Tree based methods

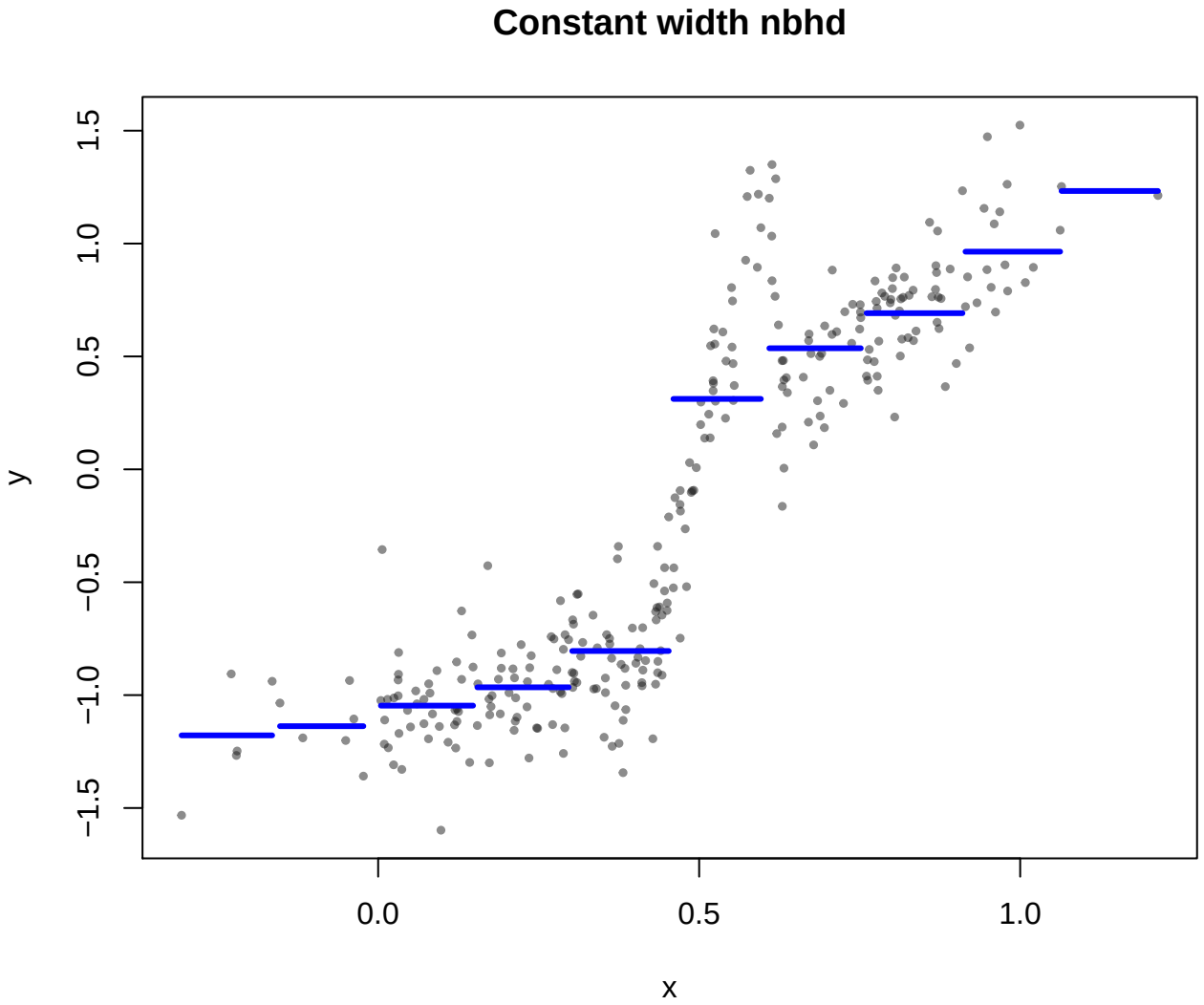
R.W. Oldford

Contents

---

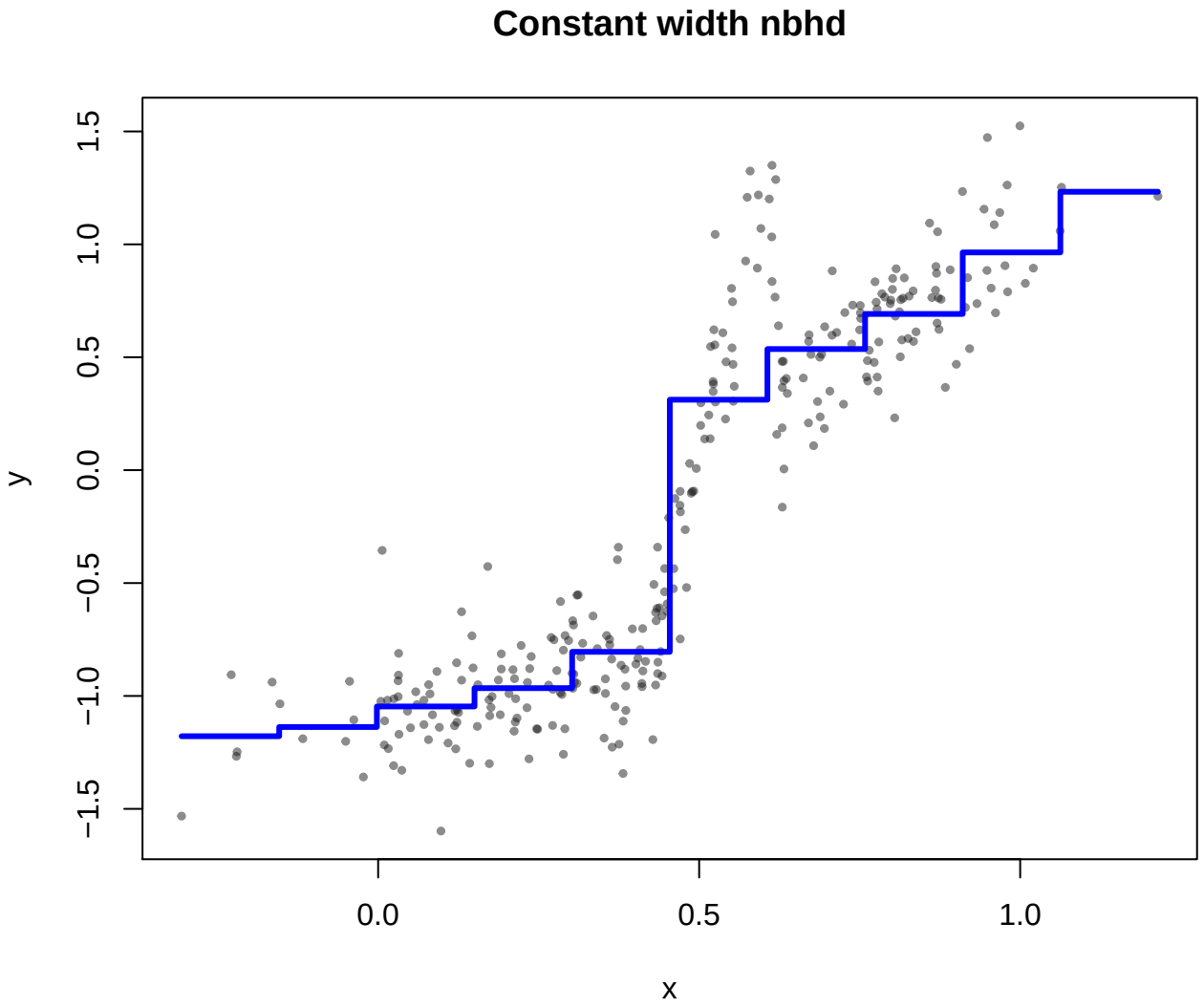
1 Revisiting neighbourhoods

When we began to look a smoothers, we considered condstructing a function locally by first dividing the x axis into neighbourhoods or regions. For example, for the fake data we had considered using the average of the ys in distinct non-overlapping neighbourhoods as in the plot below.

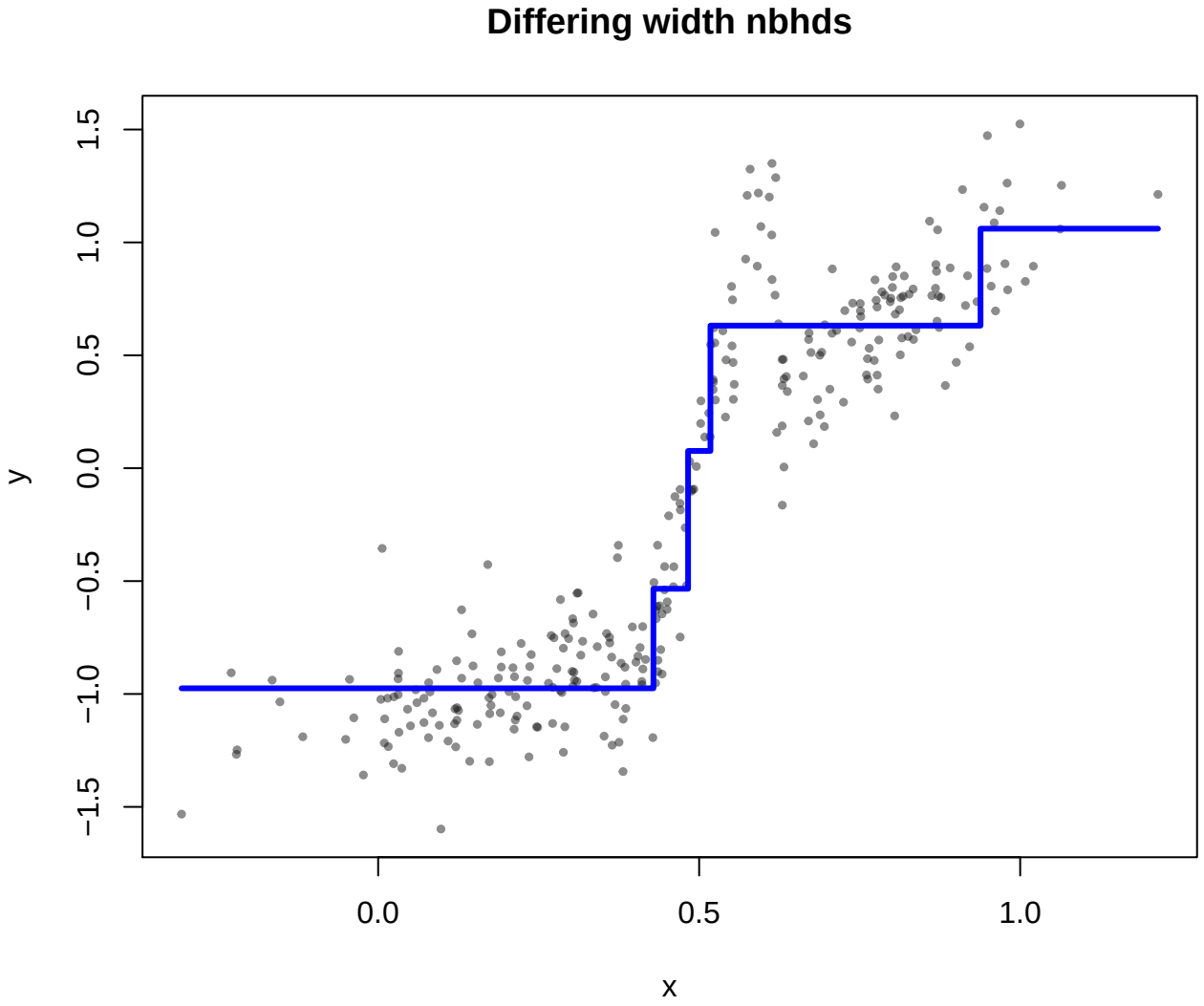


Previously, because the estimate $\widehat{\mu}(x)$ here is not smooth, we went to considerable lengths to make it smooth by using polynomials in each neighbourhood and enforcing continuity constraints between neighbourhoods.

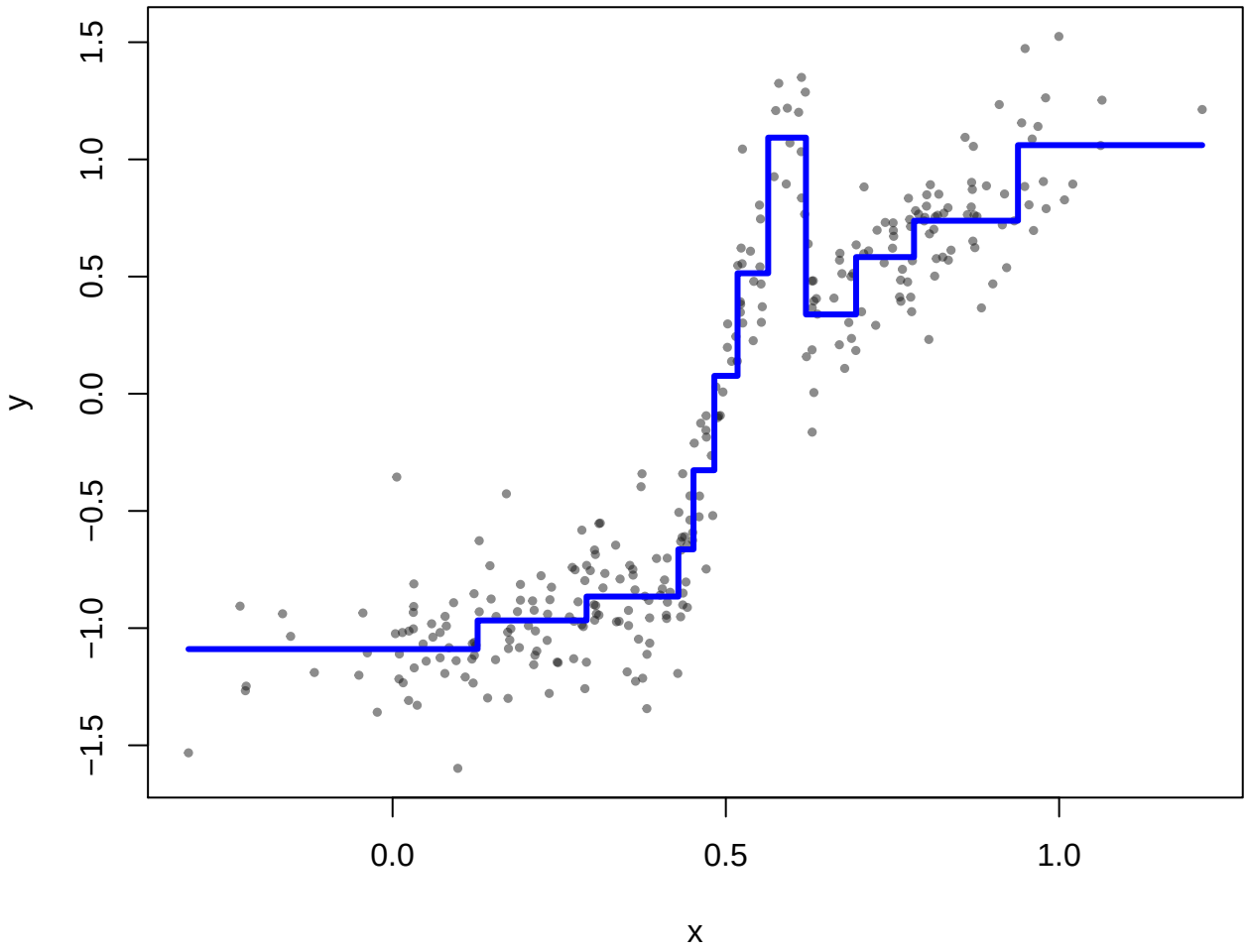
What if we chose not to force this smoothness on $\widehat{\mu}(x)$? Having flat pieces (the average of the ys in each neighbourhood) the function is at least easy to understand, especially when communicating it to others. We can even make it look a little more like our previous functions by making it continuous (if not differentiable everywhere):



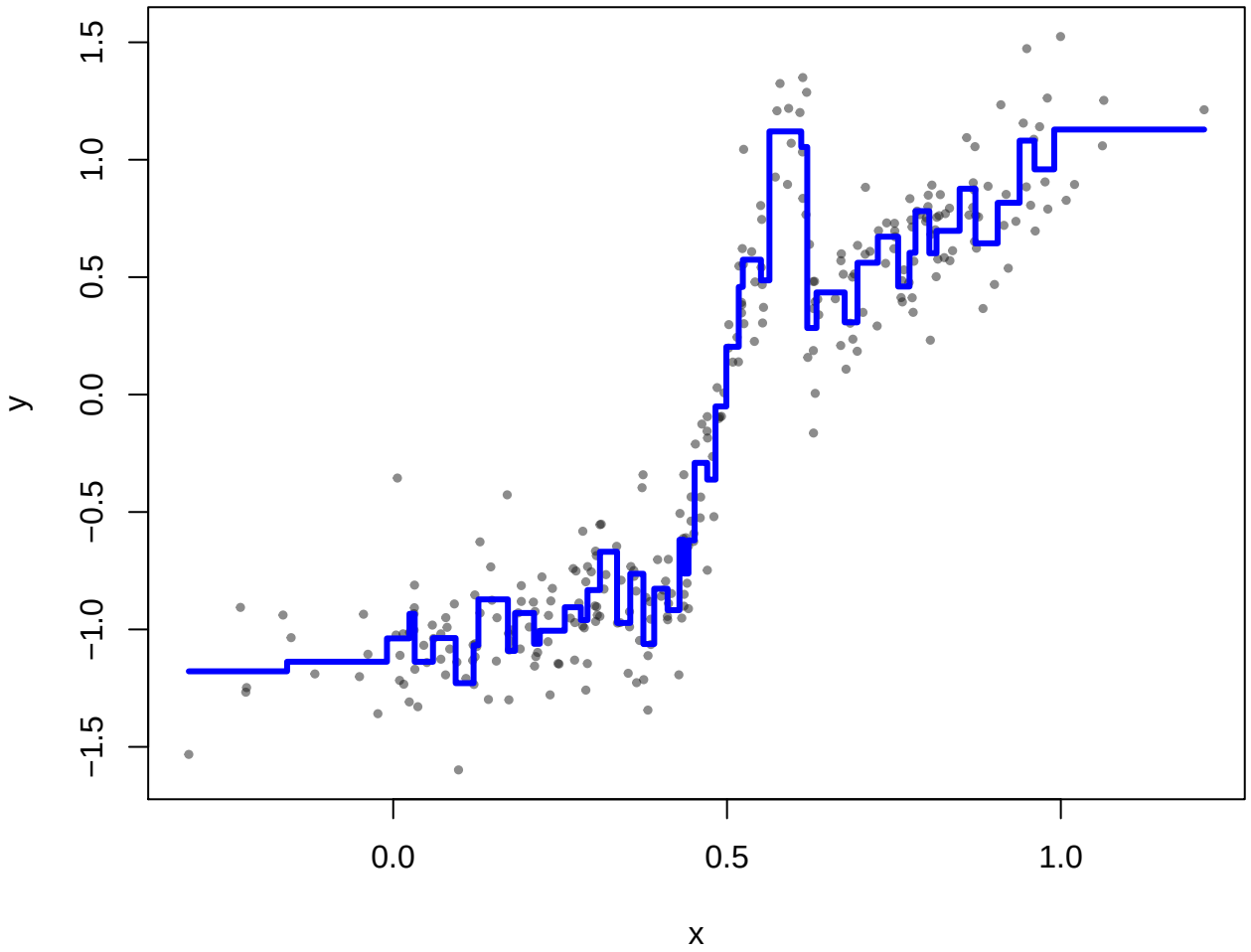
This was based on constant width neighbourhoods. Previously we also considered varying width but constant numbers of points in each neighbourhood. It might be that we could get a much better fit if we did not restrict the neighbourhoods to either be of constant width or of constant size. For example, the following has lots of points in some neighbourhoods and few in others. The neighbourhoods are also all of different widths.



Perhaps we could choose the neighbourhoods small enough where we need them to be small and get a better fit.



Or even



Clearly we can go too far. As with smoothers, we need some measure of the how well we are fitting and also how complex is the function that we are fitting.

How well we are fitting could simply be the residual sum of squares, as before. Now $\widehat{\mu}(x)$ is simply the average of the ys in each x neighbourhood. If there are K neighbourhoods, or regions, $R_1, \ldots, R_K$, each having n_k points ($\sum_{k=1}^{K} n_k = N$), then we can write

$$\begin{aligned}
\widehat{\mu}(x) &= \sum_{k=1}^{K} \frac{1}{n_k} \sum_{i=1}^{N} I_{R_k}(x)\ y_i \\
&= \sum_{k=1}^{K} I_{R_k}(x)\ \widehat{\mu}_k, \text{ say, where} \\
\widehat{\mu}_k &= \frac{1}{n_k} \sum_{i=1}^{N} I_{R_k}(x_i)\ y_i,
\end{aligned}$$

where the indicator function $I_A(x) = 1$ whenever $x \in A$ and is 0 otherwise.

The residual sum of squares is then

$$
\begin{aligned}
RSS(\widehat{\mu}) &= \sum_{i=1}^{N} \left(y_i - \widehat{\mu}(x_i)\right)^2 \\
&= \sum_{i=1}^{N} \left(y_i - \sum_{k=1}^{K} I_{R_k}(x_i)\ \widehat{\mu}_k\right)^2
\end{aligned}
$$

and the objective is to find non-overlapping regions R_k which cover all possible values of x.

Clearly, the more regions that are possible the smaller can be the $RSS(\widehat{\mu})$. As with smoothing splines we do not want too complex a function which means we do not want too many regions. The number of regions is a measure of the complexity of the resulting function $\widehat{\mu}(x)$. As with smoothing splines then, we could instead choose to to penalize complexity and minimize $RSS(\widehat{\mu}) + K$, say. That is minimize

$$
\sum_{i=1}^{N} \left(y_i - \sum_{k=1}^{K} I_{R_k}(x_i)\ \widehat{\mu}_k\right)^2 \quad + \quad \lambda\ K
$$

to trade off the quality of the fit with the complexity of the function being fitted. (N.B. complexity is being measured here by the number of distinct regions, or equivalently, the number of $\widehat{\mu}_k$s). The constant $\lambda \geq 0$ is again a "tuning" constant whose value can be chosen to increase (or decrease) the penalty for complexity.

Clearly K could be as large as N (one region for every distinct x_i, i.e. $n_k = 1$). Usually an upper bound on the complexity is imposed by not letting any region contain fewer than some minimum number

$$
n_{min} = \min_{k=1,\ldots,K}\ n_k
$$

of points. This bound guarantees that the complexity $K \leq N/n_{min}$.

Even with this restriction on complexity, there are still a great number of regions which might be considered for any value of K. Too many to imagine conducting a search over all possible partitions of the real line to find the best possible partition and its regions $R_1, \ldots, R_K$.

1.1 Using trees to construct $\widehat{\mu}(x)$

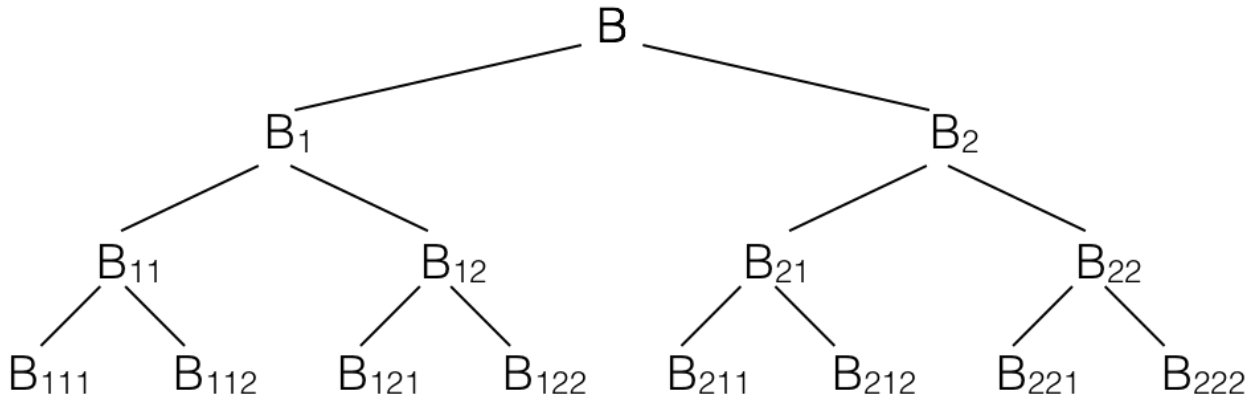
Instead, a greedy top-down algorithm is used. We first determine two neighbourhoods by finding a cut point c of the xs such that the line is split into two regions $B_1 = \{x \mid x < c\}$ and its complement $B_2 = \{x \mid x \geq c\}$. We choose c so that the residual sum of squares

$$
\begin{aligned}
RSS &= \left(y_i - \sum_{k=1}^{2} I_{B_k}(x_i)\ \widehat{\mu}_k\right)^2 \\
&= \sum_{i \in B_1} \left(y_i - \overline{y}_{B_1}\right)^2 + \sum_{i \in B_2} \left(y_i - \overline{y}_{B_2}\right)^2
\end{aligned}
$$

is as small as possible.

Once c is chosen, we then repeat the process to separately split each of B_1 and B_2. That is, for B_1 we choose another c to get B_{11} and B_{12} with $B_1 = B_{11} \bigcup B_{12}$, and similarly we split B_2 by choosing a different c to get B_{21} and B_{22}, again with $B_2 = B_{21} \bigcup B_{22}$.

Continuing recursively in this fashion, we construct a partition which looks like a binary tree:



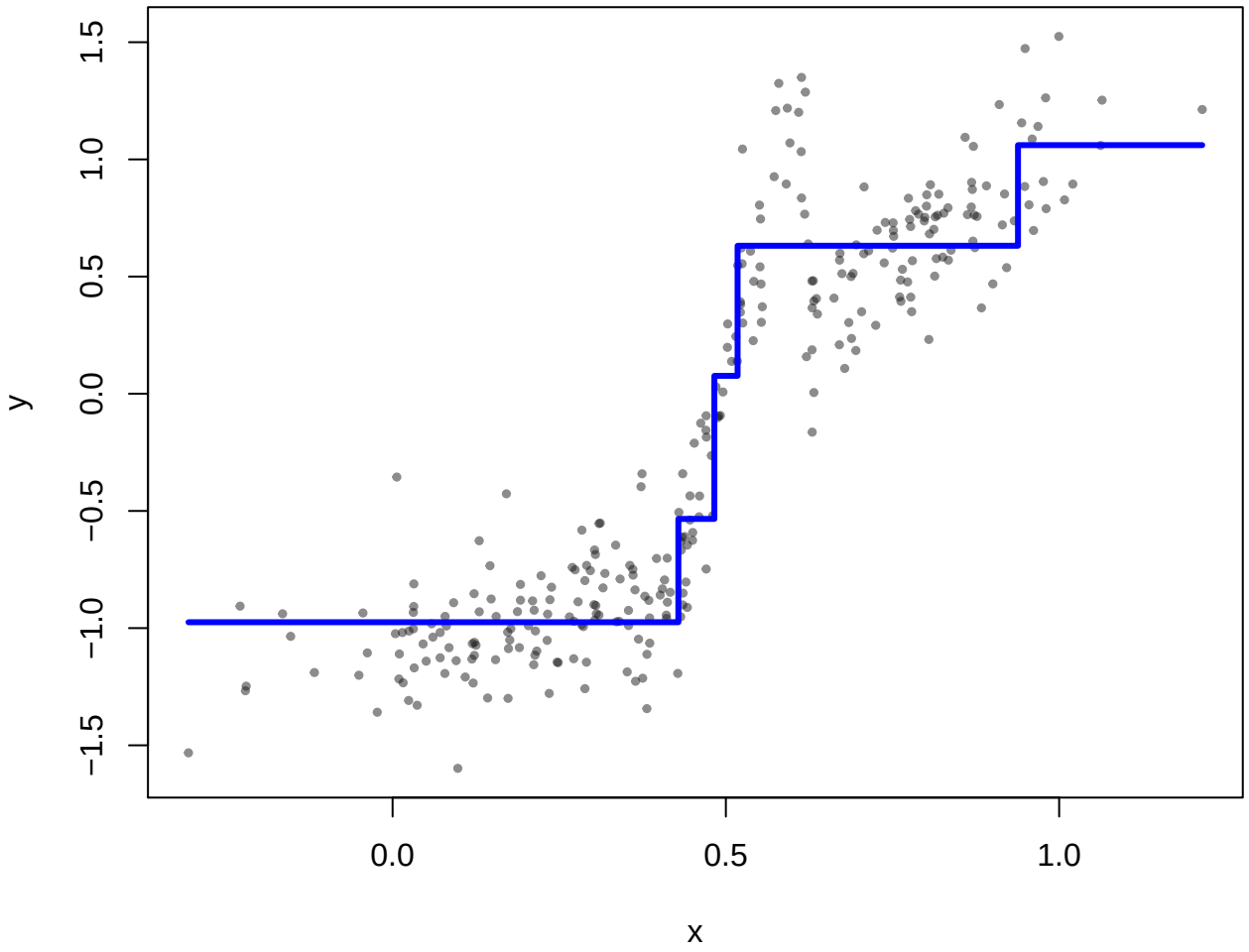
Note that the **root** of the tree is B and represents the whole real line for x in this instance and the **leaf nodes** (bottom-most nodes) correspond to the regions $R_1, \ldots, R_K$. In the above diagram $K = 8$. All other interior nodes are **branches**. (The tree, regrettably, is upside down compared to nature.)

This **recursive partitioning** could continue until every leaf node was as small as possible. We need not choose to split a node until it is as small as possible. We might, for example, choose to not split a node (region) if the result does not change the RSS enough.

We can accomplish this kind of recursive partitioning in `R` using the `tree` function from the package of the same name. As with other fitting functions, `tree(...)` takes a `formula` expression/object as its first argument.

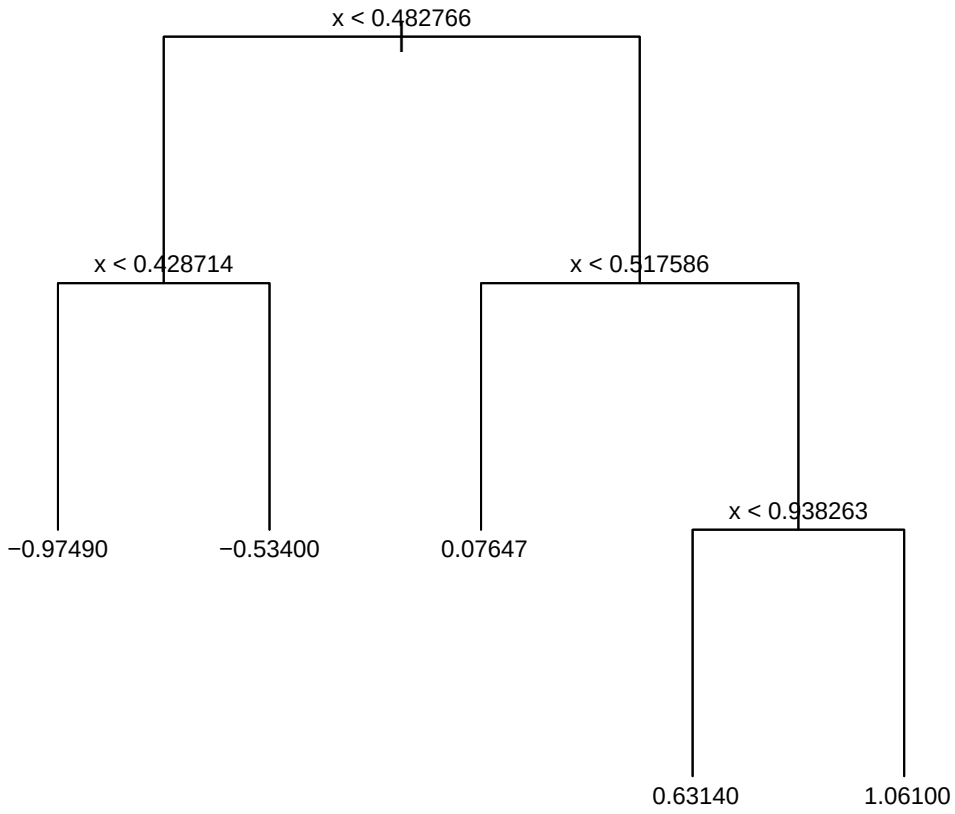
For our fake data, the default fit of a tree via recursive partitioning is had as follows:

```
library(tree)
fake.tree <- tree(y ~ x)

# the fit can now be plotted on our fake data as
# before
plot(x,y,
     col=adjustcolor("grey10", 0.5), pch=19, cex=0.5,
     main = "tree(y ~ x) on fake data")
#
# This plots the fitted values
partition.tree(fake.tree, add=TRUE, col="blue", lwd=3)
```



The tree structure itself can also be displayed

```
plot(fake.tree, type="uniform")
# And add the labels
text(fake.tree, cex=0.75)
```



At each binary split, the value used to separate the subsets is shown. For example from the root node, all data points having $x < 0.482766$ form the left branch, all others form the right. The left branch is split once more depending on whether $x < 0.428714$ or not. At each leaf, the average of the y values for all data points in that set is also reported. The leftmost branch in the display has an average y of -0.97490 for all points having $x < 0.428714$ and its partner had average y of -0.5340 for $0.428714 \leq x < 0.482766$.
Similarly, on the right side of the root, we have 0.07647 for $0.482766 \leq x < 0.517586$, 0.63140 for $0.517586 \leq x < 0.938263$, and 1.06100 for all $x \geq 0.938263$

The fitted tree is a fitted model like any other and so can be summarized

```
summary(fake.tree)
```

```
##
## Regression tree:
## tree(formula = y ~ x)
## Number of terminal nodes:  5
## Residual mean deviance:  0.06125 = 18.07 / 295
## Distribution of residuals:
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
## -0.79510 -0.16280 -0.00739  0.00000  0.13840  0.71840
```

Note that the model degrees of freedom here is equivalent to the number of terminal nodes (or leaves) in the tree. Similarly there are residuals

```
resids <- residuals(fake.tree)
```

whose distribution is summarized above, and are used to form the residual mean deviance as

```
sum(resids^2) / (length(y) - 5)
```

```
## [1] 0.0612532
```

Note also that the average y values are actually the values of $\widehat{\mu}(x)$ for all x in the set defined by that leaf. Indeed, as with any previous fitted model, the fitted tree defined $\widehat{\mu}(x)$ for all x and so can be used to predict y values at any x by calling the `predict(...)` function on the fitted tree with those x values.

```
# Get the x-values
#
newx <- seq(-0.5, 1.5, 0.5)
muhat <- predict(fake.tree, newdata=list(x=newx))
knitr::kable(round(cbind(x = newx, muhat = muhat),3))
```

x	muhat
-0.5	-0.975
0.0	-0.975
0.5	0.076
1.0	1.061
1.5	1.061

The tree structure and other summary statistics at each node can be seen from the printed representation of the tree as well:

```
fake.tree
```

```
## node), split, n, deviance, yval
##       * denotes terminal node
##
##  1) root 300 206.3000 -0.18800
##    2) x < 0.482766 161  12.1200 -0.90370
##      4) x < 0.428714 135   6.2920 -0.97490 *
##      5) x > 0.428714 26   1.5930 -0.53400 *
##    3) x > 0.482766 139  16.2000  0.64090
##      6) x < 0.517586 10   0.1966  0.07647 *
##      7) x > 0.517586 129  12.5700  0.68470
##       14) x < 0.938263 113   9.0780  0.63140 *
##       15) x > 0.938263 16   0.9101  1.06100 *
```

At each node of the tree, the (nested) splitting rule used to define the node, the number of observations in the node, the deviance of the response values at that node, and the average response, are all shown. Leaves of the tree are marked with an asterisk.

The same information is also available as a `data.frame` from the fitted tree as

```
fake.tree.df <- fake.tree$frame
fake.tree.df
```

```
##       var   n         dev        yval splits.cutleft splits.cutright
## 1       x 300 206.2903808 -0.18801841      <0.482766       >0.482766
## 2       x 161  12.1218607 -0.90367378      <0.428714       >0.428714
## 4  <leaf> 135   6.2918772 -0.97486313
## 5  <leaf>  26   1.5933942 -0.53403676
## 3       x 139  16.2012278  0.64090615      <0.517586       >0.517586
## 6  <leaf>  10   0.1966329  0.07646583
## 7       x 129  12.5716951  0.68466121      <0.938263       >0.938263
## 14 <leaf> 113   9.0777266  0.63140575
## 15 <leaf>  16   0.9100638  1.06077791
```

Here, deviance is the residual sum of squares:

```r
deviance(fake.tree)
```

```
## [1] 18.06969
```

```r
sum(resids^2)
```

```
## [1] 18.06969
```

```r
# At the root
n <- length(y)
(n-1)* var(y)
```

```
## [1] 206.2904
```

```r
mean(y)
```

```
## [1] -0.1880184
```

```r
# At the terminal nodes (leaves)
leaves <- unique(fake.tree$where)
# number of leaves
length(leaves)
```

```
## [1] 5
```

```r
# For just the first leaf, node id is
leaves[1]
```

```
## [1] 3
```

```r
# which identifies the row of the data frame for this
# leaf
fake.tree.df[leaves[1],]
```

```
##      var   n      dev       yval splits.cutleft splits.cutright
## 4 <leaf> 135 6.291877 -0.9748631
```

```r
# whose values we could reconstruct
# from the contents of the leaf.
# Here are all the elements of leaf 1
leaf1 <- fake.tree$where==leaves[1]

# size of leaf1
sum(leaf1)
```

```
## [1] 135
```

```r
# RSS at leaf1
sum(resids[leaf1]^2)
```

```
## [1] 6.291877
```

```r
# muhat(x)  (for x in leaf1)
mean(y[leaf1])
```

```
## [1] -0.9748631
```

```r
#
# Similarly at an interior node

interior <- fake.tree.df[,1] != "<leaf>"
sum(interior)
```

```
## [1] 4
```

```
#
# last interior node is made up of the last two leaves
# in rows 8 and 9
left.leaf  <-  fake.tree$where==8
right.leaf <-  fake.tree$where==9

#
#  number of data points
node.n <- sum(left.leaf) + sum(right.leaf)
node.n
```

```
## [1] 129
```

```
# Their y values:
node.y <- c(y[left.leaf],y[right.leaf])

# RSS for that node
(node.n -1) * var(node.y)
```

```
## [1] 12.5717
```

```
# average response
mean(node.y)
```

```
## [1] 0.6846612
```

1.2 How large could the tree be?

Clearly, the more splits we make the larger will be the tree and the more levels will be given to $\widehat{\mu}(x)$. Each new level corresponds to another model degree of freedom. In principle, we could continue to split until every terminal node contains a single unique x value for the data.

A bushier tree can be had via two different parameters of the `tree` function:

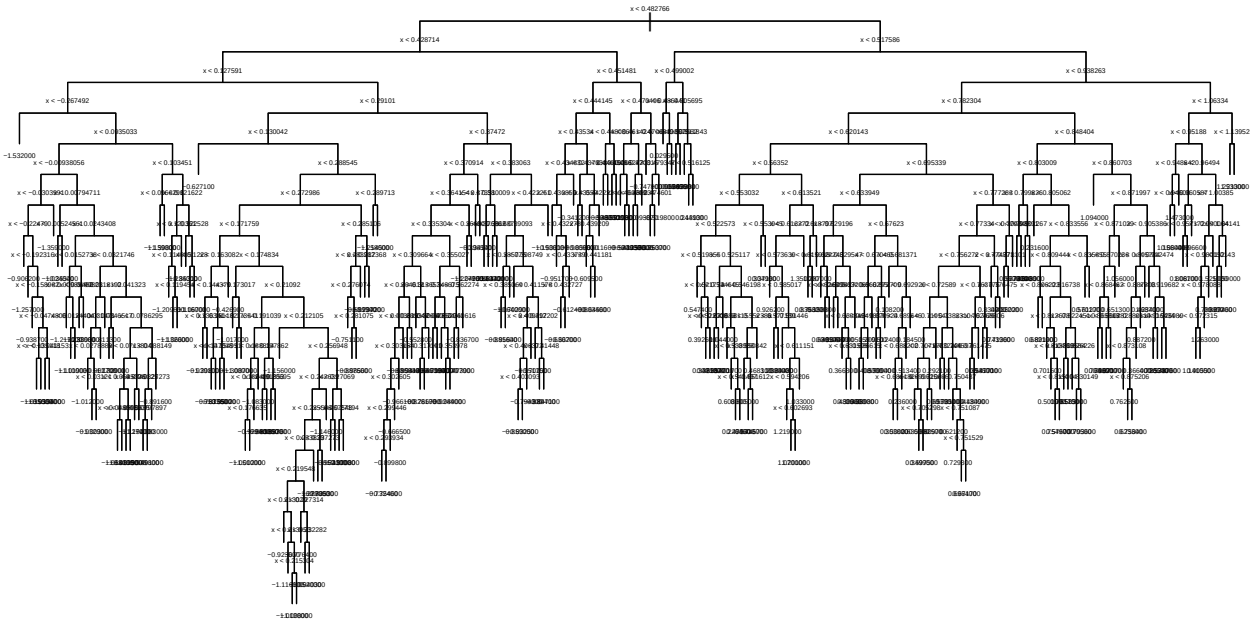
- `mindev`, the "within-node deviance must be at least this times that of the root node for the node to be split"
- `mincut`, the smallest allowed number of observations in any child (leaf) node.
- `minsize`, the smallest allowed (interior) node size (typically at least twice the `mincut`)

Below, we set these parameters to allow the greatest number of possible splits and hence the bushiest tree possible. Note that this produces a $\widehat{\mu}(x)$ function that tracks as closely to the observed data as possible.

```
fake.tree.bushy <- tree(y ~ x,
                  control=tree.control(nobs=length(x),

                                       mindev = 0,
                                       minsize= 2,
                                       mincut = 1
                                       )
                  )
```

The tree which results is very bushy.

```
plot(fake.tree.bushy, type="uniform")
# And add the labels
text(fake.tree.bushy, cex=0.35)
```

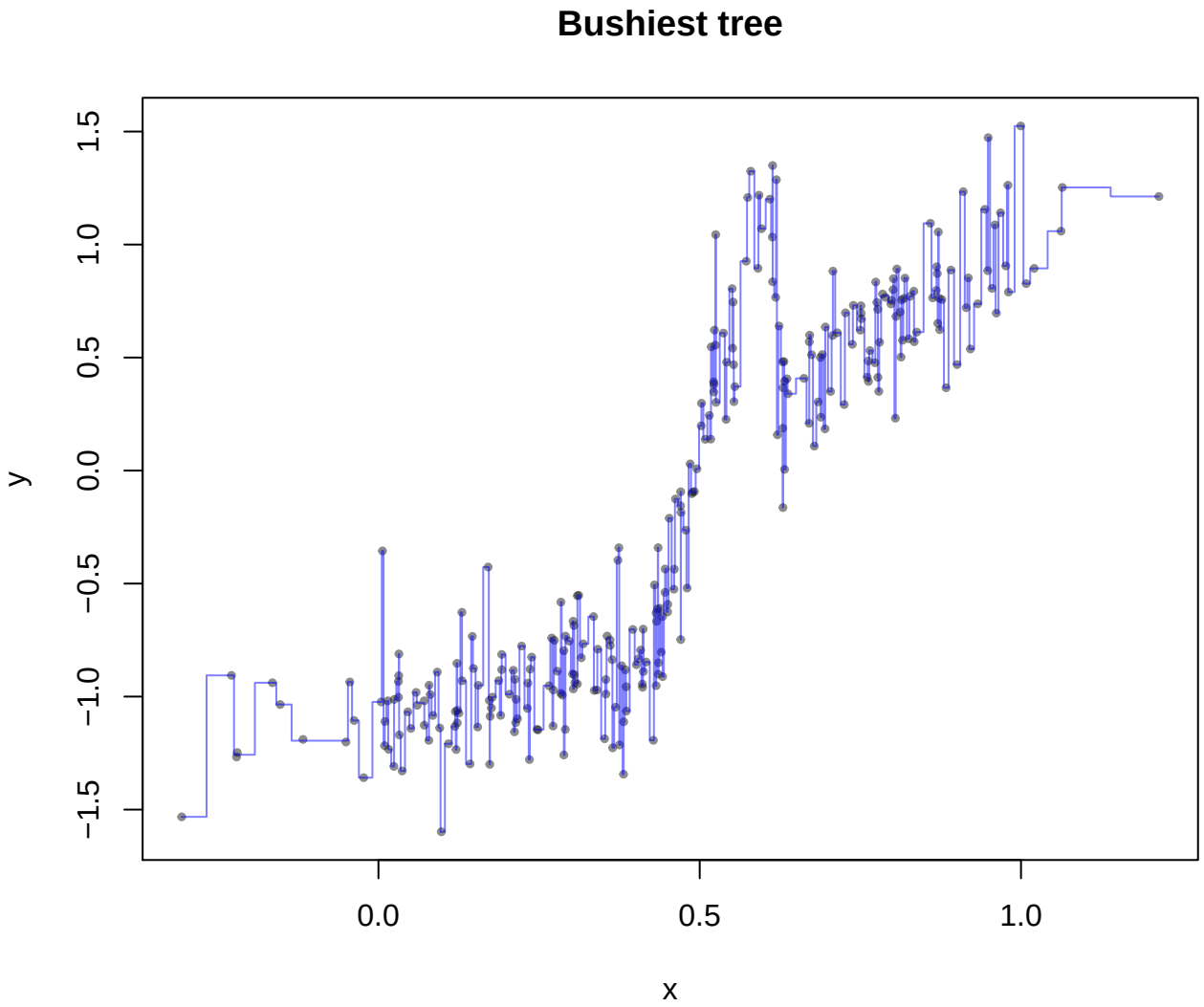


```
summary(fake.tree.bushy)
```

```
##
## Regression tree:
## tree(formula = y ~ x, control = tree.control(nobs = length(x),
##     mindev = 0, minsize = 2, mincut = 1))
## Number of terminal nodes:  286
## Residual mean deviance:  7.678e-05 = 0.001075 / 14
## Distribution of residuals:
##       Min.    1st Qu.     Median       Mean    3rd Qu.       Max.
## -0.009845  0.000000  0.000000  0.000000  0.000000  0.009845
```

Not surprisingly, a bushy tree corresponds to a flexible $\widehat{\mu}(x)$, one which uses an enormous number of degrees of freedom. The fit will be able to track the data very closely.

```
plot(x,y,
     col=adjustcolor("grey10", 0.5), pch=19, cex=0.5,
     main = "Bushiest tree")
partition.tree(fake.tree.bushy, add=TRUE,
               col=adjustcolor("blue", 0.5), lwd=1)
```



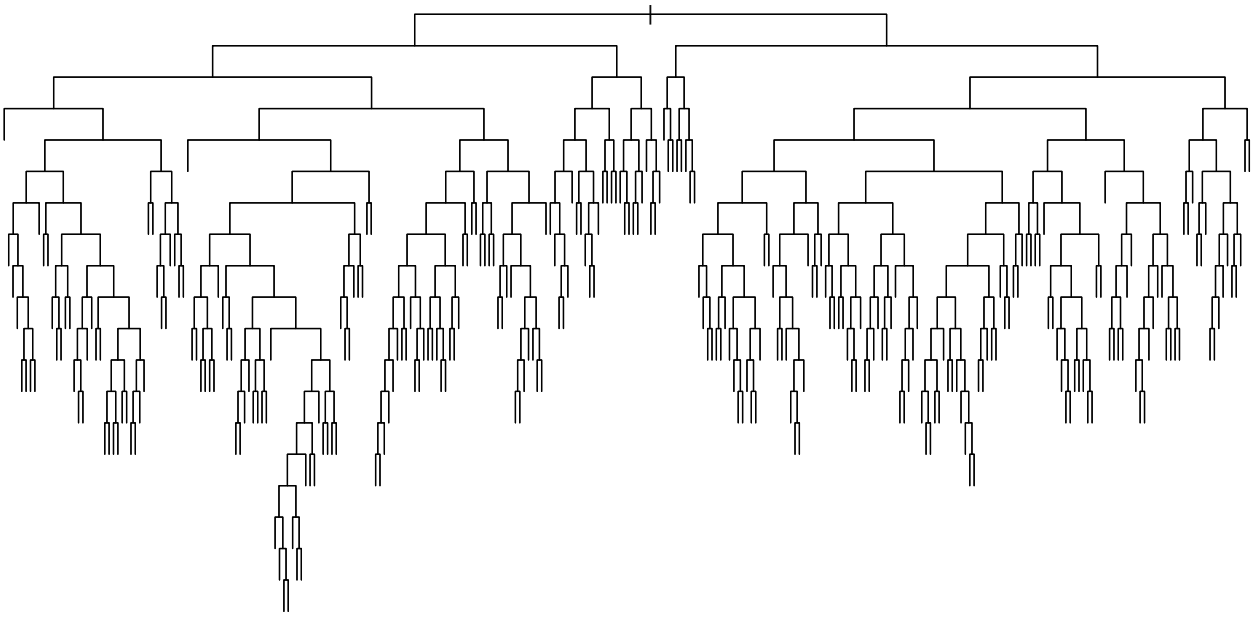
This fit goes very nearly through every available data point.

Clearly, the tree requires pruning …but how?

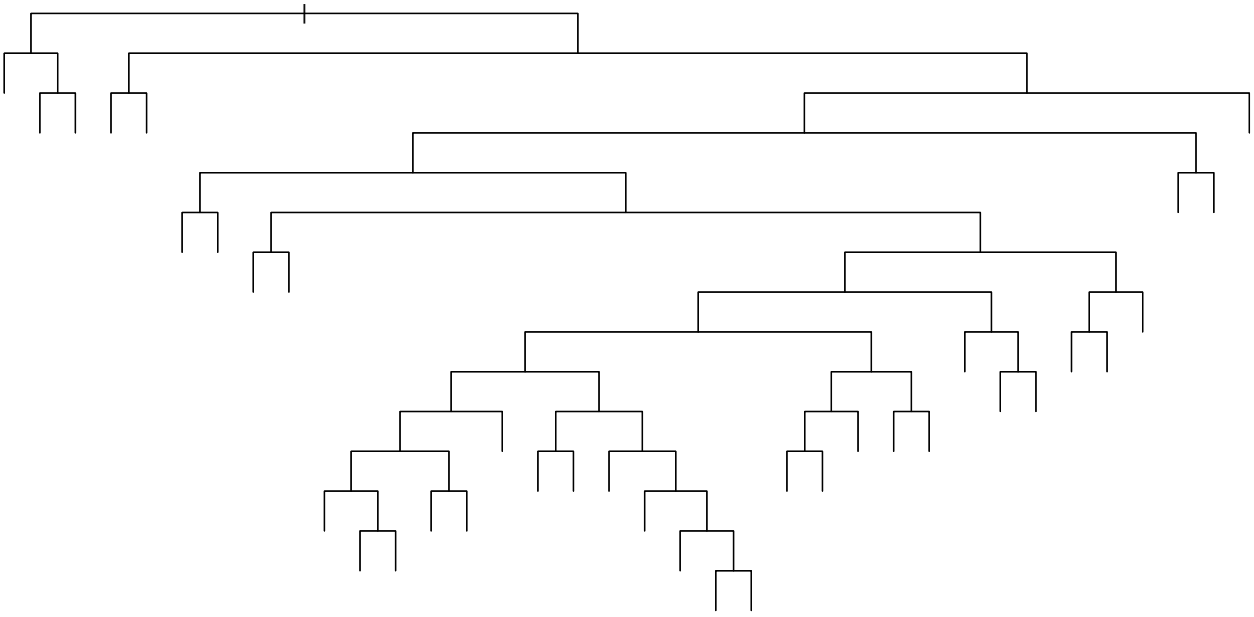
1.3 Pruning a tree

Begin with the bushy tree, we might want to cut off some branches, that is we might want to prune the tree a bit. We could do this interactively using the function `snip.tree(...)`. (N.B. this is **not** recommended; we are doing it here only for illustration.) For example, our bushy tree looked like

```
plot(fake.tree.bushy, type="uniform")
```



```
#
# Which we could prune by snipping off named nodes
fake.tree.snipped <- snip.tree(fake.tree.bushy,
                               nodes = c(4, 10, 11, 12, 13,
                                         112, 113, 228, 229,
                                         58, 59, 15))
# or, if the nodes argument is NOT given, by doing
# it interactively
# fake.tree.snipped <- snip.tree(fake.tree.bushy)
plot(fake.tree.snipped, type="uniform")
```



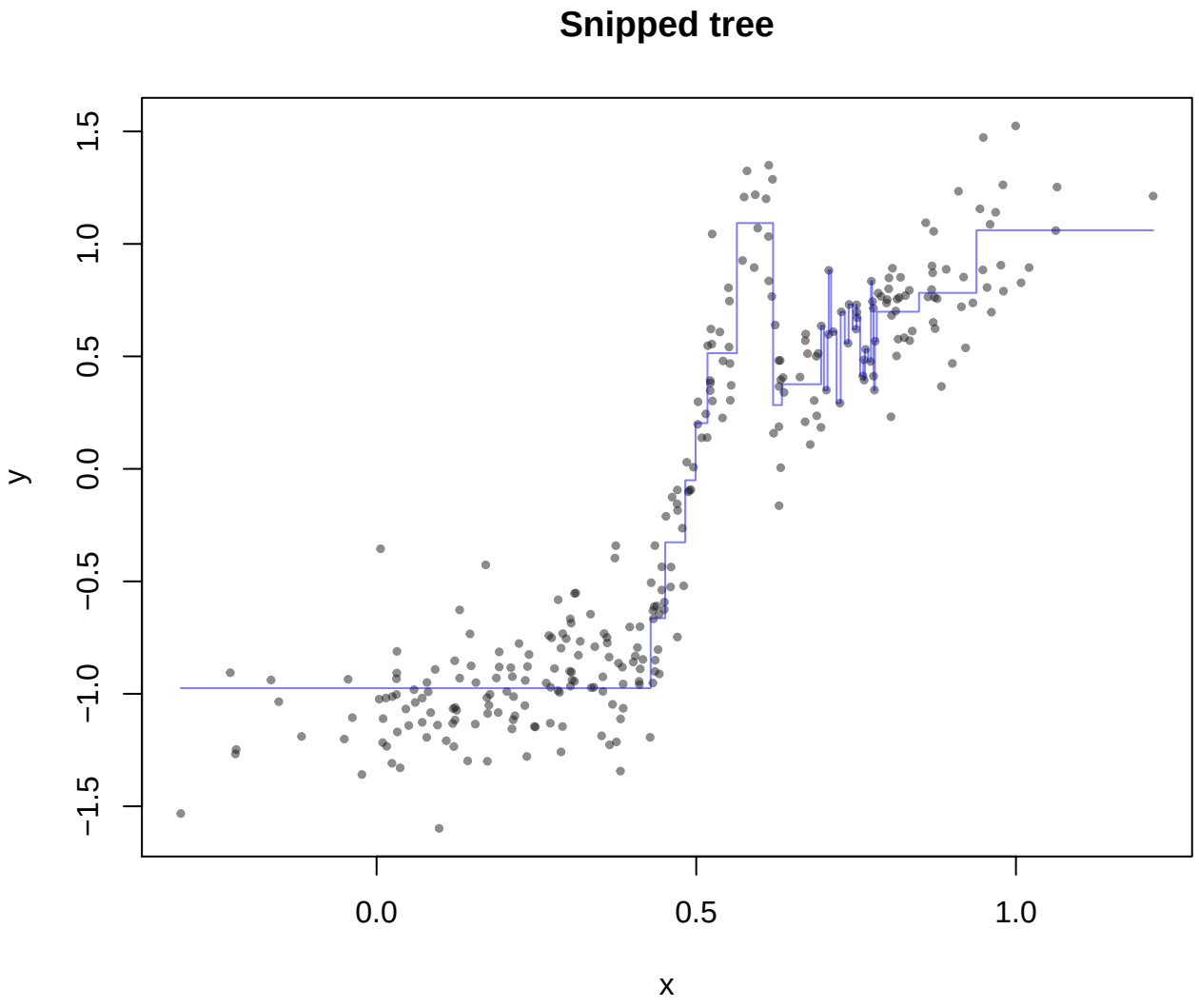
As can be seen, the resulting tree is smaller than the bushy tree having `sum(fake.tree.snipped$frame[,"var"] == "<leaf>")` = 36 leaves compared to 286 leaves on the bushy tree.

```r
# The summary is
summary(fake.tree.snipped)
```

```
##
## Regression tree:
## snip.tree(tree = fake.tree.bushy, nodes = c(4L, 10L, 11L, 12L,
## 13L, 112L, 113L, 228L, 229L, 58L, 59L, 15L))
## Number of terminal nodes:  36
## Residual mean deviance:  0.04294 = 11.34 / 264
## Distribution of residuals:
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## -0.6233 -0.1315  0.0000  0.0000  0.1075  0.6335
```

It also produces a less complex fit $\widehat{\mu}(x)$, as seen below:

```r
plot(x,y,
     col=adjustcolor("grey10", 0.5), pch=19, cex=0.5,
     main = "Snipped tree")
partition.tree(fake.tree.snipped, add=TRUE,
               col=adjustcolor("blue", 0.5), lwd=1)
```

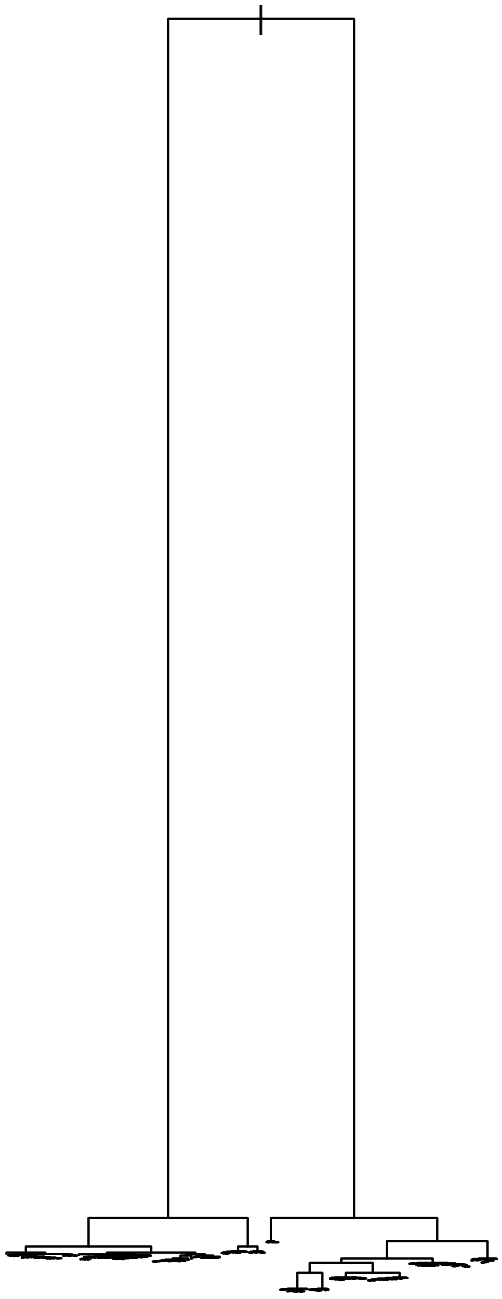


As the plot shows, snipping the tree just based on viewing the tree itself, has resulted in a fit $\widehat{\mu}(x)$ that has little variation in some regions (the left of the plot, or leftmost branch near the root) and considerably more in other regions (right side od the plot and right most branch). This may not be the best fitting tree for that number of leaves/regions (complexity).

In pruning, lopping of large branches near the root generally results in a really poor fit to a (potentially) large fraction of the data (e.g. imagine cutting all branches below the first right hand split from the root).

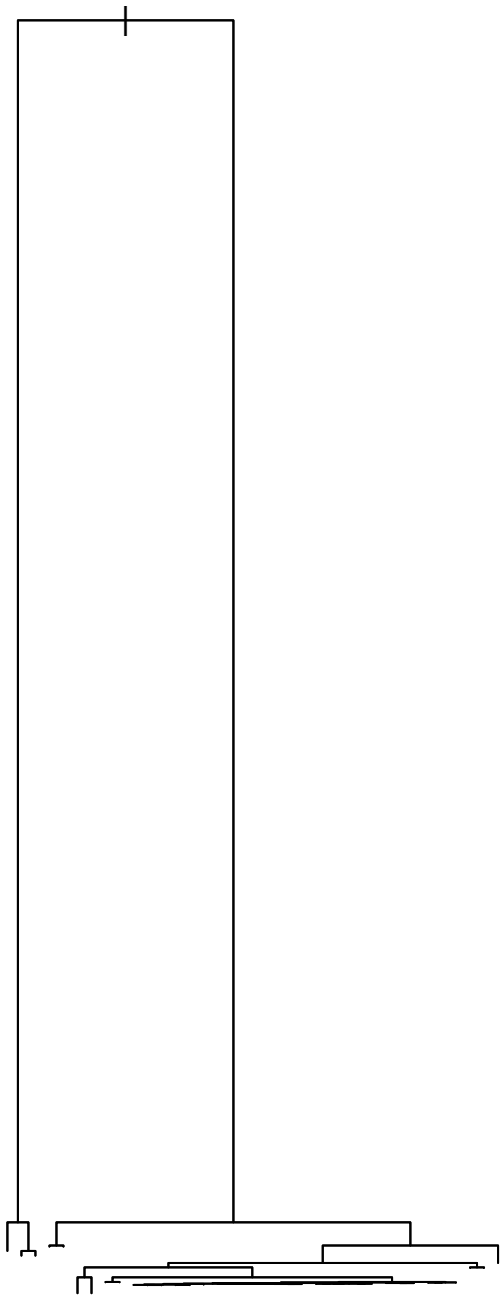
Some sense of the relative merits of the branches of our tree can be had by plotting it with 'type="proportional" (the default). This will cause the length of the branches to be proportional to the decrease they cause in RSS.

```
plot(fake.tree.bushy, type="proportional")
```



Similarly for the tree produced by just snipping the tree by eye.

```
plot(fake.tree.snipped, type="proportional")
```



Looking at this tree, we might consider snipping again, this time snipping away those branches that have lots of leaves but which add very little length to the tree.

What would be preferable would be a more systematic way to prune the tree, one that recognizes that bushy branches introduce complexity (add more regions) and that long branches are more desirable than short ones (long branches correspond to large changes in RSS).

1.3.1 What is the best pruning?

One goal is to reduce complexity and if we snip one branch at a time near the edge, each snip will reduce the complexity of our model by 1 (as measured by the number of terminal nodes or leaves being one fewer than

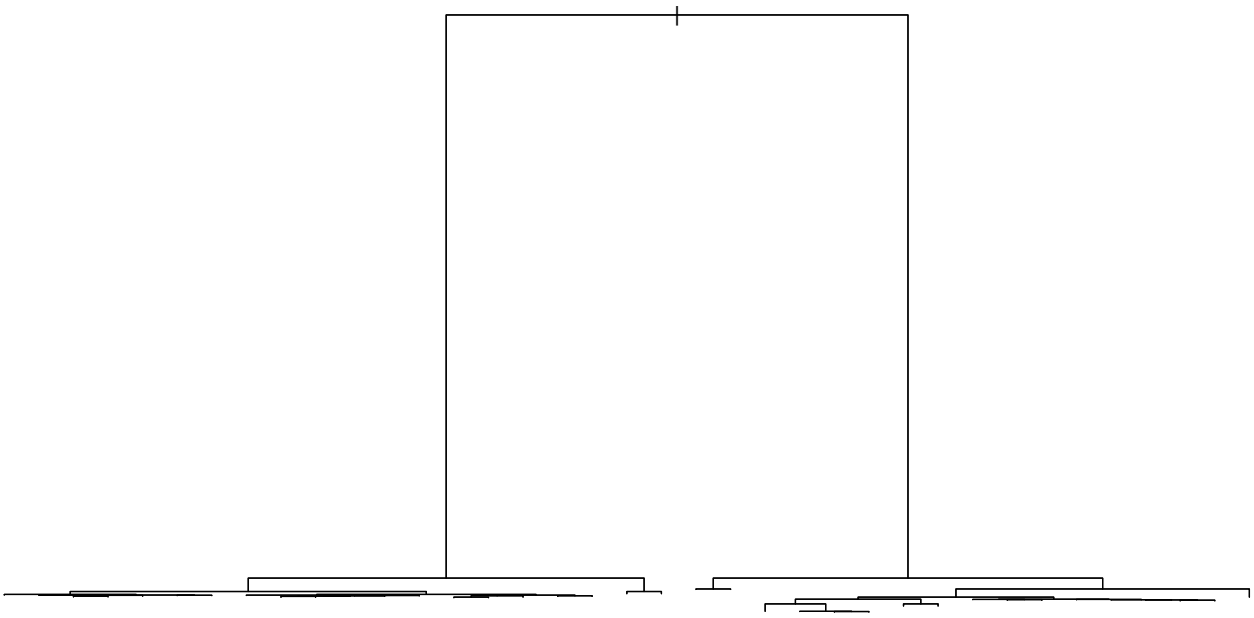
before). However, we also want to have the remaining model fit the data as well as possible. This suggests that we should choose the branch that (once snipped) will increase the RSS the least (i.e. have the least cost to the performance of our fitted model).

Looking at the pruning in this way, there is actually a sequence of subtrees which we can follow. In fact, it can be shown that the set of subtrees which minimize the cost-complexity

$$RSS + \lambda K$$

is nested. As λ increases, the optimal trees are had by a sequence of snip operations on the current tree (i.e. pruning). This produces a sequence of trees from the bushiest down to the root node. (Note however that as a λ increases, more than one node may be pruned at each snip.)

There is a function `prune.tree(...)` in the `tree` package that will allow us to choose the "best trees"" in this sequence of trees. Beginning with the bushiest tree, we could find the best sub-tree with some specified number of leaves. The snipped tree had 36 leaves; we might use `prune.tree` to find the best tree with that many leaves.

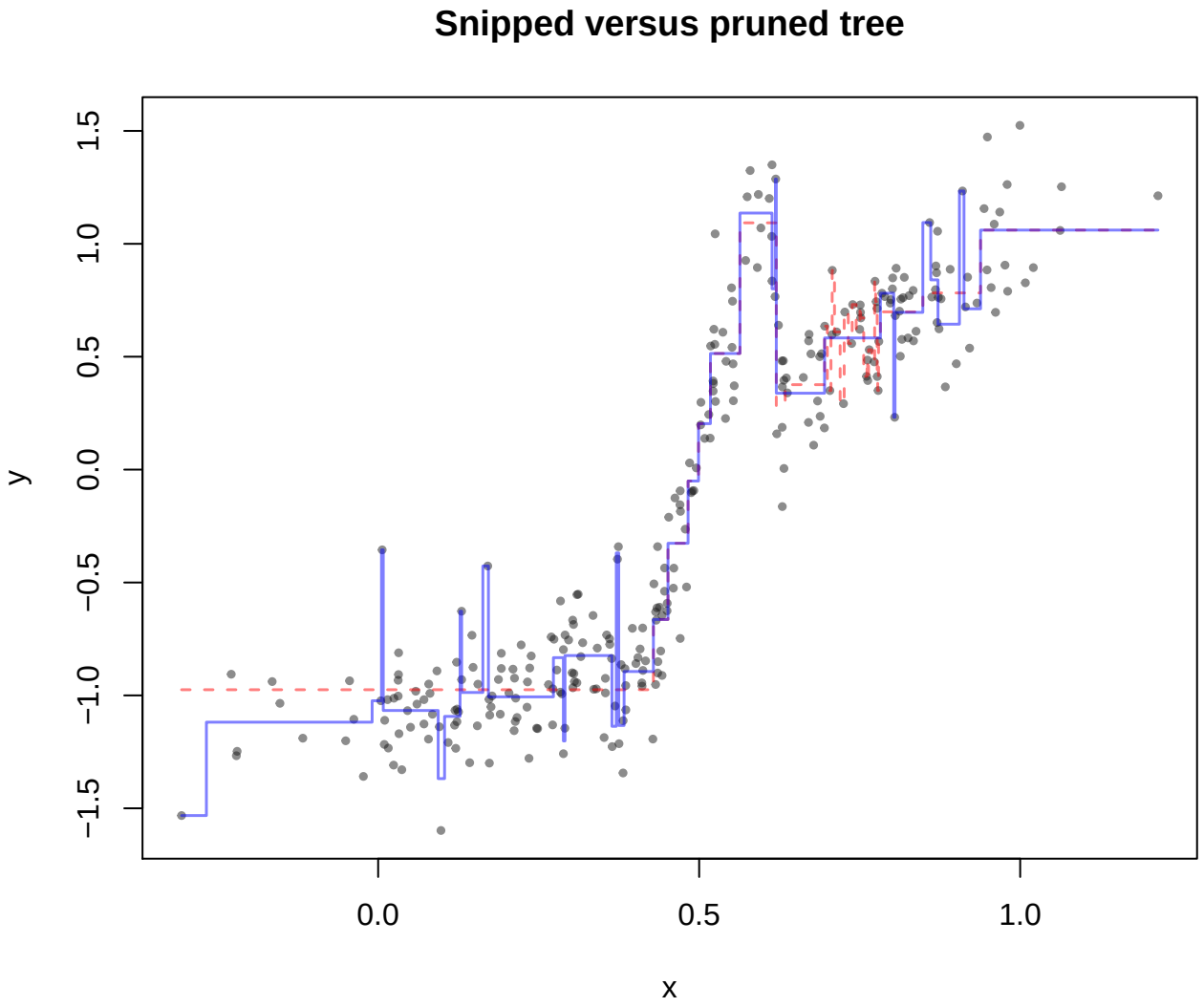


```
##
## Regression tree:
## snip.tree(tree = fake.tree.bushy, nodes = c(77L, 454L, 116L,
## 75L, 13L, 12L, 153L, 959L, 235L, 71L, 149L, 78L, 226L, 297L,
## 238L, 592L, 478L, 68L, 139L, 10L, 115L, 152L, 79L, 114L, 112L,
## 11L, 15L, 70L))
## Number of terminal nodes:  37
## Residual mean deviance:  0.02782 = 7.317 / 263
## Distribution of residuals:
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
## -0.50200 -0.11380  0.00680  0.00000  0.09413  0.52990
```

which produced one more leaf than we had from snipping. We can compare this function with the one produced by snipping in the same plot.

```
plot(x,y,
     col=adjustcolor("grey10", 0.5), pch=19, cex=0.5,
     main = "Snipped versus pruned tree")
partition.tree(fake.tree.snipped, add=TRUE,
```

```
              col=adjustcolor("red", 0.5), lwd=1.5, lty=2)
partition.tree(fake.tree.snip.best, add=TRUE,
              col=adjustcolor("blue", 0.5), lwd=1.5)
```

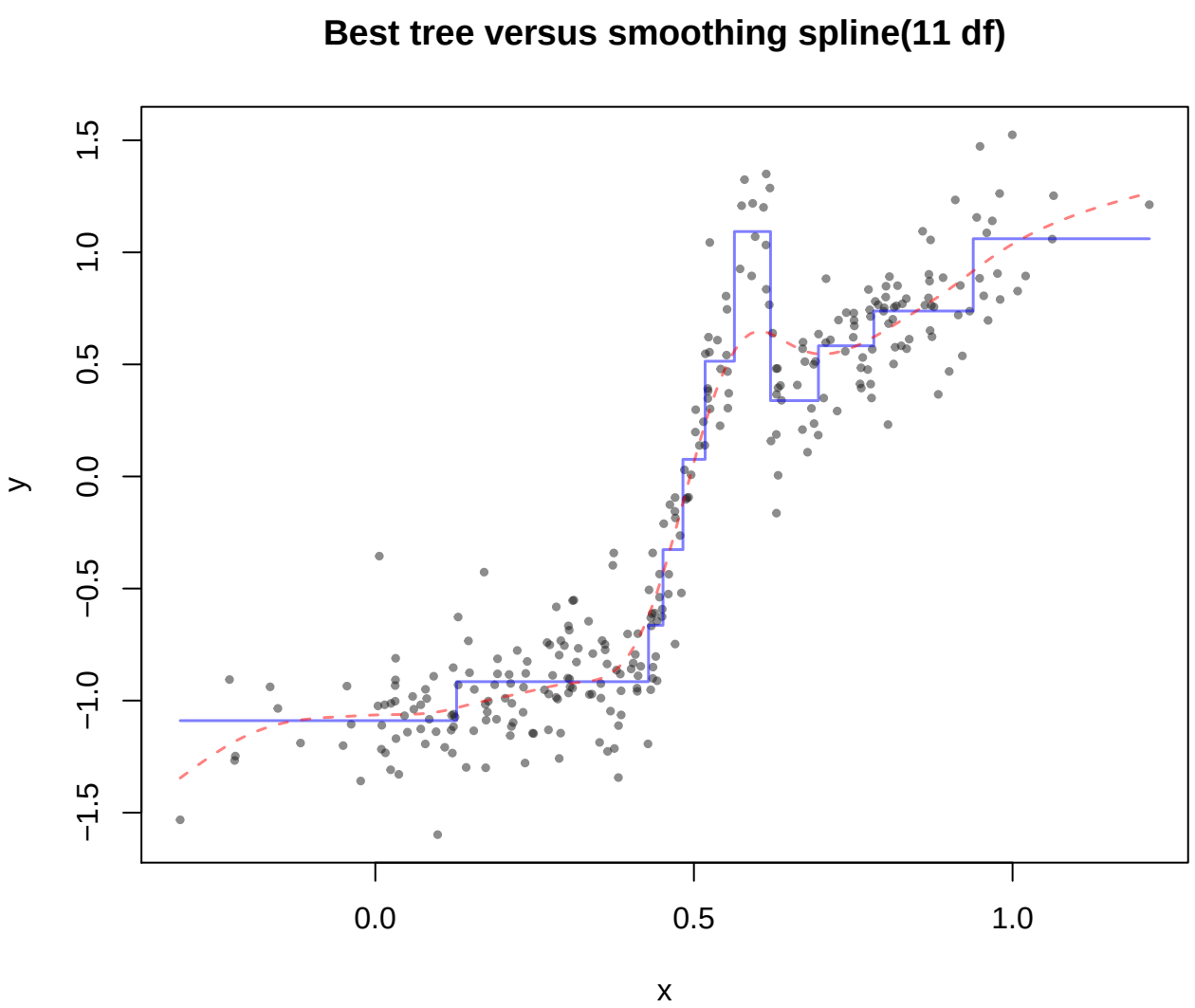


The snipped tree (in dashed red) has introduced its complexity in essentially one place. This is because we snipped off most branches close to the root, and let only one grow bushy. In contrast, the "best" pruned tree has spread the same complexity across several different x locations. The "best" pruned tree has removed small branches across the entire tree, depending on which branches will least affect the fit when pruned.

There is nevertheless more complexity than we might wish to consider for this data. Suppose we choose to have only 11 regions (which corresponds to the 11 effective degrees of freedom we often used for smoothers on this data).

```
fake.tree.best11 <- prune.tree(fake.tree.bushy, best=11)
plot(x,y,
     col=adjustcolor("grey10", 0.5), pch=19, cex=0.5,
     main = "Best tree versus smoothing spline(11 df)")
partition.tree(fake.tree.best11, add=TRUE,
              col=adjustcolor("blue", 0.5), lwd=1.5)
#
# which we could compare to a smoothing spline
```

```
library(splines)
fake.data.smooth11 <- smooth.spline(y~x, df=11)
newx <- seq(min(x),max(x), length.out = 200)
lines(newx, predict(fake.data.smooth11, x=newx)$y,
      col=adjustcolor("red", 0.5), lty=2, lwd=1.5)
```

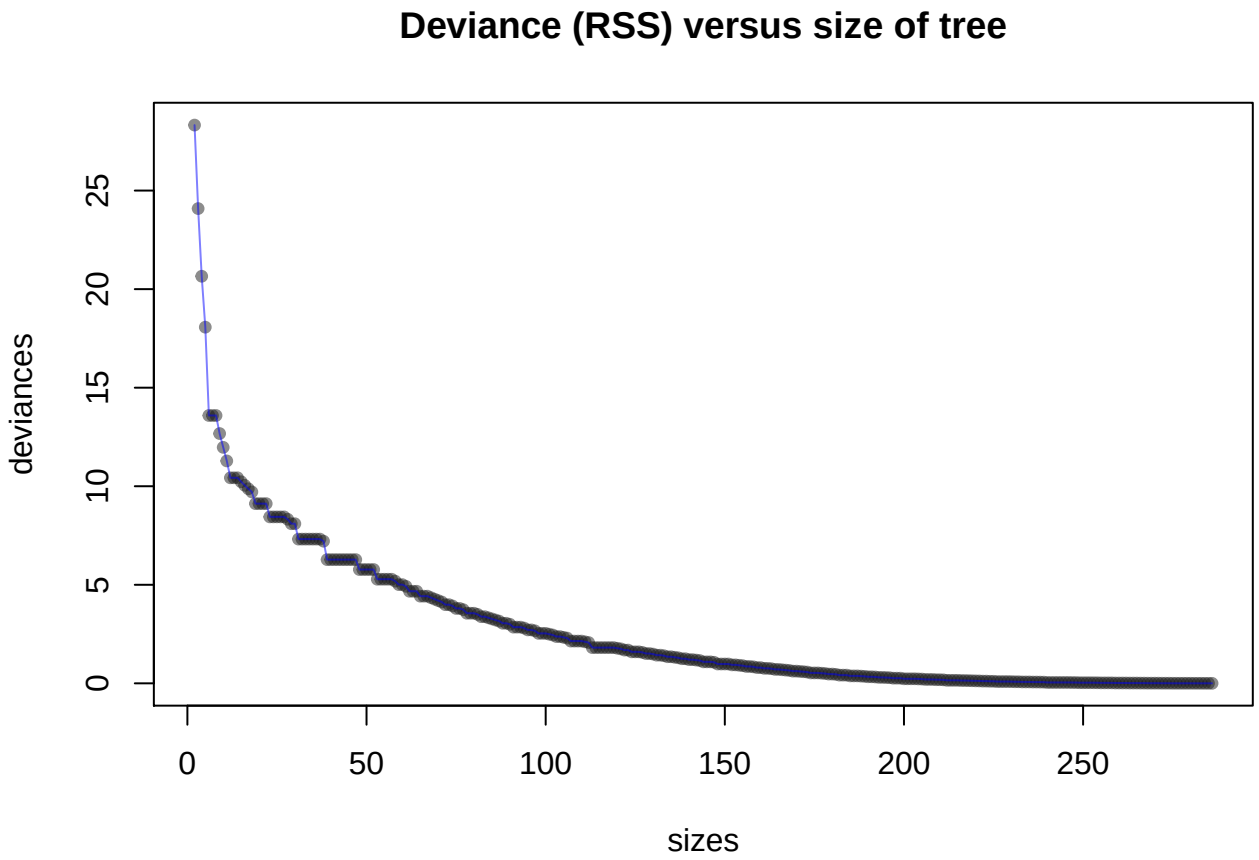


Note the qualitative differences between the smoothing spline and the regression tree fit. The tree is able to adapt to sharper changes in places (notably the middle) than is the smooth. Conversely, the smooth adapts to small (continuous and smooth) changes throughout the data whereas the tree is forced to have flat regions with abrupt (not differentiable) changes.

Some sense of how the fit depends on the size of the tree can be had by simply fitting the best tree for a range of tree sizes.

```
max_size <- sum(fake.tree.bushy$frame[,"var"] == "<leaf>")

sizes <- seq(2,max_size)
deviances <- sapply(sizes,
                    FUN = function(size) {
                      deviance(prune.tree(fake.tree.bushy,
                                          best=size)
```

```
                      )
                    }
                    )

plot(sizes, deviances,
     col=adjustcolor("grey10", 0.5), pch=19, cex=0.75,
     main = "Deviance (RSS) versus size of tree")
lines(sizes, deviances, col=adjustcolor("blue", 0.5))
```



As can be seen, the *RSS* decreases as the complexity of the tree (number of leaves) increases. As we did with smoothers, we might use the **average prediction error sum of squares** or **apse** together with **k-fold cross-validation** to compares sizes, and hopefully choose one.

The following functions are **identical** to those we had before. They are reproduced here only for convenience.

```
apse <- function(Ssamples, Tsamples, df){
  # average over the samples S
  #
  N_S <- length(Ssamples)
  mean(sapply(1:N_S,
              FUN=function(j){
                S_j <- Ssamples[[j]]
                # get the muhat function based on
                # the sample S_j
                muhat <- getmuhat(S_j, df=df)
                # average over (x_i,y_i) in a
                # single sample T_j the squares
```

```
                # (y - muhat(x))^2
                T_j <- Tsamples[[j]]
                ave_y_mu_sq(T_j,muhat)
              }
  )
  )
}

getmubar <- function(muhats) {
  # the muhats must be a list of muhat functions
  # We build and return mubar, the function that
  # is the average of the functions in muhats
  # Here is mubar:
  function(x) {
    # x here is a vector of x values on which the
    # average of the muhats is to be determined.
    #
    # sapply applies the function given by FUN
    # to each muhat in the list muhats
    Ans <- sapply(muhats, FUN=function(muhat){muhat(x)})
    # FUN calculates muhat(x) for every muhat and
    # returns the answer Ans as a matrix having
    # as many rows as there are values of x and
    # as many columns as there are muhats.
    # We now just need to get the average
    # across rows (first dimension)
    # to find mubar(x) and return it
    apply(Ans, MARGIN=1, FUN=mean)
  }
}


ave_y_mu_sq <- function(sample, predfun){
  mean(abs(sample$y - predfun(sample$x))^2)
}

ave_mu_mu_sq <- function(predfun1, predfun2, x){
  mean((predfun1(x) - predfun2(x))^2)
}


kfold <- function(N, k=N, indices=NULL){
  # get the parameters right:
  if (is.null(indices)) {
    # Randomize if the index order is not supplied
    indices <- sample(1:N, N, replace=FALSE)
  } else {
    # else if supplied, force N to match its length
    N <- length(indices)
  }
  # Check that the k value makes sense.
  if (k > N) stop("k must not exceed N")
  #
```

```
  # How big is each group?
  gsize <- rep(round(N/k), k)

  # For how many groups do we need odjust the size?
  extra <- N - sum(gsize)

  # Do we have too few in some groups?
  if (extra > 0) {
    for (i in 1:extra) {
      gsize[i] <- gsize[i] +1
    }
  }
  # Or do we have too many in some groups?
  if (extra < 0) {
    for (i in 1:abs(extra)) {
      gsize[i] <- gsize[i] - 1
    }
  }

  running_total <- c(0,cumsum(gsize))

  # Return the list of k groups of indices
  lapply(1:k,
         FUN=function(i) {
           indices[seq(from = 1 + running_total[i],
                       to = running_total[i+1],
                       by = 1)
                   ]
         }
  )
}

getKfoldSamples <- function (x, y, k, indices=NULL){
  groups <- kfold(length(x), k, indices)
  Ssamples <- lapply(groups,
                     FUN=function(group) {
                       list(x=x[-group], y=y[-group])
                     })
  Tsamples <- lapply(groups,
                     FUN=function(group) {
                       list(x=x[group], y=y[group])
                     })
  list(Ssamples = Ssamples, Tsamples = Tsamples)
}
```

What we do need to specialize is the function $\widehat{\mu}(x)$. This is now a tree function indexed by its size. Following the method used (so far) to construct the fitted tree, we can write a function that will return $\widehat{\mu}(x)$ for any sample and any size of the tree.

```
getmuhat <- function(sample, df) {
  y <- sample$y
  x <- sample$x
  size <- df
  fulltree <-  tree(y ~ x,
```

```
                    control=tree.control(nobs=length(x),
                                         mindev = 0,
                                         minsize= 2,
                                         mincut = 1
                                         )
                    )
  prunedtree <- prune.tree(fulltree, best=size)

  muhat <- function(x){predict(prunedtree,
                               newdata=list(x=x))
    }
  # return this function as the value of getmuhat
  muhat
}
```

Note that we are using the argument `df` as the **size** of the tree.

The above code is not the most efficient way to perform these calculations, especially since the full tree is calculated to get a `muhat` for *every* tree size.

Nevertheless, we can try it out for a few sizes as we did with smoothing splines to get some idea of how the $APSE$ behaves as a function of tree size.

We will use the following function to carry everything out. Again, it is essentially identical to the one we used for smoothing splines. The only differences are due to labelling the plot.

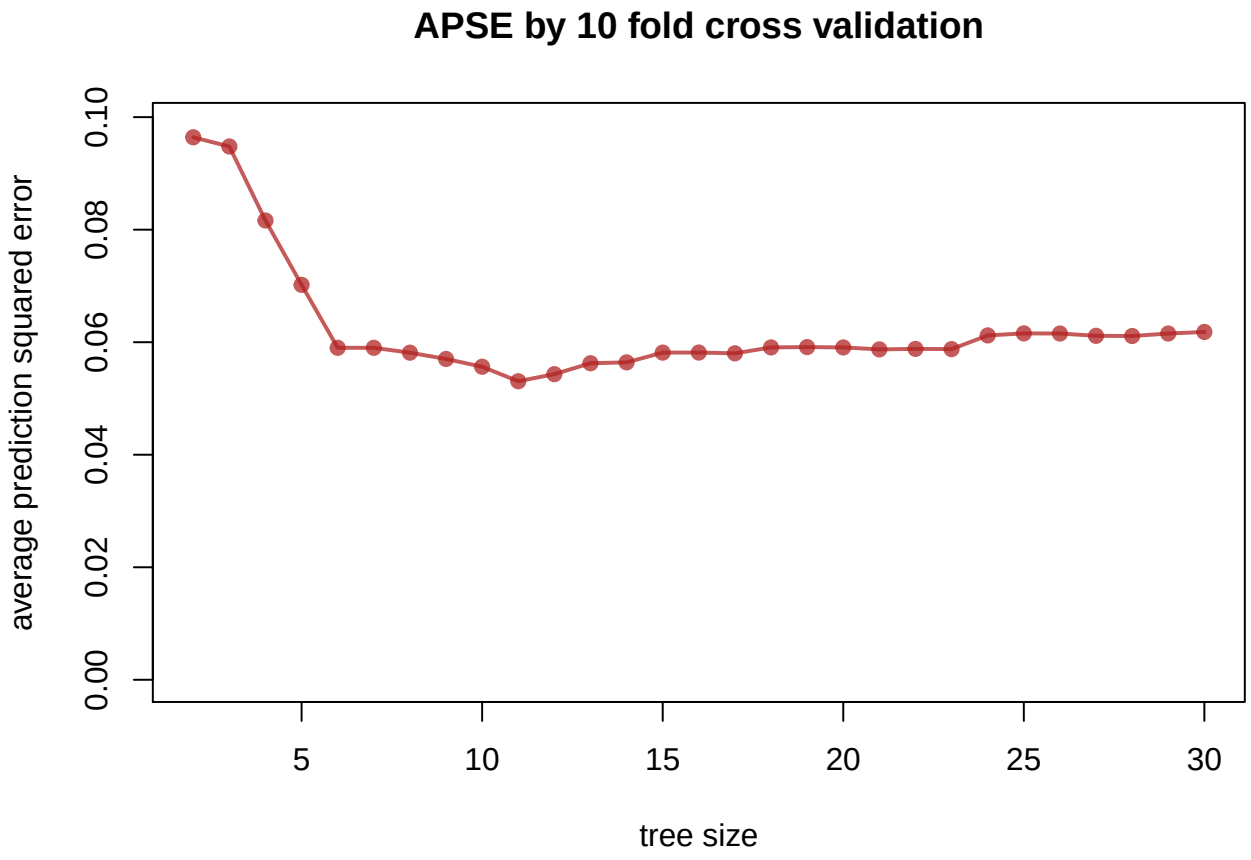
```
plot_apse <- function(x, y, df, k,
                      # plot parameters
                      xlab="df",
                      ylab="average prediction squared error",
                      main=NULL){

  if (is.null(main)) {
    main <- paste("APSE by", k, "fold cross validation")
  }

  samples <-  getKfoldSamples(x, y, k)
  apsehat <-  sapply(df,
                     FUN = function(df){
                       apse(samples$Ssamples,
                            samples$Tsamples,
                            df=df)
                     }
  )
  ylim <- extendrange(apsehat)
  ylim[1] <- 0
  plot(df, apsehat, main=main, ylab=ylab, xlab=xlab,
       ylim = ylim,
       pch=19, col= adjustcolor("firebrick", 0.75))
  lines(df, apsehat,
        col=adjustcolor("firebrick", 0.75), lwd=2)
  # return results
  apsehat
}
```

We are now ready to investigate the average prediction squared errors for this data as a function of tree sizes.

We will **not** do leave one out cross-validation because this would be too time-consuming.

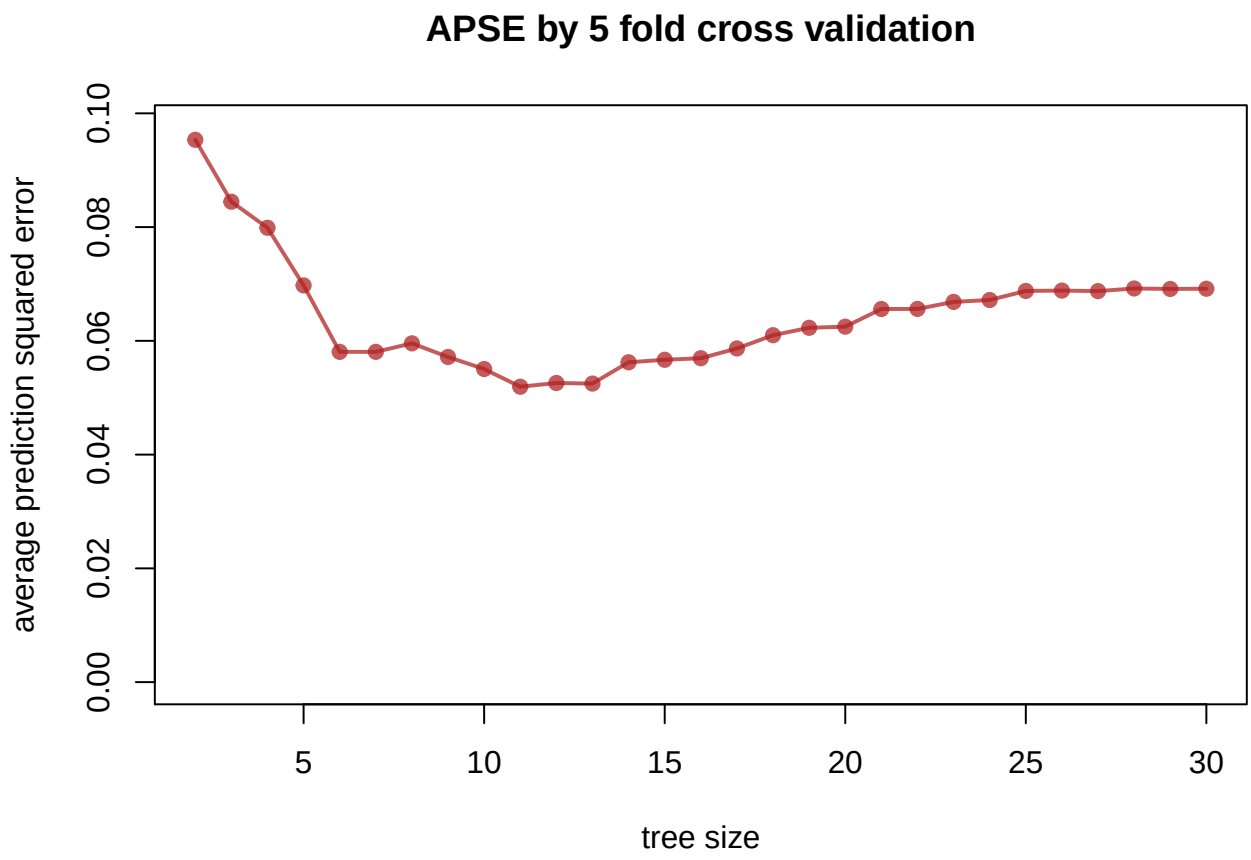
First the 10-fold cross-validation:

```
complexity <- 2:30

# 10 fold cross-validation
apsehat_10fold <- plot_apse(x, y, df=complexity, k=10,
                            xlab="tree size")
```

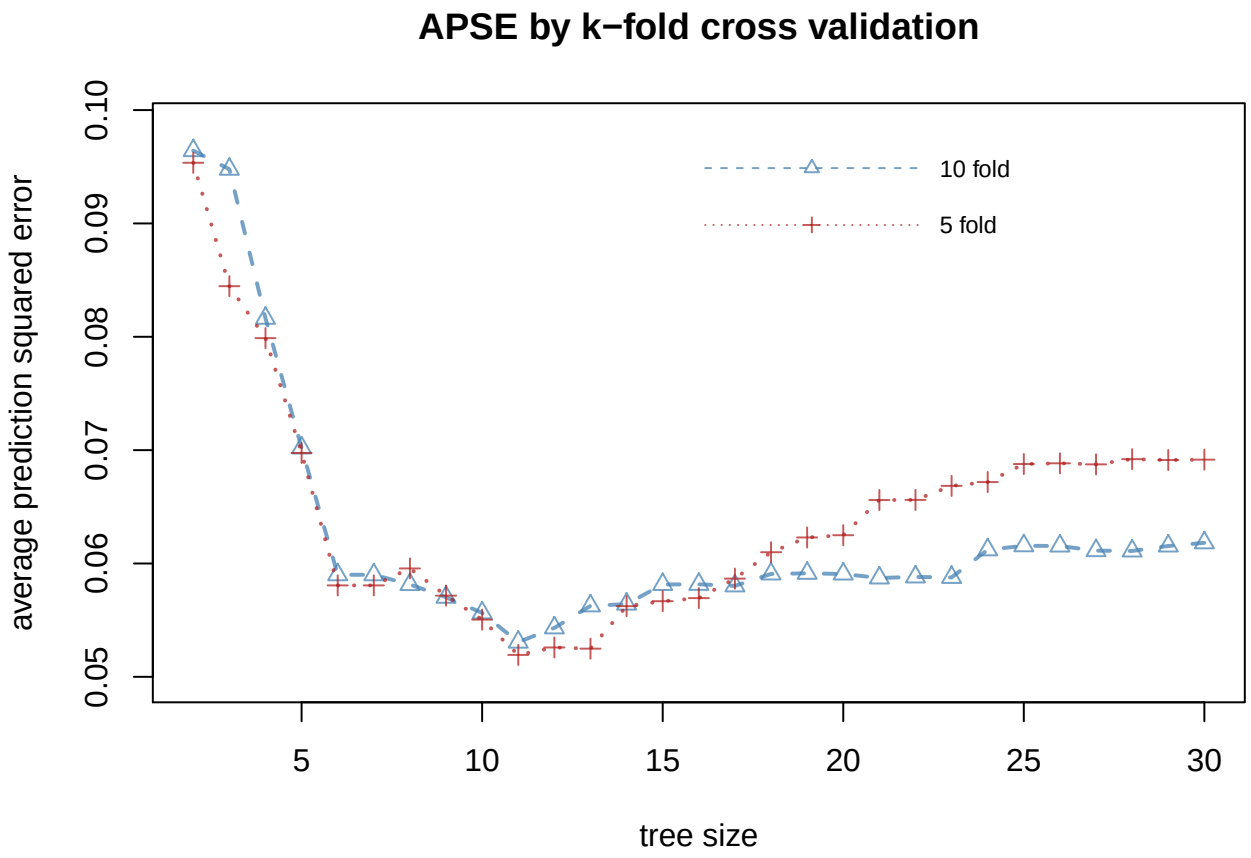


Then the 5-fold:

```
# 5 fold cross-validation
apsehat_5fold <-  plot_apse(x, y, df=complexity, k=5,
                            xlab="tree size")
```



and both together




```
##       complexity ten_fold five_fold
##  [1,]          2    0.096    0.0954
##  [2,]          3    0.095    0.0845
##  [3,]          4    0.082    0.0799
##  [4,]          5    0.070    0.0698
##  [5,]          6    0.059    0.0581
##  [6,]          7    0.059    0.0581
##  [7,]          8    0.058    0.0596
##  [8,]          9    0.057    0.0572
##  [9,]         10    0.056    0.0550
## [10,]         11    0.053    0.0519
## [11,]         12    0.054    0.0526
## [12,]         13    0.056    0.0525
## [13,]         14    0.056    0.0562
## [14,]         15    0.058    0.0567
## [15,]         16    0.058    0.0569
## [16,]         17    0.058    0.0587
## [17,]         18    0.059    0.0610
## [18,]         19    0.059    0.0623
## [19,]         20    0.059    0.0625
## [20,]         21    0.059    0.0656
## [21,]         22    0.059    0.0656
## [22,]         23    0.059    0.0668
## [23,]         24    0.061    0.0672
## [24,]         25    0.062    0.0688
## [25,]         26    0.062    0.0688
## [26,]         27    0.061    0.0687
```

```
## [27,]         28    0.061    0.0692
## [28,]         29    0.062    0.0691
## [29,]         30    0.062    0.0692
```

Both five fold and ten fold cross validation suggests a tree size of 11. These choices seem to be in agreement with the choice of degrees of freedom for the smoothing spline.

1.3.2 Cost-complexity pruning

As suggested earlier, we might incorporate a penalty for complexity into the function we are minimizing to fit the tree. The penalized RSS (or deviance) for a tree T is written as

$$\sum_{i=1}^{N} \left(y_i - \sum_{k=1}^{K} I_{R_k}(x_i)\ \mu_k \right)^2 \quad + \quad \lambda\ K$$

where $K = |T|$ is the number of regions/terminal nodes/leaves in the tree T and λ is a fixed constant. Clearly, the smaller the value of λ, the greater is the complexity of the tree. The value of λ effectively determines the size of the tree.

As was done with smoothing splines, we can think of this problem as maximizing the posterior distribution of a tree T, where the responses (conditional on x) come from the generative model

$$Y_i = \mu_T(x_i) + R_i$$

conditional on x_i and the tree T and

$$R_i \sim N(0, \sigma^2)$$

independently. Suppose each tree T's marginal probability is proportional to

$$exp\{-\lambda\ |T|\}$$

for a fixed constant λ. That is, more complex trees have lower probability of being served up by nature than do simple trees so that increasing λ reduces the probability of a complex tree.

In this framework, we could apply Bayes's theorem and find the conditional probability of any particular tree given the data. This is often called the posterior probability of the tree. Choosing the tree which *maximizes* this posterior probability is equivalent to *minimizing* the penalized RSS (or, more generally, the penalized likelihood) which we could write as

$$\begin{aligned} & \sum_{i=1}^{N} \left(y_i - \sum_{k=1}^{|T|} I_{R_k}(x_i)\ \mu_k \right)^2 \quad + \quad \lambda\ |T| \\ = & \sum_{i=1}^{N} \left(y_i - \mu_T(x_i) \right)^2 \quad + \quad \lambda\ |T|. \end{aligned}$$

However you want to think about it (minimizing a penalized RSS, maximizing a penalized likelihood, or finding the posterior mode), the same quantity is being minimized for fixed λ. This function combines **cost** (quality of fit) and **complexity** and we find a tree which minimizes the combination by pruning our largest bushiest tree. This is called **cost-complexity pruning**.

In the `tree` package, the `prune.tree(...)` function can be used to effect cost-complexity pruning. Beginning with the full tree, we prune based on λ. When $\lambda = 0$, the original tree is returned and as λ increases, the returned tree will be less complex.

For example, suppose $\lambda = 0.01$ so that complexity is not being penalized that greatly

```
##
## Regression tree:
## snip.tree(tree = fake.tree.bushy, nodes = c(7535L, 19456L, 43L,
## 897L, 1904L, 1905L, 323L, 31L, 2543L, 471L, 1217L, 82L, 45L,
## 19065L, 77L, 1832L, 8953L, 287L, 2540L, 462L, 2438L, 2443L, 225L,
## 494L, 1796L, 4477L, 454L, 8952L, 557L, 632L, 94L, 152513L, 917L,
## 1830L, 89L, 3766L, 1113L, 1116L, 1193L, 42L, 611L, 116L, 2380L,
## 3824L, 152515L, 75L, 921L, 13L, 7246L, 1810L, 461L, 142L, 1841L,
## 12L, 2381L, 4478L, 1918L))
## Number of terminal nodes:  194
## Residual mean deviance:  0.002853 = 0.3024 / 106
## Distribution of residuals:
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
## -0.113800 -0.005506  0.000000  0.000000  0.009939  0.094360
```

we still have a very bushy tree.

As we increase $\lambda = 0.1$,

```
##
## Regression tree:
## snip.tree(tree = fake.tree.bushy, nodes = c(31L, 45L, 77L, 225L,
## 1796L, 454L, 116L, 75L, 13L, 12L, 153L, 959L, 448L, 235L, 247L,
## 1798L, 120L, 71L, 149L, 1799L, 122L, 44L, 78L, 226L, 297L, 238L,
## 592L, 47L, 478L, 68L, 492L, 139L, 10L, 115L, 152L, 79L, 114L))
## Number of terminal nodes:  52
## Residual mean deviance:  0.02328 = 5.774 / 248
## Distribution of residuals:
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
## -0.50200 -0.09831  0.00000  0.00000  0.08393  0.32260
```

we have fewer terminal nodes/leaves in the tree.

When $\lambda = 1$,

```
##
## Regression tree:
## snip.tree(tree = fake.tree.bushy, nodes = c(112L, 15L, 29L, 113L,
## 6L, 57L, 5L, 4L))
## Number of terminal nodes:  8
## Residual mean deviance:  0.04654 = 13.59 / 292
## Distribution of residuals:
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
## -0.62980 -0.15270  0.01011  0.00000  0.12620  0.63350
```

we have only 8 leaves.

As with size, we could use $APSE$ and cross-validation to choose λ.

NOTE This requires very little change in our code. We need only specify $\widehat{\mu}$ to be a function of the complexity constant λ rather than of the tree size $|T|$:

```
# Only 2 changes are needed
getmuhat <- function(sample, df) {
  y <- sample$y
  x <- sample$x
  ## First change is here
  lambda <- df
  fulltree <-  tree(y ~ x,
                    control=tree.control(nobs=length(x),
```

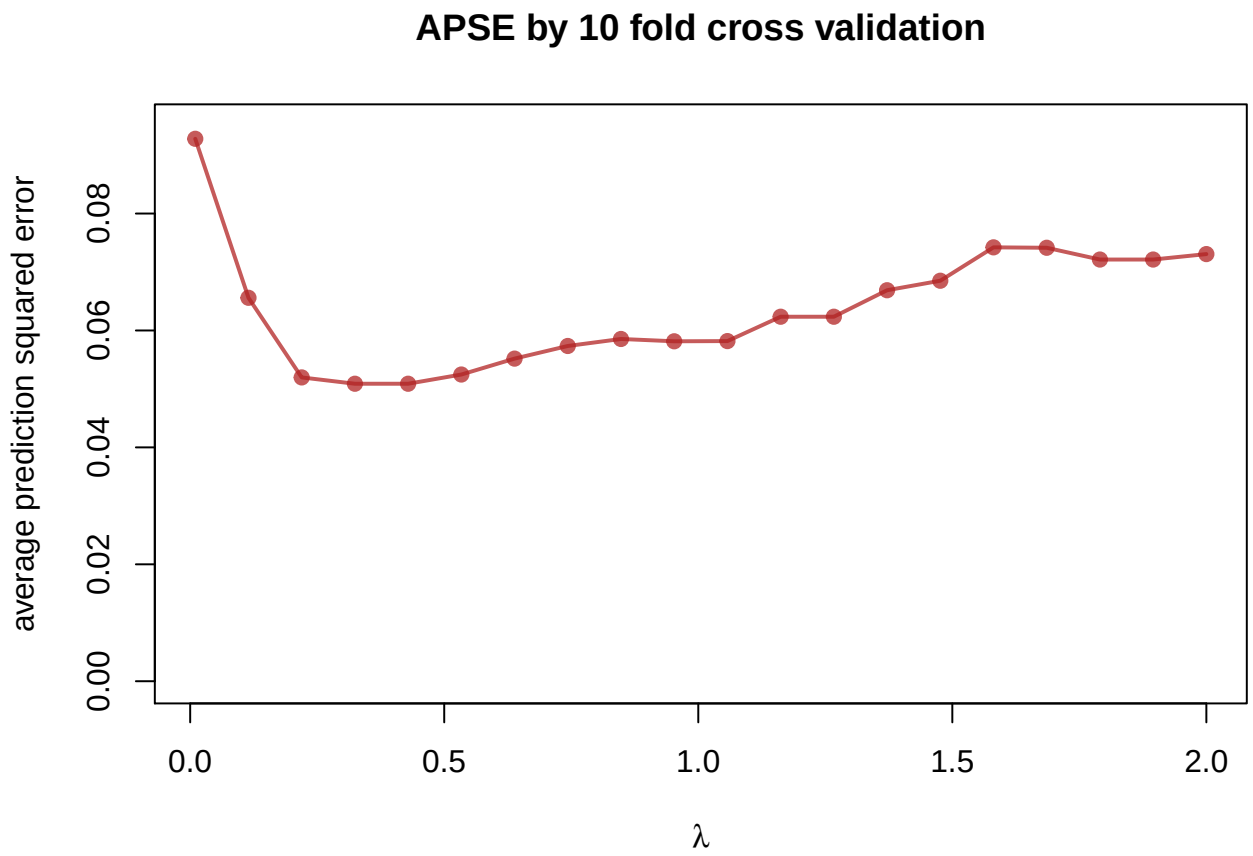
```
                                        mindev = 0,
                                        minsize= 2,
                                        mincut = 1
                                        )
                    )
  ## And second here
  prunedtree <- prune.tree(fulltree, k=lambda)

  muhat <- function(x){predict(prunedtree,
                               newdata=list(x=x))
    }
  # return this function as the value of getmuhat
  muhat
}
```

The cross-validations are now trivially written. Since complexity is now λ, we need to have it take small values:

```
complexity <- seq(0.01, 2, length.out = 20)
```

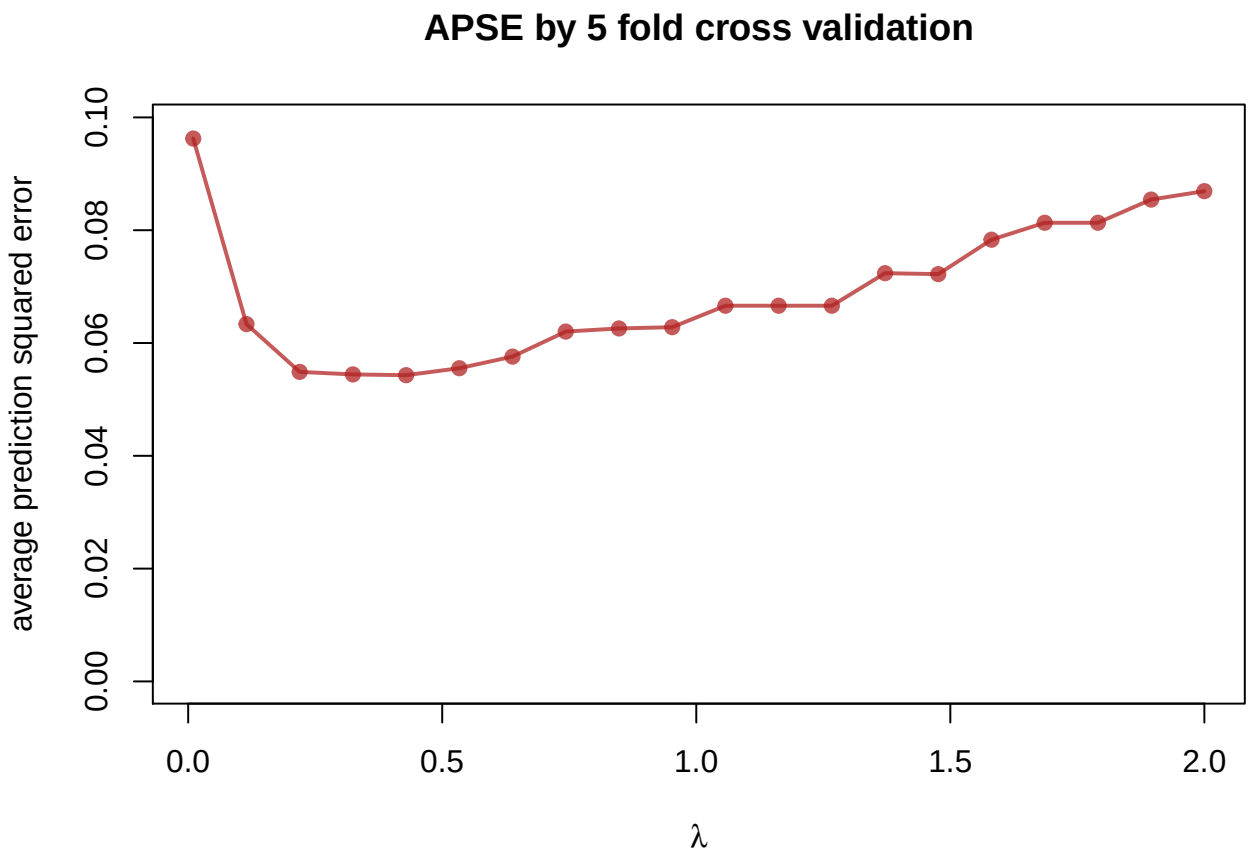
The 10-fold cross-validation:

```
# 10 fold cross-validation
apsehat_10fold <- plot_apse(x, y, df=complexity, k=10,
                            xlab=expression(lambda))
```

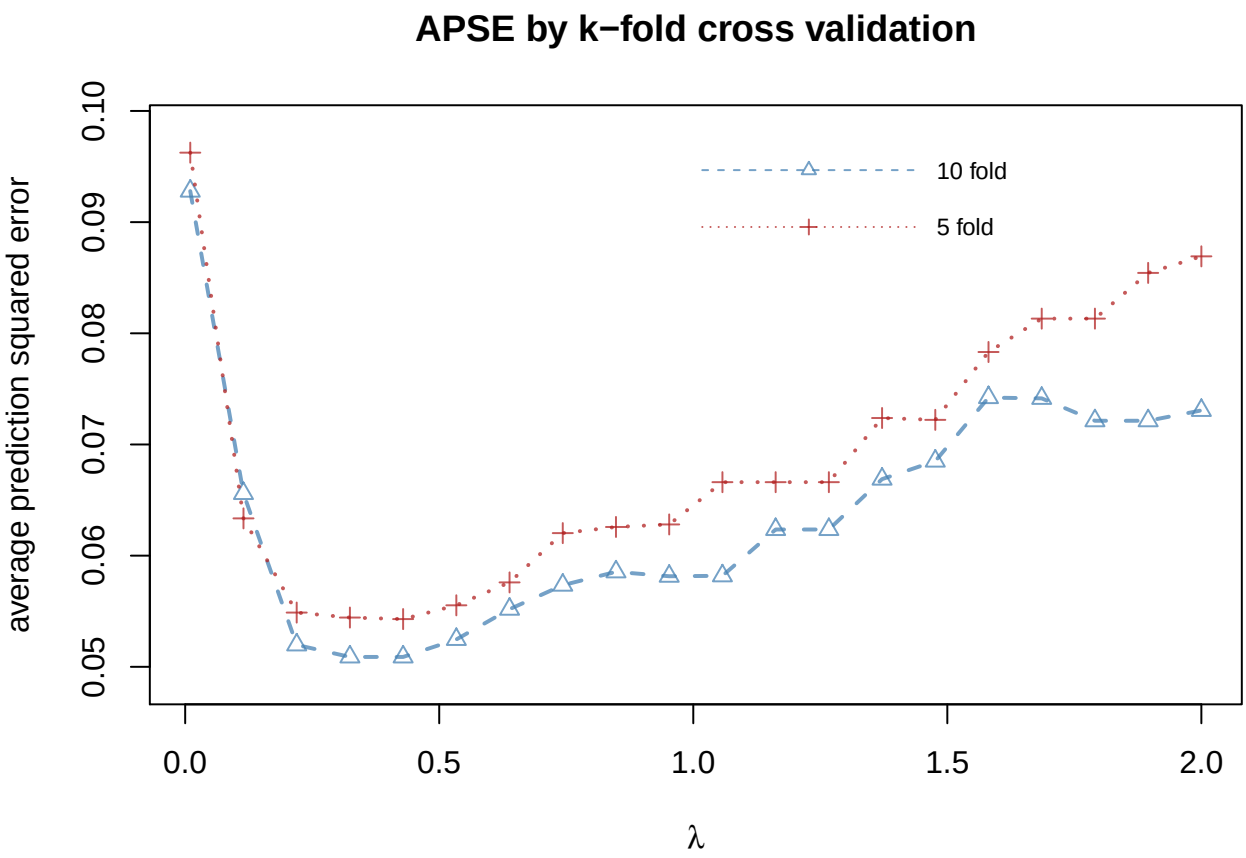


Then the 5-fold:

```r
# 5 fold cross-validation
apsehat_5fold <-  plot_apse(x, y, df=complexity, k=5,
                            xlab=expression(lambda))
```



and both together

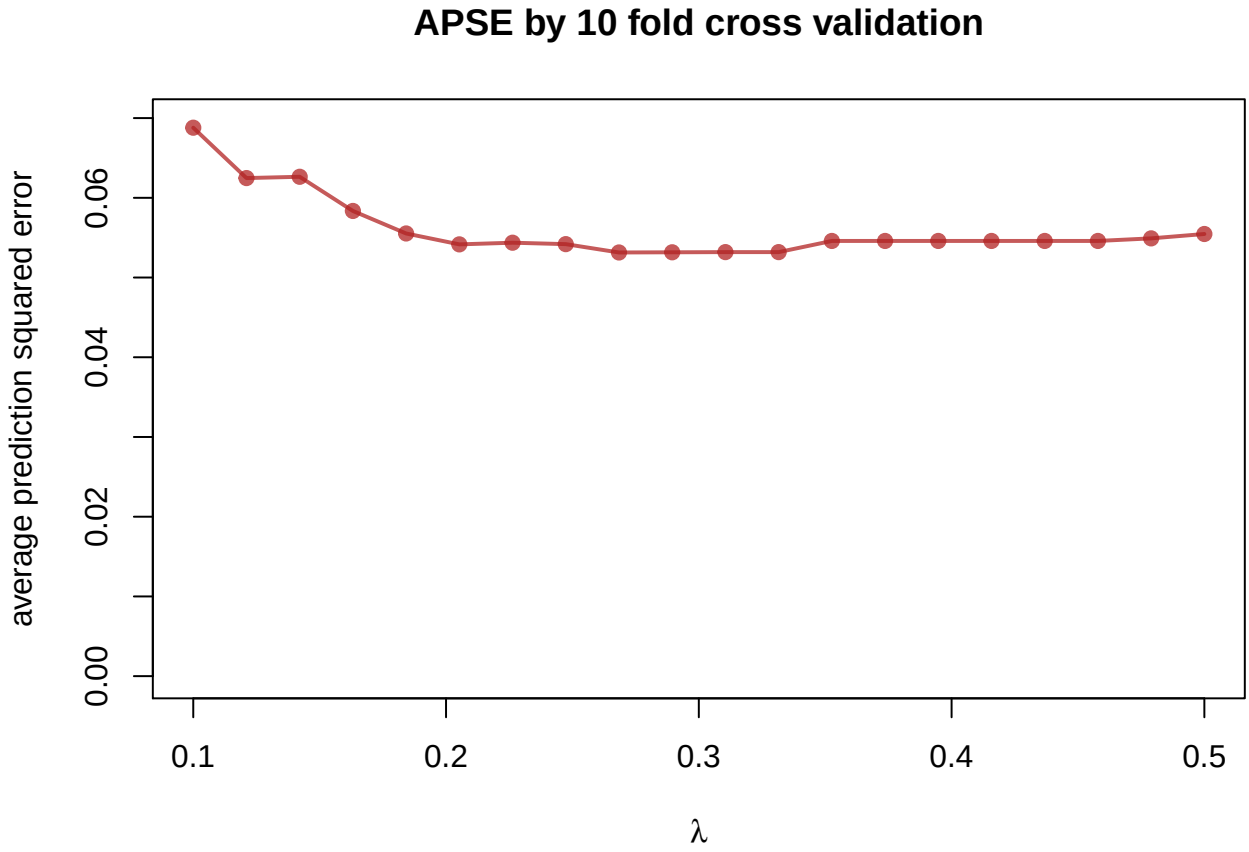


This suggests a λ value somewhat less than 0.5. We could focus on the narrower region

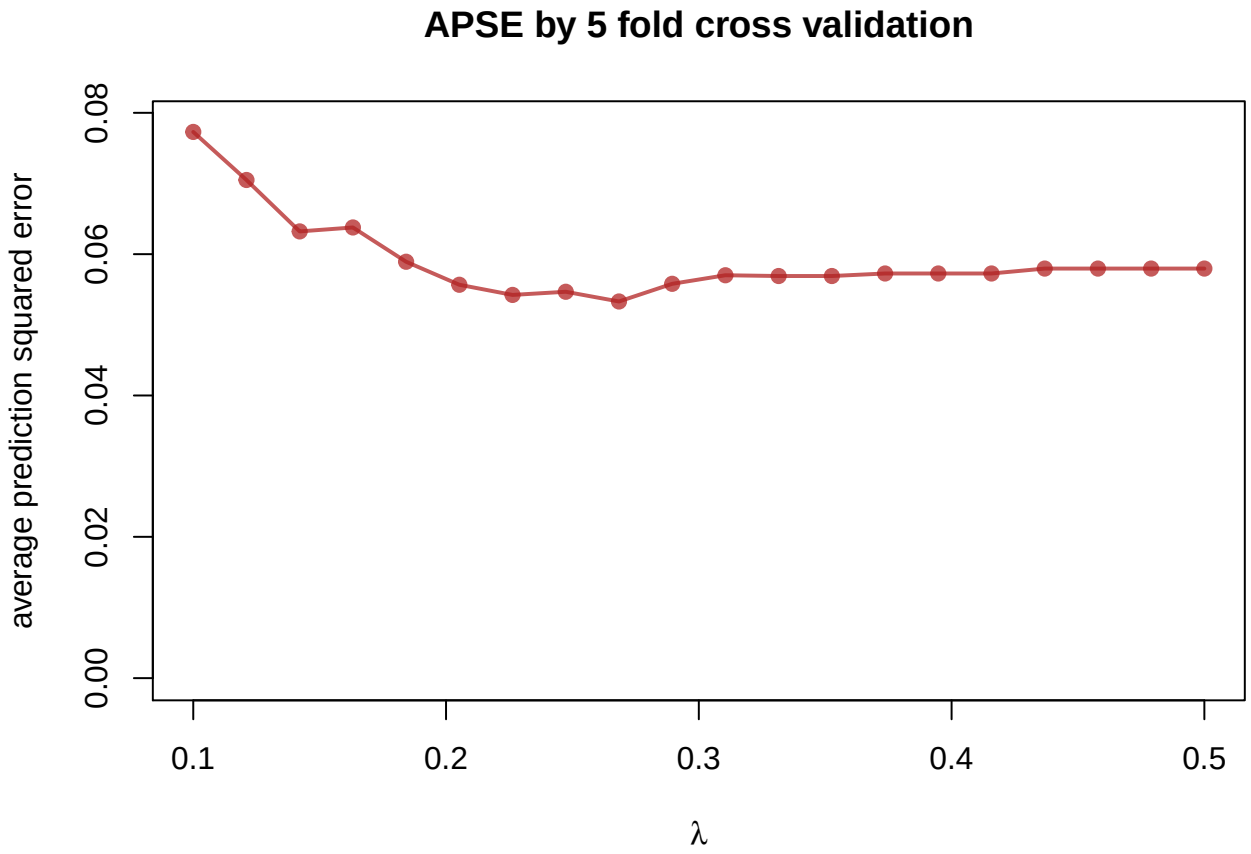
```
complexity <- seq(0.1, 0.5, length.out = 20)
```

The results are

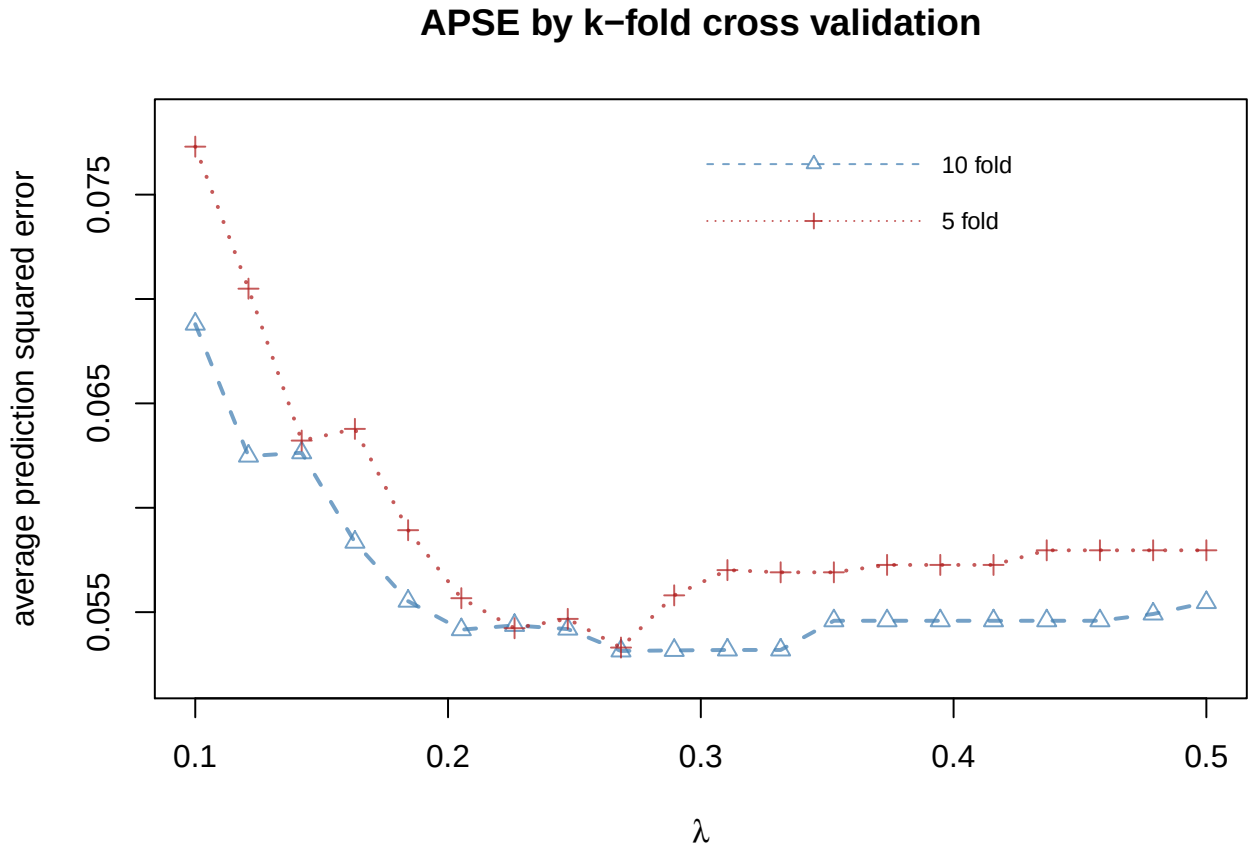
```
# 10 fold cross-validation
apsehat_10fold <- plot_apse(x, y, df=complexity, k=10,
                            xlab=expression(lambda))
```




```
# 5 fold cross-validation
apsehat_5fold <-  plot_apse(x, y, df=complexity, k=5,
                            xlab=expression(lambda))
```




```
# And plot them together
ylim <- extendrange(c(apsehat_10fold, apsehat_5fold))
# Plot the results
plot(complexity, apsehat_10fold, ylim=ylim,
     main="APSE by k-fold cross validation",
     xlab=expression(lambda),
     ylab="average prediction squared error",
     pch=2, col= adjustcolor("steelblue", 0.75))

lines(complexity, apsehat_10fold,
      col=adjustcolor("steelblue", 0.75), lwd=2, lty=2)

lines(complexity, apsehat_5fold,
      col=adjustcolor("firebrick", 0.75), lwd=2, lty=3)
points(complexity, apsehat_5fold,
       col=adjustcolor("firebrick", 0.75), pch=3)

legend(median(complexity), max(ylim),
       legend=c("10 fold", "5 fold"),
       lty =2:3, pch=c(2,3),
       col=adjustcolor(c("steelblue","firebrick"), 0.75),
       cex=0.75, y.intersp=2, bty="n", seg.len=10)
```



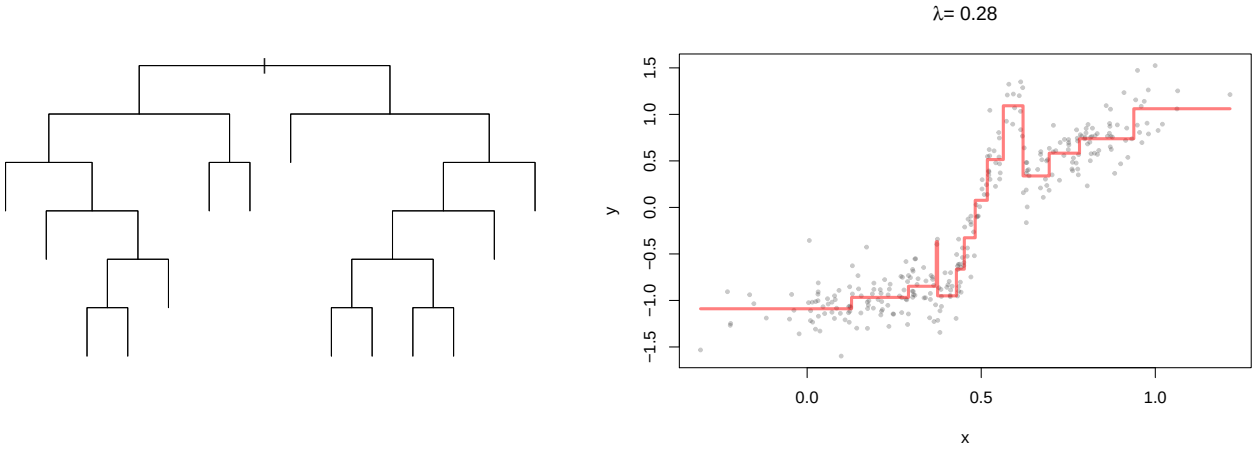

```
t(rbind(complexity,
        ten_fold=round(apsehat_10fold,5),
        five_fold=round(apsehat_5fold,5)))
```

```
##       complexity ten_fold five_fold
##  [1,]  0.1000000  0.06880   0.07730
##  [2,]  0.1210526  0.06248   0.07049
##  [3,]  0.1421053  0.06264   0.06322
##  [4,]  0.1631579  0.05835   0.06378
##  [5,]  0.1842105  0.05553   0.05893
##  [6,]  0.2052632  0.05416   0.05567
##  [7,]  0.2263158  0.05437   0.05423
##  [8,]  0.2473684  0.05419   0.05468
##  [9,]  0.2684211  0.05314   0.05330
## [10,]  0.2894737  0.05317   0.05580
## [11,]  0.3105263  0.05319   0.05701
## [12,]  0.3315789  0.05319   0.05691
## [13,]  0.3526316  0.05459   0.05691
## [14,]  0.3736842  0.05459   0.05726
## [15,]  0.3947368  0.05459   0.05726
## [16,]  0.4157895  0.05459   0.05726
## [17,]  0.4368421  0.05459   0.05796
## [18,]  0.4578947  0.05459   0.05796
## [19,]  0.4789474  0.05491   0.05796
## [20,]  0.5000000  0.05546   0.05796
```

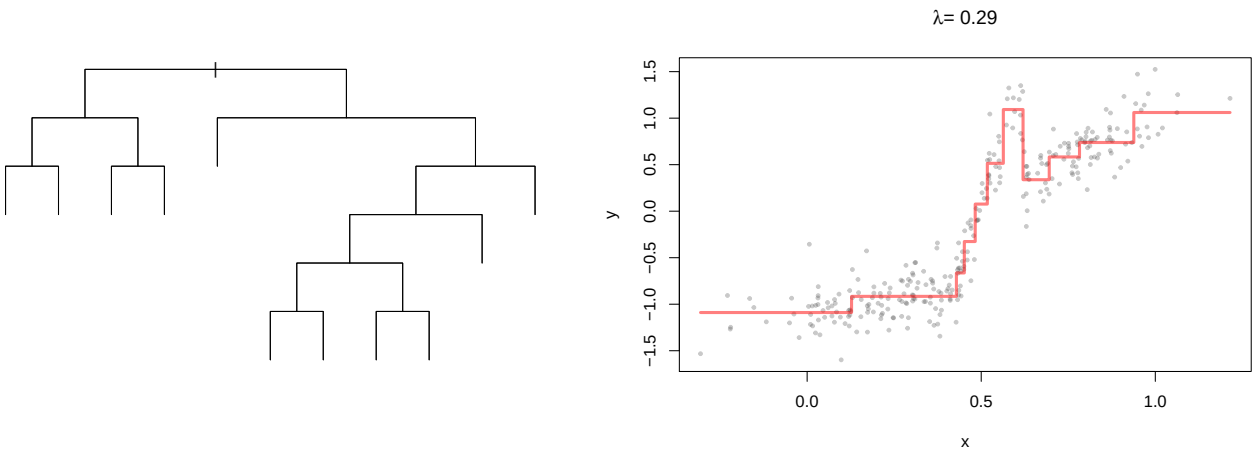
There are several ties where `prune.tree` produces the same tree. A value of $\widehat{\lambda} \approx 0.28$ or 0.29 (look around rows 9-12) seems appropriate. (Note that $\widehat{\lambda} = 0.28$ actually looks different!)

The resulting trees are as follows:

```
savePar <- par(mfrow=c(1,2))
lambda <- 0.28
fake.tree.lambda <- prune.tree(fake.tree.bushy, k=lambda)
plot(fake.tree.lambda, type="uniform")
plot(x,y,
     col=adjustcolor("grey30", 0.3), pch=19, cex=0.5,
     main = expression(paste(lambda,"= 0.28")))
partition.tree(fake.tree.lambda, add=TRUE,
               col=adjustcolor("red", 0.5), lwd=3)
```




```
lambda <- 0.29
fake.tree.lambda <- prune.tree(fake.tree.bushy, k=lambda)
plot(fake.tree.lambda, type="uniform")
plot(x,y,
     col=adjustcolor("grey30", 0.3), pch=19, cex=0.5,
     main = expression(paste(lambda,"= 0.29")))
partition.tree(fake.tree.lambda, add=TRUE,
               col=adjustcolor("red", 0.5), lwd=3)
```




```
par(savePar)
```

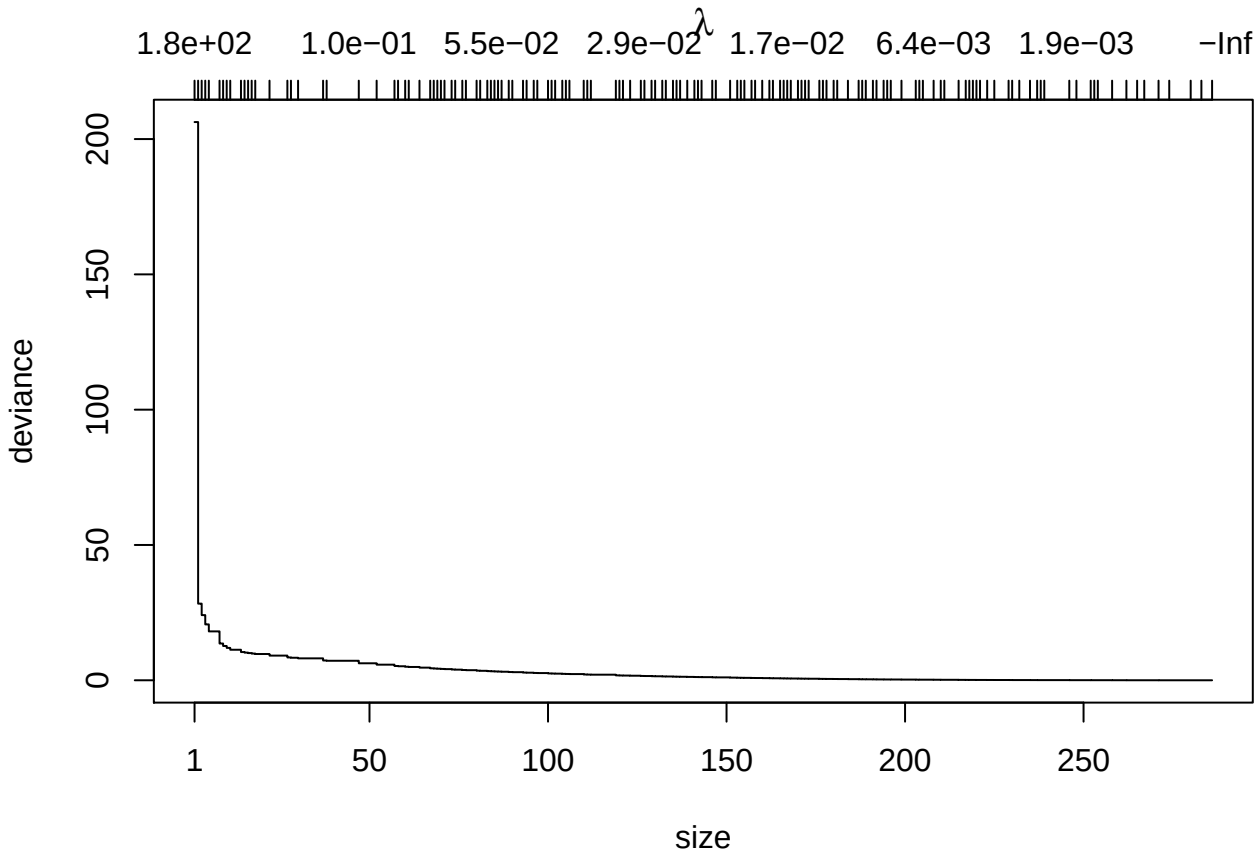
which is essentially the same one we had before based on size. This is because choosing the best λ is equivalent to choosing the best tree size in the sequence of trees. The `prune.tree(...)` function by default

produces the entire sequence of trees, from which we select.

```
fake.tree.sequence <- prune.tree(fake.tree.bushy)
names(fake.tree.sequence)
```

```
## [1] "size"   "dev"    "k"      "method"
```

```
plot(fake.tree.sequence, main=expression(lambda))
```



The values of λ appear on a separate axis along the top.

1.3.3 Computational efficiency and generality

In the above, the very generic functions developed for smoothing splines could be repurposed to use with regression trees with little to no change to the code. There is an enormous advantage to having such flexible and general code. It can be used in, or easily adapted to, new situations.

However, this comes at a price. By being so generic, little advantage can be taken of the structure of any particular estimator. The `tree` package does the opposite. It provides functions for testing only tree predictions, but at the cost of some flexibility.

The functions can be grouped as either - Training/test under user control. Here the functions are - `tree(...)` with argument `subset` to specify which data points are in the training set, and - `prune.tree(..)` with argument `newdata` to specify the data points on which the deviances are calculated at every node.
- K-fold cross validation. The function to be used here is - `cv.tree(...)` which performs a k-fold cross-validation **but** assumes that the tree being used was produced by `tree(...)` with default values for the `tree.control` parameters (i.e. not excessively bushy, as the ones we created earlier).

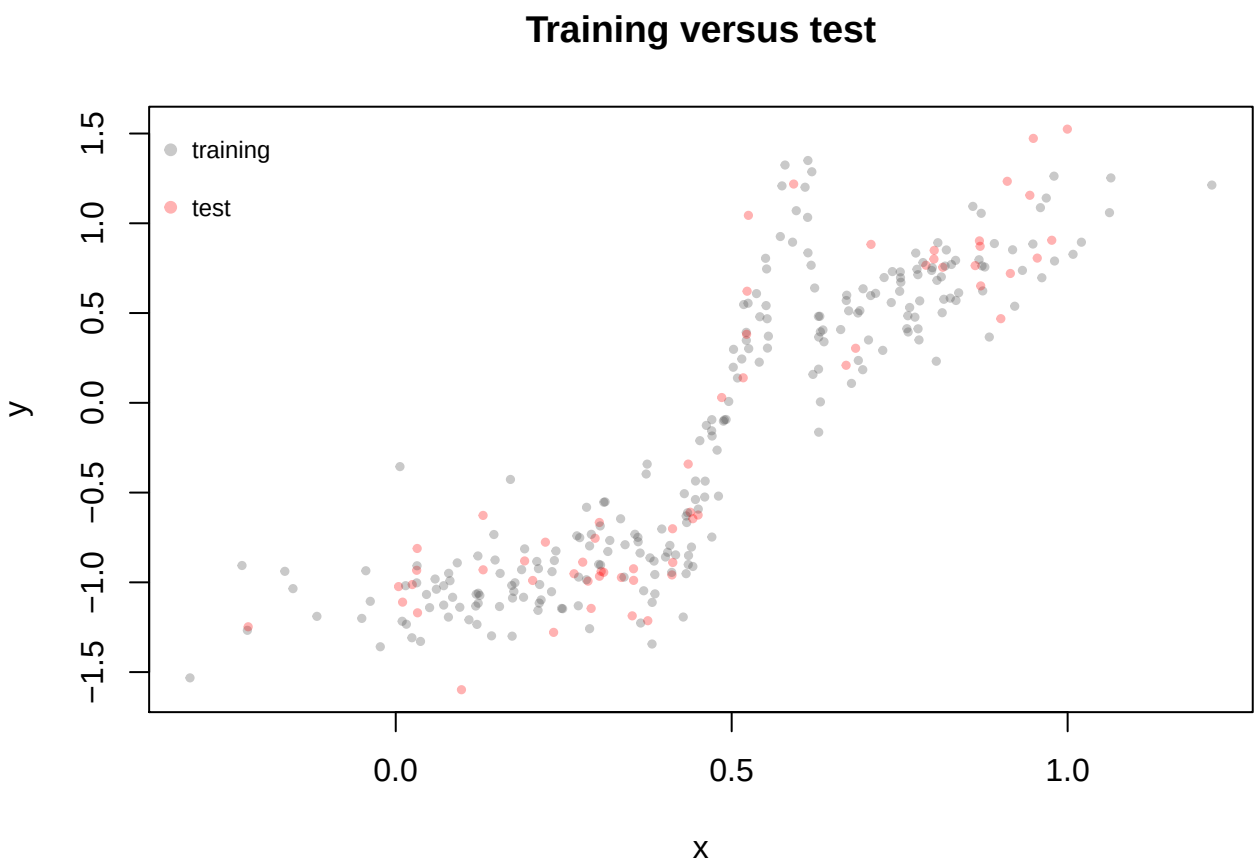
To illustrate these, let's consider the more general case (which could be used to implement any sampling or training/test division, includinf random k-fold cross-validation).

First define a training set and a test set (here at random and with no intersection).

```
n <- length(x)
set.seed(12345423) # reproduceability
training <- sample(1:n, round(0.80 * n), replace = FALSE)
test <- setdiff(1:n, training)
training <- sort(training)
```

We can see which data appear in the training and which in the test by colouring them:

```
# The data
col.train <- adjustcolor("grey30", 0.3)
col.test <- adjustcolor("red", 0.3)
col <- rep(col.train, length(x))
col[test] <- col.test
plot(x,y,
     col=col, pch=19, cex=0.5,
     main = "Training versus test")

legend("topleft",
       legend=c("training", "test"),  pch=c(19,19),
       col=c(col.train, col.test),
       cex=0.75, y.intersp=2, bty="n", seg.len=10)
```



Now build a tree based on the training set, with the parameters of your choice. Here another very bushy tree

is chosen. (N.B. don't usually start with this bushy a tree in practice; `mincut=5, minsize=10` is common and is the default for `tree(...)`.)

```
fake.tree.train <- tree(y ~ x,
                        subset = training,
                        control = tree.control(nobs = length(training),
                                               mindev=0,
                                               mincut=2,
                                               minsize=4
                                               )
                        )

summary(fake.tree.train)
```

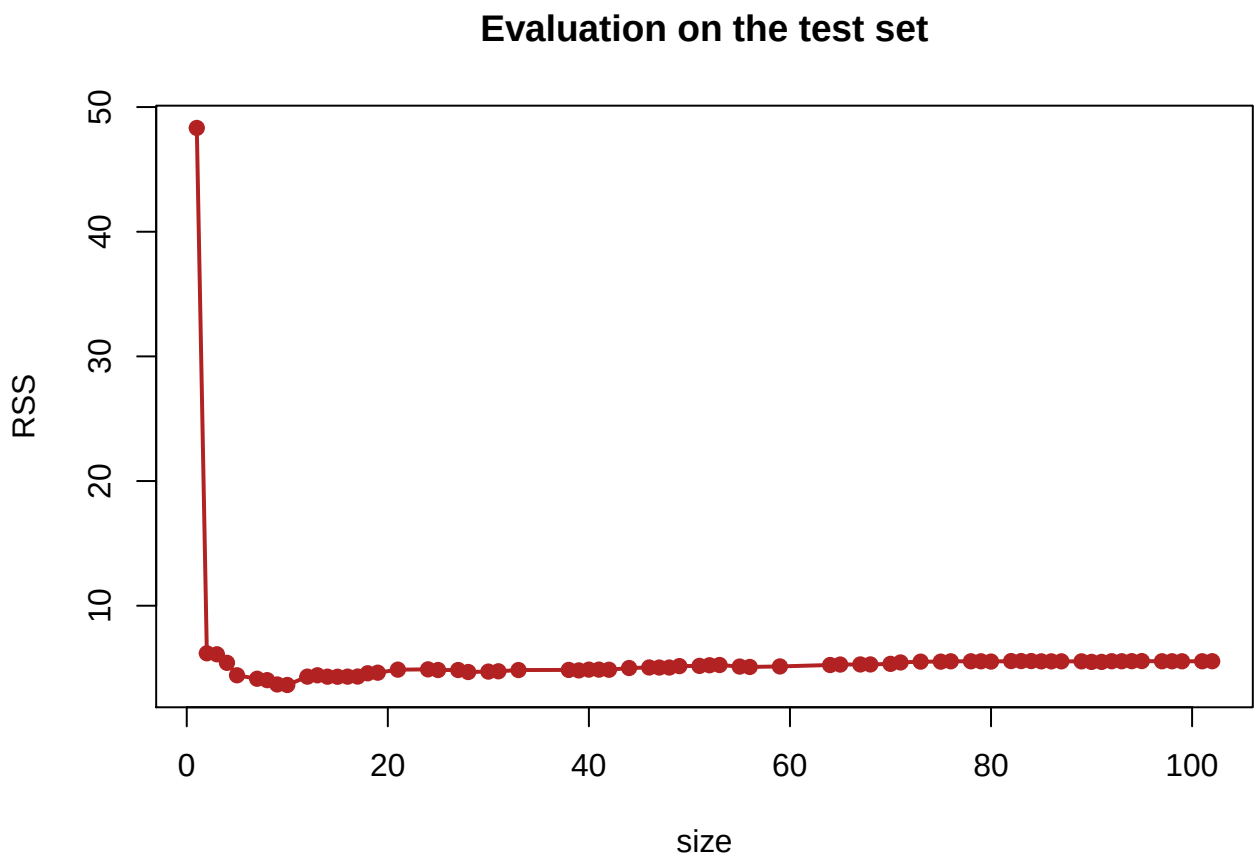
```
##
## Regression tree:
## tree(formula = y ~ x, subset = training, control = tree.control(nobs = length(training),
##     mindev = 0, mincut = 2, minsize = 4))
## Number of terminal nodes:  102
## Residual mean deviance:  0.02648 = 3.655 / 138
## Distribution of residuals:
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
## -0.501700 -0.059390  0.000966  0.000000  0.069650  0.501700
```

As before, `prune.tree` can be used to select a single tree, or to examine a sequence of trees, except now we choose to evaluate the deviance (RSS) at each node of the tree using the test set (previously we used the training set).

```
fake.tree.test.sequence <- prune.tree(fake.tree.train,
                                      # But on the test data
                                      newdata = data.frame(x=x[test], y=y[test])
                                       )
```

This provides the full sequence whose in formation we might plot to choose a tree size (or equivalently, a λ).

```
plot(fake.tree.test.sequence$size, fake.tree.test.sequence$dev,
     col="firebrick", pch=19, main="Evaluation on the test set",
     xlab="size", ylab="RSS")
lines(fake.tree.test.sequence$size, fake.tree.test.sequence$dev,
     col="firebrick", lwd=2)
```




```
# And the actual values
sel <- fake.tree.test.sequence$size > 5 & fake.tree.test.sequence$size < 15
cbind(size=fake.tree.test.sequence$size[sel], dev = fake.tree.test.sequence$dev[sel])
```

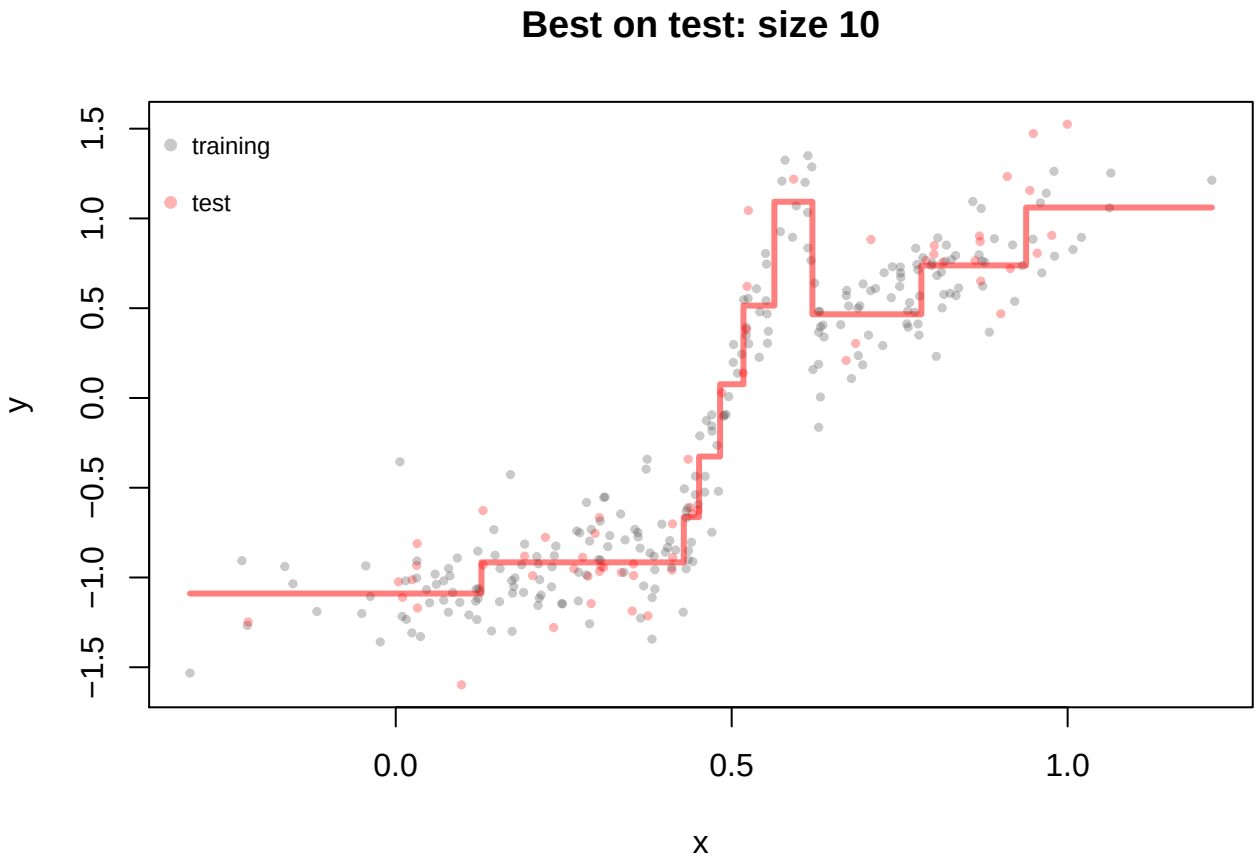
```
##      size      dev
## [1,]   14 4.306023
## [2,]   13 4.417758
## [3,]   12 4.310312
## [4,]   10 3.639601
## [5,]    9 3.682841
## [6,]    8 4.027870
## [7,]    7 4.141322
```

This suggests a tree size of 10. The analogous plots/comparisons could have been made to find a value for λ (k from `prune.tree(...)`).

Having selected a size, we would then construct the best tree of that size based on all of the data.

```
fake.tree.train.test <- prune.tree(fake.tree.bushy, # The full data tree
                                   best=10 # the suggested size
                                   )
plot(x,y,
     col=col, pch=19, cex=0.5,
     main = "Best on test: size 10")

legend("topleft",
       legend=c("training", "test"),  pch=c(19,19),
       col=c(col.train, col.test),
       cex=0.75, y.intersp=2, bty="n", seg.len=10)
```

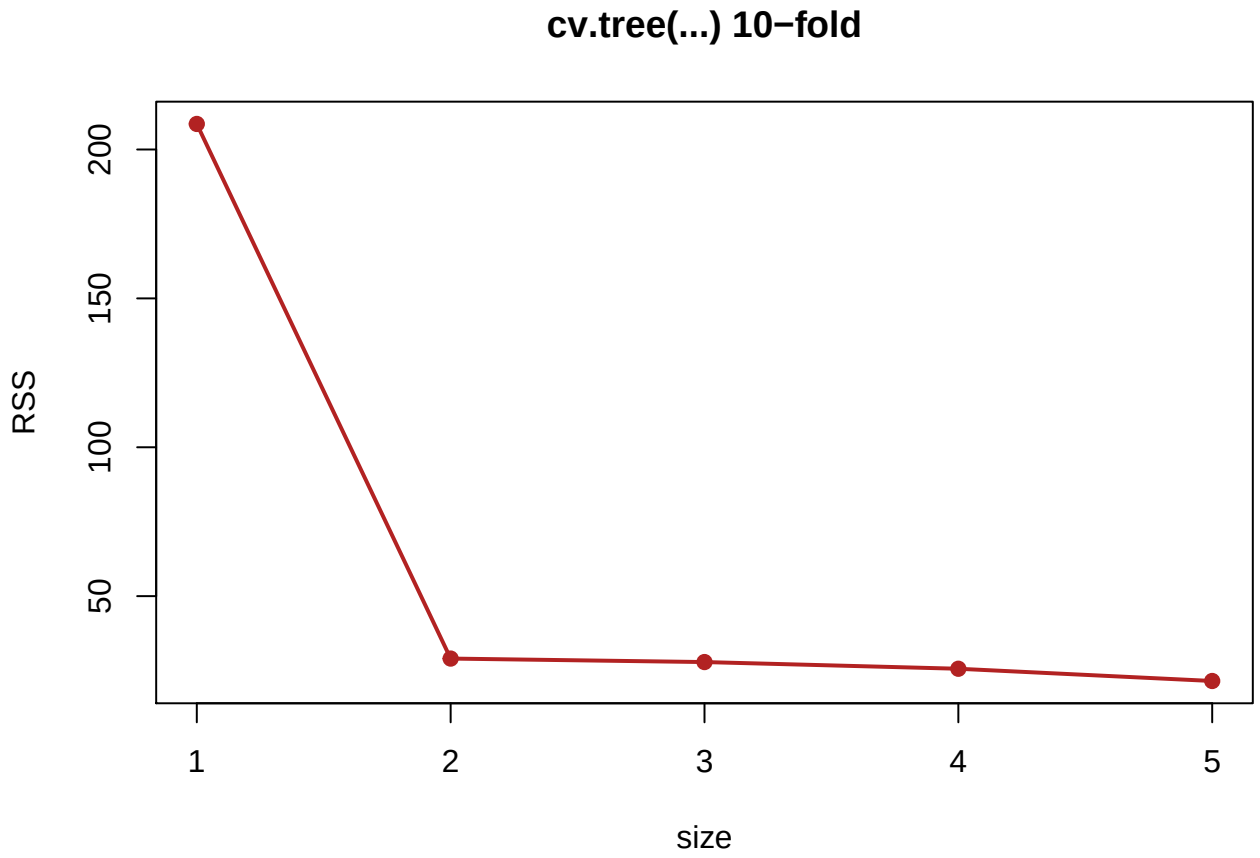
```
partition.tree(fake.tree.train.test, add=TRUE,
               col=adjustcolor("red", 0.5), lwd=3)
```



We could of course use the above functions to build our own k-fold cross-validation procedure just for trees. This would at least be flexible in how we choose a the `tree.control` parameters. It could also be more efficient computationally than the generic approach used earlier.

The `tree` package provides its own k-fold cross-validation function called `cv.tree(...)`. This is easy to use and is fast.

```
set.seed(2345235)
# Begin with a default tree (i.e. not our bushy one).
fake.tree.default <- tree(y~x)
# Now get the deviances using this tree's prune sequence and
# 10 fold cross-validation (the default of cv.tree anyway)
fake.tree.cv <- cv.tree(fake.tree.default, K= 10)
plot(fake.tree.cv$size, fake.tree.cv$dev,
     col="firebrick", pch=19, main="cv.tree(...) 10-fold",
     xlab="size", ylab="RSS")
lines(fake.tree.cv$size, fake.tree.cv$dev,
      col="firebrick", lwd=2)
```

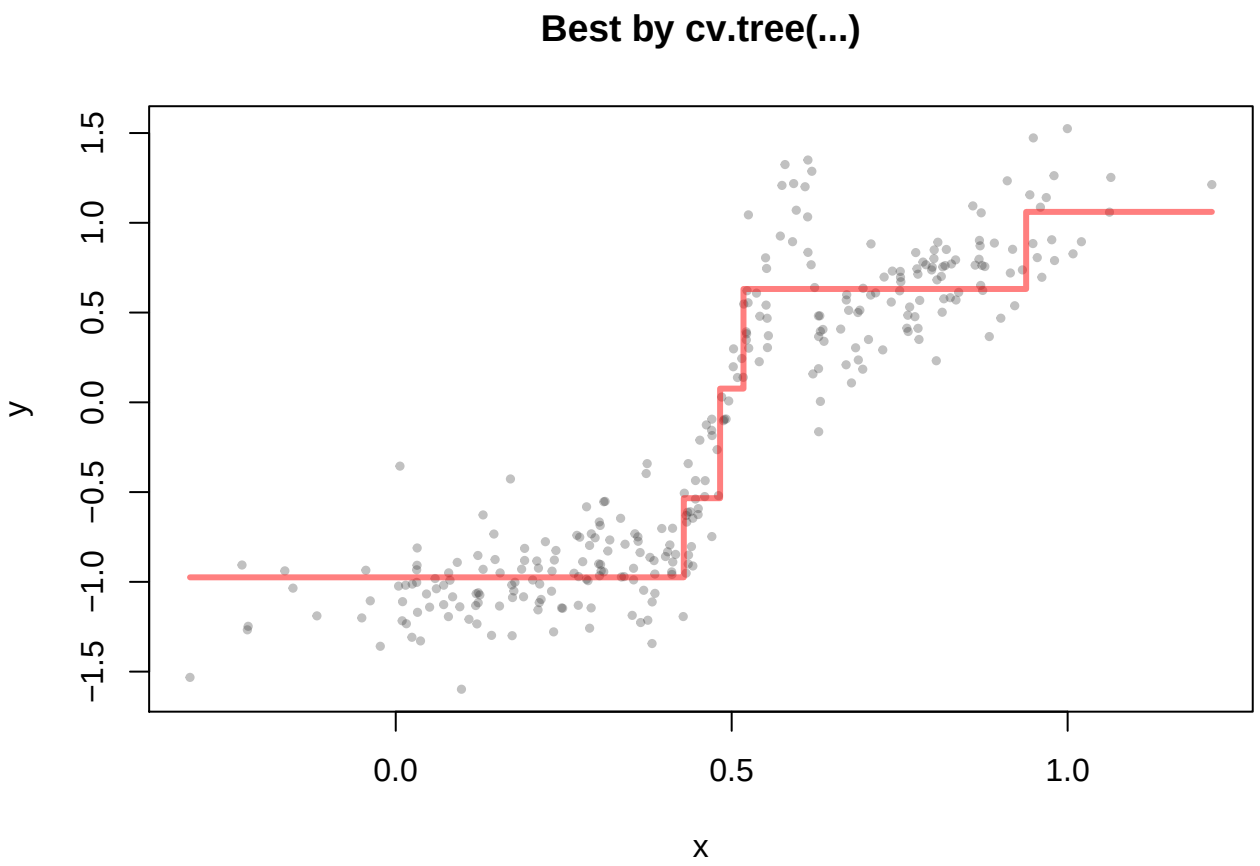



```
# And the actual values
cbind(size=fake.tree.cv$size, dev = fake.tree.cv$dev)
```

```
##      size       dev
## [1,]    5  21.50029
## [2,]    4  25.61302
## [3,]    3  27.86390
## [4,]    2  29.03055
## [5,]    1 208.58514
```

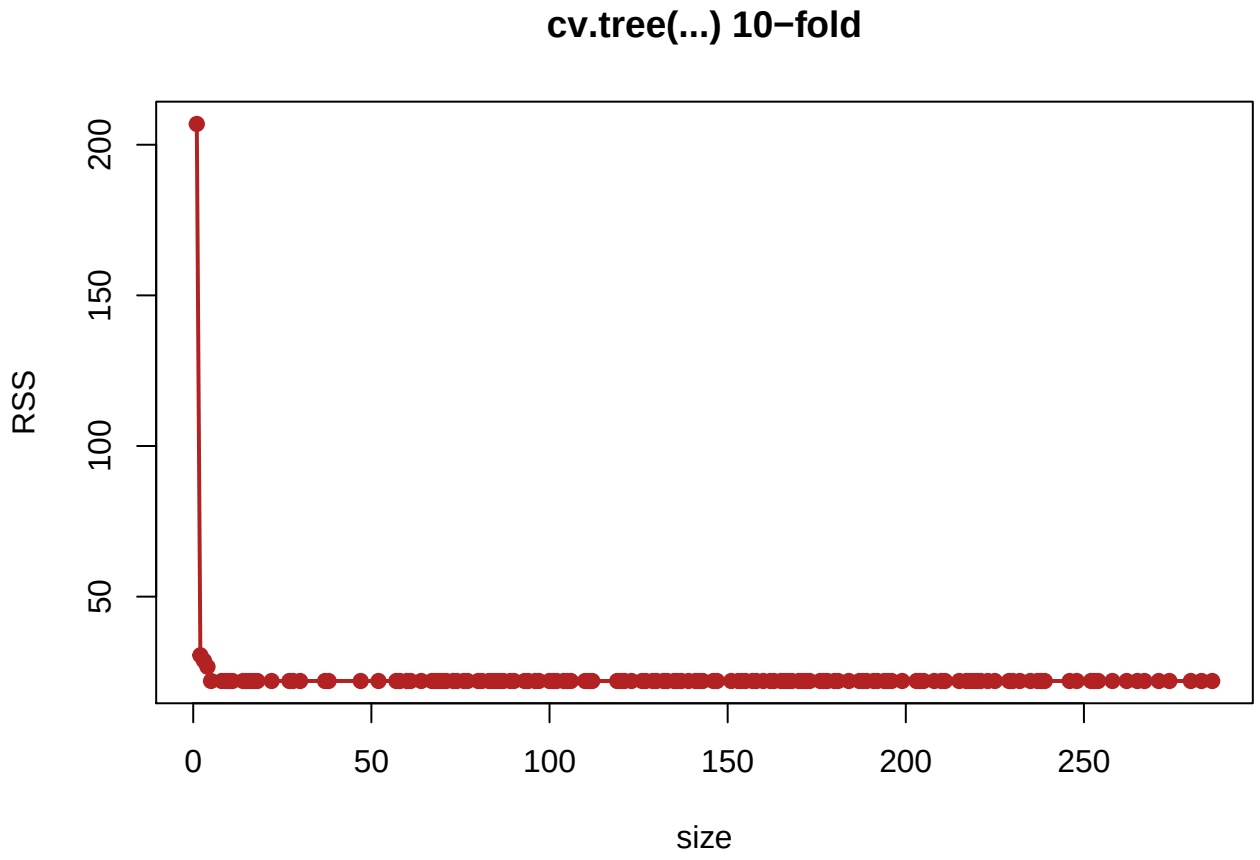
The cross-validation minimum produced by `cv.tree(...)` suggests that the best size for our tree would be 5. Consequently, we prune our tree back to size 5.

```
# The fitted function
fake.tree.cv.best <- prune.tree(fake.tree.default,
                                best=5
                                )
plot(x,y,
     col=adjustcolor("grey20", 0.3), pch=19, cex=0.5,
     main = "Best by cv.tree(...)")
partition.tree(fake.tree.cv.best, add=TRUE,
               col=adjustcolor("red", 0.5), lwd=3)
```



Important Note `cv.tree(...)` would also have produced `best=5` if given the bushy tree. We can see this as follows

```
# We can contrast this with the values when we begin with the bushy
# tree
fake.tree.b.cv <- cv.tree(fake.tree.bushy, K= 10)
plot(fake.tree.b.cv$size, fake.tree.b.cv$dev,
     col="firebrick", pch=19, main="cv.tree(...) 10-fold",
     xlab="size", ylab="RSS")
lines(fake.tree.b.cv$size, fake.tree.b.cv$dev,
      col="firebrick", lwd=2)
```




```
# And we can look at the earliest (lowest values of size)
sizes <- fake.tree.b.cv$size < 20
cbind(size=fake.tree.b.cv$size, dev=fake.tree.b.cv$dev)[sizes,]
```

```
##       size       dev
##  [1,]   18  21.99774
##  [2,]   17  21.99774
##  [3,]   16  21.99774
##  [4,]   15  21.99774
##  [5,]   14  21.99774
##  [6,]   11  21.99774
##  [7,]   10  21.99774
##  [8,]    9  21.99774
##  [9,]    8  21.99774
## [10,]    5  21.99774
## [11,]    4  26.72662
## [12,]    3  28.75475
## [13,]    2  30.53524
## [14,]    1 206.90333
```

Deviances are all identical for `size = 5` and beyond. Note that in `cv.tree(...)` the averages in the cross-validations are calculated for the same values of the cost complexity parameter λ (or `k` in `prune.tree(...)`) and **not** for the tree size.

Our problem seems to be in the `cv.tree(...)` function itself. To see the source code just type `cv.tree` in the `R` console.

```
cv.tree
```

```
## function (object, rand, FUN = prune.tree, K = 10, ...)
```

```
## {
##     if (!inherits(object, "tree"))
##         stop("not legitimate tree")
##     m <- model.frame(object)
##     extras <- match.call(expand.dots = FALSE)$...
##     FUN <- deparse(substitute(FUN))
##     init <- do.call(FUN, c(list(object), extras))
##     if (missing(rand))
##         rand <- sample(K, length(m[[1L]]), replace = TRUE)
##     cvdev <- 0
##     for (i in unique(rand)) {
##         tlearn <- tree(model = m[rand != i, , drop = FALSE])
##         plearn <- do.call(FUN, c(list(tlearn, newdata = m[rand ==
##             i, , drop = FALSE], k = init$k), extras))
##         cvdev <- cvdev + plearn$dev
##     }
##     init$dev <- cvdev
##     init
## }
## <environment: namespace:tree>
```

A careful examination identifies the problem. In performing its cross-validations `cv.tree` always calls `tree(...)` with the **default** `tree.control` parameter values and not with those tree.control values we used to constuct the bushiest tree (`object` in the code). This means, as with the bushiest tree, when interest does not lie in the default settings `cv.tree(...)` may not be appropriate.

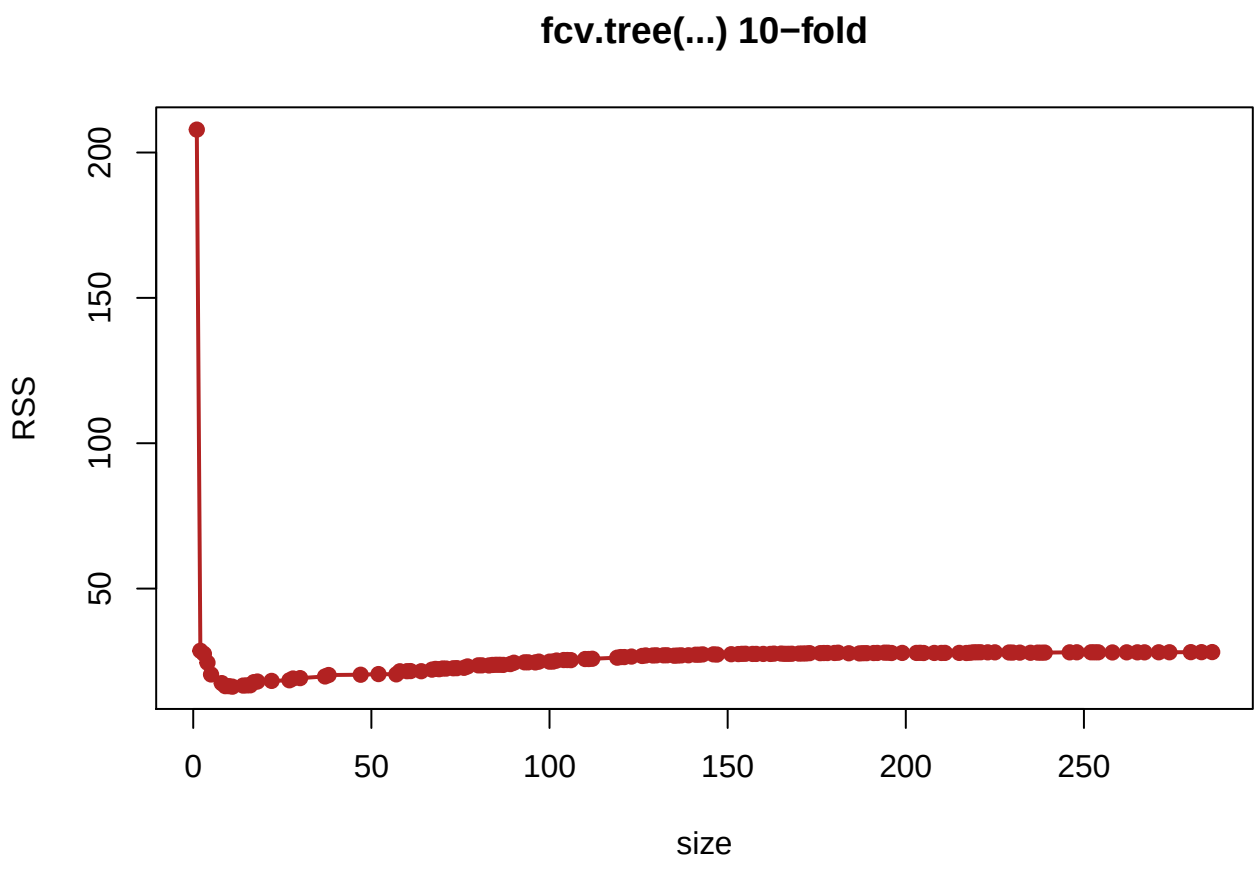
Fortunately, it is not hard to add the necessary feature. We define an alternative `fcv.tree(...)` to `cv.tree(...)` as follows:

```
#
# Below is identical to B.D. Ripley's cv.tree with only
# those exceptions noted by comments.  RWO.
fcv.tree <- function (object, rand, FUN = prune.tree, K = 10,
                      control = NULL, ...)
{ # Added the control ^^^^^^  argument ... rwo
  #
  if (!inherits(object, "tree"))
    stop("not legitimate tree")
  m <- model.frame(object)
  #
  # Added the tree package default for the control argument ... rwo
  if (is.null(control)) control <- tree.control(nobs = nrow(m),
                                                mincut = 5,
                                                minsize = 10,
                                                mindev = 0.01)
  extras <- match.call(expand.dots = FALSE)$...
  FUN <- deparse(substitute(FUN))
  init <- do.call(FUN, c(list(object), extras))
  if (missing(rand))
    rand <- sample(K, length(m[[1L]]), replace = TRUE)
  cvdev <- 0
  for (i in unique(rand)) {
    tlearn <- tree(model = m[rand != i, , drop = FALSE],
                   control = control)
    # Added the above ^^^^^^^^^^^ control argument ... rwo
    #
```

```
    plearn <- do.call(FUN, c(list(tlearn,
                                  newdata = m[rand == i, , drop = FALSE],
                                  k = init$k), extras))
    cvdev <- cvdev + plearn$dev
  }
  init$dev <- cvdev
  init
}
```

We can now try using this function in place of `cv.tree(...)`.

```
# Begin with our bushy tree and get the deviances using
# the bushy tree's prune sequence and 10 fold cross-validation.
# We have to pass on the same tree.control parameter values
# that were used to construct our tree.
fake.tree.b.cv <- fcv.tree(fake.tree.bushy, K= 10,
                           control=tree.control(nobs = length(x),
                                                mincut = 1,
                                                minsize = 2,
                                                mindev = 0.0))

plot(fake.tree.b.cv$size, fake.tree.b.cv$dev,
     col="firebrick", pch=19, main="fcv.tree(...) 10-fold",
     xlab="size", ylab="RSS")
lines(fake.tree.b.cv$size, fake.tree.b.cv$dev,
      col="firebrick", lwd=2)
```

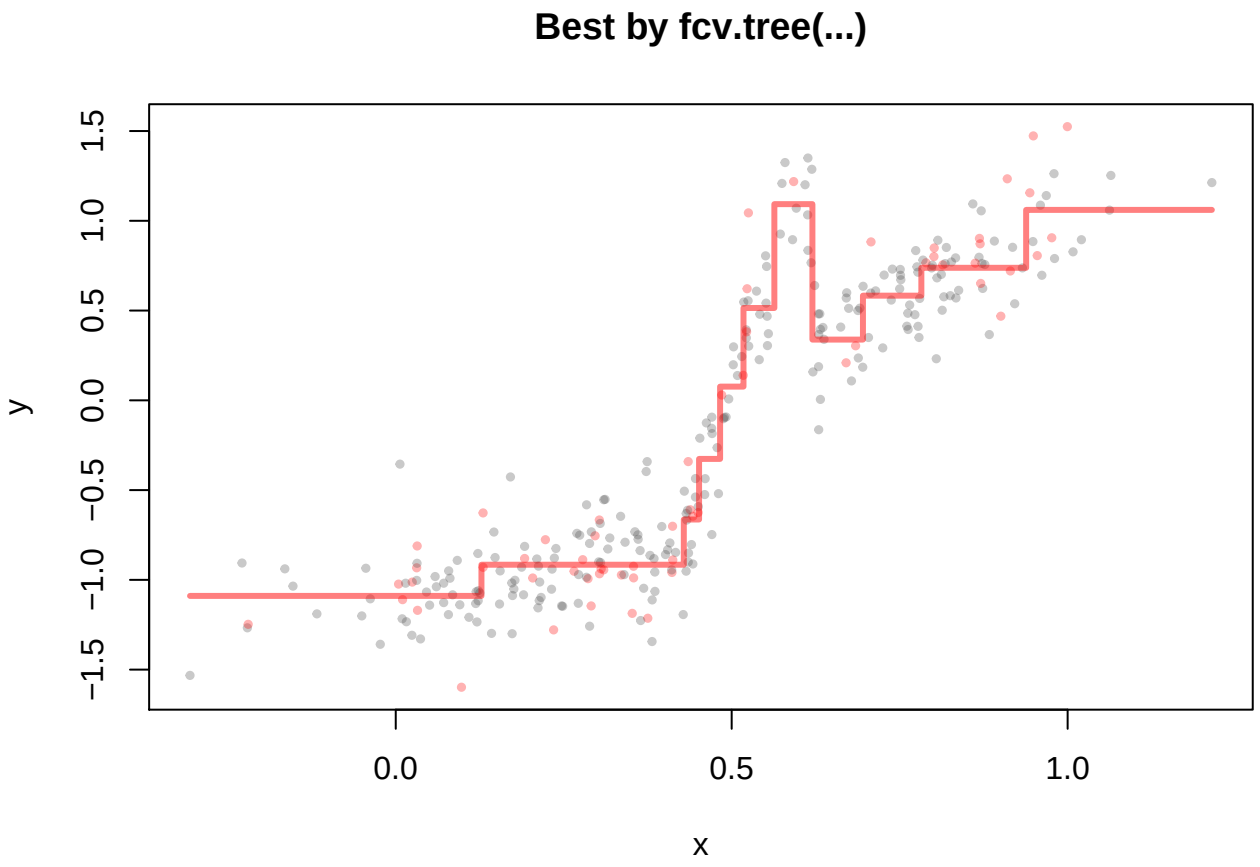


```
# And we can look at the earliest (lowest values of size)
sizes <- fake.tree.b.cv$size < 20
cbind(size=fake.tree.b.cv$size, dev=fake.tree.b.cv$dev)[sizes,]
```

```
##       size       dev
##  [1,]   18  18.02526
##  [2,]   17  17.80739
##  [3,]   16  16.73151
##  [4,]   15  16.69688
##  [5,]   14  16.63463
##  [6,]   11  16.26059
##  [7,]   10  16.43618
##  [8,]    9  16.43618
##  [9,]    8  17.48466
## [10,]    5  20.45860
## [11,]    4  24.50480
## [12,]    3  27.54869
## [13,]    2  28.61372
## [14,]    1 207.88216
```

Looking at these values suggests that the best size tree has 11 leaves.

```
# The fitted function
fake.tree.fcv.best <- prune.tree(fake.tree.bushy,
                                 best=11
                                 )
plot(x,y,
     col=col, pch=19, cex=0.5,
     main = "Best by fcv.tree(...)")
partition.tree(fake.tree.fcv.best, add=TRUE,
               col=adjustcolor("red", 0.5), lwd=3)
```



2 Multiple explanatory variates

Recursively partitioned trees are easily extended to the case where there is more than one explanatory variate.

As we have used them, trees have fitted the response y to a single continuous explanatory variate x following a greedy algorithm where at each step a region B of x was split into two pieces $B_{left} = \{x \mid x < c\}$ and $B_{right} = \{x \mid x \geq c\}$. The constant c was chosen to minimize the resulting residual sum of squares (or deviances).

When there is more than one continuous explanatory variate, there is an obvious (and greedy) step we could add to this greedy algorithm. Whenever a region B is to be split, we search **over all explanatory** variates and choose that explanatory variate whose consequent split of B into B_{left} and B_{right} would reduce the RSS the most.

That is, if we have p explanatory variates $x_1, \ldots, x_p$, then to split each region B

1. find the cut point c_i for variate x_i producing the **best split on this variate** into $B^i_{left} = \{x_i \mid x < c_i\}$ and $B^i_{right} = \{x_i \mid x \geq c_i\}$. These are *potential* splits.
2. of these potential splits, choose the variate and the split which would most reduce the RSS.

The result is a greedy binary tree, where each branch splits the data on the basis of a single explanatory variate only.

Note, if the explanatory variate is an unordered categorical variate, then rather than split on a cut point, the data are split by into two non-intersecting groups of possible categories.

2.1 Example: Ozone data revisited

Recall the `ozone` data from the package `ElemStatLearn`:

```
library(ElemStatLearn)
data(ozone)
```

which consists of 111 daily measurements taken from May until September 1973 in New York on four variates:

- `ozone`, the ozone concentration in parts per billion (ppb),
- `radiation`, the solar radiation energy measured in langleys,
- `temperature`, the maximum temperature that day in degrees Fahrenheit, and
- `wind`, the wind speed in miles per hour.

We are interested in modelling the response `ozone` as a function of the other three variates.

When we were considering smoothers (i.e. additive models, thin-plate splines, etc.) we used `ozone^(1/3)` as the response variate, so we will do the same here for consistency.

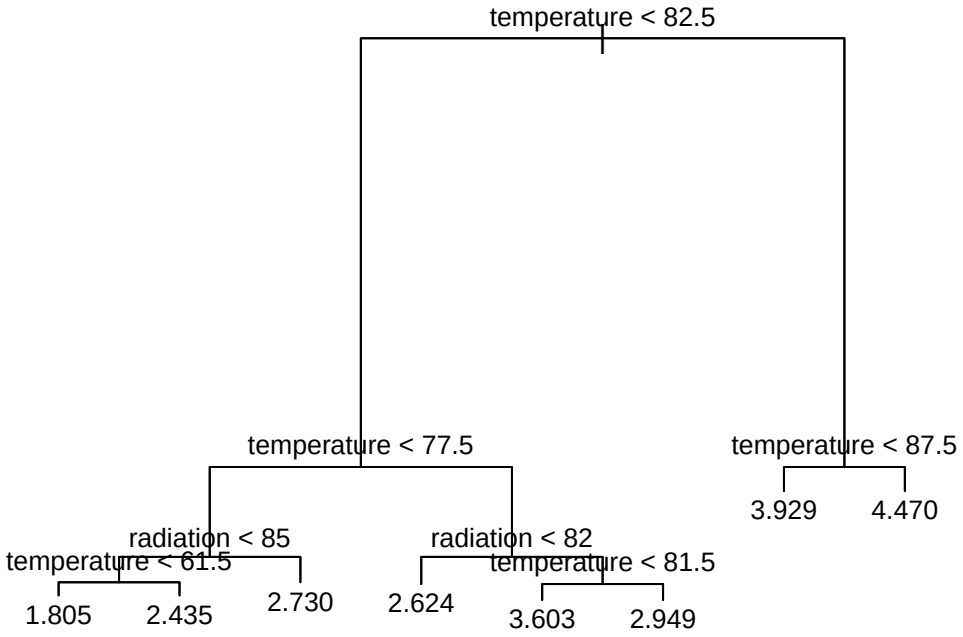
```
# In case you want to look at the data again.
library(rgl)  # Access to all of open GL graphics
par3d(FOV=1)
with(ozone,
     plot3d(radiation, temperature, ozone^(1/3),
            col=adjustcolor("steelblue", 0.5),
            type="s",
            size =2)
     )
```

```
oz.tree <- tree(ozone^(1/3) ~ radiation + temperature, data = ozone)

# We can see the structure of the tree.
oz.tree
```

```
## node), split, n, deviance, yval
##       * denotes terminal node
##
##  1) root 111 87.2100 3.248
##    2) temperature < 82.5 77 35.3700 2.828
##      4) temperature < 77.5 50 11.0600 2.572
##        8) radiation < 85 16  3.8930 2.238
##         16) temperature < 61.5 5  1.3990 1.805 *
##         17) temperature > 61.5 11  1.1310 2.435 *
##        9) radiation > 85 34  4.5340 2.730 *
##      5) temperature > 77.5 27 15.0100 3.301
##       10) radiation < 82 5  2.4310 2.624 *
##       11) radiation > 82 22  9.7670 3.454
##         22) temperature < 81.5 17  7.7900 3.603 *
##         23) temperature > 81.5 5  0.3218 2.949 *
##    3) temperature > 82.5 34  7.4700 4.199
##      6) temperature < 87.5 17  4.0370 3.929 *
##      7) temperature > 87.5 17  0.9438 4.470 *
```

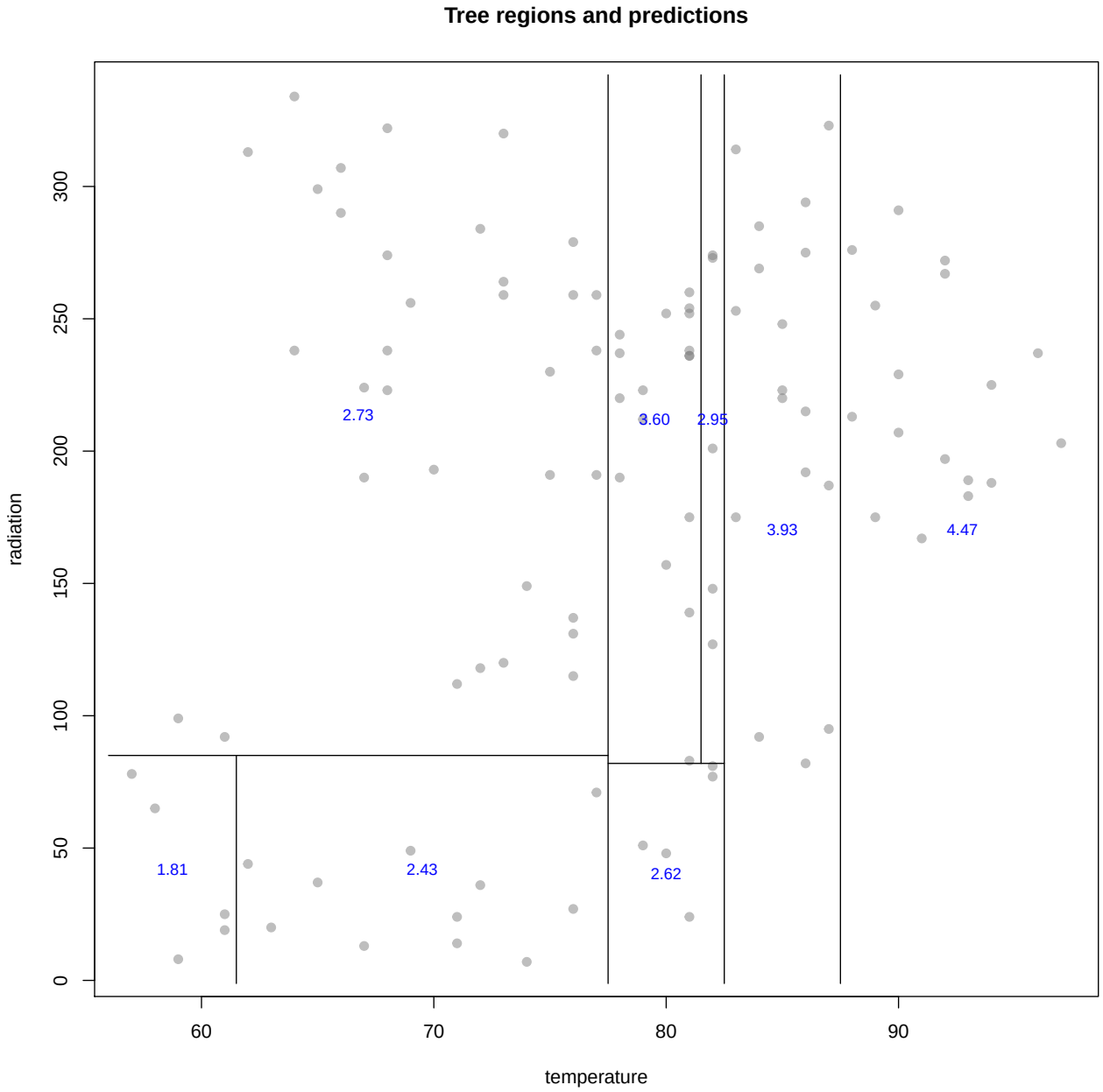
Note that the tree involves both `temperature` and `radiation`. The relative importance of the splits can be seen from the "proportional" tree:

```
plot(oz.tree, type="proportional")
text(oz.tree, cex=0.8)
```



The definition of the regions:

```
with(ozone,
     plot(temperature, radiation, pch=19,
          col=adjustcolor("grey50", 0.5), main = "Tree regions and predictions"))
partition.tree(oz.tree, add=TRUE, cex=0.8, col="blue")
```



Because there are only two explanatory see this in three dimensions, using the following function:

```
library(rgl)

plot3d.tree <- function(x,y,z, fov = NULL,
                        treecol = "grey50", control = NULL,
                        col="blue", ...) {

  if(!is.null(fov) & is.numeric(fov) & fov >= 0) par3d(FOV=fov)

  plot3d(x, y, z, col= col, ...)
  if (is.null(control)) {
      control <- tree.control(nobs=length(x),
                              mincut = 5,
                              minsize = 10,
```

```
                              mindev = 0.01)
  }
  # fit tree
  treefit <- tree(y ~ x + z, control=control)

  # determine fitted values on a grid
  xrg <- range(x)
  zrg <- range(z)
  gridx <- seq(min(xrg), max(xrg), length.out = 200)
  gridz <- seq(min(zrg), max(zrg), length.out = 200)
  heights <- outer(gridx, gridz,
                   function(x, z) {predict(treefit,
                           newdata = list(x=x, z=z))}
                   )
  # plot the fit
  rgl.surface(gridx, gridz, heights, color = treecol,
                alpha = 0.25, lit = FALSE)
}
```

The function can be used to create an interactive 3d plot:

```
x <- ozone$radiation
y <- ozone$ozone^(1/3)
z <- ozone$temperature
plot3d.tree(x, y, z, fov = 1, type="s", size=1)
# snapshot3d("regressiontree")
```

Note that the model effectively includes interaction terms.

Though plotting it is a little harder, we can easily incorporate **all** the explanatory variates:

```
oz.tree.all <- tree(ozone^(1/3) ~ radiation + temperature + wind,
                    data = ozone)

# We can see the contents of the fitted tree.
oz.tree.all
```

```
## node), split, n, deviance, yval
##       * denotes terminal node
##
##  1) root 111 87.2100 3.248
##    2) temperature < 82.5 77 35.3700 2.828
##      4) temperature < 77.5 50 11.0600 2.572
##        8) radiation < 85 16  3.8930 2.238
##         16) temperature < 61.5 5  1.3990 1.805 *
##         17) temperature > 61.5 11  1.1310 2.435 *
##        9) radiation > 85 34  4.5340 2.730
##         18) wind < 8.3 7  0.5153 3.099 *
##         19) wind > 8.3 27  2.8170 2.634 *
##      5) temperature > 77.5 27 15.0100 3.301
##       10) wind < 7.15 6  3.6080 4.147 *
##       11) wind > 7.15 21  5.8720 3.059
##         22) radiation < 133 5  0.8869 2.460 *
##         23) radiation > 133 16  2.6310 3.246 *
##    3) temperature > 82.5 34  7.4700 4.199
##      6) wind < 10.6 27  3.8430 4.348 *
```

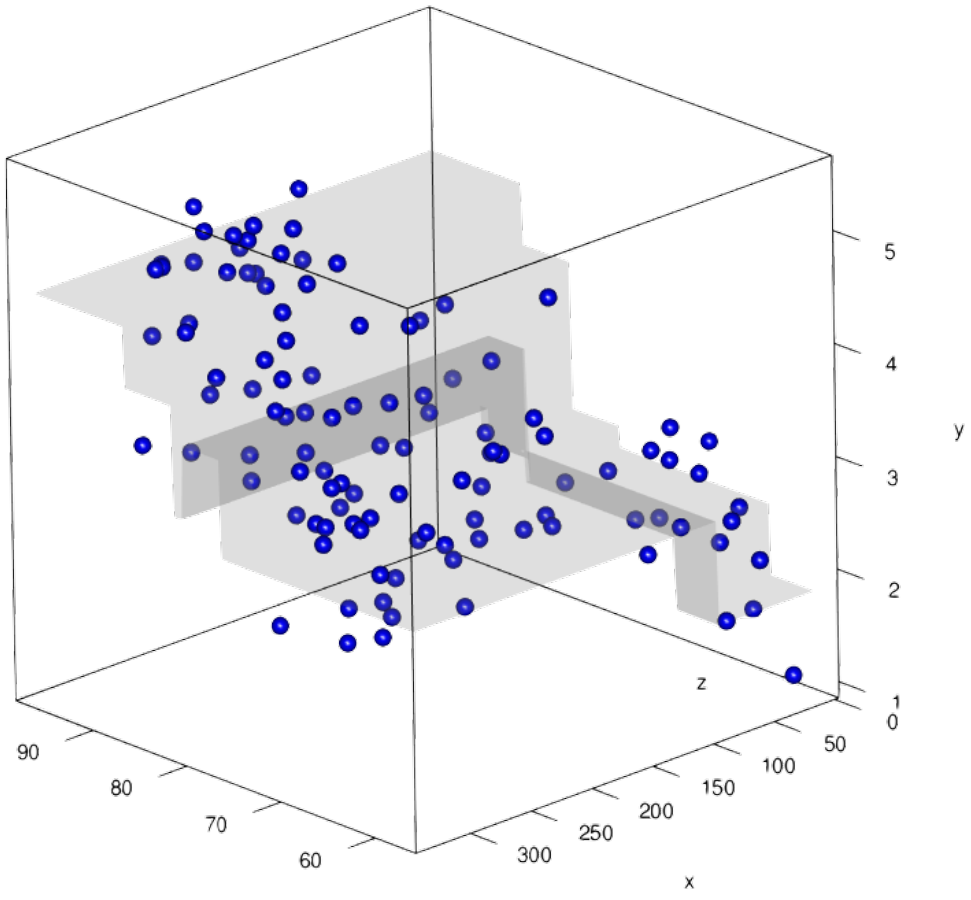
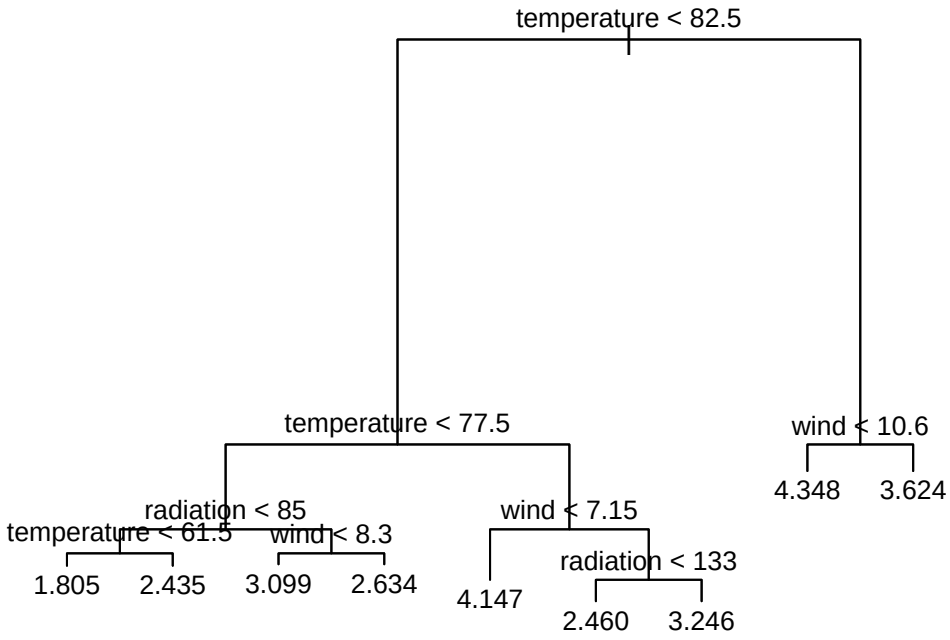


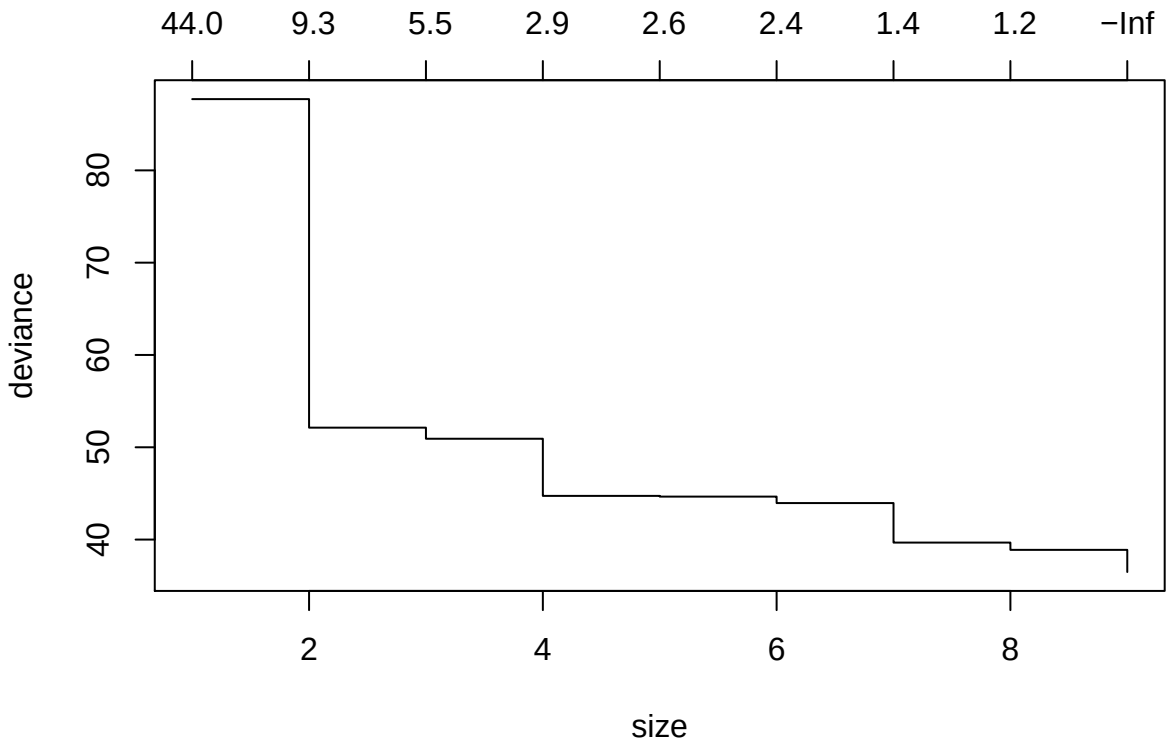
Figure 1: Fitting a regression tree to ozoneˆ(1/3), radiation, and temperature

```
##      7) wind > 10.6 7  0.7113 3.624 *
```

```
# and its tree structure
plot(oz.tree.all)
text(oz.tree.all, cex=0.8)
```



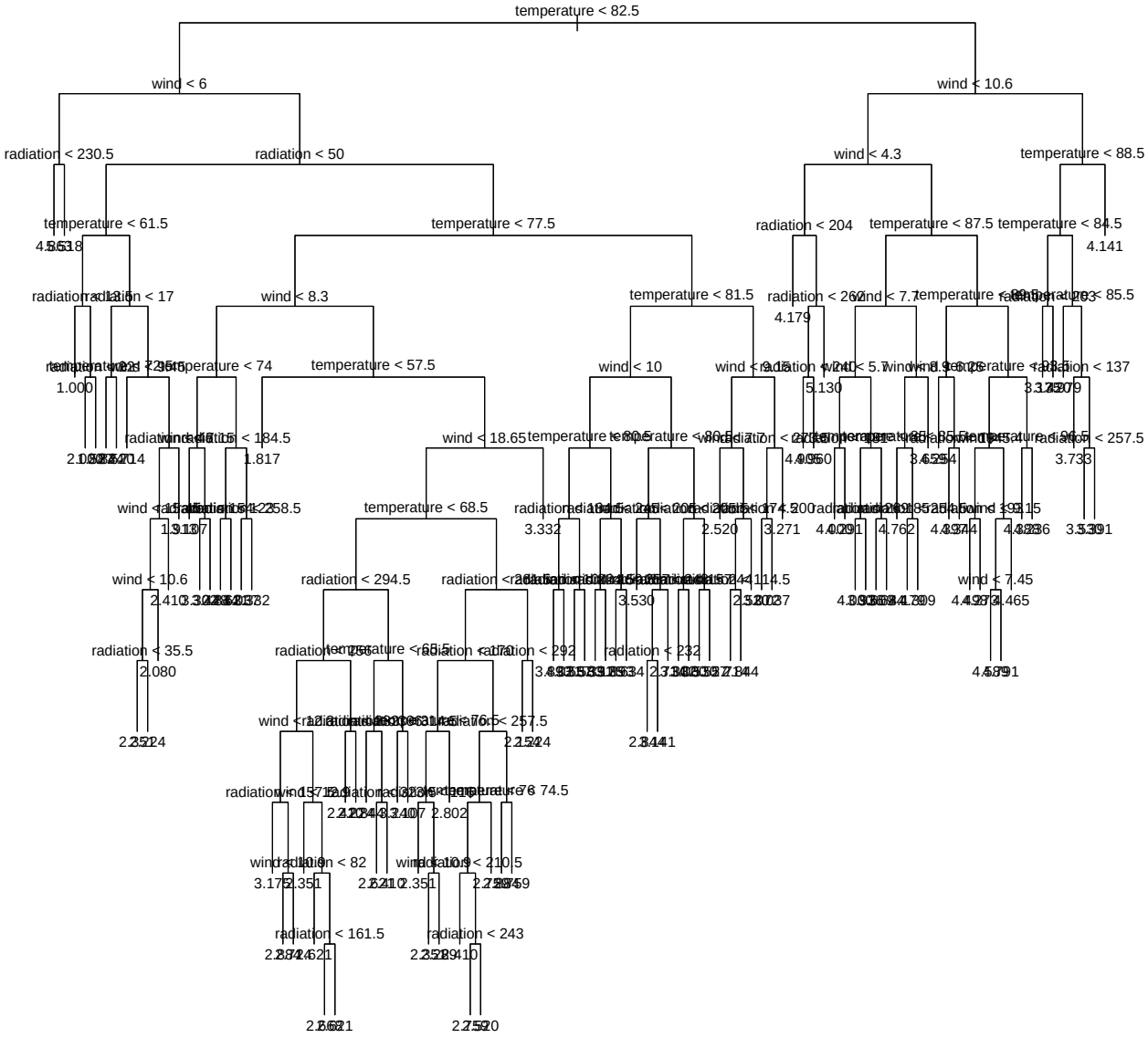

```
oz.tree.all.cv <- cv.tree(oz.tree.all)
plot(oz.tree.all.cv)
```



Suggests we use the full size tree (smallest cross-validated deviance). Might argue that we also might do with a smaller tree at some loss in predictive accuracy.

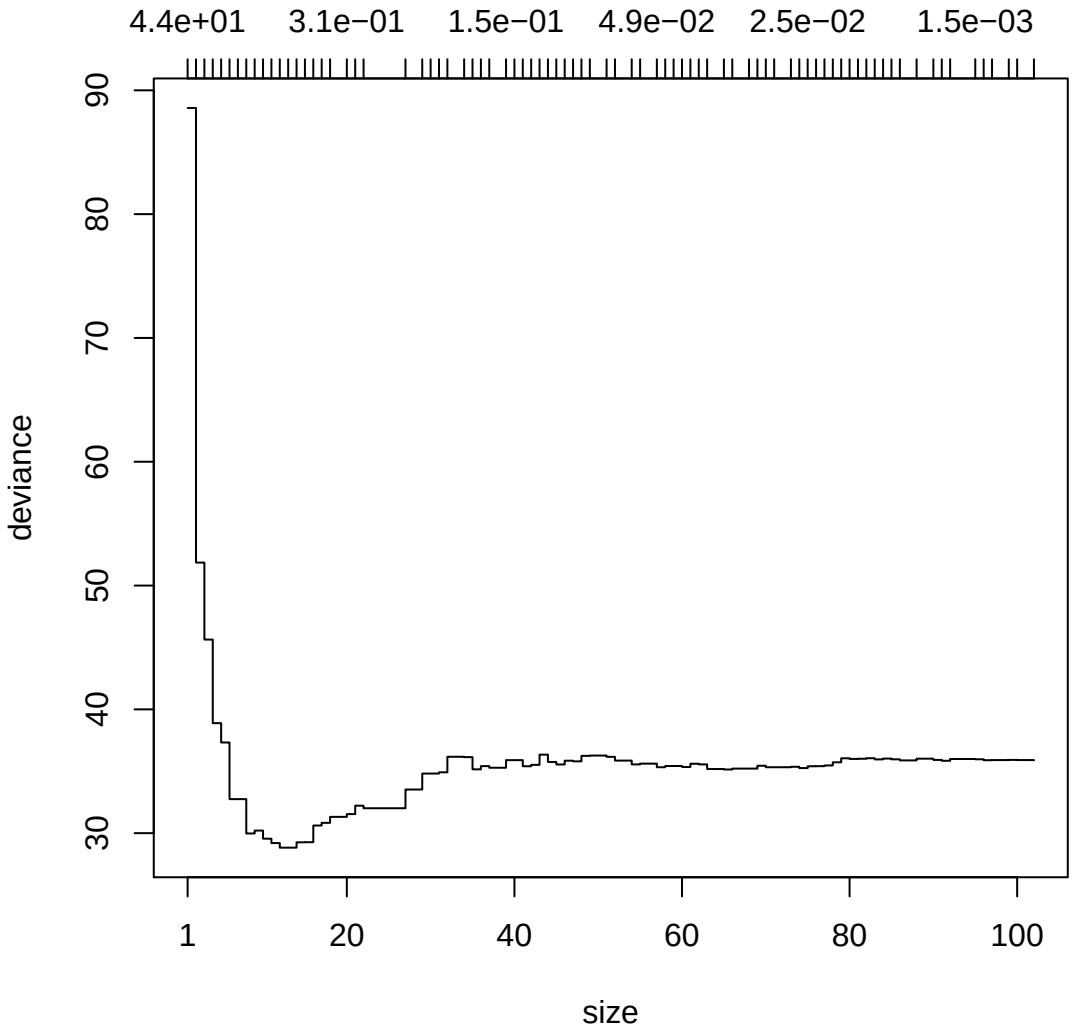
Alternatively, as before we could start with a bushier tree:

```
bushy <- tree.control(nrow(ozone), mindev = 1e-6, minsize=2)
oz.tree.all <- tree(ozone^(1/3) ~ radiation + temperature + wind,
                    data = ozone,
                    control = bushy)

# and its tree structure
plot(oz.tree.all, type="uniform")
text(oz.tree.all, cex=0.8)
```



And then try cross-validation

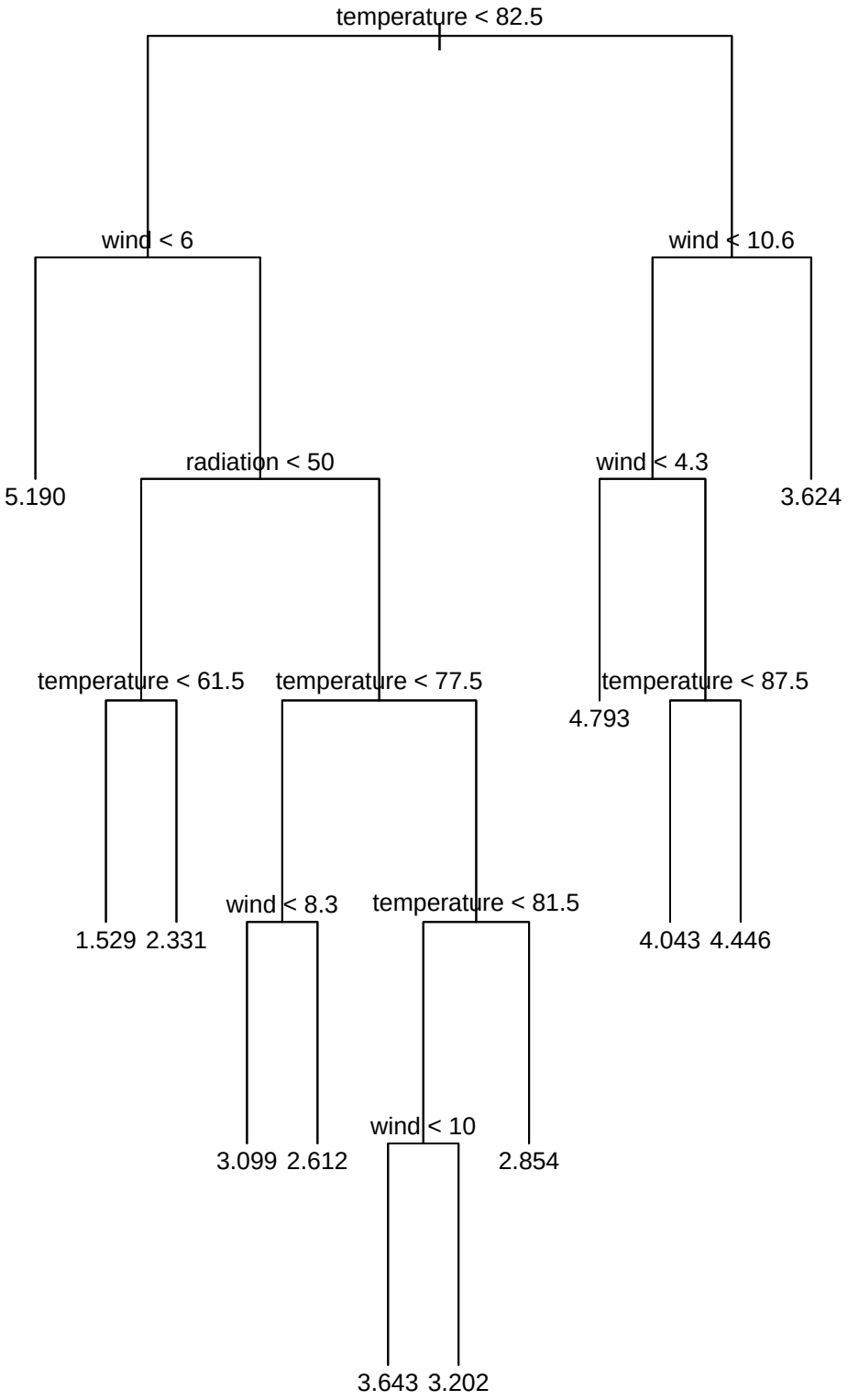
```
oz.tree.all.cv <- fcv.tree(oz.tree.all, control=bushy)
plot(oz.tree.all.cv)
```




```
#cbind(oz.tree.all.cv$size, oz.tree.all.cv$dev)
```

Looks like size could be around 12 or 13.

```
oz.tree.all.11 <- prune.tree(oz.tree.all, best = 12)
plot(oz.tree.all.11, type="uniform")
text(oz.tree.all.11, cex=0.8)
```



2.2 Example: Facebook data

Recall the `facebook` data from the course web page:

```
dataDir <- "/Users/rwoldford/Documents/Admin/courses/Stat444/Data/UCIrvineMLRep/FacebookMetrics"
completePathname <- paste(dataDir, "facebook.csv", sep="/")
facebook <- read.csv(completePathname, header=TRUE)
fb <- na.omit(facebook)
head(fb)
```

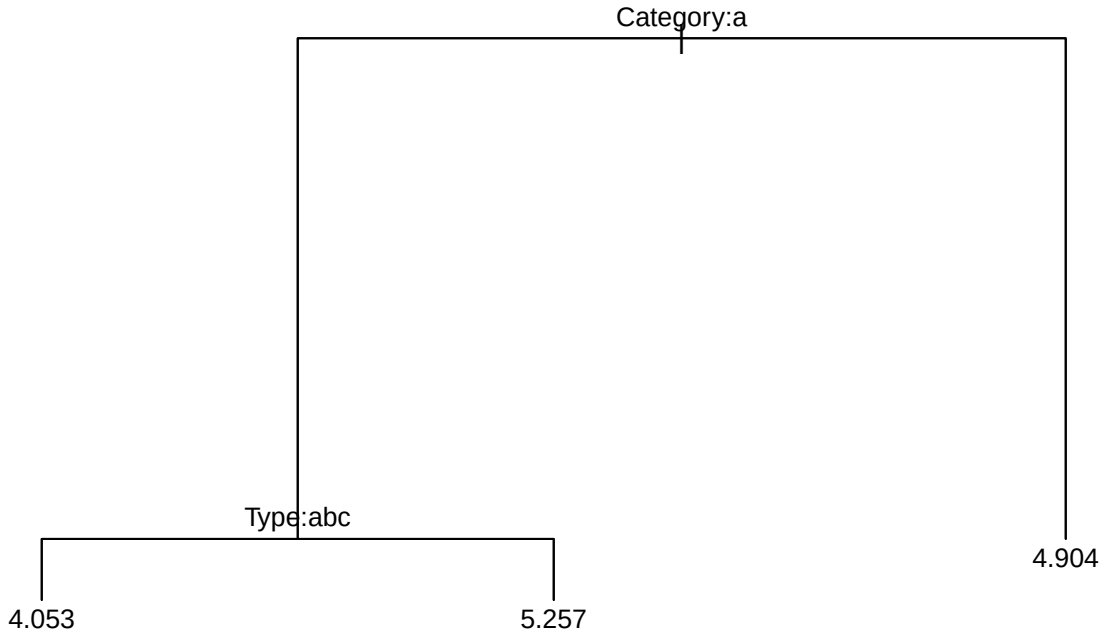
```
##   All.interactions share like comment Impressions.when.page.liked
## 1              100    17   79       4                        3078
## 2              164    29  130       5                       11710
## 3               80    14   66       0                        2812
## 4             1777   147 1572      58                       61027
## 5              393    49  325      19                        6228
## 6              186    33  152       1                       16034
##   Impressions Paid Post.Hour Post.Weekday Post.Month    Category   Type
## 1        5091    0         3            4         12     Product  Photo
## 2       19057    0        10            3         12     Product Status
## 3        4373    0         3            3         12 Inspiration  Photo
## 4       87991    1        10            2         12     Product  Photo
## 5       13594    0         3            2         12     Product  Photo
## 6       20849    0         9            1         12     Product Status
##   Page.likes
## 1     139441
## 2     139441
## 3     139441
## 4     139441
## 5     139441
## 6     139441
```

Let's consider first a tree modelling `log(like + 1)` as the response and `Type` and `Category`. This illustrates the use of **categorical** explanatory variates in the tree building process.

```
fb.tree <- tree(log(like+1) ~ Type + Category, data = facebook)
fb.tree
```

```
## node), split, n, deviance, yval
##       * denotes terminal node
##
## 1) root 499 709.900 4.556
##   2) Category: Action 214 417.500 4.092
##     4) Type: Link,Photo,Status 207 404.700 4.053 *
##     5) Type: Video 7   2.973 5.257 *
##   3) Category: Inspiration,Product 285 212.000 4.904 *
```

```
plot(fb.tree, type="proportional")
text(fb.tree, cex=0.8)
```

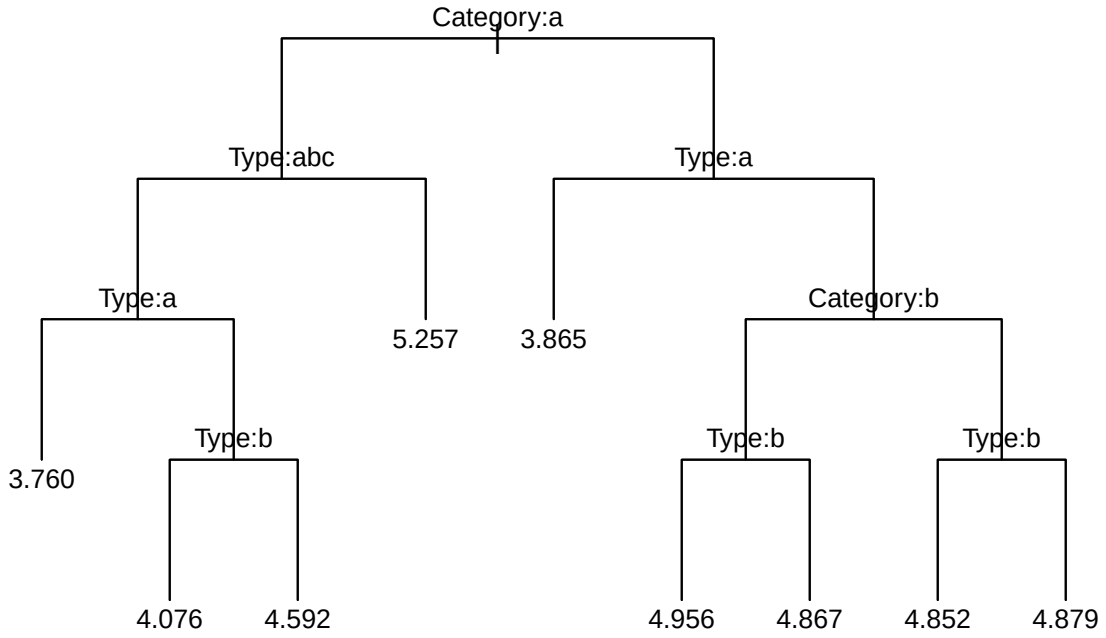


Note that the missing data in `facebook` are not a problem for regression trees. This is because, when data is missing, the prediction/fitted value for that case is the average at the deepest node it reaches in the tree.

(Note also that `partition.tree(...)` will no longer work for categorical explanatory variates.)

Let's build a bushier tree to see what happens with categorical variates.

```
fb.tree.bushy <- tree(log(like+1) ~ Type + Category, data = facebook,
                      control = tree.control(nobs = nrow(facebook),
                                             mincut = 2,
                                             mindev = 1e-6))
plot(fb.tree.bushy, type="uniform")
text(fb.tree.bushy, cex=0.8)
```




```
fb.tree.bushy
```

```
## node), split, n, deviance, yval
##       * denotes terminal node
##
##  1) root 499 709.900 4.556
##    2) Category: Action 214 417.500 4.092
##      4) Type: Link,Photo,Status 207 404.700 4.053
##        8) Type: Link 20  24.730 3.760 *
##        9) Type: Photo,Status 187 378.100 4.084
##         18) Type: Photo 184 374.600 4.076 *
##         19) Type: Status 3   2.685 4.592 *
##      5) Type: Video 7   2.973 5.257 *
##    3) Category: Inspiration,Product 285 212.000 4.904
##      6) Type: Link 2   0.272 3.865 *
##      7) Type: Photo,Status 283 209.500 4.911
##       14) Category: Inspiration 154 109.500 4.953
##         28) Type: Photo 150 108.000 4.956 *
##         29) Type: Status 4   1.435 4.867 *
##       15) Category: Product 129  99.400 4.860
##         30) Type: Photo 91  69.660 4.852 *
##         31) Type: Status 38  29.720 4.879 *
```

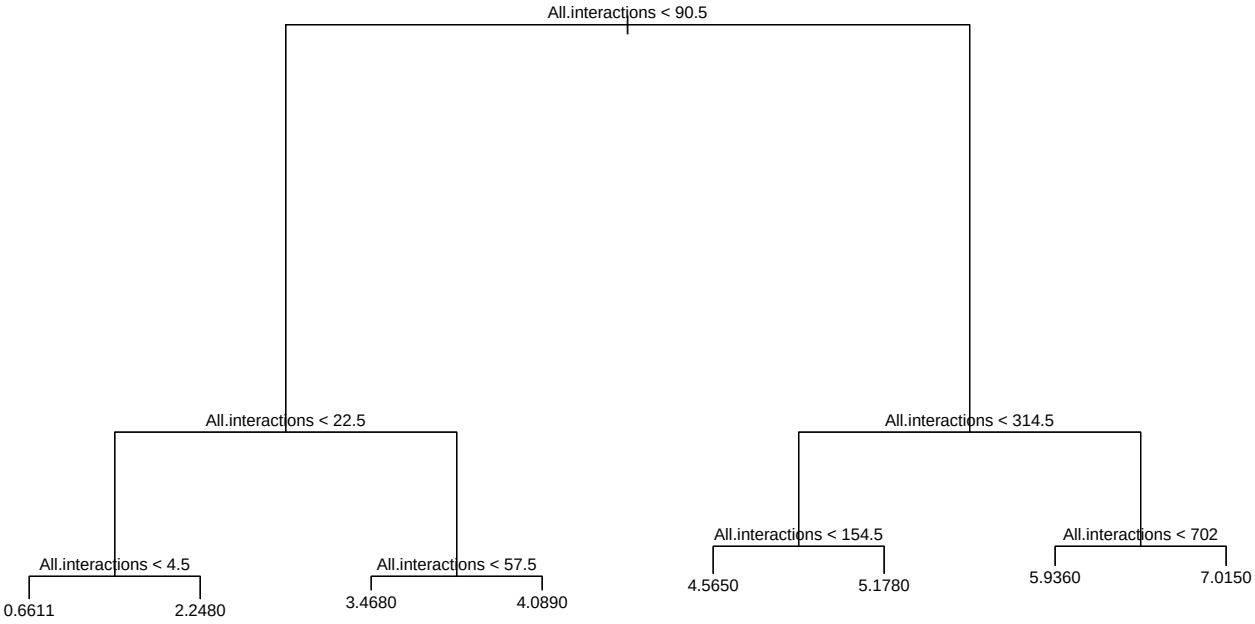
How do we interpret each node?

Now build the tree with all variates.

```
fb.tree <- tree(log(like+1) ~ All.interactions + share + comment + Impressions + Impressions.when.page.l
# Or, more simply ... and equivalently
fb.tree <- tree(log(like+1) ~. , data = facebook)
fb.tree
```

```
## node), split, n, deviance, yval
##       * denotes terminal node
##
##  1) root 495 689.100 4.5710
##    2) All.interactions < 90.5 171 172.700 3.4000
##      4) All.interactions < 22.5 38  29.000 1.7880
##        8) All.interactions < 4.5 11   4.520 0.6611 *
##        9) All.interactions > 4.5 27   4.806 2.2480 *
##      5) All.interactions > 22.5 133  16.880 3.8600
##       10) All.interactions < 57.5 49   3.298 3.4680 *
##       11) All.interactions > 57.5 84   1.645 4.0890 *
##    3) All.interactions > 90.5 324 157.900 5.1890
##      6) All.interactions < 314.5 256  33.220 4.9020
##       12) All.interactions < 154.5 115   3.118 4.5650 *
##       13) All.interactions > 154.5 141   6.288 5.1780 *
##      7) All.interactions > 314.5 68  24.250 6.2700
##       14) All.interactions < 702 47   2.379 5.9360 *
##       15) All.interactions > 702 21   4.986 7.0150 *
```

```
# Plot again with proportional lengths
plot(fb.tree, type="proportional")
text(fb.tree, cex=0.8)
```



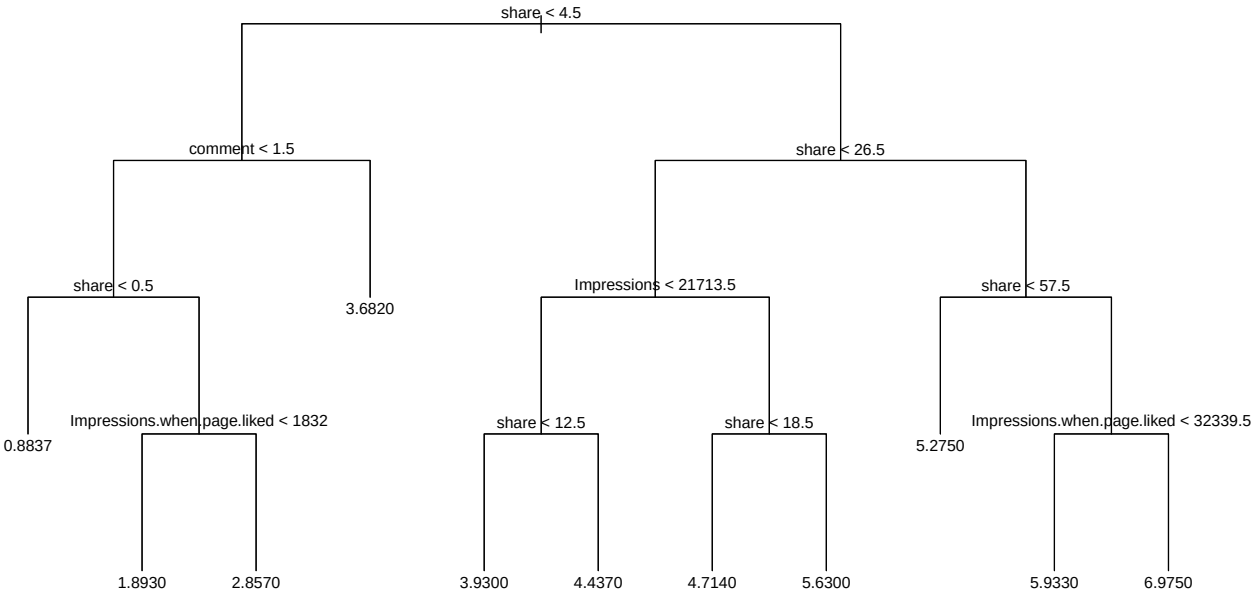
What variates seem important?

Let's try excluding `All.interactions` (the sum of `share`, `like`, and `comment`).

```
# All but the All.interactions variate
fb.tree <- tree(log(like+1) ~. -All.interactions, data = facebook)
fb.tree
```

```
## node), split, n, deviance, yval
##       * denotes terminal node
##
##  1) root 495 689.100 4.5710
##    2) share < 4.5 55  77.580 2.3760
##      4) comment < 1.5 42  41.250 1.9720
##        8) share < 0.5 12   7.587 0.8837 *
##        9) share > 0.5 30  13.770 2.4070
##         18) Impressions.when.page.liked < 1832 14   3.009 1.8930 *
##         19) Impressions.when.page.liked > 1832 16   3.826 2.8570 *
##      5) comment > 1.5 13   7.301 3.6820 *
##    3) share > 4.5 440 313.500 4.8450
##      6) share < 26.5 268  89.380 4.3990
##       12) Impressions < 21713.5 214  49.260 4.2640
##         24) share < 12.5 73  17.440 3.9300 *
##         25) share > 12.5 141  19.460 4.4370 *
##       13) Impressions > 21713.5 54  20.730 4.9350
##         26) share < 18.5 41   9.611 4.7140 *
##         27) share > 18.5 13   2.833 5.6300 *
##      7) share > 26.5 172  87.550 5.5410
##       14) share < 57.5 131  28.310 5.2750 *
##       15) share > 57.5 41  20.390 6.3900
##         30) Impressions.when.page.liked < 32339.5 23   3.290 5.9330 *
##         31) Impressions.when.page.liked > 32339.5 18   6.135 6.9750 *
```

```
# Plot the tree
plot(fb.tree, type="uniform")
text(fb.tree, cex=0.8)
```



Now which variates are important? How might we estimate the number of likes?

2.2.1 Response and explanatory variates

Given the context of the facebook data, the company is likely more interested in predicting viewer responses as a function of the features that are under the company's control.

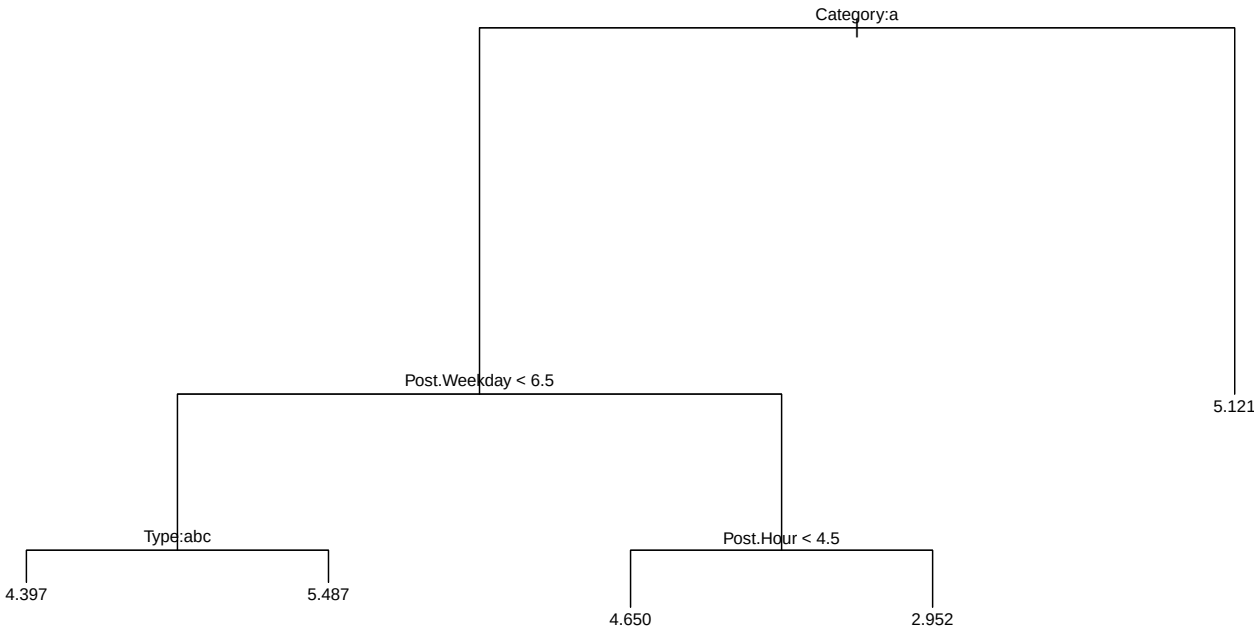
The company can control post time: `Post.Month`, `Post.Weekday`, `Post.Hour`; the `Category` and `Type`; and whether they have paid for promotion of the post or page. The remainder are arguably all user responses. The measures of impressions could be regarded as either response or explanatory – clearly some impressions that people experience are because friends shared it, while other impressions were purchased. Similarly, for `Page.likes`.

Let's model `All.interactions` as the response, and only those that are directly under the company's control as the explanatory variates. Again, we will take logarithms of `All.interactions` (+ 1).

```
fb.tree <- tree(log(All.interactions +1) ~ Paid + Post.Hour +
                                           Post.Weekday + Post.Month +
                                           Type + Category,
                data = facebook)

fb.tree
```

```
## node), split, n, deviance, yval
##       * denotes terminal node
##
##  1) root 499 723.300 4.749
##    2) Category: Action 215 440.600 4.258
##      4) Post.Weekday < 6.5 182 307.800 4.439
##        8) Type: Link,Photo,Status 175 296.500 4.397 *
##        9) Type: Video 7   3.248 5.487 *
##      5) Post.Weekday > 6.5 33  93.970 3.261
##       10) Post.Hour < 4.5 6  11.260 4.650 *
##       11) Post.Hour > 4.5 27  68.560 2.952 *
##    3) Category: Inspiration,Product 284 191.700 5.121 *
```

```
# Plot again with proportional lengths
plot(fb.tree, type="proportional")
text(fb.tree, cex=0.8)
```



Now which are the more important variates?

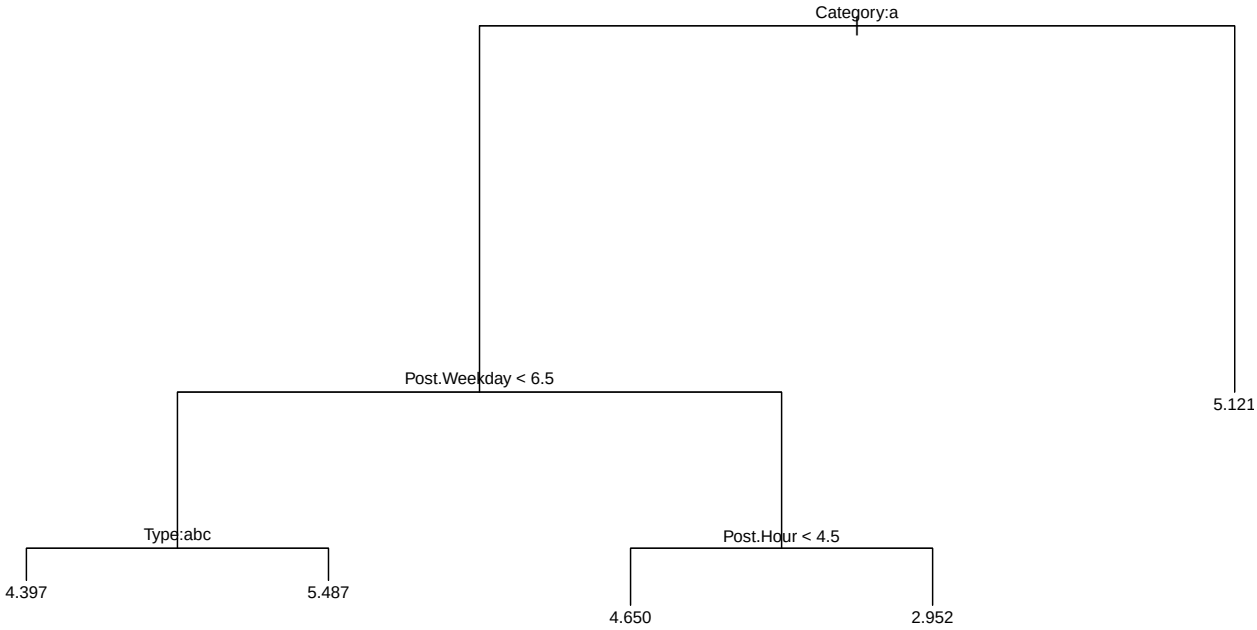
According to this tree, it seems that most important is the `Category`, namely whether it is an `Action` (special offers and contests) or not. If it is, then the time of the post (weekday and hour) matters as well as the type of post. If the post is an `Action`, then it is best that the post be a video posted any day but Saturday. Otherwise, the post should simply be either `Product` advertising or an `Inspirational` post. All others produce fewer interactions. Paying or not has no effect here. Nor does the month of the post.

Let's see whether there is a separation between those that liked the company's page or not.

```
fb.tree <- tree(log(All.interactions +1) ~ Page.likes + Paid + Post.Hour  +
                                           Post.Weekday + Post.Month +
                                           Type + Category,
                data = facebook)

fb.tree
```

```
## node), split, n, deviance, yval
##       * denotes terminal node
##
##  1) root 499 723.300 4.749
##    2) Category: Action 215 440.600 4.258
##      4) Post.Weekday < 6.5 182 307.800 4.439
##        8) Type: Link,Photo,Status 175 296.500 4.397 *
##        9) Type: Video 7   3.248 5.487 *
##      5) Post.Weekday > 6.5 33  93.970 3.261
##       10) Post.Hour < 4.5 6  11.260 4.650 *
##       11) Post.Hour > 4.5 27  68.560 2.952 *
##    3) Category: Inspiration,Product 284 191.700 5.121 *
```

```
# Plot again with proportional lengths
plot(fb.tree, type="proportional")
text(fb.tree, cex=0.8)
```



No change!

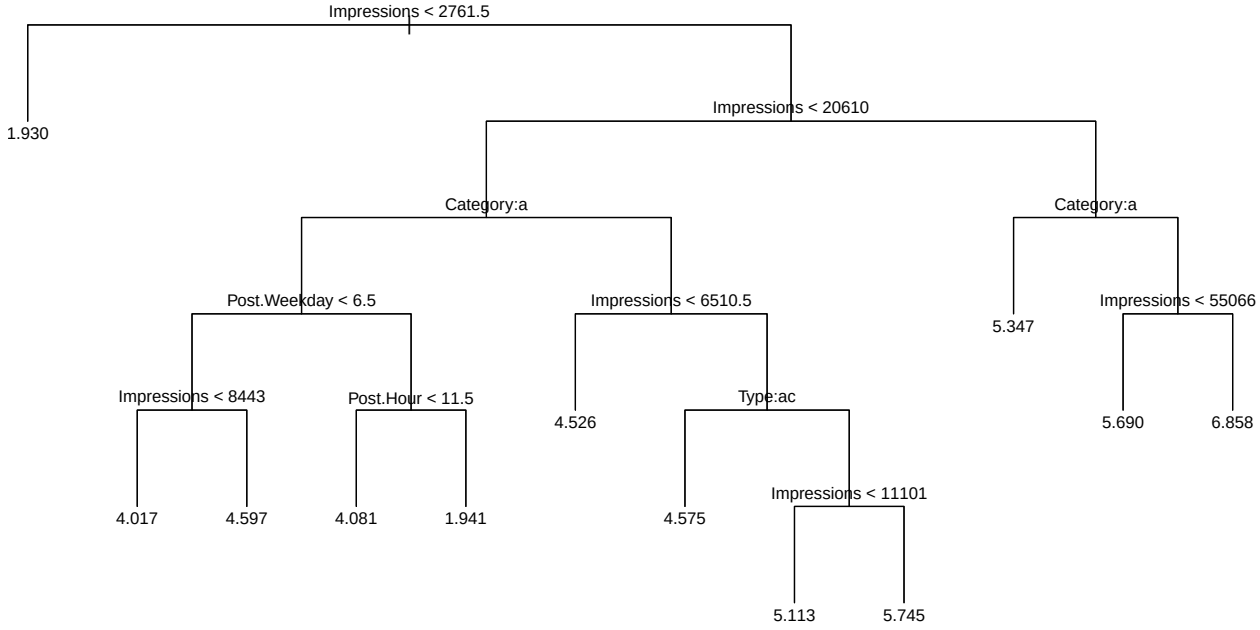
What if we add `Impressions`?

```
fb.tree <- tree(log(All.interactions +1) ~ Page.likes + Paid + Post.Hour + Impressions +
                                           Post.Weekday + Post.Month +
                                           Type + Category,
                data = facebook)

fb.tree
```

```
## node), split, n, deviance, yval
##       * denotes terminal node
##
##   1) root 499 723.300 4.749
##     2) Impressions < 2761.5 32  42.140 1.930 *
##     3) Impressions > 2761.5 467 409.400 4.943
##       6) Impressions < 20610 332 205.700 4.653
##        12) Category: Action 106  86.230 4.127
##          24) Post.Weekday < 6.5 92  50.870 4.251
##            48) Impressions < 8443 55  25.300 4.017 *
##            49) Impressions > 8443 37  18.130 4.597 *
##          25) Post.Weekday > 6.5 14  24.760 3.317
##            50) Post.Hour < 11.5 9   5.939 4.081 *
##            51) Post.Hour > 11.5 5   4.094 1.941 *
##        13) Category: Inspiration,Product 226  76.430 4.900
##          26) Impressions < 6510.5 92  11.600 4.526 *
##          27) Impressions > 6510.5 134  43.110 5.157
##            54) Type: Link,Status 22   9.164 4.575 *
##            55) Type: Photo 112  25.030 5.271
```

```
##             110) Impressions < 11101 84  14.060 5.113 *
##             111) Impressions > 11101 28   2.578 5.745 *
##       7) Impressions > 20610 135 107.600 5.654
##        14) Category: Action 79  46.250 5.347 *
##        15) Category: Inspiration,Product 56  43.440 6.086
##          30) Impressions < 55066 37  12.930 5.690 *
##          31) Impressions > 55066 19  13.390 6.858 *
```

```
# Plot again with uniform lengths
plot(fb.tree, type="uniform")
text(fb.tree, cex=0.8)
```



Not surprisingly, perhaps, `Impressions` plays an important role. If the post appears less often (`Impressions < 2761.5`), then there will be many fewer interactions with it. If the post appears often (`Impressions >= 2761.5`) then there will be a lot of interactions. If the post is an `Action`, it again seems to reduce the number of interactions. If additionally, the `Impressions` are between $2,762$ and $20,610$, then Saturdays are to be avoided, and if not, then post before noon. For `Inspiration`, or `Product` (i.e. not an `Action` category of post), then the post should be a `Photo` and not simply a `Link` or `Status` post.

See how easily a regression tree can be used as a "decision tree"?

3 Variability

Because regression trees make such abrupt changes at values of the explanatory variates, we might expect that where that cut is made will be sensitive to the data.

Let's look at a few samples to get some sense of how variable these trees can be. We could return to the fake data that we had before, since we know exactly how that data is generated (i.e. we have the generative model). Instead, let's look at the last few examples again.

3.1 Sampling with replacement

We only have a single data set in hand, say the `facebook` data. This is a sample $\mathcal{S}$ which we are using to build a regression tree $T(\mathcal{S})$. Ideally, we would like to have a population $\mathcal{P}$ from which we could draw many samples $\mathcal{S}_i$ and tree estimates $T(\mathcal{S}_i)$.

As when determining the predictive accuracy, we could use the sample $\mathcal{S}$ as a population $\mathcal{P}_0$ in place of the unavailable $\mathcal{P}$. We now draw samples $\mathcal{S}_i$ from $\mathcal{P}_0$ and construct the regression tree on each. Then we could get some idea of the variability of our regression trees.

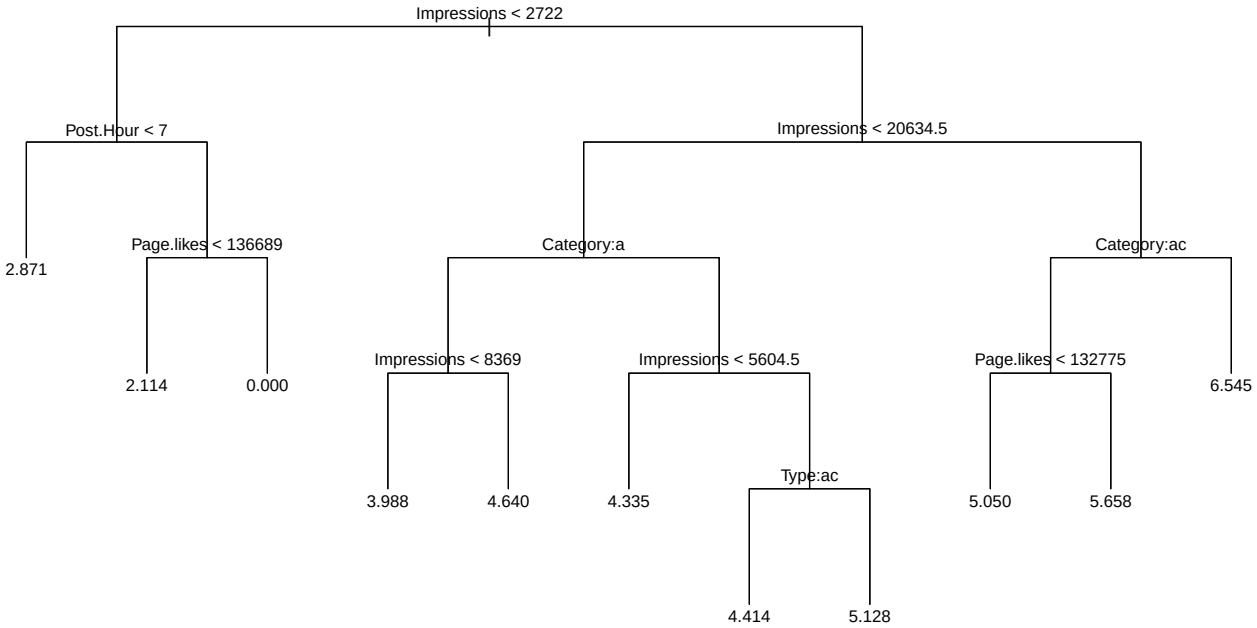
Finally, because we are interested in the variability of trees built on samples of the same size as $\mathcal{S}$, we will need to draw our samples $\mathcal{S}_i$ from $\mathcal{P}_0$ **with replacement**. (Otherwise we would just be drawing the same sample over and over.)

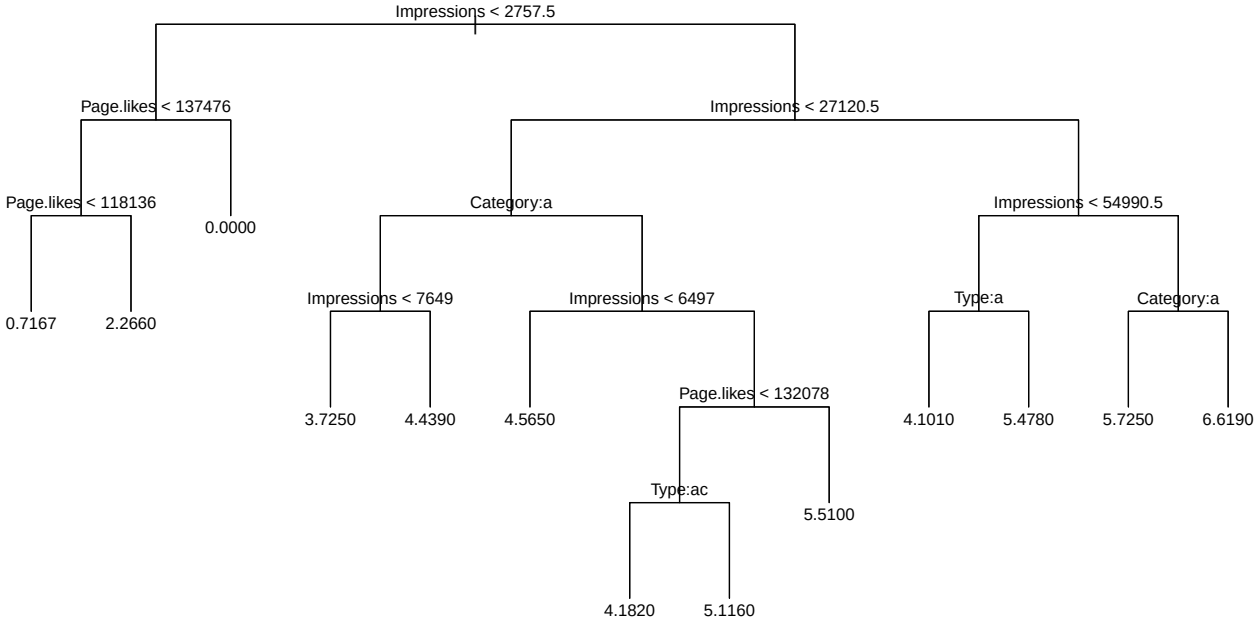
We need a function that will do that

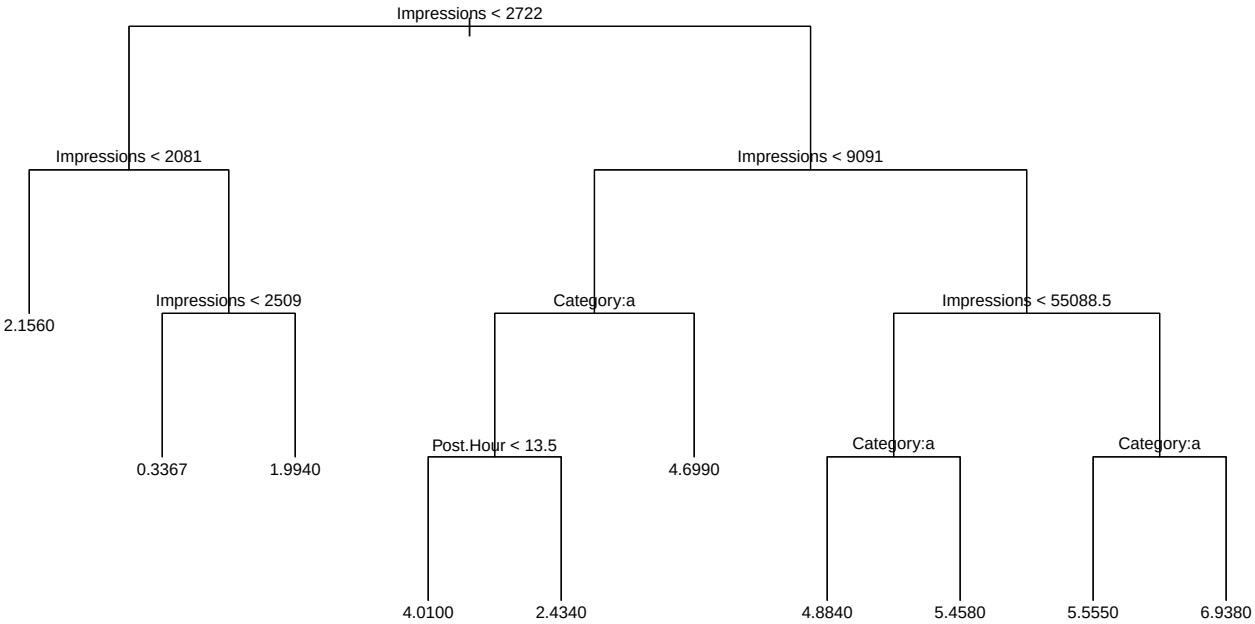
```
getSample <- function(pop, samplesize){
  selection <- sample(1:nrow(pop), samplesize, replace = TRUE)
  pop[selection,]
}

getTree <- function(formula, samp, ...){
  tree(formula = formula, data = samp, ...)
}
```

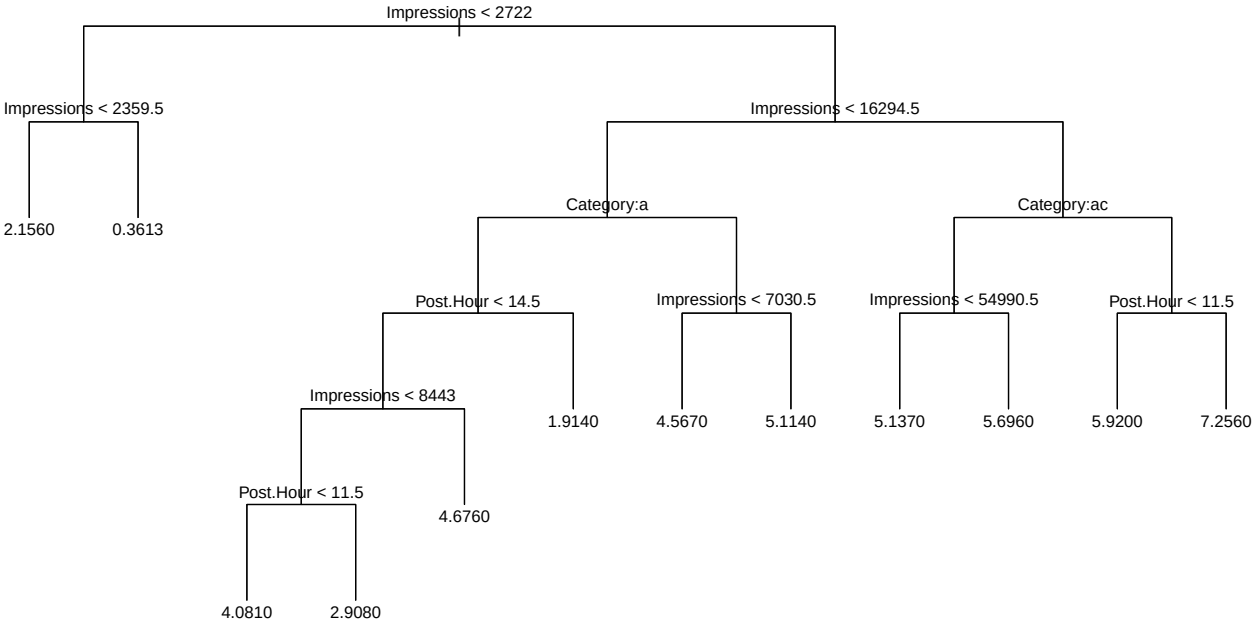
Let's try that on the `facebook` data and plot a few trees

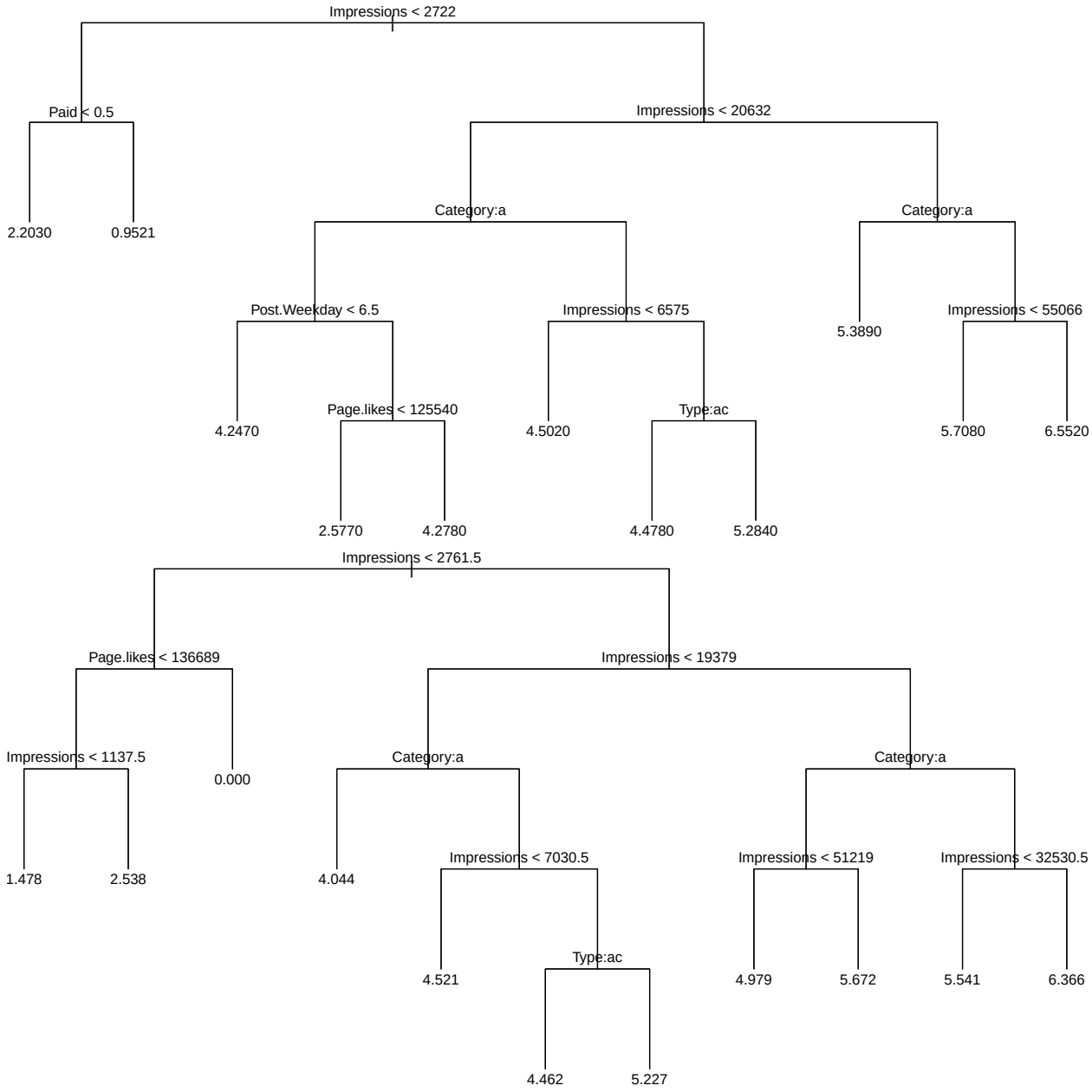
```
formula <- paste0("log(All.interactions +1) ~ ",
                  paste("Page.likes", "Paid", "Post.Hour", "Impressions",
                        "Post.Weekday", "Post.Month", "Type", "Category",
                        sep="+")
)

N <- nrow(facebook)
for(i in 1:10){
  samp <- getSample(facebook,N)
  samp.tree <- getTree(formula, samp)
  plot(samp.tree, type="uniform")
  text(samp.tree, cex=0.8)
}
```

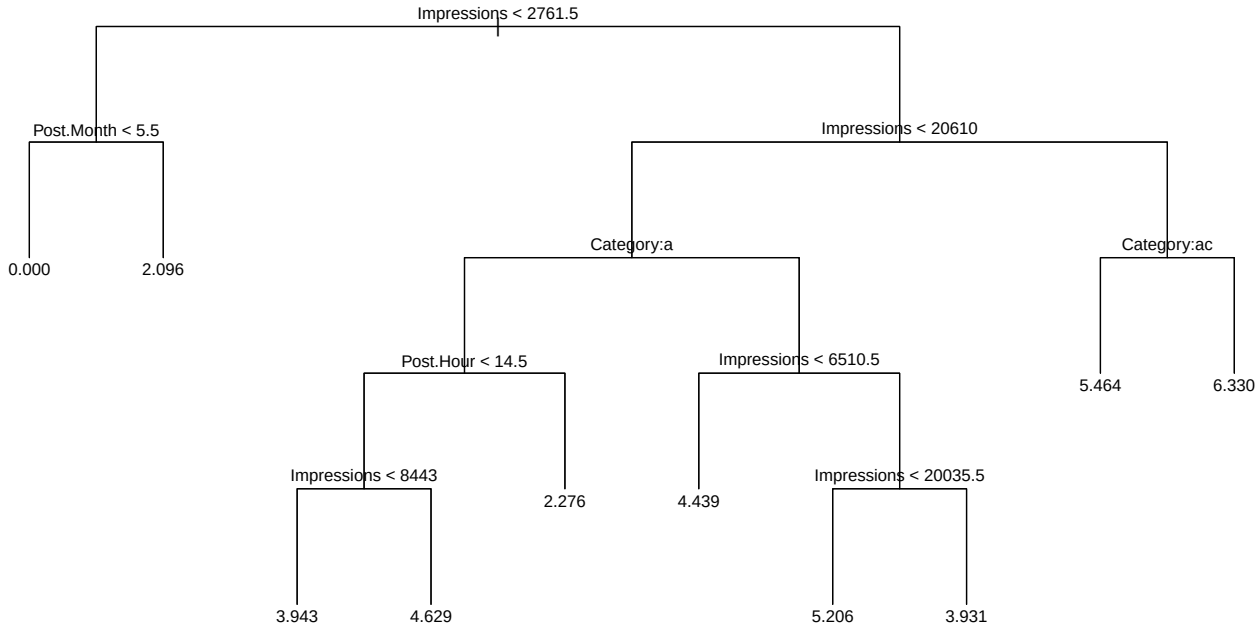

Impressions < 2722
Post.Hour < 7
Impressions < 20634.5
2.871
Page.likes < 136689
Category:a
Category:ac
2.114
0.000
Impressions < 8369
Impressions < 5604.5
Page.likes < 132775
6.545
3.988
4.640
4.335
Type:ac
5.050
5.658
4.414
5.128

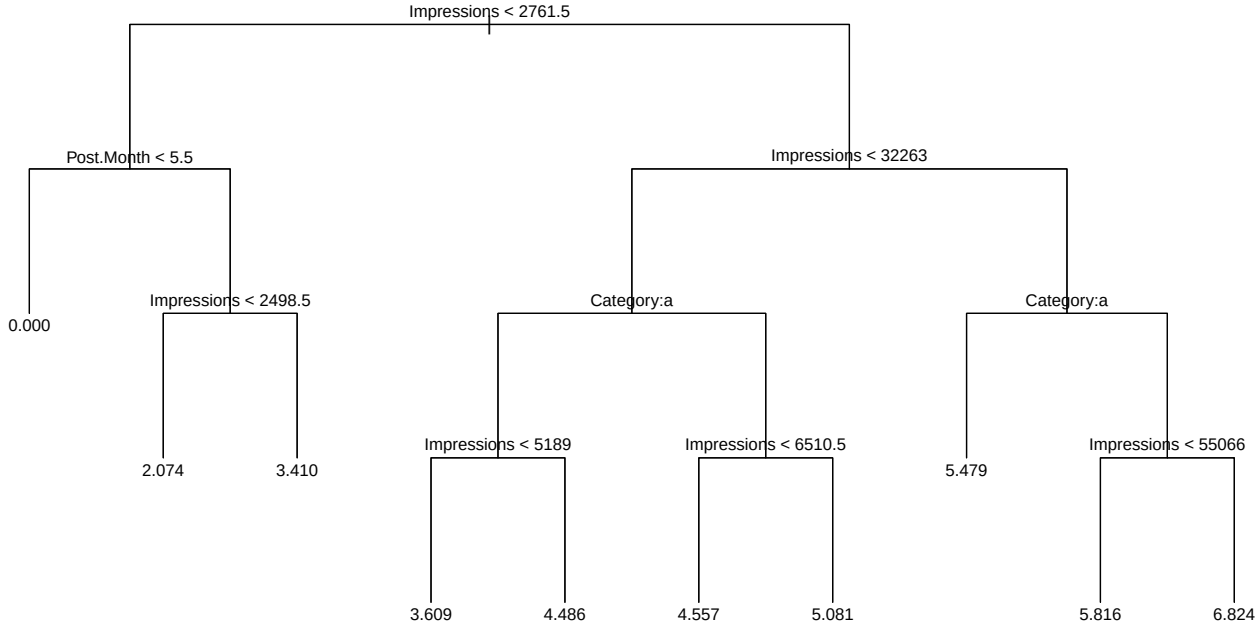

Impressions < 2757.5
Page.likes < 137476
Impressions < 27120.5
Page.likes < 118136
0.0000
Category:a
Impressions < 54990.5
0.7167
2.2660
Impressions < 7649
Impressions < 6497
Type:a
Category:a
3.7250
4.4390
4.5650
Page.likes < 132078
4.1010
5.4780
5.7250
6.6190
Type:ac
5.5100
4.1820
5.1160

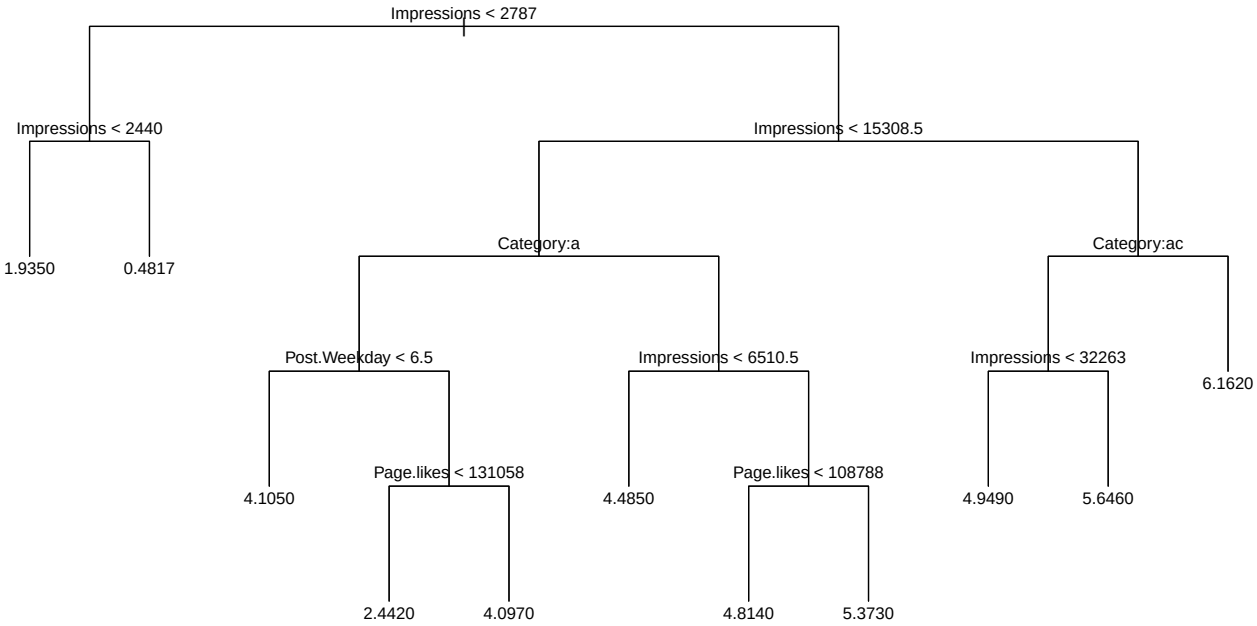

Impressions < 2722
Impressions < 2081
Impressions < 9091
2.1560
Impressions < 2509
Category:a
Impressions < 55088.5
0.3367
1.9940
Post.Hour < 13.5
4.6990
Category:a
Category:a
4.0100
2.4340
4.8840
5.4580
5.5550
6.9380

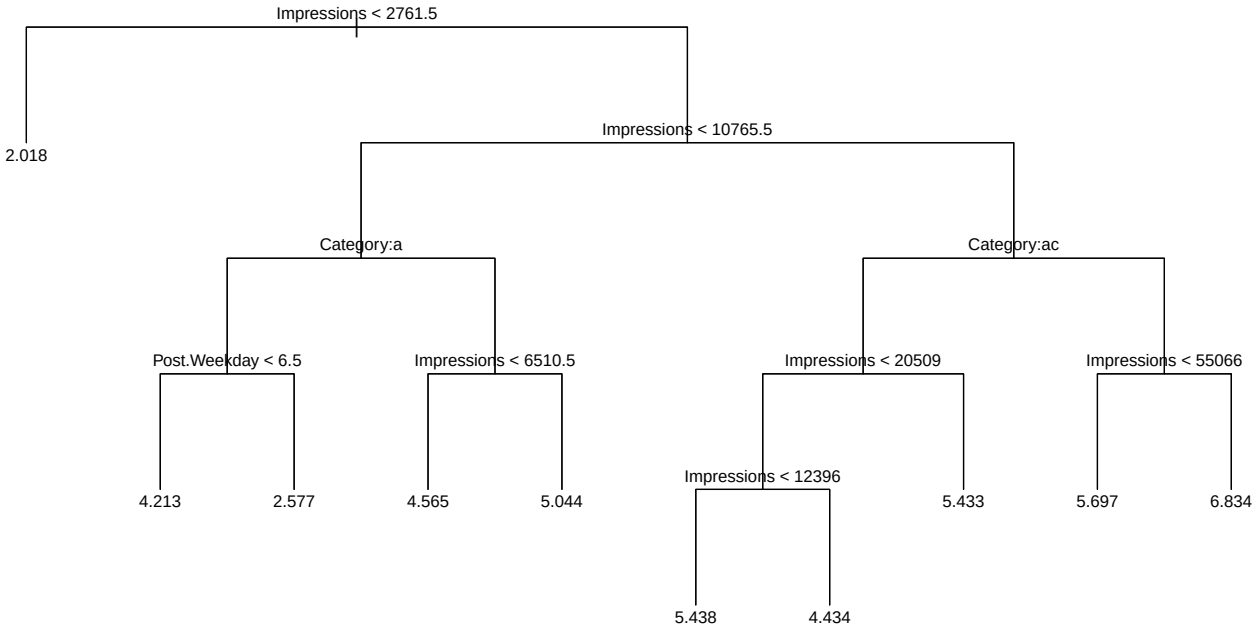

Impressions < 2722
Impressions < 2359.5
Impressions < 16294.5
2.1560
0.3613
Category:a
Category:ac
Post.Hour < 14.5
Impressions < 7030.5
Impressions < 54990.5
Post.Hour < 11.5
Impressions < 8443
1.9140
4.5670
5.1140
5.1370
5.6960
5.9200
7.2560
Post.Hour < 11.5
4.6760
4.0810
2.9080


Impressions < 2722
Paid < 0.5
Impressions < 20632
2.2030
0.9521
Category:a
Category:a
Post.Weekday < 6.5
Impressions < 6575
5.3890
Impressions < 55066
4.2470
Page.likes < 125540
4.5020
Type:ac
5.7080
6.5520
2.5770
4.2780
4.4780
5.2840
Impressions < 2761.5
Page.likes < 136689
Impressions < 19379
Impressions < 1137.5
0.000
Category:a
Category:a
1.478
2.538
4.044
Impressions < 7030.5
Impressions < 51219
Impressions < 32530.5
4.521
Type:ac
4.979
5.672
5.541
6.366
4.462
5.227


Impressions < 2761.5
Post.Month < 5.5
Impressions < 20610
0.000
2.096
Category:a
Category:ac
Post.Hour < 14.5
Impressions < 6510.5
5.464
6.330
Impressions < 8443
2.276
4.439
Impressions < 20035.5
3.943
4.629
5.206
3.931


Impressions < 2761.5
Post.Month < 5.5
Impressions < 32263
0.000
Impressions < 2498.5
Category:a
Category:a
2.074
3.410
Impressions < 5189
Impressions < 6510.5
5.479
Impressions < 55066
3.609
4.486
4.557
5.081
5.816
6.824





There a several ways in which these trees are different from each other: - different numbers of branches (and leaves) - different tree shapes - different variates defining the branches - different ordering of variates in the tree - different value used to cut the variate at any branching - different possible averages that can be taken at the leaves.

Such variation should give one some pause over interpreting the trees.

If interest lies in prediction, we could build a new predictor that was the average of the predictions from all of these trees! As with any averaging, the resulting prediction will be **less variable**!

3.2 Effect of averaging (bagging).

For any sample, we can always resample, get an estimate based on that sample, and average the predictors.

For example, first some functions

```
# It will also be handy to have a function that will return the
# newdata as a list, since that is what is expected by predict
# for that argument.
# We will write a function that takes a fitted tree (or any other fit)
# THis involves a little formula manipulation ... of interest only ...
get.newdata <- function(fittedTree, test.data){
  f <- formula(fittedTree)
  as.list(test.data[,attr(terms(f), "term.labels")])
}
#
# And a similar function that will extract the response values
# This is kind of hairy, formula manipulation ... feel free to ignore ...
get.response <- function(fittedTree, test.data){
  f <- formula(fittedTree)
  terms <- terms(f)
  response.id <- attr(terms, "response")
  response <- as.list(attr(terms, "variables"))[[response.id + 1]]
  with(test.data, eval(response))
}

get.explanatory_varnames <- function(formula){
  f <- as.formula(formula)
  terms <- terms(f)
  attr(terms, "term.labels")
}
#  The remaining functions are the important ones
#
getTrees <- function(data, formula, B=100, ...) {
  N <- nrow(data)
  Trees <- Map(function(i){
                    getTree(formula,
                            getSample(data, N),
                            ...)
                          },
               1:B
                        )
  Trees
}

# And now the function that actually calculates the average
aveTreePred <- function(Trees){
  function(newdata){
    M <- length(Trees)
    if (missing(newdata)) {
      predictions <- Map(function(tree) {
        predict(tree)
      }, Trees)
    } else {
      predictions <- Map(function(tree){
        # New data needs to be a list
        if (is.data.frame(newdata)) {
          newdata.tree <- get.newdata(tree, newdata)
```

```
        } else {
          newdata.tree <- newdata
        }
        predict(tree, newdata=newdata.tree)
      }, Trees)
    }
    # Gather the results together
    Reduce(`+`, predictions) / M
  }
}
```

These can be used as follows

```
Trees <- getTrees(facebook, formula)
treeAve <- aveTreePred(Trees)
# treeAve() would predict at the data values BUT
# there's a problem ...
```

which would work fine except that `predict.tree(...)` **does not always return the same number of values** of `y` as there were original observations. Consequently, there would be many warnings when they were added in the prediction and the predictions would be incorrect in several instances.

But you get the idea.
It turns out that we can implement this by using the recursive partioning tree algorithm as implemented in the `rpart` package. Here in place of the function `tree` we have the equivalent function `rpart(...)`. This means we need only replace the `getTree(...)` function.

```
library(rpart)
getTree <- function(formula, samp, ...){
  rpart(formula = formula, data = samp, ...)
}
```

And the following now works without generating warnings.

```
Trees <- getTrees(facebook, formula)
treeAve <- aveTreePred(Trees)
# Now do the prediction
combinedPred <- treeAve(newdata = facebook)
head(combinedPred)
```

```
##        1        2        3        4        5        6
## 4.545566 4.885136 4.538956 6.752981 5.421543 5.362613
```

Note that `rpart(...)` is very much like `tree(...)` and also has a `control` argument that can be set using the `rpart.control(...)` function. Similarly there is a `prune(...)` function for cost-complexity pruning, and an `xpred.rpart(...)` to give cross-validated predictions.

Because we are taking samples without replacement from our original sample, and these samples are the same size as the original sample, these samples are called **bootstrap** samples. Replacing our original estimate by the average over several bootstrap samples is also sometimes called **bagging** (for **bootstrap aggregation**).

The idea is straightforward and the additional jargon seems gratuitous.

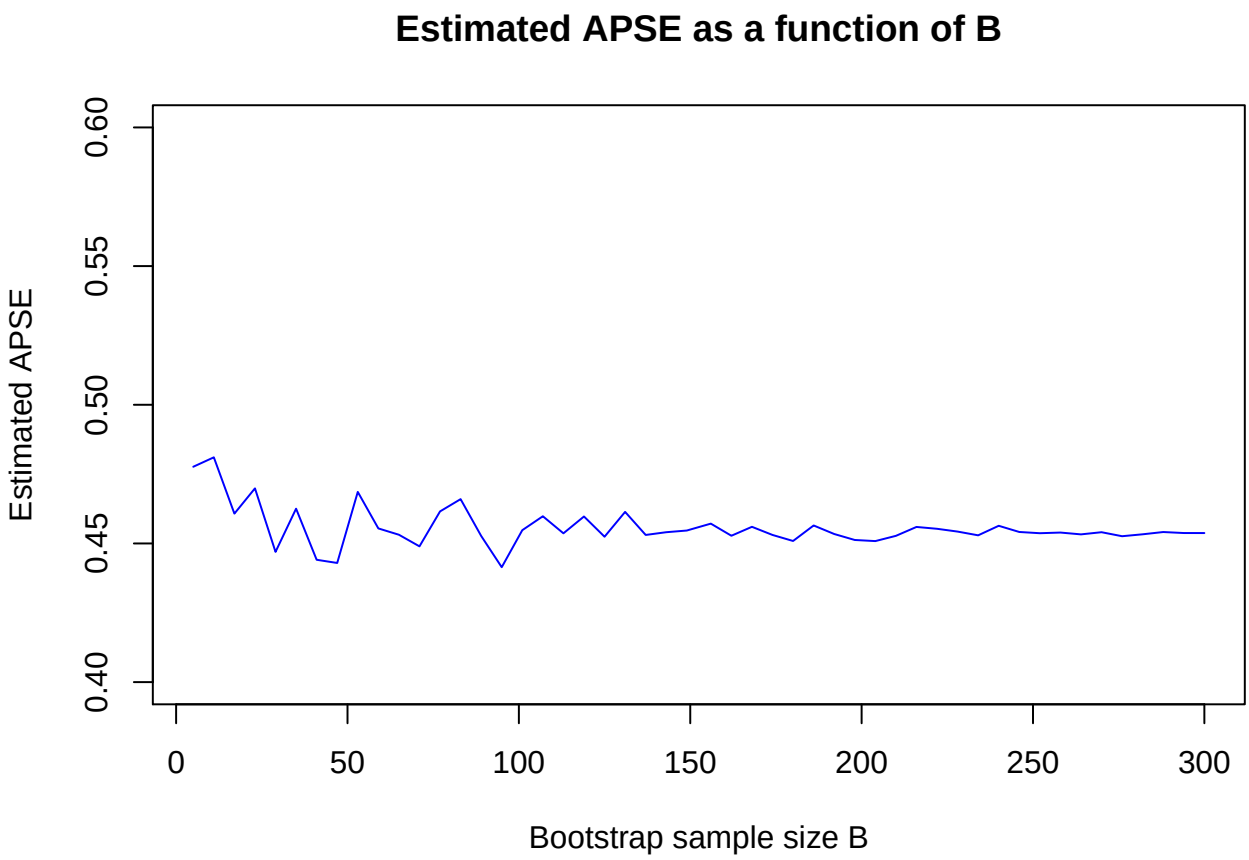
How large should B be? To get some sense of how many samples we should take in our average, we can split the data into a training and a test sample, and look at the average predicted squared error as a function of the number of samples.

```
N <- nrow(facebook)
N_train <- round(2* N /3)
N_test <- N - N_train
```

```
# Select the two sample units
id.train <- sample(1:N, N_train, replace=FALSE)
id.test <- setdiff(1:N, id.train)
# Split the data into training and test sets.
fb.train <- facebook[id.train,]
fb.test <- facebook[id.test,]
```

We can now build a number of trees on the training set and evaluate the average predicted squared error on the test set. We want to take B, the number of trees, to be large enough that the average predicted squared error has settled down.

```
# For the notes, we set the seed for reproduceability
set.seed(135232327)
# Get a lot of trees based on the training data
Trees <- getTrees(fb.train, formula, B=300)

# Now we will get the average squared predicted error for the average of these
# trees for varying numbers of trees.
Num.trees <- round(seq(5,length(Trees), length.out=50))
maxNum.trees <- length(Trees)
#

treeAve.apse <- Map(function(B) {
  # Let's sample B trees from all of the trees available
  treeAve.B <- aveTreePred(Trees[sample(1:maxNum.trees, B, replace = FALSE)])
  pred.B <- treeAve.B(newdata = fb.test)
  actual.B <- get.response(Trees[[1]], fb.test)
  error.B <- actual.B - pred.B
  mean(error.B^2)
},
Num.trees)
#
# And plot the results
plot(Num.trees, treeAve.apse, type="l", main="Estimated APSE as a function of B",
     ylab="Estimated APSE", xlab="Bootstrap sample size B", ylim=c(0.4, 0.6),
     col="blue")
```



The average predicted squared error is pretty stable by $B = 150$. We would choose B to be a value where stability was reached.

4 Multiple trees

Regression trees are easy to build, to apply, and to interpret as simple decision trees.

Their performance, however, can leave something to be desired. They will certainly be poorer predictors than the various smoothers (including for multiple explanatory variates) whenever the true underlying function $\mu(...)$ is relatively smooth at least for the same complexity. Conversely, if $\mu(...)$ is not smooth but changes abruptly then regression trees should outperform smoothers in such situations.

Regression trees can also be highly variable, as we saw with the Facebook data. Being a greedy algorithm, it is also possible that a quite different tree will produce as good or even better fit than the one we got. Both points should give us a little pause when interpreting the meaning of various branches in the tree.

4.1 Bagging (again)

Building B regression trees, each based on a bootstrap sample from the original data, we average the predictions of all B trees as described in the section on averaging trees. As was mentioned there, this is called "bootstrap aggregation" or "bagging".

As with all other estimators, we can evaluate the average prediction squared error ($APSE$) by comparing the predictions with the actual values on a test sample (as was done when bagging was introduced). We could also estimate the $APSE$ via k-fold cross-validation. A problem with cross-validation however, especially with prediction functions that average over 100s of trees, is its computational cost.

Trees constructed from bootstrap samples, however, present an opportunity to estimate their prediction errors **as they are being built**. The basic idea is that each observation will not (with high probability) appear in all bootstrap samples. That being the case, for any observation, $i = 1, \ldots, N$, we can take the average prediction of all trees from those bootstrap samples that do **not** contain i. The difference between the actual value for i and its predicted value (based on the all bootstrap samples excluding i) is its contribution (when squared) to the $APSE$.

This is sometimes called **out of bag estimation** estimation of $APSE$. We can get some idea as how the might be done via the following piece of code:

```
data <- facebook
N <- nrow(data)
# Select the two sample units
id.all <- 1:N
# Some vectors to gather the results
count <- numeric(length=N)
pred <- numeric(length=N)
B <- 300

for (i in 1:B) {
  # For each bootstrap sample
  id.bootstrap <- sample(id.all, N, replace=TRUE)
  id.inbag <- unique(id.bootstrap)
  id.outofbag <- setdiff(id.all, id.inbag)
  # Split the data into training and test sets.
  samp.boot <- data[id.inbag,]
  test.boot <- data[id.outofbag,]

  # Fit the tree on the bootstrap sample
  tree.boot <- rpart(formula = formula, data = samp.boot) #, ...)
  # Evaluate the prediction on the "out of bag" sample
  newdata.outofbag <- get.newdata(tree.boot, test.boot)
  pred.outofbag <- predict(tree.boot, newdata = newdata.outofbag)

  # Update the out of bag predictions and add to the count of predictions
  pred[id.outofbag] <- pred[id.outofbag] + pred.outofbag
  count[id.outofbag] <- count[id.outofbag] + 1
}

# Now determine the bagged predictions for each observation:
pred.bagged <- pred/count

# Each prediction is then based on approximately the
# following number of bootstrap trees:
mean(count)
```

```
## [1] 110.562
```

```
# Or about the following proportion of the bootstrap trees
# when into each observation's prediction:
mean(count/B)
```

```
## [1] 0.36854
```

```
#
# Now determine the "out of bag" estimate of the squared error
# Get the actual responses (using the formula from the last tree.boot)
```

```
actual.values <- get.response(tree.boot, data)
squared.error <- mean((actual.values - pred.bagged)^2)

# And the estimated "out of bag" APSE  is
squared.error
```

```
## [1] 0.520971
```

A little more careful coding and we could have B grow in size as well, all the while accumulating a sequence of total out of bag predictions for the sequence of B values.

As could be seen above (with `mean(count)/B` = 0.36854, about a third of the trees are aggregated to predict each observation. This is because, roughly speaking, each bootstrap tree uses only about 2/3 of the data, and so leaves about 1/3 for a test set.

Out of the bag estimation of the error can be shown to be nearly equivalent to "leave one out cross validation" estimates if B is large enough.

4.2 Random Forests

Stretching our tree metaphor a little, we can think of bagging as having produced a **forest** of trees. Each tree in this forest was produced by recursive partitioning of a bootstrap sample of the data. The average prediction from this forest of these trees has smaller variability than does a single tree.

Recall that if we have independent random variates $Y_1, \ldots, Y_B$ with variance σ^2, that the variance of the average $\overline{Y}$ is σ^2/B. Unfortunately, if the Y_is are not independent, are at least uncorrelated, then this is no longer the variance. Recall that if Y_i and Y_j have non-zero correlation $\rho = Cov(Y_i, Y_j)/\sigma^2$, then

$$Var(Y_1 + Y_2) = Var(Y_1) + Var(Y_2) + 2Cov(Y_1, Y_2) = 2\sigma^2(1 + \rho).$$

The trees produced by bootstrap samples have many observations in commonand the predictions they produce will be correlated.

To break down this correlation, we can introduce a little more randomness into the construction of the trees. Recall that in our greedy algorithm for tree construction, we examined **all p explanatory variates at each partition**, and selected the best amongst them. This allowed some variates to dominate more than others at each split, especially early on (nearer the root) in the tree growing. Recall, for example, how often `Impressions` and the `Action` category (`Category:a` in the tree) appeared in the bootstrap samples of trees from the facebook data.

Instead of every explanatory variate being allowed to be examined at every potential partition, what if we only allowed **only m explanatory variates selected at random at each partition**? This would mean that some variates were simply not considered at any step, no matter how much they might reduce the RSS then. This allows variates to appear in a tree which might never have appeared before. This is especially true early in the tree where the greedy algorithm will tend to favour the same explanatory variate for many bootstrap samples.

We proceed as before, selecting B bootstrap samples. But now for each sample, we build a tree using the same greedy algorithm but now selecting only from a random subset of m explanatory variates, a different random subset at each step. The collection of trees constructed in this way is called a **random forest**. As with bagging, we take as our prediction the average of the predictions over the B regression trees. Note that if $m = p$, then this is just bagging. That is **bagging is a special case of a random forest**.

How might we choose m? A common rule used in practice is to choose $m \approx p/3$ (people recommend $\sqrt{p}$ for classification trees).

In `R` there is a package called `randomForest` that provides us a tool set for constructing random forest regression estimates. Note that `getTree(...)` is a function in the `randomForest` package, so we may want

to remove the function that we created with that name from your R session. You can remove our function as follows

```
rm(getTree, envir = globalenv())
```

Now there will be no conflict with the `getTree(...)` function from `randomForest`. Alternatively, you could choose not to remove our `getTree(...)` function in which case you would have to write `randomForest::getTree(...)` whenever you wanted to call the `getTree(...)` function from the `randomForest` package.

We can illustrate this again on the Facebook data.

4.2.1 Bagging (random forest with $m = p$)

First we will **use random forests to do bagging** simply by choosing $m = p$.

```
library(randomForest)
#
set.seed(54321)
#
fb.bag <- randomForest(log(All.interactions +1) ~
                               Page.likes + Paid +
                               Post.Hour + Impressions +
                               Post.Weekday + Post.Month +
                               Type + Category,
                       data = facebook,
                       # Number of variates to select at each step
                       mtry = 8,
                       # facebook has NAs and randomForest will
                       # fail unless they are dealt with
                       na.action = na.omit)
fb.bag
```

```
##
## Call:
##  randomForest(formula = log(All.interactions + 1) ~ Page.likes +     Paid + Post.Hour + Impressions + Post
##                Type of random forest: regression
##                      Number of trees: 500
## No. of variables tried at each split: 8
##
##           Mean of squared residuals: 0.4824976
##                     % Var explained: 66.71
```

```
# Can predict as before
head(predict(fb.bag))
```

```
##        1        2        3        4        5        6
## 4.736613 4.714626 4.759624 6.806932 5.807787 5.152008
```

```
# Let's run this by separating a training set and test set as before
# It will be easier if we remove all NAs for now.
fb <- na.omit(facebook)
N <- nrow(fb)
N_train <- round(2* N /3)
N_test <- N - N_train
# Select the two sample units
id.train <- sample(1:N, N_train, replace=FALSE)
```

```
id.test <- setdiff(1:N, id.train)

fb.bag <- randomForest(log(All.interactions +1) ~
                               Page.likes + Paid +
                               Post.Hour + Impressions +
                               Post.Weekday + Post.Month +
                               Type + Category,
                       data = fb,
                       subset = id.train,
                       # Number of variates to select at each step
                       mtry = 8)

test <- fb[id.test,]
pred <- predict(fb.bag,
                # Note that our get.newdata will also work on our
                # formula string
                newdata = get.newdata(formula, test))
actual <- get.response(formula, test)
# estimated APSE
mean((actual-pred)^2)
```

```
## [1] 0.5083341
```

4.2.2 Random forests ($m << p$)

We can now introduce a forest where the trees are more uncorrelated (and the predictions) by choosing $m << p$. For `randomForest(...)`, the default is $p/3$

```
#
fb.rf <- randomForest(log(All.interactions +1) ~
                               Page.likes + Paid +
                               Post.Hour + Impressions +
                               Post.Weekday + Post.Month +
                               Type + Category,
                       data = fb,
                       subset = id.train,
                       # Number of variates to select at each step
                       mtry = 3)

pred <- predict(fb.rf,
                # Note that our get.newdata will also work on our
                # formula string
                newdata = get.newdata(formula, test))
actual <- get.response(formula, test)
# estimated APSE
mean((actual-pred)^2)
```

```
## [1] 0.4781065
```

4.2.3 Measuring importance of explanatory variates

A problem with averaging predictions over a forest of trees is that we have lost the ease of interpretation given by a single tree. One way around this is to consider how the total amount that the RSS has been reduced in a single tree due to splits on that variate. Each variate will have reduced the RSS by some

amount (including zero) for every tree in our forest. For each variate, we simply sum its RSS reduction over all B trees in the forest and divide by the number of trees (B) to get its average contribution, and hence importance.

For example, for our random forest estimator we can ask that the importance of each variate be recorded with the estimate.

```
fb.rf <- randomForest(log(All.interactions +1) ~
                                Page.likes + Paid +
                                Post.Hour + Impressions +
                                Post.Weekday + Post.Month +
                                Type + Category,
                      data = fb,
                      importance = TRUE,
                      subset = id.train,
                      # Number of variates to select at each step
                      mtry = 3)
```

And now look at the effect on the RSS for each variate.

```
# The type = 2 importance records the drop in RSS
importance(fb.rf, type=2)
```

```
##              IncNodePurity
## Page.likes       40.374791
## Paid              6.587544
## Post.Hour        30.164212
## Impressions     257.308814
## Post.Weekday     24.938350
## Post.Month       21.008727
## Type             15.277954
## Category         43.003517
```

From which we can see that `Impressions` and `Page.likes` appear to be the most important variates in our random forest estimate.

Alternatively, `randomForest(...)` also makes use of "out of bag" estimates to arrive at another measure of importance.

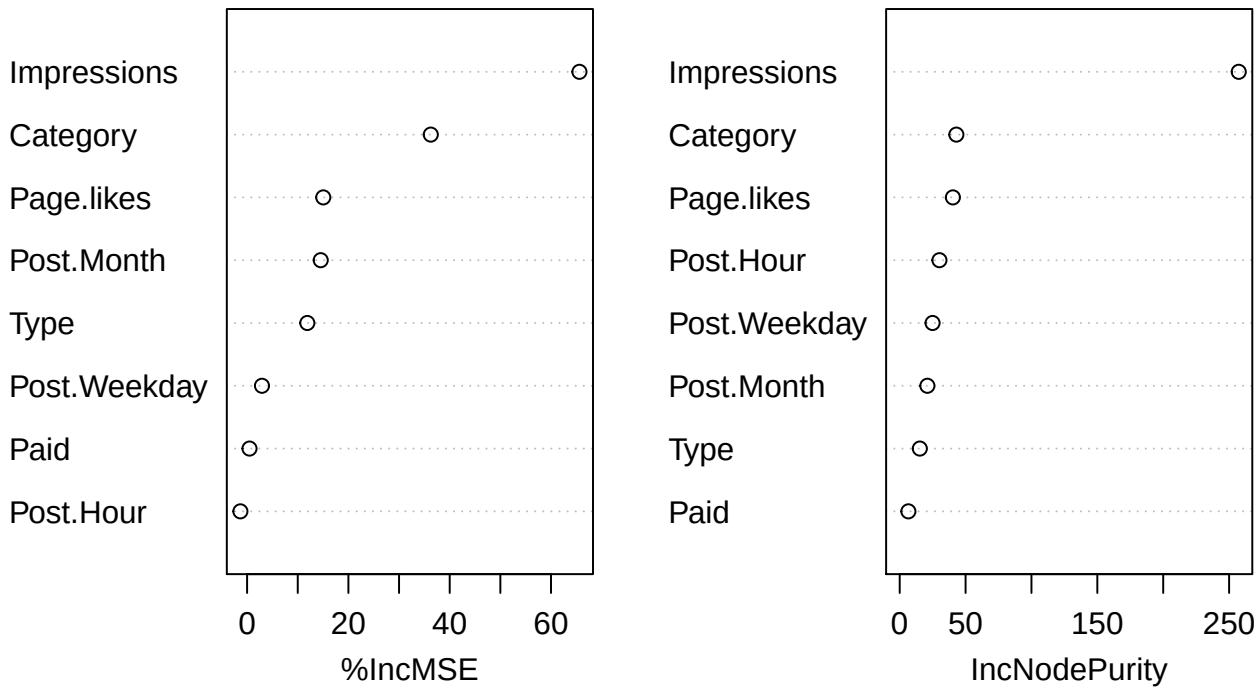
```
# The type = 1 importance records average decrease in the accuracy of predictions
importance(fb.rf, type=1)
```

```
##                %IncMSE
## Page.likes   15.0351988
## Paid          0.4748268
## Post.Hour    -1.3251245
## Impressions  65.6109403
## Post.Weekday  2.9357690
## Post.Month   14.5325205
## Type         11.8864819
## Category     36.2622327
```

This is based on the average decrease in the accuracy of predictions on "out of bag" samples when that variate is **excluded** from the model. We will work primarily with the easier to comprehend `type=1` measure of importance.

There is also a plotting function that displays these two measures of variate importance.

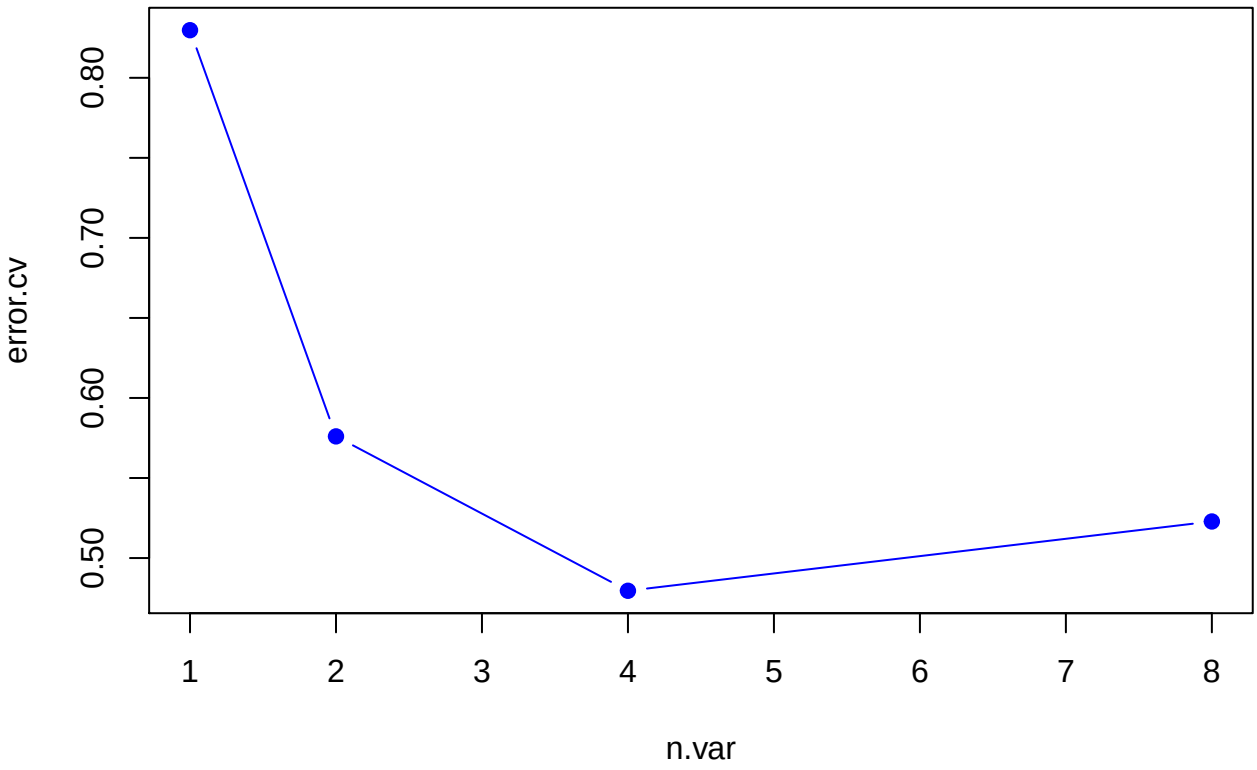
```
varImpPlot(fb.rf)
```



Which suggests that `Impressions` are (not surprisingly) far and away the most important variate.

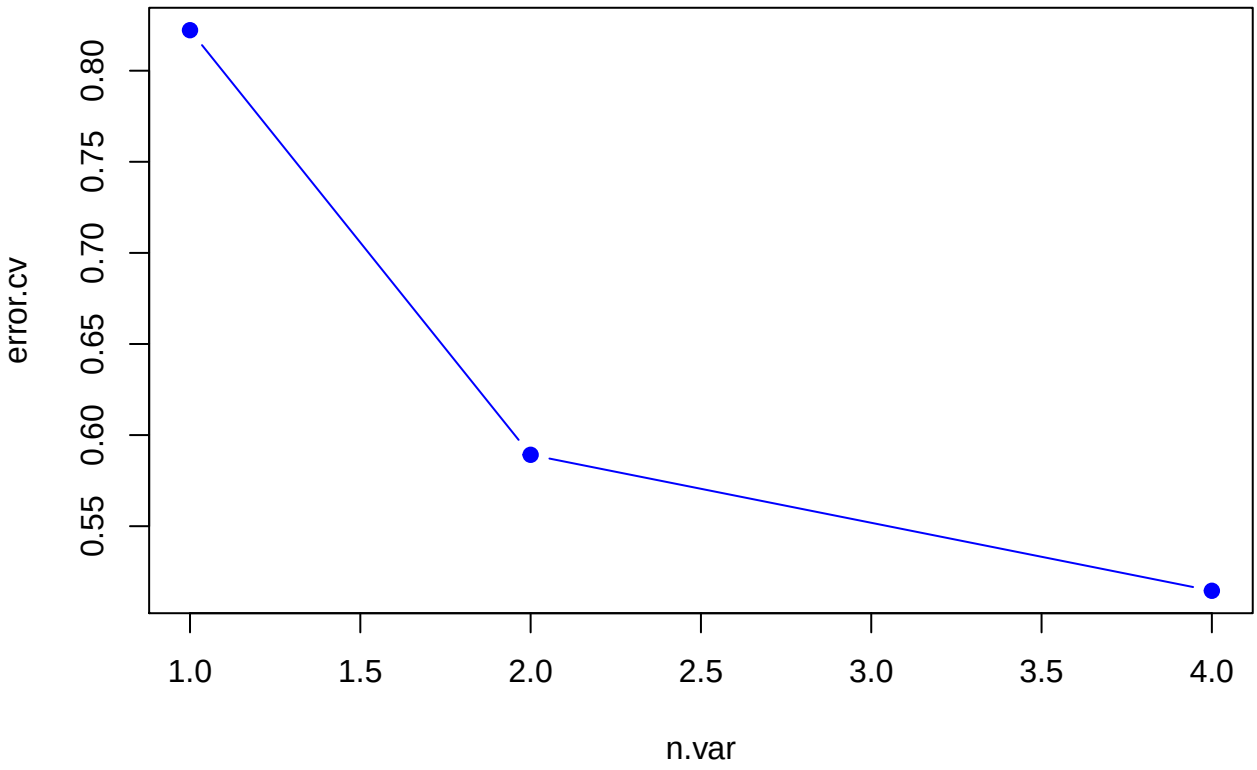
The `randomForest` package also offers the possibility of using k-fold cross-validation to determine the $APSE$ as a function of the number of variates used in the trees.

```
# Need separate response and explanatory variate data frames
trainy <- log(fb[,"All.interactions"]+ 1)
trainx <- fb[, get.explanatory_varnames(fb.rf)]
# Five fold cross-validataion
fb.rfcv <- rfcv(trainx = trainx, trainy = trainy, cv.fold = 5)
# We can plot the results
with(fb.rfcv, plot(n.var, error.cv, pch = 19, type="b", col="blue"))
```



which seems to suggest using only four variates in the fit. Choosing based on RSS, we might choose to use `Impressions`, `Page.likes`, `Category`, and `Post.Hour`.

```
fb.rfcv.4 <- randomForest(log(All.interactions +1) ~
                          Impressions + Page.likes + Category + Post.Hour,
                        data = fb,
                        importance = TRUE)

trainx <- fb[, get.explanatory_varnames(fb.rfcv.4)]
# Five fold cross-validataion
fb.rfcv <- rfcv(trainx = trainx, trainy = trainy, cv.fold = 5)
# We can plot the results
with(fb.rfcv, plot(n.var, error.cv, pch = 19, type="b", col="blue"))
```

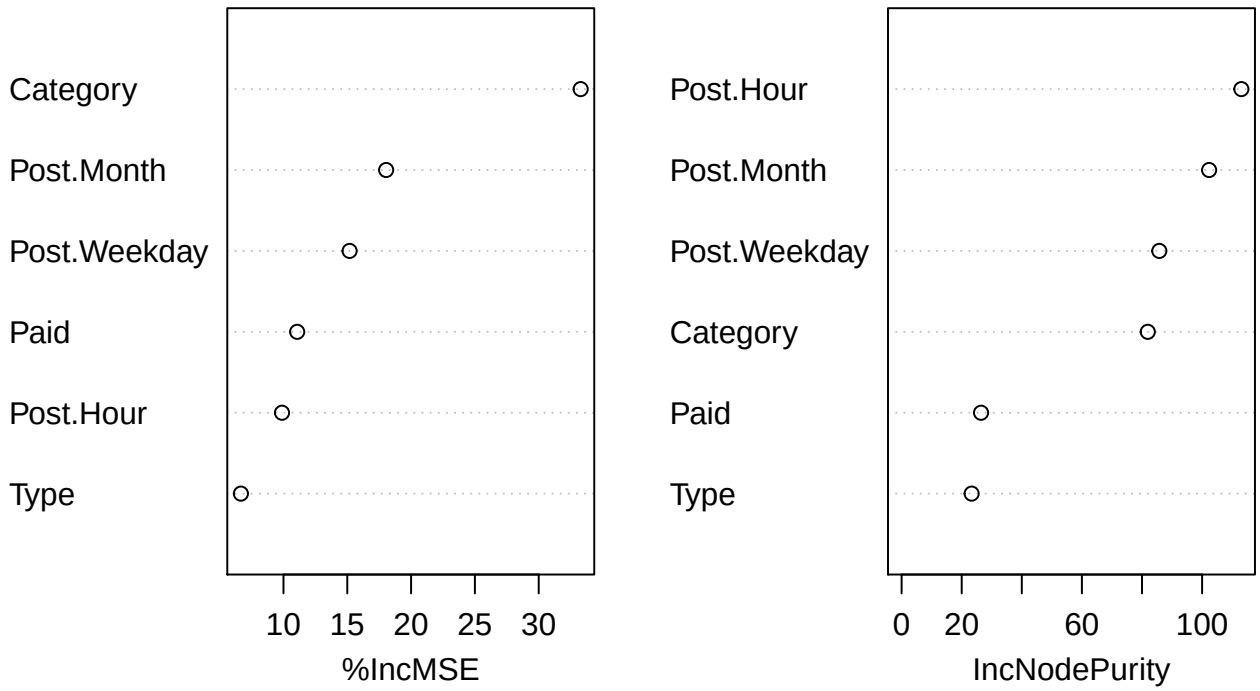


which again suggests these four variates.

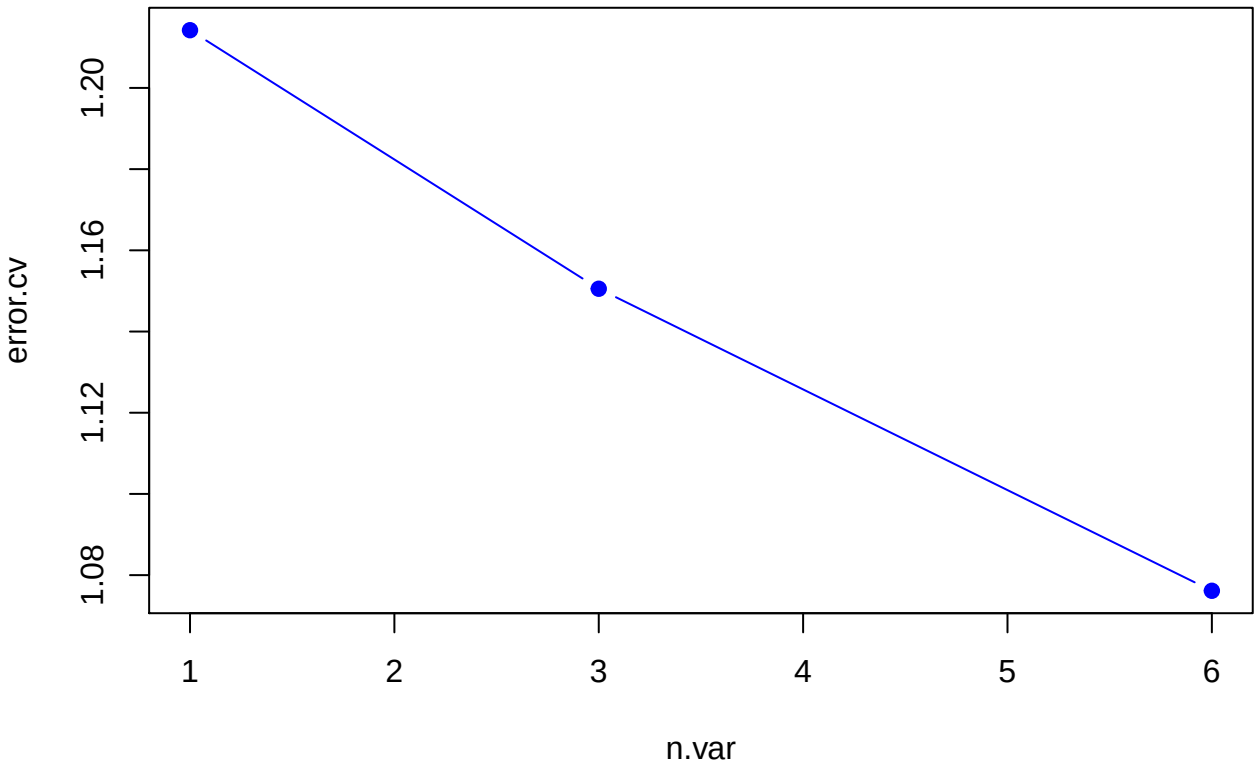
Note As remarked earlier, a more sensible modelling exercise would consider how `All.Interactions` might depend on just those explanatory variates that were under the control of the cosmetic company itself.

This suggests the following summary analysis:

```
fb.rfe <- randomForest(log(All.interactions +1) ~
                         Paid + Post.Hour +
                         Post.Weekday + Post.Month +
                         Type + Category,
                       data = fb,
                       importance = TRUE)
# Note the default values used above plus the entire data set
varImpPlot(fb.rfe)
```




```
trainx <- fb[, get.explanatory_varnames(fb.rfe)]
# Five fold cross-validataion
fb.rfcv <- rfcv(trainx = trainx, trainy = trainy, cv.fold = 5)
# We can plot the results
with(fb.rfcv, plot(n.var, error.cv, pch = 19, type="b", col="blue"))
```



This is at least suggesting that `Type` and whether advertising was `Paid` or not are the least important. For predictive purposes, `Category` seems to be the most important. The cross-validation suggests all six variates remain in the model.

Even with these tools, much has been lost in interpretability compared to a single tree.

4.3 Boosting

So far we have constructed a forest of trees (via bootstrap samples and some random selection of the variates to consider at each split) and then aggregated the tree predictions by averaging.

Rather than simple averages, we might consider some other linear combination (or weighting) of the predictions of a number of trees. That is, suppose we have B different prediction functions $\widehat{\mu}_m(x)$ for $b = 0, 1, \ldots, M$. Then we might construct a combined prediction estimate:

$$\widehat{\mu}(x) = \sum_{m=0}^{M} \beta_m \widehat{\mu}_m(x)$$

where the coefficients $\beta_0, \beta_1, \ldots, \beta_M$ are parameters of our choice. Random forests (including bagging) are a special case where $\beta_m = \frac{1}{B}$ for all m and $B = M + 1$.

Combining prediction estimates in this way is sometimes called **boosting**, the idea being that any individual prediction estimate $\widehat{\mu}_m(x)$ may not be that great but that together they "boost" one another's performance. (In machine learning, the functions $\widehat{\mu}_m(x)$ are sometimes thought of as "weak learners" that are "boosted" when combined.)

The most natural way to think of the "boost" metaphor is to imagine that the final estimate is constructed sequentially. Imagine that we have an initial estimate $\beta_0\widehat{\mu}_0(x)$ and a sequence of "steps" or "boosts" $\{\beta_m\widehat{\mu}_m(x)\}_{m=1}^{M}$. Each $\beta_m\widehat{\mu}_m(x)$ is a "boost" added to the previous estimates where $\beta_m\widehat{\mu}_m(x)$ can be constructed after looking at any or all of the preceding estimates. The final estimate is additive in the "boosts"; it is the sum of all the "boosts".

The simplest sort of boosts are those where β_m is some constant λ. In this case, boosting can be made reminiscent to shrinkage smoothers when applied to regression trees. The idea is simply that if $\widehat{\mu}_m(x)$ is a regression tree estimator, then we "shrink" its effect by scaling it by $\lambda << 1$.

We can put this together in an algorithm as follows. First some notation. Suppose our training data is written as $\{y_i, \mathbf{x}_i\}_{i \in \mathcal{S}}$ where $\mathcal{S}$ is the training sample. We also suppose that our prediction function $\widehat{\mu}_m(\mathbf{x})$ is a regression tree estimate having say d splits ($d+1$ leaves) and fitted to some data set.

1. **Initialization**: Set $\widehat{\mu}(\mathbf{x}) = 0$ and $\widehat{r}_i = y_i - 0$ for all $i \in \mathcal{S}$
2. **Boost**: For $m = 1, 2, \ldots, M$, repeat:
   a. Fit $\widehat{\mu}_m(\mathbf{x})$ to training data $\{\widehat{r}_i, \mathbf{x}_i\}_{i \in \mathcal{S}}$
   b. Update $\widehat{\mu}(\mathbf{x})$ by adding a small "boost"

$$\widehat{\mu}(\mathbf{x}) \leftarrow \widehat{\mu}(\mathbf{x}) + \lambda \widehat{\mu}_m(\mathbf{x})$$

   c. Update the residuals (i.e. the training data):

$$\widehat{r}_i \leftarrow \widehat{r}_i - \lambda \widehat{\mu}_m(\mathbf{x})$$

3. **Return** the "boosted" prediction function

$$\widehat{\mu}(\mathbf{x}) = \sum_{m=1}^{M} \lambda \widehat{\mu}_m(\mathbf{x}).$$

It's not a lot of work to code this up.

```
boostTree <- function(formula, data,
                      lam=0.01, M = 10,
                      control=rpart.control(), ...) {

  # Break the formula into pieces
  formula.sides <- strsplit(formula, "~")[[1]]
  response.string <- formula.sides[1]
  rhs.formula <- formula.sides[2]
  # Construct the boost formula
  bformula <- paste("resid", rhs.formula, sep=" ~ ")

  # Initialize the resid and explanatory variates
  resid <- get.response(formula, data)
  xvars <- get.newdata(formula, data)

  # Calculate the boostings
  Trees <- Map(
    function(i) {
      # update data frame with current resid
      rdata <- data.frame(resid=resid, xvars)

      # Fit the tree
      tree <- rpart(bformula, data = rdata, control=control, ...)

      # Update the residuals
      # (Note the  <<- assignment to escape this closure)
      resid <<- resid - lam * predict(tree)

      # Return the tree
      tree }
    , 1:M)
```

```
  # Return the boosted function
  function(newdata){
    if (missing(newdata)) {
      predictions <- Map(function(tree) {
        # Boost piece
        lam * predict(tree)
      }, Trees)
    } else {
      predictions <- Map(function(tree){
        # New data needs to be a list
        if (is.data.frame(newdata)) {
          newdata.tree <- get.newdata(tree, newdata)
        } else {
          newdata.tree <- newdata
        }
        # Boost piece
        lam * predict(tree, newdata=newdata.tree)
      }, Trees)
    }
    # Gather the results together
    Reduce(`+`, predictions)
  }
}
```

Note that the above could easily be adapted to pretty much any other predictor (not just trees).

Let's see how well this works. It's as straightforward as

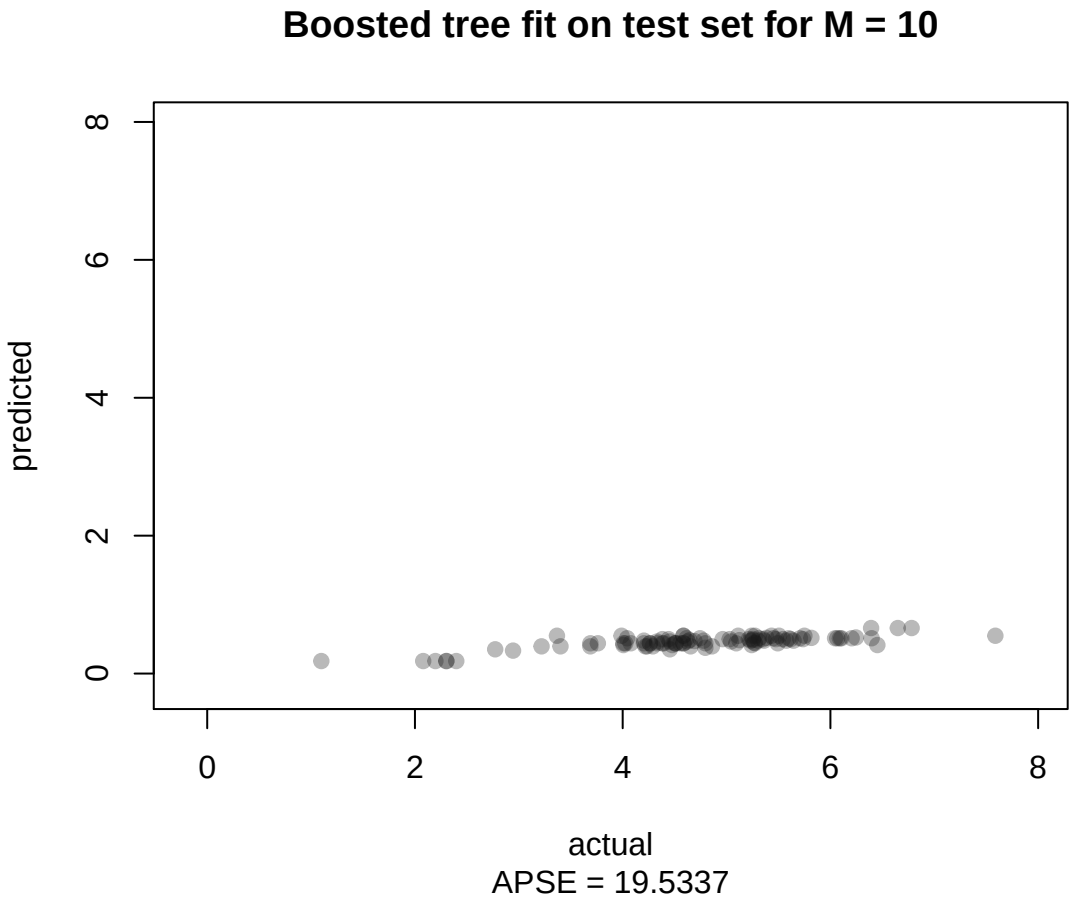
```
# Get some test and training sets
N <- nrow(facebook)
N_train <- round(0.8 * N)
N_test <- N - N_train
# Select the two sample units
id.train <- sample(1:N, N_train, replace=FALSE)
id.test <- setdiff(1:N, id.train)
# Split the data into training and test sets.
fb.train <- facebook[id.train,]
fb.test <- facebook[id.test,]

# Get the boosted function
predict.boostedTree <- boostTree(formula, data=fb.train, M=100, lam = 0.01)
# and use it to predict
pred <- predict.boostedTree(fb.test)
# results:
head(pred)
```

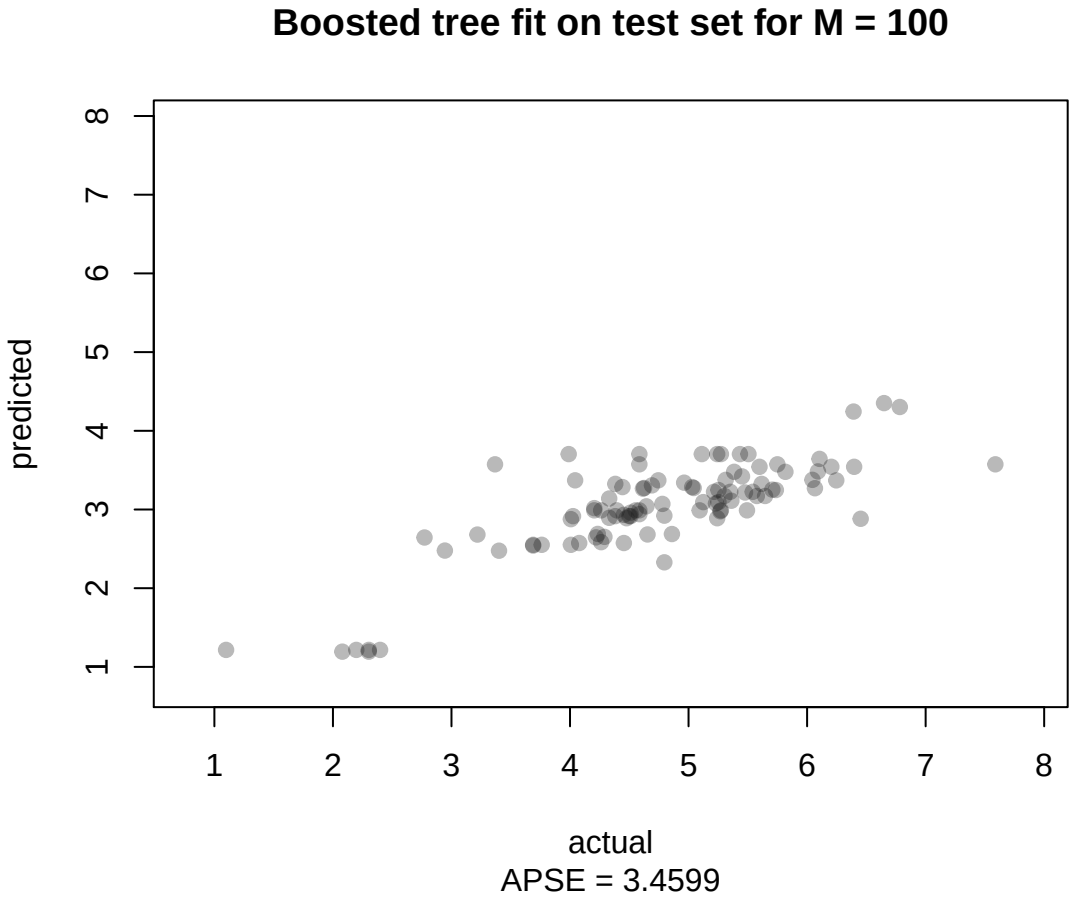
```
##        1        2        3        4        5        6
## 2.988956 3.340890 3.542262 3.306485 2.551246 3.139519
```

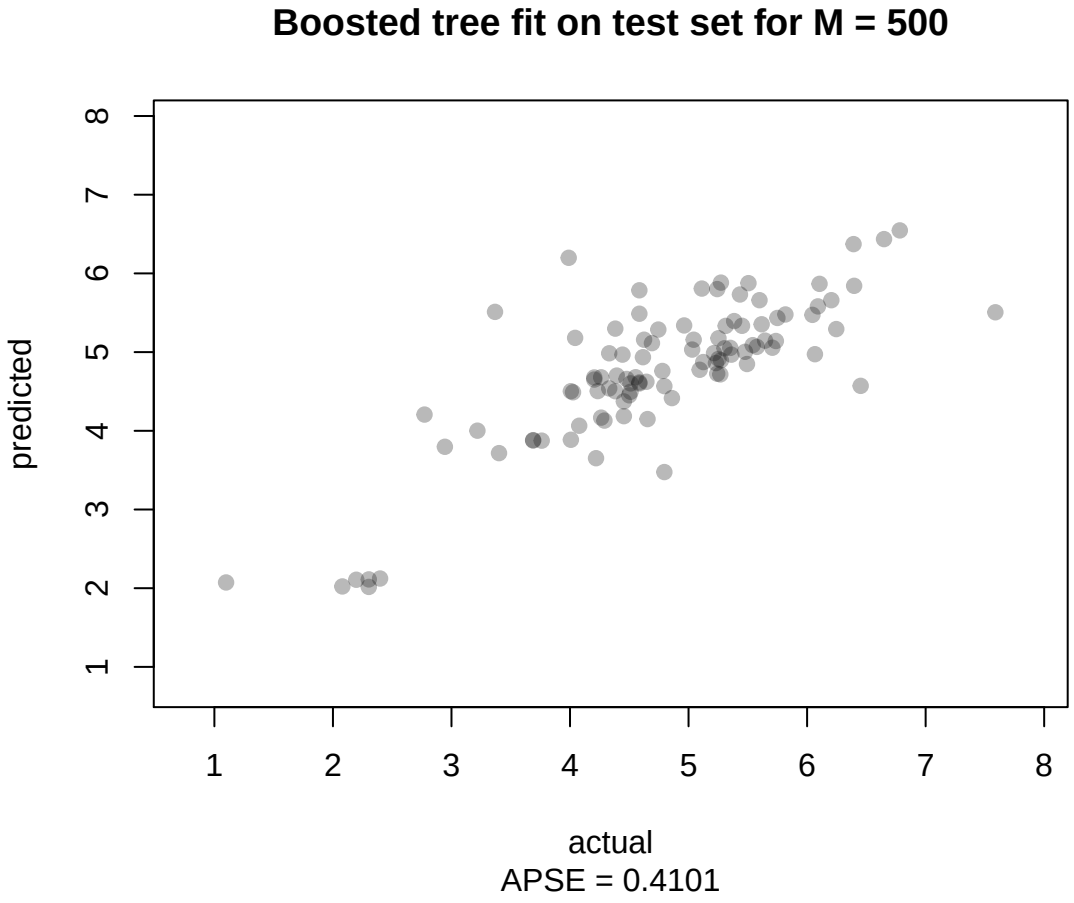
Let's see how the predictions fare for a variety of M values.

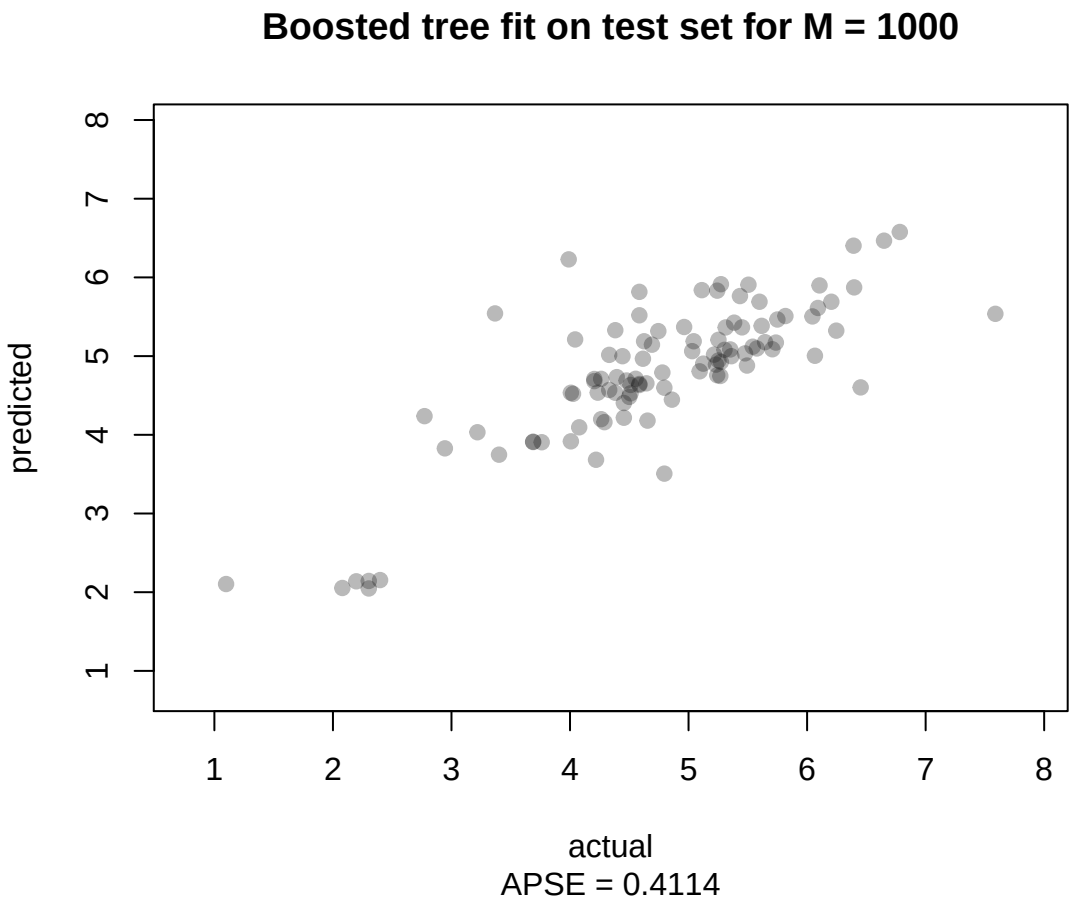
```
actual.test <- get.response(formula, fb.test)
# Now do it for varying M
for (M in c(10, 100, 500, 1000)) {
  # Get the boosted tree
  predict.boostedTree <- boostTree(formula, data=fb.train, M=M, lam = 0.01)
```

```
  # Use it to predict
  pred <- predict.boostedTree(fb.test)
  # Now see how well it does
  lims <- extendrange(c(actual.test, pred))
  plot(get.response(formula, fb.test), pred, pch=19,
       col = adjustcolor("grey10", 0.3),
       xlim=lims, ylim=lims, xlab="actual", ylab="predicted",
       main = paste("Boosted tree fit on test set for M =", M),
       sub = paste("APSE =", round(mean((actual.test - pred)^2), 4) )
      )
}
```




Boosted tree fit on test set for M = 100
predicted
1
2
3
4
5
6
7
8
actual
APSE = 3.4599

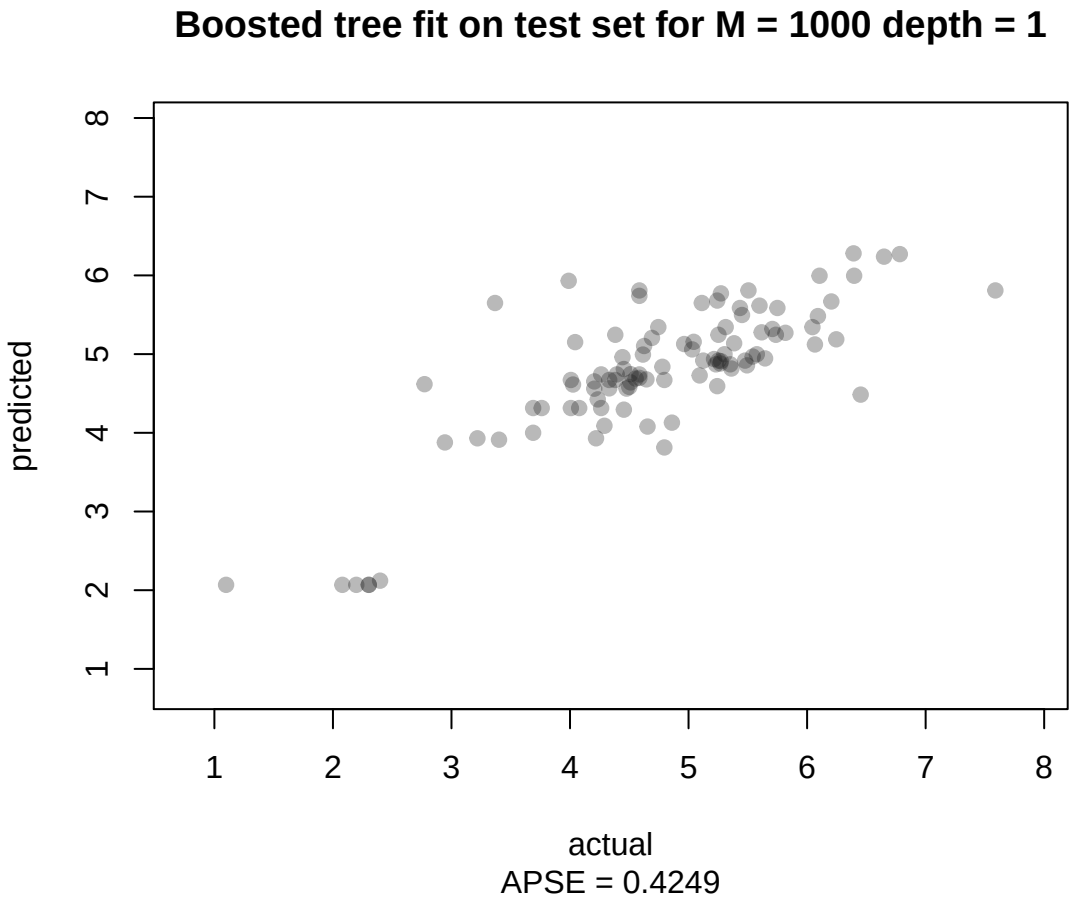

Boosted tree fit on test set for M = 500
predicted
1
2
3
4
5
6
7
8
actual
APSE = 0.4101

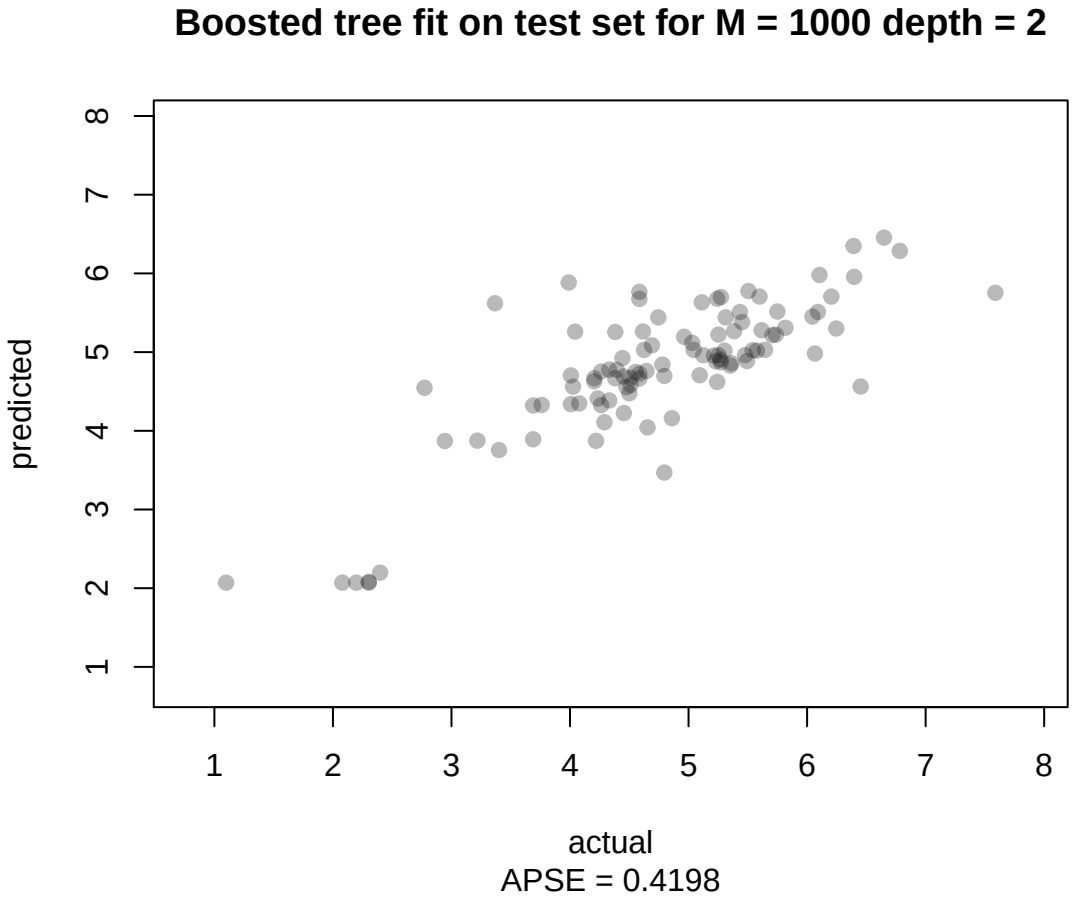


There are several choices to be made here, M, d, and λ. Typically, people take $\lambda \approx 0.01$ as above. As the plots show, the predictions improve as M increases but beyond some point there is diminishing returns.

Boosted trees are sometimes made intentionally simple (small d). We can adust this using the `control` parameter.

```
# Now do it for varying depth
for (max.depth in c(1, 2)) {
  # Get the boosted tree
  predict.boostedTree <- boostTree(formula, data = fb.train,
                                   M = 1000, lam = 0.01,
                                   control = rpart.control(maxdepth = max.depth))
  # Use it to predict
  pred <- predict.boostedTree(fb.test)
  # Now see how well it does
  lims <- extendrange(c(actual.test, pred))
  plot(get.response(formula, fb.test), pred, pch=19,
       col = adjustcolor("grey10", 0.3),
       xlim=lims, ylim=lims, xlab="actual", ylab="predicted",
       main = paste("Boosted tree fit on test set for M =", M,
                    "depth =", max.depth),
       sub = paste("APSE =", round(mean((actual.test - pred)^2), 4) )
      )
}
```


Boosted tree fit on test set for M = 1000 depth = 1
predicted
1
2
3
4
5
6
7
8
actual
APSE = 0.4249



As the plots show, the predictions can be very good for some really small trees (even maximum depth of 1!). Boosting takes a lot of very poor predictors and puts them together to get a good predictor.

We could use cross-validation to estimate any of these parameters.

4.3.1 Gradient Boosting

The regression trees are minimizing the RSS. More generally, as with M-estimates we could imagine choosing a prediction tree $\widehat{\mu}(\mathbf{x})$ that minimized

$$\boldsymbol{\rho}(\boldsymbol{\mu}) = \sum_{i=1}^{N} \rho(y_i, \mu(\mathbf{x}_i)).$$

Suppose that we had an estimate $\widehat{\mu}_{m-1}(\mathbf{x})$, how might we improve it? If we think of $\sum_{i=1}^{N} \rho(y_i, \mu(\mathbf{x}_i))$ as a function of $\mu(\mathbf{x})$, we might want to move our estimate "downhill", that is in the direction of steepest descent.

The gradient at $\widehat{\mu}_{m-1}(\mathbf{x})$ is the vector of partial derivatives

$$\mathbf{g}_m = [g_{im}] = \left[\left(\frac{\partial \boldsymbol{\rho}(\boldsymbol{\mu})}{\partial \mu_i} \right) \Big|_{\boldsymbol{\mu}=\widehat{\boldsymbol{\mu}}_{m-1}} \right]$$

and points in the direction of greatest increase in the function from the point $\widehat{\boldsymbol{\mu}}_m$.

To **decrease** the function, we want to move away from $\widehat{\mu}_{m-1}(\mathbf{x})$ in the direction of $\mathbf{g}_m$ by some amount

$$\widehat{\boldsymbol{\mu}}_m(\mathbf{x}) = \widehat{\boldsymbol{\mu}}_{m-1}(\mathbf{x}) - \lambda_m \ \mathbf{g}_m$$

where λ_m is to be chosen. A reasonable choice for λ_m would be that which minimizes $\boldsymbol{\rho}(\boldsymbol{\mu}_{m-1} - \lambda_m \, \mathbf{g}_m)$.

The negative gradient $-\mathbf{g}_m$ defines the direction of **steepest descent** and finding the minimizing λ_m is called a **line search** along that direction.

Making these improvements for M iterations from an initial estimate $\widehat{\mu}_0(\mathbf{x})$ suggests a different boosting estimate:

$$\widehat{\mu}(\mathbf{x}) = \sum_{m=0}^{M} h_m(\mathbf{x})$$

where $h_0(\mathbf{x}) = \widehat{\mu}_0(\mathbf{x})$, the initial estimate, and $h_m(\mathbf{x}) = -\lambda_m \mathbf{g}_m(\mathbf{x})$ for $m = 1, \ldots, M$.

Now rather than $h_m(\mathbf{x})$ being exactly the same as $-\lambda_m \mathbf{g}_m(\mathbf{x})$, we would like it to be of a particular form. Namely, we want our estimate to be of the form

$$\widehat{\mu}(\mathbf{x}) = \sum_{m=0}^{M} \beta_m \widehat{\mu}_m(\mathbf{x})$$

where each $\widehat{\mu}_m(\mathbf{x})$ is a regression tree of some fixed complexity (N.B. this argument is entirely general and need not be restricted to a regression tree). This can be achieved by simply replacing $-\mathbf{g}_m(\mathbf{x})$ by a regression tree, but which regression tree?
Since a regression tree is a function with parameters (i.e. the splits defining the fixed number of regions), so we can simply fit a regression tree (of the desired complexity) to pseudo-responses

$$z_i = -g_m(\mathbf{x}_i) \quad \text{for} \quad i \in \mathcal{S}.$$

We can now put this all to gether into a general algorithm called **gradient boosting** (by J. Friedman (1999)):

1. **Initialize** $\widehat{\mu}_0(\mathbf{x}) \leftarrow \arg\min_{\mu} \sum_{i \in \mathcal{S}} \quad \rho(y_i, \mu)$
2. **Boosting**: For $m = 1, \ldots, M$:
   a. Get pseudo-responses: $z_i \leftarrow - \left[\frac{\partial}{\partial \mu_i} \rho(y_i, \mu(\mathbf{x}_i)) \Big|_{\mu(\mathbf{x}) = \widehat{\mu}(\mathbf{x})_{m-1}} \right] \quad \text{for} \quad i \in \mathcal{S}$
   b. Fit a regression function h to z_is (parameterized by some vector $\boldsymbol{\theta}$):
   $\boldsymbol{\theta}_m \leftarrow \arg\min_{\beta, \boldsymbol{\theta}} \; \sum_{i \in \mathcal{S}} \left(z_i - \beta h(\mathbf{x_i}, \boldsymbol{\theta}) \right)^2$
   c. Perform the line search for λ:
   $\lambda_m \leftarrow \arg\min_{\lambda} \; \sum_{i \in \mathcal{S}} \rho(y_i, \widehat{\mu}_{m-1}(\mathbf{x}_i) + \lambda \; h(\mathbf{x_i}, \boldsymbol{\theta}_m))$
   d. Update the estimate: $\widehat{\mu}_m(\mathbf{x}) \leftarrow \widehat{\mu}_{m-1}(\mathbf{x}) + \lambda_m \; h(\mathbf{x_i}, \boldsymbol{\theta}_m))$
3. **Return**: $\widehat{\mu}_M(\mathbf{x})$

This is a general algorithm. For regression least-squares regression problems (e.g. our regression trees), we take

$$\rho(y, \mu(\mathbf{x})) = \frac{1}{2} \left(y - \mu(\mathbf{x}_i) \right)^2$$

giving $\widehat{\mu}(\mathbf{x}) = \overline{y}$, and pseudo-responses

$$z_i = y_i - \widehat{\mu}_{m-1}(\mathbf{x}_i)$$

are the current estimated residuals. For regression trees, we just use a regression tree estimator of some complexity as our function $h(\mathbf{x}, \boldsymbol{\theta})$ that we fit to the current residual (here $\boldsymbol{\theta}$ represents the splits that determine the regions of $\mathbf{x}$ which define the tree).

4.3.2 Regularization

Note that gradient boosting, like tree fitting itself, is designed to add the best improvement to the prediction as possible at each step (for the given type of estimator, here trees). The worry in such cases is that we overfit the data.

One way that we avoid this is my limiting the number of boosts, that is by restricting the size of M. The gradient boost does this. However, if we also handicap the estimator at each step we might also avoid overfitting. As with the first example of boosting, one possibility is to effect this handicap by "shrinking" the estimate at *every* step. That is, in the gradient boost algorithm described above, we replace step (d) in each boosting iteration by

$$\mathrm{d}^*. \quad \widehat{\mu}_m(\mathbf{x}) \leftarrow \widehat{\mu}_{m-1}(\mathbf{x}) + \alpha \ \times \ \lambda_m \ h(\mathbf{x_i}, \boldsymbol{\theta}_m))$$

where $\alpha \in (0, 1]$ is a **shrinkage** parameter. If $\alpha = 1$, the gradient boost is applied without shrinkage. A value of $\alpha << 1$ will handicap each prediction. The shrinkage parameter $\alpha > 0$ is sometimes called the **learning rate** or, more plainly, the **step size** parameter.

4.3.3 Application

The *R* package `gbm` implements gradient boosting for a variety of regression models, including trees. The main method is `gbm(...)$ which takes as arguments the`formula`(as a formula object, or expression), the`data`,` a`shrinkage`parameter (i.e. $\alpha \in (0,1]$), and`n.trees‘ or M, amongst numerous others.

We can try this out on the facebook data.

```
library(gbm)

fb.boost <- gbm(as.formula(formula), data=facebook, shrinkage = 1, n.trees = 100)
```

```
## Distribution not specified, assuming gaussian ...
```

```
# which prints as
fb.boost
```

```
## gbm(formula = as.formula(formula), data = facebook, n.trees = 100,
##     shrinkage = 1)
## A gradient boosted model with gaussian loss function.
## 100 iterations were performed.
## There were 8 predictors of which 8 had non-zero influence.
```

Note that it says it assumed a "gaussian" model. This is equivalent to our least-squares objective function (i.e. maximizing log-likelihood of a Gaussian model). This is controlled by the parameter `distribution`. Other distributions will produce fits for other problems (e.g. "bernoulli" for a binary response); hence the package is called "generalized boosted regression models".

Two other arguments of possible immediate interest are `interaction.depth` and `n.minobsinnode`. The latter is the minimum number of observations in a terminal node or leaf of any tree. The former refers to the depth of the tree in terms of its capacity to generate interactions. For example an `interaction.depth = 1` will split on a single explanatory variate and so no "interaction" between it and another explanatory variate can occur in fitting the response. The resulting model will be **additive** in the explanatory variates. An `interaction.depth=2` ensures that there are at most two-way interactions in the model, et cetera. That is the model that we end up fitting may also be written more familiarly as

$$\widehat{\mu}(\mathbf{x}) = \sum_j h_j(x_j) + \sum_{j,k} h_{jk}(x_j, x_k) + \sum_{j,k,l} h_{jkl}(x_j, x_k, x_l) + \cdots$$

This now looks something like we have had on many other occasions. The first sum are the "main effects", the second the "two-way interaction effect", the third the "three-way interaction effects", and so on. The argument `interaction.depth` specifies how deep the trees will be an hence how many way interactions the models might have.

A summary of the fit is had by `summary(...)` which is implemented by the `summary.gbm(...)` method and so has many arguments peculiar to gradient boost fitted objects. The summary determines the **relative influence** of each variate to the overall fit.

As discussed earlier with random forests, we can build up a measure of the importance of the jth variate in a single tree T by considering only those non-terminal nodes in the tree T which split on the jth variate, denoting this set of nodes by $N_j(T)$ (to emphasize its dependence on the tree T and variate j). Suppose at the kth such node, the change in RSS is denoted $\Delta RSS_k(T, j)$ (or $\Delta\rho$ more generally). Then we might define the (squared) importance of the variate j to be

$$I_j^2(T) = \sum_{k \in N_j(T)} \Delta RSS_k(T, j).$$

For a collection of M trees $T_1, \ldots, T_M$ obtained via boosting, the corresponding (squared) importance of variate j would be the average

$$I_j^2 = \frac{1}{M} \sum_{m=1}^{M} I_j^2(T_m).$$

The relative importance of the jth variate amongst all p explanatory variates could be obtained as the proportion of I_j^2 in the sum $\sum_{j=1}^{p} I_j^2$.

In `gbm`, this is called the `relative.influence` and a function of that name will calculate it. It takes an argument `n.trees` which defaults to the total number of trees used in the boosting.

```
relative.influence(fb.boost)
```

```
## n.trees not given. Using 100 trees.
```

```
##   Page.likes         Paid    Post.Hour  Impressions Post.Weekday
##    86.501924    15.871373    43.176402   380.966509     9.251147
##   Post.Month         Type     Category
##     2.734523    29.653367    43.966410
```

```
## Can also scale this
## NOTE the dot in the argument names :-{
relative.influence(fb.boost, scale. = TRUE, sort. = TRUE)
```

```
## n.trees not given. Using 100 trees.
```

```
##  Impressions   Page.likes     Category    Post.Hour         Type
##  1.000000000  0.227059130  0.115407546  0.113333853  0.077837203
##         Paid Post.Weekday   Post.Month
##  0.041660810  0.024283359  0.007177857
```

```
## Or, express it as a percent of total
rel.inf <- relative.influence(fb.boost, scale. = TRUE, sort. = TRUE)
```

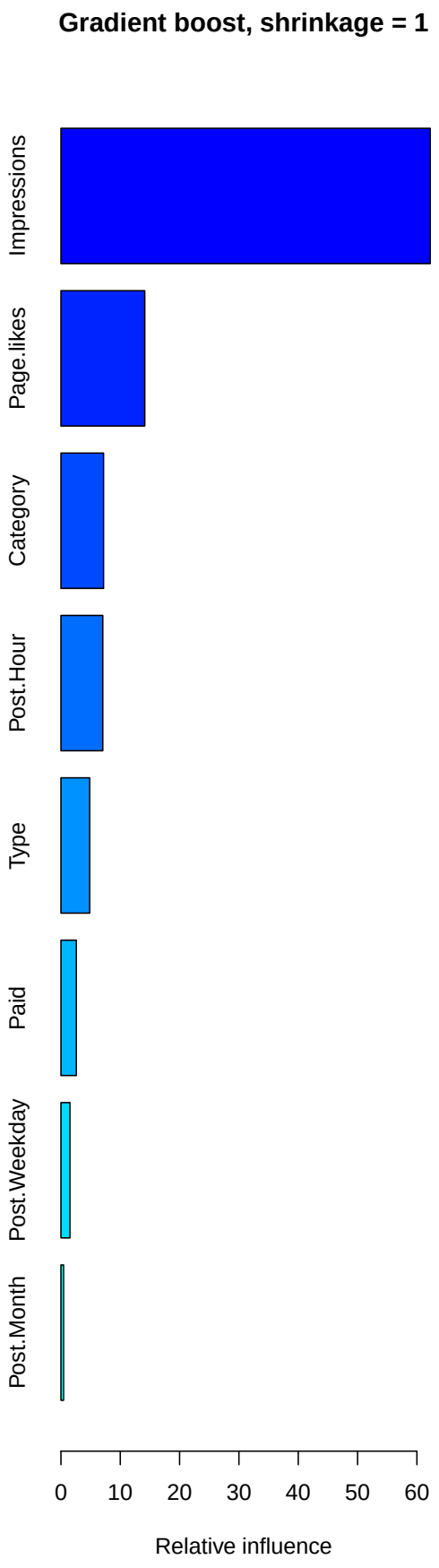
```
## n.trees not given. Using 100 trees.
```

```
rel.inf <- rel.inf/sum(rel.inf)
round(100 * rel.inf, 2)
```

```
##  Impressions   Page.likes     Category    Post.Hour         Type
##        62.24        14.13         7.18         7.05         4.84
##         Paid Post.Weekday   Post.Month
##         2.59         1.51         0.45
```

In `gbm`, the `summary(...)` function calculates this last relative influence and by default (`plotit = TRUE`) plots the the values as a bar plot. For our fitted model, we get

```
summary(fb.boost, main = "Gradient boost, shrinkage = 1")
```


Gradient boost, shrinkage = 1
Impressions
Page.likes
Category
Post.Hour
Type
Paid
Post.Weekday
Post.Month
0
10
20
30
40
50
60
Relative influence

```
##                       var    rel.inf
## Impressions   Impressions 62.2370578
## Page.likes     Page.likes 14.1314922
## Category         Category  7.1826261
## Post.Hour       Post.Hour  7.0535656
## Type                 Type  4.8443585
## Paid                 Paid  2.5928462
## Post.Weekday Post.Weekday  1.5113248
## Post.Month     Post.Month  0.4467287
```

Note that there is another argument to `gbm(...)` that adds more randomness to the set of trees that are used in the gradient boosting algorithm. That argument is `bag.fraction`. If `bag.fraction < 1`, then only a fraction of the (training) data will be selected at random to use on the next tree in the boost. The default is `bag.fraction = 0.5` so only half of the data at any time will be used to fit a tree.

This means, for example, that re-running the previous `gbm(...)` will produce slightly different results. We can see this as follows:

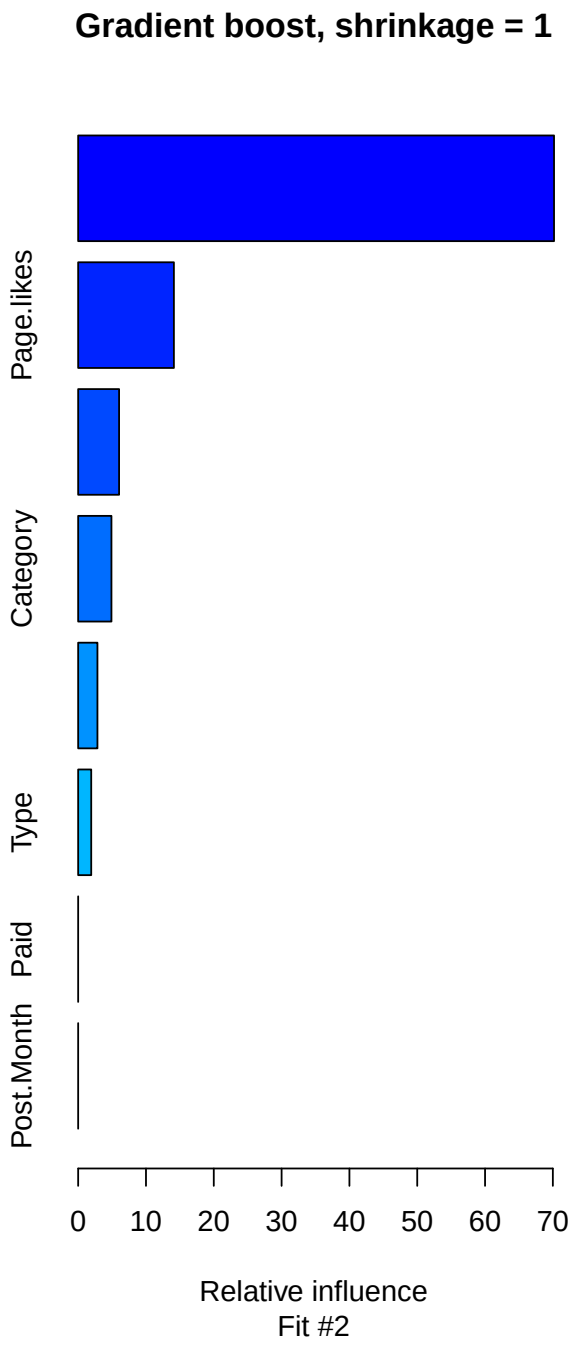
```
savePar <- par(mfrow=c(1,2))

fb.boost2 <- gbm(as.formula(formula), data=facebook,
                 shrinkage = 1, n.trees = 100)
```

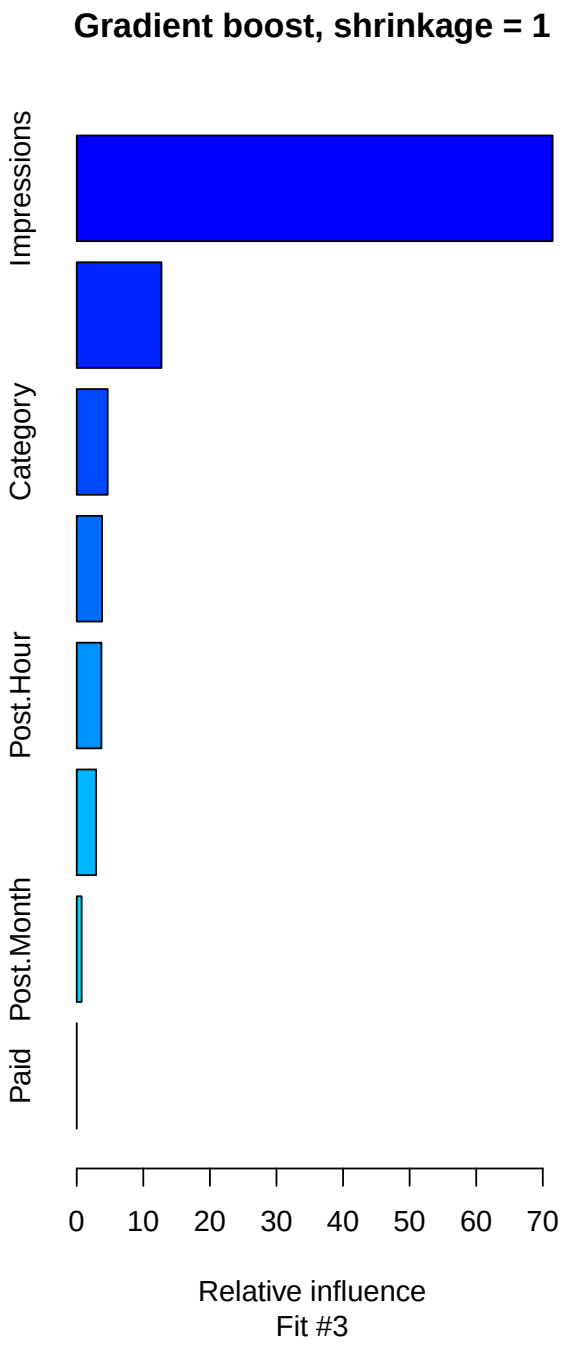
```
## Distribution not specified, assuming gaussian ...
```

```
fit2summary <- summary(fb.boost2,
                       main = "Gradient boost, shrinkage = 1",
                       sub = "Fit #2")
fb.boost3 <- gbm(as.formula(formula), data=facebook,
                 shrinkage = 1, n.trees = 100)
```

```
## Distribution not specified, assuming gaussian ...
```

```
fit3summary <- summary(fb.boost3,
                       main = "Gradient boost, shrinkage = 1",
                       sub = "Fit #3")
```






```
par(savePar)
results <- round(cbind(fit2summary, fit3summary)[,c(2,4)], 1)
names(results) <- c("Fit #2", "Fit #3")
knitr::kable(results)
```

	Fit #2	Fit #3
Impressions	70.2	71.5
Page.likes	14.1	12.7
Post.Hour	6.0	4.6
Category	4.9	3.8
Post.Weekday	2.8	3.7

	Fit #2	Fit #3
Type	1.9	2.9
Paid	0.0	0.7
Post.Month	0.0	0.0

We can remove the randomness by forcing `bag.fraction = 1`.

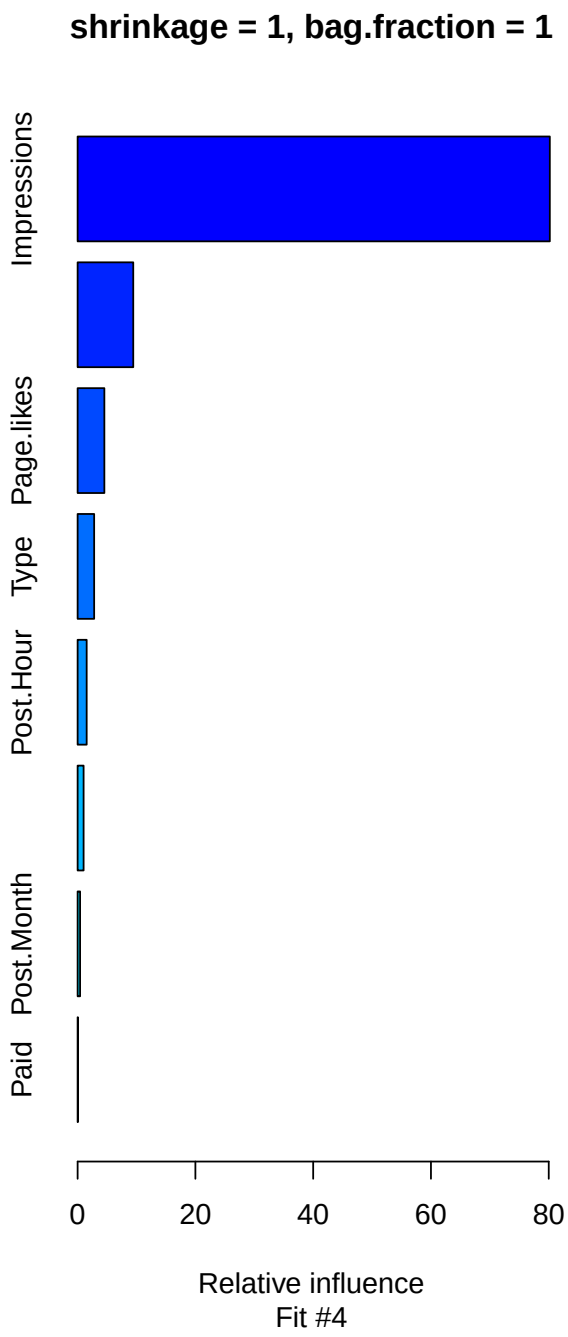
```
savePar <- par(mfrow=c(1,2))

fb.boost4 <- gbm(as.formula(formula), data=facebook,
                 shrinkage = 1, n.trees = 100,
                 bag.fraction = 1)
```

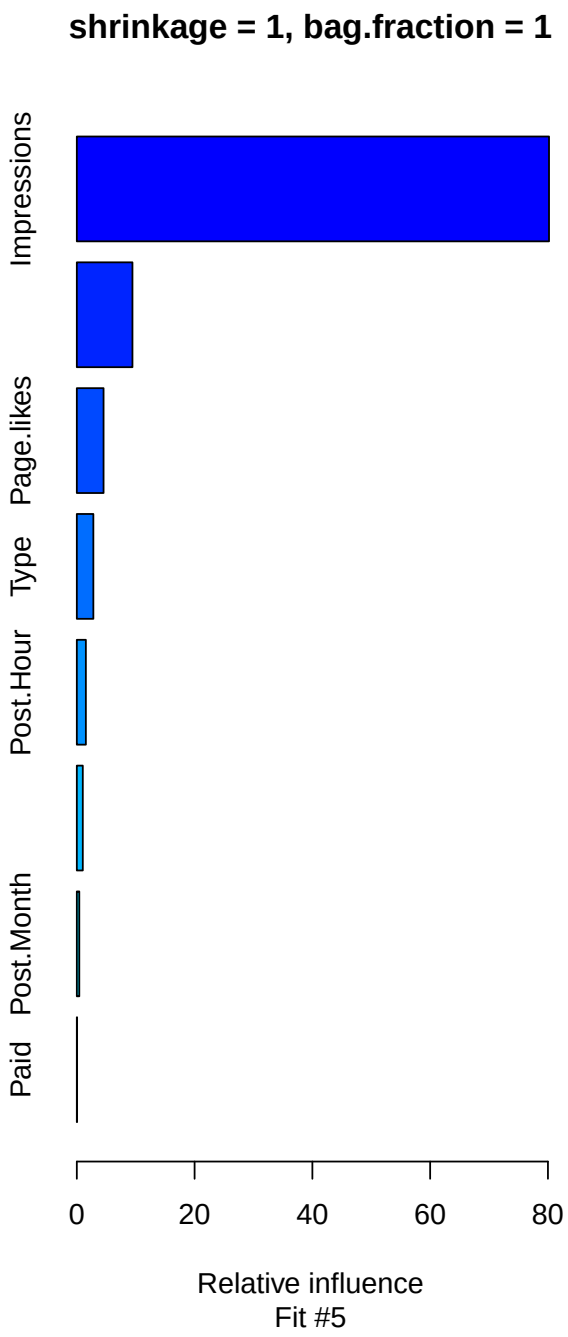
```
## Distribution not specified, assuming gaussian ...
```

```
fit4summary <- summary(fb.boost4,
                       main = "shrinkage = 1, bag.fraction = 1",
                       sub = "Fit #4")
fb.boost5 <- gbm(as.formula(formula), data=facebook,
                 shrinkage = 1, n.trees = 100,
                 bag.fraction = 1)
```

```
## Distribution not specified, assuming gaussian ...
```

```
fit5summary <- summary(fb.boost5,
                       main = "shrinkage = 1, bag.fraction = 1",
                       sub = "Fit #5")
```






```
par(savePar)
results <- round(cbind(fit4summary, fit5summary)[,c(2,4)], 1)
names(results) <- c("Fit #4", "Fit #5")
knitr::kable(results)
```

	Fit #4	Fit #5
Impressions	80.2	80.2
Category	9.4	9.4
Page.likes	4.5	4.5
Type	2.8	2.8
Post.Hour	1.5	1.5

	Fit #4	Fit #5
Post.Weekday	1.0	1.0
Post.Month	0.4	0.4
Paid	0.0	0.0

The above fits, have no shrinkage and no randomness in which observations are to be used in the fitting. It is closest to the original gradient descent boosting algorithm we described.

We will now look at getting cross-validated errors.

NOTE that the `gbm(...)` function's cross validation will fail on our model and data. There are a few problems with `gbm(...)`

First: The authors of `gbm(...)` were anticipating that the response would only have a simple function like `log` applied to a single variate name `All.interactions`. Passing `All.interactions + 1` to the log function causes the failure of `gbm(...)`. (The actual failure occurs in the function `gbmCrossValPredictions(...)`.)

To circumvent this failure, we can change our data frame to `facebook2` to have the "+1" built in, and then change the `formula` string to match.

```
facebook2 <- facebook
facebook2[, "All.interactions"] <- facebook2[, "All.interactions"] + 1
# And we will rebuild our formula string to suit
# Break the formula into pieces
formula.sides <- strsplit(formula, "~")[[1]]
rhs.formula <- formula.sides[2]
formula2 <- paste("log(All.interactions) ~", rhs.formula)
```

Second: There are a few other errors `predict.gbm(...)` and in `gbmCrossValPredictions(...)`. In the latter, the problem involves the same line with the first problem above.

To fix these, you can use the "fudged" versions of these that can be found on the course website. If you load that file ("fudgegbm.R"), then the following will work.

```
# Assuming the file is in the current working directory:
source("Rcode/fudgegbm.R")
#
alpha <- 0.01
M <- 500
set.seed(12345)
fb.boost.shrink.fudge <- fudge.gbm(as.formula(formula2),
                                   data=facebook2,
                                   distribution = "gaussian",
                                   shrinkage = alpha, n.trees = M,
                                   bag.fraction = 1,
                                   cv.folds = 5
)
```

Which gives us the cross-validated error for each tree in `fb.boost.shrink.fudge$cv.error`. The last one will be our minimum: `fb.boost.shrink.fudge$cv.error[M]` = 0.557045.

Alternatively, we can use `gbm(...)` provide the data frame we use contains **only** those explanatory variates which appear in the formula. This involves changing our data frame again.

```
facebook3 <- facebook2[,
                       c("All.interactions",
                         get.explanatory_varnames(formula2))
                         ]
```

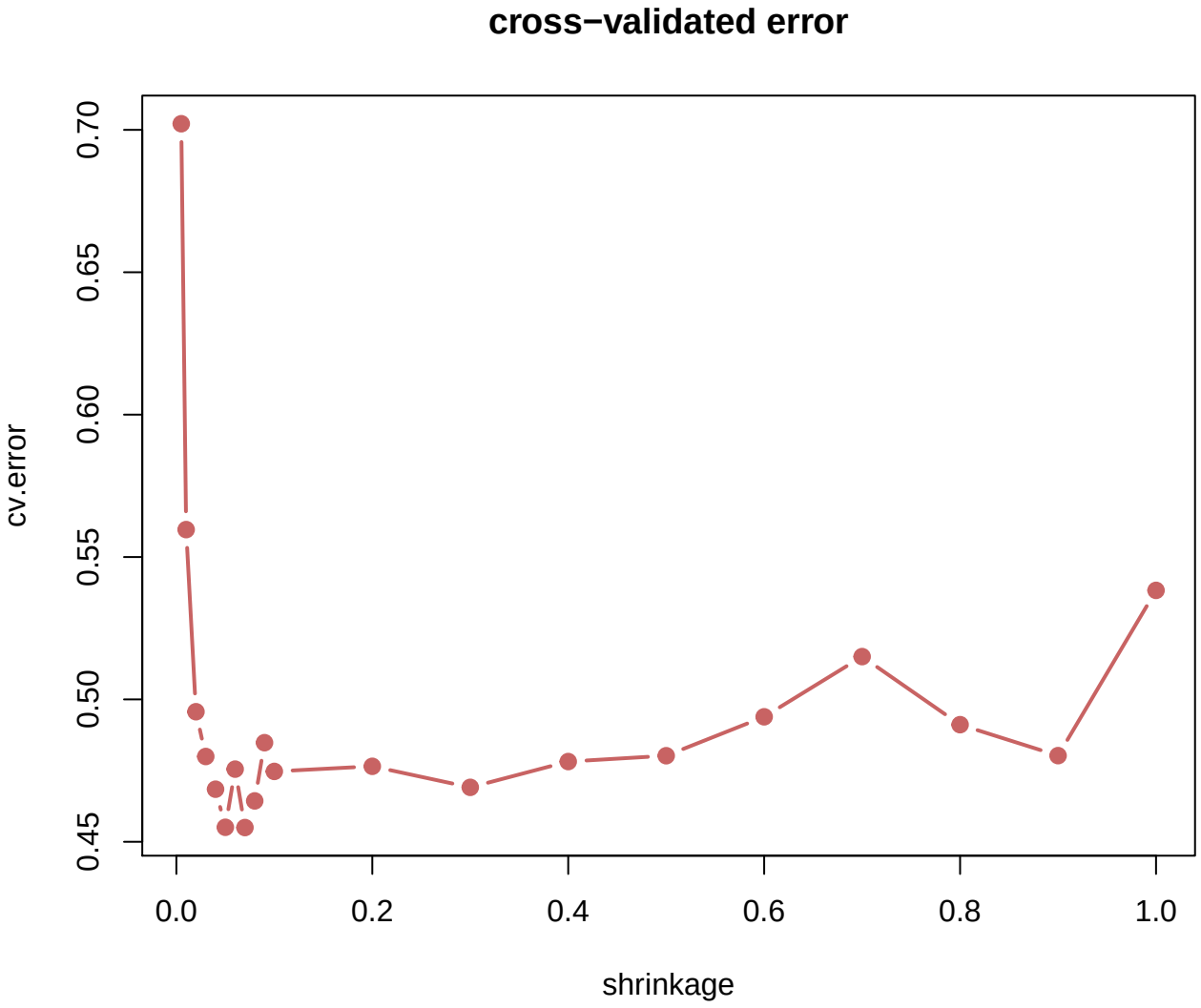
Now we can try to use `gbm(...)` directly.

```
alpha <- 0.01
M <- 500
set.seed(12345)
fb.boost.shrink <- gbm(as.formula(formula2),
                       data=facebook3,
                       distribution = "gaussian",
                       shrinkage = alpha, n.trees = M,
                       bag.fraction = 1,
                       cv.folds = 5
                       )
```

Which gives us the cross-validated error for each tree in `fb.boost.shrink$cv.error`. The last one will be our minimum: `fb.boost.shrink$cv.error[M]` = 0.557045. These are the **same as before** because we `set.seed(12345)` before each call. Otherwise, differences between these values and the previous ones would be due to cross-validation (i.e. different folds).

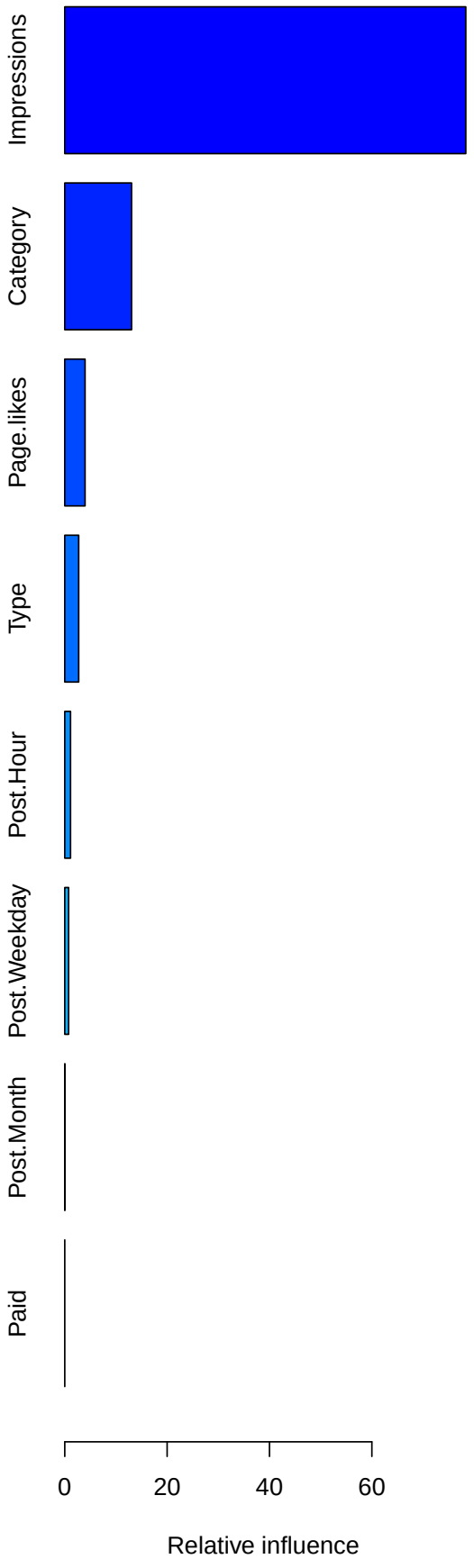
NOTE It seems that `gbm(...)` likes to think that **all* variates in the data set are also available in the model.

```
# Set up
M <- 500
k <- 5
set.seed(31853071)
# Shrinkage parameter values
alphas <- c(0.005, 0.01, 0.02, 0.03, 0.04,
            0.05, 0.06, 0.07, 0.08, 0.09,
            0.10, 0.20, 0.30, 0.40, 0.50,
            0.60, 0.70, 0.80, 0.90, 1)
n_alphas <- length(alphas)
cverror <- numeric(length = n_alphas)
Mvals <- numeric(length = n_alphas)
fit <- list(length = n_alphas)

for (i in 1:n_alphas) {
 fit[[i]] <-  fb.boost.shrink <- gbm(as.formula(formula2),
                       data=facebook3,
                       distribution = "gaussian",
                       shrinkage = alphas[i],
                       n.trees = M,
                       bag.fraction = 1,
                       cv.folds = k
                       )
 cverror[i] <-  min(fit[[i]]$cv.error)
 Mvals[i] <- which.min(fit[[i]]$cv.error)
}

plot(alphas, cverror, type = "b",
     col =  adjustcolor("firebrick", 0.7), pch=19, lwd=2,
     main = "cross-validated error", xlab = "shrinkage", ylab="cv.error")
```



which seems to suggest an $\alpha \approx 0.05$ to 0.07.

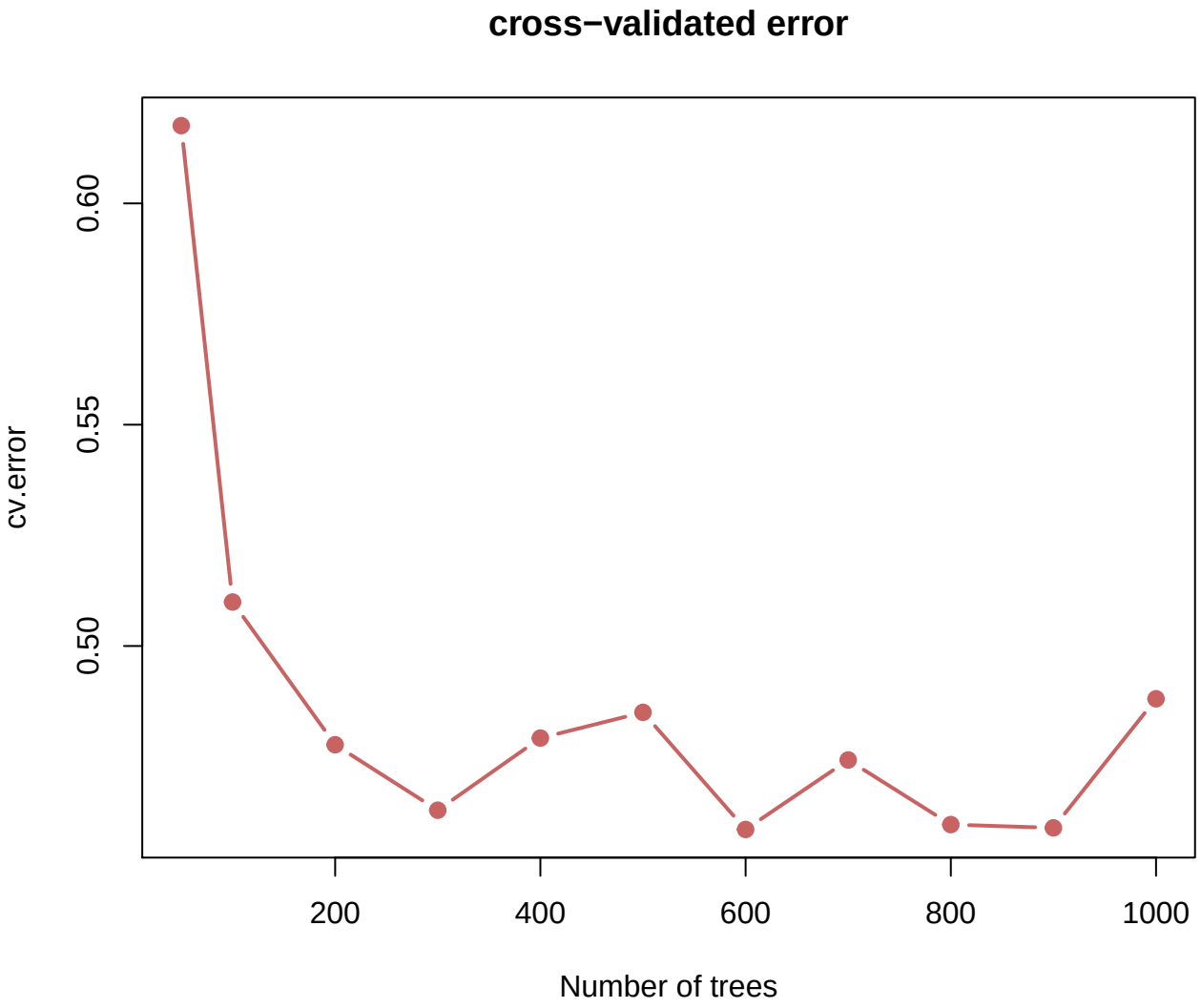
```
# Let's look at the best one of these
i <- which.min(cverror)
alpha <- alphas[i]
summary(fit[[i]])
```


Impressions
Category
Page.likes
Type
Post.Hour
Post.Weekday
Post.Month
Paid
0
20
40
60
Relative influence

```
##                       var     rel.inf
## Impressions   Impressions 78.35124081
## Category         Category 13.05284929
## Page.likes     Page.likes  3.96765969
## Type                 Type  2.72283728
## Post.Hour       Post.Hour  1.12793422
## Post.Weekday Post.Weekday  0.75661834
## Post.Month     Post.Month  0.02086037
## Paid                 Paid  0.00000000
```

We might also investigate the values of M for a fixed `alpha`.

```
# Set up
alpha <- 0.07
k <- 5
set.seed(31853071)
# Shrinkage parameter values
Ms <- c(50, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000)
n_Ms <- length(Ms)
cverror <- numeric(length = n_Ms)
Mvals <- numeric(length = n_Ms)
fit <- list(length = n_Ms)

for (i in 1:n_Ms) {
 fit[[i]] <-  fb.boost.shrink <- gbm(as.formula(formula2),
                       data=facebook3,
                       distribution = "gaussian",
                       shrinkage = alpha,
                       n.trees = Ms[i],
                       bag.fraction = 1,
                       cv.folds = k
                       )
 cverror[i] <-  min(fit[[i]]$cv.error)
}

plot(Ms, cverror, type = "b",
     col =  adjustcolor("firebrick", 0.7), pch=19, lwd=2,
     main = "cross-validated error", xlab = "Number of trees", ylab="cv.error")
```



which suggests diminishing returns beyond $M = 300$ trees.

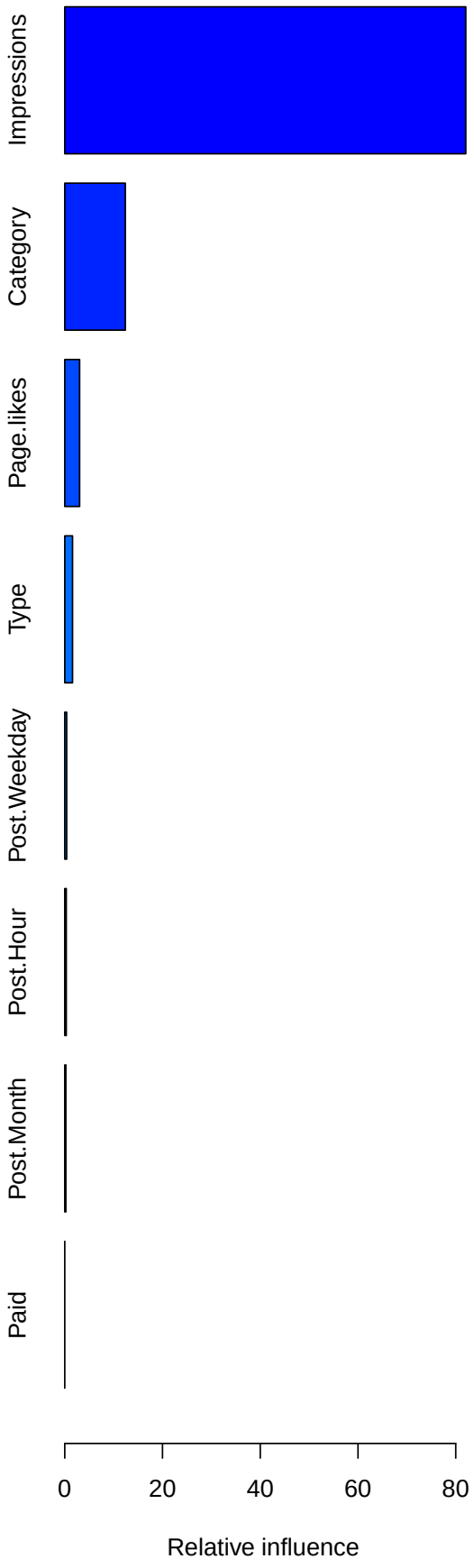
You could repeat the above with `bag.fraction = 0.5`, say, (the default) as well. (Get about the same values.)

Allowing deeper trees (interaction), with `bag.fraction < 1`

```
fb.boost.shrink <- gbm(as.formula(formula2), data=facebook3,
                       shrinkage = 0.01, n.trees = 300,
                       interaction.depth = 2)
```

```
## Distribution not specified, assuming gaussian ...
```

```
summary(fb.boost.shrink)
```


Impressions
Category
Page.likes
Type
Post.Weekday
Post.Hour
Post.Month
Paid
0
20
40
60
80
Relative influence

```
##                       var    rel.inf
## Impressions   Impressions 82.0554323
## Category         Category 12.3891845
## Page.likes     Page.likes  3.0385108
## Type                 Type  1.5943494
## Post.Weekday Post.Weekday  0.3883711
## Post.Hour       Post.Hour  0.3103564
## Post.Month     Post.Month  0.2237955
## Paid                 Paid  0.0000000
```

which has allowed some of the other terms to increase their relative influence.

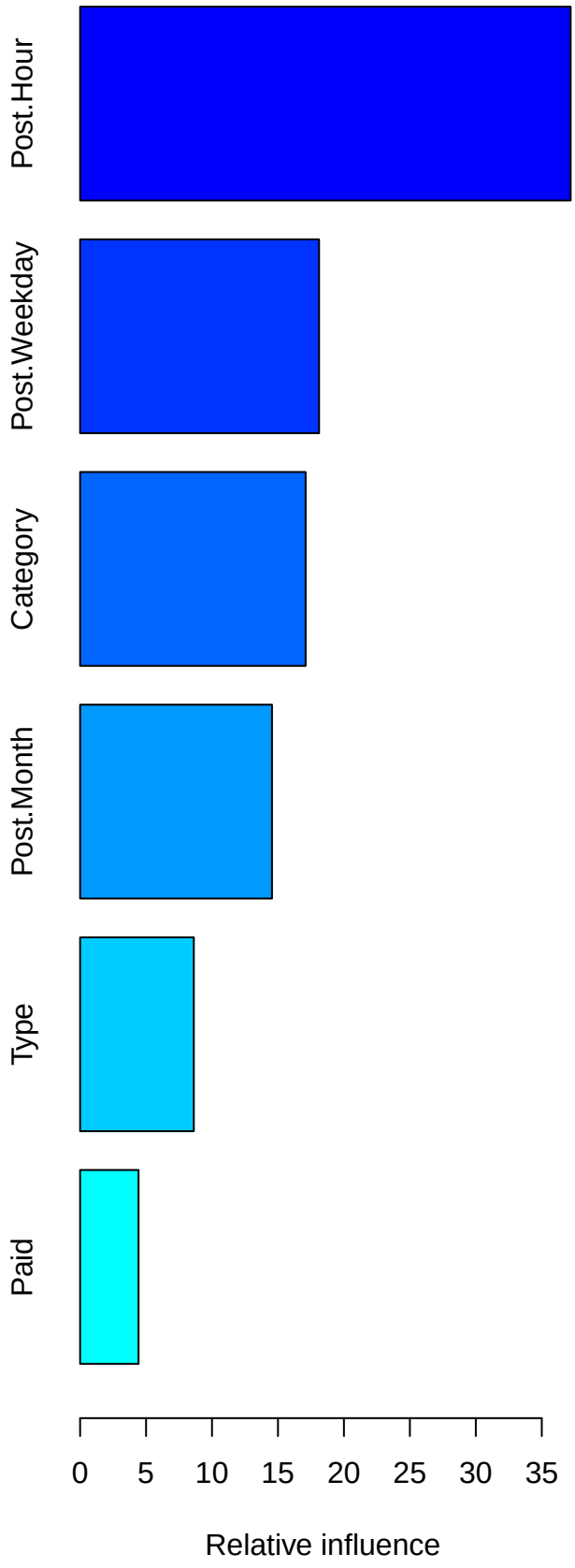
4.3.3.1 The more interesting model

Again,let's try things on the more interesting model where all explanatory variates can be controlled.

```
facebook4 <- facebook3[,
                       c("All.interactions",
                         "Paid", "Post.Hour",
                         "Post.Weekday", "Post.Month",
                         "Type", "Category")
                         ]
```

Using default `bag.fraction` but no shrinkage

```
fb.boost <- gbm(log(All.interactions) ~
                  Paid + Post.Hour +
                  Post.Weekday + Post.Month +
                  Type + Category,
                data=facebook4,
                distribution = "gaussian",
                shrinkage = 1)

summary(fb.boost)
```

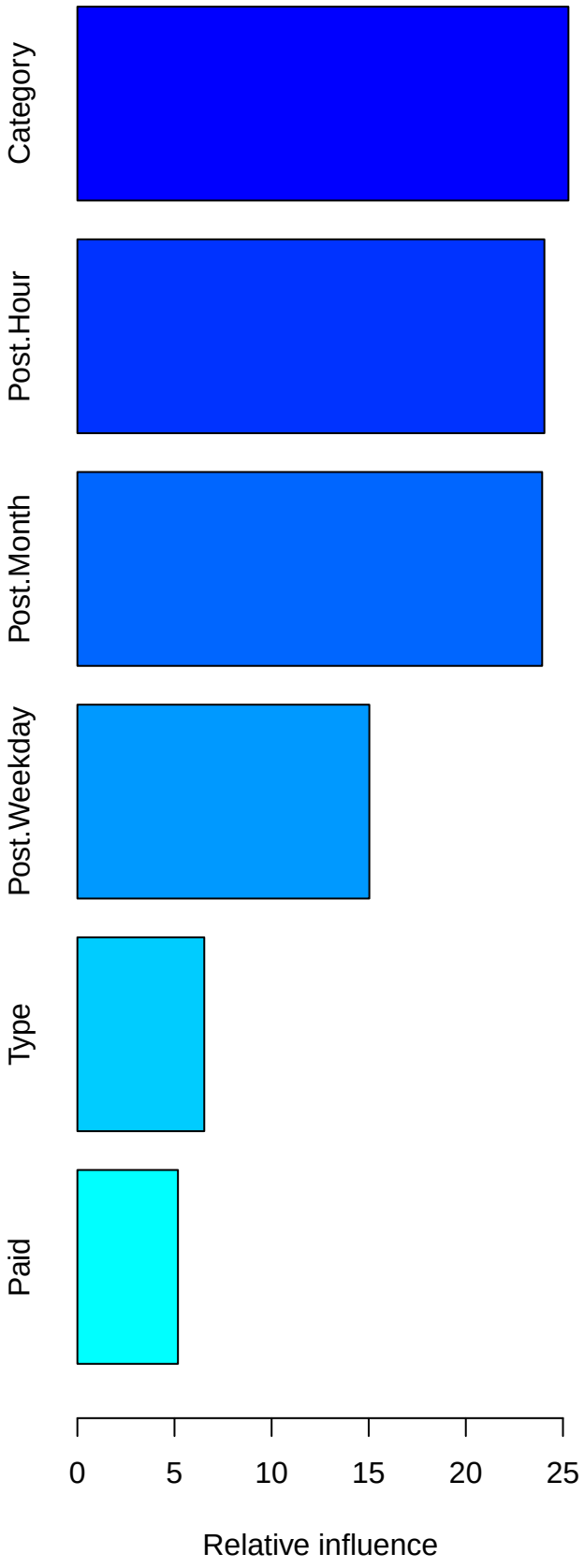

Post.Hour
Post.Weekday
Category
Post.Month
Type
Paid
0
5
10
15
20
25
30
35
Relative influence

```
##                   var   rel.inf
```

```
## Post.Hour       Post.Hour 37.189574
## Post.Weekday Post.Weekday 18.114772
## Category         Category 17.096257
## Post.Month     Post.Month 14.546673
## Type                Type  8.621446
## Paid                Paid  4.431279
```

And now using default shrinkage as well

```
fb.boost <- gbm(log(All.interactions) ~
                         Paid + Post.Hour +
                         Post.Weekday + Post.Month +
                         Type + Category,
                data=facebook4,
                distribution = "gaussian",
                shrinkage = 0.07,
                n.trees = 300,
                interaction.depth = 1)

summary(fb.boost)
```

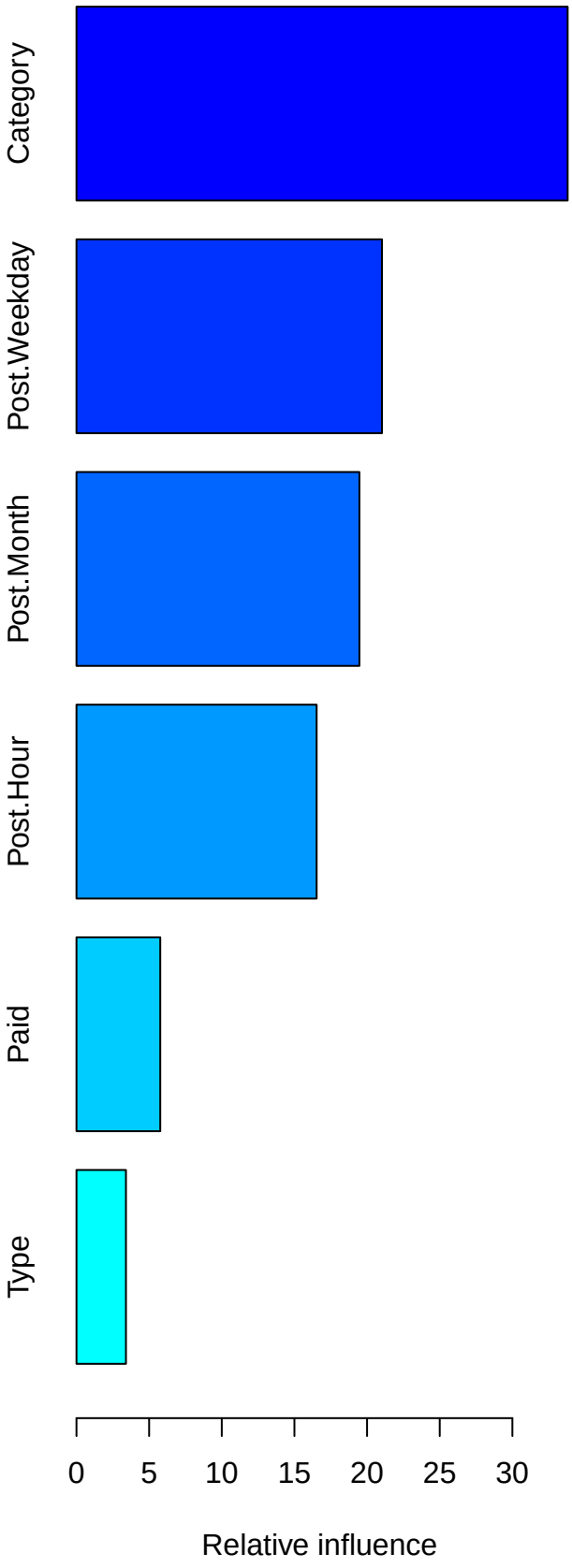

Category
Post.Hour
Post.Month
Post.Weekday
Type
Paid
0
5
10
15
20
25
Relative influence

```
##                   var   rel.inf
```

```
## Category         Category 25.281958
## Post.Hour       Post.Hour 24.044706
## Post.Month     Post.Month 23.929256
## Post.Weekday Post.Weekday 15.032074
## Type                 Type  6.533021
## Paid                 Paid  5.178985
```

And interaction

```
fb.boost <- gbm(log(All.interactions) ~
                         Paid + Post.Hour +
                         Post.Weekday + Post.Month +
                         Type + Category,
                data=facebook4,
                distribution = "gaussian",
                shrinkage = 0.01,
                n.trees = 500,
                interaction.depth = 2)

summary(fb.boost)
```


Category
Post.Weekday
Post.Month
Post.Hour
Paid
Type
0
5
10
15
20
25
30
Relative influence

```
##                    var   rel.inf
```

```
## Category         Category 33.804202
## Post.Weekday Post.Weekday 21.025411
## Post.Month     Post.Month 19.470825
## Post.Hour       Post.Hour 16.528429
## Paid                 Paid  5.765735
## Type                 Type  3.405399
```

Choosing between the various possibilities will involve a lot of cross-validation exercises.

4.3.4 Partial dependence plots

After fitting $\widehat{\mu}(\mathbf{x})$, with the $p \times 1$ vector of explanatory variates $\mathbf{x}$, we might be interested in how this prediction depends on any subset of $\mathbf{x} = \{x_1, \ldots, x_p\}$. Suppose that we are interested in a particular subset of l variates denoted by $\mathbf{z}_l = \{z_1, \ldots, z_l\}$. Let $\mathbf{z}_{\setminus l}$ be the remaining variates so that

$$\mathbf{z}_l \cup \mathbf{z}_{\setminus l} = \mathbf{x}$$

The prediction function depends on variates from both sets, i.e. with some abuse of notation we can write

$$\widehat{\mu}(\mathbf{x}) = \widehat{\mu}(\mathbf{z}_l, \mathbf{z}_{\setminus l})$$

and consider how this depends on the variates in $\mathbf{z}_l$ conditional on those $\mathbf{z}_{\setminus l}$

$$\widehat{\mu}_{\mathbf{z}_{\setminus l}}(\mathbf{z}_l) = \widehat{\mu}(\mathbf{z}_l \ \mid \ \mathbf{z}_{\setminus l})$$

Unfortunately, this is going to depend on the values of the variates in $\mathbf{z}_{\setminus l}$. We might instead consider the **marginal** function

$$\widehat{\mu}_+(\mathbf{z}_l) = \frac{1}{N} \sum_{i=1}^{N} \widehat{\mu}(\mathbf{z}_{i;l}, \mathbf{z}_{i;\setminus l}).$$

This is not too bad a summary in place of the more obvious $\widehat{\mu}_{\mathbf{z}_{\setminus l}}(\mathbf{z}_l)$ provided the dependence of the latter on $\mathbf{z}_{\setminus l}$ is not that great. If, for example, $\widehat{\mu}(\mathbf{x})$ is additive in the two variate sets, i.e.

$$\widehat{\mu}(\mathbf{x}) = \widehat{\mu}_l(\mathbf{z}_l) + \widehat{\mu}_{\setminus l}(\mathbf{z}_{\setminus l})$$

then the relationship between $\widehat{\mu}_+(\mathbf{z}_l)$ and the values of its argument $\mathbf{x}_l$ is the same, relatively, as that of $\widehat{\mu}(\mathbf{x})$ for any fixed $\mathbf{z}_{\setminus l}$. That is, the values of $\mathbf{z}_l$ and $\mathbf{z}_{\setminus l}$ cleanly separate in $\widehat{\mu}(\mathbf{x})$ and so $\widehat{\mu}_+(\mathbf{z}_l)$ is a reasonable substitute to explore that relationship.

The same is true for the of the relationship is multiplicative rather than additive. That is if

$$\widehat{\mu}(\mathbf{x}) = \widehat{\mu}_l(\mathbf{z}_l) \ \times \ \widehat{\mu}_{\setminus l}(\mathbf{z}_{\setminus l}).$$

In either case, $\widehat{\mu}_+(\mathbf{z}_l)$ is a reasonable substitute to explore the dependence of $\widehat{\mu}(\mathbf{x})$ on $\mathbf{z}_l$.

For our regression trees, each split is based on splitting a single variate at a time. This makes determining the function $\widehat{\mu}_+(\mathbf{z}_l)$ for any set of variates $\mathbf{z}_l$ fairly straight forward. For a specific set of variates $\mathbf{z}_l$, a **weighted traversal** of the tree is followed.

Start with an initial weight of 1. Beginning at the root of the tree, traverse the tree as follows. If at any branch, the splitting variate is one of the set $\mathbf{z}_l$, then the weight is not changed and we follow whichever branch, left or right applies to that variate. If, however, the splitting variate is not in $\mathbf{z}_l$ but rather in $\mathbf{z}_{\setminus l}$, then **both** branches are followed each passing on the previous weight but now multiplied by the fraction of training set observations at this split which go in each direction. The final estimate $\widehat{\mu}_+(\mathbf{z}_l)$ is simply the weighted average of the predictions of all terminal nodes visited.

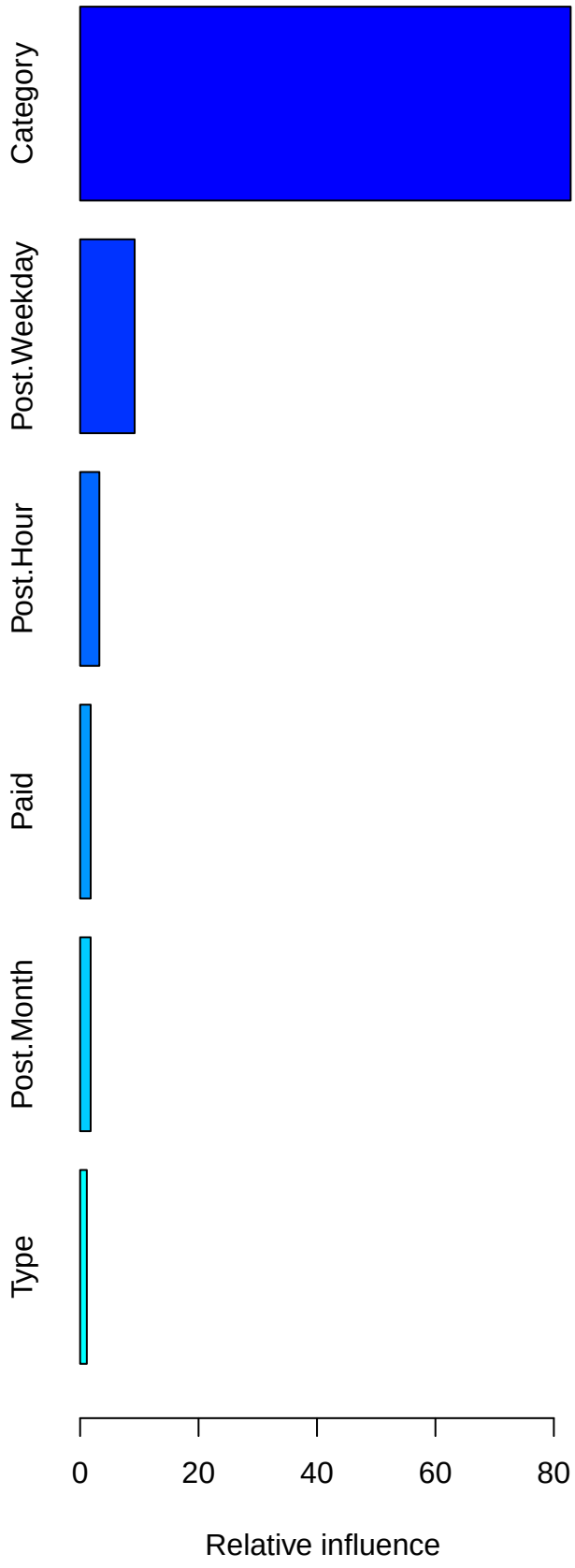
If there is a forest of trees simply use the predictions for each tree and combine them as approporiate (e.g. average for bagging; the corresponding weighted sum for boosting).

If l is small, say 1, or 2, then we could look at plots of $\widehat{\mu}_+(\mathbf{z}_l)$ versus the values of the variates of $\mathbf{z}_l$. We call such a plot a **partial dependence plot**.

The `plot(...)` function has been specialized in the `gbm` package for objects which result from a call to `gbm(...)`. The key argument is `i.var` (default `1`) which gives the indices of the variates to be plotted, that is the indices of those in $\mathbf{z}_l$.

Before producing a plot, let's get a new boosted estimate that is based on default parameters. Unlike the previous estimator, this will have only 100 as opposed to 500 trees to consider. We also use `interaction.depth =1` to ensure an additive model. We do that here just to have speedier calculations.

```
fb.boost <- gbm(log(All.interactions) ~
                          Paid + Post.Hour +
                          Post.Weekday + Post.Month +
                          Type + Category,
                data=facebook4,
                distribution = "gaussian",
                shrinkage = 0.01,
                n.trees = 100,
                interaction.depth = 1)

summary(fb.boost)
```

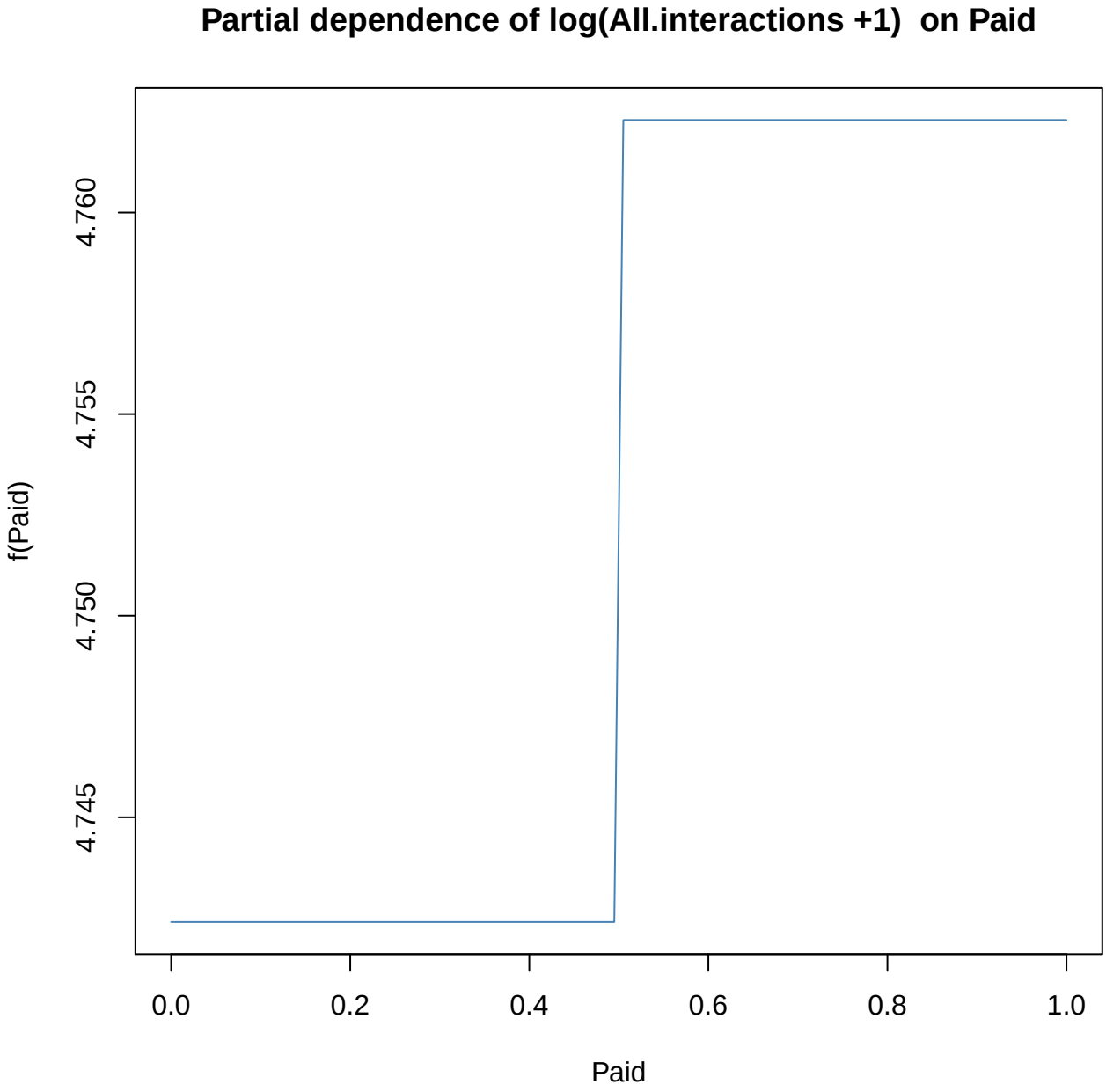

Category
Post.Weekday
Post.Hour
Paid
Post.Month
Type
0
20
40
60
80
Relative influence

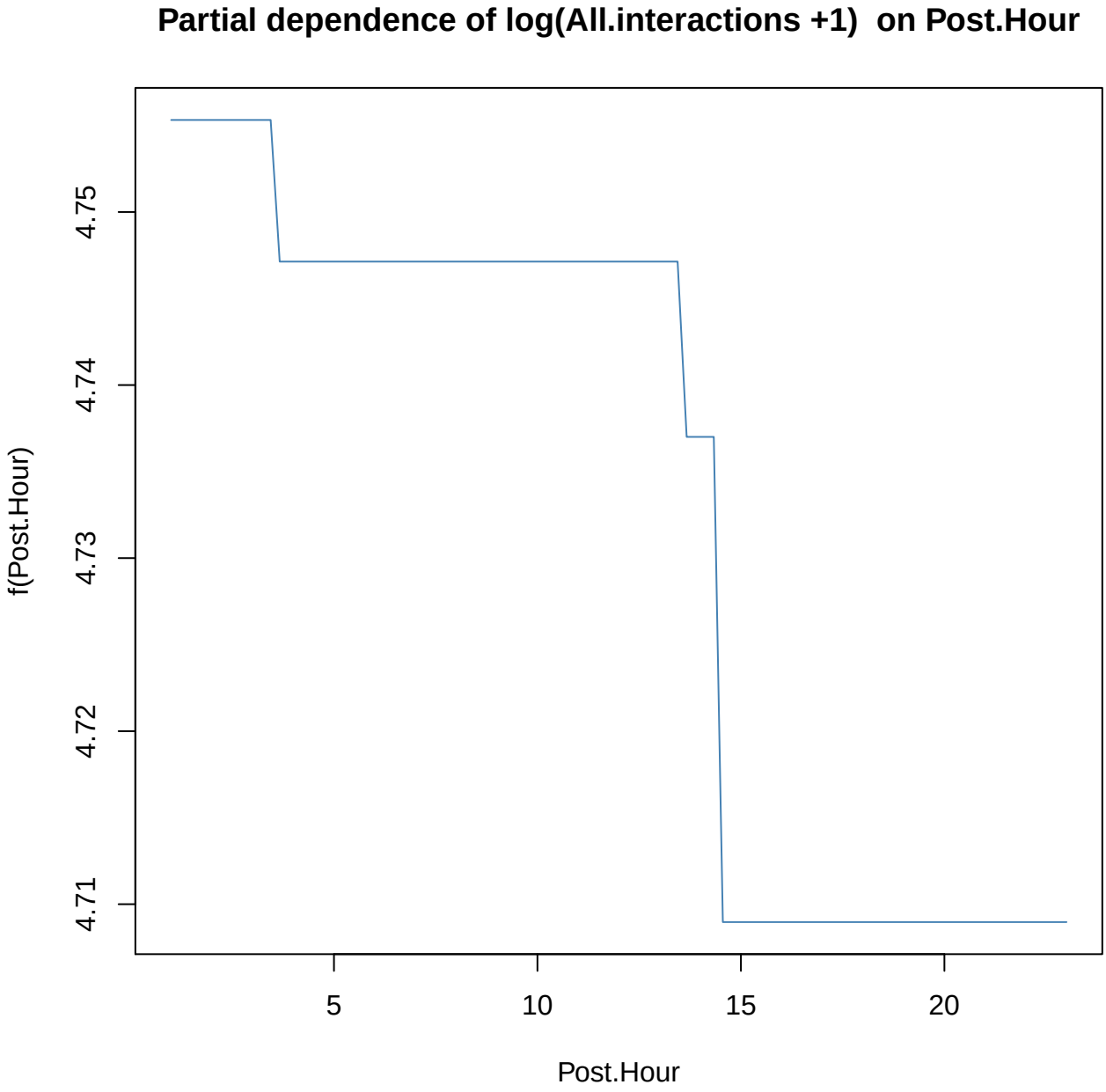
```
##                   var   rel.inf
```

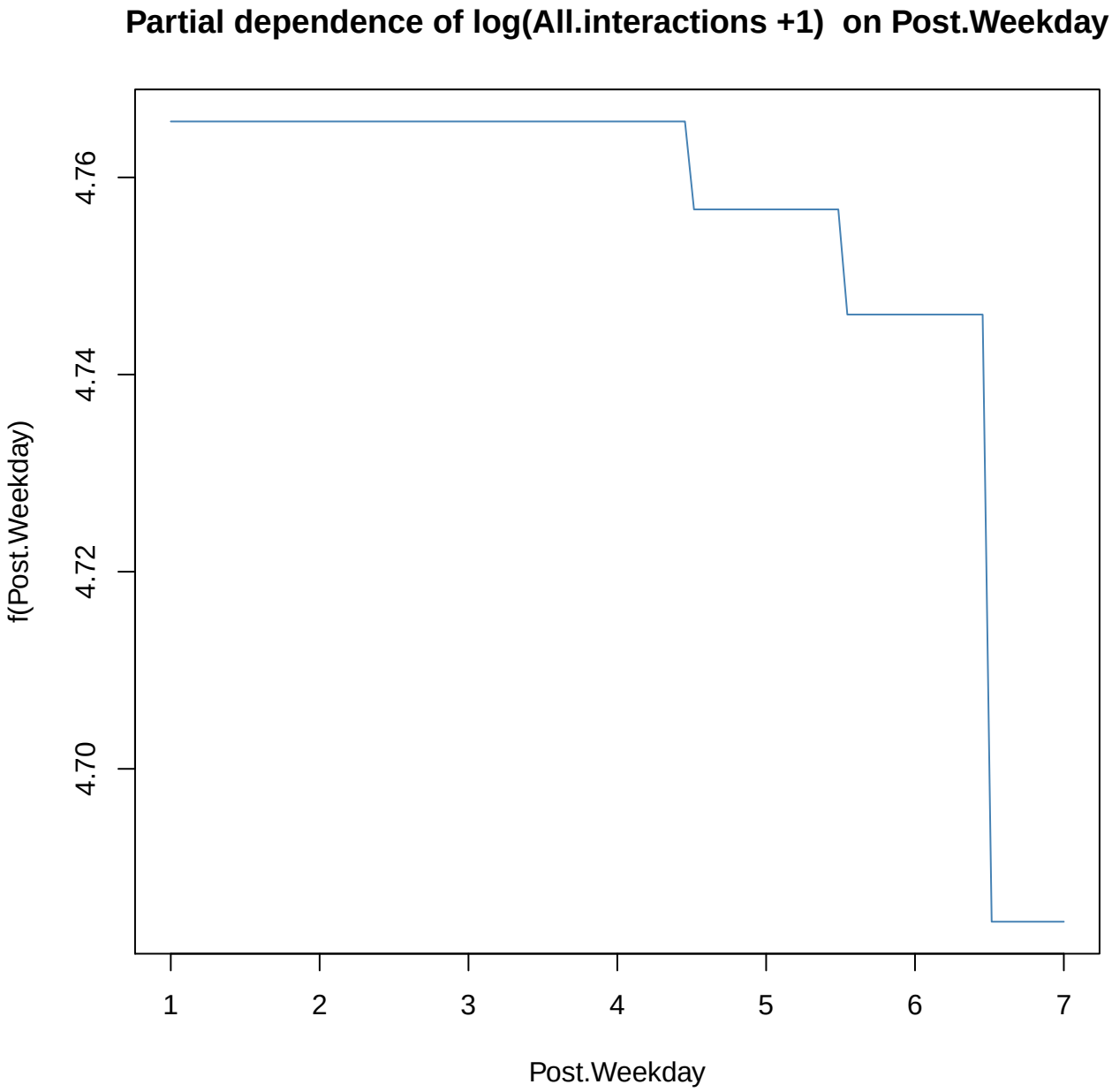
```
## Category         Category 82.836027
## Post.Weekday Post.Weekday  9.212498
## Post.Hour       Post.Hour  3.252196
## Paid                 Paid  1.787805
## Post.Month     Post.Month  1.779483
## Type                 Type  1.131991
```

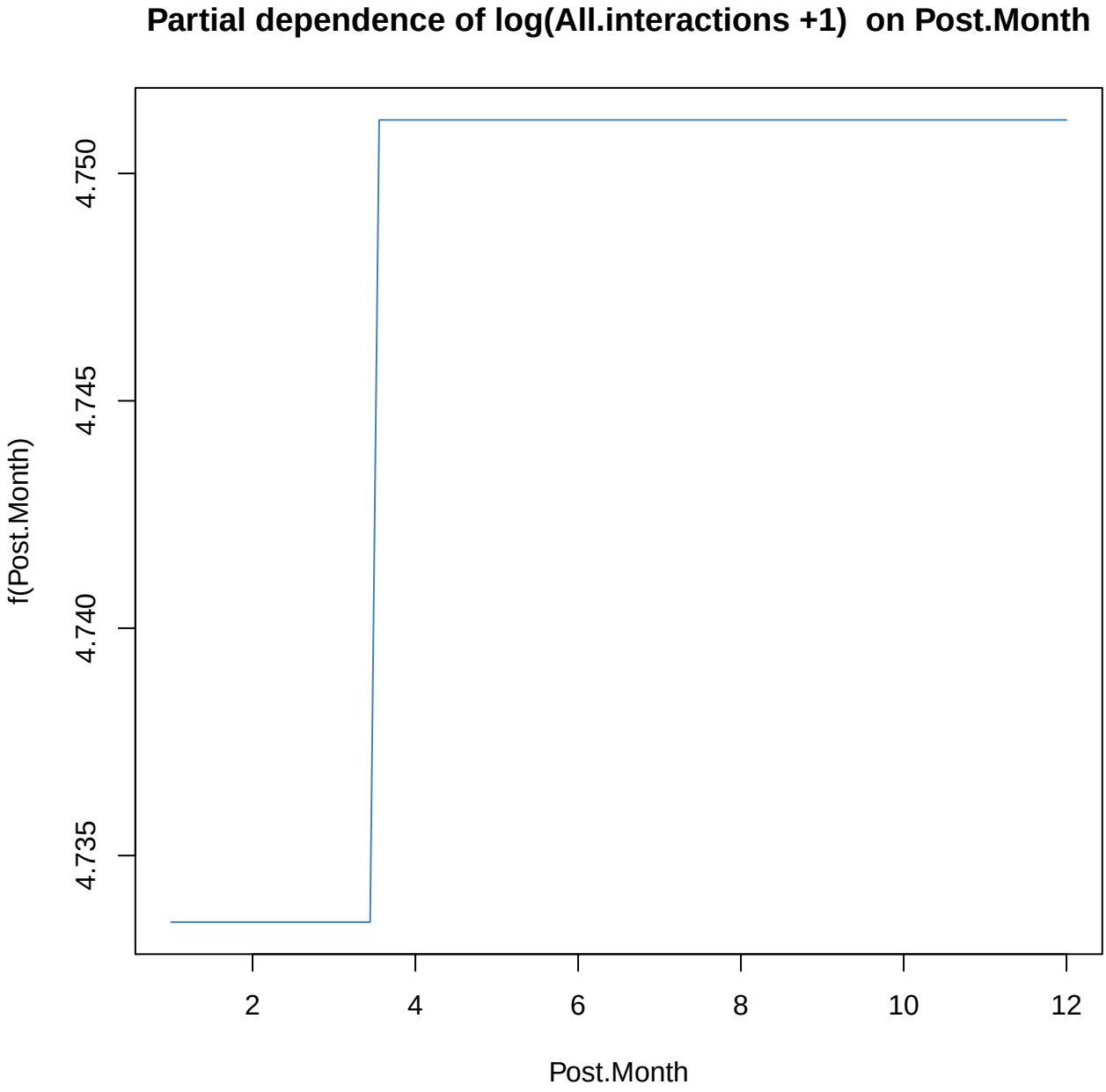
Now look at the dependence plots. First we look at one variate at a time. Think about how you would interpret each one in turn.

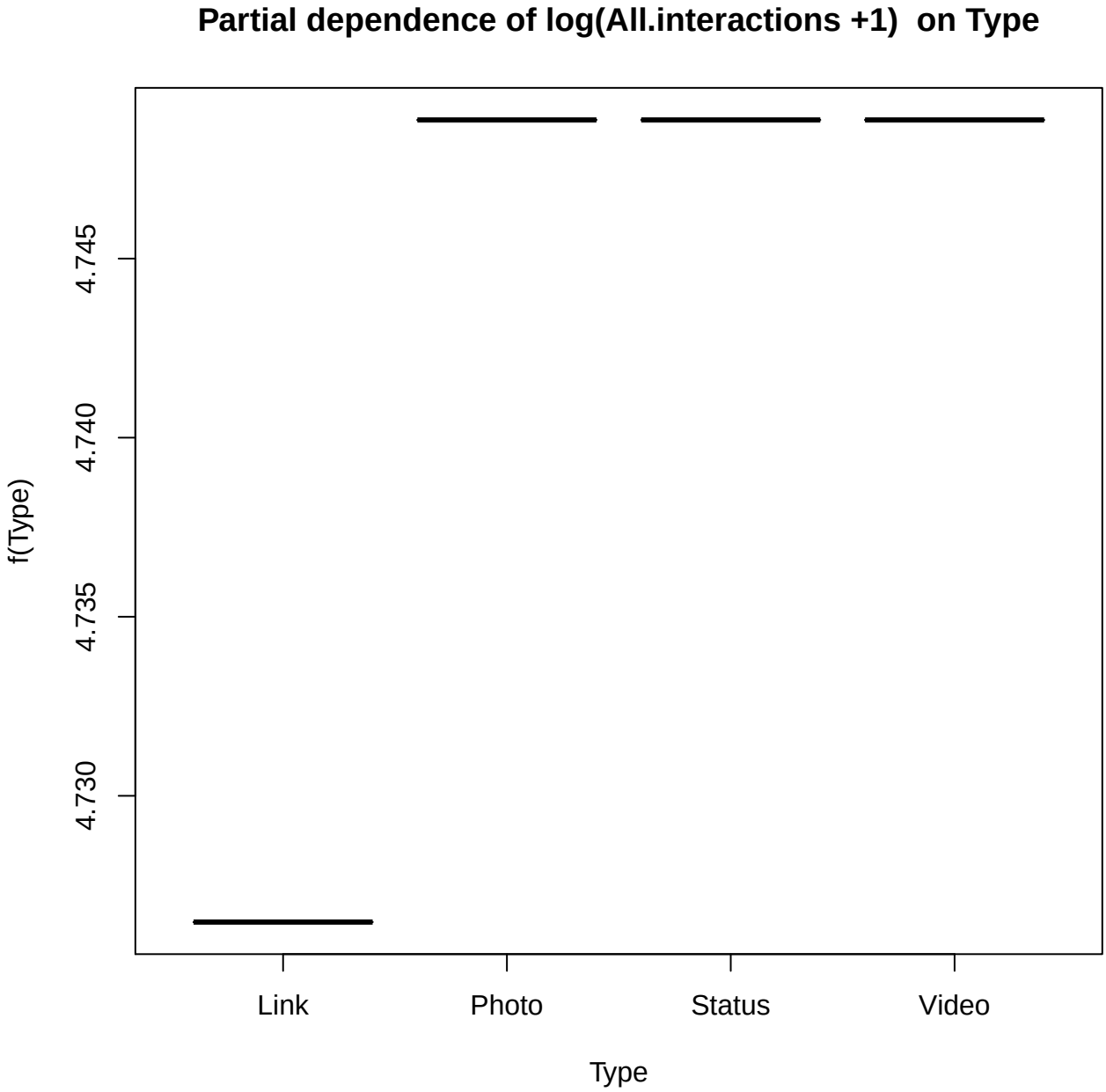
```
formula.gbm <- formula(fb.boost)
formula.sides <- strsplit(formula, "~")[[1]]
response.string <- formula.sides[1]
varnames <- get.explanatory_varnames(formula.gbm)
nvars <- length(varnames)
for (i in 1:nvars){
  plot(fb.boost, i.var = i, main = paste0("Partial dependence of ",
                                          response.string, " on ",
                                          varnames[i]),
       col="steelblue")
}
```

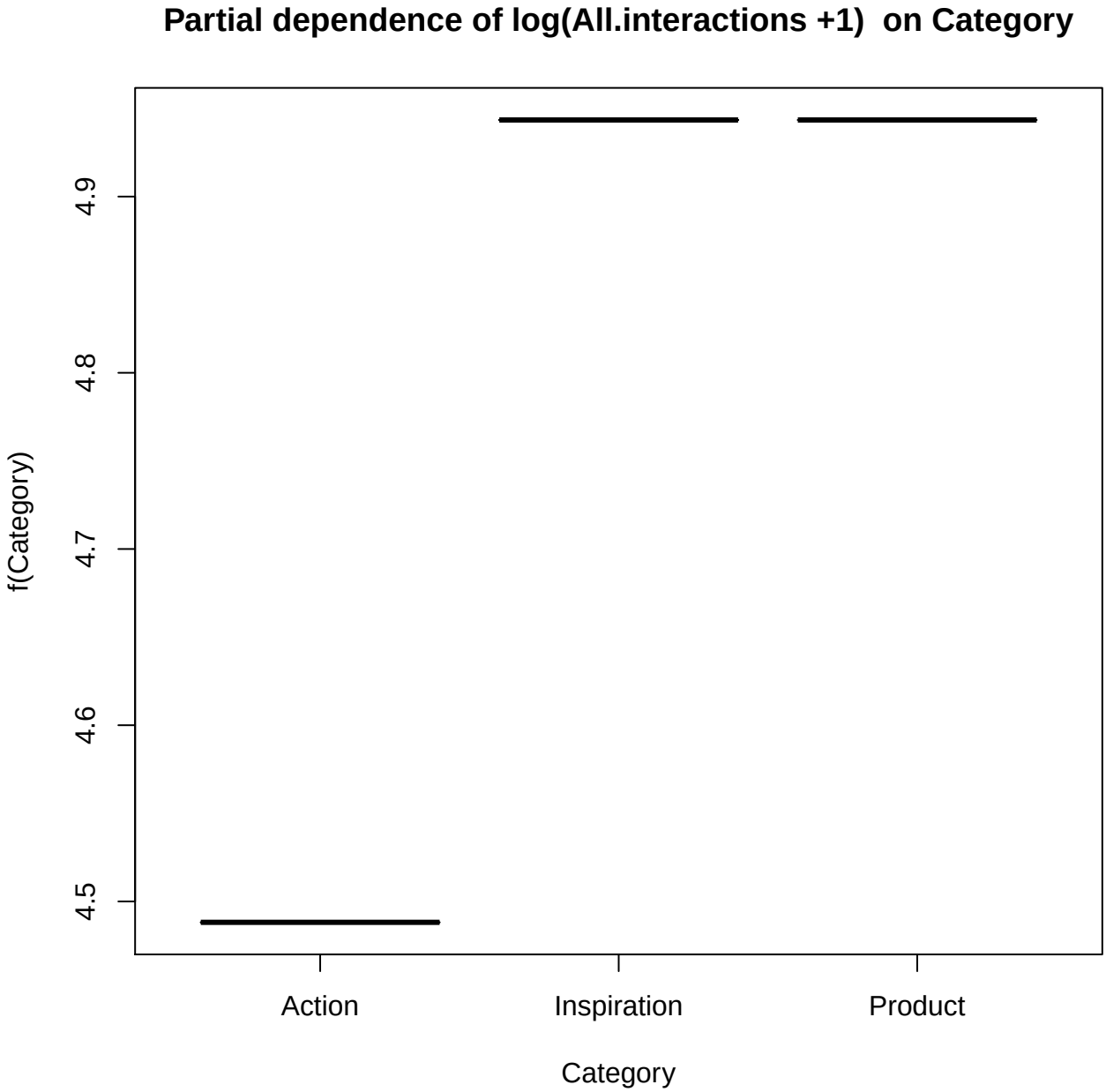

Partial dependence of log(All.interactions +1) on Paid
f(Paid)
4.745
4.750
4.755
4.760
0.0
0.2
0.4
0.6
0.8
1.0
Paid


Partial dependence of log(All.interactions +1) on Post.Hour
f(Post.Hour)
4.71
4.72
4.73
4.74
4.75
5
10
15
20
Post.Hour


Partial dependence of log(All.interactions +1) on Post.Weekday
f(Post.Weekday)
4.70
4.72
4.74
4.76
1
2
3
4
5
6
7
Post.Weekday


Partial dependence of log(All.interactions +1) on Post.Month
f(Post.Month)
4.735
4.740
4.745
4.750
2
4
6
8
10
12
Post.Month


Partial dependence of log(All.interactions +1) on Type
f(Type)
4.730
4.735
4.740
4.745
Link
Photo
Status
Video
Type



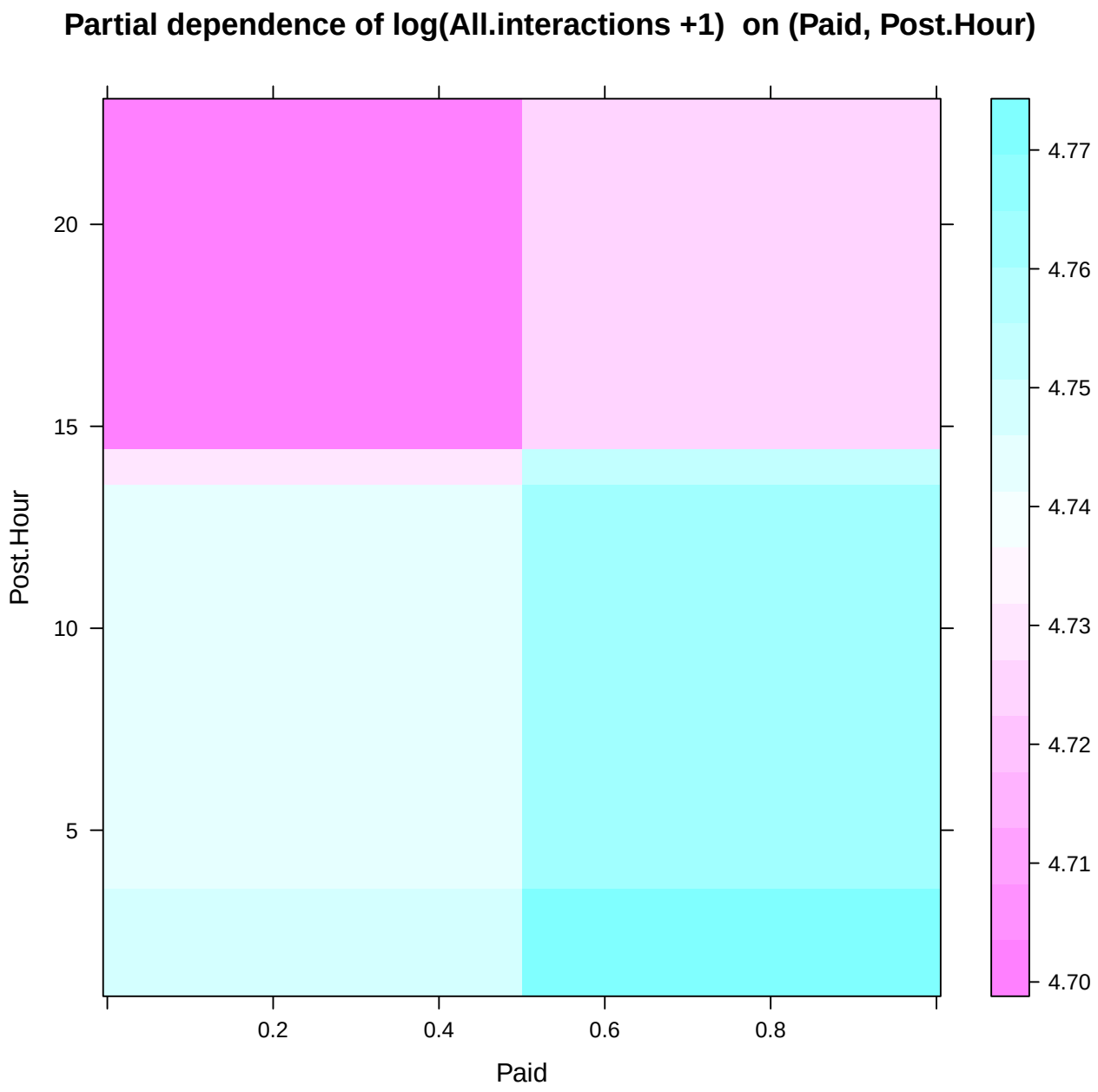
What do you think? This is especially good if we have **additive** fits (i.e. no two-way or higher order interactions).

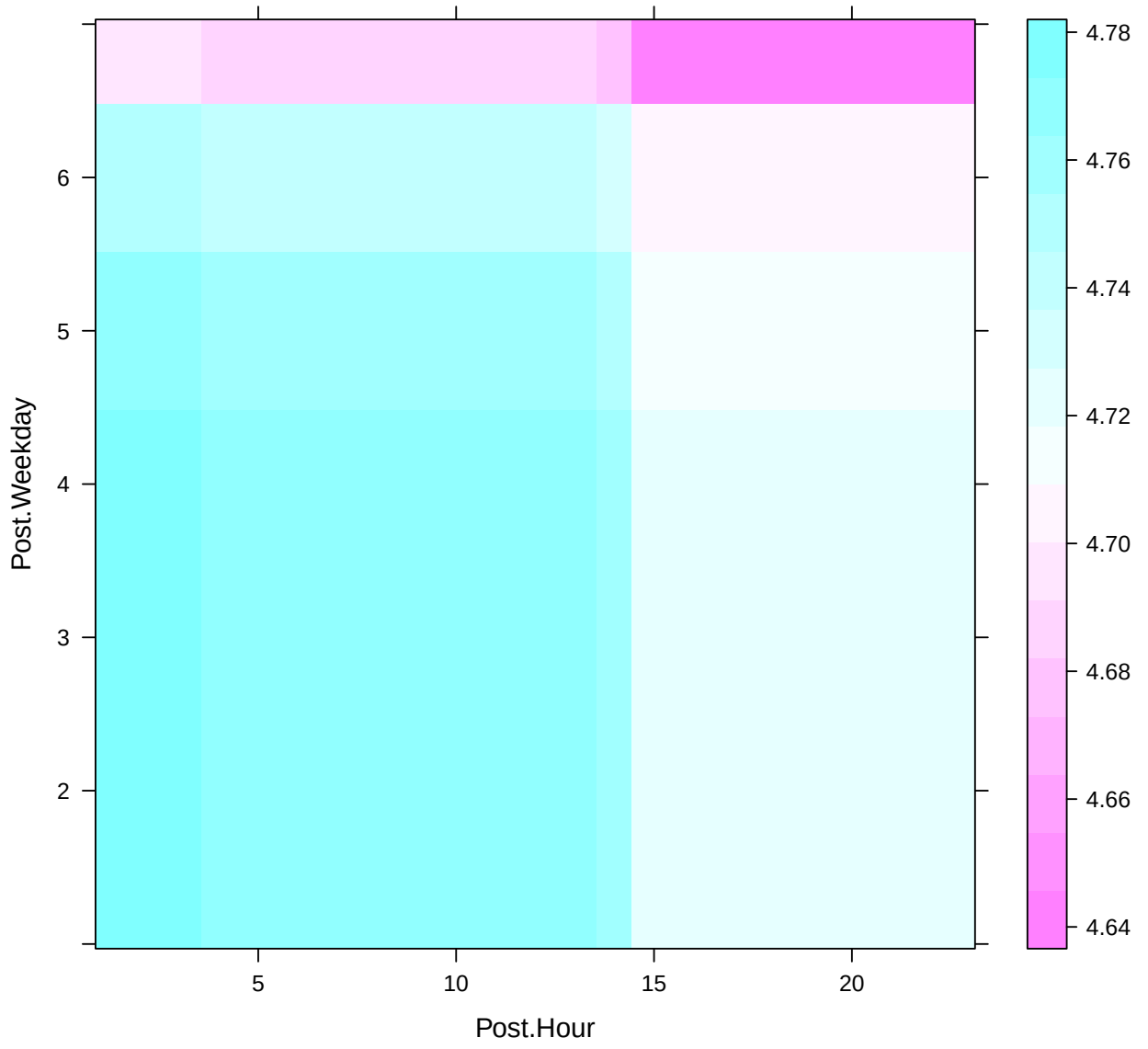
4.3.4.1 Pairs of variates

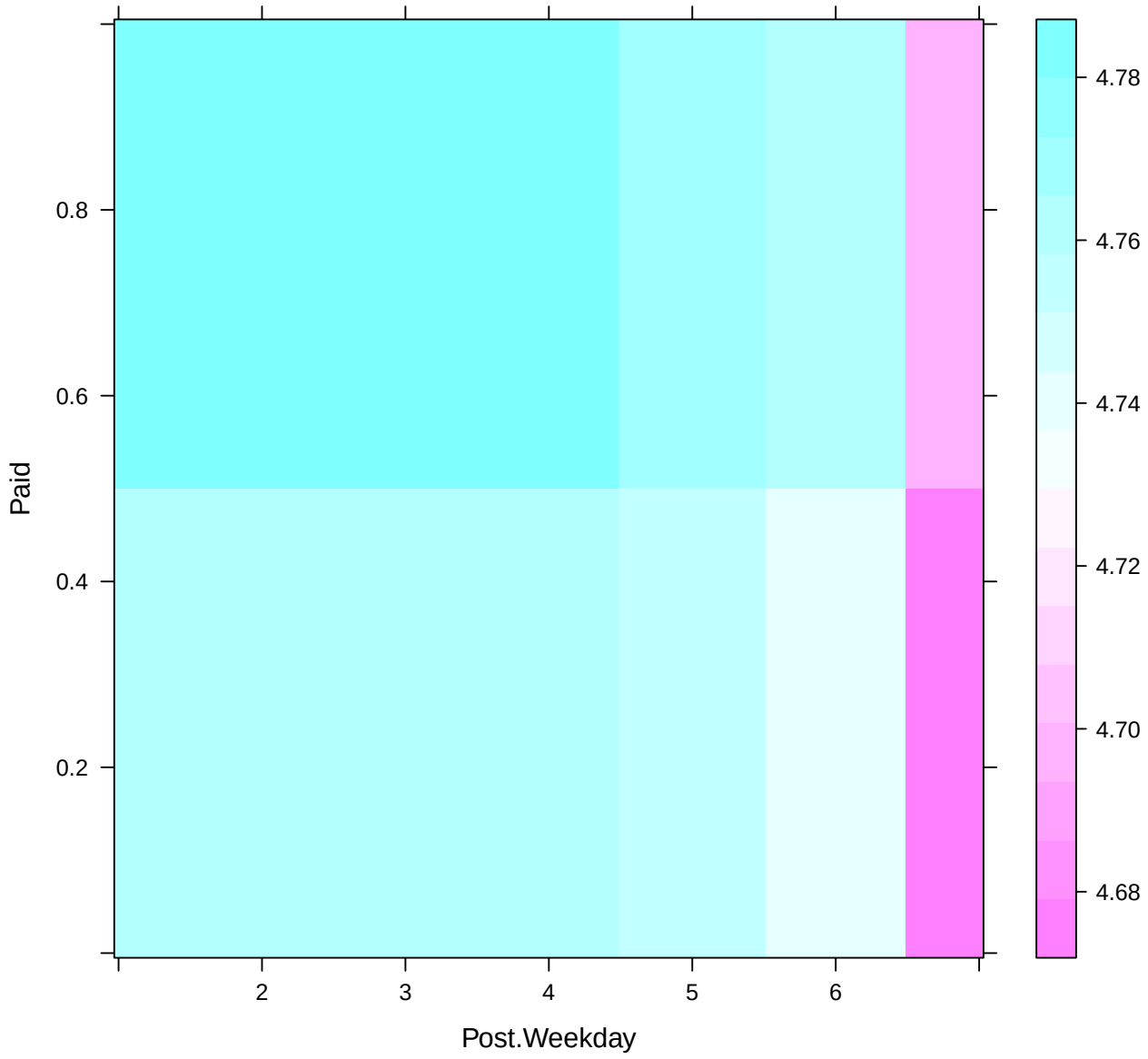
We could also look at the partial dependence on two variates. `plot(...)` here is smart enough to try to draw a plot that is appropriate for every combination! Take your time in interpreting these. Note, that in practice, one might only be interested in a subset of these. Note also how plots differ based on the nature of the variates.

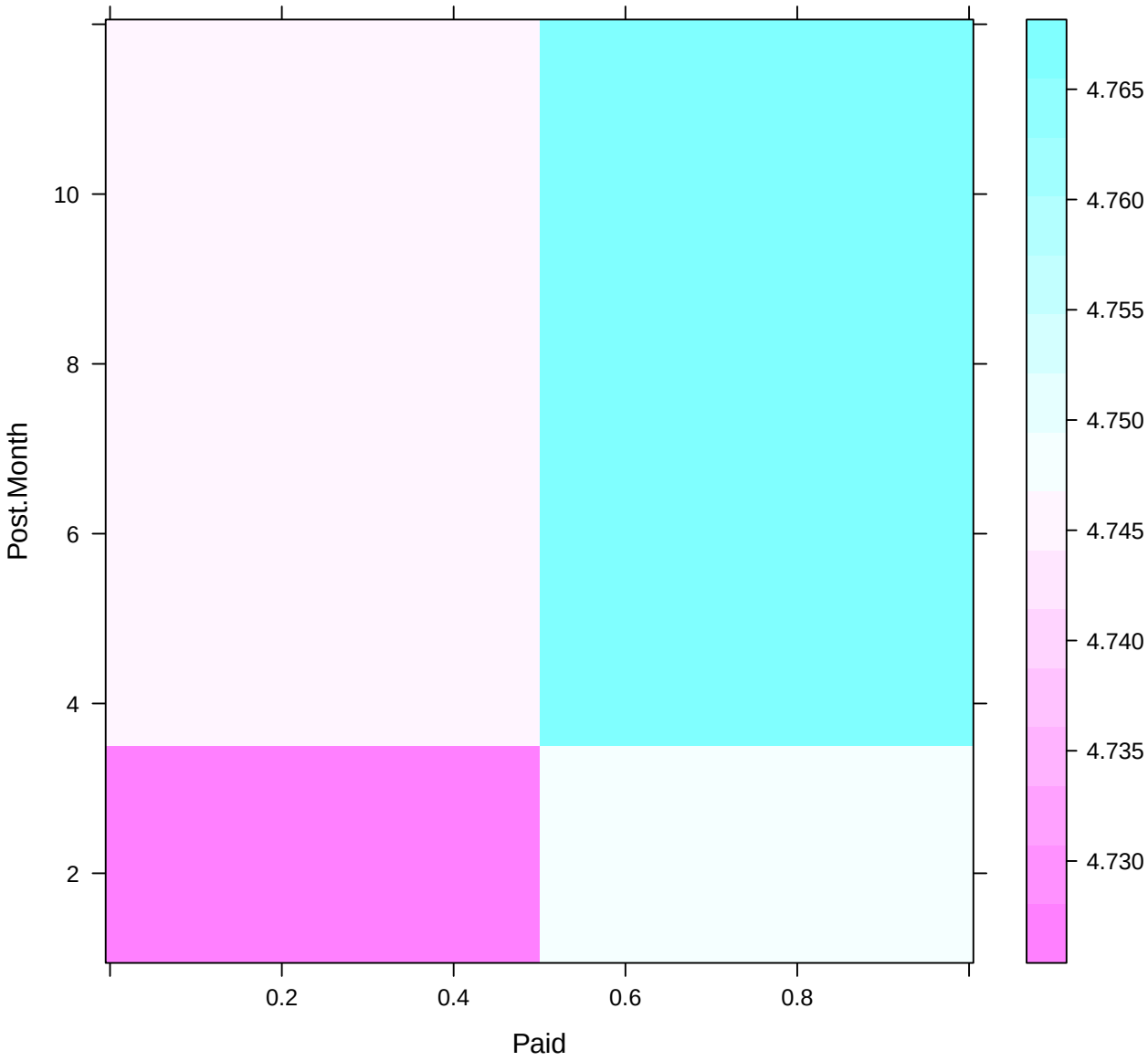
```
library(PairViz)  # PairViz package needed for sequences
all.pairs <- eseq(nvars)  # sequence where all pairs appear beside each other
for (i in 1:(length(all.pairs)-1)){
  indices <- all.pairs[i:(i+1)]
  plot(fb.boost, i.var = indices,
```

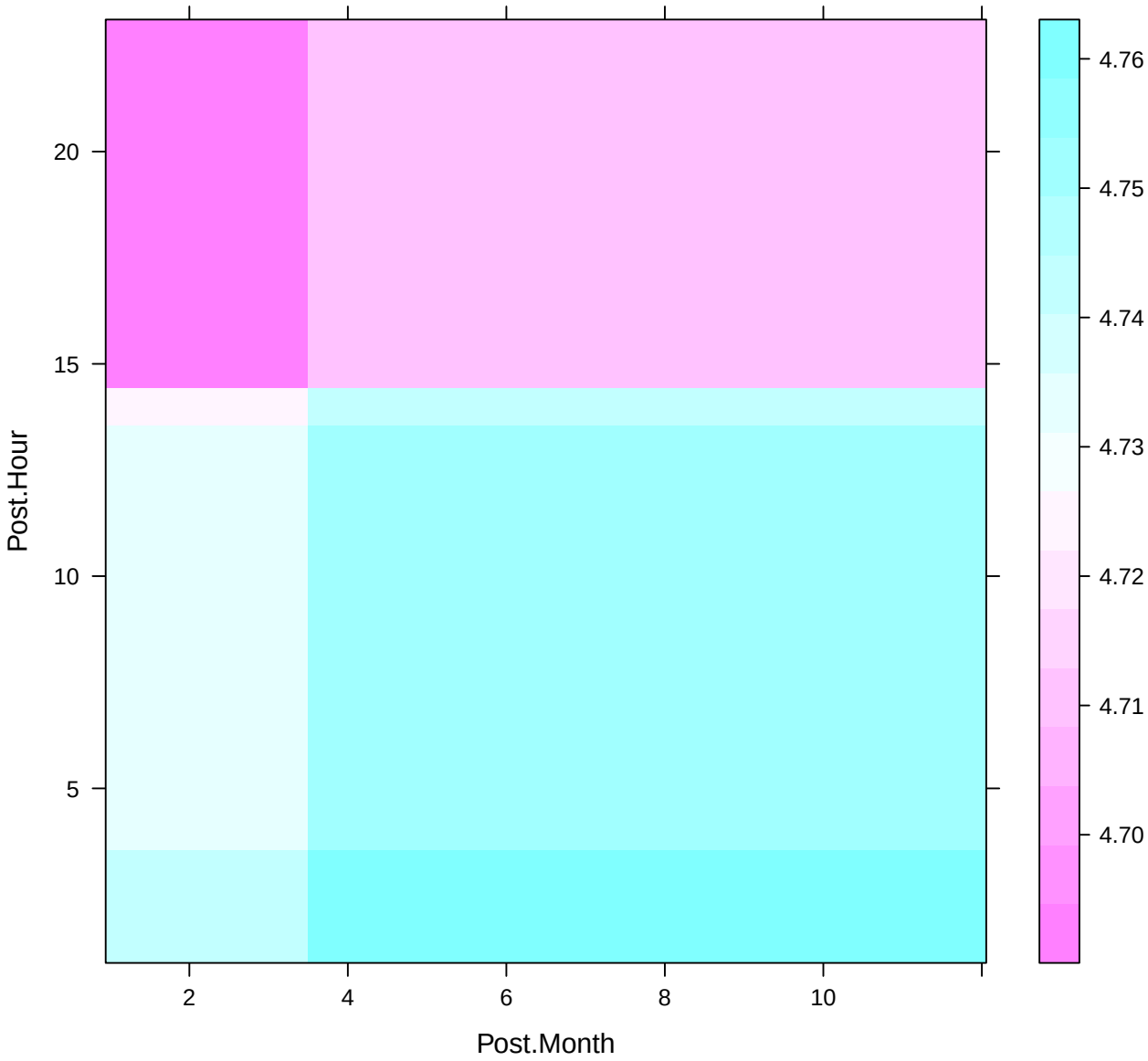
```
        main = paste0("Partial dependence of ",
                      response.string, " on (",
                      varnames[indices[1]],
                      ", " ,
                      varnames[indices[2]],
                      ")" )
        )
}
```

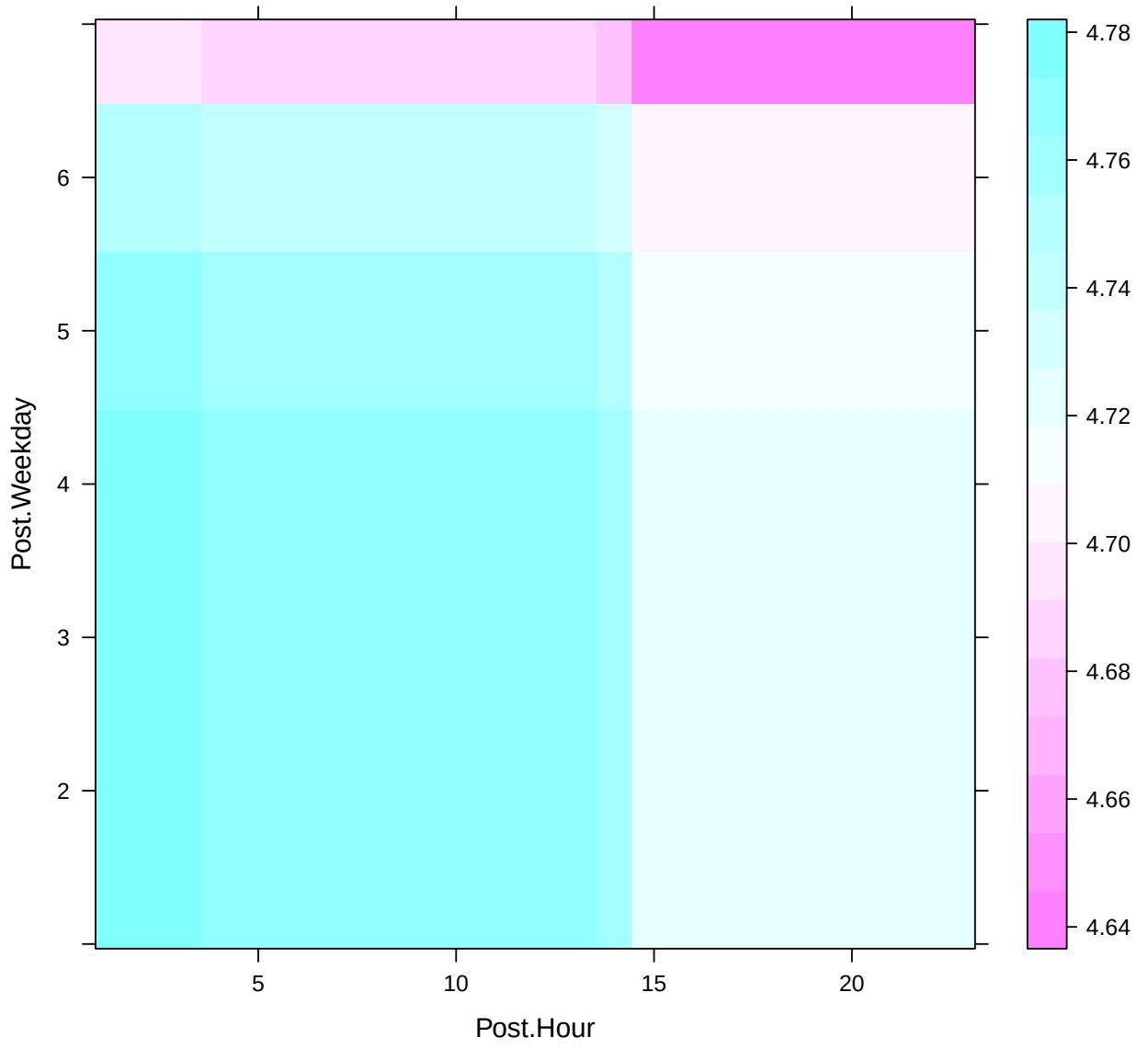


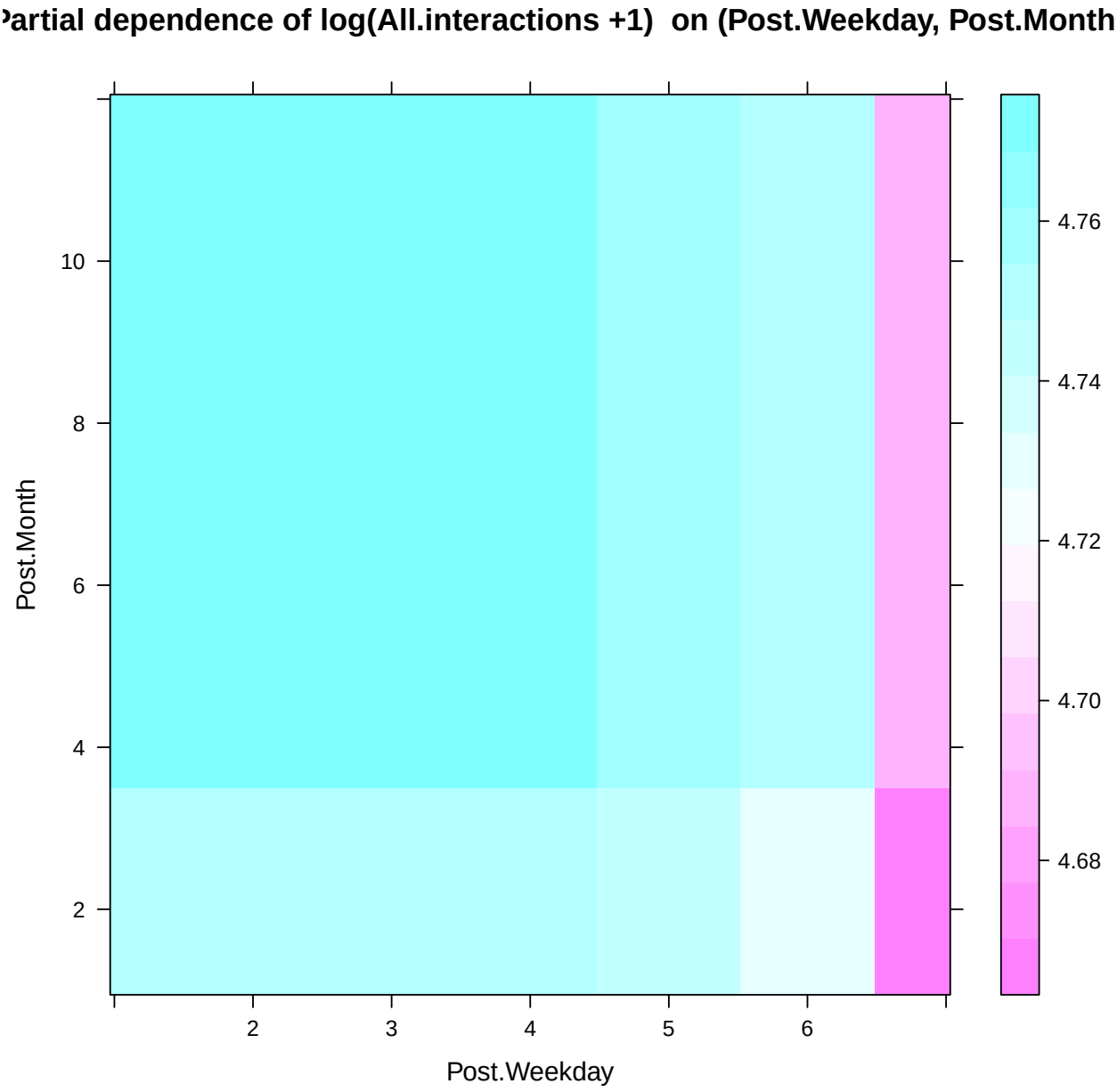

Partial dependence of log(All.interactions +1) on (Post.Hour, Post.Weekday)
Post.Weekday
Post.Hour
2
3
4
5
6
5
10
15
20
4.78
4.76
4.74
4.72
4.70
4.68
4.66
4.64

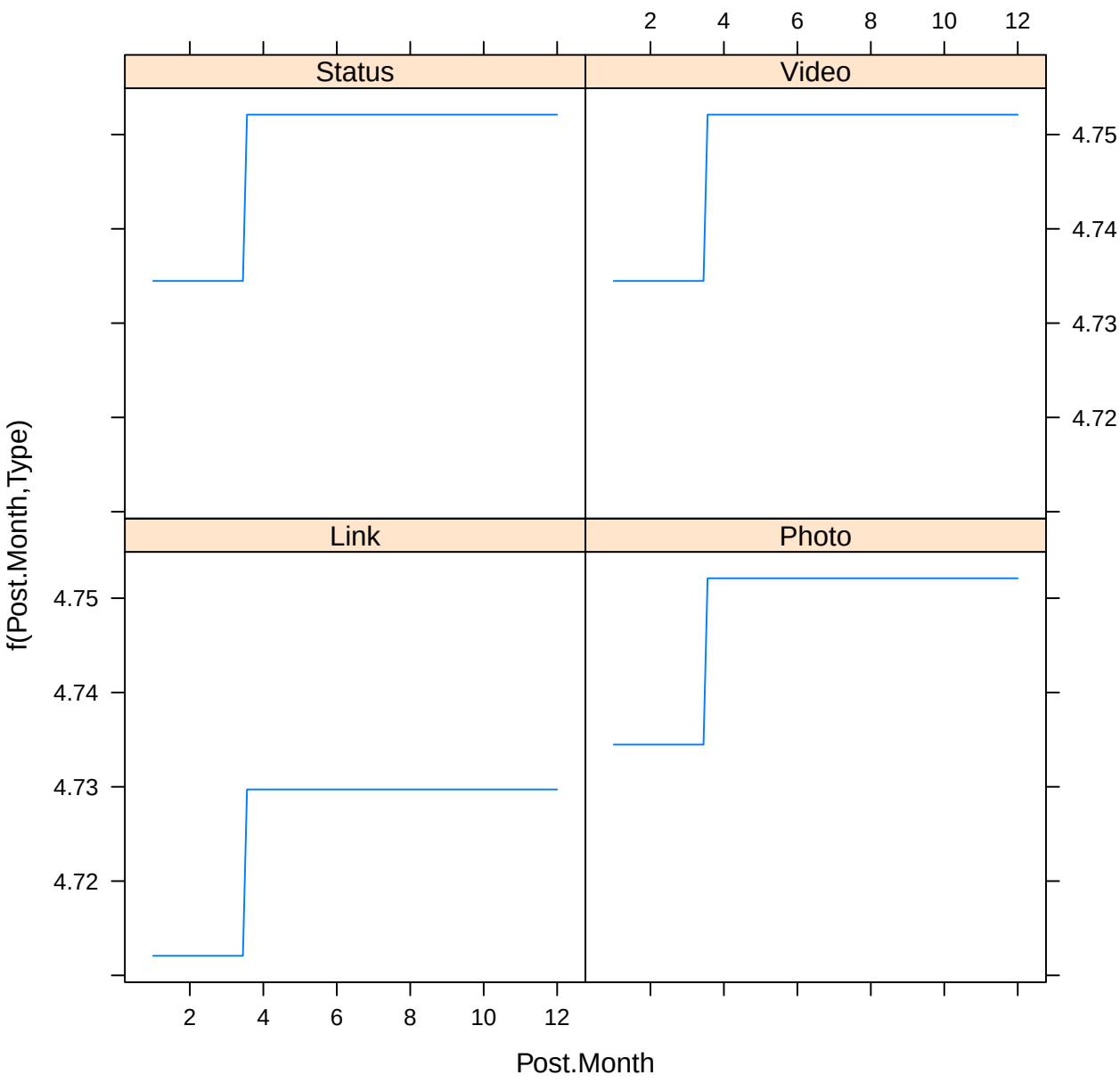

Partial dependence of log(All.interactions +1) on (Post.Weekday, Paid)
Paid
0.2
0.4
0.6
0.8
Post.Weekday
2
3
4
5
6
4.78
4.76
4.74
4.72
4.70
4.68

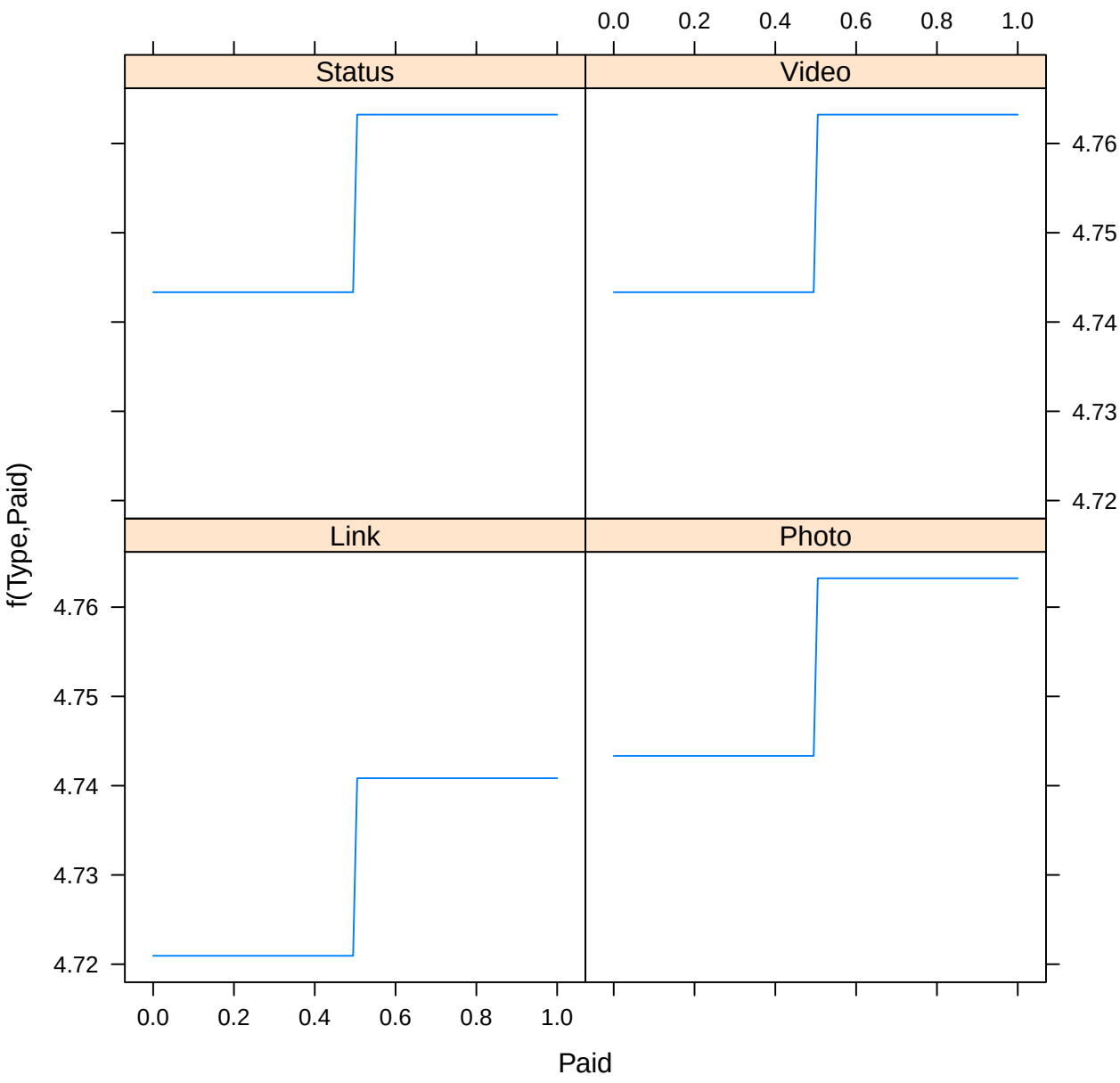

Partial dependence of log(All.interactions +1) on (Paid, Post.Month)
Post.Month
10
8
6
4
2
0.2
0.4
0.6
0.8
Paid
4.765
4.760
4.755
4.750
4.745
4.740
4.735
4.730

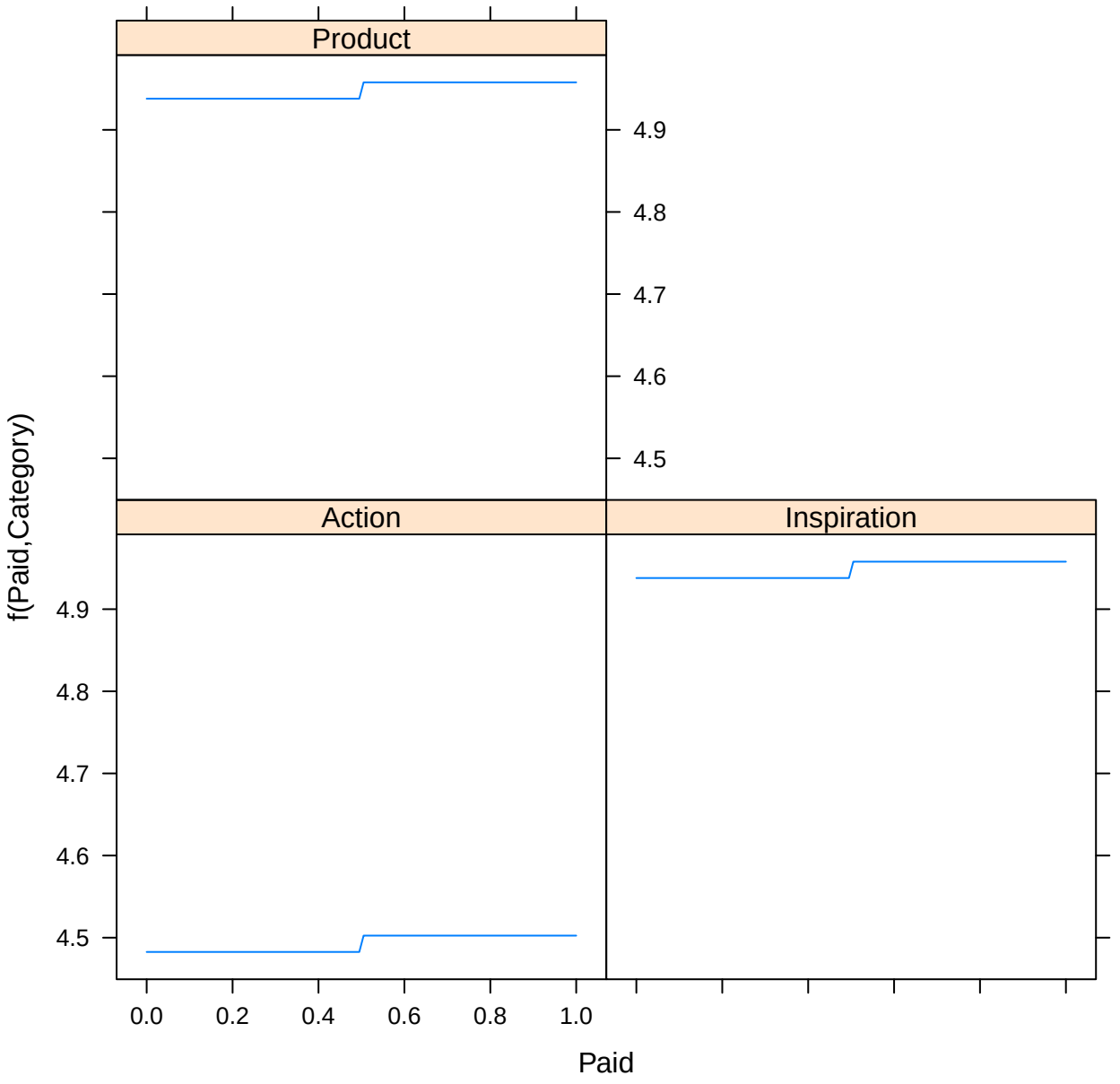

Partial dependence of log(All.interactions +1) on (Post.Month, Post.Hour)
Post.Hour
20
15
10
5
2
4
6
8
10
Post.Month
4.76
4.75
4.74
4.73
4.72
4.71
4.70

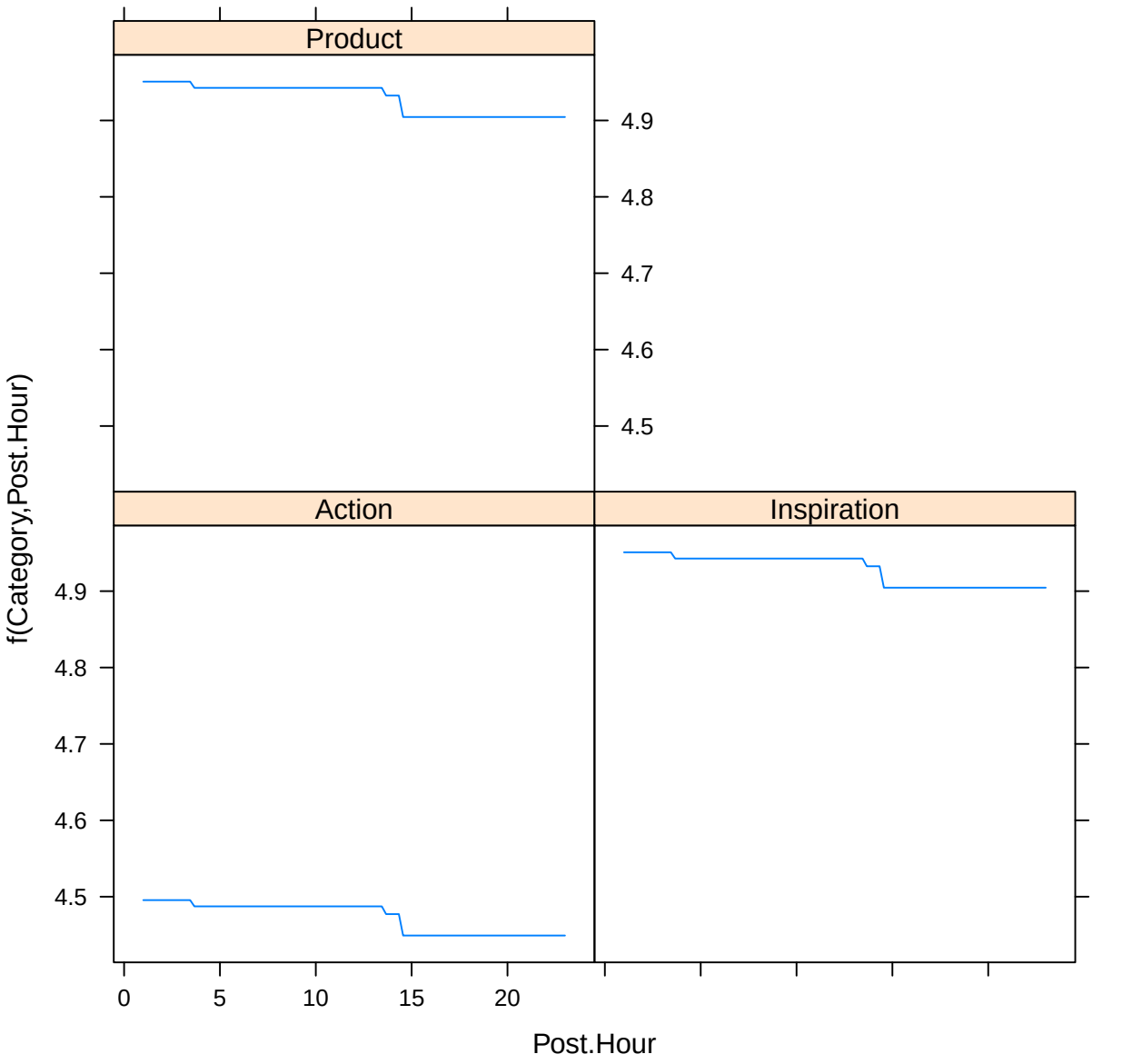

Partial dependence of log(All.interactions +1) on (Post.Hour, Post.Weekday)
Post.Weekday
Post.Hour
2
3
4
5
6
5
10
15
20
4.78
4.76
4.74
4.72
4.70
4.68
4.66
4.64

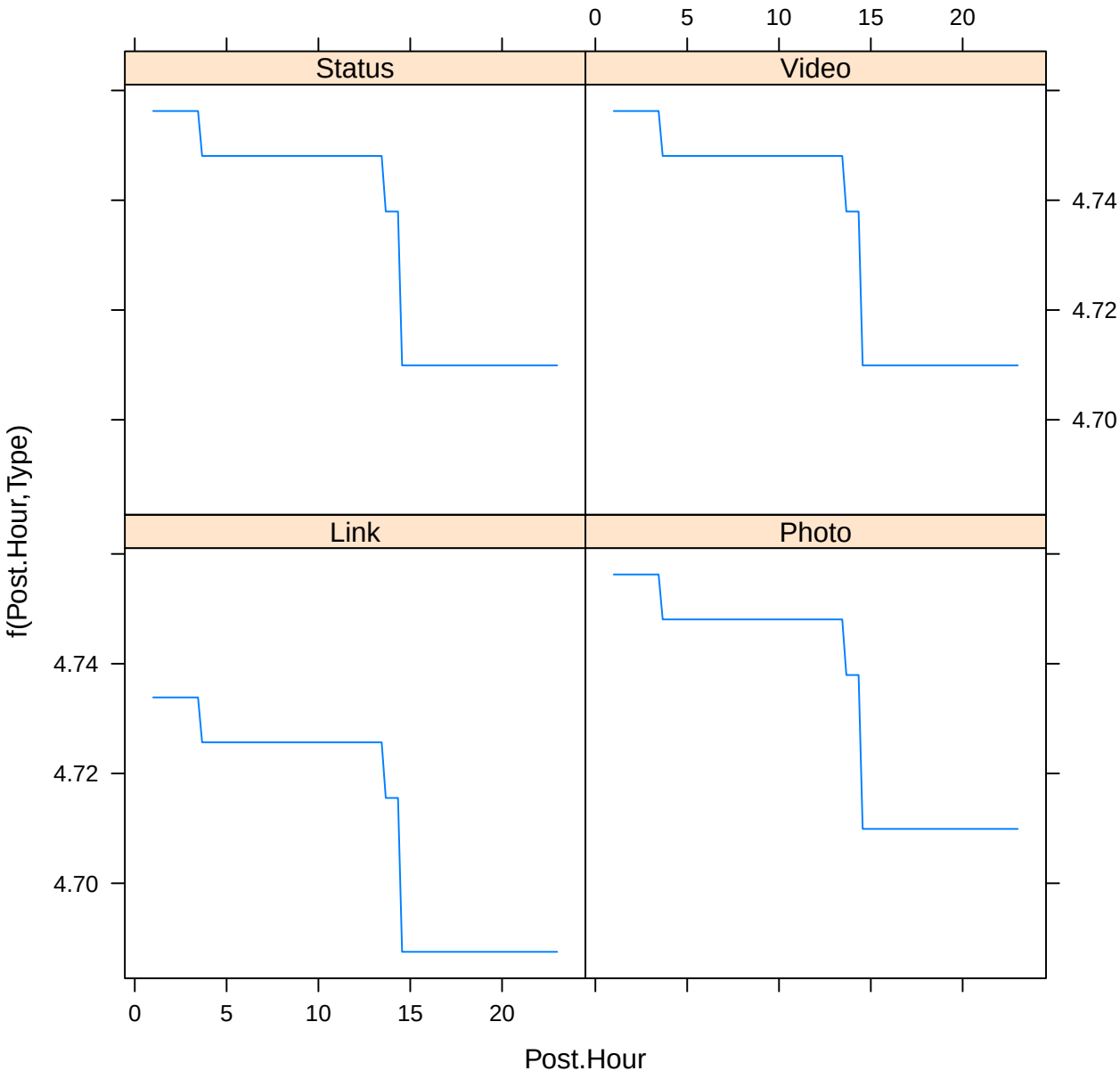

Partial dependence of log(All.interactions +1) on (Post.Weekday, Post.Month
Post.Month
Post.Weekday
2
4
6
8
10
2
3
4
5
6
4.76
4.74
4.72
4.70
4.68

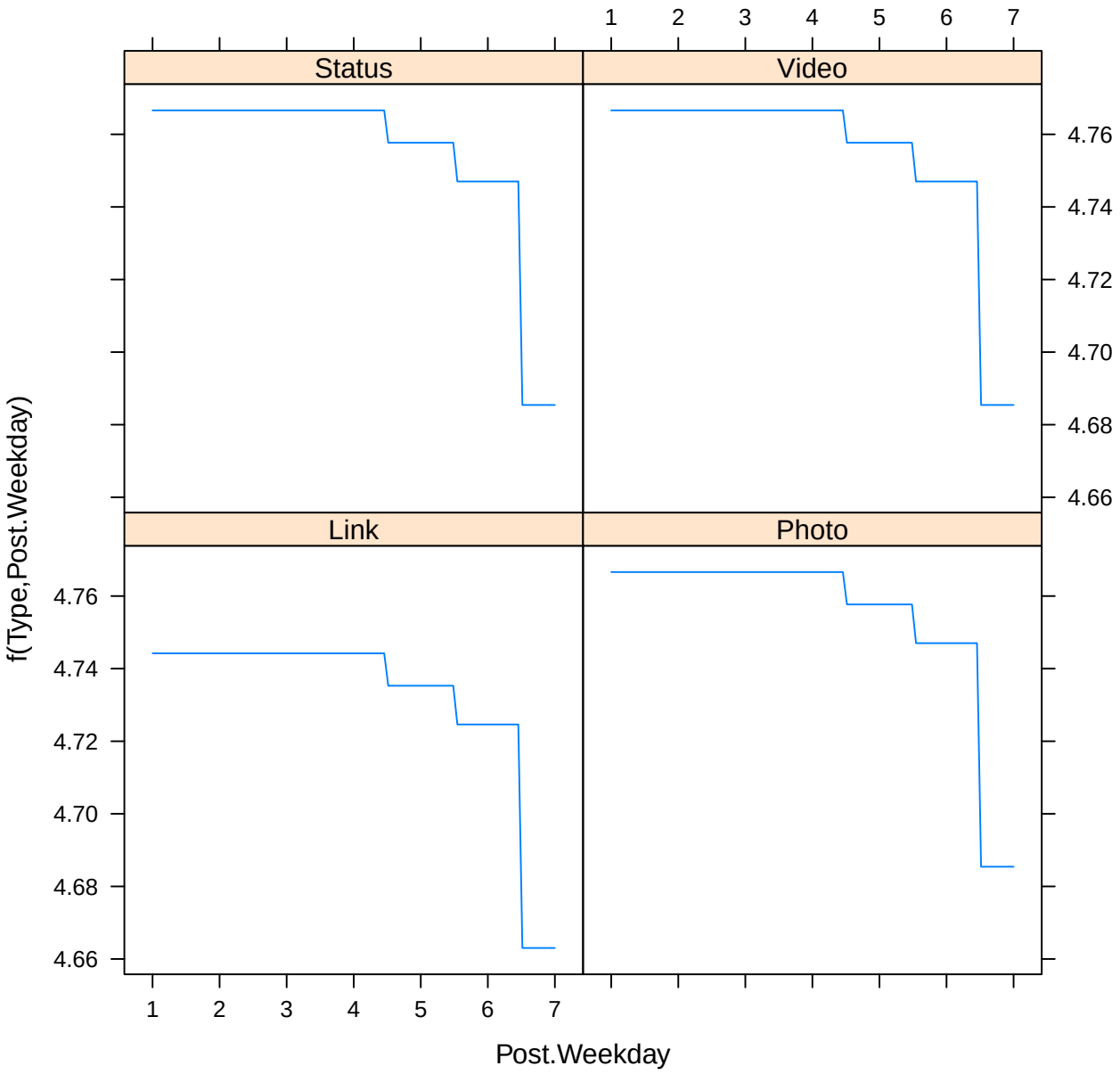

Partial dependence of log(All.interactions +1) on (Post.Month, Type)
Status
Video
Link
Photo
f(Post.Month,Type)
Post.Month
2
4
6
8
10
12
4.72
4.73
4.74
4.75


Partial dependence of log(All.interactions +1) on (Type, Paid)
Status
Video
Link
Photo
f(Type,Paid)
Paid
0.0
0.2
0.4
0.6
0.8
1.0
4.72
4.73
4.74
4.75
4.76

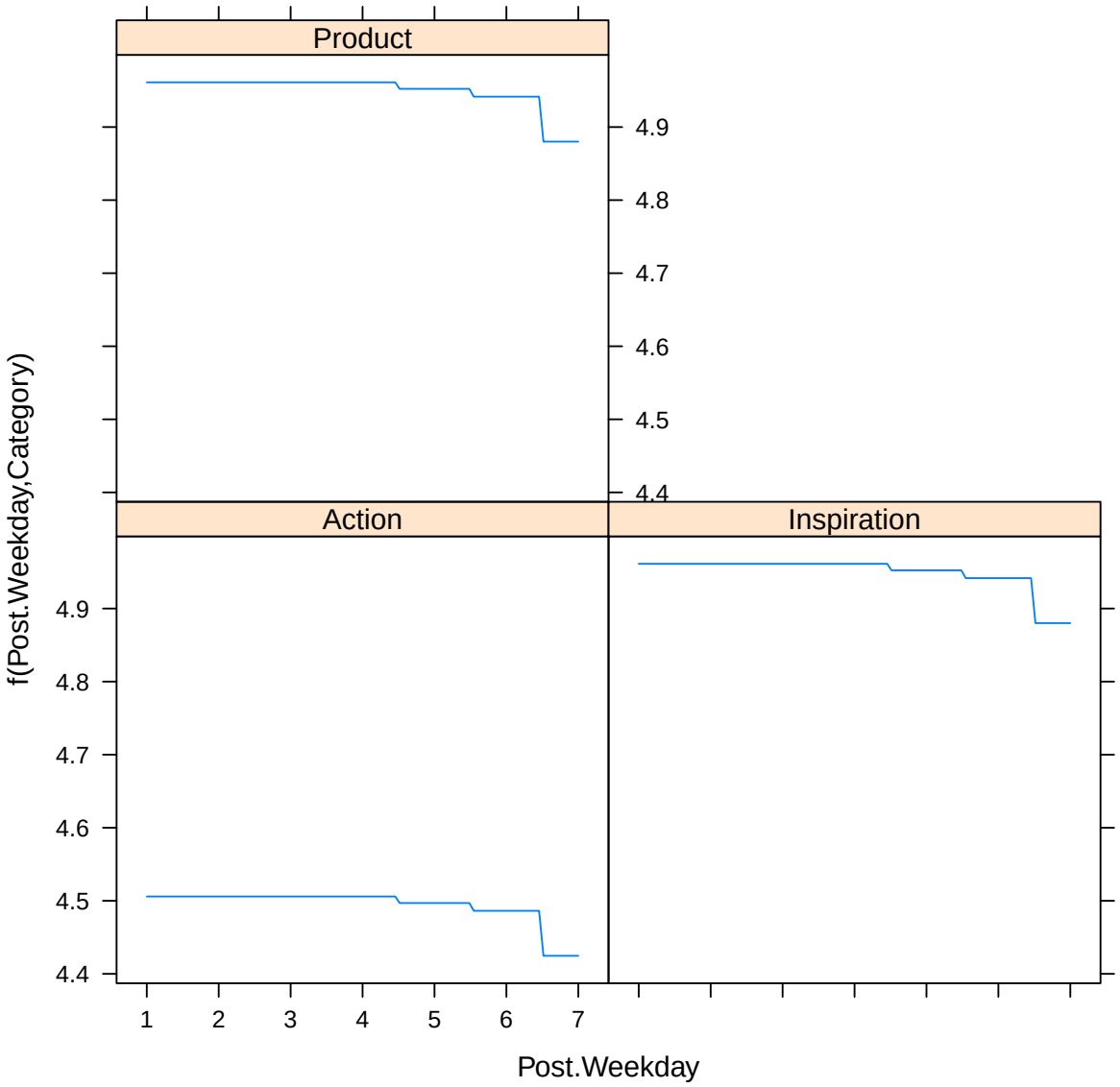

Partial dependence of log(All.interactions +1) on (Paid, Category)
Product
Action
Inspiration
f(Paid,Category)
4.9
4.8
4.7
4.6
4.5
0.0
0.2
0.4
0.6
0.8
1.0
Paid


Partial dependence of log(All.interactions +1) on (Category, Post.Hour)
Product
Action
Inspiration
f(Category,Post.Hour)
4.9
4.8
4.7
4.6
4.5
0
5
10
15
20
Post.Hour

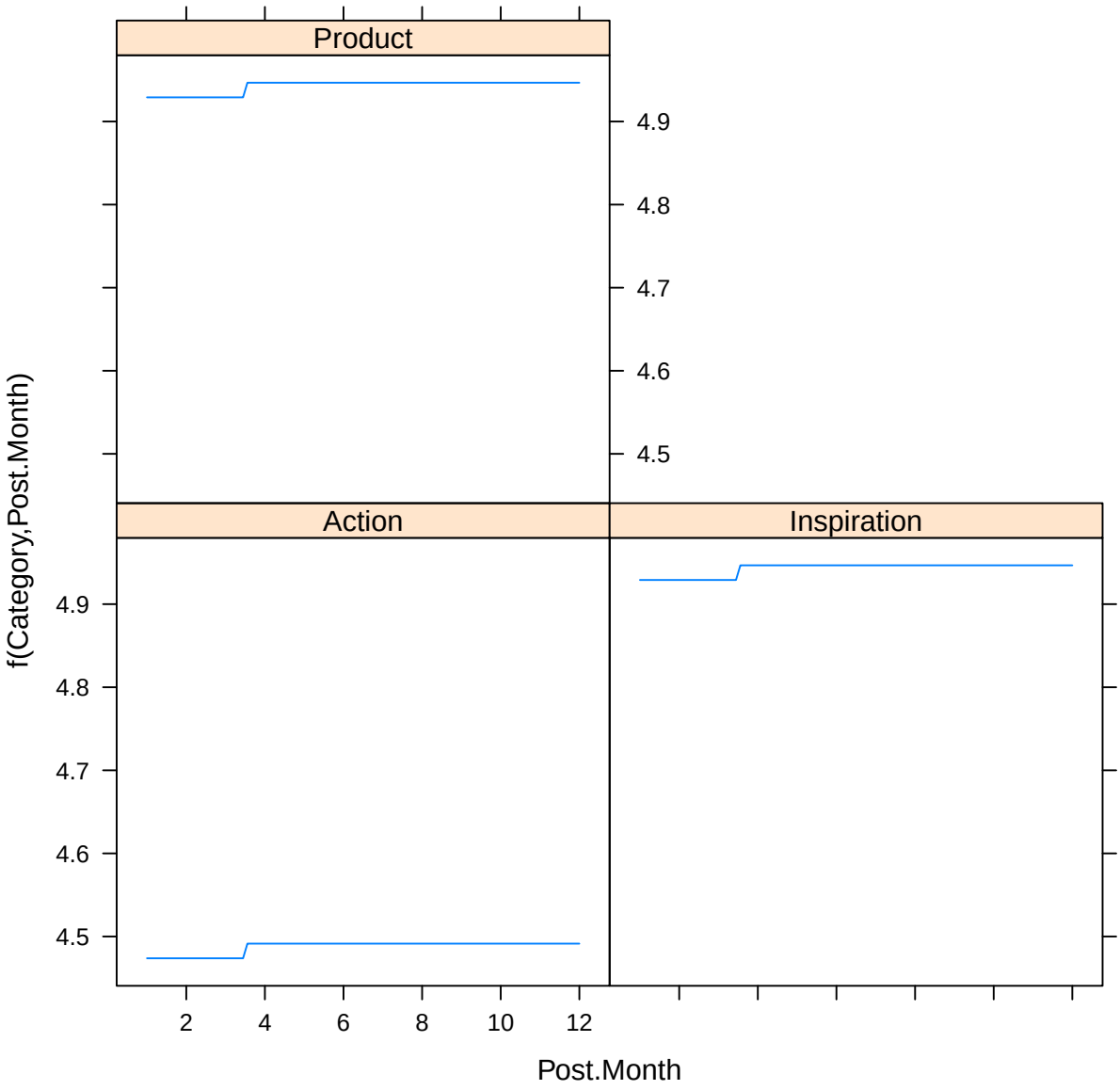

Partial dependence of log(All.interactions +1) on (Post.Hour, Type)
Status
Video
Link
Photo
f(Post.Hour,Type)
4.74
4.72
4.70
0
5
10
15
20
Post.Hour

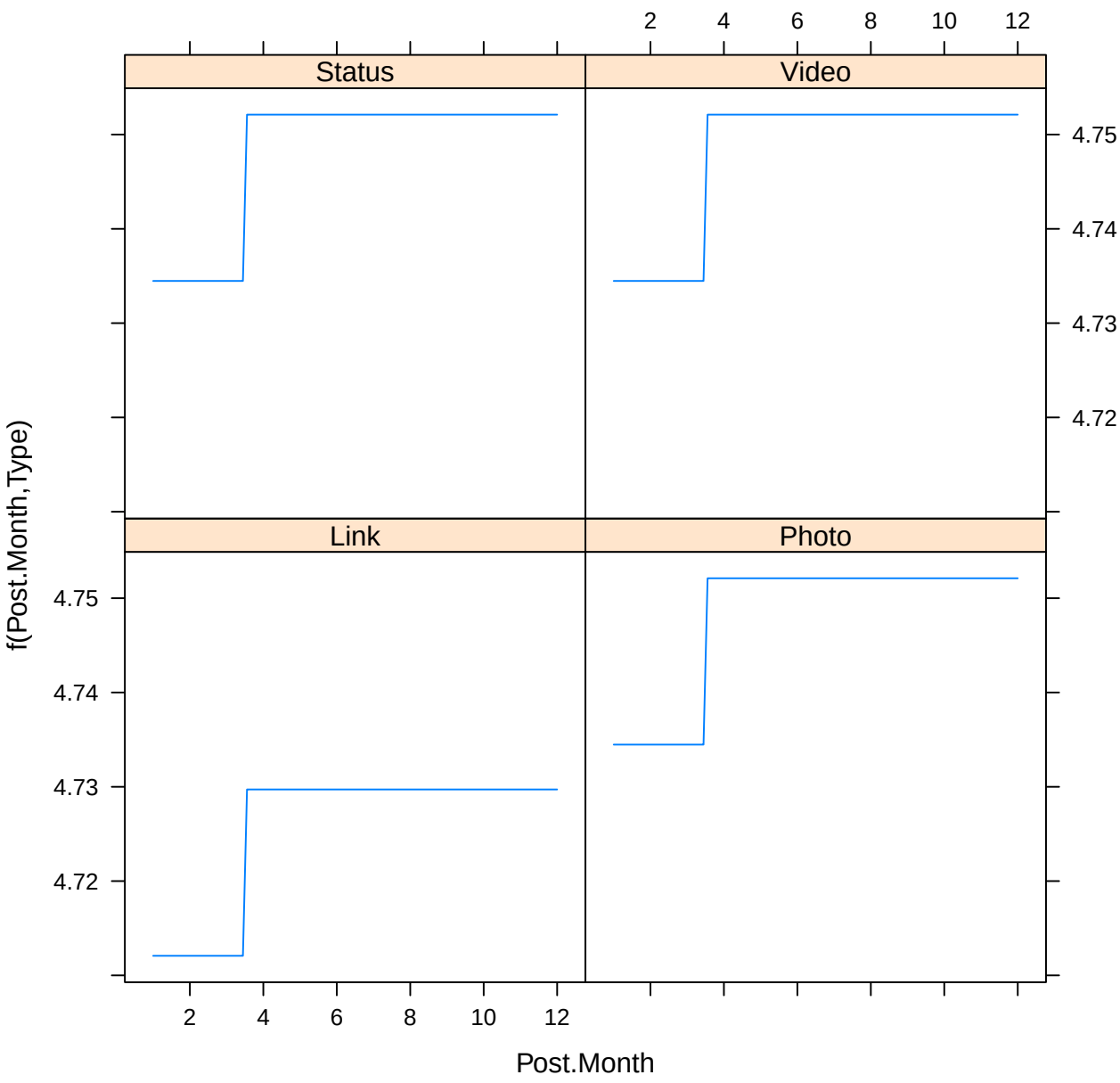

Partial dependence of log(All.interactions +1) on (Type, Post.Weekday)
Status
Video
Link
Photo
f(Type,Post.Weekday)
Post.Weekday
1
2
3
4
5
6
7
4.66
4.68
4.70
4.72
4.74
4.76

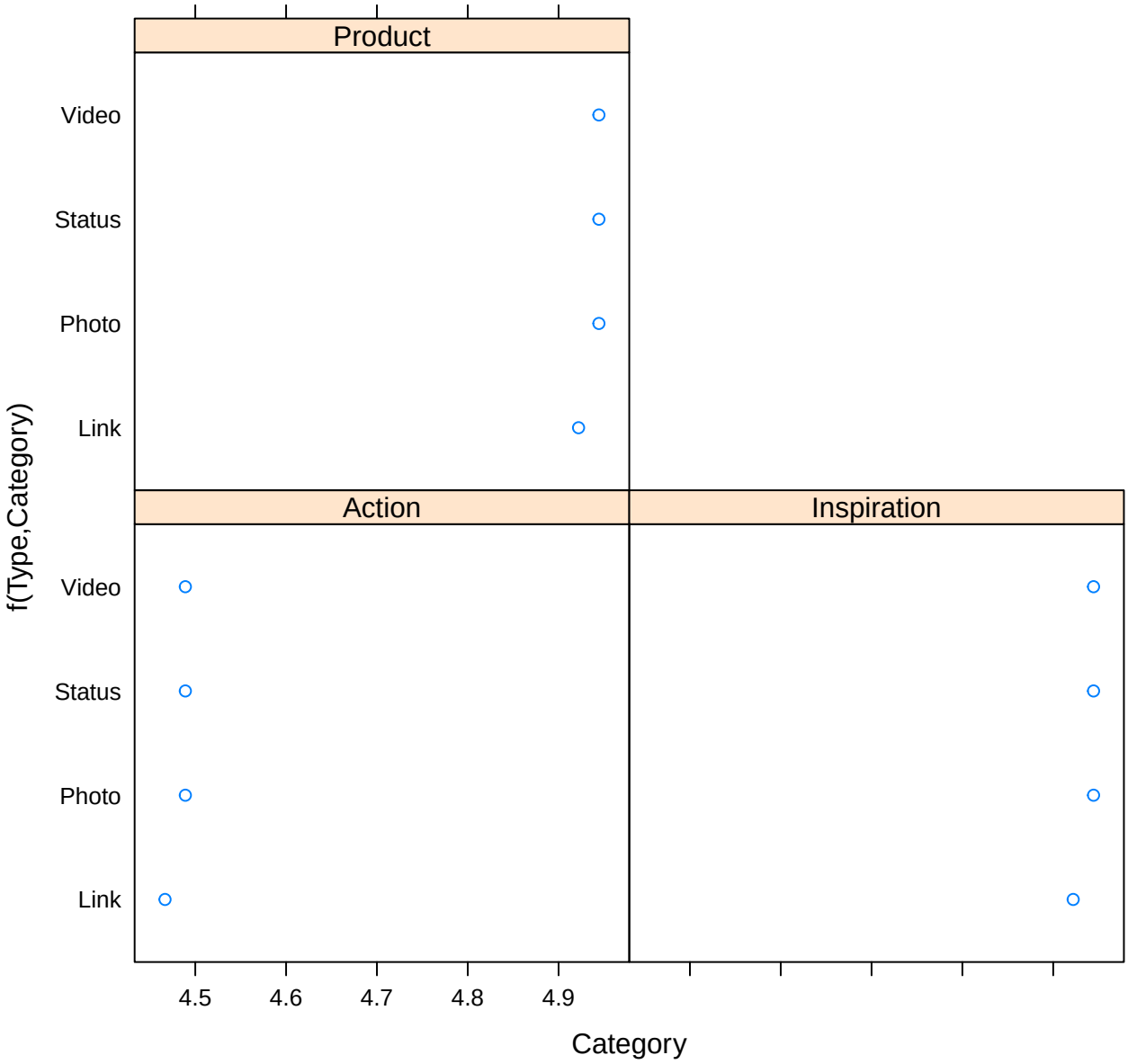
Partial dependence of log(All.interactions +1) on (Post.Weekday, Category)



Partial dependence of log(All.interactions +1) on (Category, Post.Month)




Partial dependence of log(All.interactions +1) on (Post.Month, Type)
Status
Video
Link
Photo
f(Post.Month,Type)
Post.Month
2
4
6
8
10
12
4.72
4.73
4.74
4.75



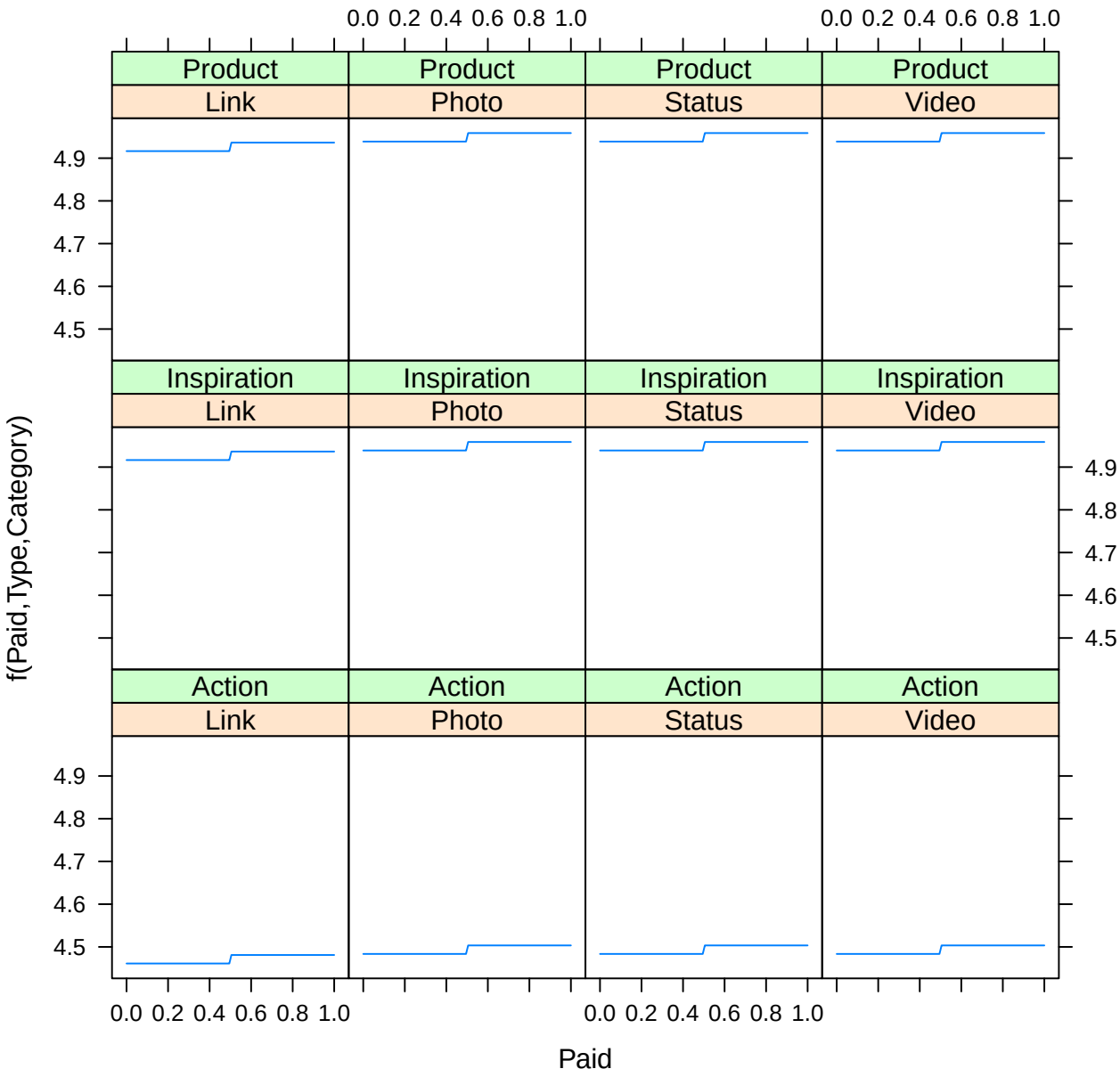
Note that we can compare the effect of two variates at once on the same plot.

4.3.4.2 Triples of variates

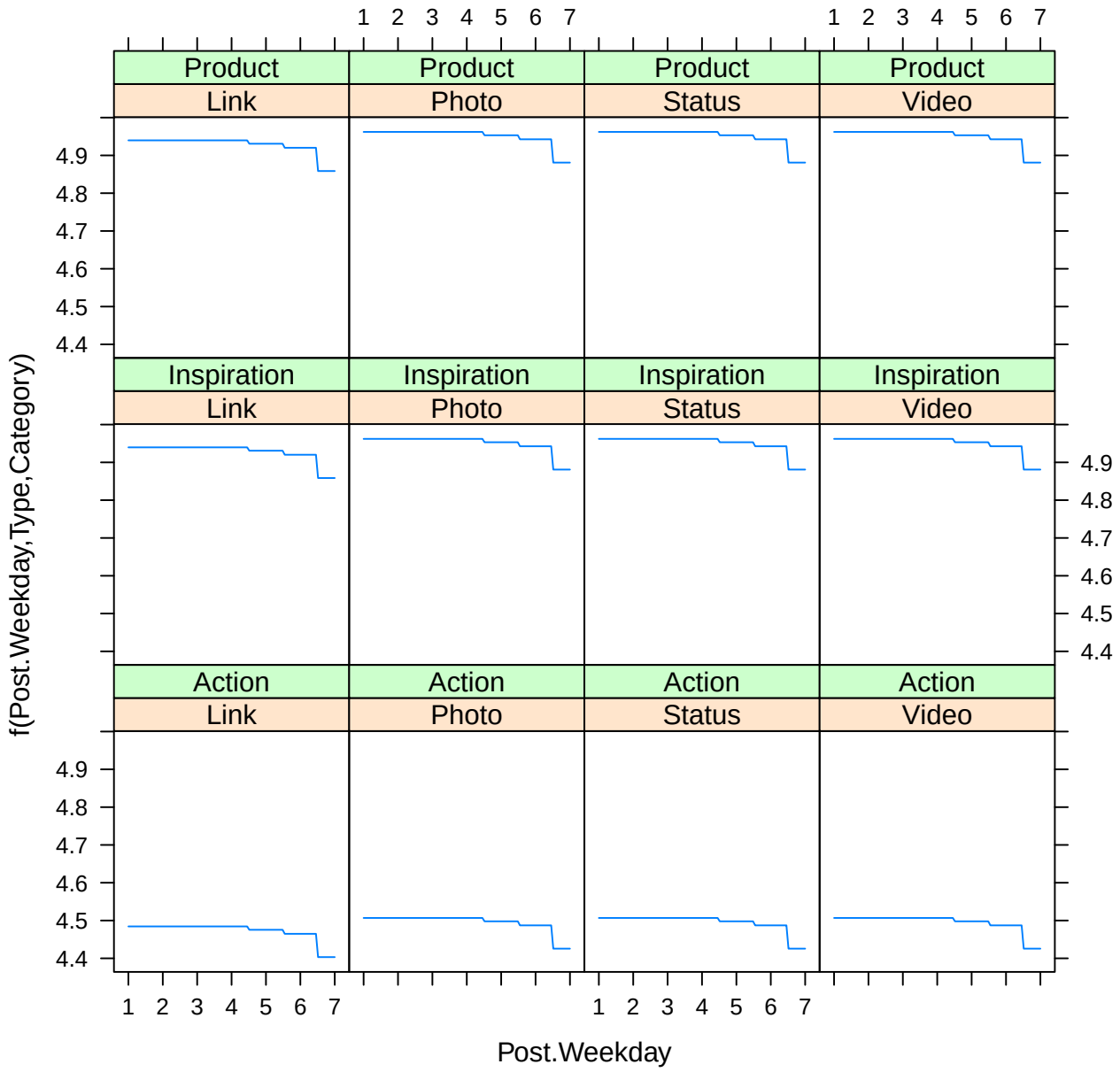
We can even try this for 3 variates at once, say on indices 1, 5, and 6.

```
do3way <- function(indices){
  plot(fb.boost, i.var = indices,
       main = paste0(response.string, " on (",
                     varnames[indices[1]],
                     ", " ,
                     varnames[indices[2]],
                     ", " ,
                     varnames[indices[3]],
```

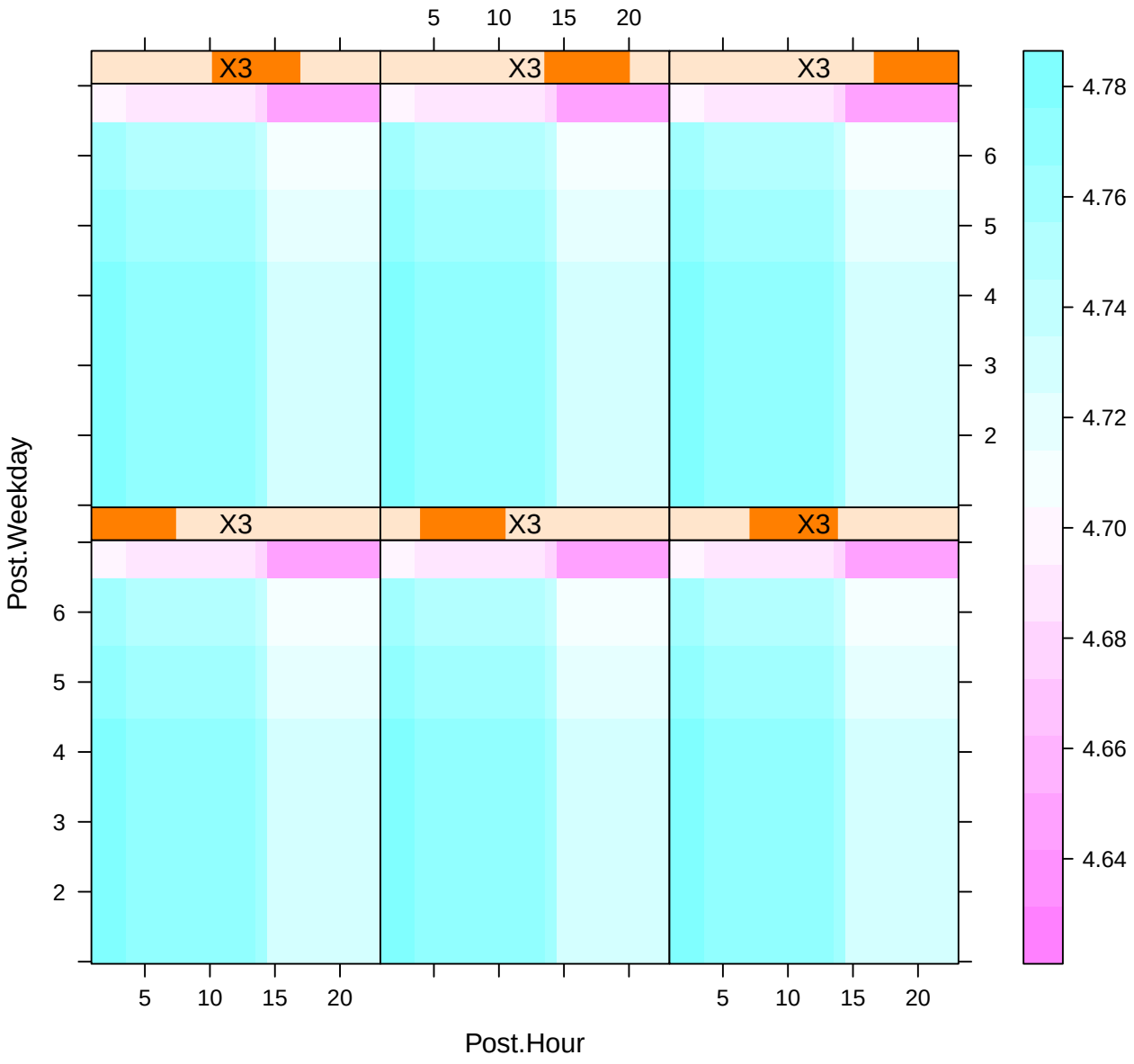
```
                    ")" )
  )
}
do3way(c(1,5,6))
```




```
do3way(c(3,5,6))
```


log(All.interactions +1) on (Post.Weekday, Type, Category)
Product
Link
Photo
Status
Video
Inspiration
Action
f(Post.Weekday,Type,Category)
Post.Weekday
1 2 3 4 5 6 7
4.4 4.5 4.6 4.7 4.8 4.9

```
do3way(c(2,3,4))
```


log(All.interactions +1) on (Post.Hour, Post.Weekday, Post.Month)
X3
X3
X3
X3
X3
X3
Post.Weekday
Post.Hour
5
10
15
20
2
3
4
5
6
4.78
4.76
4.74
4.72
4.70
4.68
4.66
4.64