User written splitting functions for RPART

Terry Therneau
Mayo Clinic

March 28, 2014

1 Splitting functions

The rpart code was written in a modular fashion with the idea that the C code would be extended to include more splitting functions. As a consequence all of the splitting functions are coordinated through a single structure in the file `func_table.h`, shown below:

```
static struct {
    int  (*init_split)();
    void (*choose_split)();
    void (*eval)();
    double (*error)();
} func_table [] = {
    {anovainit,   anova,   anovass,    anovapred},
    {poissoninit, poisson, poissondev, poissonpred},
    {giniinit, gini, ginidev, ginipred},
    {usersplit_init, usersplit, usersplit_eval, usersplit_pred}
};

#define NUM_METHODS 4  /*size of the above structure */
```

Adding a new splitting method requires the definition of four functions which initialize, choose a split, compute the predicted value or values for a node, and compute a prediction error for a new observation assigned to the node. The last of these is used only in cross-validation.

To add new splitting rules to the main routine four new C functions need to be defined, the `func_table.h` file expanded, the R function `rpart` updated to recognize the new option (it calls the C routine with the row number of func_table), and finally an initialization function written. See the `rpart.class`, `rpart.anova`, and `rpart.exp` functions for examples of the latter. The lion's share of the code, which deals with all the necessary bookkeeping, remains unchanged.

An easier route is to add a user defined splitting rules as a set of R functions. The resulting code will be considerably slower than the built-in functions; nevertheless this allows an easy way to extend `rpart`, and to test out new ideas with less effort than additions to the C base. For user defined splits only the first three functions are defined, and cross-validation does not happen automatically. (User defined methods are many times slower than the built in ones, so it may be preferable to skip cross-validation during the initial evaluation of a model.)

2 Anova function

As the first illustration we show how to add anova splitting as a user-written addition. This method is also one of those built into the C code, so we can also check the correctness of the results.

2.1 Initialization function

```
> itemp <- function(y, offset, parms, wt) {
      if (is.matrix(y) && ncol(y) > 1)
         stop("Matrix response not allowed")
      if (!missing(parms) && length(parms) > 0)
          warning("parameter argument ignored")
      if (length(offset)) y <- y - offset
      sfun <- function(yval, dev, wt, ylevel, digits ) {
                  paste("  mean=", format(signif(yval, digits)),
                        ", MSE=" , format(signif(dev/wt, digits)),
                        sep = '')
      }
      environment(sfun) <- .GlobalEnv
      list(y = c(y), parms = NULL, numresp = 1, numy = 1, summary = sfun)
  }
```

On input the function will be called with

y the response value as found in the formula. Note that rpart will normally have removed any observations with a missing response.

offset the offset term, if any, found on the right hand side of the formula

parms the vector or list (if any) supplied by the user as a `parms` argument to the call.

wt the weight vector from the call, if any

The last two arguments are optional. The initialization function needs to perform any data checks, deal with the offset vector if present, and return a list containing

y the value of the response, possibly updated

numy the number of columns of y

numresp the length of the prediction vector for each node

summary optional: a function which will produce a 1–3 line summary for the node, to be used by `summary.rpart`.

print optional: a function which will produce a one line summary for the `print` function.

text optional: a function which will produce a short label for the node, used by the `plot` function.

The parameters vector can contain whatever might be appropriate for the method, e.g. the variance of a prior distribution. The vector of parameters passed forward need not be he same as the set passed into the routine. In anova splitting there are no extra parameters.

The summary function will be called with arguments which are

- yval= the response value for the node
- dev = the deviance for the node
- wt = the sum of weights at the node (number of observations)
- ylevel = the levels of y, if y is categorical
- digits = the current setting for digits

It should return a character vector. The text function will be called with these arguments plus two more

- n= the number of observations in the node
- use.n = TRUE/FALSE, see the help page for `text.rpart`

The print function is called only with yval, ylevels, and digits.

The only puzzling line may be `environment(sfun) <- .GlobalEnv`. R ensures that any function can access variables that are *not* passed in as arguments via a mechanism called environments. As a trivial example

```
> temp <- 4
> fun1 <- function(x) {
      q <- 15
      z <- 10
      fun2 <- function(y) y + z + temp
      fun2(x^2)
  }
> fun1(5)
[1] 39
```

The definition of fun2 essentially contains a copy of `z` (and `q` as well) to ensure that this works. The exception to this is objects at the top level such as `temp` above; the user is responsible for the retention of those via their answer to the "save" question at the end of an R session.

The consequence of this is that the summary function created by a call to itemp will by default have copies of all of the external variables that it *might* make use of, in this case all of the input arguments. The summary function is in fact self-contained and makes reference only to its input arguments (doing otherwise is bad programming, in my opinion) and so needs none of these. Setting the environment of the function to `.GlobalEnv` in essence tells R to treat the function as though it had been defined at top level, i.e., the input prompt, and thus not attach these extraneous copies. I first became aware of this issue when using a huge data set, and found that the saved output of the fit took up an inordinate amount of disk space due to data copies attached to the summary function.

Do not take this as a critique of R environments in general. The rules governing them need to be responsive to a large ensemble of conditions; in this particular case the results are not ideal but in the main they are the best compromise possible.

2.2 Evaluation function

The evaluation function is called once per node. It needs to produce a deviance for the node along with a vector to be used as a label for the node.

```
> etemp <- function(y, wt, parms) {
      wmean <- sum(y*wt)/sum(wt)
      rss <- sum(wt*(y-wmean)^2)
      list(label = wmean, deviance = rss)
  }
```

As an example of a longer label, the gini splitting method for categorical outcomes returns both the chosen group and the full vector of estimated group probabilities. The deviance value that is returned does not need to be an actual deviance associated with a log-likelihood, any impurity measure for the node will work. However, the pruning algorithm used during tree construction will be most efficient if the value 0 corresponds to a perfectly pure node. (The pruning code decides when a branch would be futile and further computations on it can thus be avoided.)

2.3 Splitting function

The splitting function is where the work lies. It will be called once for every covariate at each potential split. The input arguments are

y vector or matrix of response values

wt vector of weights

x vector of x values

parms vector of user parameters, passed forward

continuous if TRUE the x variable should be treated as continuous

The data will have been subset to include only the non-missing subset of x and y at the node, and if x is continuous all values will be in the sorted order of the x variable. If x is continuous the routine should return two vectors each of length $n - 1$, where n is the length of x:

goodness the utility of the split, where larger numbers are better. A value of 0 signifies that no worthwhile split could be found (for instance if y were a constant). The ith value of goodness compares a split of observations 1 to i versus $i + 1$ to n.

direction A vector of the same length with values of -1 and +1, where -1 suggests that values with y < cutpoint be sent to the left side of the tree, and a value of +1 that values with y < cutpoint be sent to the right. This is not critical, but sending larger values of y to the right, as is done in the code below, seems to make the final tree easier to read.

The reason for returning an entire vector of goodness values is that the parent code is responsible for handling the minimum node size constraint, and also for dealing with ties. When x is continuous the split routine actually has no reason to look at x.

```
> stemp <- function(y, wt, x, parms, continuous)
  {
      # Center y
      n <- length(y)
      y <- y- sum(y*wt)/sum(wt)

      if (continuous) {
          # continuous x variable
          temp <- cumsum(y*wt)[-n]
          left.wt  <- cumsum(wt)[-n]
          right.wt <- sum(wt) - left.wt
          lmean <- temp/left.wt
          rmean <- -temp/right.wt
          goodness <- (left.wt*lmean^2 + right.wt*rmean^2)/sum(wt*y^2)
          list(goodness = goodness, direction = sign(lmean))
      } else {
          # Categorical X variable
          ux <- sort(unique(x))
          wtsum <- tapply(wt, x, sum)
          ysum  <- tapply(y*wt, x, sum)
          means <- ysum/wtsum

          # For anova splits, we can order the categories by their means
          #  then use the same code as for a non-categorical
          ord <- order(means)
          n <- length(ord)
          temp <- cumsum(ysum[ord])[-n]
          left.wt  <- cumsum(wtsum[ord])[-n]
          right.wt <- sum(wt) - left.wt
          lmean <- temp/left.wt
          rmean <- -temp/right.wt
          list(goodness= (left.wt*lmean^2 + right.wt*rmean^2)/sum(wt*y^2),
               direction = ux[ord])
      }
  }
```

The code above does the computations for all the split points at once by making use of two tricks. The first is to center the y values at zero (so the grand mean is zero), and the second takes advantage of the many ways to write the "effect" sum of squares for a simple two group anova. The key identity is

$$\sum (y_i - c)^2 = \sum (y_i - \overline{y})^2 + n(c - \overline{y})^2 \tag{1}$$

If you have an old enough statistics book, this is used to show that for a 1-way anova the between groups sum of squares SS_B is

$$SS_B = n_l(\overline{y}_l - \overline{y})^2 + n_r(\overline{y}_r - \overline{y})^2 \tag{2}$$

where n_l is the number of observations in the left group, $\overline{y}_l$ the mean of y in the left group, $\overline{y}$ the overall mean, and n_r $\overline{y}_r$ the corresponding terms for the right hand group.

Centering at zero makes $\overline{y}$ zero in (2), and the terms then can be computed for all splits at once using the cumsum function. Extension of the formulas to case weights is left as an exercise for the reader.

If the predictor variable x is categorical with k classes then there are potentially $2^{k-1} - 1$ different ways to split the node. However, for most splitting rules the optimal rule can be found by first ordering the groups by their average y value and then using the usual splitting rule on this ordered variable. For user mode rules this is assumed to be the case. The variable x is supplied as integer values 1, 2, ..., k. On return the direction vector should have k values giving the ordering of the groups, and the goodness vector $k - 1$ values giving the utility of the splits.

2.4 Test

We can test the above code to make sure that it gives the same answer as the built-in anova splitting method.

```
> library(rpart)
> mystate <- data.frame(state.x77, region=state.region)
> names(mystate) <- casefold(names(mystate)) #remove mixed case
> ulist <- list(eval = etemp, split = stemp, init = itemp)
> fit1 <- rpart(murder ~ population + illiteracy + income + life.exp +
                hs.grad + frost + region, data = mystate,
                method = ulist, minsplit = 10)
> fit2 <- rpart(murder ~ population + illiteracy + income + life.exp +
                hs.grad + frost + region, data = mystate,
                method = 'anova', minsplit = 10, xval = 0)
> all.equal(fit1$frame, fit2$frame)
[1] TRUE
> all.equal(fit1$splits, fit2$splits)
[1] TRUE
> all.equal(fit1$csplit, fit2$csplit)
[1] TRUE
> all.equal(fit1$where, fit2$where)
[1] TRUE
> all.equal(fit1$cptable, fit2$cptable)
[1] TRUE
```

The all.equal test can't be done on fit1 vs fit2 as a whole since their `call` component will differ.

3 Cross validation

To do cross-validation on user written rules one needs to use the `xpred.rpart` routine. This routine returns the predicted value(s) for each observation, predicted from a fit that did not

include that observation. The result is a matrix with one row per subject and one column for each complexity parameter value. As an example we will replicate the cross-validation results for the anova model above. In order to get the same groupings we fix the xval group membership in advance.

```
> xgroup <- rep(1:10, length = nrow(mystate))
> xfit <- xpred.rpart(fit1, xgroup)
> xerror <- colMeans((xfit - mystate$murder)^2)
> fit2b <-  rpart(murder ~ population + illiteracy + income + life.exp +
                  hs.grad + frost + region, data = mystate,
                  method = 'anova', minsplit = 10, xval = xgroup)
> topnode.error <- (fit2b$frame$dev/fit2b$frame$wt)[1]
> xerror.relative <- xerror/topnode.error
> all.equal(xerror.relative, fit2b$cptable[, 4], check.attributes = FALSE)
[1] TRUE
```

3.1 Smoothed anova

For any particular covariate consider a plot of x= split point vs y= goodness of split; figure 1 shows an example for the state data set, along with a smoothed curve. The plot points are very erratic. At one time we entertained the idea that a more stable split point would be found by finding the maximum value for a smoothed version of the plot. Exactly how to smooth is of course a major issue in such an endeavor.

Modification of the anova splitting routine to test this is quite easy — simply add a few lines to the stemp routine:

```
 ...
     goodness= (left.wt*lmean^2 + right.wt*rmean^2)/sum(wt*y^2)
rx <- rank(x[-1])  #use only the ranks of x, to preserve invariance
fit <- smooth.spline(rx, goodness, df=4)
list(goodness= predict(fit, rx)$y, direction=sign(lmean))
        }
   else {
...
```

4 Alternating logistic regression

Chen [1] used a mixed logistic regression model to fit clinical and genetic marker data.

$$E(y) = \frac{e^{\eta_1+\eta_2}}{1+e^{\eta_1+\eta_2}} \tag{3}$$

where $\eta_1 = X\beta$ is a standard linear predictor based on clinical variables X and η_2 is based on a tree model using a set of genetic markers Z. The solution was computed using alternate steps of an ordinary logistic regression on X treating η_2 as an offset, and an rpart fit for Z treating η_1 as an offset.

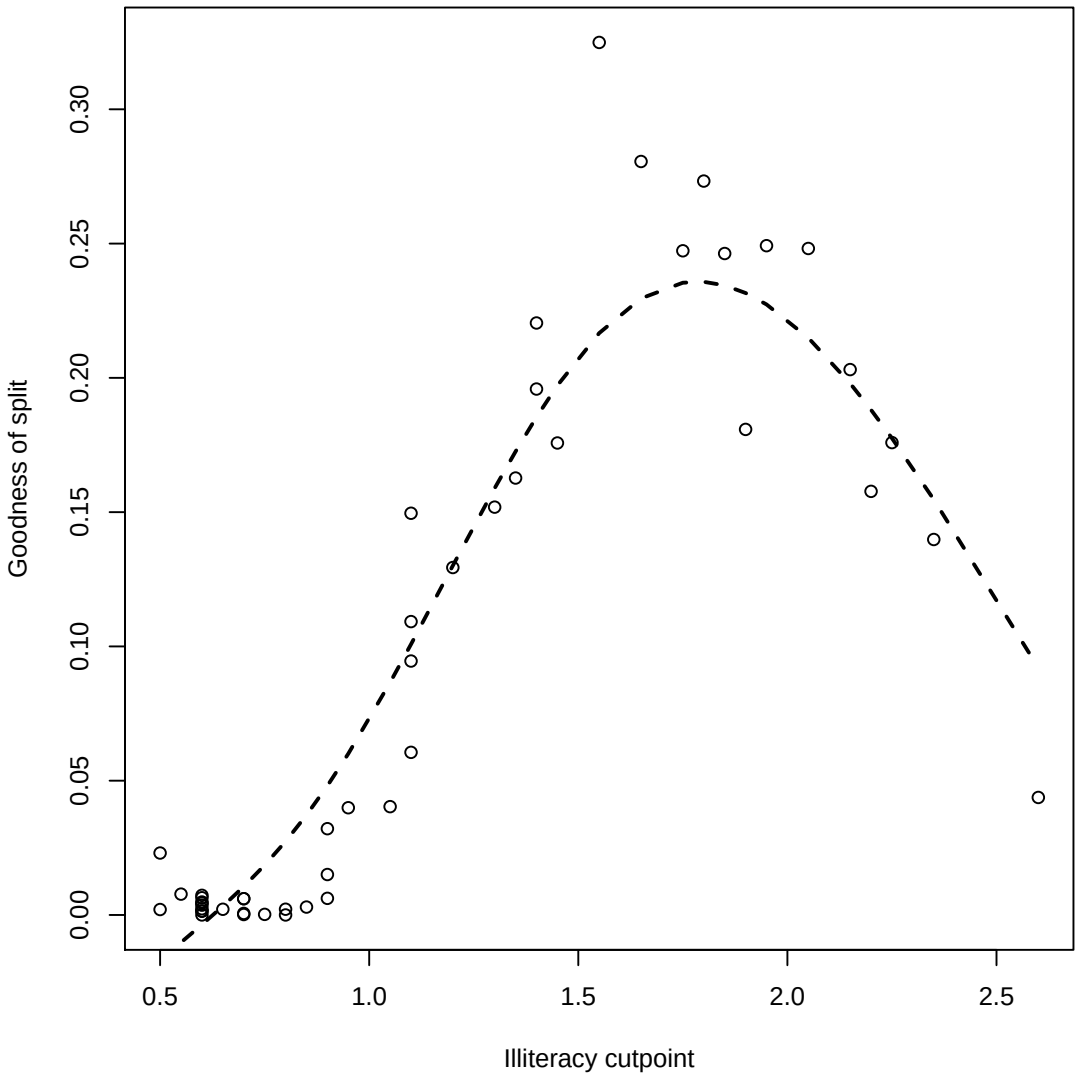

Figure 1: Goodness of split for predicting income from illiteracy, using the state data.

For many splitting rules the initialization function can take care of offset terms once and for all by modifying the y vector, but this is not the case for logistic regression. In order to make the offset visible to the splitting function we create a 2 column "response" consisting of the original y vector in the first column and the offset in the second.

```
> loginit <- function(y, offset, parms, wt)
  {
      if (is.null(offset)) offset <- 0
      if (any(y != 0 & y != 1)) stop ('response must be 0/1')

      sfun <- function(yval, dev, wt, ylevel, digits ) {
                  paste("events=",  round(yval[,1]),
                        ", coef= ", format(signif(yval[,2], digits)),
                        ", deviance=" , format(signif(dev, digits)),
                        sep = '')}
      environment(sfun) <- .GlobalEnv
      list(y = cbind(y, offset), parms = 0, numresp = 2, numy = 2,
           summary = sfun)
      }
> logeval <- function(y, wt, parms)
  {
      tfit <- glm(y[,1] ~ offset(y[,2]), binomial, weight = wt)
      list(label= c(sum(y[,1]), tfit$coef), deviance = tfit$deviance)
  }
```

The evaluation function returns the number of positive responses along with the fitted coefficient.

The splitting function attempts every possible logistic regression. A much faster version could almost certainly be written based on a score test, however.

```
> logsplit <- function(y, wt, x, parms, continuous)
  {
      if (continuous) {
          # continuous x variable: do all the logistic regressions
          n <- nrow(y)
          goodness <- double(n-1)
          direction <- goodness
          temp <- rep(0, n)
          for (i in 1:(n-1)) {
              temp[i] <- 1
              if (x[i] != x[i+1]) {
                  tfit <- glm(y[,1] ~ temp + offset(y[,2]), binomial, weight = wt)
                  goodness[i] <- tfit$null.deviance - tfit$deviance
                  direction[i] <- sign(tfit$coef[2])
              }
          }
      } else {
          # Categorical X variable
```

```
            # First, find out what order to put the categories in, which
            #  will be the order of the coefficients in this model
            tfit <- glm(y[,1] ~ factor(x) + offset(y[,2]) - 1, binomial, weight = wt)
            ngrp <- length(tfit$coef)
            direction <- rank(rank(tfit$coef) + runif(ngrp, 0, 0.1)) #break ties
            # breaking ties -- if 2 groups have exactly the same p-hat, it
            #  does not matter which order I consider them in.  And the calling
            #  routine wants an ordering vector.
            #
            xx <- direction[match(x, sort(unique(x)))] #relabel from small to large
            goodness <- double(length(direction) - 1)
            for (i in 1:length(goodness)) {
                tfit <- glm(y[,1] ~ I(xx > i) + offset(y[,2]), binomial, weight = wt)
                goodness[i] <- tfit$null.deviance - tfit$deviance
            }
        }
        list(goodness=goodness, direction=direction)
    }
```

References

[1] Chen J, Yu K, Hsing A and Therneau TM. *A partially linear tree-based regression model for assessing complex joint gene-gene and gene-environment effects*, Genet Epidemiol, 2007(3), 238-51.