Use R!

Ron Wehrens

Chemometrics with R

Multivariate Data Analysis in the Natural Sciences and Life Sciences

Springer

```
> prost.rprt2pred <-
+   predict(prost.rprt2, newdata = prost.df[even,])
> table(prost.type[even], classmat2classvec(prost.rprt2pred))

          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 (page 121).

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 model form: many scientists feel comfortable with a series of more and more specific questions 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 Section 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 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.

7.4.1 Support Vector Machines

SVMs [77, 78, 79] 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 Figure 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:

e? Do we
between
l to make.
h comple-
are very
Networks
available.
ANNs are
the num-
share one
nodelling
n the re-
rameters
, we will
ature for

between
zing the
in SVM
SVMs, is
lid lines,
hieves a
st to the
ectors –
elf. Note
del, i.e.,
The fact
ew data
focuses
ses, and

(7.20)

uations

(7.21)
(7.22)

own to
hin the

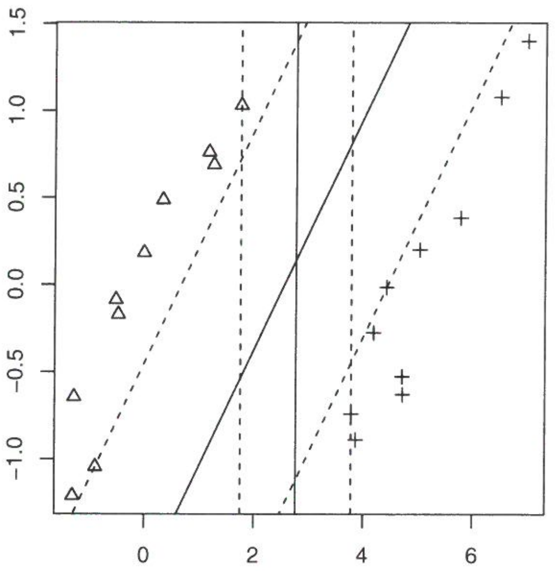

Fig. 7.11. The basic idea behind SVM classification: the separating hyperplane (here, in two dimensions, a line) is chosen in such a way that the margin is maximal. Points on the margins (the dashed lines) are called "support vectors". Clearly, the margins for the separating line with slope 2/3 are much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

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

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 analysing 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., [3]).

Package **e1071** provides an interface to the LIBSVM library[3] through the function `svm`. Application to the Barbera and Grignolino classes leads to the following results:

```
> wines.df <-
+   data.frame(vint = factor(vintages[vintages != "Barolo"]),
+              X = wines[vintages != "Barolo",])
> wines.svm <- svm(vint ~ ., data = wines.df, subset = odd)
> wines.svmpred <- predict(wines.svm, newdata = wines.df[even,])
> table(vint[even], wines.svmpred)

            wines.svmpred
             Barbera Grignolino
  Barbera         24          0
  Grignolino       0         35
```

These default settings lead to a perfect classification of the test set.

One attractive feature of SVMs is 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 perform ten-fold crossvalidation, which should give us some idea of the performance on the test set:

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

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

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

```
10-fold crossvalidation on training data:
Total Accuracy: 92
Single Accuracies:
 100 100 83.3 84.6 100 76.9 91.7 84.6 100 100
```

The summary (slightly edited to save space) shows us that rather than the complete training set of 125 samples, only 88 objects are seen as support vectors, which for SVMs is quite a large fraction. The prediction accuracies for the left out segments vary from 77 to 100%, with an overall error estimate of 8%. Let us see whether the test set can be predicted well:

```
> prost.svmpred <- predict(prost.svm, newdata = prost.df[even,])
> table(prost.type[even], prost.svmpred)

         prost.svmpred
          control pca
  control      33   7
  pca           1  83
```

Eight 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.

Extensions to More than Two Classes

The fact that only two-class situations can be tackled 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 $.5n(n-1)$ 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.

```
> plot(wines.svm, wines.df[odd,], proline ~ flavonoids)
```

This leads to the left plot in Figure 7.12. The background colours indicate the predicted class for each region in the plot, obtained in a way very similar to the code used to produce the contour lines in Figure 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.

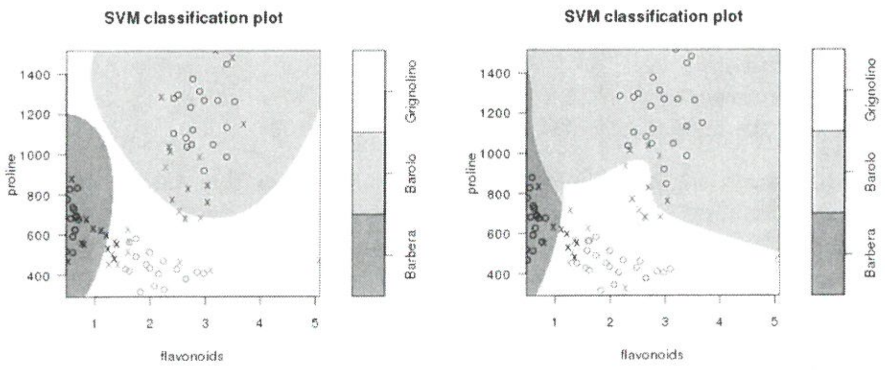

Fig. 7.12. SVM classification plots for the two-dimensional wine data (training data only). Support vectors are indicated by crosses; regular data points by open circles. Left plot: default settings of `svm`. Right plot: best SVM model with a polynomial kernel, obtained with `best.svm`.

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 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:

```
> wines.bestsvm <-
+   best.svm(vintages ~ ., data = wines.dfodd,
+            kernel = "polynomial",
+            coef0 = seq(-.5, .5, by = .1),
+            gamma = 2^(-1:1), cost = 2^(2:4))
> wines.bestsvmpred <-
+   predict(wines.bestsvm, newdata = wines.df[even,])
> sum(wines.bestsvmpred == vintages[even])
```

```
[1] 80
```

The number of correct classifications is exactly the same as with the default SVM parameters; however, the classification plot, shown in the right of Figure 7.12 looks completely different. Differences are mainly located in areas where no samples are present, but in some cases are also present in more relevant parts – consider, for example, the centers of the figures. This presents a

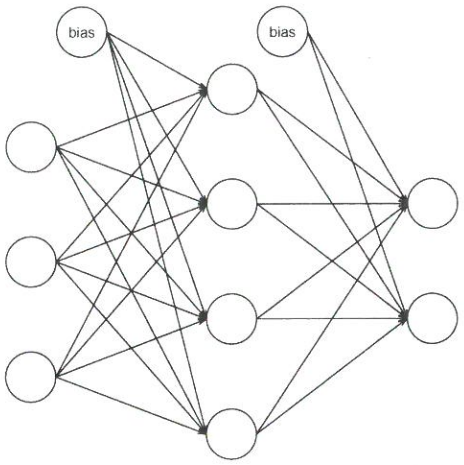

Fig. 7.13. The structure of a feedforward NN with three input units, four hidden units and two output units.

sobering illustration of the dangers we face when we trust complicated models trained with relatively few data points. Note that also the number and position of support vectors is radically 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 [80], the technique only really caught on some twenty years later with the publication of Rumelhart's and McClellands book [81]. Many different kinds of NNs have been proposed; here, we will only treat the flavour 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 Figure 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:

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

For the units in the input layer, the transformation is usually the identity function, but for the middle layer (the *hidden layer* in NN terminology) typically sigmoid transfer functions or threshold functions are used. For the hidden and output layers, special *bias units* are traditionally added, always having an output signal of +1 [34]. 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 Figure 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 called the *backpropagation* algorithm, and consists of a steepest-descent based adaption of the weights upon repeated presentation of training data. 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** [82]. 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`.

For the autoscaled wine data, training a neural net with four units in the hidden layer is done as follows:

```
> X <- scale(wines, scale = sd(wines[odd,]),
+             center = colMeans(wines[odd,]))
> w.df <- data.frame(vintage = vintages, wines = X)
> wines.nnet <- nnet(vintage ~ ., data = w.df,
+                     size = 4, subset = odd)

# weights:  71
initial  value 109.648958
iter  10 value 0.812789
iter  20 value 0.002502
final  value 0.000093
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 – the optimization easily gets stuck in a local optimum. 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 flexibly fitting technique, the training data are reproduced perfectly:

```
> training.pred <- predict(wines.nnet, type = "class")
> sum(diag(table(vintages[odd],
+                training.pred))) / length(odd)

[1] 1
```

But also the test data in this case are predicted very well – only three errors are made:

```
> table(vintages[even],
+       classmat2classvec(predict(wines.nnet, X[even,])))

             Barbera Barolo Grignolino
  Barbera         23      0          1
  Barolo           0     29          0
  Grignolino       2      0         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**. Let us see whether our (arbitrary) choice of four hidden units can be improved upon:

```
> wines.nnetmodels <-
+   tune.nnet(vintage ~ ., data = w.df[odd,], size = 1:8)
> summary(wines.nnetmodels)

Parameter tuning of `nnet':
- sampling method: 10-fold cross validation

- best parameters:
 size
    2

- best performance: 0
- Detailed performance results:
  size   error dispersion
1    1 0.12361    0.09721
2    2 0.00000    0.00000
3    3 0.02222    0.04685
4    4 0.01111    0.03514
5    5 0.01111    0.03514
6    6 0.01111    0.03514
7    7 0.01111    0.03514
8    8 0.01111    0.03514
```

Clearly, one hidden unit is not enough; two hidden units apparently suffice. In addition to the `summary` function, a `plot` method is available as well. 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 = w.df[odd,],
+             size = 1:8)
> table(vintages[even],
+       predict(best.wines.nnet, w.df[even,], type = "class"))

             Barbera Barolo Grignolino
  Barbera         23      0          1
  Barolo           0     29          0
  Grignolino       2      5         28
```

The result is quite a bit worse than the network with four hidden units shown above. It illustrates immediately the one major problem with applying neural networks to relatively small data sets: the variability of the training procedures is so large that one only has a faint idea of what optimal settings to apply. The golden rule of thumb is to use the smallest possible models. In this case, one would expect a network with only two hidden neurons to perform better than one with four, and perhaps, over a large number of training events, this may actually be the case. For individual training runs, however, it is not at all uncommon to see the kind of unexpected behaviour shown above.

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.