Session 12

Tree-based models: tree and rpart

Two libraries

- The **`tree`** library is like the S-PLUS native library and implements the traditional S-PLUS tree technology
- The **`rpart`** library is due to Beth Atkinson and Terry Therneau of the Mayo Clinic, Rochester, NY. It implements a technology much closer to the traditional CART version of trees due to Friedman, Breiman, Olshen and Stone.
- Both have their advantages and disadvantages. We mostly favour the **`rpart`** version here, but most examples can be done on the **`tree`** library as well.

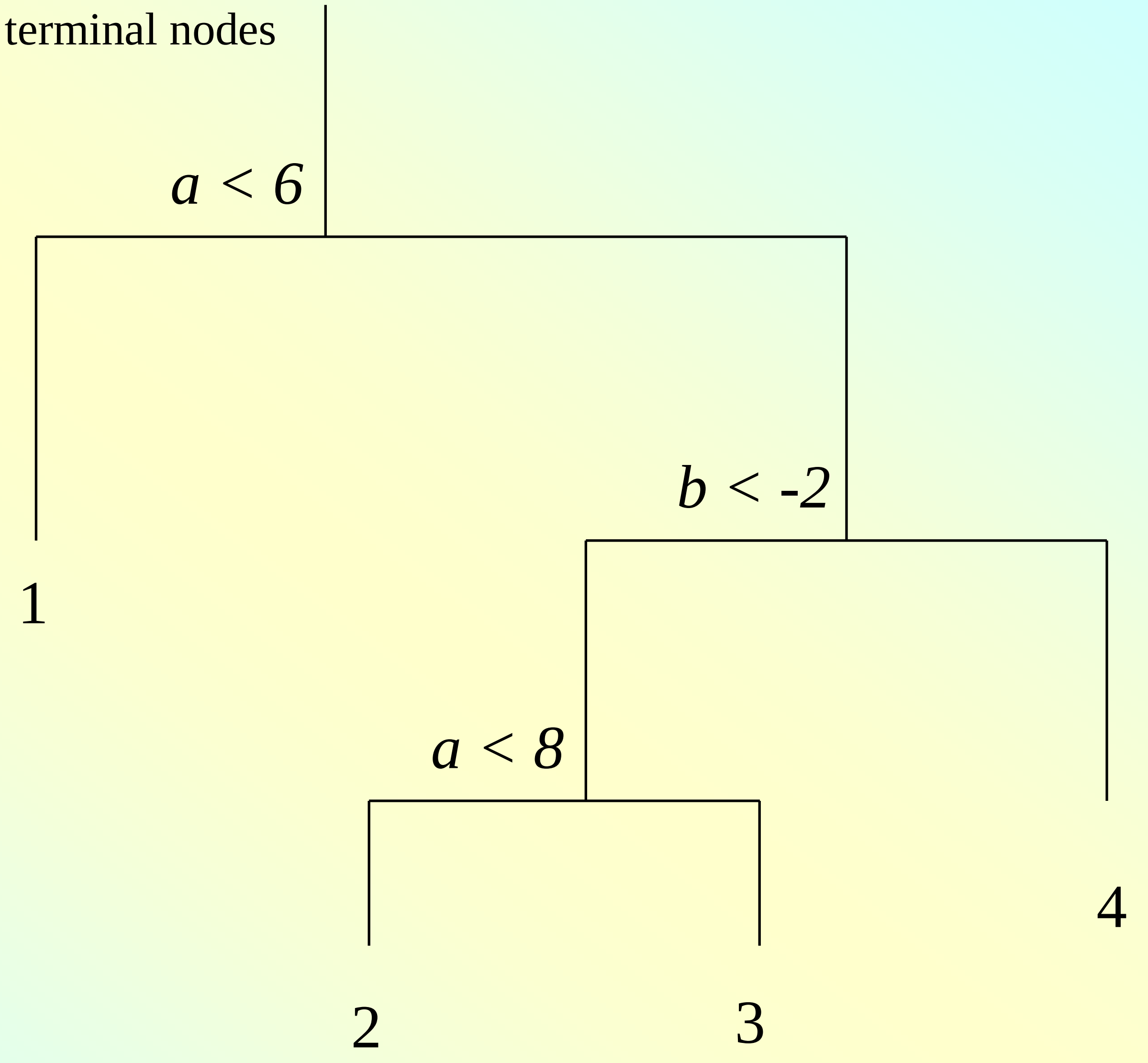
Overview

- Goal is to construct a predictor, perhaps at the cost of a safe interpretation of how it works
- Trees are easy to interpret, but relying on that interpretation can be hazardous
- Recursive partitioning: Note that this is a greedy algorithm.
- Two kinds of tree:
  - Regression trees: continuous response with deviance measured as least squares – exactly the same as for regression
  - Classification trees: factor response with deviance measured by entropy (or Shannon-Wiener Information).

Recursive partitioning

- We assume a homogeneity measure – least squares or entropy
- For a given variable, find the point at which the responses are divided into the two most homogeneous groups
- Choose the variable which does this best and divide the sample into two groups at the best point
- Apply the same procedure recursively to each side
- Stop when either the node is completely homogeneous or contains too few observations to continue

A decision tree with four terminal nodes



An Example: the CPUs data again

- Classical example from the prediction literature – a set of CPUs whose log-performance is to be predicted using some qualitative measurements

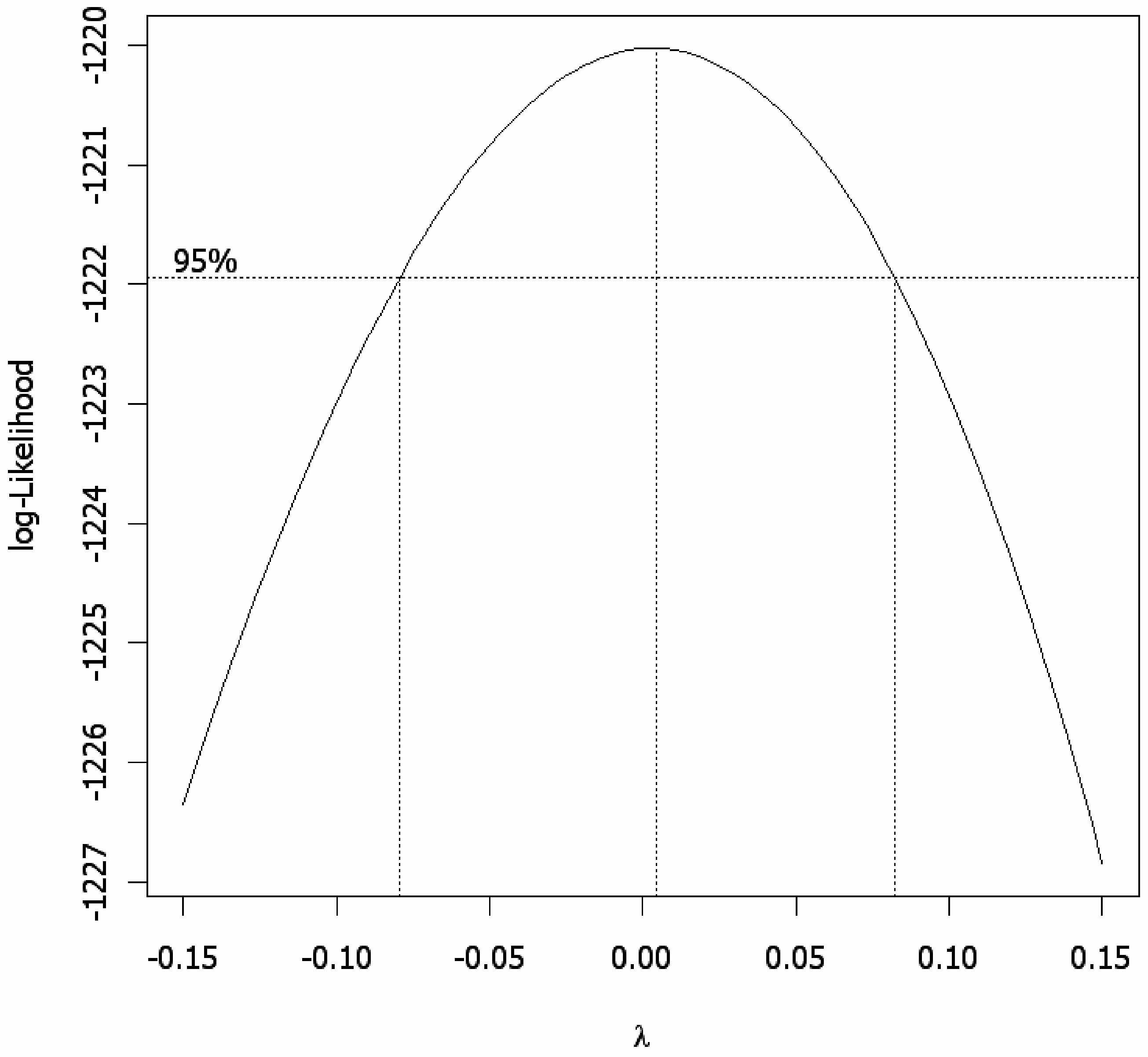
```
   names(cpus)
[1] "name"    "syct"    "mmin"    "mmax"    "cach"    "chmin"
[7] "chmax"   "perf"    "estperf"
   dim(cpus)
[1] 209   9
```

- We begin using a pruned tree
- We compare the results using a bagging approach

Transformed response scale?

- A 'log' transform seems natural
- One way of showing that it is acceptable:

```
CPUs <- cpus[, 2:8]
for(j in 1:6)
  CPUs[[j]] <- cut(rank(CPUs[[j]],ties = "r"), 5)

fm <- lm(perf ~ ., CPUs)
boxcox(fm, lambda = seq(-0.15, 0.15, len = 10))
```


log-Likelihood
-1220
-1221
-1222
-1223
-1224
-1225
-1226
-1227
95%
-0.15
-0.10
-0.05
0.00
0.05
0.10
0.15
λ

First split the data into training and test sets and set up a test function:

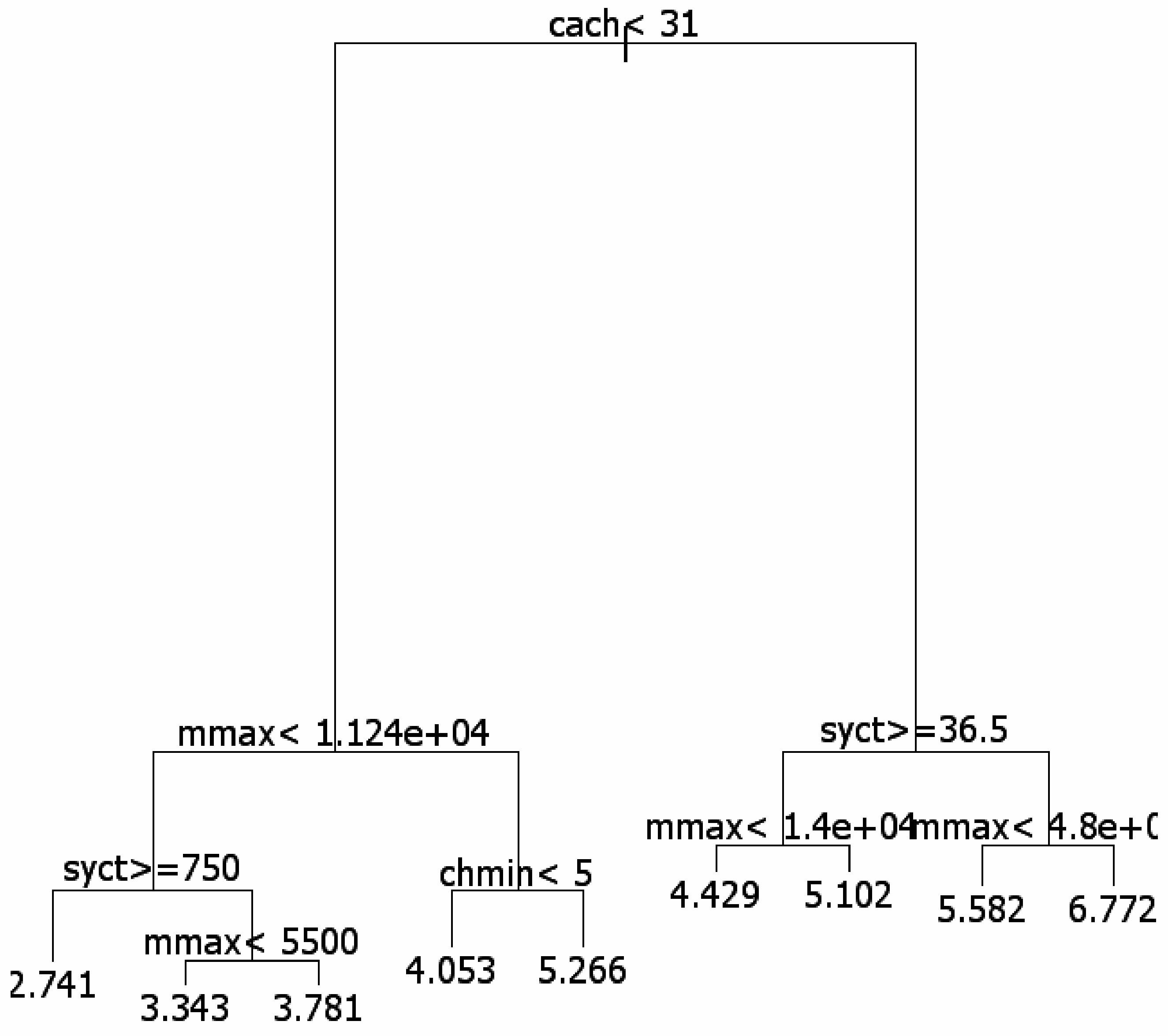
```
set.seed(38267251) # My phone number
cpus.samp <- sample(nrow(cpus), 100)

cpusTrain <- cpus[cpus.samp, 2:8] # omit name and
   manufactuer's estimate
cpusTest <- cpus[-cpus.samp, 2:8]

testPred <- function(fit, data = cpusTest) {
#
# mean squared error for the performance of a
# predictor on the test data.
#
   testVals <- log(data[, "perf"])
   predVals <- predict(fit, data[, ])
   sqrt(sum((testVals - predVals)^2)/nrow(data))
}

library(rpart)
cpus.t1 <- rpart(log(perf) ~ syct + mmin + mmax + cach + chmin
   + chmax, cpusTrain, minsplit = 3)
```

Now fit the first model with a very small minimum splitting size

```
library(rpart, first = T)
cpus.t1 <- rpart(log(perf) ~ syct + mmin + mmax +
  cach + chmin + chmax, dat1, minsplit = 3)
testPred(cpus.t1)  # not good!
[1] 0.5723122
```

See how the tree looks:

```
plot(cpus.t1)
text(cpus.t1)
```


cach< 31
mmax< 1.124e+04
syct>=36.5
syct>=750
chmin< 5
mmax< 1.4e+04
mmax< 4.8e+0
mmax< 5500
2.741
3.343
3.781
4.053
5.266
4.429
5.102
5.582
6.772

```
> cpus.t1
n= 100
node), split, n, deviance, yval
      * denotes terminal node

 1) root 100 104.7362000 4.150773
   2) cach< 31 68  29.9160800 3.628058
     4) mmax< 11240 51  11.9181500 3.391328
       8) syct>=750 9   0.5870328 2.740580 *
       9) syct< 750 42   6.7031610 3.530774
        18) mmax< 5500 24   2.3057870 3.342837 *
        19) mmax>=5500 18   2.4194180 3.781358 *
     5) mmax>=11240 17   6.5655770 4.338247
      10) chmin< 5 13   1.7793700 4.052730 *
      11) chmin>=5 4   0.2822183 5.266178 *
   3) cach>=31 32  16.7585700 5.261541
     6) syct>=36.5 19   3.9935560 4.854145
      12) mmax< 14000 7   0.6427171 4.428796 *
      13) mmax>=14000 12   1.3456220 5.102265 *
     7) syct< 36.5 13   5.0026270 5.856967
      14) mmax< 48000 10   1.5606240 5.582417 *
      15) mmax>=48000 3   0.1756196 6.772136 *
```

Pruning trees

- It is important to prune trees so that
  - They are small enough to avoid putting random variation into predictions
  - They are large enough to avoid putting systematic biases into predictions
- Cross-validation is the normal tool for this purpose
- **`rpart`** has a quick version, but tools for a more thorough version if needed
- **`tree`** has tools for the more thorough version, (but the onus is still on the user to do it thoroughly)

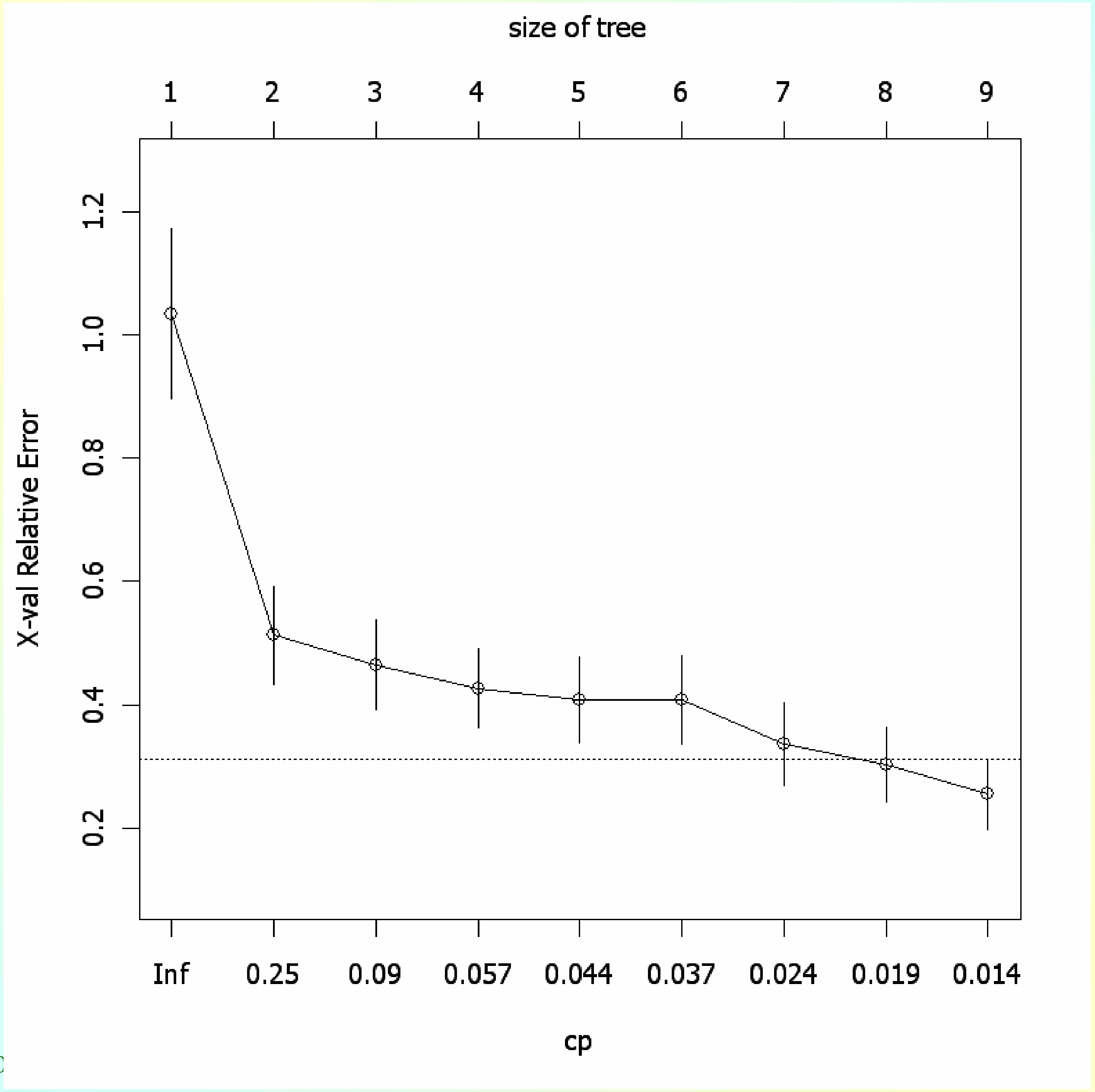
Cross-validation in trees

- Consider a cost-complexity measure:

$$D_\alpha(T) = \text{Deviance(T)} + \alpha\ \text{Size(T)}$$

- The complexity parameter, α, regulates the trade-off between accuracy in the training sample and simplicity in the result
- By building trees on rotating sections of the data and predicting for the omitted sections we get some idea on the kind of value that might be appropriate for α.
- ‘One SE’ rule suggests a choice of α

plotcp(cpus.t1)

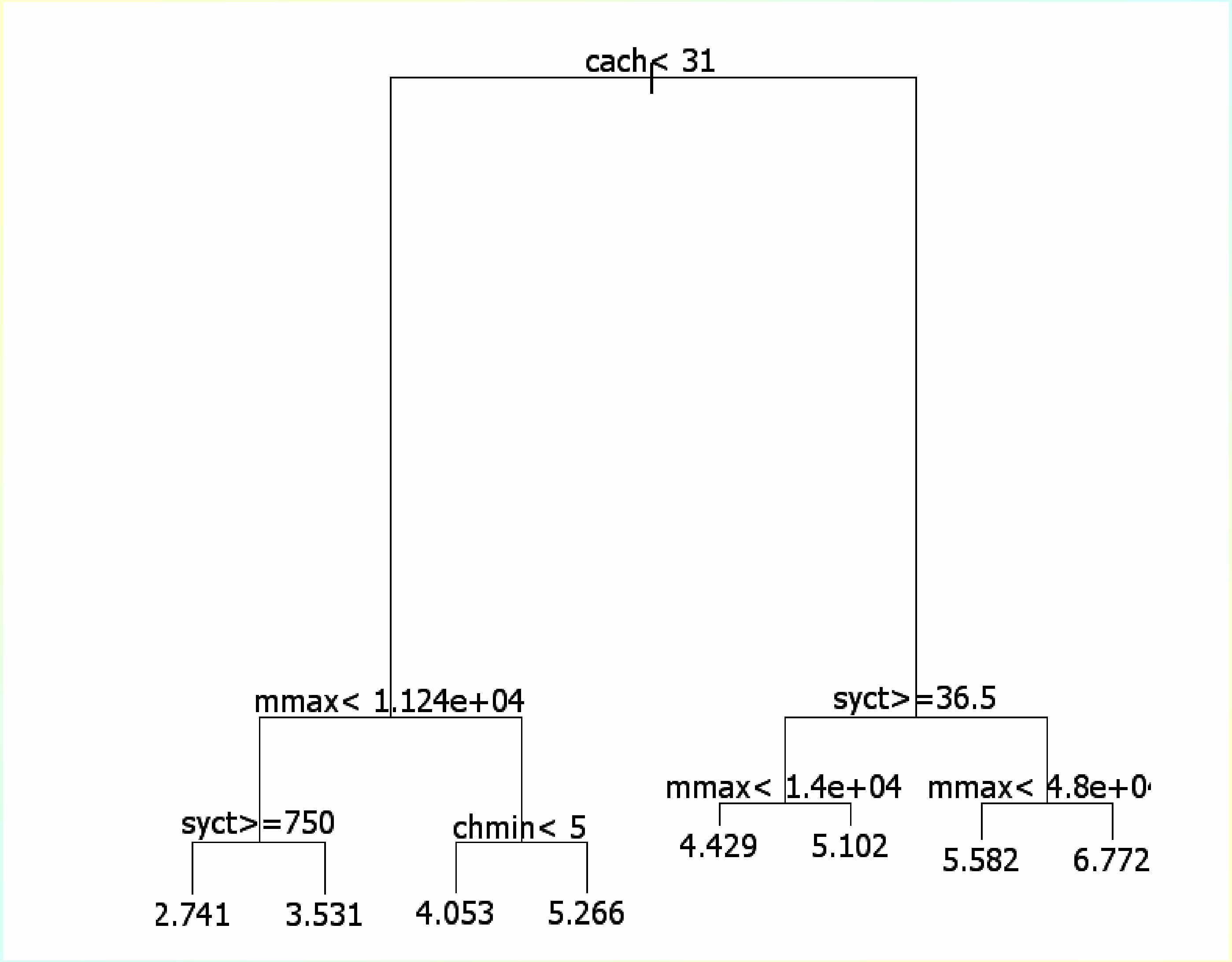


- Rather than 8 nodes this suggests that about 6 nodes are warranted.

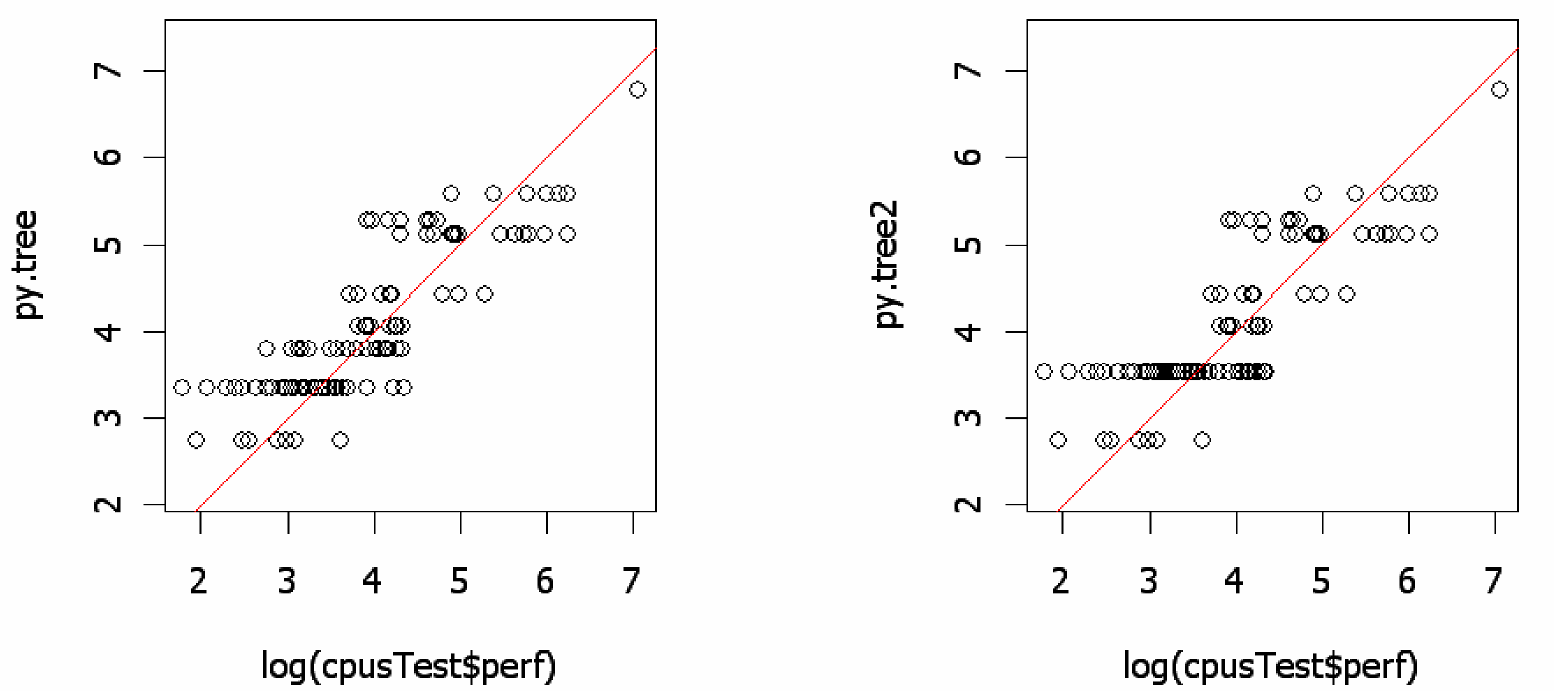
```
cpus.t2 <- prune(cpus.t1, cp=0.019)
testPred(cpus.t2)  ## slightly worse!
[1] 0.6086504
py.tree <- predict(cpus.t1, cpusTest)
py.tree2 <- predict(cpus.t2, cpusTest)
cor(cbind(log(cpusTest$perf), py.tree, py.tree2))
                      py.tree  py.tree2
         1.0000000 0.8454302 0.8247576
py.tree  0.8454302 1.0000000 0.9854434
py.tree2 0.8247576 0.9854434 1.0000000
```

- Pruning seems not to have paid off!

```
plot(cpus.t2)
text(cpus.t2)
```



```
par(mfrow = c(1,2), pty = "s")
plot(log(cpusTest$perf), py.tree, asp = 1)
abline(0, 1, col = "red")
plot(log(cpusTest$perf), py.tree2, asp = 1)
abline(0, 1, col = "red")
```



Bootstrap Aggregation (or ‘Bagging’)

- Technique for considering how different the result might have been if the algorithm were a little less greedy
- Bootstrap training samples of the data are used to construct a ‘forest’ of trees
- Predictions from each tree are averaged (regression trees) or ‘majority vote’ (for classification trees)
- How many trees in the forest is still a matter of some debate, but ‘lots’
- ‘Random Forests’ develops this idea much further.

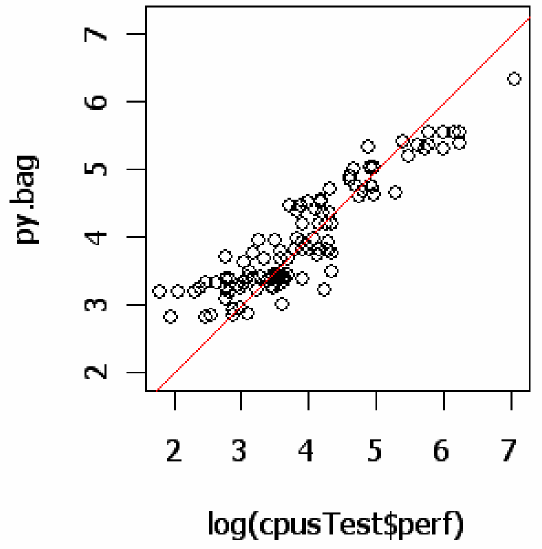
Some bagging functions

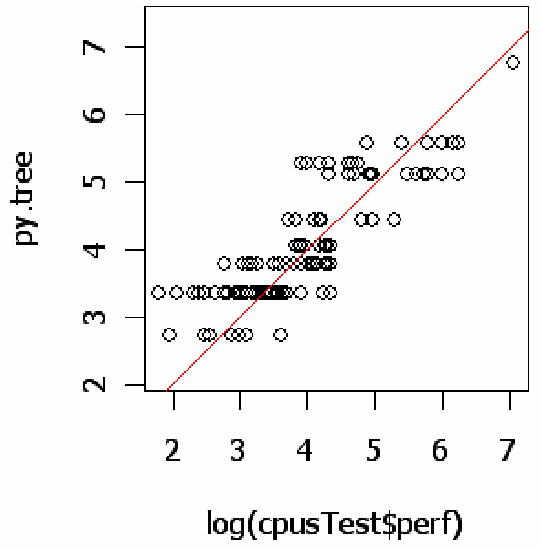
```
bsample <- function(dataFrame) # bootstrap sampling
dataFrame[sample(nrow(dataFrame), rep = T),  ]

simpleBagging <- function(object,
   data = eval(object$call$data), nBags = 200, ...) {
   bagsFull <- list()
   for(j in 1:nBags)
       bagsFull[[j]] <- update(object, data =
   bsample(data))
   oldClass(bagsFull) <- "bagRpart"
   bagsFull
}

predict.bagRpart <- function(object, newdata, ...)
   rowMeans(sapply(object, predict, newdata = newdata))
```

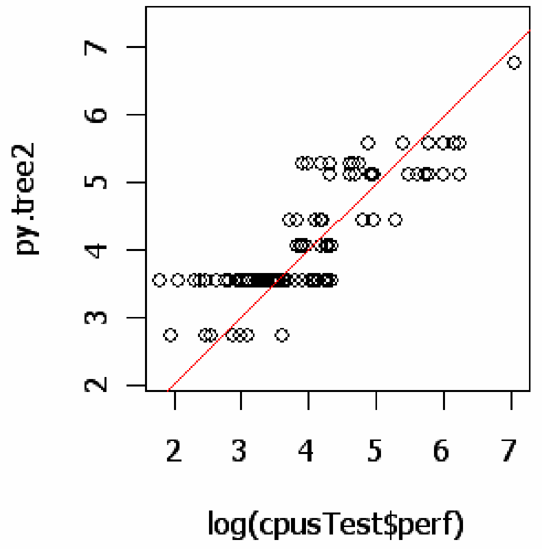
Execute and compare results

```
cpus.bag <- simpleBagging(cpus.t1)
testPred(cpus.bag)  # bit better!
[1] 0.4678958

py.bag <- predict(cpus.bag, cpusTest)
cor(cbind(log(cpusTest$perf), py.bag, py.tree,
   py.tree2))
                         py.bag   py.tree  py.tree2
         1.0000000 0.9093912 0.8454302 0.8247576
py.bag   0.9093912 1.0000000 0.9609053 0.9402384
py.tree  0.8454302 0.9609053 1.0000000 0.9854434
py.tree2 0.8247576 0.9402384 0.9854434 1.0000000
```

```
par(mfrow = c(2,2), pty = "s"); frame()

plot(log(cpusTest$perf), py.bag, asp = 1)
abline(0, 1, col = "red")
plot(log(cpusTest$perf), py.tree, asp = 1)
abline(0, 1, col = "red")
plot(log(cpusTest$perf), py.tree2, asp = 1)
abline(0, 1, col = "red")
```







The big guns

- The Random Forest technique, due to Leo Breiman and his colleagues, is a further development of bagging.
- It includes subsampling of the possible predictors at every possible split.
- Generally accepted as one of the best of the simple methods for improving the stability of trees.
- Available as the **`randomForest`** package for *R*

```
require(randomForest)
cpus.rf <- randomForest(log(perf) ~ ., cpusTrain)
testPred(cpus.rf)
[1] 0.4104117
```

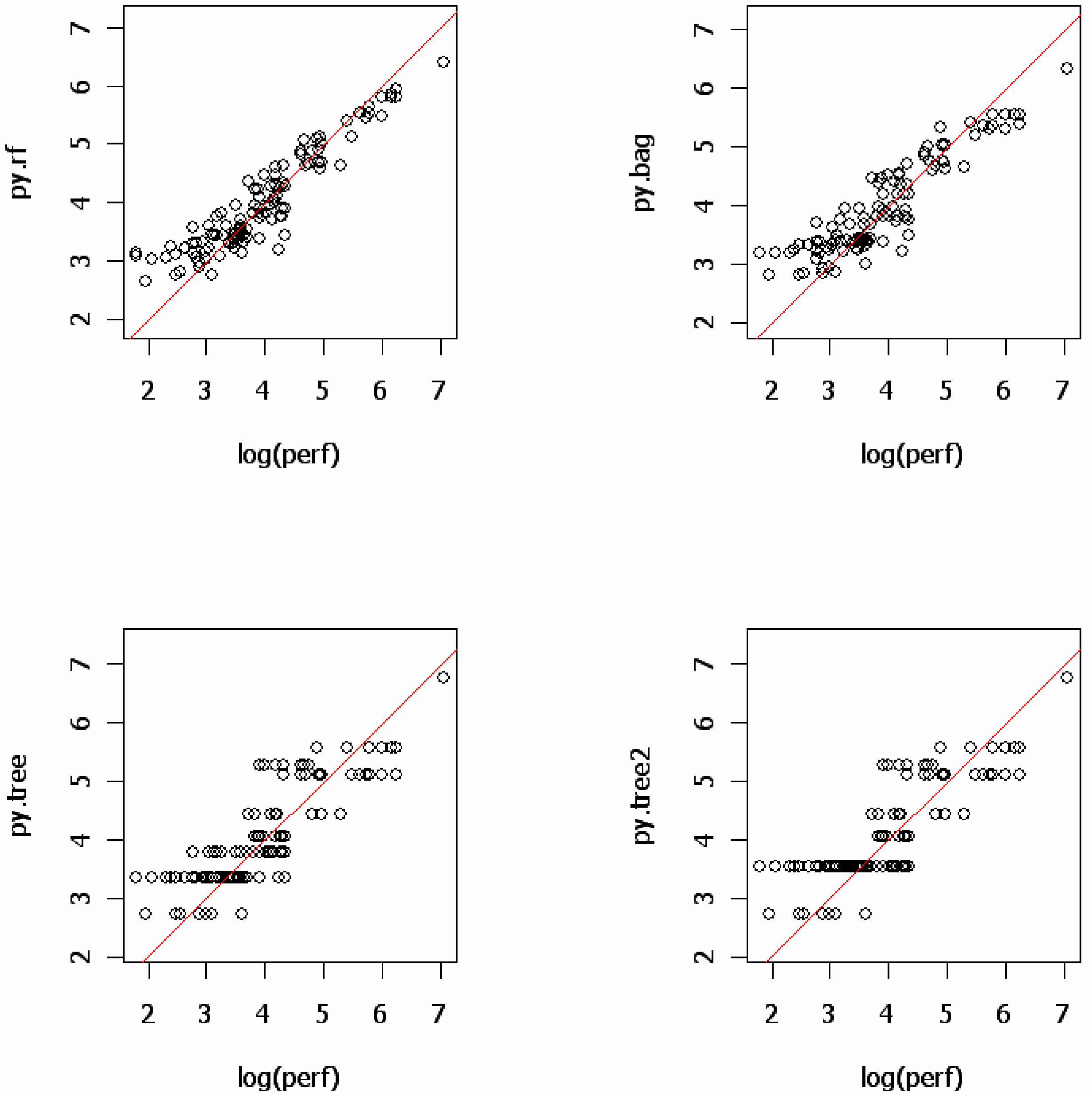
Putting it all together

```
py.rf <- predict(cpus.rf, cpusTest)
round(cor(cbind(log(cpusTest$perf),
      py.tree, py.tree2, py.bag, py.rf)),4)
```

```
                py.tree py.tree2 py.bag  py.rf
         1.0000  0.8454   0.8248 0.9094 0.9305
py.tree  0.8454  1.0000   0.9854 0.9609 0.9429
py.tree2 0.8248  0.9854   1.0000 0.9402 0.9250
py.bag   0.9094  0.9609   0.9402 1.0000 0.9905
py.rf    0.9305  0.9429   0.9250 0.9905 1.0000
```

Values against predictions

```
par(mfrow = c(2,2), pty = "s")
with(cpusTest, {
  plot(log(perf), py.rf, asp = 1)
  abline(0, 1, col = "red")
  plot(log(perf), py.bag, asp = 1)
  abline(0, 1, col = "red")
  plot(log(perf), py.tree, asp = 1)
  abline(0, 1, col = "red")
  plot(log(perf), py.tree2, asp = 1)
  abline(0, 1, col = "red")
})
```


py.rf
py.bag
py.tree
py.tree2
log(perf)
2 3 4 5 6 7

Synoptic forecasts

- The default prediction method for classification trees is to give a matrix of probabilities of class memberships
- This allows the membership situation to be more clearly appreciated
- The “class” rule simply chooses the class with maximum posterior probability
- *Bagging in classification trees*:
  - The usual recommendation is to use a ‘majority vote’ rule

Epilogue

- Tree models have brought statistical modellers and the machine learning fraternity closer together
- As predictors they offer some useful features, but suffer from instability.
- Bagging is an attempt to overcome this instability, but has only limited success.
- Breiman & Cutler’s ‘Random Forests’ offers a refinement of bagging that looks very promising.
- ‘Boosting’ is an alternative to bagging, but much more difficult to implement.
- Trees in data analysis: often revealing, but there is often a danger to read too much into the split variables.