Chapter 10
Variable Selection

Variable selection is an important topic in many types of multivariate modelling: the choice which variables to take into account to a large degree determines the result. This is true for every single technique discussed in this book, be it PCA, clustering methods, classification methods, or regression. In the unsupervised approaches, uninformative variables can obscure the "real" picture, and distances between objects can become meaningless. In the supervised cases (both classification and regression), there is the danger of chance correlations with dependent variables, leading to models with low predictive power. This danger is all the more real given the very low sample-to-variable ratios of many current data sets. The aim of variable selection then is to reduce the independent variables to those that contain relevant information, and thereby to improve statistical modelling. This should be seen both in terms of predictive performance (by decreasing the number of chance correlations) and in interpretability—often, models can tell us something about the system under study, and small sets of coefficients are usually easier to interpret than large sets.

In some cases, one is able to decrease the number of variables significantly by utilizing domain knowledge. A classical application is peak-picking in spectral data. In metabolomics, for instance, where biological fluids are analyzed by, e.g., NMR spectroscopy, one can typically quantify hundreds of metabolites. The number of metabolites is usually orders of magnitude smaller than the number of variables (ppm values) that have been measured; moreover, the metabolite concentrations lend themselves for immediate interpretation, which is not the case for the raw NMR spectra. A similar idea can be found in the field of proteomics, where mass spectrometry is used to find the presence or absence of proteins, based on the presence or absence of certain peptides. Quantification is more problematic here, so typically one obtains a list of proteins that have been found, including the number of fragments that have been used in the identification. When this step is possible it is nearly always good to do so. The only danger is to find what is already known—in many cases, data bases are used in the interpretation of the complex spectra: an unexpected compound, or a compound that is not in the data base but is present in the sample, is likely to be missed. Moreover, incorrect assignments present additional difficulties. Even so,

© Springer-Verlag GmbH Germany, part of Springer Nature 2020

R. Wehrens, *Chemometrics with R*, Use R!,
https://doi.org/10.1007/978-3-662-62027-4_10

the list of metabolites or proteins may be too long for reliable modelling or useful interpretation, and one is interested in further reduction of the data.

Very often, this variable selection is achieved by looking at the coefficients themselves: the large ones are retained, and variables with smaller coefficients are removed. The model is then refitted with the smaller set, and this process may continue until the desired number of variables has been reached. Unfortunately, as shown in Sect. 8.1.1, model coefficients can have a huge variance when correlation is high, a situation that is the rule rather than the exception in the natural sciences nowadays. As a result, coefficient size is not always a good indicator of importance. A more sophisticated approach is the one we have seen in Random Forests, where the decrease in model quality upon permutation of the values in one variable is taken as an importance measure. Especially for systems with not too many variables, however, tests for coefficient significance remain popular.

An alternative way of tackling variable selection is to use modelling techniques that explicitly force as many coefficients as possible to be zero: all these are apparently not important for the model and can be removed without changing the fitted values or the predictions. It can be shown that a ridge-regression type of approach with a penalty on the size of the coefficients has this effect, if the penalty is suitably chosen (Hastie et al. 2001)—a whole class of methods has descended from this principle, starting with the lasso (Tibshirani 1996).

One could say that the only reliable way of assessing the modelling power of a smaller set is to try it out—and if the result is disappointing, try out a different subset of variables. Given a suitable error estimate, one can employ optimization algorithms to find the subset that gives maximal modelling power. Two strategies can be followed: one is to fix the size of the subset, often dictated by practical considerations, and find the set that gives the best performance; the other is to impose some penalty on including extra variables and let the optimization algorithm determine the eventual size. In small problems it is possible, using clever algorithms, to find the globally optimal solution; in larger problems it very quickly becomes impossible to assess all possible solutions, and one is forced to accept that the global optimum may be missed.

10.1 Coefficient Significance

Testing whether coefficient sizes are significantly different from zero is especially useful in cases where the number of parameters is modest, less than fifty or so. Even if it does not always lead to the optimal subset, it can help to eliminate large numbers of variables that do not contribute to the predictive abilities of the model. Since this is a univariate approach—every variable is tested individually—the usual caveats about correlation apply. Rather than concentrating on the size and variability of individual coefficients, one can compare nested models with and without a particular variable. If the error decreases significantly upon inclusion of that variable, it can be said to be relevant. This is the basis of many stepwise approaches, especially in regression.

10.1.1 Confidence Intervals for Individual Coefficients

Let's use the wine data as an example, and predict class labels from the thirteen measured variables. We can assess the confidence intervals for the model quite easily, formulating the problem in a regression sense. For each of the three classes a regression vector is obtained. The coefficients for Grignolino, third class, can be obtained as follows:

```
> X <- wines[wines.odd, ]
> C <- classvec2classmat(vintages[wines.odd])
> wines.lm <- lm(C ~ X)
> wines.lm.summ <- summary(wines.lm)
> wines.lm.summ[[3]]
Call:
lm(formula = Grignolino ~ X)
Residuals:
    Min      1Q  Median      3Q     Max
-0.4657 -0.1387  0.0022  0.1326  0.4210

Coefficients:
                   Estimate Std. Error t value Pr(>|t|)
(Intercept)         2.77235    0.63633    4.36  4.1e-05 ***
Xalcohol           -0.12466    0.04918   -2.53   0.0133 *
Xmalic acid        -0.06631    0.02628   -2.52   0.0138 *
Xash               -0.56351    0.12824   -4.39  3.6e-05 ***
Xash alkalinity     0.03227    0.00975    3.31   0.0014 **
Xmagnesium          0.00118    0.00173    0.68   0.4992
Xtot. phenols      -0.00434    0.07787   -0.06   0.9558
Xflavonoids         0.12497    0.05547    2.25   0.0272 *
Xnon-flav. phenols  0.36091    0.23337    1.55   0.1262
Xproanth            0.09320    0.05808    1.60   0.1128
Xcol. int.         -0.04748    0.01661   -2.86   0.0055 **
Xcol. hue           0.18276    0.16723    1.09   0.2779
XOD ratio           0.00589    0.06306    0.09   0.9258
Xproline           -0.00064    0.00012   -5.33  1.0e-06 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.209 on 75 degrees of freedom
Multiple R-squared:  0.847, Adjusted R-squared:  0.82
F-statistic: 31.9 on 13 and 75 DF,  p-value: <2e-16
```

The column with the stars in the output allows us to easily spot coefficients that are significant at a certain level. To get a summary of all variables that have p values smaller than, say, 0.1 for each of the three classes, we can issue:

```
> sapply(wines.lm.summ,
+        function(x) which(x$coefficients[, 4] < .1))
$`Response Barbera`
       Xmalic acid                Xash        Xflavonoids
                 3                   4                  8
Xnon-flav. phenols          Xcol. int.          XOD ratio
                 9                  11                 13

$`Response Barolo`
    (Intercept)        Xalcohol             Xash Xash alkalinity
              1               2                4               5
    Xflavonoids        Xproanth        XOD ratio       Xproline
              8              10               13              14

$`Response Grignolino`
    (Intercept)        Xalcohol     Xmalic acid            Xash
              1               2               3               4
Xash alkalinity     Xflavonoids      Xcol. int.       Xproline
              5               8              11              14
```

Variables `ash` and `flavonoids` occur as significant for all three cultivars; six others (not counting the intercept, of course) for two out of three cultivars.

In cases where no confidence intervals can be calculated analytically, such as in PCR or PLS, we can, e.g., use bootstrap confidence intervals. For the gasoline data, modelled with PCR using four latent variables, we have calculated bootstrap confidence intervals in Sect. 9.6.2. The percentile intervals, shown in Fig. 9.6, already indicated that most regression coefficients are significantly different from zero. How does that look for the (better) BCα confidence intervals? Let's find out:

```
> gas.BCACI <-
+   t(sapply(1:ncol(gasoline$NIR),
+            function(i, x) {
+              boot.ci(x, index = i, type = "bca")$bca[, 4:5]},
+            gas.pcr.bootCI))
```

A plot of the regression coefficients with these 95% confidence intervals (Fig. 10.1) immediately shows which variables are significantly different from zero:

```
> BCAcoef <- gas.pcr.bootCI$t0
> signif <- gas.BCACI[, 1] > 0 | gas.BCACI[, 2] < 0
> BCAcoef[!signif] <- NA
```

```
> matplot(wavelengths, gas.BCACI, type = "n",
+         xlab = "Wavelength (nm)",
+         ylab = "Regression coefficient",
+         main = "Gasoline data: PCR (4 PCs)")
> abline(h = 0, col = "gray")
> polygon(c(wavelengths, rev(wavelengths)),
+         c(gas.BCACI[, 1], rev(gas.BCACI[, 2])),
+         col = "pink", border = NA)
> lines(wavelengths, BCAcoef, lwd = 2)
```

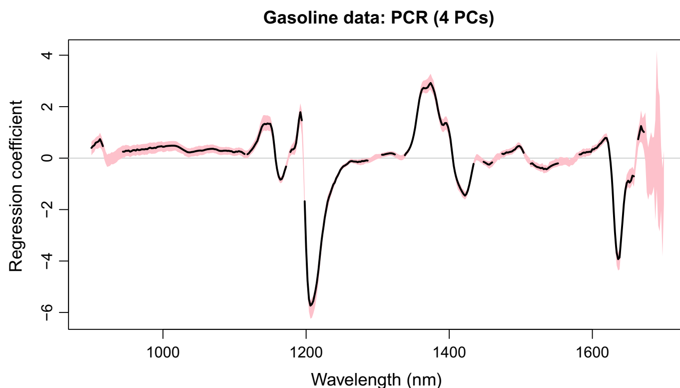

Fig. 10.1 Significance of regression coefficients for PCR using four PCs on the gasoline data; coefficients whose 95% confidence interval (calculated with the BCα bootstrap and indicated in pink) includes zero are not shown

Re-fitting the model after keeping only the 325 wavelengths leads to

```
> smallmod <- pcr(octane ~ NIR[, signif], data = gasoline,
+                 ncomp = 4, validation = "LOO")
> RMSEP(smallmod, intercept = FALSE, estimate = "CV")
1 comps  2 comps  3 comps  4 comps
 1.4342   1.4720   0.2756   0.2497
```

The error estimate is lower even than global minimum (at seven PCs) with the full data set containing 401 wavelengths. Here, one could also consider going for the three-component model which sacrifices very little in terms of RMSEP (it is still better than the seven-component model seen earlier) and has, well, one component fewer. After variable selection, refitting often leads to more parsimonious models in terms of the number of components needed. Even if the predictions are not (much) better, the improved interpretability is often seen as reason enough to consider variable selection.

Although this kind of procedure has been proposed in the literature several times, e.g., in Wehrens and van der Linden (1997), it is essentially incorrect. For the spectrum-like data, the correlations between the wavelengths are so large that the confidence intervals of individual coefficients are not particularly useful to determine which variables are significant—both errors of the first (false positives) and second kind (false negatives) are possible. Taking into account correlations and calculating so-called *Scheffé* intervals (Efron and Hastie 2016) often leads to intervals so wide that they have no practical relevance. The confidence intervals described above, for individual coefficients, at least give some idea of where important information is located.

10.1.2 Tests Based on Overall Error Contributions

In regression problems for data sets with not too many variables, the standard approach is stepwise variable selection. This can be performed in two directions: either one starts with a model containing all possible variables and iteratively discards variables that contribute least. This is called *backward selection*. The other option, *forward selection*, is to start with an "empty" model, i.e., prediction with the mean of the independent variable, and to keep on adding variables until the contribution is no longer significant.

As a criterion for inclusion, values like AIC, BIC or C_p can be employed—these take into account both the improvement in the fit as well as a penalty for having more variables in the model. The default for the R functions `add1` and `drop1` is to use the AIC. Let us consider the regression form of LDA for the wine data, leaving out the Barolo class for the moment:

```
> twowines.df <- data.frame(vintage = twovintages, twowines)
> twowines.lm0 <- lm(as.integer(vintage) ~ 1, data = twowines.df)
> add1(twowines.lm0, scope = names(twowines.df)[-1])
Single term additions

Model:
as.integer(vintage) ~ 1
                 Df Sum of Sq  RSS  AIC
<none>                        28.6 -168
alcohol           1     11.34 17.3 -226
malic.acid        1      8.75 19.9 -209
ash               1      3.15 25.5 -179
ash.alkalinity    1      1.07 27.6 -170
magnesium         1      0.72 27.9 -168
tot..phenols      1      7.57 21.1 -202
flavonoids        1     15.87 12.8 -262
non.flav..phenols 1      2.88 25.8 -178
proanth           1      4.69 23.9 -187
col..int.         1     18.07 10.6 -284
col..hue          1     15.27 13.4 -256
OD.ratio          1     17.94 10.7 -283
proline           1      3.70 24.9 -182
```

The dependent variable should be numeric, so in the first argument of the `lm` function, the formula, we convert the vintages to class numbers first. According to this model, the first variable to enter should be `col..int`—this gives the largest effect in AIC. Since we are comparing equal-sized models, this also implies that the residual sum-of-squares of the model with only an intercept and `col..int` is the smallest.

Conversely, when starting with the full model, the `drop1` function would lead to elimination of the term that contributes least:

```
> twowines.lmfull <- lm(as.integer(vintage) ~ ., data = twowines.df)
> drop1(twowines.lmfull)
Single term deletions

Model:
as.integer(vintage) ~ alcohol + malic.acid + ash + ash.alkalinity +
    magnesium + tot..phenols + flavonoids + non.flav..phenols +
    proanth + col..int. + col..hue + OD.ratio + proline
                  Df Sum of Sq  RSS  AIC
<none>                         3.65 -387
alcohol            1     0.026 3.68 -388
malic.acid         1     0.331 3.98 -378
ash                1     0.127 3.78 -384
ash.alkalinity     1     0.015 3.67 -388
magnesium          1     0.000 3.65 -389
tot..phenols       1     0.098 3.75 -385
flavonoids         1     0.821 4.47 -364
non.flav..phenols  1     0.166 3.82 -383
proanth            1     0.028 3.68 -388
col..int.          1     0.960 4.61 -361
col..hue           1     0.162 3.81 -383
OD.ratio           1     0.254 3.91 -381
proline            1     0.005 3.66 -388
```

In this case, `magnesium` is the variable with the largest negative AIC value, and this is the first one to be removed.

Concentrating solely on forward or backward selection will in practice often lead to sub-optimal solutions: the order in which the variables are eliminated or included is of great importance and the chance of ending up in a local optimum is very real. Therefore, forward and backward steps are often alternated. This is the procedure implemented in the `step` function:

```
> step(twowines.lmfull, trace = 0)

Call:
lm(formula = as.integer(vintage) ~ malic.acid + ash + tot..phenols +
    flavonoids + non.flav..phenols + col..int. + col..hue + OD.ratio,
    data = twowines.df)

Coefficients:
      (Intercept)         malic.acid                ash
           1.7220            -0.0571            -0.2359
     tot..phenols         flavonoids  non.flav..phenols
          -0.0833             0.2415             0.3821
        col..int.           col..hue           OD.ratio
          -0.0647             0.2236             0.1348
```

From the thirteen original variables, only eight remain.

Several other functions can be used for the same purpose: the **MASS** package contains functions `stepAIC`, `addterm` and `dropterm` which allows more

model classes to be considered. Package **leaps** contains function `regsubsets`[1] which is guaranteed to find the best subset, based on the branch-and-bounds algorithm. Another package implementing this algorithm is **subselect**, with the function `eleaps`.

The branch-and-bounds algorithm was first proposed in 1960 in the area of linear programming (Land and Doig 1960), and was introduced in statistics by Furnival and Wilson (1974). The title of the latter paper has led to the name of the R-package. This particular algorithm manages to avoid many regions in the search space that can be shown to be less good than the current solution, and thus is able to tackle larger problems than would have been feasible using an exhaustive search. Application of the `regsubsets` function leads to the same set of selected variables (now we can provide a factor as the dependent variable):

```
> twowines.leaps <- regsubsets(vintage ~ ., data = twowines.df)
> twowines.leaps.sum <- summary(twowines.leaps)
> names(which(twowines.leaps.sum$which[8, ]))
[1] "(Intercept)"        "malic.acid"         "ash"
[4] "tot..phenols"       "flavonoids"         "non.flav..phenols"
[7] "col..int."          "col..hue"           "OD.ratio"
```

In some special cases, approximate distributions of model coefficients can be derived. For two-class linear discriminant analysis, a convenient test statistic is given by Mardia et al. (1979):

$$F = \frac{a_i^2(m - p + 1)c^2}{t_i m(m + c^2)D^2} \tag{10.1}$$

with $m = n_1 + n_2 - 2$, n_1 and n_2 signifying group sizes, p the number of variables, $c^2 = n_1 n_2/(n_1 + n_2)$, and D^2 is the Mahalanobis distance between the class centers, based on all variables. The estimated coefficient in the discriminant function is a_i, and t_i is the i-th diagonal element in the inverse of the total variance matrix $\boldsymbol{T}$, given by

$$\boldsymbol{T} = \boldsymbol{W} + \boldsymbol{B} \tag{10.2}$$

This statistic has an F-distribution with 1 and $m - p + 1$ degrees of freedom.

Let us see what that gives for the wine data without the Barolo samples. We can re-use the code in Sect. 7.1.3, now using all thirteen variables to calculate the elements for the test statistic:

[1] It also contains the function `leaps` for compatibility reasons; `regsubsets` is the preferred function.

```
> Tii <- solve(BSS + WSS)
> Ddist <- mahalanobis(colMeans(wines.groups[[1]]),
+                      colMeans(wines.groups[[2]]),
+                      wines.pcov12)
> m <- sum(sapply(wines.groups, nrow)) - 2
> p <- ncol(wines)
> c <- prod(sapply(wines.groups, nrow)) /
+   sum(sapply(wines.groups, nrow))
> Fcal <- (MLLDA^2 / diag(Tii)) *
+   (m - p + 1) * c^2 / (m * (m + c^2 * Ddist))
> which(Fcal > qf(.95, 1, m-p+1))
      malic.acid              ash       flavonoids
               2                3                7
non.flav..phenols         col..int.          col..hue
               8               10               11
        OD.ratio
              12
```

Using this method, seven variables are shown to be contributing to the separation between Grignolino and Barbera wines on the $\alpha = 0.05$ level. The only variable missing, when compared to the earlier selected set of eight, is `tot..phenols`, which has a p-value of 0.08.

10.2 Explicit Coefficient Penalization

In the chapter on multivariate regression we already saw that several methods use the concept of shrinkage to reduce the variance of the regression coefficients, at the cost of bias. Ridge regression achieves this by explicit coefficient penalization, as shown in Eq. 8.22. Although it forces the coefficients to be closer to zero, the values almost never will be exactly zero. If that would be the case, the method would be performing variable selection: those variables with zero values for the regression coefficients can safely be removed from the data.

Interestingly enough, one can obtain the desired behavior by replacing the quadratic penalty in Eq. 8.22 by an absolute-value penalty:

$$\underset{\boldsymbol{B}}{\operatorname{argmax}} \, (\boldsymbol{Y} - \boldsymbol{X}\boldsymbol{B})^2 + \lambda|\boldsymbol{B}| \tag{10.3}$$

The penalty, consisting of the sum of the absolute values of the regression coefficients, is an L_1-norm. As already stated before, ridge regression, focusing on squared coefficients, employs an L_2-norm, and measures like AIC or BIC are using the L_0-norm, taking into account only the number of non-zero regression coefficients. In Eq. 10.3, with increasing values for parameter λ more and more regression coefficients will be exactly zero. This method has become known under the name *lasso* (Tibshirani 1996; Hastie et al. 2001); an efficient method to solve this equation—and related approaches—has become known under the name of *least-angle regression*,

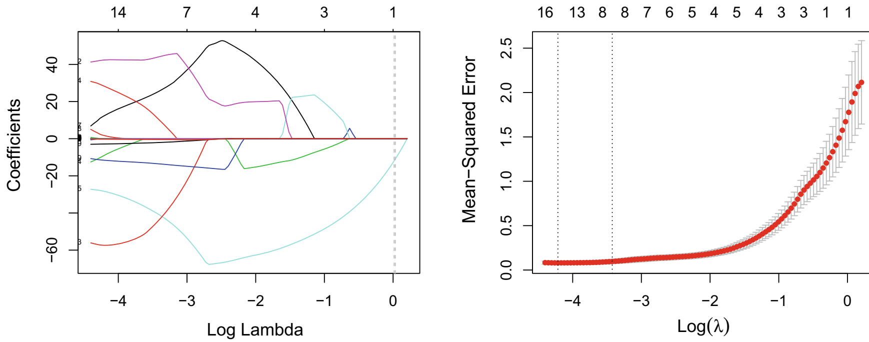

Fig. 10.2 Left: lasso model coefficients plotted against the (relative) penalty size λ. Right: validation plot

or LARS (Efron et al. 2004). Several R versions for the lasso are available. Package **glmnet** is written by the inventors of the method, and will be used here as an example. Other packages implementing similar techniques include **lars**, where slightly different defaults have been chosen for solving the lasso problem, **lpc** for "lassoed principal components" and **relaxo**, a generalization of the lasso using possibly different penalization coefficients for the variable selection and parameter estimation steps.

Rather than one set of coefficients for one given value of λ, the function `glmnet` returns an entire sequence of fits, with corresponding regression coefficients. For the odd rows of the gasoline data, the model is simply obtained as follows:

```
> gas.lasso <- glmnet(x = gasoline$NIR[gas.odd, ],
+                     y = gasoline$octane[gas.odd])
> plot(gas.lasso, xvar = "lambda", label = TRUE)
```

The result of the corresponding `plot` method is shown in the left panel of Fig. 10.2. It shows the (standardized) regression coefficients against the size of the L_1 norm of the coefficient vector. For an infinitely large value of λ, the weight of the penalty, no variables are selected. Gradually decreasing the penalty leads to a fit using only one non-zero coefficient. Its size varies linearly with the penalty—until the next variable enters the fray. The right of the plot shows the position of the entrances of new non-zero coefficients. This piecewise linear behavior is the key to the lasso algorithm, and makes it possible to calculate the whole trace in approximately the same amount of time as needed for a normal linear regression. Around the left axis (and somewhat hard to read in this default set-up), the variable numbers of some of the coefficients are shown at their "final" values, i.e., at the last value for λ, by default one percent of the value at which the first variable enters the model.

Of course, the value of the regularization parameter λ needs to be optimized. A function `cv.glmnet` is available for that, using by default ten-fold crossvalidation. Two common measures are available as predefined choices. Obviously, the model corresponding to the lowest crossvalidation error is one of them; the other is the most

sparse model that is within one standard deviation from the global optimum (Hastie et al. 2001), the same criterion also used in the **pls** package for determining the optimal number of latent variables mentioned in Sect. 8.2.2.

```
> gas.lasso.cv <- cv.glmnet(gasoline$NIR[gas.odd, ],
+                           gasoline$octane[gas.odd])
> svals <- gas.lasso.cv[c("lambda.1se", "lambda.min")]
```

The plot command for the `cv.glmnet` object leads to the validation plot in the right panel of Fig. 10.2. The global minimum in the CV curve lies at a value of -4.215, and the one-se criterion at -3.424 (both in log units, as in the figure). The associated errors can be obtained directly using the `predict` function for the crossvalidation object:

```
> gas.lasso.preds <-
+   lapply(svals,
+          function(x)
+            predict(gas.lasso,
+                    newx = gasoline$NIR[gas.even, ],
+                    s = x))
> sapply(gas.lasso.preds,
+        function(x) rms(x, gasoline$octane[gas.even]))
lambda.1se lambda.min
   0.18881    0.19463
```

The prediction error for the test set using the optimal penalty is better than the best values seen with PCR and PLS, the one with the more conservative estimate somewhat larger. In both cases, only a very small subset of the original variables are included in the model:

```
> gas.lasso.coefs <- lapply(svals,
+                           function(x) coef(gas.lasso, s = x))
> sapply(gas.lasso.coefs,
+        function(x) sum(x != 0))
lambda.1se lambda.min
         9         14
```

A further development is mixing the L_1-norm of the lasso and related methods with the L_2-norm used in ridge regression. This is known as the *elastic net* (Zou and Hastie 2005). The penalty term is given by

$$\sum_i \left(\alpha|\beta_i| + (1-\alpha)\beta_i^2 \right) \tag{10.4}$$

where the sum is over all variables. The result is that large coefficients are penalized heavily (because of the quadratic term) and that many of the coefficients are exactly zero, leading to a sparse solution.

The `glmnet` function provides ridge regression through specifying `alpha = 0` and the lasso with `alpha = 1`. It will be no surprise that values of `alpha` between zero and one lead to the elastic net:

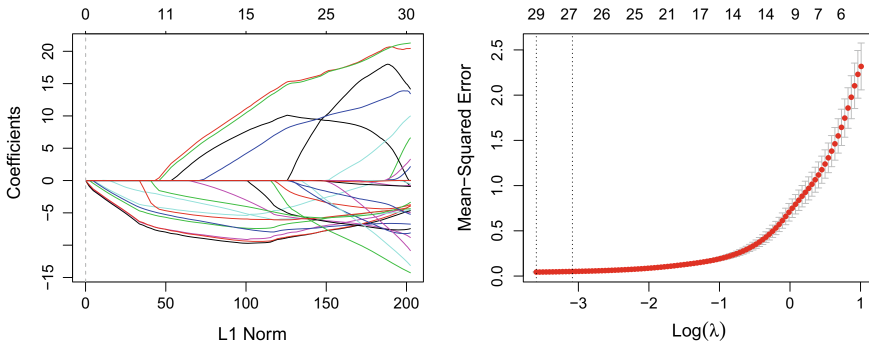

Fig. 10.3 Elastic net results for the gasoline data using $\alpha = 0.5$. The left plot shows the development of the regression coefficients upon relaxation of the penalty parameter. The right plot shows the ten-fold crossvalidation curve, optimizing λ

```
> gas.elnet <- glmnet(gasoline$NIR, gasoline$octane, alpha = .5)
> plot(gas.elnet, "norm")
```

The result is shown in the left plot in Fig. 10.3. Further inspection of the elastic net model, including the crossvalidation plot on the right side of Fig. 10.3, is completely analogous to the code shown earlier for the lasso. The performance of the elastic net in predicting the test set is slightly better than the lasso, at the expense of including more variables:

```
> sapply(gas.elnet.preds,
+        function(x) rms(x, gasoline$octane[gas.even]))
lambda.1se lambda.min
   0.15881    0.15285
> sapply(gas.elnet.coefs,
+        function(x) sum(x != 0))
[1] 28 30
```

The coefficients that are selected by the global-minimum lasso and elastic-net models are shown in Fig. 10.4. There is good agreement between the two sets; the elastic net in general selects variables in the same region as the lasso, with the exception of the area around 1000 nm with is not covered by the lasso at all. Note that the coefficient sizes for the elastic net are much smaller (in absolute size) than the ones from the lasso, a result of the L_2 penalization.

10.3 Global Optimization Methods

Given the speed of modern-day computing, it is possible to examine large numbers of different models and select the best one. However, as we already saw with leaps-and-bounds approaches, even in cases with a moderate number of variables it is practically

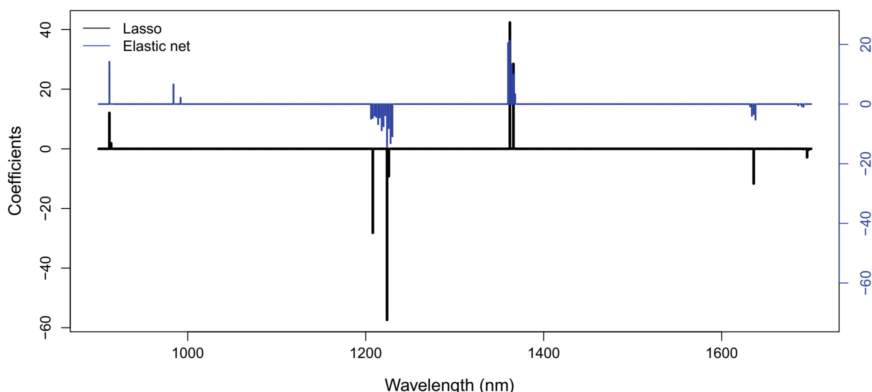

Fig. 10.4 Non-zero coefficients in the lasso and elastic net models. A small vertical offset has been added to facilitate the comparison

impossible to assess the quality of all subsets. One must, therefore, limit the number of subsets that is going to be considered to a manageable size. The stepwise approach does this by performing a very local search around the current best solution before adding or removing one variable; it can be compared to a steepest-descent strategy. The obvious disadvantage is that many areas of the search space will never be visited. For regression or classification cases with many variables, almost surely the method will find a local optimum, very often of low quality.

An alternative is given by random search—just sampling randomly from all possible subsets until time is up. Of course, the chance of finding the global optimum in this way is smaller than the chance of winning the lottery... What is needed is a search strategy that combines random elements with "gradient" information; that is, a strategy that uses information, available in solutions of higher quality, with the ability to throw that information away if needed, in order to be able to escape from local optima. This type of approaches has become known under the heading of *global search* strategies. The two best-known ones in the area of chemometrics are *Simulated Annealing* and *Genetic Algorithms*. Both will be treated briefly below.

What is quality, in this respect, again depends on the application. In most cases, the property of interest will be the quality of prediction of unseen data, which for larger data sets can conveniently be estimated by crossvalidation approaches. For data sets with few samples, this will not work very well because of the coarse granularity of the criterion: many subsets will lead to an equal number of errors. Additional information should be used to distinguish between these.

10.3.1 Simulated Annealing

In Simulated Annealing (SA, Kirkpatrick et al. 1983; Cerny 1985), a sequence of candidate solutions is assessed, starting from a random initial point. A new solution with quality E_{t+1}, not too far away from the current one (with quality E_t), is unconditionally accepted if it is better than the current one. If $E_t > E_{t+1}$ on the other hand, accepting the move corresponds to a deterioration. However, and this is the defining feature of SA, such a move can be accepted, with a probability equal to

$$p_{\mathrm{acc}} = \exp\left(\frac{E_{t+1} - E_t}{T_t}\right) \tag{10.5}$$

where T_t the state of the control parameter at the current time point t. Note that p_{acc}, defined in this way, is always between zero and one (since $E_t > E_{t+1}$). This criterion is known as the *Metropolis* criterion (Metropolis et al. 1953). Other criteria are possible, too, but are rarely used.

The name Simulated Annealing comes from an analogy to annealing in metallurgy, where crystals with fewer defects can be created by repeatedly heating and cooling a material: during the (slow) cooling, the atoms are able to find their energetically most favorable positions in a regular crystal lattice, whereas the heating allows atoms that have been caught in unfavorable positions (local optima) to "try again" in the next cooling stage. The analogy with the optimization task is clear: if an improvement is found (better atom positions) it is accepted; if not, then sometimes a deterioration in quality is accepted in order to be able to cross a ridge in the solution landscape and to find an solution that is better in the end. Very often, the control parameter is therefore indicated with T, to stress the analogy with temperature. During the optimization, it will slowly be decreasing in magnitude—the cooling—causing fewer and fewer solutions of lower quality to be accepted. In the end, only real improvements are allowed. It can be shown that SA leads to the global optimum if the cooling is slow enough (Granville et al. 1994); unfortunately, the practical importance of this proof is limited since the cooling may have to be infinitely slow. Note that random search is a special case that can be achieved simply by setting T_t to an extremely large value, leading to $p_{\mathrm{acc}} = 1$ whatever the values of E_{t+1} and E_t.

The naive implementation of an SA therefore can be very simple: one needs a function that generates a new solution in the neighborhood of the current one, an evaluation function to assess the quality of the new solution, and the acceptance function, including a cooling schedule for the search parameter T. The evaluation needs to be defined specifically for each problem. In regression or classification cases typically some estimate of prediction accuracy is used such as crossvalidation—note that the evaluation function in this schedule probably is the most time-consuming step, and since it will be executed many times (typically thousands or, in complicated

cases, even millions of solutions are evaluated by global search methods) it should be very fast. If enough data are available then one could think of using a separate test set for the evaluation, or of using quality criteria such as Mallows's C_p, or AIC or BIC values, mentioned in Chap. 9. The whole SA algorithm can therefore easily be summarized in a couple of steps:

1. Choose a starting temperature and state;
2. Generate and evaluate a new state;
3. Decide whether to accept the new state;
4. Decrease the temperature parameter;
5. Terminate or go to step 2.

Several SA implementations are available in R. We will have a look at the `optim` function from the core **stats** package which implements a general-purpose SA function.

Let us see how this works in the two-class wines example from Sect. 10.1.2, excluding the Barolo variety. This is a simple example for which it still is quite difficult to assess all possible solutions, especially since we do not force a model with a specific number of variables. We will start with the general-purpose `optim` approach, since this provides most insight in the inner workings of the SA. First we need to define an evaluation function. Here, we use the fast built-in LOO classification estimates of the `lda` function:

```
> lda.loofun <- function(selection, xmat, grouping, ...) {
+   if (sum(selection) == 0) return(100)
+   lda.obj <- lda(xmat[, selection == 1], grouping, CV = TRUE)
+   100*sum(lda.obj$class != grouping)/length(grouping)
+ }
```

Argument `selection` is a vector of numbers here, with ones at the position of the selected variables, and zeroes elsewhere. Since `optim` by default does minimization, the evaluation function returns the percentage of misclassified cases—note that if no variables are selected, a value of 100 is returned.

Now that we have defined what exactly we are going to optimize, we need to define a step function, leading from the current solution to the next. A simple approach could be to do one of three things: either remove a variable, add a variable, or replace a variable. If too few variables are selected, we could increase the number by adding one previously unselected variable randomly (so the escape clause in the evaluation function checking for zero selected variables should never be reached). That seems easy enough to put in a function:

```
> saStepFun <- function(selected, ...) {
+   maxval <- length(selected)
+   selection <- which(selected == 1)
+   newvar2 <- sample(1:maxval, 2)
+
+   ## too short: add a random number
+   if (length(selection) < 2) {
+     result <- unique(c(selection, newvar2))[1:2]
+   } else {      # generate two variable numbers
+     presentp <- newvar2 %in% selection
+     ## if both are in x, remove the first
+     if (all(presentp)) {
+       result <- selection[selection != newvar2[1]]
+     } else {    # if none are in selection, add the first
+       if (all(!presentp)) {
+         result <- c(selection, newvar2[1])
+       } else { # otherwise swap
+         result <- c(selection[selection != newvar2[presentp]],
+                     newvar2[!presentp])
+       }}}
+
+   newselected <- rep(0, length(selected))
+   newselected[result] <- 1
+   newselected
+ }
```

Both in the evaluation and step function we use the ellipses (`...`) to prevent undefined arguments to throw errors: `optim` simply transfers all arguments that are not its own to both underlying functions, where they can be used or ignored.

We will start with a random subset of five columns. This leads to the following misclassification rate:

```
> initselect <- rep(0, ncol(wines))
> initselect[sample(1:ncol(wines), 5)] <- 1
> (r0 <- lda.loofun(initselect, x = twowines,
+                   grouping = twovintages))
[1] 2.521
```

This corresponds to 3 misclassifications. How much can we improve using simulated annealing? Let's find out:

```
> SAoptimWines <-
+   optim(initselect,
+         fn = lda.loofun, gr = saStepFun, method = "SANN",
+         x = twowines, grouping = twovintages)
```

The result is a simple list with the first two elements containing the best result and the corresponding evaluation value:

```
> SAoptimWines[c("par", "value")]
$par
 [1] 1 0 0 0 1 0 1 1 0 1 1 0 1

$value
[1] 0
```

In this case, all misclassifications have been eliminated while still using only a subset of the variables, in this case 7 columns. We could try to push the number of selected variables back by adding a small penalty for every selected variable—once the ideal value of zero misclassifications has been reached the current definition of the evaluation function gives no more opportunities for further improvement.

By default, 10,000 evaluations are performed in the `optim` version of SA; this number can be changed using the `control` argument, where also the initial temperature and the cooling rate can be adjusted. In real, nontrivial problems, it will probably take some experimentation to find optimal values for these search parameters.

A more ambitious example is to predict the octane number of the gasoline samples with only a subset of the NIR wavelengths. The step function is the same as in the wine example, and the only thing we have to do is to define the evaluation function:

```
> pls.cvfun <- function(selection, xmat, response, ncomp, ...) {
+   if (sum(selection) < ncomp) return(Inf)
+   pls.obj <- plsr(response ~ xmat[, selection == 1],
+                   validation = "CV", ncomp = ncomp, ...)
+   c(RMSEP(pls.obj, estimate = "CV", ncomp = ncomp,
+           intercept = FALSE)$val)
+ }
```

In this case, we use the explicit crossvalidation provided by the `plsr` function. This adds a little bit of variability in the evaluation function since repeated application will lead to different segments—but the savings in time are quite big. We will assume that this variability is smaller than the gains we hope to make. The number of components to take into account can be specified in the extra argument of the evaluation function; the error of the model with the largest number of latent variables is returned. The `RMSEP` function returns an object of class `mvrVal`, where the `val` list element contains the numerical value of interest—this is what we will return. Now, let us try to find an optimal two-component PLS model (fewer variables often lead to less complicated models). We start with a very small model using only eight variables ($.02 \times 401$):

```
> nNIR <- ncol(gasoline$NIR)
> initselect <- rep(0, nNIR)
> initselect[sample(1:nNIR, 8)] <- 1
> SAoptimNIR1 <-
+   optim(initselect,
+         fn = pls.cvfun, gr = saStepFun, method = "SANN",
+         x = gasoline$NIR, response = gasoline$octane,
+         ncomp = 2, maxval = nNIR)
```

```
> pls.cvfun(initselect, gasoline$NIR, gasoline$octane, ncomp = 2)
[1] 1.3896
> (nvarSA1 <- sum(SAoptimNIR1$par))
[1] 190
> SAoptimNIR1$value
[1] 0.24823
```

The result is already quite good: compare this, e.g., to the values in the left panel in Fig. 8.4 (where we were looking at the crossvalidation of a model based on the odd samples only) and it is clear that the estimated error with fewer variables and two components is less than half that of the two-component model including all variables.

Still, we see that the number of variables included in the model is quite high—perhaps more sparse models can be found that are equally good or even better. In such cases, it pays to abandon the naive approach adopted above and look closer at the problem itself. We should realize we are optimizing a long parameter vector in this case, with 401 values. Many of these values are zero to start with, and we would like to retain the sparsity of the solution. Our step function, however, is not taking this into account and will suggest many steps leading to more variables. Combine that with a rather high initial temperature parameter, and it is clear that especially in the beginning many bad moves will be accepted. Finally, the evaluation function does not reward sparse solutions explicitly. Let's see what a lower starting temperature and an adapted evaluation function contribute. First we will define the latter:

```
> pls.cvfun2 <- function(selection, xmat, response, ncomp,
+                        penalty = 0.01, ...) {
+   if (sum(selection) < ncomp) return(Inf)
+   pls.obj <- plsr(response ~ xmat[, selection == 1, drop = FALSE],
+                   validation = "CV", ncomp = ncomp, ...)
+   c(RMSEP(pls.obj, estimate = "CV", ncomp = ncomp,
+          intercept = FALSE)$val) +
+     penalty * sum(selection)
+ }
```

Note that we need at least `ncomp` variables in order to fit a PLS model. Next, we will define a step function that, using the default settings, will keep the number of variables approximately equal to the starting situation. By playing with the cutoffs in the argument `plimits` (a random draw from the uniform distribution lower than the first value will lead to eliminating one of the selected variables; anything larger than the second number to the addition of a variable, and anything in between to swapping variables) one can tweak the behavior of the step function:

```
> saStepFun2 <- function(selected, plimits = c(.3, .7), ...) {
+   dowhat <- runif(1)
+
+   ## decrease selection
+   if (dowhat < plimits[1]) {
+     if (sum(selected) > 2) { # not too small...
+       kickone <- sample(which(selected == 1), 1)
+       selected[kickone] <- 0
+       return(selected)
+     }
+   }
+
+   ## increase selection
+   if (dowhat > plimits[2]) { # not too big...
+     if (sum(selected) < length(selected)) {
+       addone <- sample(which(selected == 0), 1)
+       selected[addone] <- 1
+       return(selected)
+     }
+   }
+
+   ## swap
+   kickone <- sample(which(selected == 1), 1)
+   selected[kickone] <- 0
+   addone <- sample(which(selected == 0), 1)
+   selected[addone] <- 1
+   selected
+ }
```

By changing the values of the `plimits` argument we can directly influence the number of nonzero entries in the result: e.g., the higher the first number, the bigger the chance that a variable will be removed. Let's see how that works, combined with a lower starting temperature. The default is 10—we will try a value of 1:

```
> penalty <- 0.01
> SAoptimNIR2 <-
+   optim(initselect,
+         fn = pls.cvfun2, gr = saStepFun2, method = "SANN",
+         x = gasoline$NIR, response = gasoline$octane,
+         ncomp = 2, maxval = nNIR,
+         control = list(temp = 1))
```

This leads to the following results:

```
> (nvarSA2 <- sum(SAoptimNIR2$par))
[1] 6
> SAoptimNIR2$value - penalty*nvarSA2
[1] 0.20047
```

Now we have a crossvalidation error that is clearly better than what we saw earlier but with only 6 instead of 190 variables in the model.

Several packages provide SA functions specifically optimized for variable selection. The `anneal` function in package **subselect**, e.g., can be used for variable selec-

tion in situations like discriminant analysis, PCA, and linear regression, according to the criterion employed. For LDA, this function takes the between-groups covariance matrix, the minimal and maximal number of variables to be selected, the within-groups covariance matrix and its expected rank, and a criterion to be optimized (see below) as arguments. For the wine example above, a solution to find the optimal three-variable subset would look like this:

```
> winesHmat <- ldaHmat(twowines.df[, -1], twowines.df[, 1])
> wines.anneal <-
+   anneal(winesHmat$mat, kmin = 3, kmax = 3,
+          H = winesHmat$H, criterion = "ccr12", r = 1)


> wines.anneal$bestsets
       Var.1 Var.2 Var.3
Card.3     2     7    10
> wines.anneal$bestvalues
 Card.3
0.83281
```

Repeated application (using, e.g., `nsol = 10`) in this case leads to the same solution every time. Rather than the direct estimates of prediction error, the `anneal` function uses functions of the within- and between-groups covariance matrices (Silva 2001). In this case using the `ccr12` criterion, the first root of $\boldsymbol{B}\boldsymbol{W}^{-1}$ is optimized, analogous to Fisher's formulation of LDA in Sect. 7.1.3. As an other example, Wilk's Λ is given by

$$\Lambda = \det(\boldsymbol{W}) / \det(\boldsymbol{T}) \tag{10.6}$$

and is (in a slightly modified form) available in the `tau2` criterion. For the current case where the dimensionality of the within-covariance matrices is estimated to be one, all criteria lead to the same result.

The new result differs from the subset from our own implementation in only one instance: variable 11, color hue, is swapped for the malic acid concentration. The reason, of course, is that both functions optimize different criteria. Let us see how the two solutions fare when evaluated with the criterion of the other algorithm. The value for the `ccr12` criterion of the solution using variables 7, 10 and 11, found with our own simplistic SA implementation, can be assessed easily:

```
> ccr12.coef((nrow(twowines.df) - 1) * var(twowines.df[, -1]),
+            winesHmat$H, r = 1, c(7, 10, 11))
[1] 0.82293
```

which, as expected, is slightly lower than that of the set consisting of variables 2, 7 and 10. Conversely, the prediction quality of the newer set is slightly worse (two misclassifications):

```
> selection <- rep(0, ncol(twowines))
> selection[c(2, 7, 10)] <- 1
> lda.loofun(selection, twowines.df[, -1], twowines.df[, 1])
[1] 1.6807
```

Obviously, there are probably many sets with the same or similar values for the quality criterion of interest, and to some extent it is a matter of chance which one is returned by the search algorithm. Moreover, the number of possible quality values can be limited, especially with criteria based on the number of misclassifications. This can make it more difficult to discriminate between two candidate subsets.

The `anneal` function for subset selection is also applicable in other types of problems than classification alone: e.g., for variable selection in PCA it uses a measure of similarity of the original data matrix and of the projections on the k-variable subspace—again, several different criteria are available. The speed and applicability in several domains are definite advantages of this particular implementation. However, there are some disadvantages, too: firstly, because of the formulation using covariance matrices it is hard to apply `anneal` to problems with large numbers of variables. Finding the most important discriminating variables in the prostate data set would stretch your computer to the limit—in fact, even the gasoline example requires the argument `force = TRUE` since the default is to refuse cooperation (and give a serious-looking warning) as soon as the number of variables exceeds 400.

Secondly, the function does not allow one to submit an evaluation function, and one has to do with the predefined set—crossvalidation-based approaches such as used in the examples above cannot be implemented, increasing the danger of overfitting. Finally, it can be important to monitor the progress of the optimization, or at least keep track of the speed with which improvements are found—especially when fine-tuning the SA parameters (temperature, cooling rate) one would like to have the possibility to assess acceptance rates. Currently, no such functionality is provided in the **subselect** package.

One other dedicated SA approach for variable selection can be found in the **caret** package mentioned in Chap. 7 in the form of the `safs` (simulated annealing feature selection) function. This function does allow crossvalidation-based quality measures to guide the optimization, but also supports external test sets and criteria like AIC. Parallelization is supported at several different levels.

10.3.2 Genetic Algorithms

Genetic Algorithms (GAs, Goldberg 1989) manage a population of candidate solutions, rather than one single solution as is the case with most other optimization methods. Every solution in the population is represented as a string of values, and in a process called cross-over, mimicking sexual reproduction, offspring is generated combining parts of the parent solutions. Random mutations, occurring with relatively low frequency, ensure that some diversity is maintained in the population. The quality of the offspring is measured in an evaluation phase—again in analogy with biology, this quality is often called “fitness”. Strings with a low fitness will have no or only a low probability of reproduction, so that subsequent generations will generally consist of better and better solutions. This obvious imitation of the process of natural selection has led to the name of the technique. GAs have been applied to

a wide range of problems in very diverse fields—several overviews of applications within chemistry can be found in the literature (e.g., Leardi 2001; Niazi and Leardi 2012).

Just like with Simulated Annealing, GAs need an evaluation function to obtain fitness values for trial solutions. A step function, on the other hand, is not needed: the genetic machinery (cross-over and mutation operations) will take care of that. Several parameters need to be set, such as the size of the population, the number of iterations, and the chances of crossover and mutation, but that is all. Population sizes are typically in the order of 50–100; the number of iterations in the order of several hundreds. There are some aspects, however, that are particular for GAs. The first choice we have to make is on the *representation* of the candidate solutions, i.e., the candidate subsets. For variable selection, two obvious possibilities present themselves: either a vector of indices of the variables in the subset, or a string of zeros and ones. For other optimization problems, e.g., non-linear fitting, real numbers can also be used. Secondly, the *selection* function needs to be defined. This determines which solutions are allowed to reproduce, and is the driving force behind the optimization—if all solutions would have the same probability the result would be a random search. Typical selection procedures are to use random sampling with equal probabilities for all solutions above a quality cutoff, or to use random sampling with (scaled) quality indicators as probability weights.

The **GA** package (Scrucca 2013, 2017) provides a convenient and efficient toolbox, supporting for binary, real-valued and permutation representations, and several standard genetic operators. In addition, users can define their own operators. Parallel evaluation of population members is supported (especially useful if the evaluation of a single solution takes some time), and to speed up proceedings even further, local searches can be allowed at random intervals to inject new and useful information in the population. Finally, populations can be "seeded", i.e., one can provide one or more solutions that are thought to be approximately correct.

Applying the `ga` function from the **GA** package to our gasoline data is quite easy. We can use the same evaluation function as used in the SA optimization, `pls.cvfun2`, where a small penalty is applied for solutions with more variables. Since `ga` does maximization only, we multiply the result with -1:

```
> fitnessfun <- function(...) -pls.cvfun2(...)
```

Now we are ready to go. The simplest approach would be to apply standard procedures and hope for the best:

```
> GAoptimNIR1 <-
+   ga(type = "binary", fitness = fitnessfun,
+      x = gasoline$NIR, response = gasoline$octane,
+      ncomp = 2, penalty = penalty,
+      nBits = ncol(gasoline$NIR), monitor = FALSE, maxiter = 100)
```

The result, as we may have expected, still contains many variables, and has a high crossvalidation error:

```
> (nvarGA1 <- sum(GAoptimNIR1@solution))
[1] 149
> -GAoptimNIR1@fitnessValue + penalty*nvarGA1
[1] 3.2732
```

Ouch, that does not look too good. Of course we have not been fair: the random initialization of the GA will lead to a population with approximately 50% selected variables, where the initial SA solution had only 2%. In addition, the default mutation function is biased to this 50% ratio as well: in sparse solutions it is much more likely to add a variable than to remove one. Similar to the adaptation of the step function in SA, we define the mutation function in such a way that setting bits to zero is (much) more likely than setting bits to one, a behavior that can be controlled by the value of the `bias` argument:

```
> myMutate <- function (object, parent, bias = 0.01)
+ {
+     mutate <- parent <- as.vector(object@population[parent, ])
+     n <- length(parent)
+     probs <- abs(mutate - bias)
+     j <- sample(1:n, size = 1, prob = probs)
+     mutate[j] <- abs(mutate[j] - 1)
+     mutate
+ }
```

In the **GA** package these settings are controlled by the `gaControl` function, and changes remain in effect for the rest of the session (or until changed again). Including the new mutation function and using a more reasonable initial state is easily done:

```
> gaControl("binary" = list(mutation = "myMutate"))
> popSize <- 50 # default
> initmat <- matrix(0, popSize, nNIR)
> initmat[sample(1:(popSize*nNIR), nNIR)] <- 1
>
> GAoptimNIR2 <-
+   ga(type = "binary", fitness = fitnessfun,
+      x = gasoline$NIR, response = gasoline$octane,
+      popSize = popSize, nBits = ncol(gasoline$NIR),
+      ncomp = 2, suggestions = initmat, penalty = penalty,
+      monitor = FALSE, maxiter = 100)
```

This leads to the following result:

```
> (nvarGA2 <- sum(GAoptimNIR2@solution))
[1] 4
> -GAoptimNIR2@fitnessValue + penalty*nvarGA2
[1] 0.26728
```

Clearly, this constitutes a substantial improvement over the first optimization result, getting close to the SA solution presented earlier. Of course, more experimentation can easily lead to further improvements (as is the case with SA as well).

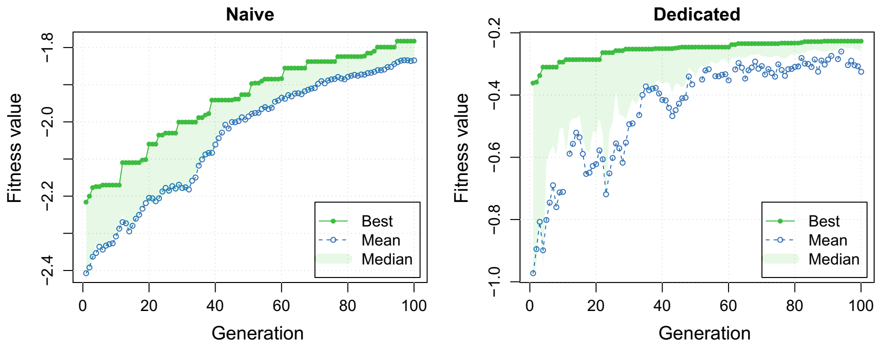

Fig. 10.5 GA optimization results for the NIR data. Left panel: naive application; right panel: application with a specific initialization matrix and a dedicated mutation function. Note the different y scales

One of the nice features of the **GA** package is that the results of calling `ga` can be plotted as well. Figure 10.5 shows the optimization trajectories of both GA runs. First, note the difference in the y-axes: the dedicated GA leads to much better fitnesses. The plots show the best result (green dots/line) as well as the average and the mean fitness values of the population at each iteration. If the latter two are very close to the best value, there is too little variation in the population and the result is likely to be quite bad. Especially with the dedicated mutation operator, one sees quite sudden jumps when "worse" solutions are introduced in the population (too few variables even lead to `Inf` values in which case no Mean is displayed), but still these solutions may contain kernels of good information.

The curves also give an idea on whether it is useful to put in additional effort: the left panel of Fig. 10.5 clearly gives the impression that further improvements are possible. In most cases, playing around with search parameters or tweaking the fitness function will have more chance of reaching good results than simply increasing the number of iterations.

In more complicated problems, speed is a big issue. Some simple tricks can be employed to speed up the optimization. Typically, several candidate solutions will survive unchanged during a couple of generations. Rigorous bookkeeping may prevent unnecessary quality assessments, which in almost all cases is the most computer-intensive part of a GA. An implementational trick that is also very often applied is to let the best few solutions enter the next generation unchanged; this process, called *elitism*, makes sure that no valuable information is thrown away and takes away the need to keep track of the best solution. Provisions can be taken to prevent premature convergence: if the population is too homogeneous the power of the crossover operator decreases drastically, and the optimization usually will not lead to a useful answer. One strategy is to disallow offspring that is equal to other candidate solutions; a second strategy is to penalize the fitness of candidate solutions that are too similar; the latter strategy is sometimes called *sharing*.

Other GA implementations are available, too, of course. The **caret** package includes a `gafs` function that is very similar to the `safs` function we saw earlier for SA. The `genetic` function in the **subselect** package provides a fast Fortran-based GA. The details of the crossover and mutation functions are slightly different from the description above—indeed, there are probably very few implementations that share the exact same crossover and mutation operators, testimony to the flexibility and power of the evolutionary paradigm. Having seen the working of the `anneal` function, most input parameters will speak for themselves:

```
> wines.genetic <-
+   genetic(winesHmat$mat, kmin = 3, kmax = 5, nger = 20,
+           popsize = 50, maxclone = 0,
+           H = winesHmat$H, criterion = "ccr12", r = 1)
> wines.genetic$bestvalues
 Card.3  Card.4  Card.5
0.83281 0.84368 0.85248
> wines.genetic$bestsets
       Var.1 Var.2 Var.3 Var.4 Var.5
Card.3     2     7    10     0     0
Card.4     2     3     7    10     0
Card.5     2     3     7    10    12
```

And indeed, the same three-variable solution is found as the optimal one. This time, also four- and five-variable solutions are returned (because of the values of the `kmin` and `kmax` arguments).

The `maxclone` argument tries to enforce diversity by replacing duplicate offspring by random solutions (which are not checked for duplicity, however). Leaving out this argument would, in this simple example, lead to a premature end of the optimization because of the complete homogeneity of the population. Both `anneal` and `genetic` provide the possibility of a further local optimization of the final best solution.

10.3.3 Discussion

Variable selection is a difficult process. Simple stepwise methods only work with a small number of variables, whereas the largest gains can be made in the nowadays typical situation of hundreds or even thousands of variables. More complicated methods containing elements of random search, such as SA or GA approaches, can have a high variability, especially in cases where correlations between variables are high. One approach is to repeat the variable selection multiple times, and to use those variables that are consistently selected. Although this strategy is intuitively appealing, it does have one flaw: suppose that variables a and b are highly correlated, and that a combination of either a or b with a third variable c leads to a good model. In repeated selection runs, c will typically be selected twice as often as a or b—if the overall selection threshold is chosen to include c but neither of a and b, the model will not work well.

In addition, the optimization criterion is important. It has been shown that LOO crossvalidation as a criterion for variable selection is inconsistent, in the sense that even with an infinitely large data set it will not choose the correct model (Shao 2003). Baumann et al. advocate the use of leave-multiple-out crossvalidation for this purpose (Baumann et al. 2002a, b), even though the computational burden is high. In this approach, the data are repeatedly split, randomly, in training and test sets, where the number of repetitions needs to be greater than the number of variables, and for every split a separate crossvalidation is performed to optimize the parameters of the modelling method such as the number of latent variables in PCR or PLS. A workable alternative is to fix the number of latent variables to a "reasonable" number, and to find the subset of variables that with this particular setting leads to the best results. This takes away the nested crossvalidation but may lead to subsets that are suboptimal. In general, one should accept the fact that there is no guarantee that the optimal subset will be found, and it is wise to accept a subset that is "good enough".