldots, K\}$ is an indexing function that indicates the partition to which the $i$-th observation ($i = 1, \ldots, n$) is allocated. In the function `cv.lqa()` the number of folds $K$ is maintained in the argument `n.fold`. Let $\hat{\mathbf{b}}_\lambda^{-k} = (\hat{\beta}_\lambda^{-k}, (\hat{\boldsymbol{\beta}}_\lambda^{-k})^\top)^\top$, $k \in \{1, \ldots, K\}$ denote the estimate during the cross-validation for a given value of tuning parameter $\boldsymbol{\lambda}$. The index $-k$ indicates that $\hat{\mathbf{b}}_\lambda^{-k}$ has been estimated based on the cross-validation index subset $\{i \in \{1, \ldots, n\} : \kappa(i) \neq k\}$, that is all of the training data except the subset that belongs to the $k$-th partition. Let $\hat{\mu}_i^\lambda = h(\hat{\beta}_\lambda^{-k} + \mathbf{x}_i^\top \hat{\boldsymbol{\beta}}_\lambda^{-k})$ denote the estimated response of the $i$-th observation. For the deviance we write

$$\mathrm{Dev}_\lambda^k = 2 \sum_{i:\kappa(i)=k} \left\{ \ell_i(\hat{\mathbf{b}}_{\max})) - \ell_i(\hat{\mathbf{b}}_\lambda^{-k}) \right\}, \quad k = 1, \ldots, K$$

and denote $\mathbf{H}_\lambda^{-k}$ as the hat matrix corresponding to $\hat{\mathbf{b}}_\lambda^{-k}$. This deviance and the hat matrix are important ingredients for most of the loss functions used to evaluate the model with tuning parameter $\boldsymbol{\lambda}$. Using a loss function $L(y_i, \hat{\mu}_i^{\lambda})$ the cross-validation function is

$$\mathrm{CV}(\boldsymbol{\lambda}) = \frac{1}{K} \sum_{k=1}^{K} \sum_{i:\kappa(i)=k} L(y_i, \hat{\mu}_i^{\lambda}).$$

This definition varies from the typical ones such as (7.49) in Hastie et al. (2009), p. 242. But from a computational point of view it is more well-arranged. This will become clear, hopefully, in the examples below. The loss functions already implemented in the **lqa** package are summarized in Table 4.

| Criterion | Formula | Function name |
|---|---|---|
| AIC | $\mathrm{AIC}(\boldsymbol{\lambda}) = \mathrm{Dev}_{\lambda}^{k} + 2\,\mathrm{tr}(\mathbf{H}_{\lambda}^{-k})$ | `aic.loss()` |
| BIC | $\mathrm{BIC}(\boldsymbol{\lambda}) = \mathrm{Dev}_{\lambda}^{k} + \log(n)\,\mathrm{tr}(\mathbf{H}_{\lambda}^{-k})$ | `bic.loss()` |
| GCV | $\mathrm{GCV}(\boldsymbol{\lambda}) = n\,\mathrm{Dev}_{\lambda}^{k} / \{n - \mathrm{tr}(\mathbf{H}_{\lambda}^{-k})\}^2$ | `gcv.loss()` |
| squared loss | $\mathrm{SL}(\boldsymbol{\lambda}) = \sum_{i=1}^{n} (y_i - \hat{\mu}_i^{\lambda})^2$ | `squared.loss()` |
| deviance loss | $\mathrm{DL}(\boldsymbol{\lambda}) = \mathrm{Dev}_{\lambda}^{k}$ | `dev.loss()` |

Table 4: Implemented loss functions.

Note, if `loss.func = gcv.loss` is chosen then this does not match with given validation data. This is due to the construction of the GCV as an approximation to leave-one-out cross-validation. If there are nonetheless given some validation data then `cv.lqa()` will replace them by the training data arguments `y.train` and `x.train`.

However, the estimate $\hat{\mathbf{b}}_{\lambda}^{-k}$ and the hat matrix $\mathbf{H}_{\lambda}^{-k}$ are part of the output of `lqa()`. But to compute terms like $\mathrm{Dev}_{\lambda}^{k}$ and $\hat{\mu}_i^{\lambda}$ the **lqa** package provides the function `predict.lqa()`. This function is primarily used internally, but you can use it directly as well. See the **lqa** manual (Ulbricht, 2010a) for documentation of it.

The function `predict.lqa()` returns an object `pred.obj` of the class `pred.lqa`. This object is the input argument for all loss functions in **lqa**. Therefore it is worth to look at it a little bit more closer. The object `pred.obj` contains the following elements

| | |
|---|---|
| `deviance` | the deviance based on the new observations. This element provides $\mathrm{Dev}_{\lambda}^{k}$. |
| `tr.H` | the trace of the hat matrix of the design matrix used to fit the model, e.g. $\mathbf{H}_{\lambda}^{-k}$. This is just an extraction from the `lqa.obj` object that is used as input argument in `predict.lqa()`. |
| `n.newobs` | the number of new observations. |
| `eta.new` | the estimated new predictors. |
| `mu.new` | the estimated new responses, e.g. a vector containing $\hat{\mu}_i^{\lambda}$. |
| `lqa.obj` | the `lqa.obj` argument of the `predict.lqa()` function. This is the return object of the `lqa` function, so that you can in principle access all of its elements. |
| `new.y` | the vector of new observations. |

Consequently you can use all of those arguments in your loss function. Exemplarily, we show the source code of the `gcv.loss` function:

```
R> gcv.loss <- function (pred.obj) {
+    dev <- pred.obj$deviance
+    tr.H <- pred.obj$tr.H
+    nobs <- pred.obj$n.newobs
+    nobs * dev / ((nobs - tr.H)^2)}
```

The output of a loss function must be a non-negative scalar.

## 4.3 Examples for using the `cv.lqa()` function

We want to continue the example from Section 4.1. We still want to apply the lasso penalty but are now looking for an optimal tuning parameter. Since we did not have simulated a validation data set we use a five-fold cross-validation based on the deviance. Consequently, optimality of $\boldsymbol{\lambda}$ means minimizing the cross-validation score. To keep the output small we restrict to only five tuning parameter candidates. Then the function call and its result are:

```
R> cv.obj <- cv.lqa (y, x, family = binomial (), penalty.family = lasso,
    lambda.candidates = list (c (0.001, 0.05, 1, 5, 10)), n.fold = 5,
    loss.func = "dev.loss")
R> cv.obj

cv.lqa(y.train = y, x.train = x, lambda.candidates = list(c(0.001,
    0.05, 0.5, 1, 5)), family = binomial(), penalty.family = lasso,
    n.fold = 5, loss.func = "dev.loss")

loss function:  dev.loss
validation data set used:  FALSE
number of folds =  5

loss matrix:
                  fold 1   fold 2    fold 3    fold 4   fold 5     mean
lambda1 = 0.001 20.37285 29.55333  9.305895  8.726739 18.32540 17.25684
lambda1 = 0.05  18.41687 27.10695  7.914941  9.013015 16.03634 15.69762
lambda1 = 0.5   17.75243 19.67935  8.849785 13.260267 15.62594 15.03356
lambda1 = 1     19.31417 17.05229 10.995958 17.149802 17.19262 16.34097
lambda1 = 5     26.12830 20.94446 23.028678 30.310342 26.12799 25.30795

lambda.opt =  0.5


Call:  lqa.default(x = x.train, y = y.train, family = family, penalty =
  penalty, control = control, intercept = intercept, standardize =
  standardize)

Coefficients:
[1] -8.560e-01  2.812e-01  1.404e+00  5.820e-02 -1.355e-07 -2.474e-01

Degrees of Freedom: 99 Total (i.e. Null);  97.8255 Residual
Null Deviance:      123.8
Residual Deviance: 62.05        AIC: 66.4
```

The `print()` function for `cv.obj` shows the function call first. Afterwards information on the applied loss function, the existence of a validation data set, and the number of cross-validation folds are given. Thereafter the loss matrix is printed where each row corresponds to one tuning parameter candidate and the first columns correspond to the `n.fold` folds. The last column shows the values of CV($\boldsymbol{\lambda}$). As you can see, in our example $\lambda = 0.5$ delivers the smallest value and hence is regarded as optimal tuning parameter. Note that the difference to CV(0.05) is only small. Afterwards the so called `best.obj`, that is the GLM with the chosen penalty family and the optimal tuning parameter, is printed. This includes its (unstandardized) estimated coefficients and some model summary statistics.

The `cv.lqa()` function can deal with cross-validation for up to three-dimensional tuning parameters. In this case printing of the loss matrix (which then in fact is a loss array) becomes quite striking for the user. Therefore a mean array is provided in those cases which just gives the CV scores and hence lowers the dimension of the loss array about one. To illustrate this we want to do cross-validation for our simulated data and now apply the fused lasso penalty. For $\lambda_2$ we consider just three candidates. The function call is

```
R> cv.obj2 <- cv.lqa (y, x, family = binomial (), penalty.family =
+    fused.lasso, lambda.candidates = list (c (0.001, 0.05, 0.5, 1, 5),
+    c (0.001, 0.01, 0.5)), n.fold = 5, loss.func = "dev.loss")
R> cv.obj2
```

We just extract the mean array and the optimal tuning parameter in the following.

```
mean array:
                 lambda2 = 0.001 lambda2 = 0.01 lambda2 = 0.5
  lambda1 = 0.001       14.01300       13.55434      13.21141
  lambda1 = 0.05        13.14641       12.93533      13.24855
  lambda1 = 0.5         13.58486       13.53301      14.41157
  lambda1 = 1           15.48474       15.47385      16.22203
  lambda1 = 5           24.94260       24.94256      24.94281

lambda.opt =  0.05 0.01
```

The mean array displays the values of CV($\boldsymbol{\lambda}$) for all tuning parameter candidates combinations. As you can see CV($\lambda_1 = 0.05, \lambda_2 = 0.01$) delivers the minimum. By the way, at this position it should become clear why it is necessary to name your tuning parameters in objects of the `penalty` class. Otherwise the labeling of rows and columns in the mean array could be misleading.

## 4.4 Examples for using the `plot.lqa()` function

In many situations it is preferable to look at the coefficient build-ups of a penalization method. For those cases you can use the `plot.lqa()` function. Its usage is

```
plot.lqa(y, x, family, penalty.family, intercept = TRUE,
         standardize = TRUE, lambdaseq = NULL, offset.values = NULL,
         show.standardized = FALSE, add.MLE = TRUE,
         control = lqa.control(), ...)
```

where the input parameters are

| | |
|---|---|
| `y` | the vector of observed responses. |
| `x` | the design matrix. If `intercept = TRUE` then it does not matter whether a column of ones is already included in `x.train` or not. The function adjusts it if necessary. |
| `family` | identifies the exponential family of the response and the link function of the model. See the description of the R function `family` for further details. |
| `penalty.family` | a function or character argument identifying the penalty family. |
| `intercept` | logical. If `intercept = TRUE` then an intercept is included in the model (this is recommended). |
| `standardize` | logical. If `standardize = TRUE` the data are standardized (this is recommended). |
| `lambdaseq` | a sequence of tuning parameter candidates for the dimension you want to plot. |
| `offset.values` | a vector of the same dimension as your tuning parameter. At the position of the dimension you want to plot there must be entry `NA`. The other positions should be filled with given (and fixed) tuning parameter values, as e.g. returned optimized values from `cv.lqa()`. See examples below. |
| `show.standardized` | logical. If `show.standardize = TRUE` the standardized coefficients are plotted, otherwise the unstandardized coefficients are plotted. |
| `add.MLE` | logical. If `add.MLE = TRUE` the unrestricted MLE is also plotted. Note this only works for $n > p$ settings. Otherwise this argument is set to `FALSE` automatically. |

| | |
|---|---|
| `control` | list of control parameters as returned by `lqa.control()`. See the **lqa** manual (Ulbricht, 2010a) for details. |
| `...` | further arguments |

This function plots the coefficient build-ups for a given dimension of your tuning parameter(s). The argument `lambdaseq` can be omitted. In this case a default sequence

```
R>  lambdaseq <- exp (seq (-10, 6, length = 60))
```

is used. If your penalty consists of more than one tuning parameter you must identify the relevant dimension to plot using `offset.values` where you state the fixed values for the other tuning parameters.

We want to plot the coefficient build-up for our simulated example with fused lasso penalty. Using the results from the last section, we set $\lambda_2 = 0.01$. For this value held fixed the coefficients cannot converge to the MLE when $\lambda_1 \downarrow 0$. Therefore we use `add.MLE = FALSE`. The resulting function call is

```
R> plot.lqa (y, x, family = binomial (), penalty.family = fused.lasso,
+    offset.values = c (NA, 0.01), add.MLE = FALSE)
```

The corresponding plot is given in Figure 1. Note that the grouping effect does not cover $x_3$. This is due to the small weight $\lambda_2 = 0.01$ on the second penalty term that is responsible for the fusion of correlated regressors. Note that the grouping effect among $x_1$ and $x_2$ is kept all the time. Furthermore, the relevance of the regressors is recognized as $x_4$ is selected to join the active set at last.

## References

Bondell, H. D. and B. J. Reich (2008). Simultaneous regression shrinkage, variable selection and clustering of predictors with oscar. *Biometrics 64*, 115–123.

Chambers, J. (1998). *Programming with Data. A Guide to the S Language.* New York: Springer.

Fahrmeir, L., T. Kneib, and S. Lang (2009). *Regression - Modelle, Methoden und Anwendungen* (2nd ed.). Berlin: Springer.

Fahrmeir, L. and G. Tutz (2001). *Multivariate Statistical Modelling based on Generalized Linear Models* (2nd ed.). New York: Springer.

Frank, I. E. and J. H. Friedman (1993). A statistical view of some chemometrics regression tools (with discussion). *Technometrics 35*, 109–148.

Hastie, T., R. Tibshirani, and J. H. Friedman (2009). *The Elements of Statistical Learning* (2nd ed.). New York: Springer.

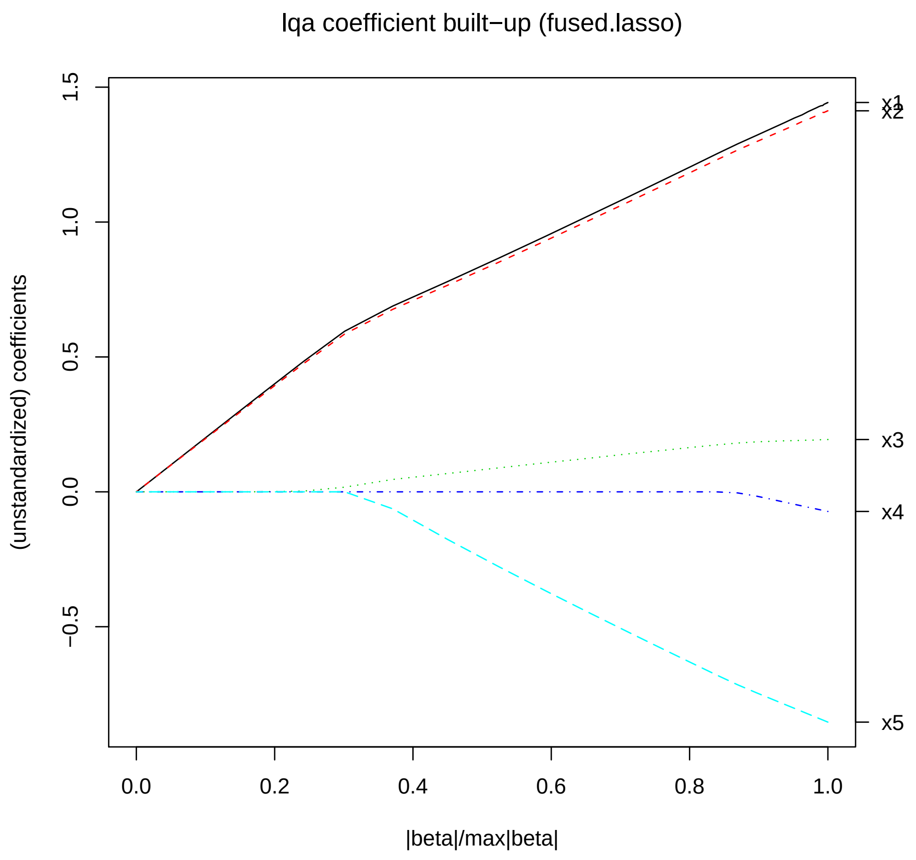

Figure 1: The resulting plot as returned from the function `plot.lqa()` when it is applied to the simulated example and $\lambda_2 = 0.01$ is held fixed.

Hothorn, T., P. Bühlmann, T. Kneib, M. Schmid, and B. Hofner (2009). *mboost: Model-Based Boosting.* R package version 1.0-7.

Lawson, C. and R. Hanson (1974). *Solving Least Squares Problems.* Englewood Cliffs, NJ: Prentice-Hall.

Leisch, F. (2008). Creating R packages: A tutorial. In P. Brito (Ed.), *Compstat 2008—Proceedings in Computational Statistics.* Physica Verlag, Heidelberg, Germany.

McCullagh, P. and J. A. Nelder (1989). *Generalized Linear Models* (2nd ed.). New York: Chapman & Hall.

R Development Core Team (2009). *R: A Language and Environment for Statistical Computing.* Vienna, Austria: R Foundation for Statistical Computing. ISBN 3-900051-07-0.

Tibshirani, R. (1996). Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society B 58*, 267–288.

Tibshirani, R., M. Saunders, S. Rosset, J. Zhu, and K. Knight (2005). Sparsity and smoothness via the fused lasso. *Journal of the Royal Statistical Society B 67*, 91–108.

Tutz, G. and J. Ulbricht (2009). Penalized regression with correlation based penalty. *Statistics and Computing 19*, 239–253.

Ulbricht, J. (2010a). *lqa: Local Quadratic Approximation.* R package version 1.0-2.

Ulbricht, J. (2010b). *Variable Selection in Generalized Linear Models.* Ph. D. thesis, LMU Munich.

Venables, W. and B. Ripley (2000). *S Programming.* New York: Springer.

Wood, S. N. (2006). *Generalized Additive Models: An Introduction with R.* Boca Raton: Chapman & Hall/CRC.