Plotting \texttt{rpart} trees with the \texttt{rpart.plot} package

Stephen Milborrow
November 25, 2016

Contents
1 Introduction 2
2 Quick start 2
3 Main arguments 2
4 FAQ 6
5 Compatibility with \texttt{plot.rpart} and \texttt{text.rpart} 9
6 Customizing the node labels 10
7 Examples using the color and palette arguments 15
8 Branch widths 24
9 Trimming a tree with the mouse 25
10 Using \texttt{plotmo} in conjunction with \texttt{prp} 26
11 The graph layout algorithm 29

An Example

```
temp < 68
ibh >= 3574
dpg < −9.5
ibt < 226
temp >= 68
ibh < 3574
dpg >= −9.5
ibt >= 226
n=330  100%
n=214  65%
n=108  33%
n=106  32%
n=35  11%
n=71  22%
n=55  17%
n=61  18%
```

```
ozone
12
ozone
7.4
ozone
9.7
ozone
5.1
ozone
6.5
ozone
11
ozone
20
ozone
16
ozone
23
```

```
temp < 68
ibh >= 3574
dpg < −9.5
ibt < 226
temp >= 68
ibh < 3574
dpg >= −9.5
ibt >= 226
n=330  100%
n=214  65%
n=108  33%
n=106  32%
n=35  11%
n=71  22%
n=55  17%
n=61  18%
```
1 Introduction

The functions in the `rpart.plot` R package plot `rpart` trees [7, 8]. The next page shows some examples.

The workhorse function is `prp`. It automatically scales and adjusts the displayed tree for best fit. It combines and extends the `plot.rpart` and `text.rpart` functions in the `rpart` package.

Sections 2 and 3 of this document (the Quick Start and the Main Arguments) are the most important. The remaining sections may be skipped or read in any order.

I assume you have already looked at the vignette included with the `rpart` package [8]:

*An Introduction to Recursive Partitioning Using the RPART Routines* by Therneau and Atkinson.

2 Quick start

The easiest way to plot a tree is to use `rpart.plot`. This function is a simplified front-end to the workhorse function `prp`, with only the most useful arguments of that function. Its arguments are defaulted to display a tree with colors and details appropriate for the model’s response (whereas `prp` by default displays a minimal unadorned tree).

As described in the section below, the overall characteristics of the displayed tree can be changed with the `type` and `extra` arguments.

The intensity of a node’s color is proportional to the value predicted at the node. The color scheme can be changed with the `box.palette` argument. For details see the help page and Section 7.1. Examples:

```
box.palette = "auto"  automatically choose a palette (default for rpart.plot, Figure 1)
box.palette = 0       uncolored (white) boxes (default for prp)
box.palette = "Grays" a range of grays ("Grays" is one of the built in palettes)
box.palette = "gray"  uniform gray boxes
```

3 Main arguments

This section is an overview of the important arguments to `prp` and `rpart.plot`. For most users these arguments should suffice and the many other arguments can be ignored.

Use `type` to determine the basic plotting style, as shown in Figure 2.

Use `extra` to add more details to the node labels, as shown in Figures 3 and 4. Use `under=TRUE` to put those details under the boxes. With `extra="auto"` (the default for `rpart.plot`), a suitable value for `extra` will be chosen automatically (based on the type of response for the model). Figure 1 illustrates. The help page has details.

Use `digits`, `varlen`, and `faclen` to display more significant digits and more characters in names. In particular, use the special values `varlen=0` and `faclen=0` to display full variable and factor names.

The character size will be adjusted automatically unless `cex` is explicitly set. Use `tweak` to adjust the automatically calculated size, often something like `tweak=0.8` or `tweak=1.2`.

You may also want to look at `fallen.leaves` (put the leaves at the bottom), `uniform` (vertically space the nodes uniformly or proportionally to the fit), `shadow` (add shadows to the node boxes), and `box.palette` (change the colors of the boxes).
A model with a **binary** response.

```r
binary.model <- rpart(survived~., data=ptitanic, cp=.02)
rpart.plot(binary.model)
```

Each node shows
- the predicted class (died or survived),
- the predicted probability of survival,
- the percentage of observations in the node.

A model with a **continuous** response (an anova model).

```r
anova.model <- rpart(Mileage~., data=cu.summary)
rpart.plot(anova.model)
```

Each node shows
- the predicted value,
- the percentage of observations in the node.

A model with a **multi-class** response.

```r
multi.class.model <- rpart(Reliability~., data=cu.summary)
rpart.plot(multi.class.model)
```

Each node shows
- the predicted class (Much worse, worse, ..., Much better),
- the predicted probability of each class,
- the percentage of observations in the node.

In this example, the class better is never predicted by the model and doesn’t appear in the legend. The position of the legend can be adjusted with the `legend.x` and `legend.y` arguments.

**Figure 1:** `rpart.plot` with default arguments and different kinds of model
When dealing with the many arguments of \texttt{prp}, it helps to remember that the display has four constituents: the \textit{node labels}, the \textit{split labels}, the \textit{branch lines}, and the optional \textit{node numbers}. Each of these constituents has a complete set of \texttt{col} etc. arguments. Thus we have, for example, \texttt{col} (the color of the node label text), \texttt{split.col} (the split text), \texttt{branch.col} (the branch lines), and \texttt{nn.col} (the optional node numbers).

Standard graphics parameters such as \texttt{col} can be passed in as \ldots arguments. So where the help page refers to the \texttt{col} argument, what is meant is the \texttt{col} argument passed in as a \ldots argument, and if it is not passed in, the value of \texttt{par("col")}. Such parameters typically affect only the node labels, not the split labels or other constituents of the display.

Figure 2: The type argument.
Figure 3: The extra argument with an anova model. Percentages are included by adding 100 to extra.

Figure 4: The extra argument with a class model. This figure also illustrates under=TRUE which puts the extra data under the box.
4 FAQ

4.1 The text is too small. Can I make it bigger?

Set fallen.leaves to FALSE.

Use the tweak argument to make the text larger, e.g. tweak=1.2. This may cause overlapping labels. However, there is a little elbow room because of the whitespace between the labels.

Alternatively, we can reduce the whitespace around the text, allowing prp to (automatically) use a larger type size. Do this by reducing the gap between boxes and the box space around the text (try gap=0 and/or space=0).

Text size will often be too small with uniform=FALSE, arguably a bug in prp.

4.2 The graph is too cluttered. Can I reduce the clutter?

Use the tweak argument to make the text smaller, e.g. tweak=.8.

Or use an explicit value for cex, experimenting until the displayed graph looks right.

Consider using compress=FALSE and ycompress=FALSE, so prp does not shift nodes around to make space. Figure 26 on page 29 illustrates the effect of compress and ycompress.

4.3 I always use the same arguments to prp. Can I reduce the amount of typing?

There is a standard R recipe for this kind of thing. Create a wrapper function with the defaults we want:

```r
p <- function(x, type=4, extra=100, under=TRUE, branch.lty=3, ...) {
  prp(x=x, type=type, extra=extra, under=under, branch.lty=branch.lty, ...)
}
```

Calling `p(tree)` will draw the tree using our defaults, which can be overridden when necessary. We can pass any additional arguments to prp via our function’s ... argument.

The next step is to put the above code into your .Rprofile file so the function is always available. Locating that file can be the hardest part of the exercise. Under Windows 7, we can use

```
C:\Users\username\Documents\.Rprofile
```

Enter ?.Rprofile at the R prompt for details.
4.4 How do I reproduce the graph on the cover page of this document?

The code for most of the graphs in this document is in the file `inst/slowtests/user-manual-figs.R` included in the source release of `rpart.plot`.

To answer your specific question, use the following code to reproduce the graph on the cover page:

```r
pdf("front.pdf")
library(earth) # for the ozone1 data
data(ozone1)
a <- rpart(O3~., data=ozone1, cp=.024)
prp(a, main="An Example",
    type=4, fallen=T, branch=.3, round=0, leaf.round=9,
    clip.right.labs=F, under.cex=1,
    box.palette="GnYlRd",
    prefix="ozone\n", branch.col="gray", branch.lwd=2,
    extra=101, under=T, lt=" < ", ge=" >= ", cex.main=1.5)
dev.off() # close pdf file
```

4.5 How do I emulate `fancyRpartPlot` in the `rattle` package?


Invoking `rpart.plot` with the default arguments gives results similar to `fancyRpartPlot`, but with slightly fewer display elements (Figure 5 left side):

```r
rpart.plot(binary.model) # left graph
```

For this binary-response model we can get a graph that is very similar to `fancyRpartPlot` using the following arguments (Figure 5 middle):

```r
rpart.plot(binary.model, # middle graph
           extra=104, box.palette="GnBu",
           branch.lty=3, shadow.col="gray", nn=TRUE)
```

Rattle 2016−Jun−06 17:28:44 milbo

Figure 5: Using `rpart.plot` to get the same graphs as `fancyRpartPlot`. 
4.6 Please explain the warning Unrecognized rpart object

This warning is issued if your rpart object is not one of those recognized by prp, but prp is nevertheless attempting to plot the tree. You typically see this warning if your tree was created by a package that is not rpart although based on rpart.

Details: The prp node-labeling code recognizes the following values for the method field of a tree: "anova", "class", "poisson", "exp", and "mrt" (Appendix 11). If method is not one of those values, prp looks at the object’s frame to deduce if it can treat the object as it were an "anova" or "class" rpart model. If so, prp’s extra argument will work as described on prp’s help page. If not, prp calls the “text” function for the object (see the rpart documentation).

Note to package writers: To allow prp to be easily extended to recognize your trees, (i) your object’s class should be c("yourclass", "rpart") not plain "rpart", or (ii) your object’s method should be not be one of the standard strings mentioned above and not "user".

4.7 Citing the package

If you use this package in a published document, please do the right thing and cite it [4]:


@Manual{rpart.plotpackage,
  author = {Stephen Milborrow},
  title = {{rpart.plot: Plot rpart Models. An Enhanced Version of plot.rpart}},
  year = {2016},
  note = {R package},
  url = {http://CRAN.R-project.org/package=rpart.plot} }

5 Compatibility with plot.rpart and text.rpart

Here’s how to get prp to behave like plot.rpart:

- Instead of all=TRUE, use type=1 (type supersedes all and fancy, and provides more options).
- Instead of fancy=TRUE, use type=4.
- Instead of use.n=TRUE, use extra=1 (extra supersedes use.n and provides more options).
- Instead of pretty=0, use faclen=0 (faclen supersedes pretty).
- Instead of fwidth and fheight, use round and leaf.round to change the roundness of the node boxes, and space and yspace to change the box space around the label. But those arguments are not really equivalent. For square leaf-boxes use leaf.round=0.
- Instead of margin, use Margin (the name was changed to prevent partial matching with mar).
- Use border.col=0 to not draw boxes around the nodes.
- The post.rpart function may be approximated with:
  ```
  postscript(file="tree.ps", horizontal=TRUE)
  prp(tree, type=4, extra=1, clip.right.labs=FALSE, leaf.round=0)
  dev.off()
  ```

- plot.rpart’s default value for uniform is FALSE; prp’s is TRUE (because with uniform=FALSE and extra>0 the plot often requires too small a text size).
- plot.rpart’s default value for branch is 1; prp’s is 0.2 (because after applying compress and ycompress that arguably looks better).
- xpd=TRUE is often necessary with plot.rpart but is unneeded with prp. (See par’s help page for information on xpd.)

Ideally prp’s arguments should be totally compatible with plot.rpart. I hope you will agree that the above discrepancies are in some sense necessary, given the approach taken by prp.

5.1 An example: reproducing the plot produced by example(rpart)

The following code draws the first graph from example(rpart) and then draws a graph in the same style with prp:

```r
fit <- rpart(Kyphosis ~ Age + Number + Start, data=kyphosis) # from example(rpart)
par(mfrow=c(1,2), xpd=NA) # side by side comparison
plot(fit)
text(fit, use.n=TRUE)
prp(fit, extra=1, uniform=F, branch=1, yesno=F, border.col=0, xsep="/")
```
6 Customizing the node labels

In this section we look at ways of customizing the data displayed at each node.

To start off, consider using the extra argument to display additional information. See Figures 3 and 4 and the prp help page for details.

To simply display a constant string at each leaf use the prefix argument (Figure 6):

```r
data(ptitanic)
tree <- rpart(survived~., data=ptitanic, cp=.02)
prp(tree, extra=7, prefix="fraction\n")
```

We will use this model as a running example. In the data the response survived is a factor and thus by default rpart builds a class tree. The cp argument is used to keep the tree small for simplicity, and extra=7 is used to display the fitted probability of survival but not the fitted class.

An aside: By default rpart will treat a logical response as an integer and build an anova model, which is usually inappropriate for a binary response. So if your response is logical, first convert it to a factor so rpart builds a class model:

```r
my.data$response <- factor(my.data$response, labels=c("No", "Yes"))
```

Or explicitly use method="class" when invoking rpart, although that may be easy to forget.

The prefix argument can be a vector, allowing us to display node-specific text in much the same way that node-specific colors are displayed in Section 7.

If we need something more flexible we can define a labeling function to generate the node text. The usual rpart way of doing that is to associate a function with the rpart object (functions$text). However, prp does not call that function for the standard rpart methods. (This change was necessary for the extra argument.) So here we look at a different approach which is in fact often easier. We pass our labeling function to prp using the node.fun argument. The example below displays the deviance at each node (Figure 7):

```r
node.fun1 <- function(x, labs, digits, varlen)
{  
    paste("dev", x$frame$dev)
}
prp(tree, node.fun=node.fun1)
```
or, more concisely:

```
prp(tree, node.fun=function(x, labs, digits, varlen) paste("dev", x$frame$dev))
```

The labeling function should return a vector of label strings, with labels corresponding to rows in `x$frame`. The function must have all the arguments shown in the examples, even if it does not use them. Apart
from labs, these arguments are copies of those passed to prp. The labs argument is a vector of the labels generated by prp in the usual manner. This argument is useful if we want to include those labels but add text of our own. As an example, we modify the function above to include the text prp usually prints at the node (Figure 8):

```r
node.fun2 <- function(x, labs, digits, varlen)
{
    paste(labs, "\ndev", x$frame$dev)
}
prp(tree, extra=6, node.fun=node.fun2)
```

Text after a double newline in the labels is drawn below the box. So to put the deviances below the box, change \n to \n\n (Figure 9):

```r
node.fun3 <- function(x, labs, digits, varlen)
{
    paste(labs, "\n\ndev", x$frame$dev)
}
prp(tree, extra=6, node.fun=node.fun3)
```

We used a class model in the above examples, but the same approach can of course be used with other rpart methods.
6.1 Customizing the split labels

In a similar manner, we can also generate custom split labels using prp’s split.fun argument.

Sometimes the standard split labels are very wide, especially when a variable has multiple factor levels. Figure 10 is an example that wraps long split labels over multiple lines. The right plot was produced by the following code. The maximum length of each line is 25 characters. Change the 25 to suit your purposes.

```r
library(rpart)
library(sprplot)

# Example data
library(MASS)
data(iris)
iris <- within(iris, {Species <- ifelse(Species == "setosa", "Cmp", Species)
                      
                      Price <- log10(Price)})

# Training the model
tree <- rpart(Price/1000 ~ Mileage + Type + Country, data=iris)

# Custom split function
split.fun <- function(x, labs, digits, varlen, faclen)
{
  # replace commas with spaces (needed for strwrap)
  labs <- gsub(",", " ", labs)
  for(i in 1:length(labs)) {
    # split labs[i] into multiple lines
    labs[i] <- paste(strwrap(labs[i], width=25), collapse="\n")
  }
  labs
}

# Plotting
prp(tree, split.fun=split.fun)
```

![Figure 10: Default](image1.png) ![Figure 10: Long labels split with a split.fun](image2.png)
An alternative approach which is often easier, although not as flexible, is to use `split.prefix` and related arguments. The bottom left graph of Figure 1 is an example. To regenerate that graph, use `example(prp)`.

Note that we can generate labels of the form

"is pclass 2nd or 3rd?"

using

```
split.prefix="is ", split.suffix="?", eq=" ", facsep=" or ".
```

The graph can sometimes look better if we add a new line to the split label, so for example

```
Type = Cmp,Sml,Spr,Van
Country = Brz,Frn,Jpn,J/U,Kor,Mxc,USA
Type = Sml
Country = J/U,Mxc,USA
Country = Frn,Kor,USA
Type = Mdm
```

becomes narrower:

```
Type = Cmp,Sml,Spr,Van
Country = Brz,Frn,Jpn,J/U,Kor,Mxc,USA
Type = Sml
Country = J/U,Mxc,USA
Country = Frn,Kor,USA
Type = Mdm
```

Figure 11 demonstrates this technique. It was produced by the following code:

```
tree <- rpart(Price/1000 ~ Mileage + Type + Country, cu.summary)
split.fun <- function(x, labs, digits, varlen, faclen)
{
  gsub(" = ", ":
", labs)
}
prp(tree, extra=100, under=T, yesno=F, split.fun=split.fun)
```

Figure 11: Inserting a newline into the split labels with `split.fun`. 
7 Examples using the color and palette arguments

This first part of this chapter describes the `box.palette` argument (introduced in version 2.0.0 of the `rpart.plot` package, June 2016).

The second part describes the `col` and related arguments, which are a more flexible way of controlling colors, although more complicated to use.

7.1 The `box.palette` argument

The `box.palette` argument is generally the easiest way to add color to a plot. The color of each node box is chosen from the vector of colors in `box.palette`: small fitted values are displayed with colors at the start of the vector; large values with colors at the end.

7.1.1 `box.palette = "auto"`

With `box.palette="auto"`, a palette is chosen automatically (the choice is determined by the model’s response type). This is the default for the `rpart.plot` function (whereas the default for `prp` is `box.palette=0`, no color).

The following table shows how `extra` and `box.palette` are chosen automatically when specified as "auto". Figure 1 on page 3 gives some examples.

<table>
<thead>
<tr>
<th>model response</th>
<th>auto extra</th>
<th>auto <code>box.palette</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>continuous</td>
<td>100</td>
<td>&quot;Blues&quot;</td>
</tr>
<tr>
<td>two class (binary)</td>
<td>106</td>
<td>&quot;BuGn&quot;</td>
</tr>
<tr>
<td>three class</td>
<td>104</td>
<td>list(&quot;Reds&quot;, &quot;Grays&quot;, &quot;Greens&quot;)</td>
</tr>
<tr>
<td>four class</td>
<td>104</td>
<td>list(&quot;Reds&quot;, &quot;Oranges&quot;, &quot;Grays&quot;, &quot;Greens&quot;)</td>
</tr>
<tr>
<td>five class</td>
<td>104</td>
<td>list(&quot;Reds&quot;, &quot;Oranges&quot;, &quot;Grays&quot;, &quot;Blues&quot;, &quot;Greens&quot;)</td>
</tr>
<tr>
<td>etc.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The predefined palettes (like "Blues") begin with an upper case letter to disambiguate them from the standard colors like "blue" (since partial matching is supported). Predefined palette are converted internally into vectors of colors. The palettes use the ColorBrewer palettes [1] as a starting point, with some interpolation to bring the number of colors per palette to nine.

The “bifurcated” palettes (like "BuGn") are useful for binary responses.

Fitted values of NA are displayed in cross-hatched white boxes, ignoring `box.palette`.

7.1.2 `box.palette` with a binary response (class model)

Here are some examples for a model with a binary response (next page).
The code is (Figure 12):

```r
data(ptitanic)
tree <- rpart(survived~., data=ptitanic, cp=.02)
prp(tree, extra=6, box.palette="auto")  # left plot, "auto" becomes "BuGn"
prp(tree, extra=6, box.palette=c("pink", "palegreen3"))  # left mid plot
prp(tree, extra=6, box.palette="Blues")  # right mid plot, predefined palette
prp(tree, extra=6, box.palette="-Blues")  # right plot
```

Notice that in the last graph we reverse the order of the colors by prefixing the palette name with "-". The same technique can also be used with "auto", i.e. `box.palette="-auto"` is supported.

### 7.1.3 box.palette with a model with a continuous response (anova model)

Here are some examples for a model with a continuous response (Figure 13):

```r
data(cu.summary)
tree <- rpart(Mileage~., data=cu.summary)
prp(tree, box.palette="auto")  # left plot, "auto" becomes "Blues"
prp(tree, box.palette=c("pink", "palegreen3"))  # left mid plot
prp(tree, box.palette="RdYlGn")  # right mid plot, predefined palette
prp(tree, box.palette="-RdYlGn")  # right plot
```

Figure 13: The `box.palette` argument with a continuous response.
7.2 The col and related arguments

This section describes the col argument, which is a more flexible way of controlling colors than \texttt{box.palette}, although more complicated to use.

Arguments like \texttt{col} and \texttt{lty} are recycled and can be vectors, indexed on the row number in the tree’s \texttt{frame}. Thus the call \texttt{prp(tree, split.col = c("red", "blue"))} would allocate \texttt{red} to the node in first row of \texttt{frame}, blue to the second row, red to the third row, and so on. But that is not very useful, because splits and leaves appear in “random” order in \texttt{frame}, as can be seen in the example below. Note the node numbers along the left margin (we could plot those node numbers with \texttt{nn=TRUE} and their row indices with \texttt{ni=TRUE}):

```
> tree$frame
   var  n  wt  dev yval complexity ncompete nsurrogate yval2.1 yval2.2 yval2.3 yval2.4 yval2.5
1   sex 1309 1309  500 1.000  4.000  1.000  800.000  500.000  0.618  0.382
2   age  843  843  161 1.000  3.000  1.000  682.000  161.000  0.809  0.191
4 <leaf>  796  796  136 1.000  0.000  0.000  660.000  136.000  0.829  0.171
5  sibsp  47  47  22  2.000  3.000  2.000  22.000  25.000  0.468  0.532
10 <leaf>  20  20   1  1.000  0.000  0.000  19.000  1.000  0.950  0.050
11 <leaf>  27  27   3  2.000  0.000  0.000  3.000  24.000  0.111  0.889
  3 <leaf> 466 466 127  2.000  0.000  0.000 127.000 339.000  0.273  0.727
```

Here’s something more useful (Figure 14). We use the fitted value at a node (the \texttt{yval} field in \texttt{frame}) to determine the color of the node:

```
data(ptitanic)
tree <- rpart(survived~., data=ptitanic, cp=.02)
prp(tree, extra=6,
    box.col=c("pink", "palegreen3")[tree$frame$yval])
```

Figure 14: Using the fitted value and the \texttt{box.col} argument to determine the color of the boxes.
Figure 15 is a similar example for a tree with a continuous response (a.k.a a regression or anova tree). This example is based on code kindly supplied by Josh Browning:

```r
heat.tree <- function(tree, low.is.green=FALSE, ...) {
  y <- tree$frame$yval
  if(low.is.green)
    y <- -y
  max <- max(y)
  min <- min(y)
  cols <- rainbow(99, end=.36)[
    ifelse(y > y[1], (y-y[1]) * (99-50) / (max-y[1]) + 50,
      (y-min) * (50-1) / (y[1]-min) + 1)
  ]
  prp(tree, branch.col=cols, box.col=cols, ...)
}
data(ptitanic)
tree <- rpart(age ~ ., data=ptitanic)
heat.tree(tree, type=4, varlen=0, faclen=0, fallen.leaves=TRUE)
```

![Tree Diagram]

Figure 15: *Using the fitted value and the box.col argument to determine the color of regression nodes.*
The effect of the example on the previous page can be achieved more simply with the `box.palette` argument (Figure 16):

```r
data(ptitanic)
tree <- rpart(age ~ ., data=ptitanic)
rpart.plot(tree, type=4, extra=0, branch.1ty=3, box.palette="RdYlGn")
```

(The colors are paler than in the previous example. This is so the text in the node boxes can be read more easily.)

Figure 16: Using the fitted value and the `box.palette` argument to determine the color of regression nodes.
The following code creates a series of images – a movie – which shows how the tree is pruned on node complexity. Figure 17 is one of the plots produced by this code.

```r
complexities <- sort(unique(tree$frame$complexity)) # a vector of complexity values
for(complexity in complexities) {
  cols <- ifelse(tree$frame$complexity >= complexity, 1, "gray")
  dev.hold() # hold screen output to prevent flashing
  prp(tree, col=cols, branch.col=cols, split.col=cols)
  dev.flush()
  Sys.sleep(1) # pause for one second
}
```

Figure 17: Using the color arguments to indicate a node’s complexity. Nodes with a complexity greater than a certain value (0.021) are grayed out in this example.
The following code shows how a tree is constructed in depth-first fashion, node by node (Figure 18):

```r
tree1 <- rpart(survived~., data=ptitanic)
par(mfrow=c(4,3))
for(iframe in 1:nrow(tree1$frame)) {
  cols <- ifelse(1:nrow(tree1$frame) <= iframe, "black", "gray")
  prp(tree1, col=cols, branch.col=cols, split.col=cols)
}
```

Figure 18: Depth first tree construction.
The following code highlights a node and all its ancestors (Figure 19):

```r
# return the given node and all its ancestors (a vector of node numbers)
path.to.root <- function(node)
{
    if(node == 1) # root?
        node
    else # recurse, \%/ 2 gives the parent of node
        c(node, path.to.root(node \%/ 2))
}

node <- 11 # 11 is our chosen node, arbitrary for this example
nodes <- as.numeric(row.names(tree$frame))
cols <- ifelse(nodes %in% path.to.root(node), "sienna", "gray")
prp(tree, nn=TRUE,
    col=cols, branch.col=cols, split.col=cols, nn.col=cols)
```

Figure 19: Node 11 and its ancestors are highlighted.
7.3 Manipulating rpart models

Here are some code fragments demonstrating additional techniques for manipulating rpart models. It is worthwhile coming to grips with frame – look at `print(tree$frame)` and `print.default(tree)`. Sometimes we with work with node numbers and sometimes it is necessary to work with row numbers in frame:

```r
nodes <- as.numeric(row.names(tree$frame)) # node numbers in the order they appear in frame

node %% 2 # parent of node
c(node * 2, node * 2 + 1) # left and right child of node
inode <- match(node, nodes) # row index of node in frame

is.leaf <- tree$frame$var == "<leaf>" # logical vec, indexed on row in frame

nodes[is.leaf] # the leaf node numbers

is.left <- nodes %% 2 == 0 # logical vec, indexed on row in frame

ifelse(is.left, nodes+1, nodes-1) # siblings

get.children <- function(node) # node and all its children
  if(is.leaf[match(node, nodes)]) {
    node
  } else
    c(node,
      get.children(2 * node), # left child
      get.children(2 * node + 1)) # right child
```
8 Branch widths

It can be informative to have branch widths proportional to the number of observations. In the example on
the right side of Figure 20, the small number of observations at the bottom split is immediately obvious. We
can also estimate the relative number of males and females from the widths at the root split.

The right side of the figure was generated with:

```
prp(tree, branch.type=5, yesno=FALSE, faclen=0)
```

Note the `branch.type` argument. Other values of `branch.type` can be used to get widths proportional to
the node’s deviance, complexity, and so on. See the `prp` help page for details.

But be aware that the human eye is not good at estimating widths of branches at an angle. In Figure 21 the
left branch has the same width as the right branch, although one could be forgiven for thinking otherwise.
Width here should be measured horizontally, but the eye refuses to do that. The illusion is triggered by
the different slopes in this extreme example (whereas in a plotted tree the left and right branches at a split
usually have similar slopes and the illusion is irrelevant).

![Diagram of a tree with branch widths]

**Figure 20:** left The percentage of observations in a node. right That information represented by the width of the branches.

```
sex = male

age >= 9.5

- died
  - 61%
  - sibsp >= 2.5
    - died
      - 2%
    - survived
      - 2%

survived
36%
```

**Figure 21:** Misleading branch widths. The two branches have the same width, measured horizontally.
9 Trimming a tree with the mouse

Set \texttt{snip=TRUE} to display a tree and interactively trim it with the mouse.

If we click on a split it will be marked as deleted. If we click on an already-deleted split it will be undeleted (if its parent is not deleted). Information on the node is printed as we click.

When we have finished trimming, click on the \texttt{QUIT} button or right click, and \texttt{prp} will return the trimmed tree (in the \texttt{obj} field). Example (Figure 22):

```r
data(ptitanic)
tree <- rpart(survived~., data=ptitanic, cp=.012)
new.tree <- prp(tree, snip=TRUE)$obj  # interactively trim the tree
prp(new.tree) # display the new tree
```

You might like to prefix the above code with \texttt{par(mfrow=c(1,2))} to display the original and trimmed trees side by side.

Additionally, we can use the \texttt{snip.fun} argument to specify a function to be invoked after each mouse click. The following example prints the trimmed tree's performance after each click – using this technique we can manually select a desired performance-complexity trade-off.

```r
data(ptitanic)
tree <- rpart(survived ~ ., data=ptitanic)

my.snip.fun <- function(tree) { # tree is the trimmed tree
  # should really use indep test data here
  cat("fraction predicted correctly: ",
       sum(predict(tree, type="class") == ptitanic$survived) /
       length(ptitanic$survived),
       "\n")
}

prp(tree, snip=TRUE, snip.fun=my.snip.fun)
```

Figure 22: \textit{Interactively trimming a tree with snip=TRUE.}
10 Using plotmo in conjunction with prp

Another useful graphical technique is to plot the model’s response while changing the values of the predictors. Figure 23 illustrates this on the kyphosis data:

```r
library(rpart.plot)

tree <- rpart(Kyphosis~., data=kyphosis)

prp(tree, extra=7)  # left graph

library(plotmo)

plotmo(tree, type="prob", nresponse="present")  # middle graph
# type="prob" is passed to predict()

plotmo(tree, type="prob", nresponse="present",  # right graph
       type2="image", ngrid2=200,  # type2="image" for an image plot
       pt.col=ifelse(kyphosis$Kyphosis=="present", "red", "lightblue"))
```

The above code uses `plotmo` [5] to plot the regression surfaces. The figure actually shows just a subset of the plots produced by the calls to `plotmo`, with some adjustments for printing.

Note how each “cliff” in the middle graph corresponds to a split in the tree. (The slight slope of the cliffs is an artifact of the `persp` plot – the cliffs should be vertical.)

The `type="prob"` argument is passed internally in `plotmo` to `predict.rpart`, which returns a two column response, and the `nresponse="present"` argument selects the second column. In other words, we are plotting the predicted probability of kyphosis after surgery. We could instead plot the predicted class by using `type="class"`.

![Tree Diagram](image1.png)

![3D Plot](image2.png)

![Scatter Plot](image3.png)

Figure 23: The same tree represented in three different ways. The middle and right graphs show the predicted probability as a function of the predictors. The right graph is an aerial view of the middle graph.
Only two predictors were used in the `kyphosis` tree. More complex models with many predictors can be viewed in a piecemeal fashion by looking at the action of one or two predictors at a time. For example, Figure 24 shows a tree built from the `ozone` data:

```r
library(earth) # build a tree with the ozone1 data
data(ozone1)
tree <- rpart(O3~., data=ozone1)
prp(tree, main="ozone level") # left graph
plotmo(tree) # middle and right graphs
```

The model predicts the ozone level, or air pollution, as a function of several variables: Also shown are regression surfaces for the variables in the upper splits. (Once again, as in the previous figure, this figure actually shows just a subset of the plots produced by the call to `plotmo`.)

The `plotmo` graphs are created by varying two variables while holding all others at their median values. Thus the graphs show only a thin slice of the data, but are nonetheless helpful. They are most informative when the variables being plotted do not have strong interactions with the other variables. The `plotmo` vignette has further discussion and examples.

Figure 24: A tree built from the `ozone` data, and regression surfaces for the predictors at the upper splits.

Figure 25: Surfaces for other models using the `ozone` data. Compare to the right graph of Figure 24.
It is interesting to compare the `rpart` tree to other models (Figure 25). The linear model gives a flat surface. MARS [6] generates a surface by combining hinge functions (see http://en.wikipedia.org/wiki/Multivariate_adaptive_regression_splines). The random forest [3] smooths out the surface by averaging lots of trees.

There are a large number of possible variable pairs (from the 9 predictors in the ozone data we can form $9 \times 8/2 = 36$ pairs). The options to plotmo in the code below select just the pair in the graphs. See the plotmo help page for details. The code is:

```r
a <- lm(O3~., data=ozone1)
plotmo(a, degree1=NA, all2=TRUE, degree2=18)  # left graph, linear model

library(earth)  # earth is an implementation of MARS (MARS is a trademarked term)
a <- earth(O3~., data=ozone1, degree=2)
plotmo(a, degree1=NA, all2=TRUE, degree2=18)  # middle graph, MARS

library(randomForest)
a <- randomForest(O3~., data=ozone1)
plotmo(a, degree1=NA, all2=TRUE, degree2=5)  # right graph, random forest
```
11 The graph layout algorithm

For the curious, this section is an overview of the algorithm used by prp to lay out the graph. The current implementation is not perfect but suffices for most trees. The more-or-less standard approach for positioning labels, simulated annealing, is not used because an objective function cannot (easily) be calculated efficiently. A central issue is a chicken-and-egg problem: we need the cex to determine the best positions for the labels but we need the positions to determine the cex.

Initially, prp calculates the tentative positions of the nodes. If compress=TRUE (the default), it slides nodes horizontally into available space where possible. It uses the same code as plot.rpart to do all this, with a little extension for fallen.leaves. Figure 26 shows the same tree plotted with different settings of the compress and ycompress arguments (we will get to ycompress in a moment). In the middle plot see how age>=16 has been shifted left, for example.

If cex=NULL (meaning calculate a suitable cex automatically, the default), prp then calculates the cex needed to display the labels and their boxes with at least gap and ygap between the boxes. (Whether the boxes are invisible or not is immaterial to the graph layout algorithm.) This is accomplished with a binary search for the appropriate cex. A search is necessary because:

(a) It is virtually impossible to calculate the required scale analytically taking into account the many parameters such as adj, yshift, and space. For example, sometimes a smaller cex causes more overlapping as boxes shift around with the scale change.

(b) Font sizes are discrete, so the font size we get may not be the font size we asked for. This is especially a problem with a small cex where there is a large relative jump between the type size and the next smaller size.

Note that prp will only decrease the cex; it never increases the cex above 1 (but that can be changed with max.auto.cex).

If the initial cex is less than 0.7 (actually ycompress.cex), prp then tries to make additional space as follows (assuming ycompress=TRUE, the default). If type=0, 1, or 2, it shifts alternate nodes vertically, looking for

![Figure 26: The compress and ycompress arguments](image)
the shift in shift.amounts that allows the biggest type size. If type=3 or 4 it tries alternating the leaves if fallen.leaves=TRUE.

The shift is retained only if makes possible a type size gain of at least 10% (actually accept.cex). The shifted tree is not as “tidy” as the original tree, but the larger text is usually worth the untidiness (but not always). Compare the middle and right plots in Figure 26.

Finally, for small trees where there is too much white space, prp compacts the tree horizontally and/or vertically by changing xlim and ylim (Figure 27). This can be disabled with the xcompact and ycompact arguments.

Arguably the most serious limitation of the current implementation is its inability to display results on test data (on the tree derived from the training data).

**Acknowledgments**

I have leaned heavily on the code in plot.rpart and text.rpart. Those functions were written by Terry Therneau and Beth Atkinson, and were ported to R by Brian Ripley. The functions were descended from Linda Clark and Daryl Pregibon’s S-Plus tree package. But please note that the prp code was written independently and I must take responsibility for the excessive number of arguments, etc. I’d also like to thank Beth Atkinson for her feedback.
Appendix: mvpart mrt models

Note: In December 2014 the mvpart package was removed from CRAN.

The extra argument of prp has a special meaning for mvpart mrt models [2], as shown in the figure below. (Internally, mvpart sets the method field of the tree to "mrt", and prp recognizes that.) As always we can print percentages by adding 100 to extra. The type and other arguments work in the usual way.

Example:

```r
library(rpart.plot)
library(mvpart)
data(spider)
set.seed(10)
response <- data.matrix(spider[,1:3, drop=F])
tree <- mvpart(response~herbs+reft+moss+sand+twigs+water, data=spider, method="mrt", xv="min")
prp(tree, type=1, extra=111, under=TRUE)
```

Figure 28: The extra argument with an mrt model.
References


