Part IX
Discriminant Methods – Error Rate Estimation, and Low-Dimensional Views

1 rpart Analyses – the Combined `Pima` Dataset

Note the rpart terminology:

size	Number of leaves	Used in plots from `plotcp()`
`nsplit`	Number of splits = size - 1	Used in `printcp()` output
`cp`	Complexity parameter	Appears as CP in graphs and printed output. A smaller `cp` gives a more complex model, i.e., more splits.
rel error	Resubstitution error measure	Multiply by baseline error to get the corresponding absolute error measure. In general, treat this error measure with scepticism.
xerror	Crossvalidation error estimate	Multiply by baseline error to get the corresponding absolute error measure.

After loading the *MASS* package, type `help(Pima.tr)` to get a description of these data. They are relevant to the investigation of conditions that may pre-dispose to diabetes.

The `Pima` data frame combines `Pima.tr` and `Pima.te` from the *MASS* package, thus:

```
> library(e1071)
> library(MASS)
> Pima <- rbind(Pima.tr, Pima.te)
```

1.1 Fitting the model

Fit an rpart model to the `Pima` data:

```
> library(rpart)
> Pima.rpart <- rpart(type ~ ., data = Pima, method = "class")
> plotcp(Pima.rpart)
```

The formula `type` $\sim$ `.` has the effect of using as explanatory variables all columns of `Pima` except `type`. The parameter cp is a complexity parameter; it penalizes models that are too complex. A small penalty leads to more splits. Note that cp, at this initial fit, has to be small enough so that the minimum of the cross-validated error is attained.

Try this several times. The result will vary somewhat from run to run. Why?

One approach is to choose in a manner that makes the model on average best, i.e., choose the absolute minimum of the cross-validated error. If the fitting has been repeated several times, the cross-validation errors can be averaged over the separate runs.

A more cautious approach is to choose a model where there is some modest certainty that the final split is giving a better than chance improvement. For this, choose the smallest number of splits so that cross-validated error rate lies under the dotted line. For example, in one of my runs, this happened for cp=0.038, with size=4. The resulting tree can be cut back to this size with:

```
> Pima.rpart4 <- prune(Pima.rpart, cp = 0.037)
```

The value of cp is chosen to be less than 0.038, but more than the value that led to a further split.

Plot the tree, thus:

```
> plot(Pima.rpart4)
> text(Pima.rpart4)
```

Note also the printed output from

```
> printcp(Pima.rpart)

Classification tree:
rpart(formula = type ~ ., data = Pima, method = "class")

Variables actually used in tree construction:
[1] age   bmi   glu   npreg ped

Root node error: 177/532 = 0.33271

n= 532

        CP nsplit rel error  xerror     xstd
1 0.265537      0   1.00000 1.00000 0.061400
2 0.056497      1   0.73446 0.77966 0.057116
3 0.025424      3   0.62147 0.74011 0.056141
4 0.020716      5   0.57062 0.70621 0.055249
5 0.014124      9   0.48588 0.70621 0.055249
6 0.011299     11   0.45763 0.74576 0.056284
7 0.010000     14   0.42373 0.75706 0.056567
```

Get the absolute cross-validated error by multiplying the root node error by xerror. With `nsplit=3` (a tree of size 4 leaves), this is, in the run I did, 0.3327*0.7006 = 0.233. (The accuracy is obtained by subtracting this from 1.0, i.e., about 77%.)

Where it is not clear from the graph where the minimum (or the minimum+SE, if that is used) lies, it will be necessary to resort to use this printed output for that purpose. It may be necessary to use a value of `cp` that is smaller than the default in the call to `rpart()`, in order to be reasonably sure that the optimum (according to one or other criterion) has been found.

Exercise 1: Repeat the above several times, i.e.

```
> library(rpart)
> Pima.rpart <- rpart(type ~ ., data = Pima, method = "class")
> plotcp(Pima.rpart)
```

(a) Overlay the several plots of the cross-validated error rate against the number of splits. Why does the cross=validated error rate vary somewhat from run to run? Average over the several runs and choose the optimum size of tree based on (i) the minimum cross-validated error rate, and (ii) the minimum cross-validated error rate, plus one SE.

(b) Show the relative error rate on the same graph.

NB: You can get the error rate estimates from:

```
> errmat <- printcp(Pima.rpart)

Classification tree:
rpart(formula = type ~ ., data = Pima, method = "class")

Variables actually used in tree construction:
[1] age   bmi   glu   npreg ped

Root node error: 177/532 = 0.33271

n= 532
```

```
        CP nsplit rel error  xerror     xstd
1 0.265537      0   1.00000 1.00000 0.061400
2 0.056497      1   0.73446 0.80791 0.057772
3 0.025424      3   0.62147 0.72316 0.055701
4 0.020716      5   0.57062 0.68927 0.054783
5 0.014124      9   0.48588 0.66667 0.054139
6 0.011299     11   0.45763 0.64407 0.053470
7 0.010000     14   0.42373 0.64972 0.053640

> colnames(errmat)

[1] "CP"        "nsplit"    "rel error" "xerror"    "xstd"

> resub.err <- 1 - 0.3327 * errmat[, "rel error"]
> cv.err <- 1 - 0.3327 * errmat[, "xerror"]

]
```

Exercise 2: Prune the model back to give the optimum tree, as determined by the one SE rule. How does the error rate vary with the observed value of `type`? Examine the confusion matrix. This is most easily done using the function `xpred()`. (The following assumes that 3 leaves, i.e., `cp` less than about 0.038 and greater than 0.011, is optimal.)

```
> Pima.rpart <- prune(Pima.rpart, cp = 0.037)
> cvhat <- xpred.rpart(Pima.rpart4, cp = 0.037)
> tab <- table(Pima$type, cvhat)
> confusion <- rbind(tab[1, ]/sum(tab[1, ]), tab[2, ]/sum(tab[2,
+     ]))
> dimnames(confusion) <- list(ActualType = c("No", "Yes"), PredictedType = c("No",
+     "Yes"))
> print(confusion)

          PredictedType
ActualType        No       Yes
       No  0.8647887 0.1352113
       Yes 0.4971751 0.5028249
```

The table shows how the predicted accuracy changes, depending on whether the correct type is `Yes` or `No`.

How would you expect the overall estimate of predictive accuracy to change, using the same fitted rpart model for prediction:

- if 40% were in the `Yes` category?
- if 20% were in the `Yes` category?

2 rpart Analyses – Pima.tr and Pima.te

Exercise 3 These exercises will use the two data sets `Pima.tr` and `Pima.te`, and carrying out a variation on the calculations of Exercise 2.

What are the respective proportions of the two types in the two data sets?

Exercise 3a: Repeat Exercise 1, but now use the data frame `Pima.tr` to determine the model, and comparing cross-validation accuracies from calculations with `Pima.tr` with accuracies when `Pima.te` is used as the test data:

```
> trPima.rpart <- rpart(type ~ ., data = Pima.tr, method = "class")
> plotcp(trPima.rpart)
> printcp(trPima.rpart)

Classification tree:
rpart(formula = type ~ ., data = Pima.tr, method = "class")

Variables actually used in tree construction:
[1] age bmi bp  glu ped

Root node error: 68/200 = 0.34

n= 200

        CP nsplit rel error  xerror     xstd
1 0.220588      0   1.00000 1.00000 0.098518
2 0.161765      1   0.77941 0.88235 0.095305
3 0.073529      2   0.61765 0.79412 0.092331
4 0.058824      3   0.54412 0.79412 0.092331
5 0.014706      4   0.48529 0.66176 0.086846
6 0.010000      7   0.44118 0.75000 0.090647
```

as above, e.g.

```
> trPima.rpart <- prune(trPima.rpart, cp = 0.02)
```

How does the error rate vary between the `No`'s and `Yes`'s. How does the expected error rate vary with the proportion of `No`'s in the target population?

Exercise 3b: Refit the model. Choose values of `cp` that will give the points on the graph obtained from `plotcp(trPima.rpart)`. Suitable values of `cp` are those that appear on the graph given by `plotcp()`. These (except for the first; what happens there?) are the geometric means of the successive pairs that are printed, and can be obtained thus:

```
> trPima.rpart <- rpart(type ~ ., data = Pima.tr, method = "class")
> cp.all <- printcp(trPima.rpart)[, "CP"]

Classification tree:
rpart(formula = type ~ ., data = Pima.tr, method = "class")

Variables actually used in tree construction:
[1] age bmi bp  glu ped

Root node error: 68/200 = 0.34

n= 200

        CP nsplit rel error  xerror     xstd
1 0.220588      0   1.00000 1.00000 0.098518
2 0.161765      1   0.77941 1.00000 0.098518
3 0.073529      2   0.61765 0.76471 0.091224
4 0.058824      3   0.54412 0.77941 0.091785
5 0.014706      4   0.48529 0.66176 0.086846
6 0.010000      7   0.44118 0.75000 0.090647
```

```
> n <- length(cp.all)
> cp.all <- sqrt(cp.all * c(Inf, cp.all[-n]))
> nsize <- printcp(trPima.rpart)[, "nsplit"] + 1

Classification tree:
rpart(formula = type ~ ., data = Pima.tr, method = "class")

Variables actually used in tree construction:
[1] age bmi bp  glu ped

Root node error: 68/200 = 0.34

n= 200

        CP nsplit rel error  xerror     xstd
1 0.220588      0   1.00000 1.00000 0.098518
2 0.161765      1   0.77941 1.00000 0.098518
3 0.073529      2   0.61765 0.76471 0.091224
4 0.058824      3   0.54412 0.77941 0.091785
5 0.014706      4   0.48529 0.66176 0.086846
6 0.010000      7   0.44118 0.75000 0.090647
```

Observe that `nsize` is one greater than the number of splits.

Prune back successively to these points. In each case determine the cross-validated error for the training data and the error for test data. Plot both these errors against the size of tree, on the same graph. Are they comparable? The following will get you started:

```
> tr.cverr <- printcp(trPima.rpart)[, "xerror"] * 0.34

Classification tree:
rpart(formula = type ~ ., data = Pima.tr, method = "class")

Variables actually used in tree construction:
[1] age bmi bp  glu ped

Root node error: 68/200 = 0.34

n= 200

        CP nsplit rel error  xerror     xstd
1 0.220588      0   1.00000 1.00000 0.098518
2 0.161765      1   0.77941 1.00000 0.098518
3 0.073529      2   0.61765 0.76471 0.091224
4 0.058824      3   0.54412 0.77941 0.091785
5 0.014706      4   0.48529 0.66176 0.086846
6 0.010000      7   0.44118 0.75000 0.090647
```

```
> n <- length(cp.all)
> trPima0.rpart <- trPima.rpart
> te.cverr <- numeric(n)
> for (i in n:1) {
+     trPima0.rpart <- prune(trPima0.rpart, cp = cp.all[i])
+     hat <- predict(trPima0.rpart, newdata = Pima.te, type = "class")
+     tab <- table(hat, Pima.te$type)
+     te.cverr[i] <- 1 - sum(tab[row(tab) == col(tab)])/sum(tab)
+ }
```

of splits.

3 Analysis Using *svm*

Exercise 4: Compare the result also with the result from an SVM (Support Vector Machine) model. For getting predictions for the current test data, it will be necessary, for models fitted using `svm()`, to use the `newdata` argument for `predict()`. Follow the prototype

```
> library(e1071)
> library(MASS)
> trPima.svm <- svm(type ~ ., data = Pima.tr)
> hat <- predict(trPima.svm, newdata = Pima.te)
> tab <- table(Pima.te$type, hat)
> 1 - sum(tab[row(tab) == col(tab)])/sum(tab)
> confusion.svm <- rbind(tab[1, ]/sum(tab[1, ]), tab[2, ]/sum(tab[2,
+     ]))
> print(confusion.svm)
```

4 Analysis Using *randomForest*

Exercise 5: Repeat the previous exercise, but now using `randomForest()`

```
> library(randomForest)
> Pima.rf <- randomForest(type ~ ., data = Pima.tr, xtest = Pima.te[,
+     -8], ytest = Pima.te$type)
> Pima.rf

Call:
 randomForest(formula = type ~ ., data = Pima.tr, xtest = Pima.te[,      -8], ytest = Pima.te$type)
               Type of random forest: classification
                     Number of trees: 500
No. of variables tried at each split: 2

        OOB estimate of  error rate: 28%
Confusion matrix:
     No Yes class.error
No  110  22   0.1666667
Yes  34  34   0.5000000
                Test set error rate: 24.1%
Confusion matrix:
     No Yes class.error
No  190  33   0.1479821
Yes  47  62   0.4311927
```

Note that OOB = Out of Bag Error rate, calculated using a cross-validation type approach with the training data, i.e., with the data specified by the `data` parameter. Notice that `randomForest()` is set up to give, following the one function call, an assessment of predictive error for a completely separate set of test data that has had no role in training the model. Where such a test set is available, this provides a reassuring check that `randomForest()` is not over-fitting.

Exercise 6: The function `tuneRF()` can be used for such limited tuning as `randomForest()` allows. Look up `help(tuneRF()`, and run this function in order to find an optimal value for the parameter `mtry`. Then repeat the above with this optimum value of `mtry`, and again compare the OOB error with the error on the test set.

Exercise 7: Comment on the difference between `rpart()` and `randomForest()`: (1) in the level of automation; (2) in error rate for the data sets for which you have a comparison; (3) in speed of execution.

5 Prior Proportions

Exercise 8: Analysis with and without specification of a prior Try the following:

```
> Pima.rf <- randomForest(type ~ ., data = Pima, method = "class")
> Pima.rf.4 <- randomForest(type ~ ., data = Pima, method = "class",
+     classwt = c(0.6, 0.4))
> Pima.rf.1 <- randomForest(type ~ ., data = Pima, method = "class",
+     classwt = c(0.9, 0.1))
```

What class weights have been implicitly assumed, in the first calculation?

Compare the three confusion matrices. Why are they different?

6 Plots that show the "distances" between points

In analyses with `randomForest`, the proportion of times (over all trees) that any pair of observations ("points") appears at the same terminal node can be use as a measure of proximity between the pair. An ordination method can then be used to find a low-dimensional representation of the points that as far as possible preserves the distances or (for non-metric scaling) preserves the ordering of the distances. Here is an example:

```
> Pima.rf <- randomForest(type ~ ., data = Pima, proximity = TRUE)
> Pima.prox <- predict(Pima.rf, proximity = TRUE)
> Pima.cmd <- cmdscale(1 - Pima.prox$proximity)
> Pima.cmd3 <- cmdscale(1 - Pima.prox$proximity, k = 3)
> library(lattice)
> cloud(Pima.cmd3[, 1] ~ Pima.cmd3[, 2] * Pima.cmd3[, 2], groups = Pima$type)
```

Exercise 9: What is the plot from `cloud()` saying? Why is this not overly surprising?:

7 Discrimination with Multiple Groups

The package *mclust* has the data set `diabetes`. It does not automatically become available when the package is attached; instead, it is necessary to bring it into the workspace by typing

```
> data(diabetes)
```

Here is a 3-dimensional cloud plot:

```
> cloud(insulin ~ glucose + sspg, groups = class, data = diabetes)
```

7.1 Linear discriminant analysis

We will first try linear discriminant analysis. We specify `CV=TRUE`, in order to obtain predictions of class membership that are derived from leave-one-out cross-validation. We then run the calculations a second time, now with `CV=FALSE`, in order to obtain an object from which we can obtain the discriminant scores

```
> diabetes.lda <- lda(class ~ insulin + glucose + sspg, data = diabetes,
+     CV = TRUE)
> diabetes.lda
> tab <- table(diabetes$class, diabetes.lda$class)
> 1 - sum(tab[row(tab) == col(tab)])/sum(tab)
> confusion.lda <- rbind(tab[1, ]/sum(tab[1, ]), tab[2, ]/sum(tab[2,
+     ]), tab[3, ]/sum(tab[3, ]))
> dimnames(confusion.lda) <- list(Actual = c("chemical", "normal",
```

```
+     "overt"), "Predicted (cv)" = c("chemical", "normal", "overt"))
> print(confusion.lda)
> hat.lda <- predict(lda(class ~ insulin + glucose + sspg, data = diabetes))
> plot(hat.lda$x, col = unclass(diabetes$class) + 1)
```

7.2 Support vector machines

Here, we will specify the use of tenfold cross-validation, in order to estimate the error rate. The "decision values" will be used to define co-ordinate axes, so that data can be plotted.

```
> diabetes.svm <- svm(class ~ insulin + glucose + sspg, data = diabetes,
+     cross = 10)
> summary(diabetes.svm)
> pred <- predict(diabetes.svm, diabetes[, -1], decision.values = TRUE)
> decision <- attributes(pred)$decision.values
> decision.cmd <- cmdscale(dist(decision))
> plot(decision.cmd, col = unclass(diabetes$class) + 1)
```

The cross-validated prediction accuracy is less than using `lda()`. There is however substantial scope for using another kernel, and/or setting various tuning parameters. If there is such tuning, care is needed in obtaining an accuracy measure that is not biased on account of the tuning. Two possibilities are: (1) divide the data into training and test sets, and use the test set accuracy rather than the cross-validation accuracy; or (2) repeat the tuning at each cross-validation fold.

Also possible is to use a scaled version of the decision values. For example, the following uses the *MASS* program `isoMDS()` to obtain a 2-dimensional representation. The function `isoMDS()` requires a starting configuration; we use the values from `cmdscale()` for that purpose:

```
> diabetes.mds <- isoMDS(dist(decision), decision.cmd)
```

It turns out that observations 4 and 80 have zero distance, which `isoMDS()` is unable to handle. We therefore omit observation 80.

```
> decision1 <- decision[-80, ]
> diabetes.mds <- isoMDS(dist(decision1), decision.cmd[-80, ])

initial  value 0.454312
iter   5 value 0.387661
final  value 0.384746
converged

> plot(diabetes.mds$points, col = unclass(diabetes$class)[-80] +
+     1)
```

7.3 Use of `randomForest()`

First, fit a tree-based discriminant model, calculating at the same time the proximities:

```
> diabetes.rf <- randomForest(class ~ ., data = diabetes, proximity = TRUE)
> print(diabetes.rf)
> diabetes.rf.cmd <- cmdscale(1 - diabetes.rf$proximity)
> plot(diabetes.rf.cmd, col = unclass(diabetes$class) + 1)
```

The clear separation between the three groups, on this graph, is remarkable. It is however consistent with the OOB error rate of 2%.

Exercise 10: Make a table that compares `lda()`, `svm()` and `randomForest()`, with respect to error rate for the three classes separately.