Chapter 7
Classification

The goal of classification, also known as supervised pattern recognition, is to provide a model that yields the optimal discrimination between several classes in terms of predictive performance. It is closely related to clustering. The difference is that in classification it is clear what to look for: the number of classes is known, and the classes themselves are well-defined, usually by means of a set of examples, the training set. Labels of objects in the training set are generally taken to be error-free, and are typically obtained from information other than the data we are going to use in the model. For instance, one may have data—say, concentration levels of several hundreds of proteins in blood—from two groups of people, healthy, and not-so-healthy, and the aim is to obtain a classification model that distinguishes between the two states on the basis of the protein levels. The diagnosis may have been based on symptoms, medical tests, family history and subjective reasoning of the doctor treating the patient. It may not be possible to distinguish patients from healthy controls on the basis of protein levels, but if one would be able to, it would lead to a simple and objective test.

Apart from having good predictive abilities, an ideal classification method also provides insight in what distinguishes different classes from each other—which variable is associated with an observed effect? Is the association positive or negative? Especially in the natural sciences, this has become an important objective: a gene, protein or metabolite, characteristic for one or several classes, is often called a *biomarker*. Such a biomarker, or more often, set of biomarkers, can be used as an easy and reliable diagnostic tool, but also can provide insight or even opportunities for intervention in the underlying biological processes. Unfortunately, biomarker identification can be extremely difficult. First of all, in cases where the number of variables exceeds the number of cases, it is quite likely that several (combinations of) variables show high correlations with class labels even though there may not be causal relationships. Furthermore, there is a trend towards more complex non-linear modelling methods (often indicated with terms like Machine Learning or Artificial Intelligence) where

R. Wehrens, *Chemometrics with R*, Use R!,
https://doi.org/10.1007/978-3-662-62027-4_7

the relationship between variables and outcome can no longer be summarized in simple coefficient values. Hence, interpretation of such models is often impossible.

What is needed as well is a reliable estimate of the success rate of the classifier. In particular, one would like to know how the classifier will perform in the future, on new samples, of course comparable to the ones used in setting up the model. This error estimate is obtained in a validation step—Chap. 9 provides an overview of several different methods. These are all the more important when the classifier of interest has tunable parameters. These parameters are usually optimized on the basis of estimated prediction errors, but as a result the error estimates are positively biased, and a second validation layer is needed to obtain an unbiased error estimate. In this chapter, we will take a simple approach and divide the data in a representative part that is used for building the model (the training set), and an independent part used for testing (the test set). The phrase "independent" is of utmost importance: if, e.g., autoscaling is applied, one should use the column means and standard deviations of the training set to scale the test set. First scaling the complete data set and then dividing the data in training and test sets is, in a way, cheating: one has used information from the test data in the scaling. This usually leads to underestimates of prediction error.

That the training data should be representative seems almost trivial, but in some cases this is hard to achieve. Usually, a random division works well, but also other divisions may be used. In Chap. 4 we have seen that the odd rows of the `wine` data set are very similar to the even rows. In a classification context, we can therefore use the even rows as a training set and the odd rows as a test set:

```
> wines.odd <- seq(1, nrow(wines), by = 2)
> wines.even <- seq(2, nrow(wines), by = 2)
> wines.trn <- wines[wines.odd, ]
> wines.tst <- wines[wines.even, ]
> vint.trn <- vintages[wines.odd]
> vint.tst <- vintages[wines.even]
```

Note that classes are represented proportional to their frequency in the original data in both the training set and the test set. In a couple of cases we will illustrate methods in two dimensions only, looking at the `flavonoids` and `proline` variables in the wine data set:

```
> wines2.trn <- wines.trn[, c(7, 13)]
> wines2.tst <- wines.tst[, c(7, 13)]
```

There are many different ways of using the training data to predict class labels for future data. Discriminant analysis methods use a parametric description of means and covariances. Essentially, observations are assigned to the class having the highest probability density. Nearest-neighbor methods, on the other hand, focus on similarities with individual objects and assign objects to the class that is prevalent in the neighborhood; another way to look at it is to see nearest-neighbor methods as local density estimators. Similarities between objects can also be used directly, e.g., in kernel methods; the most well-known representative of this type of methods is Support Vector Machines (SVMs). A completely different category of classifiers is formed

by tree-based approaches. These create a model consisting of a series of binary decisions. Finally, neural-network based classification will be discussed.

Here we will concentrate on the main concepts and show how they should be implemented using standard approaches. Often these are directly supported by modelling methods themselves. An alternative is to use the **caret** package (short for Classification and Regression Training, Kuhn 2008; Kuhn and Johnson 2013), which provides tools for data splitting and validation in the contexts of classification and regression, but also many other topics mentioned in this book. The manual pages and the vignette of the **caret** package provide more information.

7.1 Discriminant Analysis

In discriminant analysis, one assumes normal distributions for the individual classes: $N_p(\mu_k, \Sigma_k)$, where the subscript p indicates that the data are p-dimensional (McLachlan 2004). One can then classify a new object, which can be seen as a point in p-dimensional space, to the class that has the highest probability density ("likelihood") at that point—this type of discriminant analysis is therefore indicated with the term "Maximum-Likelihood" (ML) discriminant analysis.

Consider the following univariate example with two groups (Mardia et al. 1979): group one is $N(0, 5)$ and group 2 is $N(1, 1)$. The likelihoods of classes i are given by

$$L_i\left(x; \mu_i, \sigma_i\right) = \frac{1}{\sigma_i\sqrt{2\pi}} \exp\left[-\frac{(x-\mu_i)^2}{2\sigma_i^2}\right] \tag{7.1}$$

It is not too difficult to show that $L_1 > L_2$ if

$$\frac{12}{25}x^2 - x + 1/2 - \ln 5 > 0$$

which in this case corresponds to the regions outside the interval $[-0.9, 2.9]$. In more general terms, one can show (Mardia et al. 1979) that for one-dimensional data $L_1 > L_2$ when

$$x^2\left(\frac{1}{\sigma_1^2} - \frac{1}{\sigma_2^2}\right) - 2x\left(\frac{\mu_1}{\sigma_1^2} - \frac{\mu_2}{\sigma_2^2}\right) + \left(\frac{\mu_1^2}{\sigma_1^2} - \frac{\mu_2^2}{\sigma_2^2}\right) < 2\ln\frac{\sigma_2}{\sigma_1} \tag{7.2}$$

This unrestricted form, where every class is individually described with a mean vector and covariance matrix, leads to quadratic class boundaries, and is called "Quadratic Discriminant Analysis" (QDA). Obviously, when $\sigma_1 = \sigma_2$ the quadratic term disappears, and we are left with a linear class boundary—"Linear Discriminant Analysis" (LDA). Both techniques will be treated in more detail below.

Another way of describing the same classification rules is to make use of the Mahalanobis distance:

$$d(\boldsymbol{x}, i) = \left(\boldsymbol{x} - \boldsymbol{\mu}_i\right)^T \boldsymbol{\Sigma}_i^{-1} \left(\boldsymbol{x} - \boldsymbol{\mu}_i\right) \tag{7.3}$$

Loosely speaking, this expresses the distance of an object to a class center in terms of the standard deviation in that particular direction. Thus, a sample $\boldsymbol{x}$ is simply assigned to the closest class, using the Mahalanobis metric $d(\boldsymbol{x}, i)$. In LDA, all classes are assumed to have the same covariance matrix $\boldsymbol{\Sigma}$, whereas in QDA every class is represented by its own covariance matrix $\boldsymbol{\Sigma}_i$.

7.1.1 Linear Discriminant Analysis

It is easy to show that Eq. 7.2 in the case of two groups with equal variances reduces to

$$|\boldsymbol{x} - \boldsymbol{\mu}_2| > |\boldsymbol{x} - \boldsymbol{\mu}_1| \tag{7.4}$$

Each observation $\boldsymbol{x}$ will be assigned to class 1 when it is closer to the mean of class 1 than of class 2, something that makes sense intuitively as well. Another way to write this is

$$\boldsymbol{\alpha}^T (\boldsymbol{x} - \boldsymbol{\mu}) > 0 \tag{7.5}$$

with

$$\boldsymbol{\alpha} = \boldsymbol{\Sigma}^{-1}(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2) \tag{7.6}$$

$$\boldsymbol{\mu} = (\boldsymbol{\mu}_1 + \boldsymbol{\mu}_2)/2 \tag{7.7}$$

This formulation clearly shows the linearity of the class boundaries. The separating hyperplane passes through the midpoint between the cluster centers, but is not necessarily perpendicular to the segment connecting the two centers.

In reality, of course, one does not know the true means $\boldsymbol{\mu}_i$ and the true covariance matrix $\boldsymbol{\Sigma}$. One then uses the plugin estimate $\boldsymbol{S}$, the estimated covariance matrix.[1] In LDA, it is obtained by pooling the individual covariance matrices $\boldsymbol{S}_i$:

$$\boldsymbol{S} = \frac{1}{n - G} \sum_{i=1}^{G} n_i \boldsymbol{S}_i \tag{7.8}$$

where there are G groups, n_i is the number of objects in group i, and the total number of objects is n.

[1] In statistics this is known as the *sample covariance matrix*.

For the wine data, this can be achieved as follows:

```
> wines.counts <- table(vint.trn)
> ngroups <- length(wines.counts)
> wines.groups <- split(as.data.frame(wines.trn), vint.trn)
> wines.covmats <- lapply(wines.groups, cov)
> wines.wcovmats <- mapply('*', wines.covmats, wines.counts,
+                          SIMPLIFY = FALSE)
> wines.pooledcov <-
+   Reduce("+", wines.wcovmats) / (nrow(wines.trn) - ngroups)
```

This piece of code illustrates a convenient feature of the `lapply` function: when the first argument is a vector, it can be used as an index for a function taking also other arguments—here, a list and a vector. Each of the three covariance matrices is multiplied by a weight corresponding to the number of objects in that class. In the final step, the `Reduce` function adds the three weighted covariance matrices. An alternative is to use a plain and simple loop:

```
> wines.pooledcov2 <- matrix(0, ncol(wines), ncol(wines))
> for (i in 1:3) {
+   wines.pooledcov2 <- wines.pooledcov2 +
+     cov(wines.groups[[i]]) * nrow(wines.groups[[i]])
+ }
> wines.pooledcov2 <-
+   wines.pooledcov2 / (nrow(wines.trn) - ngroups)


> range(wines.pooledcov2 - wines.pooledcov)
[1] 0 0
```

The number of parameters that must be estimated in LDA is relatively small: the pooled covariance matrix contains $p(p+1)/2$ numbers, and each cluster center p parameters. For G groups this leads to a total of $Gp + p(p+1)/2$ estimates—for the wine data, with three groups and thirteen variables, this implies 130 estimates.

The LDA classification itself is now easily performed: first we calculate the Mahalanobis distances (using the `mahalanobis` function) to the three class centers using the pooled covariance matrix, and then we determine which of these three is closest for every sample in the training set:

```
> distances <-
+   sapply(1:ngroups,
+          function(i, samples, means, covs)
+            mahalanobis(samples, colMeans(means[[i]]), covs),
+          wines.trn, wines.groups, wines.pooledcov)
> trn.pred <- apply(distances, 1, which.min)
```

Let's compare our predictions with the vintages of the odd-numbered samples:

```
> table(vint.trn, trn.pred)
             trn.pred
vint.trn       1  2  3
  Barbera     24  0  0
  Barolo       0 29  0
  Grignolino   0  0 36
```

The reproduction of the training data is perfect, much better than we have seen with clustering, which is not surprising since the LDA builds the model (in this case the pooled covariance matrix) using the information from the training set with the explicit aim of discriminating between the classes. However, we should not think that future observations are predicted with equal success. The test data should give an indication of what to expect:

```
> distances <-
+   sapply(1:ngroups,
+          function(i, samples, means, covs)
+            mahalanobis(samples, colMeans(means[[i]]), covs),
+          wines.tst, wines.groups, wines.pooledcov)
> tst.pred <- apply(distances, 1, which.min)
> table(vint.tst, tst.pred)
             tst.pred
vint.tst       1  2  3
  Barbera     24  0  0
  Barolo       0 29  0
  Grignolino   1  0 34
```

One Grignolino sample has been classified in the class of the Barbera samples—a very good result, confirming that the problem is not very difficult. Nevertheless, the difference with the unsupervised clustering approaches is obvious.

Of course, R already contains an `lda` function (in package **MASS**):

```
> wines.ldamod <- lda(wines.trn, grouping = vint.trn,
+                     prior = rep(1, 3)/3)
> wines.lda.testpred <- predict(wines.ldamod, new = wines.tst)
> table(vint.tst, wines.lda.testpred$class)

vint.tst     Barbera Barolo Grignolino
  Barbera         24      0          0
  Barolo           0     29          0
  Grignolino       1      0         34
```

The `prior = rep(1, 3)/3` argument in the `lda` function is used to indicate that all three classes are equally likely a priori. In many cases it makes sense to incorporate information about prior probabilities. Some classes may be more common than others, for example. This is usually reflected in the class sizes in the training set and therefore is taken into account when calculating the pooled covariance matrix, but it is not explicitly used in the discrimination rule. However, it is relatively simple to do

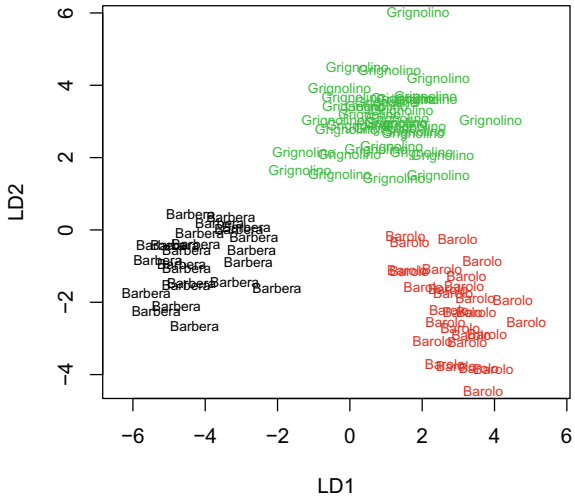

Fig. 7.1 Projection of the training data from the wine data set in the linear discriminant space. It is easy to see that linear class boundaries can be drawn so that all training objects are classified correctly

so: instead of maximising L_i one now maximises $\pi_i L_i$, where π_i is an estimate of the prior probability of class i. In the two-group case, this has the effect of shifting the critical value of the discriminant function with an amount of $\log(\pi_2/\pi_1)$ in Eq. 7.5. This approach is sometimes referred to as the Bayesian discriminant rule, and is the default behaviour of the `lda` function. Obviously, when all prior probabilities are equal, the Bayesian and ML discriminant rules coincide. Also in the example above, using the relative frequencies as prior probabilities would not have made any difference to the predictions—the three vintages have approximately equal class sizes.

The `lda` function comes with the usual supporting functions for printing and plotting. An example of what the plotting function provides is shown in Fig. 7.1:

```
> plot(wines.ldamod, col = as.integer(vint.trn))
```

The training samples are projected in the space of two new variables, the Linear Discriminants (LDs). In comparison to the PCA scoreplot from Fig. 4.1, class separation has clearly increased. Again, this is the result of the way in which the LDs have been chosen: whereas the PCs in PCA account for as much variance as possible, in LDA the LDs maximize separation. This will be even more clear when we view LDA in the formulation by Fisher (see Sect. 7.1.3).

One particularly attractive feature of the `lda` function as it is implemented in **MASS** is the possibility to choose different estimators of means and covariances. In particular, the arguments `method = "mve"` and `method = "t"` are interesting as they provide robust estimates.

7.1.2 Crossvalidation

When the number of samples is low, there are two important disadvantages of dividing a data set into two parts, one for training and one for testing. The first is that with small test sets, the error estimates are on a very rough scale: if there are ten samples in the test set, the errors are always multiples of ten percent. Secondly, the quality of the model will be lower than it can be: when building the classification model one needs all information one can get, and leaving out a significant portion of the data in general is not helpful. Only with large sets, consisting of, say, tens or hundreds of objects per class, it is possible to create training and test sets in such a way that modelling power will suffer very little while giving a reasonably precise error estimate. Even then, there is another argument against a division into training and test sets: such a division is random, and different divisions will lead to different estimates of prediction error. The differences may not be large, but in some cases they can be important, especially in the case of outliers and/or extremely unlucky divisions.

One solution would be to try a large number of random divisions and to average the resulting estimates. This is indeed a valid strategy—we will come back to this in Chap. 9. A very popular formalization of this principle is called *crossvalidation* (Stone 1974). The general procedure is as follows: one leaves out a certain part of the data, trains the classifier on the remainder, and uses the left-out bit—sometimes called the out-of-bag, or OOB, samples—to estimate the error. Next, the two data sets are joined again, and a new test set is split off. This continues until all objects have been left out exactly once. The crossvalidation error in classification is simply the number of misclassified objects divided by the total number of objects in the training set. If the size of the test set equals one, every sample is left out in turn—the procedure has received the name Leave-One-Out (LOO) crossvalidation. It is shown to be unbiased but can have appreciable variance: on average, the estimate is correct, but individual components may deviate considerably.

More stable results are usually obtained by leaving out a larger fraction, e.g., 10% of the data; such a crossvalidation is known as ten-fold crossvalidation. The largest errors cancel out (to some extent) so that the variance decreases; however, one pays the price of a small bias because of the size difference of the real training set and the training set used in the crossvalidation (Efron and Tibshirani 1993). In general, the pros outweigh the cons, so that this procedure is quite often applied. It also leads to significant speed improvements for larger data sets, although for the simple techniques presented in this chapter it is not likely to be very important. The whole crossvalidation procedure is illustrated in Fig. 7.2.

For LDA (and also QDA), it is possible to obtain the LOO crossvalidation result without complete refitting—upon leaving out one object, one can update the Mahalanobis distances of objects to class means and derive the classifications of the left-out samples quickly, without doing expensive matrix operations (Ripley 1996). The `lda` function returns crossvalidated predictions in the list element `class` when given the argument `CV = TRUE`:

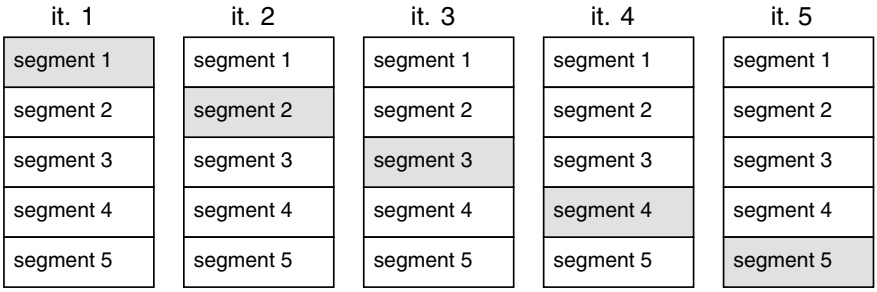

Fig. 7.2 Illustration of crossvalidation; in the first iteration, segment 1 of the data is left out during training and used as a test set. Every segment in turn is left out. From the prediction errors of the left-out samples the overall crossvalidated error estimate is obtained

```
> wines.ldamod <- lda(wines.trn, grouping = vint.trn,
+                     prior = rep(1, 3)/3, CV = TRUE)
> table(vint.trn, wines.ldamod$class)

vint.trn     Barbera Barolo Grignolino
  Barbera         24      0          0
  Barolo           0     28          1
  Grignolino       1      0         35
```

So, where the training set can be predicted without any errors, LOO crossvalidation on the training set leads to an estimated error percentage of $2/89 = 2.25\%$, twice the error on the test set. This difference in itself is not very alarming—error estimates also have variance.

7.1.3 Fisher LDA

A seemingly different approach to discriminant analysis is taken in Fisher LDA, named after its inventor, Sir Ronald Aylmer Fisher. Rather than assuming a particular distribution for individual clusters, Fisher devised a way to find a sensible rule to discriminate between classes by looking for a linear combination of variables $\boldsymbol{a}$ maximizing the ratio of the between-groups sums of squares $\boldsymbol{B}$ and the within-groups sums of squares $\boldsymbol{W}$ (Fisher 1936):

$$\boldsymbol{a}^T \boldsymbol{B} \boldsymbol{a} / \boldsymbol{a}^T \boldsymbol{W} \boldsymbol{a} \tag{7.9}$$

These sums of squares are calculated by

$$\boldsymbol{W} = \sum_{i=1}^{G} \overset{\sim}{\boldsymbol{X}}_i^T \overset{\sim}{\boldsymbol{X}}_i \tag{7.10}$$

$$\boldsymbol{B} = \sum_{i=1}^{G} n_i (\bar{\boldsymbol{x}}_i - \bar{\boldsymbol{x}})(\bar{\boldsymbol{x}}_i - \bar{\boldsymbol{x}})^T \tag{7.11}$$

where $\overset{\sim}{\boldsymbol{X}}_i$ is the mean-centered part of the data matrix containing objects of class i, and $\bar{\boldsymbol{x}}_i$ and $\bar{\boldsymbol{x}}$ are the mean vectors for class i and the whole data matrix, respectively. Put differently: $\boldsymbol{W}$ is the variation *around* the class centers, and $\boldsymbol{B}$ is the variation of the class centers around the global mean. It also holds that the total variance $\boldsymbol{T}$ is the sum of the between and within-groups variances:

$$\boldsymbol{T} = \boldsymbol{B} + \boldsymbol{W} \tag{7.12}$$

Fisher's criterion is equivalent to finding a linear combination of variables $\boldsymbol{a}$ corresponding to the subspace in which distances between classes are large and distances within classes are small—compact classes with a large separation. It can be shown that maximizing Eq. 7.9 leads to an eigenvalue problem, and that the solution $\boldsymbol{a}$ is given by the eigenvector of $\boldsymbol{B}\boldsymbol{W}^{-1}$ corresponding with the largest eigenvalue. An object $\boldsymbol{x}$ is then assigned to the closest class, i, which means that for all classes $i \neq j$ the following inequality holds:

$$|\boldsymbol{a}^T\boldsymbol{x} - \boldsymbol{a}^T\bar{\boldsymbol{x}}_i| < |\boldsymbol{a}^T\boldsymbol{x} - \boldsymbol{a}^T\bar{\boldsymbol{x}}_j| \tag{7.13}$$

Interestingly, although Fisher took a completely different starting point and did not explicitly assume normality or equal covariances, in the two-group case Fisher LDA leads to exactly the same solution as ML-LDA. Consider the discrimination between Barbera and Grignolino wines:

```
> wns <- wines[vintages != "Barolo", c(7, 13)]
> vnt <- factor(vintages[vintages != "Barolo"])
> wines.odd2 <- seq(1, nrow(wns), by = 2)
> wines.even2 <- seq(2, nrow(wns), by = 2)
```

To enable easy visualization, we will restrict ourselves to only two variables, `flavonoids` and `proline`. Fisher LDA is performed by the following code:

```
> wines.counts <- table(vnt)
> wines.groups <- split(as.data.frame(wns), vnt)
> WSS <-
+   Reduce("+", lapply(wines.groups,
+                      function(x) {
+                        crossprod(scale(x, scale = FALSE))}))
> BSS <-
+   Reduce("+", lapply(wines.groups,
+                      function(x, y) {
+                        nrow(x) * tcrossprod(colMeans(x) - y)},
+                      colMeans(wns)))
> FLDA <- eigen(solve(WSS, BSS))$vectors[, 1]
> FLDA / FLDA[1]
[1]  1.00000000 -0.00087649
```

Application of ML-LDA, Eq. 7.5, leads to

```
> wines.covmats <- lapply(wines.groups, cov)
> wines.wcovmats <- lapply(1:length(wines.groups),
+                          function(i, x, y) x[[i]]*y[i],
+                          wines.covmats, wines.counts)
> wines.pcov12 <- Reduce("+", wines.wcovmats) / (length(vnt) - 2)
> MLLDA <-
+   solve(wines.pcov12,
+         apply(sapply(wines.groups, colMeans), 1, diff))
> MLLDA / MLLDA[1]
 flavonoids     proline
 1.00000000 -0.00087476
```

Setting the first element of the discrimination functions equal to 1 makes the comparison easier: the vector $\boldsymbol{a}$ in Eq. 7.9 can be rescaled without any effect on both allocation rules Eqs. 7.13 and 7.5. In the two-group case, both ML-LDA and Fisher-LDA lead to the same discrimination function.

For problems with more than two groups, the results are different unless the sample means are collinear: Fisher LDA aims at finding *one* direction discriminating between the classes. An example is shown in Fig. 7.3, where the boundaries between the three classes in the two-dimensional subset of the wine data are shown for Fisher LDA and ML-LDA.

The Fisher LDA boundaries for more than two classes are produced by code essentially identical to the code for the two-group case earlier in this section: one should replace the lines

```
> wns <- wines[vintages != "Barolo", c(7, 13)]
> vnt <- factor(vintages[vintages != "Barolo"])
```

by

```
> wns <- wines2.trn
> vnt <- vint.trn
```

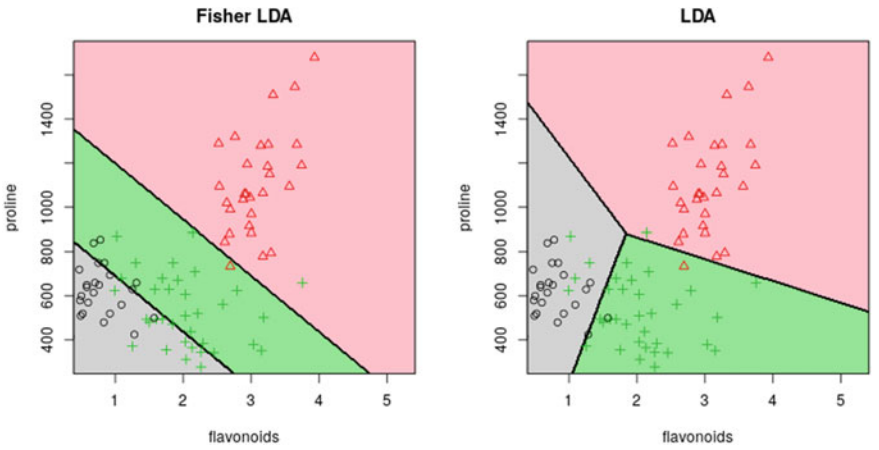

Fig. 7.3 Class boundaries for the wine data (proline and flavonoids only) for Fisher LDA (left) and ML-LDA (right). Models are created using the odd rows of the wine data (training data); the plotting symbols indicate the even rows (test data), as mentioned in the text

Then, after calculating the discriminant function `FLDA`, predictions are made at positions in a regular grid:

```
> xcoo <- seq(.4, 5.4, length = 251)
> ycoo <- seq(250, 1750, length = 251)
> gridXY <- data.matrix(expand.grid(xcoo, ycoo))
> scores <- gridXY %*% FLDA
> meanscores <- c(t(sapply(wines.groups, colMeans)) %*% FLDA)
> Fdistance <- outer(scores, meanscores,
+                    FUN = function(x, y) abs(x - y))
> Fclassif <- apply(Fdistance, 1, which.min)
```

The distances of the scores of all gridpoints to the scores of the class means are calculated using the `outer` function—this leads to a three-column matrix. The classification, corresponding to the class with the smallest distance, is obtained using the function `which.min`.

Finally, the class boundaries are visualized using the functions `image` and `contour`; the points of the test set are added afterwards.

```
> softbrg <- colorRampPalette(c("lightgray", "pink", "lightgreen"))
> image(x = xcoo, y = ycoo,
+       z = matrix(Fclassif, nrow = length(xcoo)),
+       xlab = "flavonoids", ylab = "proline",
+       main = "Fisher LDA", col = softbrg(3))
> box()
```

```
> contour(x = xcoo, y = ycoo, drawlabels = FALSE,
+         z = matrix(Fclassif, nrow = length(xcoo)),
+         add = TRUE)
> points(wines2.tst, col = wine.classes[wines.even],
+        pch = wine.classes[wines.even])
```

The result is shown in the left plot of Fig. 7.3. The right plot is produced analogously:

```
> wines.ldamod <- lda(wines2.trn,
+                     grouping = vint.trn,
+                     prior = rep(1, 3)/3)
> colnames(gridXY) <- colnames(wines)[c(7, 13)]
> lda.2Dclassif <- predict(wines.ldamod, newdata = gridXY)$class
> lda.2DCM <- matrix(as.integer(lda.2Dclassif), nrow = length(xcoo))
> image(x = xcoo, y = ycoo, z = lda.2DCM,
+       xlab = "flavonoids", ylab = "proline",
+       main = "LDA", col = softbrg(3))
> box()
> contour(x = xcoo, y = ycoo, z = lda.2DCM, drawlabels = FALSE,
+         add = TRUE)
> points(wines2.tst, col = wine.classes[wines.even],
+        pch = wine.classes[wines.even])
```

Immediately it is obvious that although the error rates of the two classifications are quite similar for the test set, large differences will occur when data points are further away from the class centers. The class means are reasonably close to a straight line, so that Fisher LDA does not fail completely; however, for multi-class problems it is not a good idea to impose parallel class boundaries, as is done by Fisher LDA using only one eigenvector. It is better to utilize the information in the second and higher eigenvectors of $\boldsymbol{W}^{-1}\boldsymbol{B}$ as well (Mardia et al. 1979); these are sometimes called *canonical variates*, and the corresponding form of discriminant analysis is known as *canonical discriminant analysis*. The maximum number of canonical variates that can be extracted is one less than the number of groups.

7.1.4 Quadratic Discriminant Analysis

Quadratic discriminant analysis (QDA) takes the same route as LDA, with the important distinction that every class is described by its own covariance matrix, rather than one identical (pooled) covariance matrix for all classes. Given our exposé on LDA, the algorithm for QDA is pretty simple: one calculates the Mahalanobis distances of all points to the class centers, and assigns each point to the closest class. Let us see what this looks like in two dimensions:

```
> wines2.groups <- split(as.data.frame(wines2.trn), vint.trn)
> wines2.covmats <- lapply(wines2.groups, cov)
> ngroups <- length(wines2.groups)
> distances <- sapply(1:ngroups,
+                     function(i, samples, means, covs) {
+                       mahalanobis(samples,
+                                   colMeans(means[[i]]),
+                                   covs[[i]]) },
+                     wines2.tst, wines2.groups, wines2.covmats)
> test.pred <- apply(distances, 1, which.min)
> table(vint.tst, test.pred)
             test.pred
vint.tst       1  2  3
  Barbera     19  0  5
  Barolo       0 28  1
  Grignolino   3  1 31
```

Ten samples are misclassified in the test set. To see the class boundaries in two-dimensional space, we use the same visualization as seen in the previous section:

```
> qda.mahal.dists <-
+   sapply(1:ngroups,
+          function(i, samples, means, covs) {
+            mahalanobis(samples,
+                        colMeans(means[[i]]),
+                        covs[[i]]) },
+          gridXY, wines2.groups, wines2.covmats)
> qda.2Dclassif <- apply(qda.mahal.dists, 1, which.min)
> qda.2DCM <- matrix(qda.2Dclassif, nrow = length(xcoo))
> image(x = xcoo, y = ycoo, z = qda.2DCM,
+       xlab = "flavonoids", ylab = "proline",
+       main = "QDA", col = softbrg(3))
> box()
> contour(x = xcoo, y = ycoo, z = qda.2DCM, drawlabels = FALSE,
+         add = TRUE)
> points(wines2.tst, col = as.integer(vint.tst),
+        pch = as.integer(vint.tst))
```

The result is shown in the left plot of Fig. 7.4. The quadratic form of the class boundaries is clearly visible. Again, only the test set objects are shown.

Using the `qda` function from the **MASS** package, modelling the odd rows and predicting the even rows is done just like with `lda`. Let's build a model using all thirteen variables:

```
> wines.qda <- qda(wines.trn, vint.trn,
+                  prior = rep(1, 3)/3)
> test.qdapred <- predict(wines.qda, newdata = wines.tst)
> table(vint.tst, test.qdapred$class)
```

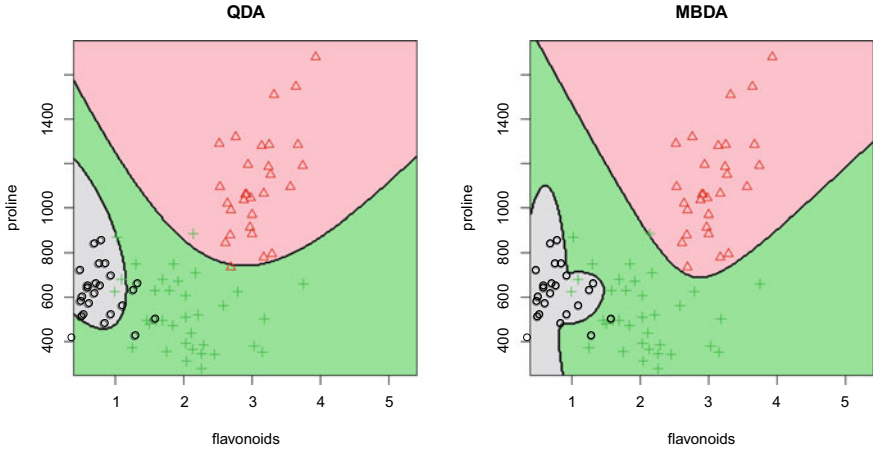

Fig. 7.4 Class boundaries for the wine data (proline and flavonoids only) for QDA (left) and MBDA (right). Models are created using the odd rows of the wine data (training data); the plotting symbols indicate the even rows (test data)

```
vint.tst     Barbera Barolo Grignolino
  Barbera         24       0          0
  Barolo           0      29          0
  Grignolino       0       0         35
```

In this case, all test set predictions are correct.

The optional correction for unequal class sizes (or account for prior probabilities) is done in exactly the same way as in LDA. Several other arguments are shared between the two functions: both `lda` and `qda` can be called with the `method` argument to obtain different estimates of means and variances: the standard plug-in estimators, maximum likelihood estimates, or two forms of robust estimates. The `CV` argument enables fast LOO crossvalidation.

7.1.5 Model-Based Discriminant Analysis

Although QDA uses more class-specific information, it still is possible that the data are not well described by the individual covariance matrices, e.g., in case of non-normally distributed data. In such a case one can employ more greedy forms of discriminant analysis, utilizing very detailed descriptions of class densities. In particular, one can describe each class with a mixture of normal distributions, just like in model-based clustering, and then assign an object to the class for which the overall mixture density is maximal. Thus, for every class one estimates *several* means and covariance matrices—one describes the pod by a set of peas. Obviously, this technique can only be used when the ratio of objects to variables is very large.

Package **mclust** contains several functions for doing *model-based discriminant analysis* (MBDA), or *mixture discriminant analysis*, as it is sometimes called as well. The easiest one is `MclustDA`:

```
> wines.MclustDA <-
+   MclustDA(wines.trn, vint.trn, G = 1:5, verbose = FALSE)
```

In this case, we have restricted the number of gaussians, to be used for each individual class, to be at most five. The `summary` method for the fitted object gives quite a lot of information:

```
> summary(wines.MclustDA)
------------------------------------------------
Gaussian finite mixture model for classification
------------------------------------------------

MclustDA model summary:

 log.likelihood  n  df     BIC
          -1479 89 151 -3635.8

Classes       n Model G
  Barbera    24   VEI 3
  Barolo     29   EEI 3
  Grignolino 36   VEI 2

Training classification summary:

            Predicted
Class        Barbera Barolo Grignolino
  Barbera         24      0          0
  Barolo           0     28          1
  Grignolino       0      0         36

Training error = 0.011236
```

Again, the BIC value is employed to select the optimal model complexity. For the Barolo class, a mixture of three gaussians seems optimal; these all have the same diagonal covariance matrix (indicated with model EEI). The two other classes can be described by mixtures of two and three diagonal covariance matrices, respectively, with varying volume—see Sect. 6.3 for more information on model definition in **mclust**. Classification for the training set is quite successful: only one object is misclassified. Predictions for the test set can be obtained in the usual way:

```
> wines.McDApred <-
+   predict(wines.MclustDA, newdata = wines.tst)$classification
> sum(wines.McDApred != vint.tst)
[1] 1
```

Also here, only one sample is misclassified (a Barolo is seen as a Grignolino).

If more control is needed over the training process, functions `MclustDAtrain` and `MclustDAtest` are available in the **mclust** package. To visualize the class boundaries in the two dimensions of the wine data set employed earlier for the other forms of discriminant analysis, we can use

```
> wines.mclust2D <-
+   MclustDA(wines2.trn, vint.trn, G = 1:5, verbose = FALSE)
```

The model is simpler than the model employed for the full, 13-dimensional data, which seems logical. Prediction and visualization is done by

```
> wines.mclust2Dpred <- predict(wines.mclust2D, gridXY)
> mbda.2DCM <- matrix(as.integer(wines.mclust2Dpred$classification),
+                     nrow = length(xcoo))


> image(x = xcoo, y = ycoo, z = mbda.2DCM,
+       main = "MBDA", xlab = "flavonoids", ylab = "proline",
+       col = softbrg(3))
> box()
> contour(x = xcoo, y = ycoo, z = mbda.2DCM, drawlabels = FALSE,
+         add = TRUE)
> points(wines2.tst,
+        col = as.integer(vint.tst),
+        pch = as.integer(vint.tst))
```

The class boundaries, shown in the right plot of Fig. 7.4, are clearly much more adapted to the densities of the individual classes, compared to the other forms of discriminant analysis we have seen.

7.1.6 Regularized Forms of Discriminant Analysis

At the other end of the scale we find methods that are suitable in cases where we cannot afford to use very complicated descriptions of class density. One form of regularized DA (RDA) strikes a balance between linear and quadratic forms (Friedman 1989): the idea is to apply QDA using covariance matrices $\widetilde{\Sigma}_k$ that are shrunk towards the pooled covariance matrix Σ:

$$\widetilde{\Sigma}_k = \alpha \widehat{\Sigma}_k + (1 - \alpha)\Sigma \tag{7.14}$$

where $\widehat{\Sigma}_k$ is the empirical covariance matrix of class k. In this way, characteristics of the individual classes are taken into account, but they are stabilized by the pooled variance estimate. The parameter α needs to be optimized, e.g., by using crossvalidation.

In cases where the number of variables exceeds the number of samples, more extreme regularization is necessary. One way to achieve this is shrinkage towards

the unity matrix (Hastie et al. 2001):

$$\widetilde{\Sigma} = \alpha\Sigma + (1-\alpha)I \tag{7.15}$$

Equivalent formulations are given by:

$$\widetilde{\Sigma} = \kappa\Sigma + I \tag{7.16}$$

and

$$\widetilde{\Sigma} = \Sigma + \kappa I \tag{7.17}$$

with $\kappa \geq 0$. In this form of RDA, again the regularized form $\widetilde{\Sigma}$ of the covariance is used, rather than the empirical pooled estimate Σ. Matrix $\widetilde{\Sigma}$ is not singular so that the matrix inversions in Eqs. 7.3 or 7.6 no longer present a problem. In the extreme case, one can use a diagonal covariance matrix (with the individual variances on the diagonal) leading to diagonal LDA (Dudoit et al. 2002), also known as Idiot's Bayes (Hand and Yu 2001). Effectively, all dependencies between variables are completely ignored. For so-called "fat" matrices, containing many more variables than objects, often encountered in microarray research and other fields in the life sciences, such simple methods often give surprisingly good results.

7.1.6.1 Diagonal Discriminant Analysis

As an example, consider the odd rows of the prostate data, limited to the first 1000 variables. We are concentrating on the separation between the control samples and the cancer samples:

```
> prostate <- rowsum(t(Prostate2000Raw$intensity),
+                    group = rep(1:327, each = 2),
+                    reorder = FALSE) / 2
> prostate.type <- Prostate2000Raw$type[seq(1, 654, by = 2)]
>
> prost <- prostate[prostate.type != "bph", 1:1000]
> prost.type <- factor(prostate.type[prostate.type != "bph"])
> prost.df <- data.frame(type = prost.type, prost = prost)
> prost.odd <- seq(1, length(prost.type), by = 2)
> prost.even <- seq(2, length(prost.type), by = 2)
```

Although it is easy to re-use the code given in Sects. 7.1.1 and 7.1.4, plugging in diagonal covariance matrices, here we will use the `dDA` function from the **sfsmisc** package:

```
> prost.dlda <-
+   dDA(prost[prost.odd, ], as.integer(prost.type)[prost.odd])
```

By default, the same covariance matrix is used for all classes, just like in LDA. Here, the result for the predictions on the even samples is not too bad:

```
> prost.dldapred <- predict(prost.dlda, prost[prost.even, ])
> table(prost.type[prost.even], prost.dldapred)
         prost.dldapred
           1  2
  control 32  8
  pca      7 77
```

Approximately 88% of the test samples are predicted correctly. Allowing for different covariance matrices per class, we arrive at diagonal QDA, which does slightly worse for these data:

```
> prost.dqda <-
+   dDA(prost[prost.odd, ], as.integer(prost.type)[prost.odd],
+       pool = FALSE)
> prost.dqdapred <- predict(prost.dqda, prost[prost.even, ])
> table(prost.type[prost.even], prost.dqdapred)
         prost.dqdapred
           1  2
  control 38  2
  pca     16 68
```

7.1.6.2 Shrunken Centroid Discriminant Analysis

In the context of microarray analysis, it has been suggested to combine RDA with the concept of "shrunken centroids" (Tibshirani et al. 2003)—the resulting method is indicated as SCRDA (Guo et al. 2007) and is available in the R package **rda**. As the name suggests, class means are shrunk towards the overall mean. The effect is that the points defining the class boundaries (the centers) are closer, which may lead to a better description of local structure. These shrunken class means are then used in Eq. 7.3, together with the diagonal covariance matrix also employed in DLDA. For a more complete description, see, e.g., (Hastie et al. 2001).

Let us see how SCRDA does on the prostate example. Application of the `rda` function is straightforward.[2] The function takes two arguments, α and δ, where α again indicates the amount of unity matrix in the covariance estimate, and δ is a soft threshold, indicating the minimal coefficient size for variables to be taken into account in the classification:

```
> prost.rda <-
+   rda(t(prost[prost.odd, ]), as.integer(prost.type)[prost.odd],
+       delta = seq(0, .4, len = 5), alpha = seq(0, .4, len = 5))
```

Printing the fitted object shows some interesting results:

[2]Note that in this function the variables are in the *rows* of the data matrix and not, as usual, in the columns—hence the use of the transpose function.

```
> prost.rda
Call:
rda(x = t(prost[prost.odd, ]), y = as.integer(prost.type)[prost.odd],
    alpha = seq(0, 0.4, len = 5), delta = seq(0, 0.4, len = 5))
$nonzero
     delta
alpha    0 0.1 0.2 0.3 0.4
  0   1000 433 193 121  92
  0.1 1000 220  34   3   0
  0.2 1000 192  19   4   0
  0.3 1000 179  18   4   0
  0.4 1000 195  24   4   0

$errors
     delta
alpha  0 0.1 0.2 0.3 0.4
  0   36  38  39  39  39
  0.1 10  16  32  41  41
  0.2  7  21  43  41  41
  0.3  4  23  44  41  41
  0.4  2  20  44  41  41
```

Increasing values of δ lead to a rapid decrease in the number of non-zero coefficients; however, these sparse models do not lead to very good predictions, and the lowest value for the training error is found at $\alpha = .4$ and $\delta = 0$. Obviously, the training error is not the right criterion to decide on the optimal values for these parameters. This we can do using the `rda.cv` crossvalidation function, and subsequently we can use the test data as a means to estimate the expected prediction error:

```
> prost.rdacv <-
+   rda.cv(prost.rda, t(prost[prost.odd, ]),
+          as.integer(prost.type)[prost.odd])
```

Inspection of the result (not shown) reveals that the optimal value of α would be .2, with no thresholding ($delta = 0$). Predictions with these values lead to the following result:

```
> prost.rdapred <-
+   predict(prost.rda,
+           t(prost[prost.odd, ]), as.integer(prost.type)[prost.odd],
+           t(prost[prost.even, ]), alpha = .2, delta = 0)
> table(prost.type[prost.even], prost.rdapred)
         prost.rdapred
           1  2
  control 30 10
  pca      4 80
```

Overall, fourteen samples are misclassified, only slightly better than the DLDA model. This sort of behavior is more general than one might think: for fat data, the simplest models are often among the top performers.

7.2 Nearest-Neighbor Approaches

A completely different approach, not relying on any distributional assumptions whatsoever, is formed by techniques focusing on distances between objects, and in particular on the closest objects. These techniques are known under the name of k-nearest-neighbors (KNN), where k is a number to be determined. If $k = 1$, only the closest neighbor is taken into account, and any new object will be assigned to the class of its closest neighbor in the training set. If $k > 1$, the classification is straightforward in cases where the k nearest neighbors are all of the same class. If not, a majority vote is usually performed. Class areas can be much more fragmented than with LDA or QDA; in extreme cases one can even find patch-work-like patterns. The smaller the number k, the more irregular the areas can become: only one object is needed to assign its immediate surroundings to a particular class.

As an example, consider the KNN classification for the first sample in the test set of the wine data (sample number two), based on the training set given by the odd samples. One starts by calculating the distance to all samples in the training set. Usually, the Euclidean distance is used—in that case, one should scale the data appropriately to avoid large numbers to dominate the results. For the wine data, autoscaling is advisable. The `mahalanobis` function has a useful feature that allows one to calculate the distance of one object to a set of others. The covariance matrix is given as the third argument. Thus, the Euclidean distance between samples in the autoscaled wine data can be calculated in two ways, either from the autoscaled data using a unit covariance matrix, or from the unscaled data using the estimated column standard deviations. Consider the wine data, scaled according to the means and variances of the odd rows (the training set). Calculating the distance to the second sample in these two ways leads to the following result:

```
> wines.trn.sc <- scale(wines.trn)
> wines.tst.sc <- scale(wines.tst,
+                       scale = apply(wines.trn, 2, sd),
+                       center = colMeans(wines.trn))


> dist2sample2a <- mahalanobis(wines.trn.sc,
+                              wines.tst.sc[1, ],
+                              diag(13))
> dist2sample2b <- mahalanobis(wines.trn,
+                              wines.tst[1, ],
+                              diag(apply(wines.trn, 2, var)))
>
> range(dist2sample2a - dist2sample2b)
[1] -7.1054e-15  1.4211e-14
```

Clearly, the two lead to the same result. Next, we order the training samples according to their distance to the second sample:

```
> nearest.classes <- vint.trn[order(dist2sample2a)]
> table(nearest.classes[1:10])

   Barbera     Barolo Grignolino
         0         10          0
```

The closest ten objects are all of the Barolo class—apparently, there is little doubt that object 2 also should be a Barolo.

Rather than using a diagonal of the covariance matrix, one could also use the complete estimated covariance matrix of the training set. This would lead to the Mahalanobis distance:

```
> dist2sample2 <- mahalanobis(wines.trn,
+                             wines.tst[1, ],
+                             cov(wines.trn))
> nearest.classes <- vint.trn[order(dist2sample2)]
> table(nearest.classes[1:10])

   Barbera     Barolo Grignolino
         0          6          4
```

Note that autoscaling of the data is not necessary because we explicitly include the covariance matrix of the training data. Clearly, the results depend on the distance measure employed. Although the closest three samples are Barolo wines, the next three are Grignolinos; values of k between 5 and 9 would lead to a close call or even a tie. Several different strategies to deal with such cases can be employed. The simplest is to require a significant majority for any classification—in a 5-NN classification one may require at least four of the five closest neighbors to belong to the same class. If this is not the case, the classification category becomes "unknown". Although this may seem a weakness, in many applications it is regarded as a strong point if a method can indicate some kind of reliability—or lack thereof—for individual predictions.

The **class** package contains an implementation of the KNN classifier using Euclidean distances, `knn`. Its first argument is a matrix constituting the training set, and the second argument is the matrix for which class predictions are required. The class labels of the training set are given in the third argument. It provides great flexibility in handling ties: the default strategy is to choose randomly between the (tied) top candidates, so that repeated application can lead to different results:

```
> knn(wines.sc[wines.odd, ], wines.sc[68, ], cl = vint.trn, k = 4)
[1] Barbera
Levels: Barbera Barolo Grignolino
> knn(wines.sc[wines.odd, ], wines.sc[68, ], cl = vint.trn, k = 4)
[1] Grignolino
Levels: Barbera Barolo Grignolino
```

Apparently, there is some doubt about the classification of sample 68—it can be either a Barbera or Grignolino. Of course, this is caused by the fact that from the four closest neighbors, two are Barberas and two are Grignolinos. Requiring at least three votes for an unambiguous classification (`l = 3`) leads to:

```
> knn(wines.sc[wines.odd, ], wines.sc[68, ], cl = vint.trn,
+     k = 4, l = 3)
[1] <NA>
Levels: Barbera Barolo Grignolino
```

In many cases it is better not to have a prediction at all, rather than a highly uncertain one.

The value of k is crucial. Unfortunately, no rules of thumb can be given on the optimal choice, and this must be optimized for every data set separately. One simple strategy is to monitor the performance of the test set for several values of K and pick the one that leads to the smallest number of misclassifications. Alternatively, LOO crossvalidation can be employed:

```
> wines.knnresult <- rep(0, 10)
> for (i in 1:10) {
+   wines.knncv <- knn.cv(wines.sc[wines.odd, ], vint.trn, k = i)
+   wines.knnresult[i] <-
+     sum(diag(table(vint.trn, wines.knncv)))
+ }
> round(100 * wines.knnresult / length(wines.odd), 1)
 [1] 92.1 92.1 96.6 92.1 95.5 96.6 95.5 95.5 96.6 94.4
```

In this example, k values of three, six and nine show the best prediction—differences are not large.

An alternative is to use the convenience function `tune.knn` in package **e1071**. This function by default uses ten-fold crossvalidation for a range of values of k:

```
> (knn.tuned <- tune.knn(wines.sc[wines.odd, ], vint.trn, k = 1:10))

Parameter tuning of 'knn.wrapper':

- sampling method: 10-fold cross validation

- best parameters:
 k
 3

- best performance: 0.022222
```

Plotting the `knn.tuned` object leads to the left plot of Fig. 7.5—the differences with the LOO results we saw earlier show what kind of variability is to be expected with crossvalidated error estimates. Indeed, repeated application of the tune function will—for these data—lead to quite different estimates for the optimal value of k:

```
> bestKs <- rep(0, 1000)
> for (i in 1:1000)
+   bestKs[i] <- tune.knn(wines.sc[wines.odd, ],
+                         vint.trn,
+                         k = 1:10)$best.parameters[1, 1]
```

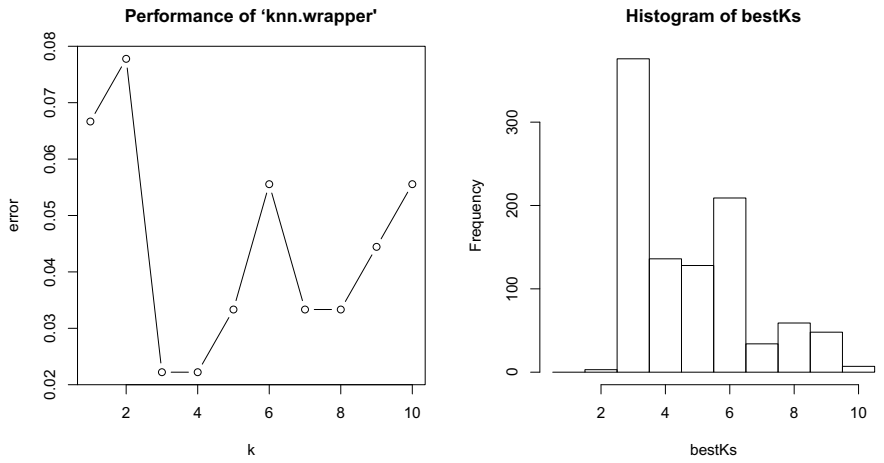

Fig. 7.5 Optimization of k for the wine data using the `tune` wrapper function. Left plot: one crossvalidation curve. Right plot: optimal values of k in 1000 crossvalidations

The right plot of Fig. 7.5 shows a histogram of the best k values. In the large majority of cases, $k = 2$ is best. This is partly caused by a built-in preference for small values of k in the script: the smallest value of k that gives the optimal predictions is stored, even though larger values may lead to equally good predictions, something that given the rather small size of our data set can easily occur.

Although application of these simple strategies allow one to choose the optimal parameter settings, the optimal error associated with this setting (e.g., 97.8% in the LOO example) is not an estimation of the prediction error of future samples, because the test set is used in this procedure to fine-tune the method. Another layer of validation is necessary to find the estimated prediction error; see Chap. 9. The 1-nearest neighbor method enjoys great popularity, despite coming out worst in the above comparison—there, it is almost never selected. Nevertheless, it has been awarded a separate function in the **class** package: `knn1`. Most often, odd values of K smaller than ten are considered.

One potential disadvantage of the KNN method is that in principle, the whole training set—the training set in a sense *is* the model!—should be saved, which can be a nuisance for large data sets. Predictions for new objects can be slow, and storing really large data sets may present memory problems. However, things are not so bad as they seem, since in many cases one can safely prune away objects without sacrificing information. For the wine data, it is obvious that in large parts of the space there is no doubt: only objects from one class are present. Many of these objects can be removed and one then still will get exactly the same classification for all possible new objects.

7.3 Tree-Based Approaches

A wholly different approach to classification is formed by tree-based approaches. These proceed in a way that is very similar to medical diagnosis: the data are "interrogated" and a series of questions are posed which finally lead to a classification. Modelling, in this metaphore, is to decide which questions are most informative. As a class, tree-based methods possess some unique advantages. They can be used for both classification and regression. Since the model is based on sequential decisions on individual variables, scaling is not important: every variable is treated at its own scale and no "overall" measure needs to be computed. Variable selection comes with the method—only those variables that contribute to the result are incorporated in the tree. Trees form one of the few methods that can accommodate variables of a very different nature, e.g., numerical, categorical and ordinal, in one single model. Their handling of missing values is simple and elegant. In short, trees can be used for almost any classification (and regression) problem.

Currently, tree-based modelling comes in two main branches: Breiman's Classification and Regression Trees (CART, Breiman et al. 1984) and Quinlan's See5/C5.0 (and its predecessors, C4.5, Quinlan 1993, and ID3, Quinlan 1986). Both are commercial and not open-source software, but R comes with two pretty faithful representation of CART in the form of the `rpart` and `tree` functions, from the packages with the same names. Since the approaches by Quinlan and Breiman have become more similar with every new release, and since no See5/C5.0 implementation is available in R, we will here only focus on `rpart` (Therneau and Atkinson 1997)—one of the recommended R packages—to describe the main ideas of tree-based classification. The differences between CART and the implementation in the **tree** package are small; consult the manual pages and (Therneau and Atkinson 1997) for more information.

7.3.1 Recursive Partitioning and Regression Trees

Recursive Partitioning and Regression Trees, which is what the acronym `rpart` stands for, can be explained most easily by means of an example. Consider, again, the the two-dimensional subset of the wine data encountered earlier:

```
> wines2.df <- data.frame(vint = vintages, wines[, c(7, 13)])
> wines2.rpart <- rpart(vint ~ ., subset = wines.odd,
+                       data = wines2.df, method = "class")
```

In setting up the tree model, we explicitly indicate that we mean classification (`method = "class"`): the `rpart` function also provides methods for survival analysis and regression, and though it tries to be smart in guessing what exactly is required, it is better to explicitly provide the `method` argument. The result is an object of class `rpart`:

```
> wines2.rpart
n= 89

node), split, n, loss, yval, (yprob)
      * denotes terminal node

1) root 89 53 Grignolino (0.269663 0.325843 0.404494)
  2) flavonoids< 1.235 23  1 Barbera (0.956522 0.000000 0.043478) *
  3) flavonoids>=1.235 66 31 Grignolino (0.030303 0.439394 0.530303)
    6) proline>=739 30  2 Barolo (0.000000 0.933333 0.066667) *
    7) proline< 739 36  3 Grignolino (0.055556 0.027778 0.916667) *
```

The top node, where no splits have been defined, is labelled as "Grignolino" since that is the most abundant variety—36 out of 89 objects (a fraction of 0.4045) are Grignolinos. The first split is on the `flavonoids` variable. A value smaller than 1.235 leads to node 2, which is consisting almost completely of Barbera samples (more than 95 percent). This node is not split any further, and in tree terminology is indicated as a "leaf". A flavonoid value larger than 1.235 leads to node three that is split further into separate Barolo and Grignolino leaves. Of course, such a tree is much easier to interpret when depicted graphically:

```
> plot(wines2.rpart, margin = .12)
> text(wines2.rpart, use.n = TRUE)
> cl.id <- as.integer(wines2.df$vint[wines.odd])
> plot(wines2.df[wines.odd, 2:3], pch = cl.id, col = cl.id)
> segments(wines2.rpart$splits[1, 4], par("usr")[3],
+          wines2.rpart$splits[1, 4], par("usr")[4], lty = 2)
> segments(wines2.rpart$splits[1, 4], wines2.rpart$splits[2, 4],
+          par("usr")[2], wines2.rpart$splits[2, 4], lty = 2)
```

This leads to the plots in Fig. 7.6. The tree on the left shows the splits, corresponding to the tessellation of the surface in the right plot. The plot command sets up the coordinate system en plots the tree; the `margin` argument is necessary to reserve some space for the annotation added by the `text` command. At every split, the test, stored in the `splits` element in the `X.rpart` object, is shown. At the final nodes (the "leaves"), the results are summarized: one Barbera (red triangles) is classified as a Grignolino (green pluses), two Barolos (black circles) are in the Grignolino area; two Grignolinos are thought to be Barolos, one a Barbera.

The `rpart` function has a familiar formula interface also used in, e.g., `lm`. For the wine data, we will predict the classes of the even rows, again based on the odd rows:

```
> wines.df <- data.frame(vint = vintages, wines)
> wines.rpart <- rpart(vint ~ ., subset = wines.odd,
+                      data = wines.df, method = "class")
```

Plotting the `rpart` object leads to Fig. 7.7. The flavonoids and proline variables are again important in the classification, now in addition to the colour intensity.

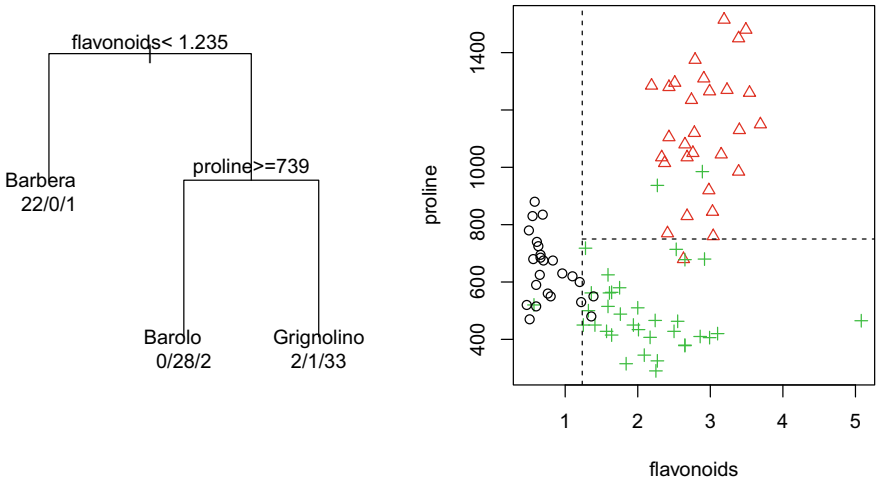

Fig. 7.6 Tree object from `rpart` using default settings (left). The two nodes lead to the class boundaries visualized in the right plot. Only points for the even rows, the test set, are shown

Prediction is done using the `predict.rpart` function, which returns a matrix of class probabilities, simply estimated from the composition of the training samples in the end leaves:

```
> wines.rpart.predict <- predict(wines.rpart,
+                                newdata = wines.df[wines.even, ])
> wines.rpart.predict[31:34, ]
   Barbera    Barolo Grignolino
62       0 0.032258    0.96774
64       0 0.032258    0.96774
66       0 0.142857    0.85714
68       0 0.032258    0.96774
```

In this rather simple problem, most of the probabilities are either 0 or 1, but here some Grignolinos are shown that have a slight chance of actually being Barolos, according to the tree model. The uncertainties are simply the misclassification rates of the training model: row 66 ends up in a lead containing seven training samples, one of which is a Barolo and six are Grignolinos. The other rows end up in the large Grignolino group, containing also one Barolo sample. A global overview is more easily obtained by plotting the probabilities:

```
> matplot(wines.rpart.predict, xlab = "sample number (test set)")
```

This leads to the plot in Fig. 7.8. Clearly, most of the Barolos and Barberas are classified with complete confidence, corresponding with "pure" leaves. The Grignolinos on the other hand always end up in a leaf also containing one Barolo sample. When using the `type = "class"` argument to the prediction function, the result is immediately expressed in terms of classes, and can be used to assess the overall prediction quality:

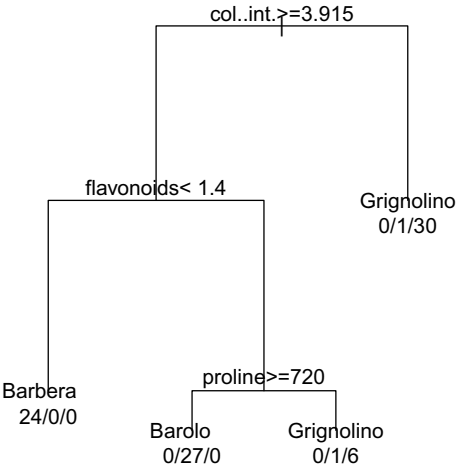

Fig. 7.7 Fitted tree using `rpart` on the odd rows of the `wine` data set (all thirteen variables)

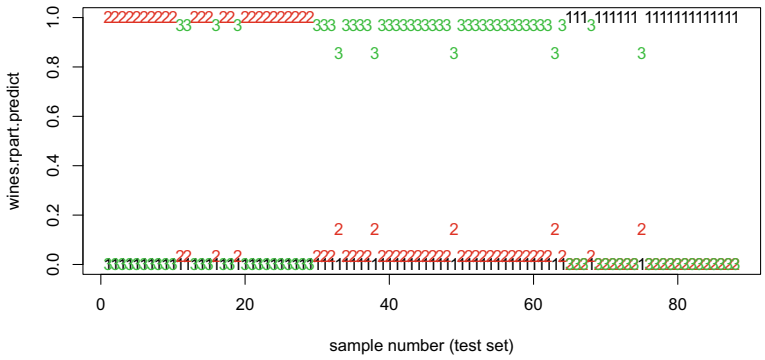

Fig. 7.8 Classification probabilities of the test set of the wine data for the tree shown in Fig. 7.7. Within the first twenty samples we see four incorrect predictions: the true class is Barolo (indicated by "2") but there is some confusion with Grignolino ("3"). Similarly, in the right part of the plot two Barbera samples are seen as Grignolinos

```
> table(vint.tst,
+       predict(wines.rpart, newdata = wines.df[wines.even, ],
+               type = "class"))

vint.tst     Barbera Barolo Grignolino
  Barbera         22      0          2
  Barolo           0     25          4
  Grignolino       0      0         35
```

This corresponds to the six misclassifications seen in Fig. 7.8.

7.3.1.1 Constructing the Tree

The construction of the optimal tree cannot be guaranteed to finish in polynomial time (a so-called NP-complete problem), and therefore one has to resort to simple approximations. The standard approach is the following. All possible splits—binary divisions of the data—in all predictor variables are considered; the one leading to the most "pure" branches is selected. The term "pure" in this case signifies that, in one leaf, only instances of one class are present. For categorical variables, tests for unique values are used; for continuous variables, all data points are considered as potential split values. This simple procedure is applied recursively until some stopping criterion is met.

The crucial point is the definition of "impurity": several different measures can be used. Two criteria are standing out (Ripley 1996): the Gini index, and the entropy. The Gini index of a node is given by

$$I_G(p) = \sum_{i \neq j} p_i p_j = 1 - \sum_j p_j^2 \tag{7.18}$$

and is minimal (exactly zero) when the node contains only samples from one class—p_i is the fraction of samples from class i in the node. A simple function calculating the Gini index looks like this:

```
> gini <- function(x, clss) {
+   p <- table(clss) / length(clss)
+   gini.parent <- 1 - sum(p^2)
+
+   gini.index <-
+     sapply(sort(x), function(splitpoint) {
+       left.ones <- clss[x < splitpoint]
+       right.ones <- clss[x >= splitpoint]
+       nleft <- length(left.ones)
+       nright <- length(right.ones)
+
+       if ((nleft == 0) | (nright == 0)) return (NA)
+
+       p.left <- table(left.ones) / nleft
+       p.right <- table(right.ones) / nright
+
+       (nleft * (1 - sum(p.left^2)) +
+        nright * (1 - sum(p.right^2))) /
+         (nleft + nright)
+     })
+
+   gini.index - gini.parent
+ }
```

This function takes a vector `x`, for instance values for the `proline` variable in the `wines` data, and a class vector. Impurity values are calculated where each value in the sorted vector `x` is considered as a split point. To really quantify improvement

after splitting at that node, the Gini index of the parent node is subtracted: the more negative the number, the bigger the improvement.

The other impurity criterion is based on entropy, where the entropy of a node is defined by

$$I_E(p) = -\sum_j p_j \log p_j$$

which again is minimal when the node is pure and contains only samples of one class (where we define $0 \log 0 = 0$).

The optimal split is the one that minimizes the average impurity of the new left and right branches (whatever criterion is used):

$$P_l I(p_l) + P_r I(p_r)$$

where P_l and P_r signify the sample fractions and $I(p_l)$ and $I(p_r)$ are the impurities of the left and right branches, respectively.

As an illustration, again consider the two-dimensional subset of the odd rows of the wine data, using variables `flavonoids` and `proline`. Since the data are continuous, we consider all values as potential splits, and calculate the Gini and entropy indices. For the two-dimensional wine data this leads to:

```
> wines2.df.odd <- wines2.df[wines.odd, ]
> Ginis <- sapply(wines2.df.odd[, -1], gini, wines2.df.odd$vint)
> apply(Ginis, 2, min, na.rm = TRUE)
flavonoids    proline
  -0.24683   -0.24127
> (idx <- which.min(Ginis[, 1]))
[1] 24
> (bestSplit <- sort(wines2.df.odd[, "flavonoids"])[idx])
[1] 1.25
```

Plotting the Gini values for the two columns leads to the left panel in Fig. 7.9. Because of the lower Gini index, corresponding to more pure leaves, the first split should be on the `flavonoids` column. The split point equals the 24th sorted value. Next we divide the data into two sets, one to the left of the `bestSplit` value, and one to the right. Here we show only the result for the right split following results:

```
> wr <- wines2.df.odd[wines2.df.odd$flavonoids >= bestSplit, ]
> GinisR <- sapply(wr[, -1], gini, wr$vint)
> apply(GinisR, 2, min, na.rm = TRUE)
flavonoids    proline
  -0.18750   -0.38321
> (idxR <- which.min(GinisR[, 2]))
[1] 37
> (bestSplitR <- sort(wr[, "proline"])[idxR])
[1] 760
```

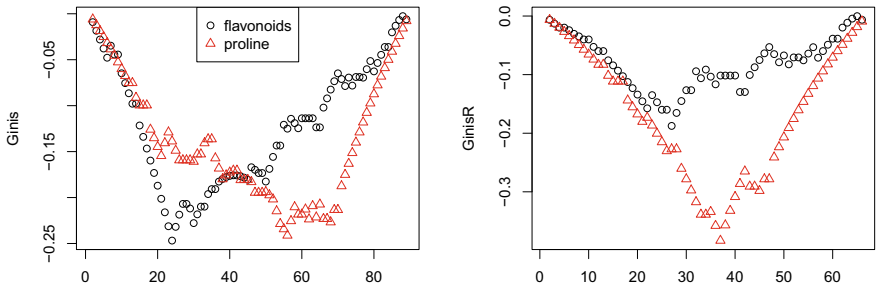

Fig. 7.9 Impurity values (Gini indices) for all possible split points in the two-dimensional subset of the wine data. The left panel points to the `flavonoids` variable for selecting the first split point; the right panel shows that the subsequent fit with the biggest gain is in the `proline` variable

Clearly, the second split should be done for the `proline` column at the level 760. The two splits correspond exactly to the results in Fig. 7.6. The Gini values for the right split are shown in the right panel of Fig. 7.9.

Obviously, one can keep on splitting nodes until every sample in the training set is a leaf in itself, or in any case until each single leaf contains only instances of one class. Such a tree is able to represent the training data perfectly, but whether the predictions of such a tree are reliable is quite another matter. In fact, these trees generally will not perform very well. By describing every single feature of the training set, the tree is not able to generalize. This is an example of *overfitting* (or overtraining, as it is sometimes called as well), something that is likely to occur in methods that have a large flexibility—in the case of trees, the freedom to keep on adding nodes.

The way this problem is tackled in constructing optimal trees is to use *pruning*, i.e., trimming useless branches. When exactly a branch is useless needs to be assessed by some form of validation—in **rpart**, tenfold crossvalidation is used by default. One can therefore easily find out whether a particular branch leads to a decrease in prediction error or not.

More specifically, in pruning one minimizes the cost of a tree, expressed as

$$C(T) = R(T) + \alpha|T| \tag{7.19}$$

In this equation, T is a tree with $|T|$ leafs, $R(T)$ the "risk" of the tree—e.g., the proportion of misclassifications—and α a complexity penalty, chosen between 0 and ∞. One can see α as the cost of adding another node. It is not necessarily to build up the complete tree to calculate this measure: during the construction the cost of the current tree can be assessed and if it is above a certain value, the process stops. This cost is indicated with the complexity parameter (`cp`) in the `rpart` function, which is normalized so that the root node has a complexity value of one.

Once again looking at the first 1000 variables of the `control` and `pca` classes in the prostate data, one can issue the following commands to construct the full tree with no misclassifications in the training set:

```
> prost.df <- data.frame(type = prost.type, prost = prost)
> prost.rpart <-
+   rpart(type ~ ., data = prost.df, subset = prost.odd,
+         control = rpart.control(cp = 0, minsplit = 0))
```

The two extra arguments tell the `rpart` function to keep on looking for splits even when the complexity parameter, `cp`, gets smaller than 0.1 and the minimal number of objects in a potentially split node, `minsplit`, is smaller than 20 (the default values). This leads to a tree with seven leaves. Printing the `prost.rpart` object would show that the training data are indeed predicted perfectly. However, four of the terminal nodes contain only three or fewer samples: it seems these are introduced to repair some individual cases. Indeed, the test data are not predicted with the same level of accuracy:

```
> prost.rpartpred <-
+   predict(prost.rpart, newdata = prost.df[prost.even, ])
> table(prost.type[prost.even], classmat2classvec(prost.rpartpred))

          control pca
  control      29  11
  pca          12  72
```

Pruning could decrease the complexity without sacrificing much accuracy in the description of the training set, and hopefully would increase the generalizing abilities of the model. To see what level of pruning is necessary, the table of complexity values can be printed:

```
> printcp(prost.rpart)

Classification tree:
rpart(formula = type ~ ., data = prost.df, subset = prost.odd,
    control = rpart.control(cp = 0, minsplit = 0))

Variables actually used in tree construction:
[1] prost.4909 prost.5013 prost.5110 prost.5261 prost.5489 prost.5866

Root node error: 41/125 = 0.328

n= 125

      CP nsplit rel error xerror  xstd
1 0.5610      0    1.0000  1.000 0.128
2 0.2927      1    0.4390  0.829 0.121
3 0.0488      2    0.1463  0.634 0.111
4 0.0244      4    0.0488  0.561 0.106
5 0.0000      6    0.0000  0.561 0.106
```

Also a graphical representation is available:

```
> plotcp(prost.rpart)
```

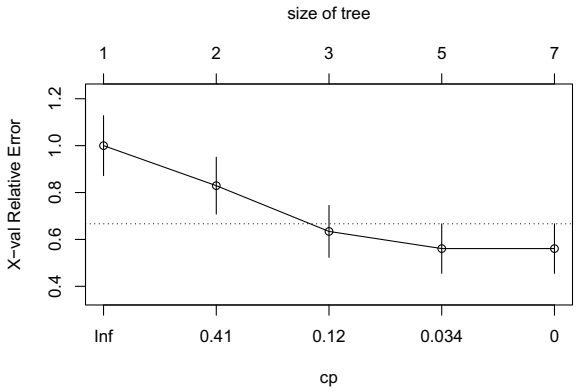

Fig. 7.10 Complexity pruning of a tree: in this case, three terminal nodes are optimal (lowest prediction error at lowest complexity)

This leads to Fig. 7.10. Both from this figure and the complexity table shown above, it is clear that the tree with the lowest prediction error and the least number of nodes is obtained at a value of `cp` equal to 0.12. Usually, one chooses the complexity corresponding to the minimum of the predicted error plus one standard deviation, indicated by the dotted line in Fig. 7.10. The tree created with a `cp` value of 0.12, containing only two leaves rather than the original seven, leads to a higher number of misclassifications (six rather than zero) in the training set, but unfortunately also to a slightly higher number of misclassifications in the test set:

```
> prost.rpart2 <-
+   rpart(type ~ ., data = prost.df, subset = prost.odd,
+         control = rpart.control(cp = 0.12))
> prost.rpart2pred <-
+   predict(prost.rpart2, newdata = prost.df[prost.even, ])
> table(prost.type[prost.even], classmat2classvec(prost.rpart2pred))

         control pca
control      29  11
pca          15  69
```

Either way, the result is quite a bit worse than what we have seen earlier with RDA (Sect. 7.1.6.2).

Apart from the 0/1 loss function normally used in classification (a prediction is either right or wrong), **rpart** allows to specify other, more complicated loss functions as well—often, the cost of a false positive is very different from the cost of a false negative decision. Another useful feature in the **rpart** package is the possibility to provide prior probabilities for all classes.

7.3.2 Discussion

Trees offer a lot of advantages. Perhaps the biggest of them is the appeal of the particular form of the model: many scientists feel comfortable with a series of more and more specific questions, eventually leading to an unambiguous answer. The implicit variable selection makes model interpretation much easier, and alleviates many problems with missing values, and variables of mixed types (boolean, categorical, ordinal, numerical).

There are downsides too, of course. The number of parameters to adjust is large, and although the default settings quite often lead to reasonable solutions, there may be a temptation to keep fiddling until an even better result is obtained. This, however, can easily lead to overfitting: although the data are faithfully reproduced, the model is too specific and lacks generalization power. As a result, predictions for future data are generally of lower quality than expected. And as for the interpretability of the model: this is very much dependent on the composition of the training set. A small change in the data can lead to a completely different tree. As we will see, this is a disadvantage that can be turned into an advantage: combinations of tree-based classifiers often give stable and accurate predictions. These so-called Random Forests, taking away many of the disadvantages of simple tree-based classifiers while keeping the good characteristics, enjoy huge popularity and will be treated in Sect. 9.7.2.

7.4 More Complicated Techniques

When relatively simple models like LDA or KNN do not succeed in producing models with good predictive capabilities, one can ask the question: why do we fail? Is it because the data just do not contain enough information to build a useful model? Or are the models we have tried too simple? Do we need something more flexible, perhaps nonlinear? The distinction between information-poor data and complicated class boundaries is often hard to make.

In this section, we will treat two popular nonlinear techniques from the domain of Machine Learning with complementary characteristics: whereas *Support Vector Machines* (SVMs) are very useful when the number of objects is not too large, *Artificial Neural Networks* (ANNs) should only be applied when there are ample training cases available. Conversely, SVMs are applicable in high-dimensional cases whereas ANNs are not: very often, a data reduction step like PCA is employed to bring the number of variables down to a manageable size. These two techniques do share one important property: they are very flexible indeed, and capable of modelling the most complex relationships. This puts a large responsibility on the researcher for thorough validation, especially since there are several parameters to tune. Because the theory behind the methods is rather extensive, we will only sketch the contours—interested readers are referred to the literature for more details.

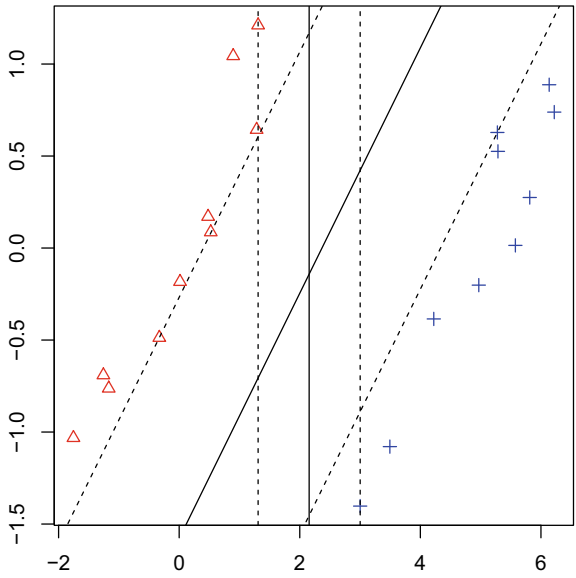

Fig. 7.11 The basic idea behind SVM classification: the separating hyperplane (here, in two dimensions, a line) is chosen in such a way that the margin is maximal. Points on the margins (the dashed lines) are called "support vectors". Clearly, the margins for the separating line with slope $2/3$ are much further apart than for the vertical boundary

7.4.1 Support Vector Machines

SVMs (Vapnik 1995; Cristianini and Shawe-Taylor 2000; Schölkopf and Smola 2002) in essence are binary classifiers, able to discriminate between two classes. They aim at finding a separating hyperplane maximizing the distance between the two classes. This distance is called the *margin* in SVM jargon; a synthetic example, present in almost all introductions to SVMs, is shown in Fig. 7.11. Although both classifiers, indicated by the solid lines, perfectly separate the two classes, the classifier with slope $2/3$ achieves a much bigger margin than the vertical line. The points that are closest to the hyperplane are said to lie on the margins, and are called *support vectors*—these are the only points that matter in the classification process itself. Note however that all other points have been used in setting up the model, i.e., in determining which points are support vectors in the first place. The fact that only a limited number of points is used in the predictions for new data is called the *sparseness* of the model, an attractive property in that it focuses attention to the region that matters, the boundary between the classes, and ignores the exact positions of points far from the battlefield.

More formally, a separating hyperplane can be written as

$$\boldsymbol{w}\boldsymbol{x} - b = 0 \tag{7.20}$$

The margin is the distance between two parallel hyperplanes with equations

$$\boldsymbol{w}\boldsymbol{x} - b = -1 \tag{7.21}$$

$$\boldsymbol{w}\boldsymbol{x} - b = 1 \tag{7.22}$$

and is given by $2/||\boldsymbol{w}||$. Therefore, maximizing the margin comes down to minimizing $||\boldsymbol{w}||$, subject to the constraint that no data points fall within the margin:

$$c_i(\boldsymbol{w}\boldsymbol{x}_i) \leq 1 \tag{7.23}$$

where c_i is either -1 or 1, depending on the class label. This is a standard quadratic programming problem.

It can be shown that these equations can be rewritten completely in terms of inner products of the support vectors. This so-called *dual representation* has the big advantage that the original dimensionality of the data is no longer of interest: it does not really matter whether we are analyzing a data matrix with two columns, or a data matrix with ten thousand columns. By applying suitable *kernel functions*, one can transform the data, effectively leading to a representation in higher-dimensional space. Often, a simple discrimination function can be obtained in this high-dimensional space, which translates into an often complex class boundary in the original space. Because of the dual representation, one does not need to know the exact transformation—it suffices to know that it exists, which is guaranteed by the use of kernel functions with specific properties. Examples of suitable kernels are the polynomial and gaussian kernels. More details can be found in the literature (e.g., Hastie et al. 2001).

Package **e1071** provides an interface to the LIBSVM library[3] through the function `svm`. Autoscaling is applied by default. Modelling the Barbera and Grignolino classes leads to the following results:

```
> wns.df <-
+   data.frame(vint = vnt,
+              flavonoids = wns[,"flavonoids"],
+              proline = wns[,"proline"])
> wns.svm <- svm(vint ~ ., data = wns.df[wines.odd2, ])
> wns.svmpred <- predict(wns.svm, wns.df[wines.even2, ])
> table(wns.df$vint[wines.even2], wns.svmpred)
            wns.svmpred
             Barbera Grignolino
  Barbera         22          2
  Grignolino       4         31
```

These default settings lead to a reasonable of the test set.

One attractive feature of SVMs is that they are able to handle fat data matrices (where the number of features is much larger than the number of objects) without any problem. Let us see, for instance, how the standard SVM performs on the prostate data. We will separate the cancer samples from the other control class—again, we are considering only the first 1000 variables. Using the `cross = 10` argument, we

[3] See http://www.csie.ntu.edu.tw/~cjlin/libsvm/.

perform ten-fold crossvalidation, which should give us some idea of the performance on the test set:

```
> prost.svm <- svm(type ~ ., data = prost.df, subset = prost.odd,
+                  cross = 10)
> summary(prost.svm)

Call:
svm(formula = type ~ ., data = prost.df, cross = 10, subset = prost.odd)


Parameters:
   SVM-Type:  C-classification
 SVM-Kernel:  radial
       cost:  1
      gamma:  0.001

Number of Support Vectors:  88

 ( 38 50 )


Number of Classes:  2

Levels:
 control pca

10-fold cross-validation on training data:

Total Accuracy: 92
Single Accuracies:
 83.333 84.615 100 92.308 100 84.615 100 100 91.667 84.615
```

This summary shows us that rather than the complete training set of 125 samples, only 88 are seen as support vectors (for SVMs already quite a large fraction). The prediction accuracies for the left out segments vary from 83 to 83%, with an overall error estimate of 92%. Let us see whether the test set can be predicted well:

```
> prost.svmpred <- predict(prost.svm, newdata = prost.df[prost.even,])
> table(prost.type[prost.even], prost.svmpred)
         prost.svmpred
          control pca
  control      33   7
  pca           1  83
```

Six misclassifications out of 124 cases, nicely in line with the crossvalidation error estimate, is better than anything we have seen so far—not a bad result for default settings.

7.4.1.1 Extensions to More than Two Classes

The fact that only two-class situations can be tackled by basic forms of SVMs is a severe limitation: in reality, it often happens that we should discriminate between several classes. The standard approach is to turn one multi-class problem into several two-class problems. More specifically, one can perform one-against-one testing, where every combination of single classes is assessed, or one-against-all testing. In the latter case, the question is rephrased as: "to be class A or not to be class A"—the advantage is that, in the case of n classes, only n comparisons need to be made, whereas in the one-against-one case $n(n-1)/2$ models must be fitted. The disadvantage is that the class boundaries may be much more complicated: class "not A" may be very irregular in shape. The default in the function `svm` is to assess all one-against-one classifications, and use a voting scheme to pinpoint the final winning class.

To show how this works we again concentrate on two dimensions only so that we can visualize the results. First we set up the SVM model using the odd-numbered rows only:

```
> wines.svm <- svm(vint ~ flavonoids + proline, data = wines.df,
+                  subset = wines.odd)
> wines.svmpred.trn <- predict(wines.svm)
> wines.svmpred.tst <-
+   predict(wines.svm, newdata = wines.df[wines.even, ])
> sum(wines.svmpred.trn == vint.trn) / length(wines.odd)
[1] 0.91011
> sum(wines.svmpred.tst == vint.tst) / length(wines.even)
[1] 0.90909
```

Predictions are very good, both for the training data (the odd rows of the data frame) and the test data (the even rows). Next, we can plot the class boundaries, and project the values of the test data on top to get a visual impression:

```
> plot(wines.svm, wines.df[wines.even, ], proline ~ flavonoids,
+      color.palette = softbrg)
```

This code leads to the left plot in Fig. 7.12. The background colours, indicate the predicted class for each region in the plot. They are obtained in a way very similar to the code used to produce the contour lines in Fig. 7.3 and similar plots. Plotting symbols show the positions of the support vectors—these are shown as crosses, whereas regular data points, unimportant for this SVM model, are shown as open circles. The 8 misclassifications can easily be spotted in the figure.

7.4.1.2 Finding the Right Parameters

The biggest disadvantage of SVMs is the large number of tuning parameters. One should choose an appropriate kernel, and, depending on this kernel, values for two

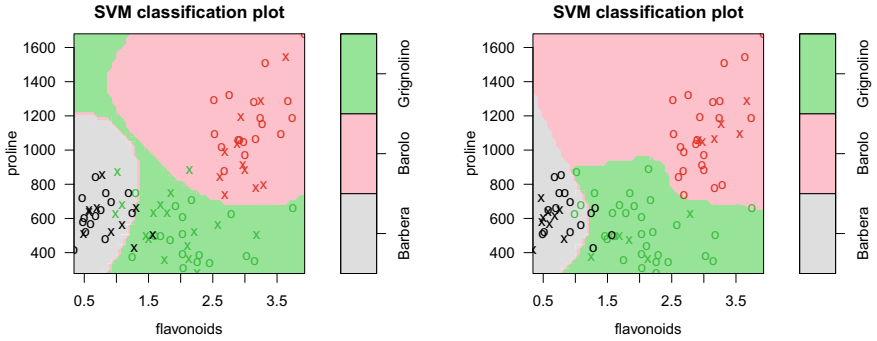

Fig. 7.12 SVM classification plots for the two-dimensional wine data (training data only). Support vectors are indicated by crosses; regular data points by open circles. Left plot: default settings of `svm`. Right plot: best SVM model with a polynomial kernel, obtained with `best.svm`

or three parameters. A special convenience function, `tune`, is available in the **e1071** package, which, given a choice of kernel, varies the settings over a grid, calculates validation values such as crossvalidated prediction errors, and returns an object of class `tune` containing all validation results. A related function is `best` which returns the model with the best validation performance. If we wanted to find the optimal settings for the three parameters `coef0`, `gamma` and `cost` using a polynomial kernel (the default kernel is a radial basis function), we could do it like this:

```
> set.seed(7)
> wines.bestsvm <-
+   best.svm(vint ~ flavonoids + proline, data = wines.df,
+            kernel = "polynomial",
+            coef0 = seq(-.5, .5, by = .1),
+            gamma = 2^(-1:1), cost = 2^(2:4))
```

The predictions with these settings then lead to the following results:

```
> wines.bestsvmpred.trn <-
+   predict(wines.bestsvm, newdata = wines.df[wines.odd, ])
> wines.bestsvmpred.tst <-
+   predict(wines.bestsvm, newdata = wines.df[wines.even, ])
> sum(wines.bestsvmpred.trn == vint.trn) / length(vint.trn)
[1] 0.92135
> sum(wines.bestsvmpred.tst == vint.tst) / length(vint.tst)
[1] 0.92045
```

For both the training and test data, one fewer misclassification error is made; however, the classification plot, shown in the right of Fig. 7.12 looks quite different from the earlier version. The differences in areas where no samples are present may seem not particularly interesting—however, they may become very relevant when new samples are classified. Note that also the number and position of support vectors is quite different.

7.4.2 *Artificial Neural Networks*

Artificial Neural Networks (ANNs, also shortened to neural networks, NNs) form a family of extremely flexible modelling techniques, loosely based on the way neurons in human brains are thought to be connected—hence the name. Although the principles of NNs had already been defined in the fifties of the previous century with Rosenblatt's perceptron (Rosenblatt 1962), the technique only really caught on some twenty years later with the publication of Rumelhart's and McClellands book (Rumelhard and McClelland 1986). Many different kinds of NNs have been proposed; here, we will only treat the flavor that has become known as *feed-forward neural networks*, *backpropagation networks*, after the name of the training rule (see below), or *multi-layer perceptrons*.

Such a network consists of a number of units, typically organized in three layers, as shown in Fig. 7.13. When presented with input signals s_i, a unit will give an output signal s_o corresponding to a transformation of the sum of the inputs:

$$s_o = f\left(\sum_i s_i\right) \tag{7.24}$$

For the units in the input layer, the transformation is usually the identity function, but for the middle layer (the *hidden layer* typically sigmoid transfer functions or

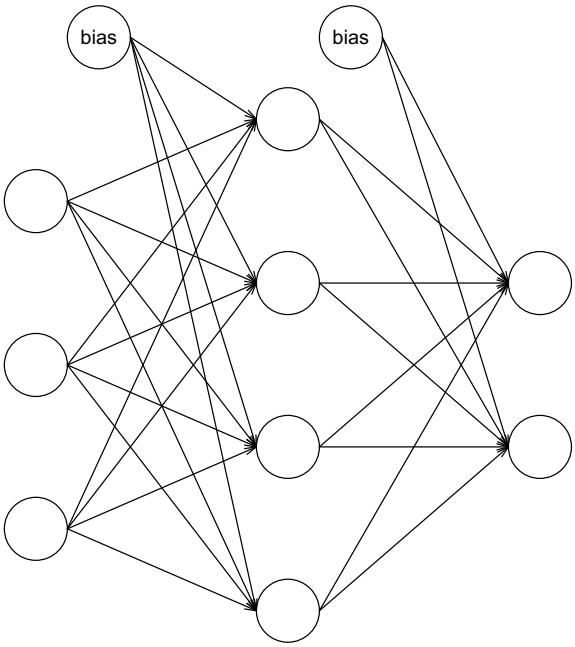

Fig. 7.13 The structure of a feedforward NN with three input units, four hidden units, two bias units and two output units

threshold functions are used. For the hidden and output layers, special *bias units* are traditionally added, always having an output signal of +1 (Ripley 1996). Network structure is very flexible. It is possible to use multiple hidden layers, remove links between specific units, to add connections skipping layers, or even to create feedback loops where output is again fed to special input units. However, the most common structure is to have a fully connected network such as the one depicted in Fig. 7.13, consisting of one input layer, one hidden layer and one output layer. One can show that adding more hidden layers will not lead to better predictions (although in some cases it is reported to speed up training). Whereas the numbers of units in the input and output layers are determined by the data, the number of units in the hidden layer is a parameter that must be optimized by the user.

Connections between units are weighted: an output signal from a particular unit is sent to all connected units in the next layer, multiplied by the respective weights. These weights, in fact form the model for a particular network topology—training the network comes down to finding the set of weights that gives optimal predictions. The most popular training algorithm is based on a steepest-descent based adaption of the weights upon repeated presentation of training data. The gradient is determined by what is called the *backpropagation* rule, a simple application of the chain rule in obtaining derivatives. Many other training algorithms have been proposed as well.

In R, several packages are available providing general neural network capabilities, such as **AMORE** and **neuralnet** (Günther and Fritsch 2010). We will use the **nnet** package, one of the recommended R packages, featuring feed-forward networks with one hidden layer, several transfer functions and possibly skip-layer connections. It does not employ the usual backpropagation training rule but rather the optimization method provided by the R function `optim`. The target values, here the labels of the vintages, have to be presented as a membership matrix, here containing three columns, one for each type of wine. Each row contains `1` at the correct label of the sample, and zeros in the other two positions. The conversion of a factor to a membership matrix is done by the `classvec2classmat` function from the **kohonen** package—below, the first three lines of the membership matrix (all Barolos) are shown:

```
> membership.trn <- classvec2classmat(vint.trn)
> head(membership.trn, 3)
     Barbera Barolo Grignolino
[1,]       0      1          0
[2,]       0      1          0
[3,]       0      1          0
```

For the (autoscaled) training set of the wine data, the network is trained as follows:

```
> wines.nnet <- nnet(x = wines.trn.sc,
+                    y = membership.trn,
+                    size = 4)
# weights:  71
initial  value 64.600576
iter  10 value 28.923577
```

```
iter  20 value 0.012965
iter  30 value 0.002048
iter  40 value 0.000696
iter  50 value 0.000530
final  value 0.000096
converged
```

Although the autoscaling is not absolutely necessary (the same effect can be reached by using different weights for the connections of the input units to the hidden layer) it does make it easier for the network to reach a good solution—without autoscaling the data, the optimization easily gets stuck in a local optimum. Here convergence is reached very quickly. In practice, multiple training sessions should be performed, and the one with the smallest (crossvalidated) training error should be selected. An alternative is to use a (weighted) prediction using all trained networks.

As expected for such a flexible fitting technique, the training data are reproduced perfectly:

```
> membership.pred <- predict(wines.nnet)
> training.pred <- classmat2classvec(membership.pred)
> table(vint.trn, training.pred)
             training.pred
vint.trn      Barbera Barolo Grignolino
  Barbera          24      0          0
  Barolo            0     29          0
  Grignolino        0      0         36
```

Luckily, also the test data are predicted very well here:

```
> table(vint.tst,
+       classmat2classvec(predict(wines.nnet, wines.tst.sc)))

vint.tst      Barbera Barolo Grignolino
  Barbera          24      0          0
  Barolo            0     29          0
  Grignolino        1      1         33
```

Several default choices have been made under the hood of the `nnet` function: the type of transfer functions in the hidden layer and in the output layer, the number of iterations, whether least-squares fitting or maximum likelihood fitting is done (default is least-squares), and several others. The only explicit setting in this example is the number of units in the hidden layer, and this immediately is the most important parameter, too. Choosing too many units will lead to a good fit of the training data but potentially bad generalization—overfitting. Too few hidden units will lead to a model that is not flexible enough.

A convenience function `tune.nnet` is available in package **e1071**, similar to `tune.svm`. This will test neural networks of a specific architecture a number of times (the default is five) and collect measures of predictive performance (obtained by either crossvalidation or bootstrapping, see Chap. 9). Let us see whether our

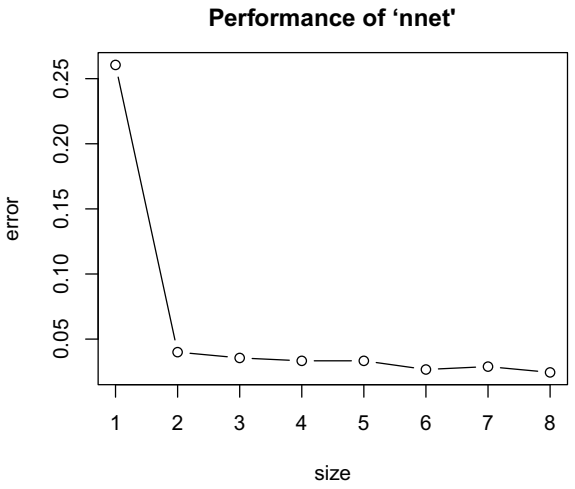

Fig. 7.14 Tuning neural networks: selecting the optimal number of nodes in the hidden layer

(arbitrary) choice of four hidden units can be improved upon, now using the formula interface of the `nnet` function:

```
> wines.trn.sc.df <- data.frame(vintage = vint.trn, wines.trn.sc)
> (wines.nnetmodels <-
+    tune.nnet(vintage ~ ., data = wines.trn.sc.df,
+              size = 1:8, trace = FALSE))
```

Generic `summary` and a `plot` methods are available—for the corresponding plot, see Fig. 7.14. Clearly, one hidden unit is not enough, and two hidden units are not much worse than four, or even eight (although changing the y scale could make us rethink that statement). Instead of using `tune.nnet`, one can also apply `best.nnet`—this function directly returns the trained model with the optimal parameter settings:

```
> best.wines.nnet <-
+   best.nnet(vintage ~ ., data = wines.trn.sc.df,
+             size = 1:8, trace = FALSE)
> table(vint.tst,
+       predict(best.wines.nnet,
+               newdata = data.frame(wines.tst.sc),
+               type = "class"))

vint.tst     Barbera Barolo Grignolino
  Barbera         24      0          0
  Barolo           0     29          0
  Grignolino       4      0         31
```

Indeed, we see one fewer misclassification. Note that here, "optimal" simply means the network with the lowest crossvalidation error. However, this may be too optimistic: especially with very flexible models like ANNs and the tree-based methods

we saw earlier, overfitting is a real danger. The idea is that too complex models have enough flexibility to learn data "by heart" whereas models of the right complexity are forced to focus on more general principles. One rule of thumb is to use the simplest possible model that is not worse than the best model to be as conservative as possible. In this case, one would expect a network with only two hidden neurons to perform better for new, unseen, data than a network with four or eight. We will come back to this in Chap. 9.

In the example above we used the default stopping criterion of the `nnet` function, which is to perform 100 iterations of complete presentations of the training data. In several publications, scientists have advocated continuously monitoring prediction errors throughout the training iterations, in order to prevent the network from overfitting. In this approach, training should be stopped as soon as the error of the validation set starts increasing. Apart from the above-mentioned training parameters, this presents an extra level of difficulty which becomes all the more acute with small data sets. To keep these problems manageable, one should be very careful in applying neural networks in situations with few cases; the more examples, the better.

7.4.2.1 Deep Learning

Since 2010, a novel development in neural networks called Deep Learning (DL, Goodfellow et al. 2016) has taken center stage with applications in areas like computer vision (Uijlings et al. 2013; Gatys et al. 2016; Badrinarayanan et al. 2017), speech recognition (Hinton et al. 2012; Deng et al. 2013; Nassif et al. 2019) and many others. At that point in time, developments in GPUs, graphics processing units allowing massively parallel computations coincided with easy access to large data sets such as ImageNet (Russakovsky et al. 2015), a collection of millions of annotated images. Open-source software was available, too, and the interest of companies like Google and Microsoft made sure large steps were made. Today, many of the top-performing approaches in difficult benchmark problems are based on Deep Learning.

So what is different, compared to the neural networks in the previous sections? From a structural viewpoint, not that much. Just like the neural networks from the nineties can be seen as perceptrons stitched together in a particular structure, DL networks can be described as collections of neural networks such as the ones in Fig. 7.13. What *is* new is that many more layers are used (DL networks with more than one hundred layers are no exception) and that layers are included with a purpose: in image processing applications we typically see, amongst others, convolutional layers and pooling layers, applied in alternating fashion. Each of these layers serves a particular purpose—subsequent layers are not fully connected like in the network of Fig. 7.13 but only connected if there is a reason for it. In this way, the DL network is able to aggregate the raw input data into more and more abstract features that eventually will be combined to obtain the final answer. The increased amount of structure within the DL net restricts the number of weights that need to be optimized. Regularization methods (see, e.g., Sects. 8.4 and 10.2) are employed routinely in order to keep the weights small and prevent overfitting (Efron and Hastie 2016), effectively

removing the number of training iterations as a parameter to be optimized: more is better in these cases. Still, the training is a daunting task: many weights are optimized, and for this a large (large!) number of training examples needs to be provided.

A major hurdle for classification applications is that in almost all cases the training examples need to be annotated, i.e., the ground truth needs to be known. Modern sensing devices like cameras have no problems in generating terabytes and more of data, but what the true class of the image is still needs to be decided, an area that is being exploited commercially nowadays. Chemometrics is typically concerned with data characterized by multivariate responses, recorded for relatively few samples, so DL applications are still rare but they will certainly come.