MATH9952 Modern Applied Statistical Models
ASSIGNMENT MARKING SCHEME

Student No. D16126734 Name: JERRY KIELY

Assignment 3: tree Models.

Part 1:	15
Part 2:	18
Part 3:	40
Total Marks from 100	73

Comments:

Good coding used to perform the analysis - but should address not have been a factor predictor?
Also, it is unclear how missing values in many of the predictors were handled.
Your report reads well, but is somewhat short and omits some technical details on the fitting, pruning and validation parts of the analysis.

Building a Tree Model with the Retention Dataset

Jerry Kiely

11 April 2017

Introduction

This analysis is concerned with data relating to student retention in the Engineering faculty of DIT. It is the same data considered in Assignment 3, but with more potential predictors available.

The purpose of the analysis will be to use rpart models to build a predictive model for student retention. The data includes risk risk factors regarding prior academic performance (e.g. leaving certificate results, leaving certificate maths grade), personal characteristics (gender, home address, CAO choices made etc).

The Algorithm

rpart takes a data set and recursively partitions it based on a splitting criteria - choosing the split that maximizes the reduction in impurity for the node. For example in the case of the root, and splitting on lcpoints, deciding where to split within lcpoints involves:

- iterating over all values of lcpoints in turn
- calculating the reduction in impurity for splitting at each value
- keeping track of the value that results in the maximum impurity reduction
- splitting at the value that results in the maximum imppurity reduction

this process is repeated at every level until other criteria such as maxdepth or minsplit come in to play. The impurity of a node can be calculated using a gini score or an entropy score. An example of this process is given in the code - specifically finding the splitting point for the lcpoints predictor using a gini score.

The Data

First we read the data in. For the sake of readibility we will add a column called "result"" that will hold a string value - "failed" or "passed". Also we will drop the "id" column as unnecessary, and the "overall6" column as redundant. Next we split the data into a training and validation set. The training set will be used to create our model, and the validation set will be used to see how well it performs, and to help with the process of improving it through pruning.

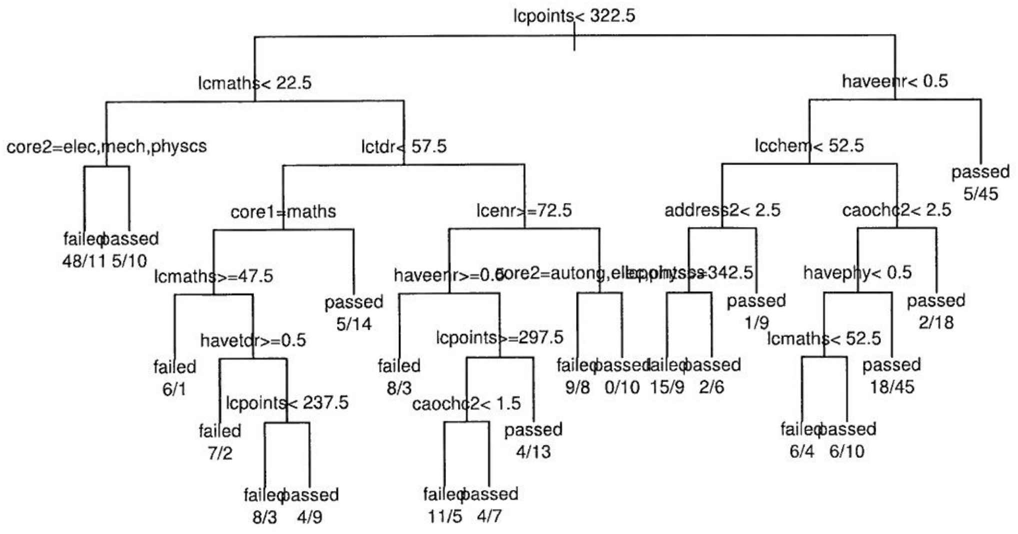

Figure 1: Tree built with default parameters and cp = 0

The Model

We fit a model with all variables included as potential predictors. The default values are used for all parameters except for the complexity parameter, which is set to 0. The plot of the tree for this model is included in Figure 1.

In Figure 2 we see a plot of the complexity parameters for the tree. The value of interest is the left-most value below the dotted line:

- below the dotted line because we are interested in the complexity parameter with the the lowest cross validation error
- left-most because we favour a smaller tree over a deeper, more complex, tree which could suffer from

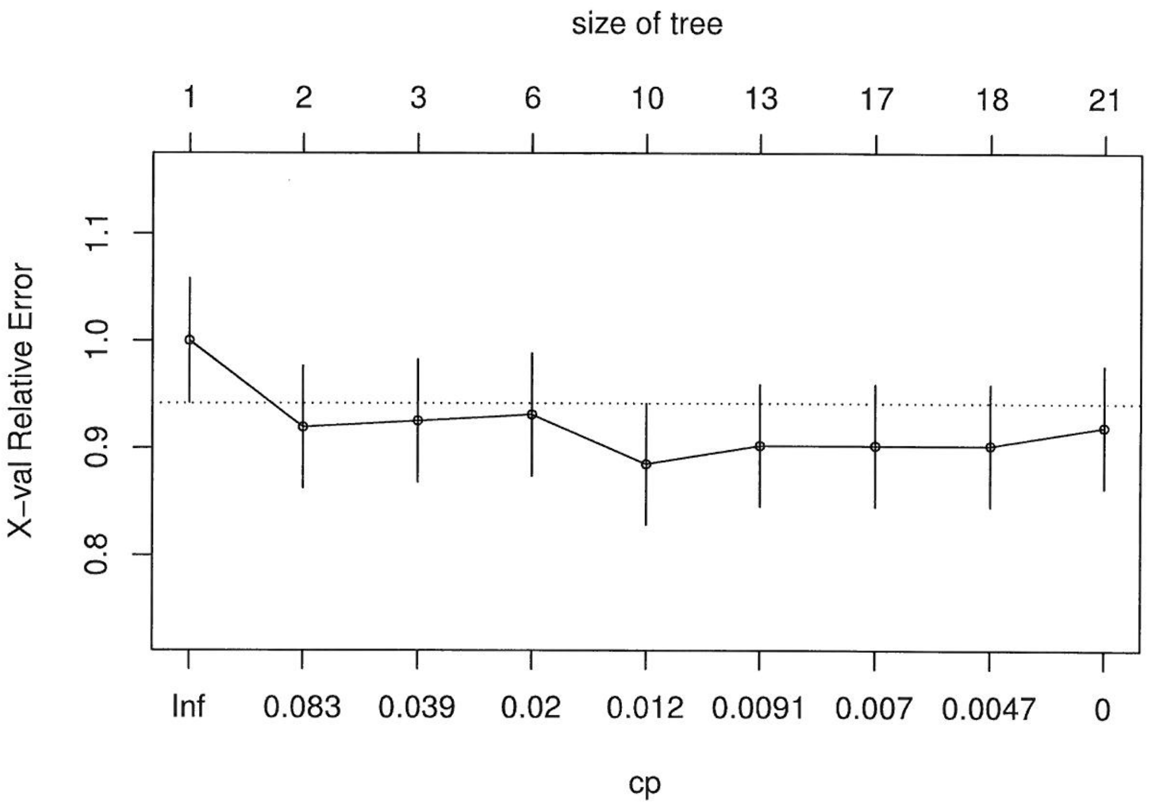

Figure 2: Complexity parameters versus cross validation results versus size of tree

overfitting

so we choose a value of 0.0095785 for our complexity parameter, and prune our tree based on that.

Table 1: table detailing complexity parameters

CP	nsplit	rel error	xerror	xstd
0.1321839	0	1.0000000	1.0000000	0.0578211
0.0517241	1	0.8678161	0.9195402	0.0570275
0.0287356	2	0.8160920	0.9252874	0.0570936
0.0143678	5	0.7298851	0.9310345	0.0571582
0.0095785	9	0.6666667	0.8850575	0.0565998
0.0086207	12	0.6379310	0.9022989	0.0568203
0.0057471	16	0.6034483	0.9022989	0.0568203
0.0038314	17	0.5977011	0.9022989	0.0568203
0.0000000	20	0.5862069	0.9195402	0.0570275

In Figure 3 we see our pruned tree. As can be seen it is a lot less complex than the earlier version.

The Analysis

Now we can analyze the results of our pruning. We will now predict based on our validatition set to see how well our model performs.

Table 2: Confusion matrix for pruned tree

	failed	passed
failed	0.4230769	0.5769231
passed	0.1946903	0.8053097

Looking at the confusion matrix in Table 2 we seem to do a better job of predicting passing than predicting failing - i.e. the true failed is below 0.5 and the true passed well above 0.5. An option might be to consider treating the model as a predictor of successfully passing rather than failing.

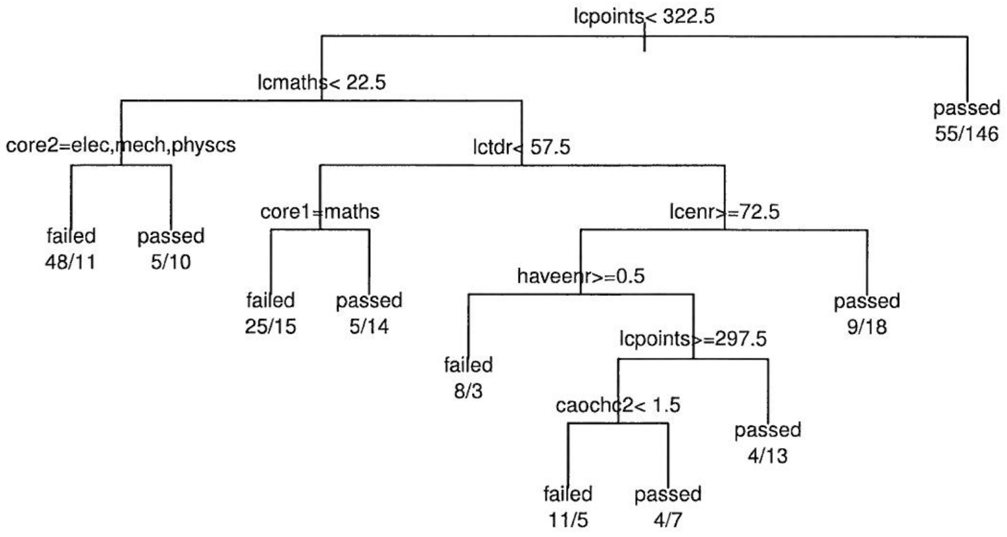

Figure 3: Tree pruned with found complexity parameter

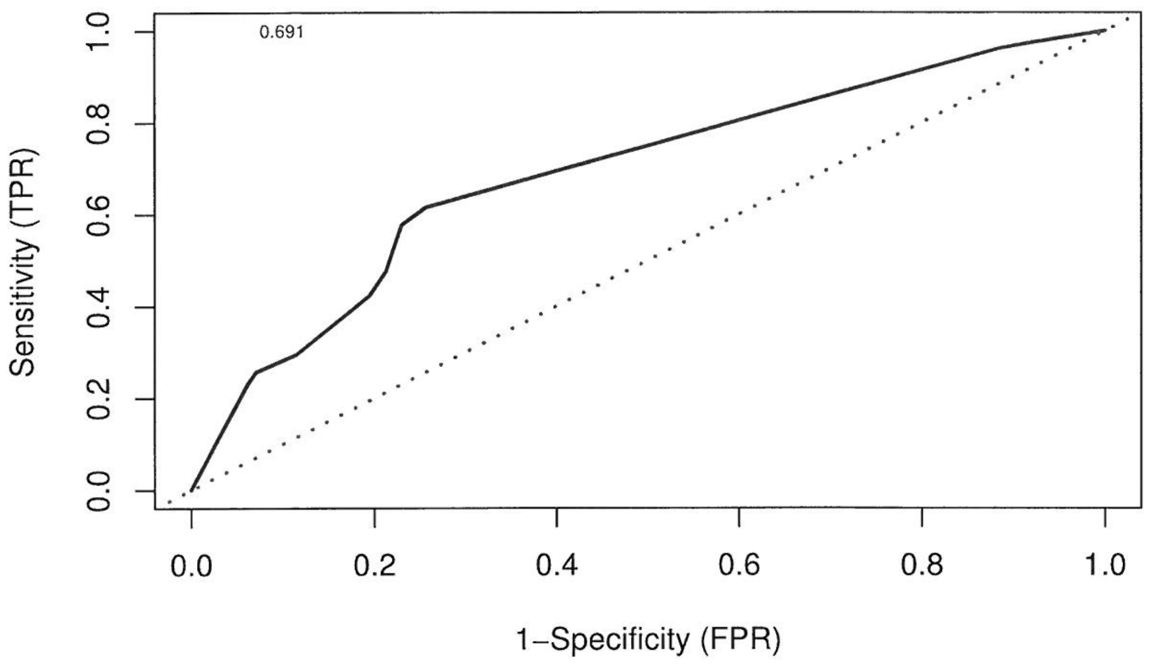

Figure 4: ROC curve with AUC value

We will next look at an ROC curve for the model in Figure 4. The blue curve gives an indication of how much better than random guessing our model is, with the red dotted line representing the random guess. The value in red is the AUC, or the area under the curve, for the ROC curve. A better than random model would have an AUC of > 0.5 (the area under the red dotted line).

```
library(rpart)
library(ROCR)


# function to build a partition tree

fit.build = function(formula, data) {

  rpart(formula = formula, data = data, method = 'class')
}


# function to build a partition tree with Xtreme parameter values

fit.buildX = function(formula, data) {

  rpart(formula = formula, data = data, method = 'class', cp = -0, minsplit = 1, minbucket = 1, maxdepth =
30)
}


# function to build a partition tree with Xtreme parameter values

fit.buildP = function(formula, data, cp = 0.01, minsplit = 20, minbucket = round(minsplit/3), maxdepth = 30)
{

  rpart(formula = formula, data = data, method = 'class', cp = cp, minsplit = minsplit, minbucket =
minbucket, maxdepth = maxdepth)
}


# function to prune the partition tree

fit.prune = function(fit, cp) {

  prune(fit, cp = cp)
}


# function to plot a partition tree with labels

fit.plot = function(fit) {

  plot(fit, margin = 0.01, uniform = T)
  text(fit, use.n = T, xpd = T, minlength = 6)
}


# function to plot and print complexity parameter information

fit.plotcp = function(fit) {

  plotcp(fit)
  printcp(fit)
}


# function that returns the complexity parameter that corresponds to the min cross validation error

fit.mincvcp = function(fit) {

  fit$cptable[which.min(fit$cptable[, "xerror"]), "CP"]
}
```

```
# function to make predition, and to make human readable

fit.predict = function(fit, newdata) {

  predict(fit, newdata = newdata)
}


# function to print the performance of the prediction

fit.performance = function(fit, pred, val, pred.labels) {

  outcome = rep(pred.labels[1], nrow(pred))
  outcome[pred[, 1] > 0.5] = pred.labels[2]

  prop.table(xtabs(~ val$result + outcome), 1)
}


# function to plot the analysis of the fit

fit.analysis = function(fit, pred, val, pred.labels) {

  p.scores = prediction(pred[,1], val$result, label.ordering = pred.labels)
  p.perf   = performance(p.scores, "tpr", "fpr")
  p.auc    = performance(p.scores, "auc")

  plot(p.perf, col = "blue", lwd = 2, xlab = '1-Specificity (FPR)', ylab = 'Sensitivity (TPR)')
  abline(a = 0, b = 1, col = 'red', lty = 3, lwd = 2)
  text(0.1, 1, round(p.auc@y.values[[1]], 3), col = 'red')
}
```

```
set.seed(27041970)
```

✓

```
source("~/rpart_functions.R")


retention           = read.csv("~/students_retention_full.csv", header = T)
outcome             = c('failed', 'passed')
retention$result    = factor(retention$overall6, levels = 0:1, labels = outcome)
retention$overall6 = NULL
retention$id        = NULL

attach(retention)
table(retention$result)

include    = rbinom(dim(retention)[1], 1, 2/3)
training   = subset(retention, include == 1)
validation = subset(retention, include == 0)
```

address 2 as factor ?

```
fit0 = fit.buildP(result ~ ., data = training)
fit.plot(fit0)
fit.plotcp(fit0)

prediction = fit.predict(fit0, validation)
fit.performance(fit0, prediction, validation, rev(outcome))
fit.analysis(fit0, prediction, validation, rev(outcome))


fit1 = fit.buildX(result ~ ., data = training)
fit.plot(fit1)
fit.plotcp(fit1)

prediction = fit.predict(fit1, validation)
fit.performance(fit1, prediction, validation, rev(outcome))
fit.analysis(fit1, prediction, validation, rev(outcome))


fit2 = fit.prune(fit1, cp = 0.0095785)
fit.plot(fit2)
fit.plotcp(fit2)

prediction = fit.predict(fit2, validation)
fit.performance(fit2, prediction, validation, rev(outcome))
fit.analysis(fit2, prediction, validation, rev(outcome))


fit3 = fit.prune(fit1, cp = 0.0114943)
fit.plot(fit3)
fit.plotcp(fit3)

prediction = fit.predict(fit3, validation)
fit.performance(fit3, prediction, validation, rev(outcome))
fit.analysis(fit3, prediction, validation, rev(outcome))


fit4 = fit.prune(fit1, cp = 0.0517241)
fit.plot(fit4)
fit.plotcp(fit4)

prediction = fit.predict(fit4, validation)
fit.performance(fit4, prediction, validation, rev(outcome))
fit.analysis(fit4, prediction, validation, rev(outcome))


fit5 = fit.prune(fit1, cp = fit.mincvcp(fit1))
fit.plot(fit5)
fit.plotcp(fit5)

prediction = fit.predict(fit5, validation)
fit.performance(fit5, prediction, validation, rev(outcome))
fit.analysis(fit5, prediction, validation, rev(outcome))
```

```
# for the specific example of lcpoints we will find the splitting point using gini scoring.

score   = function(p1, p2) { 1 - p1^2 - p2^2 }
# score = function(p1, p2) { -((p1 * log(p1)) + (p2 * log(p2))) }

calculate = function(split1, split2) {

  length1 = nrow(split1)
  length2 = nrow(split2)
  prob1   = prop.table(table(split1$result))
  prob2   = prop.table(table(split2$result))

  (length1 / length) * score(prob1[1], prob1[2]) + (length2 / length) * score(prob2[1], prob2[2])
}

length  = nrow(training)
points  = sort(unique(training$lcpoints))
minimum = c(1.0, 0)

for (point in points) {

  value = calculate(training[ training$lcpoints  < point, ], training[ training$lcpoints >= point, ])
  if (is.finite(value) && value < minimum[1])
    minimum = c(value, point)
}

minimum




detach(retention)
```