

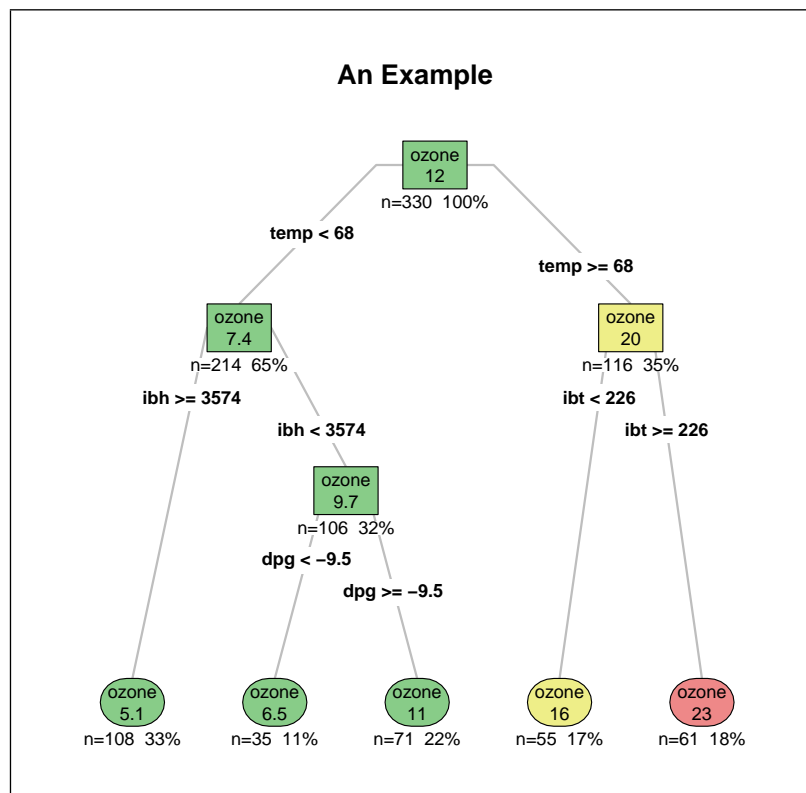
Plotting `rpart` trees with `prp`

Stephen Milborrow

February 25, 2011

Contents

1	Introduction	2
2	Overview	3
3	FAQ	6
4	Compatibility with <code>plot.rpart</code> and <code>text.rpart</code>	7
5	Customizing the node labels	8
6	Examples using the color arguments	11
7	Branch widths	15
8	Trimming a tree with the mouse	16
9	The graph layout algorithm	17
10	Acknowledgments	18



1 Introduction

The `prp` function plots `rpart` trees. 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. Figure 1 below shows some examples.

Section 2 of this document (the Overview) is the most important. The remaining sections may be skipped or read in any order.

I assume you have already looked at *An Introduction to Recursive Partitioning Using the RPART Routines* by Therneau and Atkinson:

Short version 2000 <http://mayoresearch.mayo.edu/mayo/research/biostat/upload/rpartmini.pdf>

Long version 1997 <http://mayoresearch.mayo.edu/mayo/research/biostat/upload/61.pdf>

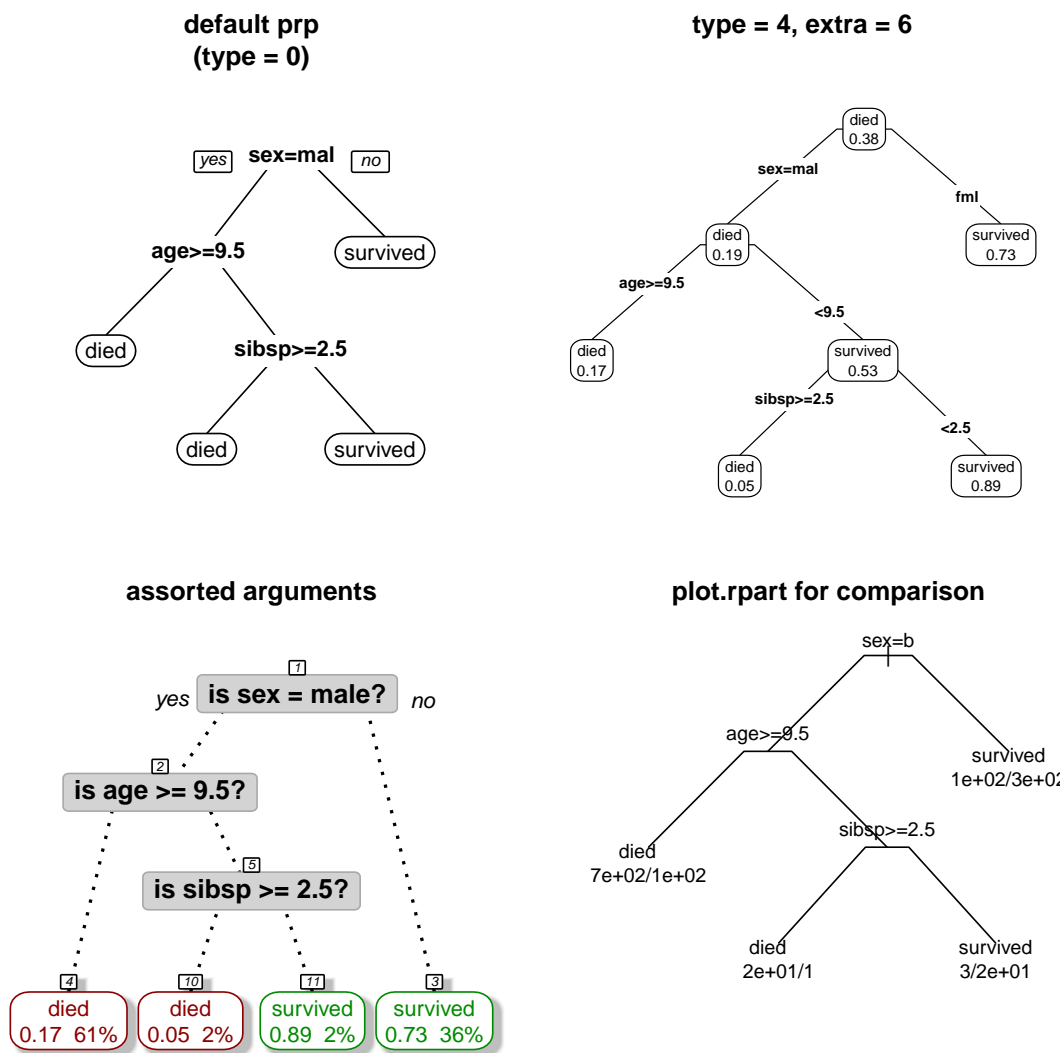


Figure 1: Some `prp` examples, with a `plot.rpart` graph for comparison. The data is the Titanic data with `survival` as the response (`sibsp` is the number of siblings or spouses aboard).

2 Overview

This section is an overview of the important arguments to `prp`. 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 below.

Use `extra` to add more details to the node labels, as shown in Figures 3 and 4 overleaf. Use `under=TRUE` to put those details under the boxes.

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.

Use `border.col` and `split.border.col` to add or remove boxes around the labels.

You may also want to look at `fallen.leaves` (put the leaves at the bottom), `branch` (control the angle of the branch lines), and `uniform` (vertically space the nodes uniformly or proportionally to the fit).

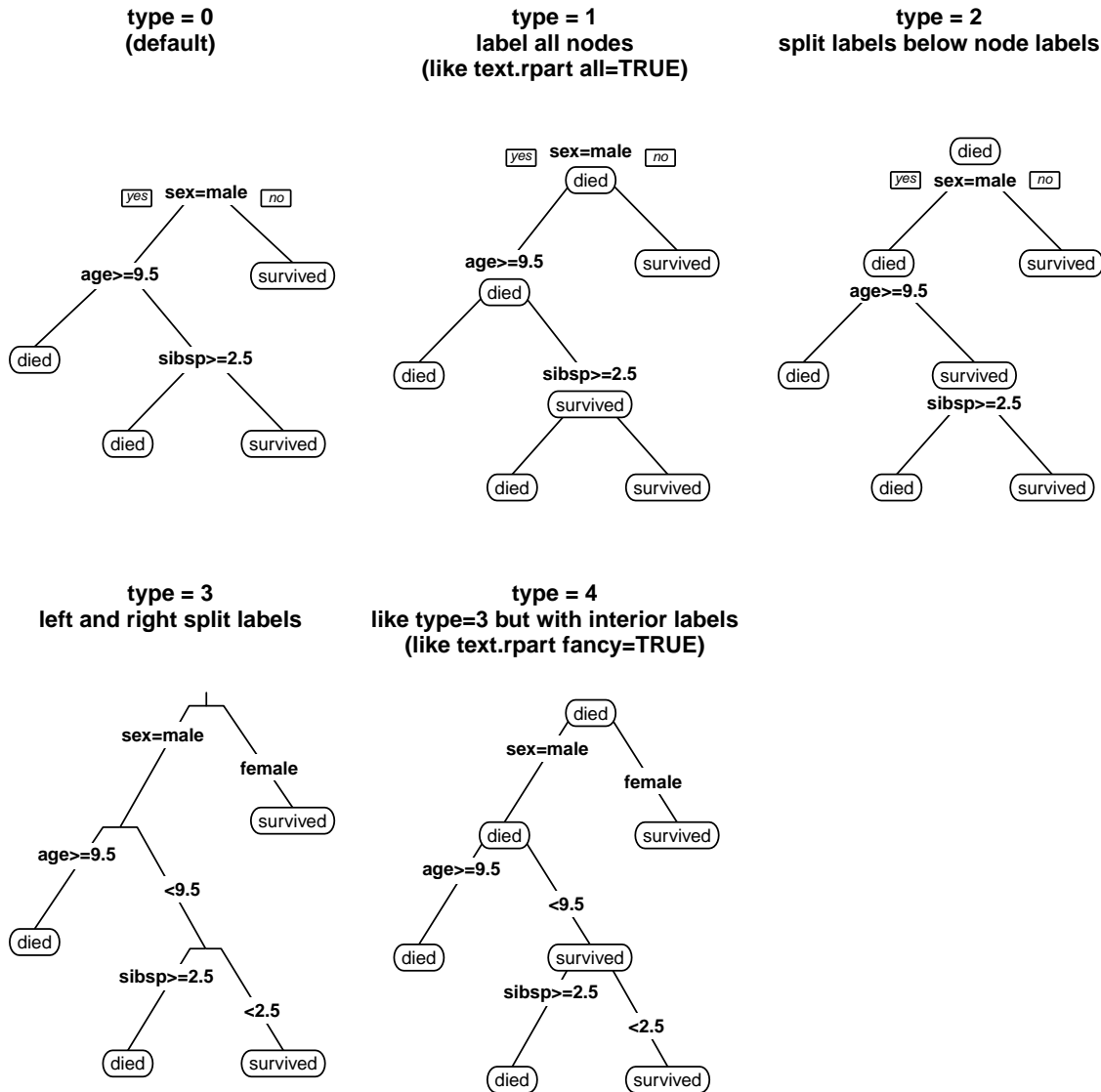


Figure 2: *The type argument.*

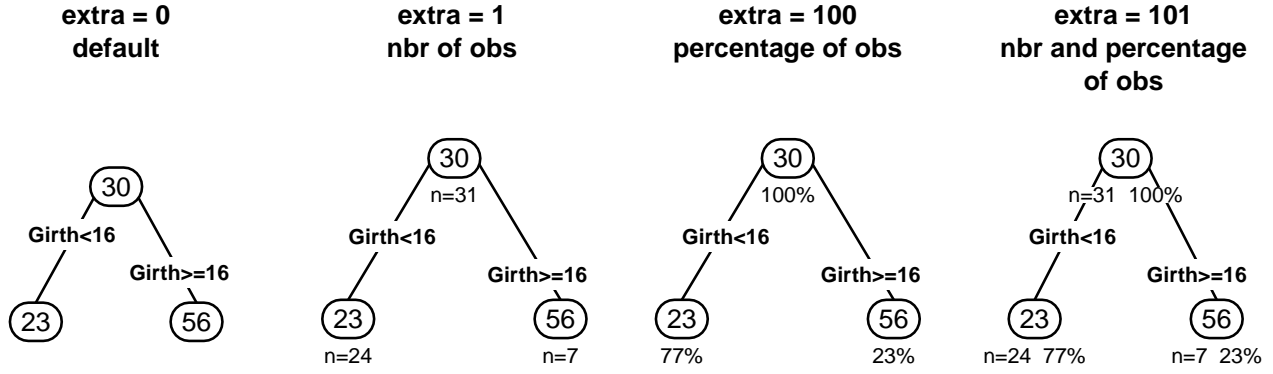


Figure 3: The `extra` argument with an anova model.

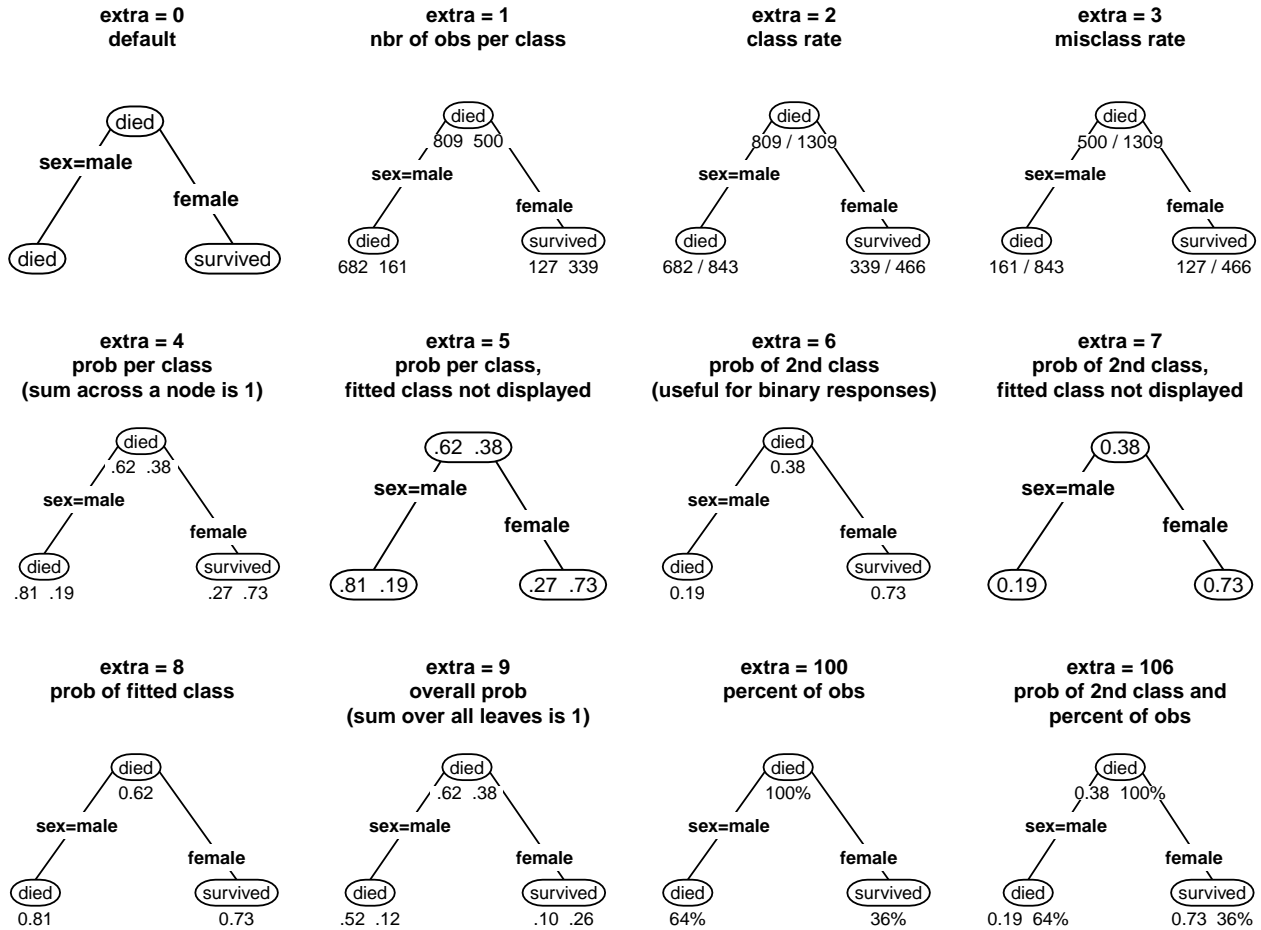


Figure 4: The `extra` argument with a class model. This figure also illustrates `under=TRUE` which puts the `extra` data under the box.

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

It helps to remember that the display has four constituents: the *node labels*, the *split labels*, the *branch lines*, and the optional *node numbers*. Each of these constituents has a complete set of `col` etc. arguments. Thus we have, for example, `col` (the color of the node label text), `split.col` (the split text), `branch.col` (the branch lines), and `nn.col` (the optional node numbers).

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

3 FAQ

3.1 The text is too small with the default arguments. Can I make it bigger?

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

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

Alternatively, you 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`).

3.2 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.

Also consider using `compress=FALSE` and `ycompress=FALSE`, so `prp` does not shift nodes around to make space. Figure 15 on page 17 illustrates the effect of `compress` and `ycompress`.

3.3 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 you want:

```
p <- function(x, type=4, extra=100, under=TRUE, leaf.round=0, ...)
{
  prp(x=x, type=type, extra=extra, under=under, leaf.round=leaf.round, ...)
}
```

Calling `p(tree)` will draw the tree using your defaults, which can be overridden when necessary. You can pass any additional arguments to `prp` via your 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 is the hardest part of the exercise. Under Windows 7, you can use

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

Enter `?Rprofile` at the R prompt for the gnarly details.

4 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).
- The `post` 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()
```
- 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 `edge` (the name was changed to prevent partial matching with `mar`).
- `plot.rparts`'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.rparts`'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 no longer needed with `prp`.

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.

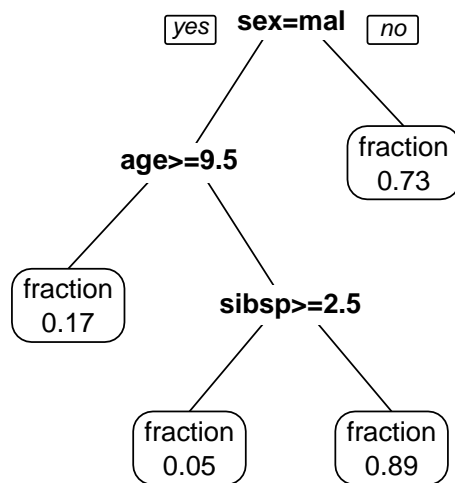


Figure 5: *Adding a constant prefix “fraction” to the node labels using `prefix="fraction"`.*

5 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 5):

```
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:

```
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 6.

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 unless `method="user"`. (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 6):

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

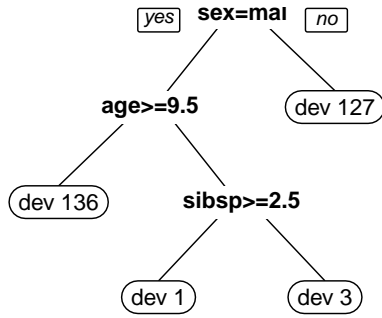



Figure 6: *Printing text at the nodes with `node.fun`.*

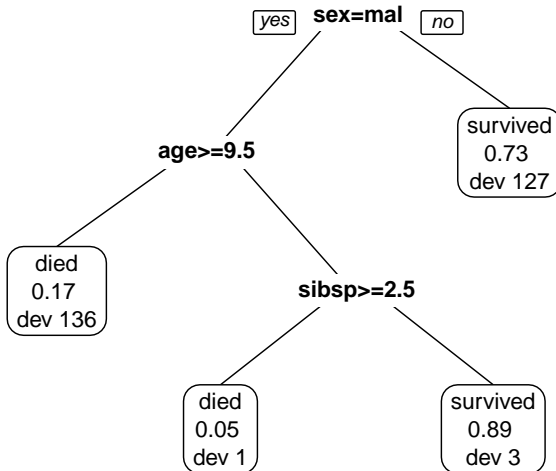


Figure 7: *Adding extra text to the node labels with `node.fun`.*

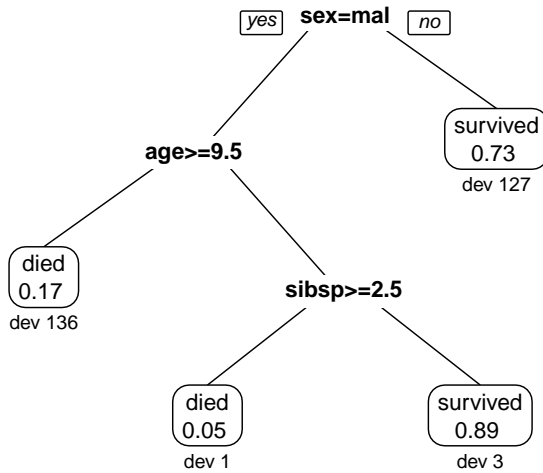


Figure 8: *Same as Figure 7, but with double newlines `\n\n` in the labels to move text below the boxes.*

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 7):

```
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 8):

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

In a similar manner, we can also generate custom *split* labels by setting the `split.fun` argument to a function. However, it is easier to use `split.prefix` and related arguments, when those suffice for the needs at hand. The bottom left plot of Figure 1 is an example. We can generate labels of the form "is pclass 2nd or 3rd?" using `split.prefix="is "`, `split.suffix="?", eq=" ", facsep=" or "`.

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

6 Examples using the color arguments

Arguments like `col` and `lty` are recycled and can be vectors, indexed on the row number in the tree's `frame`. Thus the call `prp(tree, split.col = c("red", "blue"))` would allocate `red` to the node in first row of `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 `frame`, as can be seen in the example below. Note the node numbers along the left margin (we could plot those node numbers with `nn=TRUE` and their row indices with `ni=TRUE`):

```
> tree$frame
      var    n   wt dev yval complexity ncomplete nsurrogate yval2.1 yval2.2 yval2.3 yval2.4 yval2.5
1    sex 1309 1309 500   1   0.424          4          1    1.000 809.000 500.000   0.618   0.382
2    age  843  843 161   1   0.021          3          1    1.000 682.000 161.000   0.809   0.191
4 <leaf>  796  796 136   1   0.000          0          0    1.000 660.000 136.000   0.829   0.171
5 sibsp   47   47  22   2   0.021          3          2    2.000  22.000  25.000   0.468   0.532
10 <leaf>   20   20   1   1   0.020          0          0    1.000  19.000   1.000   0.950   0.050
11 <leaf>   27   27   3   2   0.020          0          0    2.000   3.000  24.000   0.111   0.889
3 <leaf>  466  466 127   2   0.015          0          0    2.000 127.000 339.000   0.273   0.727
```

Here's something more useful (Figure 9). We use the fitted value at a node (the `yval` field in `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])
```

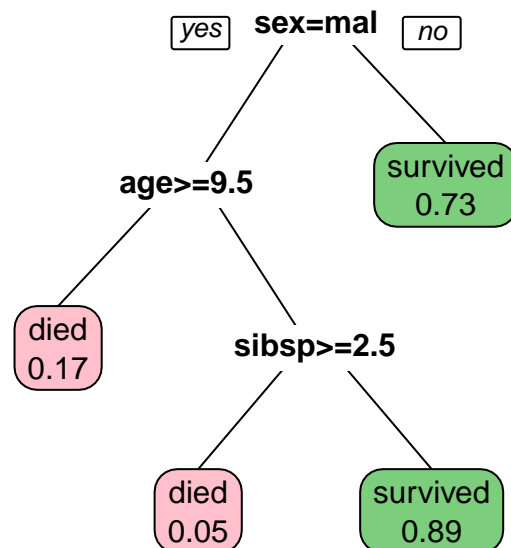


Figure 9: Using the fitted value and the `box.col` argument to determine the color of the boxes.

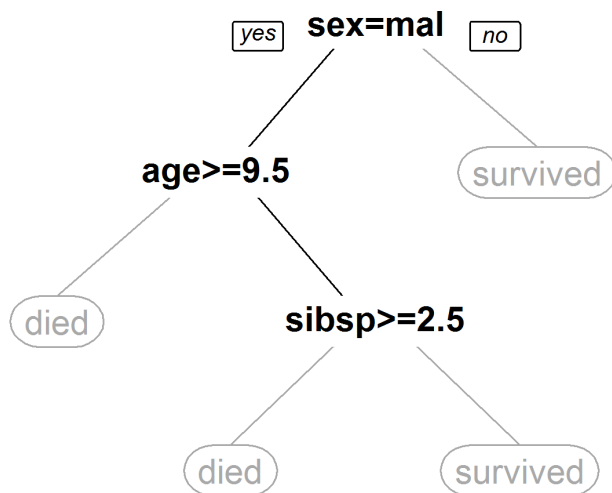


Figure 10: *Using the color arguments to indicate a nodes's complexity. Nodes with a complexity greater than a certain value (0.021) are grayed out.*

The following code creates a series of images — a movie — which shows how the tree is pruned on node complexity:

```

complexities <- sort(unique(tree$frame$complexity)) # a vector of complexity values
for(complexity in complexities) {
  cols <- ifelse(tree$frame$complexity >= complexity, 1, "darkgray")
  prp(tree, col=cols, branch.col=cols, split.col=cols)
  Sys.sleep(1) # wait one second
}

```

Figure 10 shows one of the plots produced by above code. (Screen flashing while the code is running is caused by `prp`'s dummy calls to `plot`, which are necessary to figure out the graph layout for the final plot. There seems to be no way to avoid the flashing when plotting to the screen.)

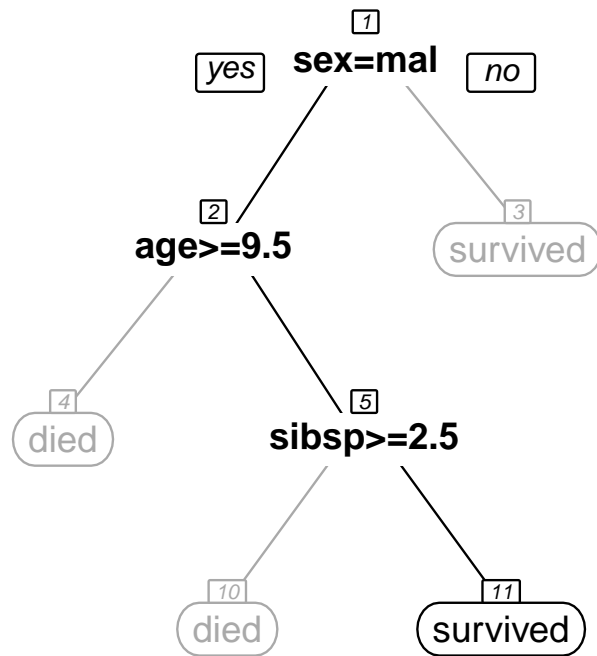


Figure 11: A node and all its ancestors highlighted.

The following code highlights a node and all its ancestors (Figure 11):

```

# 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), "red", "darkgray")
prp(tree, nn=TRUE, col=cols, branch.col=cols, split.col=cols, nn.col=cols)

```

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 will work with node numbers and sometimes it is necessary to work with row numbers in `frame`:

```
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
```

7 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 12, 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 13 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).

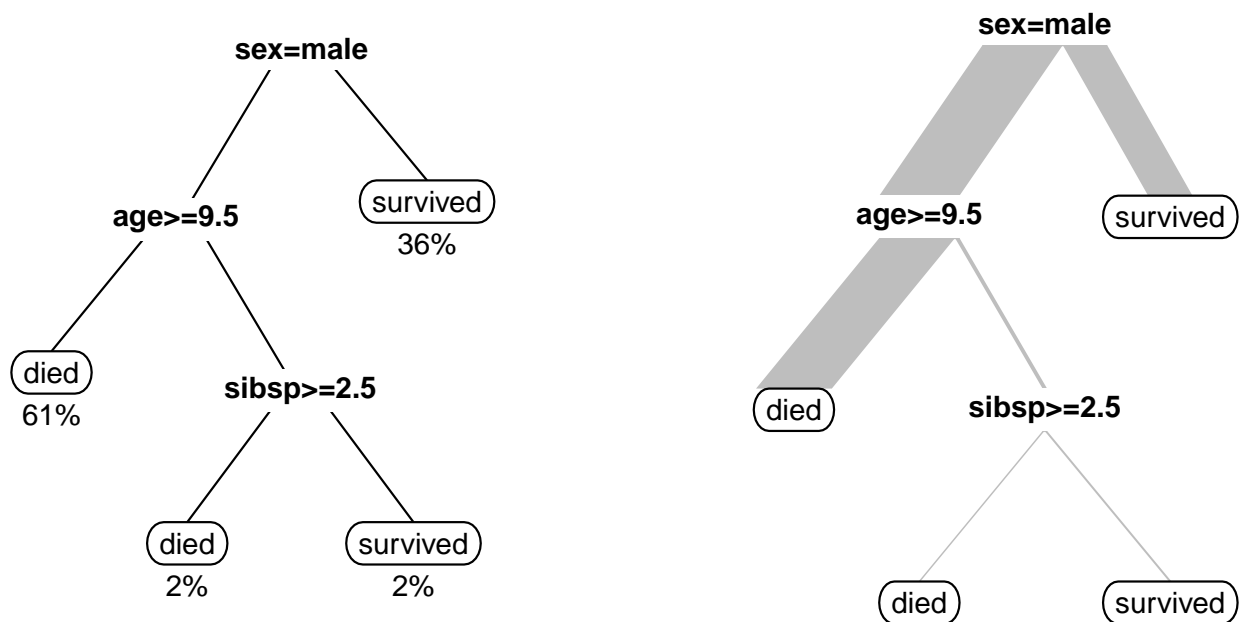


Figure 12: *left* The percentage of observations in a node.
right That information represented by the width of the branches.



Figure 13: *Misleading branch widths. The two “branches” have the same width.*

8 Trimming a tree with the mouse

Set `snip=TRUE` to display a tree and interactively trim it with the mouse.

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

When you have finished trimming, click on the QUIT button or right click, and `prp` will return the trimmed tree (in the `obj` field). Example (Figure 14):

```
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 `par(mfrow=c(1,2))` to display the original and trimmed trees side by side.

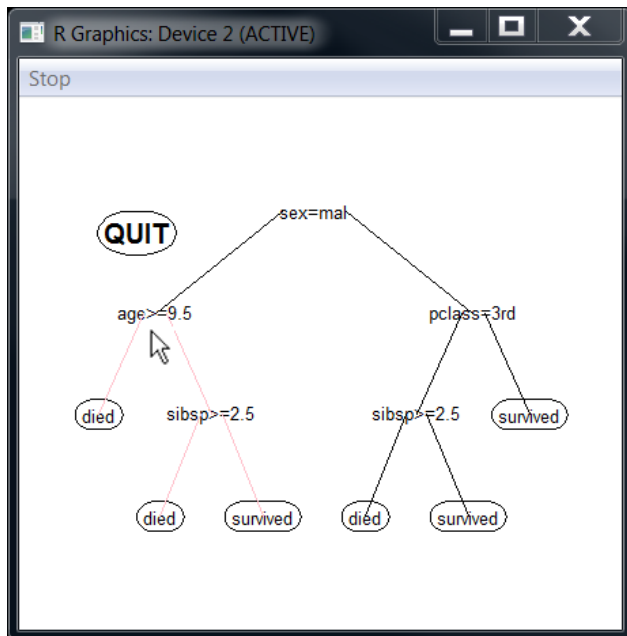


Figure 14: *Interactively trimming a tree with `snip=TRUE`.*

9 The graph layout algorithm

For the curious, this section is an overview of the algorithm used 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 15 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 ask 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

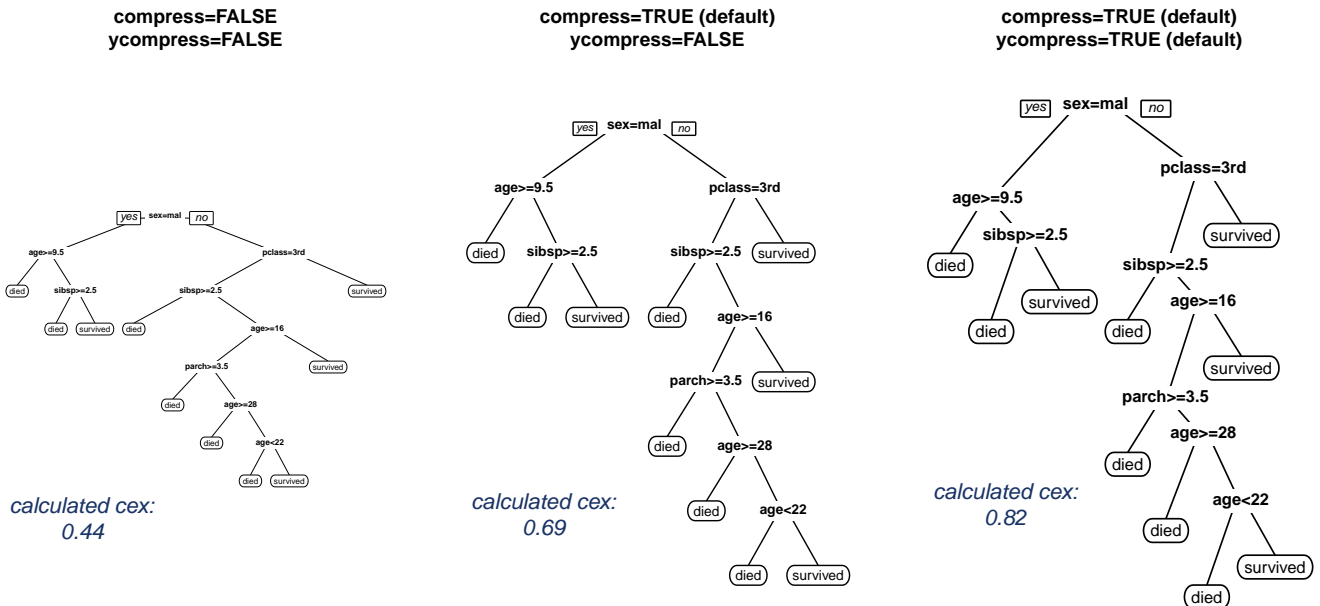


Figure 15: The `compress` and `ycompress` arguments

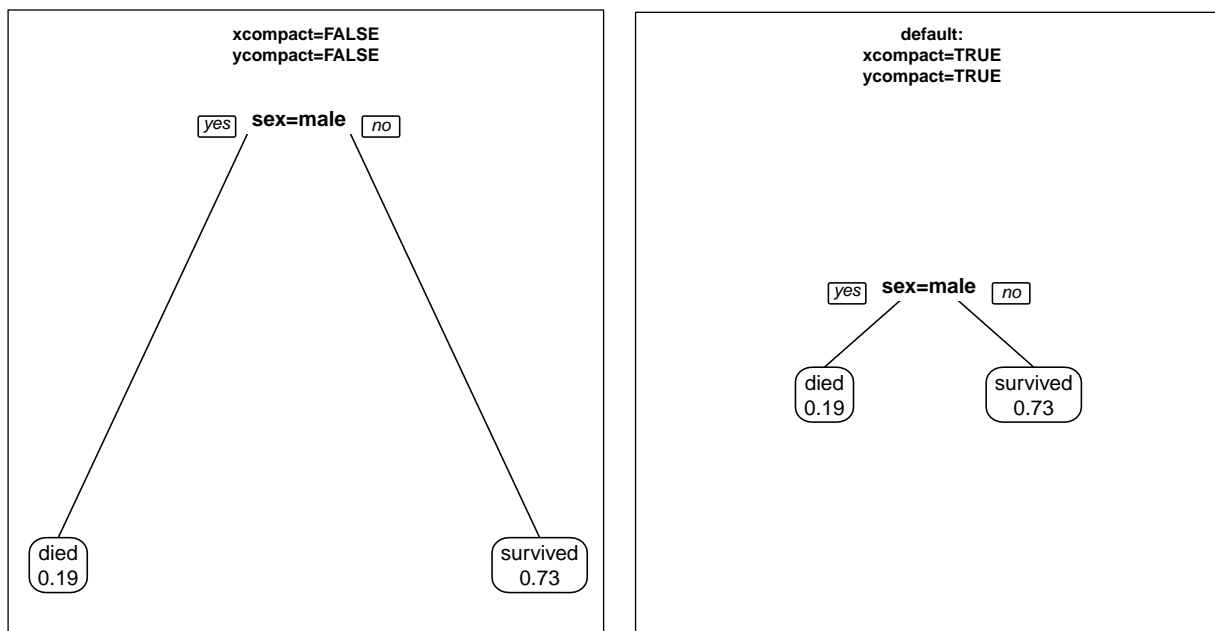


Figure 16: *Small trees are compacted by default, as shown on the right.*

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 15.

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 16). 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).

10 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 take responsibility for the excessive number of arguments, etc. I’d also like to thank Beth Atkinson for her feedback.