Chapter 9
Validation

Validation is the assessment of the quality of a predictive model, in accordance with the scientific paradigm in the natural sciences: a model that is able to make accurate predictions (the position of a planet in two weeks' time) is—in some sense—a "correct" description of reality. In many applications in the natural sciences, unfortunately, validation is hard to do: chemical and biological processes often exhibit quite significant variation unrelated to the model parameters. An example is the circadian rhythm: metabolomic samples, be it from animals or plants, will show very different characteristics when taken at different time points. When the experimental meta-data on the exact time point of sampling are missing, it will be very hard to ascribe differences in metabolite levels to differences between patients and controls, or different varieties of the same plant. Only a rigorous and consistent experimental design will be able to prevent this kind of fluctuations. Moreover, biological variation between individuals often dominates measurement variation. The bigger the variation, the more important it is to have enough samples for validation. Only in this way, reliable error estimates can be obtained.

The main goal usually is to estimate the expected error when applying the model to new, unseen data: the root-mean-square error of prediction (RMSEP). In general, the expected squared error at a point x is given by

$$\mathrm{Err}(x) = E[(Y - \hat{f}(x))^2] \tag{9.1}$$

which can be decomposed as follows:

$$\mathrm{Err}(x) = (E[\hat{f}(x)] - f(x))^2 + E[(\hat{f}(x) - E[\hat{f}(x)])^2] + \sigma_e^2 \tag{9.2}$$

where E denotes the usual expectation operator. The first term is the squared bias, corresponding to systematic differences between model predictions and measurements, the second is the variance, and σ_e^2 corresponds to the remaining, irreducible error. In many cases one must strike a balance between bias and variance. Biased regression methods like, e.g., ridge regression and PLS achieve lower MSE values by decreasing the variance component, but pay a price by accepting bias. If it is possible

© 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_9

to derive confidence intervals for these, this not only provides an idea of the stability of the model, but it can also be useful in determining which variables actually are important in the model.

A second validation aspect, next to estimating the RMSEP, is to assess the size and variability of the model coefficients, summarized here with the term *model stability*. This is especially true for linear models where one can hope to interpret individual coefficients, and perhaps less so for non-linear models. In the example of multiple regression with a singular covariance matrix in Sect. 8.1.1, the variance of the coefficients effectively is infinite, indicating that the model is highly unstable.

Finally, there is the possibility of making use of prior knowledge. Particularly in the natural sciences, one can often assess whether the features that seem important make sense. In a regression model for spectroscopic data, for instance, one would expect wavelengths with large regression coefficients to correspond to peaks in the spectra—large coefficients in areas where no peaks are present would indicate a not too reliable model. Since in most forms of spectroscopy it is possible to associate spectral features with physico-chemical phenomena (specific vibrations, electron transitions, atoms, ...) one can often even say something about the expected sign and magnitude of the regression coefficients. Should these be very different than expected, one may be on to something big—but more likely, one should treat the predictions of such a model with caution, even when the model appears to fit the data well. Typically, more experiments are needed to determine which of the two situations applies. Because of the problem-specific character of this particular type of validation, we will not treat it any further, but will concentrate on the error estimation and model stability aspects.

9.1 Representativity and Independence

One key aspect is that both error estimates and confidence intervals for the model coefficients are derived from the available data (the training data), but that the model will only be relevant when these data are *representative* for the system under study. If there is any systematic difference between the data on which the model is based and the data for which predictions are required, these predictions will be suboptimal and in some cases even horribly wrong. These systematic differences can have several causes: a new machine operator, a new supplier of chemicals or equipment, new schedules of measurement time ("from now on, Saturdays can be used for measuring as well")—all these things may cause new data to be slightly but consistently different from the training data, and as a result the predictive models are no longer optimal. In analytical laboratories, this is a situation that often occurs, and one approach dealing with this is treated in Sect. 11.6.

Especially with extremely large data sets, validation is sometimes based on only one division in a training set and a test set. If the number of samples is very large, the sheer size of the data will usually prevent overfitting and the corresponding error estimates can be quite good. However, it depends on how the training and test

sets are constructed. A random division is to be preferred; to be even more sure, several random divisions may be considered. This would also allow one to assess the variability in the validation estimates, and is definitely advisable when computing resources allow it.

One can check whether the training data are really representative for the test data: pathological cases where this is not the case can usually be recognized by simple visualization, (e.g., using PCA). However, one should be very careful not to reject a division too easily: as soon as one starts to use the test data, in this case, to assess whether the division between training and test date is satisfactory, there is the risk of biasing the results. The training set should not only be representative of the test set, but also completely *independent*. An example is the application of the Kennard–Stone algorithm (Kennard and Stone 1969) to make the division in training and test sets. The algorithm selects training samples from the complete data set to cover the complete space of the independent variables as good as possible. However, if the training samples are selected in such a way that they are completely surrounding the test samples, the prediction error on the test set will probably be lower than it should be—it is biased. Of course, when the algorithm is only used to decrease the number of samples in the training set, and the test set has been set aside before the Kennard–Stone algorithm is run, then there is no problem (provided the discarded training set samples are not added to the test set!) and we can still treat the error on the test set as an unbiased estimate of what we can expect for future samples.

If the available data can be assumed to be representative of the future data, we can use them in several ways to assess the quality of the predictions. The main point in all cases is the same: from the data at hand, we simulate a situation where unseen data have to be predicted. In *crossvalidation*, this is done by leaving out part of the data, and building the model on the remainder. In *bootstrapping*, the other main validation technique, the data are resampled with replacement, so that some data points are present several times in the training set, and others (the "out-of-bag", or OOB, samples) are absent. The performance of the model(s) on the OOB samples is then an indication of the prediction quality of the model for future samples.

In estimating errors, one should take care not to use *any* information of the test set: if the independence of training and test sets is compromised error estimates become biased. An often-made error is to scale (autoscaling, mean-centering) the data before the split into training and test sets. Obviously, the information of the objects in the test set *is* being used: column means and standard deviations are influenced by data from the test set. This leads to biased error estimates—they are, in general, lower that they should be. In the crossvalidation routines of the **pls** package, for example, scaling of the data is done in the correct way: the OOB samples in a crossvalidation iteration are scaled using the means (and perhaps variances) of the in-bag samples. If, however, other forms of scaling are necessary, this can not be done automatically. The **pls** package provides an explicit `crossval` function, which makes it possible to include sample-specific scaling functions in the calling formula:

```
> gasoline.mscpcr <- pcr(octane ~ msc(NIR), data = gasoline,
+                        ncomp = 4)
> gasoline.mscpcr.cv <- crossval(gasoline.mscpcr, length.seg = 1)
> RMSEP(gasoline.mscpcr.cv, estimate = "CV")
(Intercept)      1 comps      2 comps      3 comps      4 comps
     1.5430       1.4589       0.8901       0.2598       0.2668
```

This particular piece of code applies multiplicative scatter correction (MSC, see Sect. 3.2) on all in-bag samples, and scales the OOB samples in the same way, as it should be done. Interestingly, this leads to a PCR model where three components would be optimal, one fewer component than without the MSC scaling.

A final remark concerns more complicated experimental designs. The general rule is that the design should be taken into account when setting up the validation. As an example, consider a longitudinal experiment where multiple measurements of the same objects at different time points are present in the data. When applying subsampling approaches like crossvalidation to such data sets one should leave out complete objects, rather than individual measurements: obviously multiple measurements of the same object, even taken at different times, are not independent. Randomly sampling individual data points would probably lead to over-optimistic validation estimates.

9.2 Error Measures

A distinction has to be made between the prediction of a continuous variable (regression), and a categorical variable, as in classification. In regression, the root-mean-square error of validation (RMSEV) is given, analogously to Eq. 8.12, by

$$\text{RMSEV} = \sqrt{\frac{\sum_i (\hat{y}_{(i)} - y_{(i)})^2}{n}} \tag{9.3}$$

where $y_{(i)}$ is the out-of-bag sample in a crossvalidation or bootstrap. That is, the predictions are made for samples that have not been used in building the model. A summary of these prediction errors can be used as an estimate for future performance. In this case, the average of the sum of squared errors is taken—sometimes there are better alternatives.

For classification, the simplest possibility is to look at the fraction of correctly classified observations. in R:

```
> err.rate <- function(x, y) sum(x != y) / length(x)
```

A more elaborate alternative is to assign each type of misclassification a *cost*, and to minimize a loss function consisting of the total costs associated with misclassifications. In a two-class situation, for example, this makes it possible to prevent false negatives at the expense of accepting more false positives; in a medical context, it may be the case that a specific test should recognize all patients with a specific

disease, even if that means that a few people without the disease are also tagged. Missing a positive sample (a false negative outcome) in this example has much more radical consequences than the reverse, incorrectly calling a healthy person ill.

A related alternative is to focus on the two components of classification accuracy, *sensitivity* and *specificity*. Sensitivity, also known as the *recall rate* or the *true positive rate*, is the fraction of objects from a particular class k which are actually assigned to that class:

$$\text{sensitivity}_k = \frac{TP_k}{TP_k + FN_k} \tag{9.4}$$

where TP_k is the number of True Positives (i.e., objects correctly assigned to class k) and FN_k is the number of False Negatives (objects belonging to class k but classified otherwise). A sensitivity of one indicates that all objects of class k are assigned to the correct class—note that many other objects, not of class k, may be assigned to that class as well.

Specificity is related to the purity of class predictions, and summarizes the fraction of objects in class k that belong elsewhere:

$$\text{specificity}_k = \frac{TN_k}{FP_k + TN_k} \tag{9.5}$$

TN_k and FP_k indicate True Negatives and False Positives for class k, respectively. A specificity of one indicates that no objects have been classified as class k incorrectly. The measure 1—specificity is sometimes referred to as the *false positive rate*.

In practice, one will have to compromise between specificity and sensitivity: usually, sensitivity can be increased at the expense of specificity and vice versa by changing parameters of the classification procedure. For two-class problems, a common visualization is the Receiver Operating Characteristic (ROC, Brown and Davis 2006), which plots the true positive rate against the false positive rate for several values of the classifier threshold. Consider, e.g., the optimization of k, the number of neighbors in the KNN classification of the wine data. Let us focus on the distinction between Barbera and Grignolino, where we (arbitrarily) choose Barbera as the positive class, and Grignolino as negative.

```
> X <- wines[vintages != "Barolo", ]
> vint <- factor(vintages[vintages != "Barolo"])
> kvalues <- 1:12
> ktabs <- lapply(kvalues,
+                  function(i) {
+                    kpred <- knn.cv(X, vint, k = i)
+                    table(vint, kpred)
+                  })
```

For twelve different values of k we calculate the crossvalidated predictions and we save the crosstable. From the resulting list we can easily calculate true positive and false positive rates:

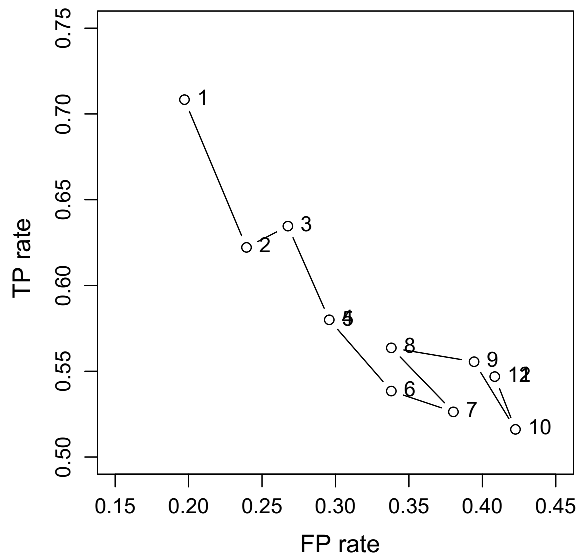

Fig. 9.1 ROC curve (zoomed in to display only the relative part) for the discrimination between Grignolino and Barbera wines using different values of k in KNN classification. Predictions are LOO-crossvalidated

```
> TPrates <- sapply(ktabs, function(x) x[1, 1]/sum(x[, 1]))
> FPrates <- sapply(ktabs, function(x) 1 - x[2, 2]/sum(x[2, ]))
> plot(FPrates, TPrates, type = "b",
+      xlim = c(.15, .45), ylim = c(.5, .75),
+      xlab = "FP rate", ylab = "TP rate")
> text(FPrates, TPrates, 1:12, pos = 4)
```

In this case, the result, shown in Fig. 9.1, leaves no doubt that $k = 1$ gives the best results: it shows the lowest fraction of false positives (i.e., Grignolinos predicted as Barberas) as well as the highest fraction of true positives. The closer a point is to the top left corner (perfect prediction), the better.

Note that a careful inspection of model residuals should be a standard ingredient of any analysis. Just summing up the number of misclassifications, or squared errors, is not telling the whole story. In some parts of the data space one might see, e.g., more misclassifications or larger errors than in other parts. For simple univariate regression, standard plots exist (simply plotting an `lm` object in R will give a reasonable subset)—for multivariate models techniques like PCA can come in handy, but there is ample opportunity for creativity from the part of the data analyst.

9.3 Model Selection

In the field of *model selection*, one aims at selecting the best model amongst a series of possibilities, usually based on some quality criterion such as an error estimate. What makes model selection slightly special is that we are not interested in the error

estimates themselves, but rather in the order of the sizes of the errors: we would like to pick the model with the smallest error. Also biased error estimates are perfectly acceptable when the bias does not influence our choice, and in some cases biased estimates are even preferable since they often have a lower variance. We will come back to this in later sections, discussing different resampling-based estimates.

9.3.1 Permutation Approaches

A form of resampling that we have not yet touched upon is *permutation*, randomly redistributing labels in order to simulate one possible realization of the data under a null hypothesis. The concept is most easily explained in the context of classification. Suppose we have a classifier that distinguishes between two classes, A and B, each represented by the same number of samples. Let's say the classifier achieves a 65% correct prediction rate. The question is whether this could be due to chance. In a permutation test, one would train the same classifier many times on a data set in which class labels A and B would be randomly assigned to samples. Since in such a permutation there is no relation between the dependent and independent variables, one would expect a success rate of 50%. In practice one will see variation. Comparing the prediction rate observed with the real data with the quantiles of the permutation prediction rates gives an estimate of the p value, or in other words, tells you whether the model is significant or not. In the example above: if we would do 500 permutations and in 78 of them we would find prediction rates above 65%, we should conclude that our classifier is not doing significantly better than a chance process. If only three of the 500 permutations would lead to prediction rates of 65% or more, on the other hand, we would declare our model significant.

So where other forms of validation try to obtain an error measure, permutation testing as described above aims to assess significance. While the principle remains the same, there are other ways in which the permutation test can be used. One example was given in Sect. 8.2.2 in the context of establishing the optimal number of components in a PCR or PLS regression model. There, residuals of models using A and $A + 1$ components, respectively, are being permuted and the true sum of squares is then compared to the null distribution given by the set of permutations. If there is no significant difference between the two, the smallest model with A components is preferred. In the remainder of this chapter we'll focus on establishing estimates for the magnitude of the prediction errors.

9.3.2 Model Selection Indices

Resampling approaches such as crossvalidation can be time-consuming, especially for large data sets or complicated models. In such cases simple, direct estimates could form a valuable alternative. The most common ones consist of a term indicating

the agreement between empirical and predicted values, and a penalty for model complexity. Important examples are Mallows's C_p (Mallows 1973) and the AIC and BIC values (Akaike 1974; Schwarz 1978), already encountered in Sect. 6.3. The C_p value is a special case of AIC for general models, adjusting the expected value in such a way that it is approximately equal to the prediction error. In a regression context, these two measures are equal, given by

$$\text{AIC} = C_p = \text{MSE} + 2 \times p\,\hat{\sigma}^2/n \tag{9.6}$$

$$\text{BIC} = \text{MSE} + \log n \times p\,\hat{\sigma}^2/n \tag{9.7}$$

where n is the number of objects, p is the number of parameters in the model, MSE is the mean squared error of calibration, and $\hat{\sigma}^2$ is an estimate of the residual variance—an obvious choice would be $\text{MSE}/(n-p)$ (Efron and Tibshirani 1993). It can be seen that, for any practical data size, BIC penalizes more heavily than C_p and AIC, and therefore will choose more parsimonious models. For model selection in the life sciences, these statistics have never really been very popular. A simple reason is that it is hard to assess the "true" value of p: how many degrees of freedom do you have in a PLS or PCR regression? Methods like crossvalidation are more simple to apply and interpret—and with computing power being cheap, scientists happily accept the extra computational effort associated with it.

9.3.3 Including Model Selection in the Validation

Up to now we have concentrated on validation approaches such as crossvalidation for particular models, e.g., a PLS model with four components, or a KNN classifier with $k = 3$. Typically, we would repeat this validation for other numbers of latent variable, or other values of k, and base the selection of the best model on some kind of decision rule (e.g., the approaches mentioned in Sect. 8.2.2 for choosing the number of latent variables in a multivariate regression model). As has been stated before, the CV error estimate associated with the selected model is to be interpreted in a relative way, indicating which of the models under comparison is the best one—it's value should not be taken absolutely. The reason is the model selection process: we choose this model precisely *because* it has the lowest error, and so we introduce a downward bias.

There is another option. Rather than performing crossvalidation (or bootstrapping or any other validation technique) on one fully specified model, one could also use it on the complete procedure, including the model selection (Efron and Hastie 2016). That is, if we decide to choose the optimal number of latent variables in a PLS model using the one-sigma rule mentioned in Sect. 8.2.2, we could simply apply crossvalidation on the overall procedure, including applying the selection rule. The

resulting error estimate[1] now *is* an unbiased estimate of what we can expect for future data. Note that in each crossvalidation iteration the optimal number of components may be different. This usually adds variance, so error estimates obtained with this procedure are expected to be larger than the estimates we have been discussing until now, which makes sense.

9.4 Crossvalidation Revisited

Crossvalidation, as we already have seen, is a simple and trustworthy method to estimate prediction errors. There are two main disadvantages of LOO crossvalidation. The first is the time needed to perform the calculations. Especially for data sets with many objects and time-consuming modelling methods, LOO may be too expensive to be practical. There are two ways around this problem: the first is to use fast alternatives to direct calculations—in some cases analytical solutions exist, or fast and good approximations. A second possibility is to focus on leaving out larger segments at a time. This latter option also alleviates the second disadvantage of LOO crossvalidation—the relatively large variability of its error estimates.

9.4.1 LOO Crossvalidation

Let us once again look at the equation for the LOO crossvalidation error:

$$\varepsilon_{CV}^2 = \frac{1}{n}\sum_{i=1}^{n}\left(y_{(i)} - \hat{y}_{(i)}\right)^2 = \frac{1}{n}\sum_{i=1}^{n}\varepsilon_{(i)}^2 \tag{9.8}$$

where subscript (i) indicates that observation i is being predicted while not being part of the training data. Although the procedure is simple to understand and implement, it can take a lot of time to run for larger data sets. However, for many modelling methods it is not necessary to calculate the n different models explicitly. For ordinary least-squares regression, for example, one can show that the ith residual of a LOO crossvalidation is given by

$$\varepsilon_{(i)}^2 = \varepsilon_i^2/(1 - h_{ii}) \tag{9.9}$$

where ε_i^2 is the squared residual of sample i when it is *included* in the training set, and h_{ii} is the ith diagonal element of the hat matrix $\boldsymbol{H}$, given by

$$\boldsymbol{H} = \boldsymbol{X}\left(\boldsymbol{X}^T\boldsymbol{X}\right)^{-1}\boldsymbol{X}^T \tag{9.10}$$

[1] This in effect is an example of double crossvalidation, since the selection rule internally uses crossvalidation, too. We'll come back to this in a later section in this chapter.

Therefore, the LOO error estimate can be obtained without explicit iteration by

$$\varepsilon^2_{CV} = \frac{1}{n} \sum_{i=1}^{n} \left(\frac{y_i - \hat{y}_i}{1 - h_{ii}} \right)^2 \tag{9.11}$$

This shortcut is available in all cases where it is possible to write the predicted values as a product of a type of hat matrix $\boldsymbol{H}$, independent of y, and the measured y values:

$$\hat{y} = \boldsymbol{H} y \tag{9.12}$$

Generalized crossvalidation (GCV, Craven and Wahba 1979) goes one step further: instead of using the individual diagonal elements of the hat matrix h_{ii}, the average diagonal element is used:

$$\varepsilon^2_{GCV} = \frac{1}{n\left(1 - \sum_{j=1}^{n} h_{jj}\right)^2} \sum_{i=1}^{n} \left(y_i - \hat{y}_i\right)^2 \tag{9.13}$$

Applying these equations to PCR leads to small differences with the usual LOO estimates, since the principal components that are estimated when leaving out each sample in turn will deviate slightly (assuming there are no gross outliers). Consider the (bad) fit of the one-component PCR model for the gasoline data, calculated with explicit construction of n sets of size $n - 1$:

```
> gasoline.pcr <- pcr(octane ~ ., data = gasoline,
+                     validation = "LOO", ncomp = 1)
> RMSEP(gasoline.pcr, estimate = "CV")
(Intercept)      1 comps
      1.543        1.447
```

The estimate based on Eq. 9.11 is obtained by

```
> gasoline.pcr2 <- pcr(octane ~ ., data = gasoline, ncomp = 1)
> X <- gasoline.pcr2$scores
> HatM <- X %*% solve(crossprod(X), t(X))
> sqrt(mean((gasoline.pcr2$residuals/(1 - diag(HatM)))^2))
[1] 1.4187
```

The GCV estimate from Eq. 9.13 deviates more from the LOO result:

```
> sqrt(mean((gasoline.pcr2$residuals/(1 - mean(diag(HatM))))^2))
[1] 1.3888
```

If one is willing to ignore the variation in the PCs introduced by leaving out individual objects, as may be perfectly acceptable in the case of data sets with many objects, this provides a way to significantly speed up calculations. The example above was four times faster than the explicit loop, as is implemented in the `pcr` function with the `validation = "LOO"` argument. For PLS, it is a different story: there, the latent variables are estimated using y, and Eq. 9.12 does not hold.

9.4.2 Leave-Multiple-Out Crossvalidation

Instead of leaving out one sample at a time, it is also possible to leave out a sizeable fraction, usually 10% of the data; the latter is also called “ten-fold crossvalidation”. This approach has become quite popular—not only is it roughly ten times faster, it also shows less variability in the error estimates (Efron and Tibshirani 1993). Again, there is a bias-variance trade-off: the variance may be smaller, but a small bias occurs because the model is based on a data set that is appreciably smaller than the “real” data set, and therefore is slightly pessimistic by nature.

This “leave-multiple-out” (LMO) crossvalidation is usually implemented in a random way: the order of the rows of the data matrix is randomized, and consecutive chunks of roughly equal size are used as test sets. In case the data are structured, it is possible to use non-randomized chunks: the functions in the **pls** package have special provisions for this. The following lines of code lead, e.g., to interleaved sample selection:

```
> gasoline.pcr <- pcr(octane ~ ., data = gasoline,
+                     validation = "CV", ncomp = 4,
+                     segment.type = "interleaved")
> RMSEP(gasoline.pcr, estimate = "CV")
(Intercept)      1 comps      2 comps      3 comps      4 comps
     1.5430       1.4261       1.4457       1.2179       0.2468
```

An alternative is to use `segment.type = "consecutive"`. Also, it is possible to construct the segments (i.e., the crossvalidation sets) by hand or otherwise, and explicitly present them to the modelling function using the `segments` argument. See the manual pages for more information.

9.4.3 Double Crossvalidation

In all cases where crossvalidation is used to establish optimal values for modelling parameters, the resulting error estimates are not indicative of the performance of future observations. They are biased, in that they are used to pick the optimal model. Another round of validation is required. This leads to *double crossvalidation* (Stone 1974), as visualized in Fig. 9.2: the inner crossvalidation loop is used to determine the optimal model parameters, very often, in chemometrics, the optimal number of latent variables, and the outer crossvalidation loop assesses the corresponding prediction error. At the expense of more computing time, one is able to select optimal model parameters as well as estimate prediction error.

The problem is that usually one ends up selecting different parameter settings in different crossvalidation iterations: leaving out segment 1 may lead to a PLS model with two components, whereas segment two may seem to need four PLS components. Which do you choose? Averaging is no solution—again, one would be using information which is not supposed to be available, and the resulting error

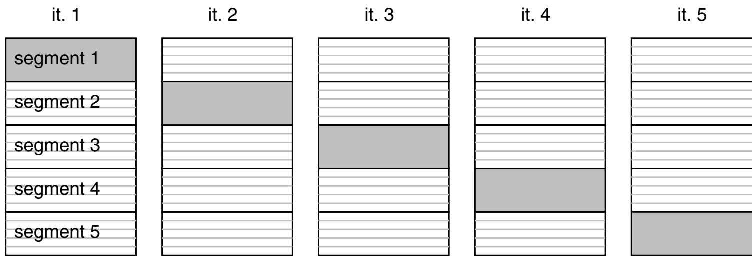

Fig. 9.2 Double crossvalidation: the inner CV loop, indicated by the gray horizontal lines, is used to estimate the optimal parameters for the modelling method. The outer loop, a five-fold crossvalidation, visualized by the gray rectangles, is used to estimate the prediction error

estimates would be biased. One approach is to use all optimal models simultaneously, and average the predictions (Smit et al. 2007). The disadvantage is that one loses the interpretation of one single model; however, this may be a reasonable price to pay. Other so-called ensemble methods will be treated in Sects. 9.7.1 and 9.7.2.

9.5 The Jackknife

Jackknifing (Efron and Tibshirani 1993) is the application of crossvalidation to obtain statistics other than error estimates, usually pertaining to model coefficients. The jackknife can be used to assess the bias and variance of regression coefficients. The jackknife estimate of bias, for example, is given by

$$\widehat{\text{Bias}}_{jck}(b) = (n-1)(\bar{b}_{(i)} - b) \tag{9.14}$$

where b is the regression coefficient[2] obtained with the full data, and $b_{(i)}$ is the coefficient from the data with sample i removed, just like in LOO crossvalidation. The bias estimate is simply the difference between the average of all these LOO estimates, and the full-sample estimate, multiplied by the factor $n-1$.

Let us check the bias of the PLS estimates on the gasoline data using two latent variables. The `plsr` function, when given the argument `jackknife = TRUE`,[3] is keeping all regression coefficients of a LOO crossvalidation in the `validation` element of the fitted object, so finding the bias estimates is not too difficult:

[2] In a multivariate setting we should use an index such as b_j—to avoid complicated notation we skip that for the moment.

[3] Information on this functionality can be found in the manual page of function `mvrCv`.

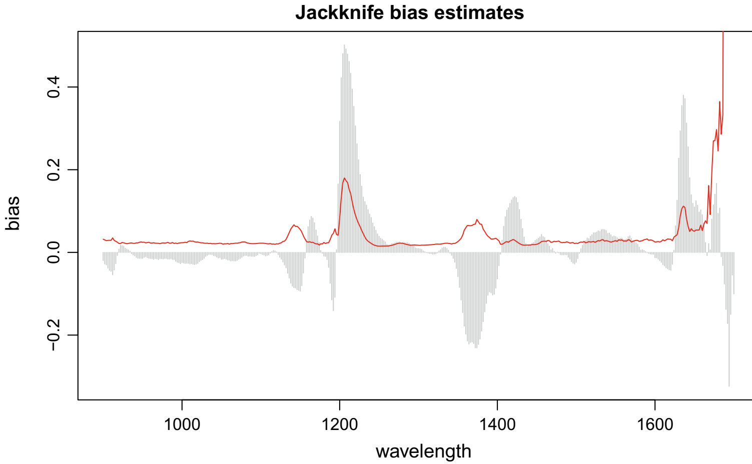

Fig. 9.3 Jackknife estimates of bias (in gray) and variance (red line) for a two-component PLS model on the gasoline data

```
> gasoline.pls <- plsr(octane ~ ., data = gasoline,
+                      validation = "LOO", ncomp = 2,
+                      jackknife = TRUE)
> n <- length(gasoline$octane)
> b.oob <- gasoline.pls$validation$coefficients[, , 2, ]
> bias.est <- (n-1) * (rowMeans(b.oob) - coef(gasoline.pls))
> plot(wavelengths, bias.est, xlab = "wavelength", ylab = "bias",
+      type = "h", main = "Jackknife bias estimates",
+      col = "gray")
```

The result is shown in Fig. 9.3—clearly, the bias for specific coefficients can be appreciable.

The jackknife estimate of variance is given by

$$\widehat{\mathrm{Var}}_{jck}(b) = \frac{n-1}{n} \sum (b_{(i)} - \bar{b}_{(i)})^2 \tag{9.15}$$

and is implemented in `var.jack`. Again, an object of class `mvr` needs to be supplied that is fitted with `jackknife = TRUE`:

```
> var.est <- var.jack(gasoline.pls)
> lines(wavelengths, var.est, col = "red")
```

The variance is shown as the red line in Fig. 9.3. In the most important regions, bias seems to dominate variance.

Several variants of the jackknife exist, including some where more than one sample is left out (Efron and Tibshirani 1993). In practice, however, the jackknife has been replaced by the more versatile bootstrap.

9.6 The Bootstrap

The bootstrap (Efron and Tibshirani 1993; Davison and Hinkley 1997) is a generalization of the ideas behind crossvalidation: again, the idea is to generate multiple data sets that, after analysis, shed light on the variability of the statistic of interest as a result of the different training set compositions. Rather than splitting up the data to obtain training and test sets, in *non-parametric bootstrapping* one generates a training set—a bootstrap sample—by sampling with replacement from the data. Whereas the measured data set is one possible realization of the underlying population, an individual bootstrap sample is, analogously, one realization from the complete set. Since we may have sufficient knowledge of difference between the complete set and the empirical realizations, simply by generating more bootstrap samples, we can study the distribution of the statistic of interest θ. In non-parametric bootstrapping applied to regression problems, there are two main approaches for generating a bootstrap sample. One is to sample (again, with replacement) from the *errors* of the initial model. Bootstrap samples are generated by adding the resampled errors to the original data. This strategy is appropriate when the $\boldsymbol{X}$ data can be regarded as fixed and the model is assumed to be correct. In other cases, one can sample complete cases, i.e., rows from the data matrix, to obtain a bootstrap sample. In such a bootstrap sample, some rows are present multiple times; others are absent.

In *parametric bootstrapping* on the other hand, one describes the data with a parametric distribution, from which then random bootstrap samples are generated. In the life sciences, high-dimensional data are the rule rather than the exception, and therefore any parametric description of a data set is apt to be based on very sparse data. Consequently, the parametric bootstrap has been less popular in this context.

What method is used to generate the bootstrap distribution, parametric boostrapping or non-parametric bootstrapping, is basically irrelevant for the subsequent analysis. Typically, several hundreds to thousands bootstrap samples are analyzed, and the variability of the statistic of interest is monitored. This enables one to make inferences, both with respect to estimating prediction errors and confidence intervals for model coefficients.

9.6.1 Error Estimation with the Bootstrap

Because a bootstrap sample will effectively never contain all samples in the data set, there are samples that have not been involved in building the model. These out-of-bag samples can conveniently be used in estimation of prediction errors. A popular estimator is the so-called 0.632 estimate $\hat{\varepsilon}_{0.632}$, given by

$$\hat{\varepsilon}^2_{0.632} = 0.368 \text{ MSEC} + 0.632\bar{\varepsilon}^2_B \tag{9.16}$$

where $\bar{\varepsilon}_B^2$ is the average squared prediction error of the OOB samples in the B bootstrap samples, and MSEC is the mean squared training error (on the complete data set). The factor $0.632 \approx (1 - e^{-1})$ is approximately the probability of a sample to end up in a bootstrap sample (Efron and Tibshirani 1993). In practice, the 0.632 estimator is the most popular form for estimating prediction errors; a more sophisticated version, correcting possible bias, is known as the 0.632+ estimator (Efron and Tibshirani 1997) but in many cases the difference is small.

As an example, let us use bootstrapping rather than crossvalidation to determine the optimal number of latent variables in PCR fitting of the gasoline data. In this case, the independent variables are not fixed, and there is some uncertainty on whether the model is correct. This leads to the adoption of the resampling cases paradigm. We start by defining bootstrap sample indices—in this case we take 500 bootstrap samples.

```
> B <- 500
> ngas <- nrow(gasoline)
> boot.indices <-
+   matrix(sample(1:ngas, ngas * B, replace = TRUE), ncol = B)
> sort(boot.indices[, 1])[1:20]
 [1]  2  2  3  3  4  6  8  8  8  8 11 12 14 15 15 16 17 19 20 21
```

Among others, objects 1 and 5 are absent from the first bootstrap sample, (partially) shown here as an example. Other samples, such as 2 and 3, occur multiple times. Similar behaviour is observed for the other 499 bootstrap samples. We now build a PCR model for each bootstrap sample and record the predictions of the out-of-bag objects. The following code is not particularly memory-efficient but easy to understand:

```
> npc <- 5
> predictions <- array(NA, c(ngas, npc, B))
> for (i in 1:B) {
+   gas.bootpcr <- pcr(octane ~ ., data = gasoline,
+                      ncomp = npc, subset = boot.indices[, i])
+   oobs <- (1:ngas)[-boot.indices[, i]]
+   predictions[oobs, , i] <-
+     predict(gas.bootpcr,
+             newdata = gasoline$NIR[oobs, ])[, 1, ]
+ }
```

Next, the OOB errors for the individual objects are calculated, and summarized in one estimate:

```
> diffs <- sweep(predictions, 1, gasoline$octane)
> sqerrors <- apply(diffs^2, c(1, 2), mean, na.rm = TRUE)
> sqrt(colMeans(sqerrors))
[1] 1.48695 1.50077 1.24562 0.28667 0.27598
```

Finally, the out-of-bag errors are combined with the calibration error to obtain the 0.632 estimate:

```
> gas.pcr <- pcr(octane ~ ., data = gasoline, ncomp = npc)
> RMSEP(gas.pcr, intercept = FALSE)
1 comps  2 comps  3 comps  4 comps  5 comps
 1.3656   1.3603   1.1097   0.2305   0.2260
> error.632 <- .368 * colMeans(gas.pcr$residuals^2) +
+   .632 * colMeans(sqerrors)
> sqrt(error.632)
       1 comps 2 comps 3 comps 4 comps 5 comps
octane  1.4435  1.4507  1.1974 0.26737 0.25873
```

The result is an upward correction of the too optimistic training set errors. We can compare the 0.632 estimate with the LOO and ten-fold crossvalidation estimates:

```
> gas.pcr.cv <- pcr(octane ~ ., data = gasoline, ncomp = npc,
+                   validation = "CV")
> gas.pcr.loo <- pcr(octane ~ ., data = gasoline, ncomp = npc,
+                    validation = "LOO")
> bp <- barplot(sqrt(error.632),
+               ylim = c(0, 1.6), col = "peachpuff")
> lines(bp, sqrt(c(gas.pcr.cv$validation$PRESS) / ngas),
+       col = 2, lwd = 2)
> lines(bp, sqrt(c(gas.pcr.loo$validation$PRESS) / ngas),
+       col = 4, lty = 2, lwd = 2)
> legend("topright", lty = 1:2, col = c(2, 4), lwd = 2,
+        legend = c("CV", "LOO"))
```

The result is shown in Fig. 9.4. The estimates in general agree very well—the differences that can be seen are the consequence of the stochastic nature of both ten-fold crossvalidation and bootstrapping: every time a slightly different result will be obtained.

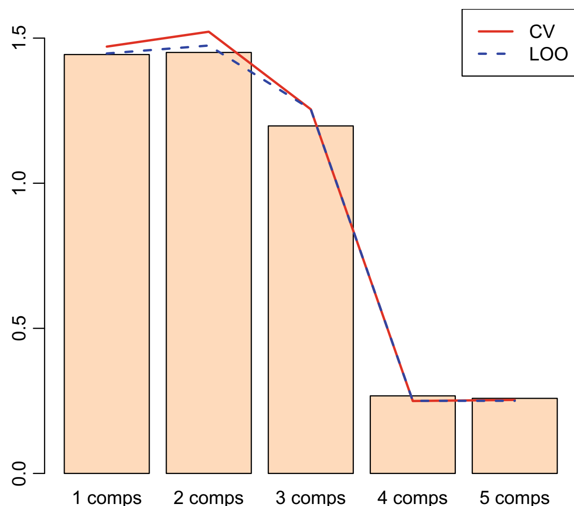

Fig. 9.4 Error estimates for PCR on the gasoline data: bars indicate the result of the 0.632 bootstrap, the solid line is the ten-fold crossvalidation, and the dashed line the LOO crossvalidation

It now should be clear what is the philosophy behind the 0.632 estimator. What it estimates, in fact, is the amount of optimism associated with the RMSEC value, $\hat{\omega}_{0.632}$:

$$\hat{\omega}_{0.632} = 0.632(\mathrm{MSEC} - \bar{\varepsilon}_B) \tag{9.17}$$

The original estimate is then corrected for this optimism:

$$\hat{\varepsilon}_{0.632} = \mathrm{MSEC} + \hat{\omega}_{0.632} \tag{9.18}$$

which leads to Eq. 9.16.

Several R packages are available that contain functions for bootstrapping. Perhaps the two best known ones are **bootstrap**, associated with Efron and Tibshirani (1993), and **boot**, written by Angelo Canty and implementing functions from Davison and Hinkley (1997). The former is a relatively simple package, maintained mostly to support Efron and Tibshirani (1993)—**boot**, a recommended package, is the primary general implementation of bootstrapping in R. The implementation of the 0.632 estimator using **boot** is done in a couple of steps (Davison and Hinkley 1997, p. 324). First, the bootstrap samples are generated, returning the statistic to be bootstrapped—in this case, the prediction errors[4]:

```
> gas.pcr.boot632 <-
+   boot(gasoline,
+        function(x, ind) {
+          mod <- pcr(octane ~ ., data = x,
+                     subset = ind, ncomp = 4)
+          gasoline$octane -
+            predict(mod, newdata = gasoline$NIR, ncomp = 4)},
+        R = 499)
```

The optimism is assessed by only considering the errors of the out-of-bag samples. For every bootstrap sample, we can find out which samples are constituting it using the `boot.array` function:

```
> dim(boot.array(gas.pcr.boot632))
[1] 499  60
> boot.array(gas.pcr.boot632)[1, 1:10]
 [1] 0 1 0 1 2 1 0 1 2 1
```

Just like when we did the resampling ourselves, some objects are absent from this bootstrap sample (here, as an example, using the first, only showing the first ten objects), and others are present multiple times. Averaging the squared errors of the OOB objects leads to the 0.632 estimate:

[4] In Davison and Hinkley (1997) and the corresponding **boot** package the number of bootstrap samples is typically a number like 499 or 999—the original sample then is added to the bootstrap set. Most other implementations use 500 and 1000. The differences are not very important in practice.

```
> in.bag <- boot.array(gas.pcr.boot632)
> oob.error <- mean((gas.pcr.boot632$t^2)[in.bag == 0])
> app.error <- MSEP(pcr(octane ~ ., data = gasoline, ncomp = 4),
+                    ncomp = 4, intercept = FALSE)
> sqrt(.368 * c(app.error$val) + .632 * oob.error)
[1] 0.26572
```

This error estimate is very similar to the four-fold crossvalidation result in Sect. 9.4.2 (0.2468). Note that it is not exactly equal to the 0.632 estimate in Sect. 9.6.1 (0.26737) because different bootstrap samples have been selected, but again the difference is small.

9.6.2 Confidence Intervals for Regression Coefficients

The bootstrap may also be used to assess the variability of a statistic such as an error estimate. A particularly important application in chemometrics is the standard error of a regression coefficient from a PCR or PLS model. Alternatively, confidence intervals can be built for the regression coefficients. No analytical solutions such as those for MLR exist in these cases; nevertheless, we would like to be able to say something about which coefficients are actually contributing to the regression model.

Typically, for an interval estimate such as a confidence interval, more bootstrap samples are needed than for a point estimate, such as an error estimate. Several hundred bootstrap samples are taken to be sufficient for point estimates; several thousand for confidence intervals. Taking smaller numbers may drastically increase the variability of the estimates, and with the current abundance of computing power there is rarely a case for being too economical.

The simplest possible approach is the *percentile* method: estimate the models for B bootstrap samples, and use the $B\alpha/2$ and $B(1-\alpha/2)$ values as the $(1-\alpha)$ confidence intervals. For the gasoline data, modelled with PCR using four PCs, these bootstrap regression coefficients are obtained by:

```
> B <- 1000
> ngas <- nrow(gasoline)
> boot.indices <-
+   matrix(sample(1:ngas, ngas * B, replace = TRUE), ncol = B)
> npc <- 4
> gas.pcr <- pcr(octane ~ ., data = gasoline, ncomp = npc)
> coefs <- matrix(0, ncol(gasoline$NIR), B)
> for (i in 1:B) {
+   gas.bootpcr <- pcr(octane ~ ., data = gasoline,
+                      ncomp = npc, subset = boot.indices[, i])
+   coefs[, i] <- c(coef(gas.bootpcr))
+ }
```

A plot of the area covered by the regression coefficients of all bootstrap samples is shown in Fig. 9.5:

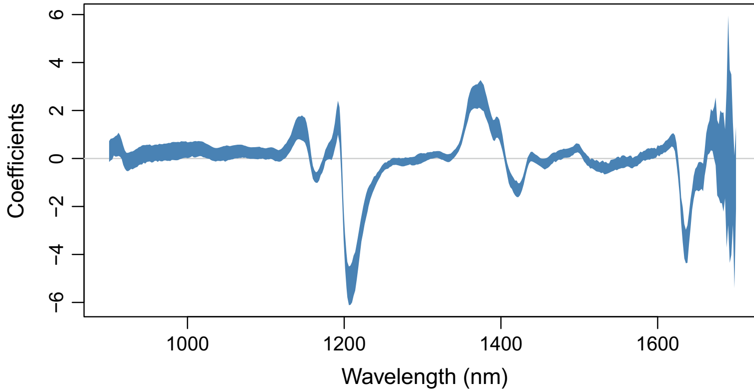

Fig. 9.5 Regression coefficients from all 1000 bootstrap samples for the gasoline data, using PCR with four latent variables

```
> matplot(wavelengths, coefs, type = "n",
+         ylab = "Coefficients", xlab = "Wavelength (nm)")
> abline(h = 0, col = "gray")
> polygon(c(wavelengths, rev(wavelengths)),
+         c(apply(coefs, 1, max), rev(apply(coefs, 1, min))),
+         col = "steelblue", border = NA)
```

Some of the wavelengths show considerable variation in their regression coefficients, especially the longer wavelengths above 1650 nm.

In the percentile method using 1000 bootstrap samples, the 95% confidence intervals are given by the 25th and 975th ordered values of each coefficient:

```
> coef.stats <- cbind(apply(coefs, 1, quantile, .025),
+                     apply(coefs, 1, quantile, .975))
> matplot(wavelengths, coef.stats, type = "n",
+         xlab = "Wavelength (nm)",
+         ylab = "Regression coefficient")
> abline(h = 0, col = "gray")
> polygon(c(wavelengths, rev(wavelengths)),
+         c(coef.stats[, 1], rev(coef.stats[, 2])),
+         col = "pink", border = NA)
> lines(wavelengths, c(coef(gas.pcr)))
```

The corresponding plot is shown in Fig. 9.6. Since the most extreme values will be removed by the percentile strategy, these CIs are more narrow than the area covered by the bootstrap coefficients from Fig. 9.5. Clearly, for most coefficients, zero is not in the confidence interval. A clear exception is seen in the longer wavelengths: there, the confidence intervals are very wide, indicating that this region contains very little relevant information.

The percentile method was the first attempt at deriving confidence intervals from bootstrap samples (Efron 1979) and has enjoyed huge popularity; however, one can show that the intervals are, in fact, incorrect. If the intervals are not symmetric (and it

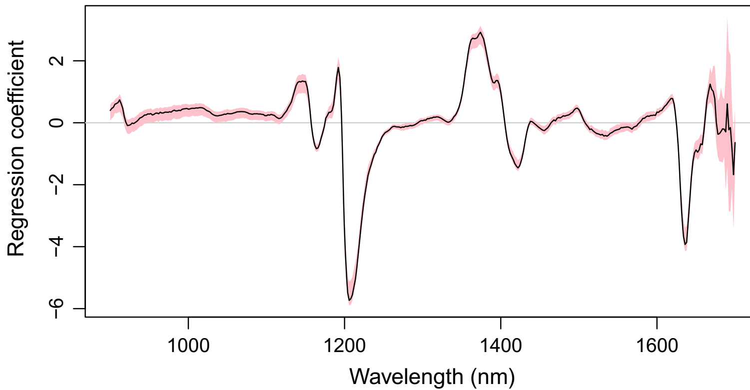

Fig. 9.6 Regression vector and 95% confidence intervals for the individual coefficients, for the PCR model of the gasoline data with four PCs. Confidence intervals are obtained with the bootstrap percentile method

can be seen in Fig. 9.6 that this is quite often the case—it is one of the big advantages of bootstrapping methods that they are able to define asymmetric intervals), it can be shown that the percentile method uses the skewness of the distribution the wrong way around (Efron and Tibshirani 1993). Better results are obtained by so-called *studentized* confidence intervals, in which the statistic of interest is given by

$$t_b = \frac{\hat{\theta}_b - \hat{\theta}}{\hat{\sigma}_b} \tag{9.19}$$

where $\hat{\theta}_b$ is the estimate for the statistic of interest, obtained from the bth bootstrap sample, $\hat{\sigma}_b$ is the standard deviation of that estimate, and $\hat{\theta}$ is the estimate obtained from the complete original data set. In the example of regression, $\hat{\theta}$ corresponds to the regression coefficient at a certain wavelength. Often, no analytical expression exists for $\hat{\sigma}_b$, and it should be obtained by other means, e.g., crossvalidation, or an inner bootstrap loop. Using the notation of $t_{B\alpha/2}$ as an approximation for the $\alpha/2$th quantile of the distribution of t_b, the studentized confidence intervals are given by

$$\hat{\theta} - t_{B(1-\alpha/2)} \leq \theta \leq \hat{\theta} - t_{B\alpha/2} \tag{9.20}$$

Several other ways of estimating confidence intervals exist, most notably the bias-corrected and accelerated (BCα) interval (Efron and Tibshirani 1993; Davison and Hinkley 1997).

The **boot** package provides the function `boot.ci`, which calculates several confidence interval estimates in one go. Again, first the bootstrap sampling is done and the statistics of interest are calculated:

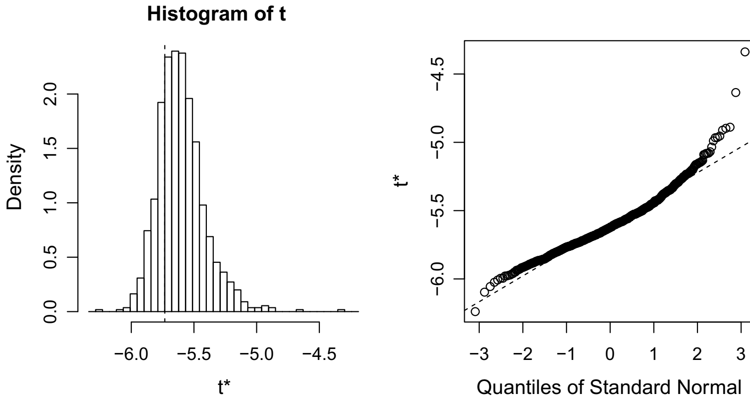

Fig. 9.7 Bootstrap plot for the regression coefficient at 1206 nm; in all bootstrap samples the coefficient is much smaller than zero

```
> gas.pcr.bootCI <-
+   boot(gasoline,
+        function(x, ind) {
+          c(coef(pcr(octane ~ ., data = x,
+                     ncomp = npc, subset = ind)))},
+        R = 999)
```

Here we use `R = 999` to conform to the setup of the **boot** package—the actual sample is seen as the 1000th element of the set. The regression coefficients are stored in the `gas.pcr.bootCI` object, which is of class `"boot"`, in the element named `t`:

```
> dim(gas.pcr.bootCI$t)
[1] 999 401
```

Plots of individual estimates can be made through the `index` argument:

```
> smallest <- which.min(gas.pcr.bootCI$t0)
> plot(gas.pcr.bootCI, index = smallest)
```

From the plot, shown in Fig. 9.7, one can see the distribution of the values for this coefficient in all bootstrap samples—the corresponding confidence interval will definitely not contain zero. The dashed line indicates the estimate based on the full data; these estimates are stored in the list element `t0`.

Confidence intervals for individual coefficients can be obtained from the `gas.pcr.bootCI` object as follows:

```
> boot.ci(gas.pcr.bootCI, index = smallest, type = c("perc", "bca"))
BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS
Based on 999 bootstrap replicates

CALL :
boot.ci(boot.out = gas.pcr.bootCI, type = c("perc", "bca"),
        index = smallest)

Intervals :
Level     Percentile            BCa
95%   (-5.909, -5.157 )   (-6.239, -5.538 )
Calculations and Intervals on Original Scale
Warning : BCa Intervals used Extreme Quantiles
Some BCa intervals may be unstable
```

The warning messages arise because in the extreme tails of the bootstrap distribution it is very difficult to make precise estimates—in such cases one really needs more bootstrap samples to obtain somewhat reliable estimates. Nevertheless, one can see that the intervals agree reasonably well; the BCα intervals are slightly shifted downward compared to the percentile intervals. For this coefficient, in absolute value the largest of the set, neither contains zero, as expected. In total, the percentile intervals show 318 cases where zero is not in the 95% confidence interval; the BCα intervals lead to 325 such cases.

It is interesting to repeat this exercise using a really large number of principal components, say twenty (remember, the gasoline data set only contains sixty samples). We would expect much more variation in the coefficients, since the model is more flexible and can adapt to changes in the training data much more easily. More variation means wider confidence intervals, and fewer "significant" cases, where zero is not included in the CI. Indeed, using twenty PCs leads to only 71 significant cases for the percentile intervals, and 115 for BCα (and an increased number of warning messages from the `boot` function as well).

9.6.3 Other R Packages for Bootstrapping

The bootstrap is such a versatile technique, that it has found application in many different areas of science. This has led to a large number of R packages implementing some form of the bootstrap—at the moment of writing, the package list of the CRAN repository contains already four other packages in between the packages **boot** and **bootstrap** already mentioned. To name just a couple of examples: package **FRB** contains functions for applying bootstrapping in robust statistics; **DAIM** provides functions for error estimation including the 0.632 and 0.632+ estimators. Using **EffectiveDose** it is possible to estimate the effects of a drug, and in particular to determine the effective dose level—bootstrapping is provided for the calculation of confidence intervals. Packages **meboot** and **BootPR** provide machinery for the application of bootstrapping in time series.

9.7 Integrated Modelling and Validation

Obtaining a good multivariate statistical model is hardly ever a matter of just loading the data and pushing a button: rather, it is a long and sometimes seemingly endless iteration of visualization, data treatment, modelling and validation. Since these aspects are so intertwined, it seems to make sense to develop methods that combine them in some way. In this section, we consider approaches that combine elements of model fitting with validation. The first case is bagging (Breiman 1996), where many models are fitted on bootstrap sets, and predictions are given by the average of the predictions of these models. At the same time, the out-of-bag samples can be used for obtaining an unbiased error estimate. Bagging is applicable to all classification and regression methods, but will give benefits only in certain cases; the classical example where it works well is given by trees (Breiman 1996)—see below. An extension of bagging, also applied to trees, is the technique of random forests (Breiman 2001). Finally, we will look at boosting (Freund and Schapire 1997), an iterative method for binary classification giving progressively more weight to misclassified samples. Bagging and boosting can be seen as meta-algorithms, because they consist of strategies that, in principle at least, can be combined with any model-fitting algorithm.

9.7.1 Bagging

The central idea behind bagging is simple: if you have a classifier (or a method for predicting continuous variables) that on average gives good predictions but has a somewhat high variability, it makes sense to average the predictions over a large number of applications of this classifier. The problem is how to do this in a sensible way: just repeating the same fit on the same data will not help. Breiman proposed to use bootstrapping to generate the variability that is needed. Training a classifier on every single bootstrap sets leads to an ensemble of models; combining the predictions of these models would then, in principle, be closer to the true answer. This combination of bootstrapping and aggregating is called bagging (Breiman 1996).

The package **ipred** implements bagging for classification, regression and survival analysis using trees—the **rpart** implementation is employed. For classification applications, also the combination of bagging with kNN is implemented (in function `ipredknn`). We will focus here on bagging trees. The basic function is `ipredbag`, while the function `bagging` provides the same functionality using a formula interface. Making a model for predicting the octane number for the gasoline data is very easy:

```
> (gasoline.bagging <- ipredbagg(gasoline$octane[gas.odd],
+                                gasoline$NIR[gas.odd, ],
+                                coob = TRUE))

Bagging regression trees with 25 bootstrap replications
Out-of-bag estimate of root mean squared error:  0.9181
```

The OOB error is quite high. Predictions for the even-numbered samples can be obtained by the usual `predict` function:

```
> gs.baggpreds <-
+   predict(gasoline.bagging, gasoline$NIR[gas.even, ])
> resids <- gs.baggpreds - gasoline$octane[gas.even]
> sqrt(mean(resids^2))
[1] 1.6738
```

This is not a very good result. Nevertheless, one should keep in mind that default settings are often suboptimal and some tweaking may lead to substantial improvements.

Doing classification with bagging is equally simple. Here, we show the example of discriminating between the `control` and `pca` classes of the prostate data, again using only the first 1000 variables as we did in Sect. 7.1.6.1:

```
> prost.bagging <- bagging(type ~ ., data = prost.df,
+                          subset = prost.odd)
> prost.baggingpred <- predict(prost.bagging,
+                              newdata = prost.df[prost.even, ])
> table(prost.type[prost.even], prost.baggingpred)
         prost.baggingpred
          control pca
  control      30  10
  pca           4  80
```

which doubles the number of misclassifications compared to the SVM solution in Sect. 7.4.1 but still is a lot better than the single-tree result.

So when does bagging improve things? Clearly, when a classification or regression procedure changes very little with different bootstrap samples, the result will be the same as the original predictions. It can be shown (Breiman 1996) that bagging is especially useful for predictors that are unstable, i.e., predictors that are highly adaptive to the composition of the data set. Examples are trees, neural networks (Hastie et al. 2001) or variable selection methods.

9.7.2 Random Forests

The combination of bagging and tree-based methods is a good one, as we saw in the last section. However, Breiman and Cutler saw that more improvement could be obtained by injecting extra variability into the procedure, and they proposed a number

of modifications leading to the technique called Random Forests (Breiman 2001). Again, bootstrapping is used to generate data sets that are used to train an ensemble of trees. One key element is that the trees are constrained to be very simple—only few nodes are allowed, and no pruning is applied. Moreover, at every split, only a subset of all variables is considered for use. Both adaptations force diversity into the ensemble, which is the key to why improvements can be obtained with aggregating.

It can be shown (Breiman 2001) that an upper bound for the generalization error is given by

$$\hat{E} \leq \bar{\rho}(1 - q^2)/q^2$$

where $\bar{\rho}$ is the average correlation between predictions of individual trees, and q is a measure of prediction quality. This means that the optimal gain is obtained when many good yet diverse classifiers are combined, something that is intuitively logical—there is not much point in averaging the outcomes of identical models, and combining truly bad models is unlikely to lead to good results either.

The R package **randomForest** provides a convenient interface to the original Fortran code of Breiman and Cutler. The basic function is `randomForest`, which either takes a formula or the usual combination of a data matrix and an outcome vector:

```
> wines.df <- data.frame(vint = vintages, wines)
> (wines.rf <- randomForest(vint ~ ., subset = wines.odd,
+                           data = wines.df))

Call:
 randomForest(formula = vint ~ ., data = wines.df, subset = wines.odd)
               Type of random forest: classification
                     Number of trees: 500
No. of variables tried at each split: 3

        OOB estimate of  error rate: 4.49%
Confusion matrix:
           Barbera Barolo Grignolino class.error
Barbera         24      0          0    0.000000
Barolo           0     28          1    0.034483
Grignolino       2      1         33    0.083333
```

The `print` method shows the result of the fit in terms of the error rate of the out-of-bag samples, in this case less than 5%. Because the algorithm fits trees to many different bootstrap samples, this error estimate comes for free. Prediction is done in the usual way:

```
> wines.rf.predict <-
+   predict(wines.rf, newdata = wines.df[wines.even, ])
> table(wines.rf.predict, vintages[wines.even])

wines.rf.predict Barbera Barolo Grignolino
      Barbera         24      0          0
      Barolo           0     29          0
      Grignolino       0      0         35
```

So prediction for the even rows in the data set is perfect here. Note that repeated training may lead to small differences because of the randomness involved in selecting bootstrap samples and variables in the training process. Also in many other applications random forests have shown very good predictive abilities (see, e.g., reference Svetnik et al. 2003 for an application in chemical modelling).

So it seems the most important disadvantage of tree-based methods, the generally low quality of the predictions, has been countered sufficiently. Does this come at a price? At first sight, yes. Not only does a random forest add complexity to the original algorithm in the form of tuning parameters, the interpretability suffers as well. Indeed, an ensemble of trees would seem more difficult to interpret than one simple sequence of yes/no questions. Yet in reality things are not so simple. The interpretability, one of the big advantages of trees, becomes less of an issue when one realizes that a slight change in the data may lead to a completely different tree, and therefore a completely different interpretation. Such a small change may, e.g., be formed by the difference between successive crossvalidation or bootstrap iterations—thus, the resulting error estimate may be formed by predictions from trees using different variables in completely different ways.

The technique of random forests addresses these issues in the following ways. A measure of the importance of a particular variable is obtained by comparing the out-of-bag errors for the trees in the ensemble with the out-of-bag errors when the values for that variable are permuted randomly. Differences are averaged over all trees, and divided by the standard error. If one variable shows a big difference, this means that the variable, in general, is important for the classification: the scrambled values lead to models with decreased predictivity. This approach can be used for both classification (using, e.g., classification error rate as a measure) and regression (using a value like MSE). An alternative is to consider the total increase in node purity.

In package **randomForest** this is implemented in the following way. When setting the parameter `importance = TRUE` in the call to `randomForest`, the importances of all variables are calculated during the fit—these are available through the extractor function `importance`, and for visualization using the function `varImpPlot`:

```
> wines.rf <- randomForest(vint ~ ., data = wines.df,
+                          importance = TRUE)
> varImpPlot(wines.rf)
```

The result is shown in Fig. 9.8. The left plot shows the importance measured using the mean decrease in accuracy; the right plot using the mean decrease in node impurity, as measured by the Gini index. Although there are small differences, the overall picture is the same using both indices.

The second disadvantage, the large number of parameters to set in using tree-based models, is implicitly taken care of in the definition of the algorithm: by requiring all trees in the forest to be small and simple, no elaborate pruning schemes are necessary, and the degrees of freedom of the fitting algorithm have been cut back drastically.

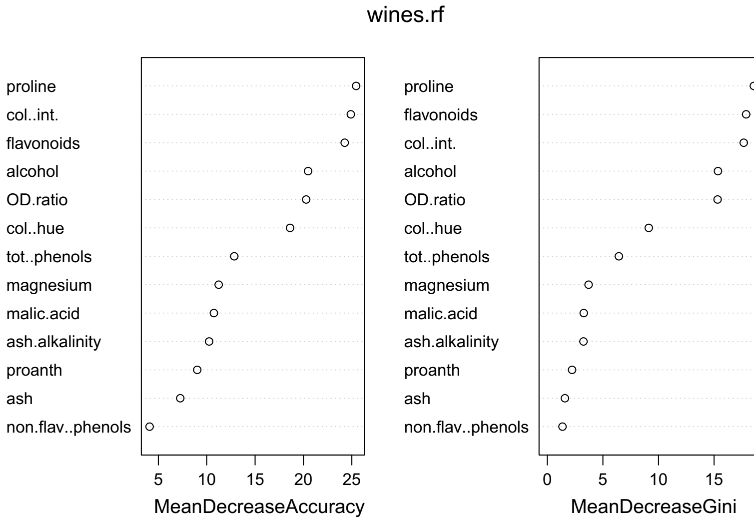

Fig. 9.8 Assessment of variable importance by random forests: the left plot shows the mean decrease in accuracy and the right the mean decrease in Gini index, both after permuting individual variable values

Furthermore, it appears that in practice random forests are very robust to changes in settings: averaging many trees also takes away a lot of the dependence on the exact value of parameters. In practice, the only parameter that is sometimes optimized is the number of trees (Efron and Hastie 2016), and even that usually has very little effect. This has caused random forests to be called one of the most powerful off-the-shelf classifiers available.

Just like the classification and regression trees seen in Sect. 7.3, random forests can also be used in a regression setting. Take the gasoline data, for instance: training a model using the default settings can be achieved with the following command.

```
> gasoline.rf <- randomForest(gasoline$NIR[gas.odd, ],
+                             gasoline$octane[gas.odd],
+                             importance = TRUE,
+                             xtest = gasoline$NIR[gas.even, ],
+                             ytest = gasoline$octane[gas.even])
```

For interpretation purposes, we have used the `importance = TRUE` argument, and we have provided the test samples at the same time. The results, shown in Fig. 9.9, are better than the ones from bagging:

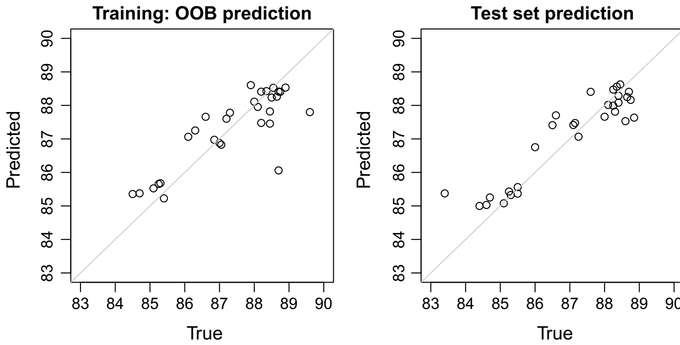

Fig. 9.9 Predictions for the gasoline data using random forests. Left plot: OOB predictions for the training data—right plot: test data

```
> pl.range <- c(83, 90)
> plot(gasoline$octane[gas.odd], gasoline.rf$predicted,
+      main = "Training: OOB prediction", xlab = "True",
+      ylab = "Predicted", xlim = pl.range, ylim = pl.range)
> abline(0, 1, col = "gray")
> plot(gasoline$octane[gas.even], gasoline.rf$test$predicted,
+      main = "Test set prediction", xlab = "True",
+      ylab = "Predicted", xlim = pl.range, ylim = pl.range)
> abline(0, 1, col = "gray")
```

However, there seems to be a bias towards the mean—the absolute values of the predictions at the extremes of the range are too small. Also the RMS values confirm that the test set predictions are much worse than the PLS and PCR estimates of 0.21:

```
> resids <- gasoline.rf$test$predicted - gasoline$octane[gas.even]
> sqrt(mean(resids^2))
[1] 0.63721
```

One of the reasons can be seen in the variable importance plot, shown in Fig. 9.10:

```
> rf.imps <- importance(gasoline.rf)
> plot(wavelengths, rf.imps[, 1] / max(rf.imps[, 1]),
+      type = "l", xlab = "Wavelength (nm)",
+      ylab = "Importance", col = "gray")
> lines(wavelengths, rf.imps[, 2] / max(rf.imps[, 2]), col = 2)
> legend("topright", legend = c("Error decrease", "Gini index"),
+        col = c("gray", "red"), lty = 1)
```

Both criteria are dominated by the wavelengths just above 1200 nm. Especially the Gini index leads to a sparse model, whereas the error-based importance values clearly are much more noisy. Interestingly, when applying random forests to the first derivative spectra of the gasoline data set (not shown) the same feature around 1200 nm is important, but the response at 1430 nm comes up as an additional feature.

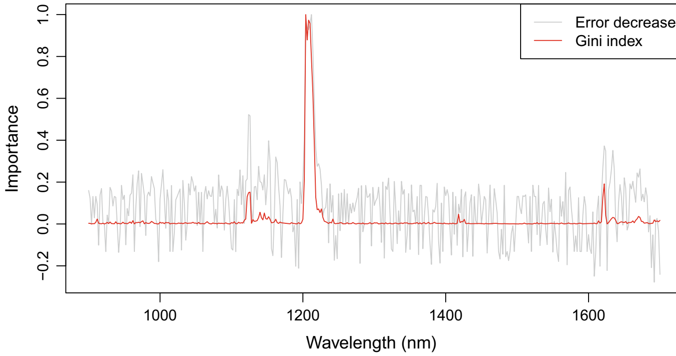

Fig. 9.10 Variable importance for modelling the gasoline data with random forests: basically, only the wavelengths just above 1200 nm seem to contribute

Although the predictions improve somewhat, they are still nowhere near the PLS and PCR results shown in Chap. 8.

For comparison, we also show the results of random forests on the prediction of the even samples in the prostate data set:

```
> prost.rf <-
+   randomForest(x = prost[prost.odd, ],
+                y = prost.type[prost.odd],
+                x.test = prost[prost.even, ],
+                y.test = prost.type[prost.even])
> prost.rfpred <- predict(prost.rf, newdata = prost[prost.even, ])
> table(prost.type[prost.even], prost.rfpred)
         prost.rfpred
          control pca
  control      30  10
  pca           4  80
```

Again, a slight improvement over bagging can be seen.

9.7.3 Boosting

In boosting (Freund and Schapire 1997), validation and classification are combined in a different way. Boosting, and in particular in the adaBoost algorithm that we will be focusing on in this section, is an iterative algorithm that in each iteration focuses the attention to misclassified samples from the previous step. Just as in bagging, in principle any modelling approach can be used; also similar to bagging, not all combinations will show improvements. Other forms of boosting have appeared since the original adaBoost algorithm, such as gradient boosting, popular in the statistics community (Friedman 2001; Efron and Hastie 2016). One of the most powerful

new variants is XGBoost (which stands for Extreme Gradient Boosting, Chen and Guestrin 2016), available in R through package **xgboost**.

The main idea of adaBoost is to use weights on the samples in the training set. Initially, these weights are all equal, but during the iterations the weights of incorrectly predicted samples increase. In adaBoost, which stands for adaptive boosting, the changes in the weight of object i is given by

$$D_{t+1}(i) = \frac{D_t(i)}{Z_t} \times \begin{cases} e^{-\alpha_t} & \text{if correct} \\ e^{\alpha_t} & \text{if incorrect} \end{cases} \tag{9.21}$$

where Z_t is a suitable normalization factor, and α_t is given by

$$\alpha_t = 0.5 \ln \left(\frac{1 - \epsilon_t}{\epsilon_t} \right) \tag{9.22}$$

with ϵ_t the error rate of the model at iteration t. In prediction, the final classification result is given by the weighted average of the T predictions during the iterations, with the weights given by the α values.

The algorithm itself is very simple and easily implemented. The only parameter that needs to be set in an application of boosting is the maximal number of iterations. A number that is too large would potentially lead to overfitting, although in many cases it has been observed that overfitting does not occur (see, e.g., references in Freund and Schapire 1997).

Boosting trees in R is available in package **ada** (Michailides et al. 2006), which directly follows the algorithms described in reference (Friedman et al. 2000). Let us revisit the prostate example, also tackled with SVMs (Sect. 7.4.1):

```
> prost.ada <- ada(type ~ ., data = prost.df, subset = prost.odd)
> prost.adapred <-
+   predict(prost.ada, newdata = prost.df[prost.even, ])
> table(prost.type[prost.even], prost.adapred)
         prost.adapred
          control pca
  control      30  10
  pca           3  81
```

The result is equal to the one obtained with bagging. The development of the errors in training and test sets can be visualized using the default `plot` command. In this case, we should add the test set to the `ada` object first[5]:

```
> prost.ada <- addtest(prost.ada,
+                      prost.df[prost.even, ],
+                      prost.type[prost.even])
> plot(prost.ada, test = TRUE)
```

[5] We could have added the test set data to the original call to `ada` as well—see the manual page.

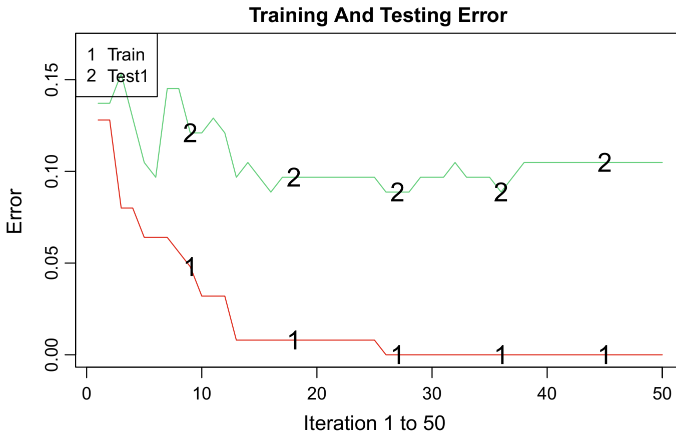

Fig. 9.11 Development of prediction errors for training and test sets of the prostate data (two classes, only 1000 variables) using `ada`

This leads to the plot in Fig. 9.11. The final error on the test set is less than half of the error at the beginning of the iterations. Clearly, both the training and testing errors have stabilized already after some twenty iterations.

The version of boosting employed in this example is also known as *Discrete adaboost* (Friedman et al. 2000; Hastie et al. 2001), since it returns 0/1 class predictions. Several other variants have been proposed, returning membership probabilities rather than crisp classifications and employing different loss functions. In many cases they outperform the original algorithm (Friedman et al. 2000).

Since boosting is in essence a binary classifier, special measures must be taken to apply it in a multi-class setting, similar to the possibilities mentioned in Sect. 7.4.1.1. A further interesting connection with SVMs can be made (Schapire et al. 1998): although boosting does not explicitly maximize margins, as SVMs do, it does come very close. The differences are, firstly, that SVMs use the L_2 norm, the sum of the squared vector elements, whereas boosting uses L_1 (the sum of the absolute values) and L_∞ (the largest value) norms for the weight and instance vectors, respectively. Secondly, boosting employs greedy search rather than kernels to address the problem of finding discriminating directions in high-dimensional space. The result is that although there are intimate connections, in many cases the models of boosting and SVMs can be quite different.

The obvious drawback of focusing more and more on misclassifications is that these may be misclassifications with a reason: outlying observations, or samples with wrong labels, may disturb the modelling to a large extent. Indeed, boosting has been proposed as a way to detect outliers.