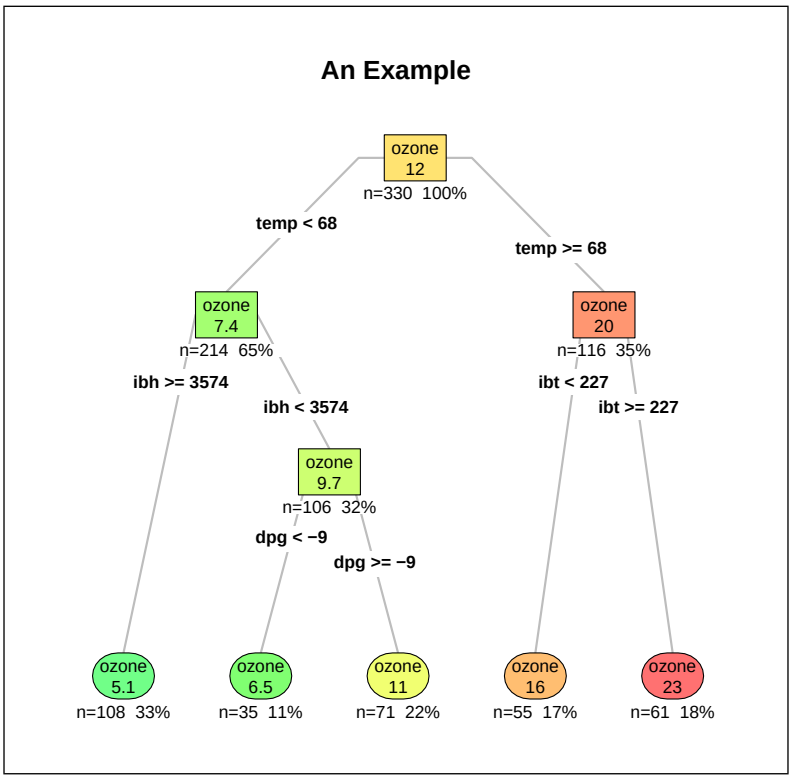
Plotting `rpart` trees with the `rpart.plot` package

Stephen Milborrow

August 7, 2026

Contents

1 Introduction **2**
2 Quick start **2**
3 Main arguments **2**
4 Printing rules with `rpart.rules` **6**
5 FAQ **9**
6 Customizing the node labels **13**
7 Examples using the color and palette arguments **18**
8 Branch widths **27**
9 Trimming a tree with the mouse **28**
10 Using `plotmo` in conjunction with `prp` **29**
11 Compatibility with `plot.rpart` and `text.rpart` **32**
12 The graph layout algorithm **33**



1 Introduction

The functions in the `rpart.plot` R package plot `rpart` trees [6,7]. The next page shows some examples (Figure 1).

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. Section 4 describes `rpart.rules`, which prints a tree as a set of rules. 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 [7]:
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

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 overall plotting style, as shown in Figure 2.

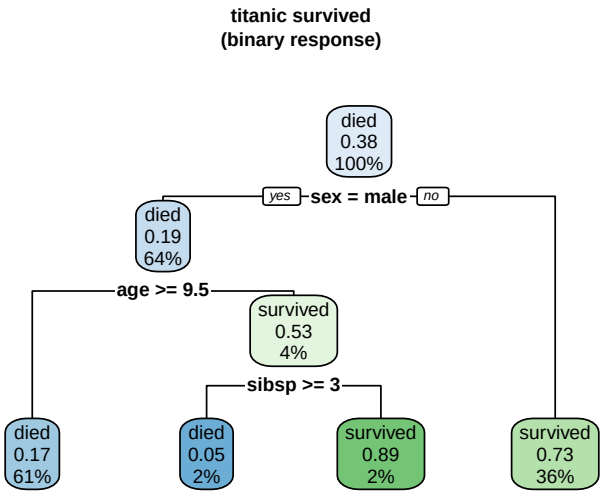
Use `extra` to add more details to the node labels, as shown in Figures 3 and 4. Use `under = TRUE` to put those details under the boxes (instead of in the boxes). With `extra = "auto"` (the default for `rpart.plot`), a suitable value for `extra` will be chosen automatically (based on the type of response 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 calculated automatically, unless `cex` is explicitly set. Use `tweak` to adjust the automatically calculated size, often something like `tweak = 0.8` or `tweak = 1.2`. Using `tweak` is often easier than specifying `cex`.

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:

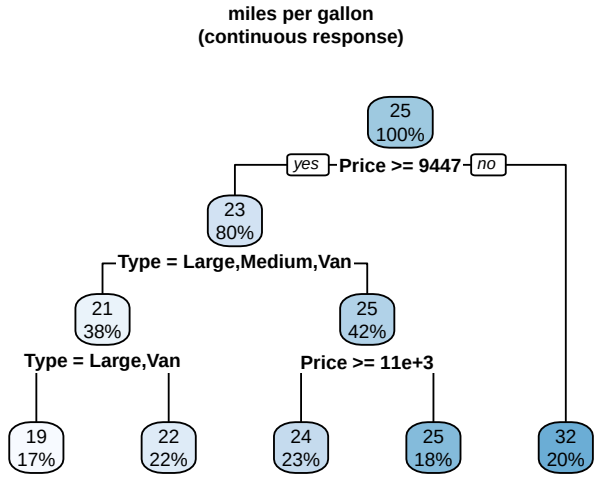
`box.palette = "auto"`	automatically choose a palette	(default for `rpart.plot`, Figure 1)
`box.palette = 0`	uncolored (white) boxes	(default for `prp`)
`box.palette = "Grays"`	a range of grays	(`"Grays"` is one of the built in palettes)
`box.palette = "gray"`	uniform gray boxes	



A model with a **binary** response.

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

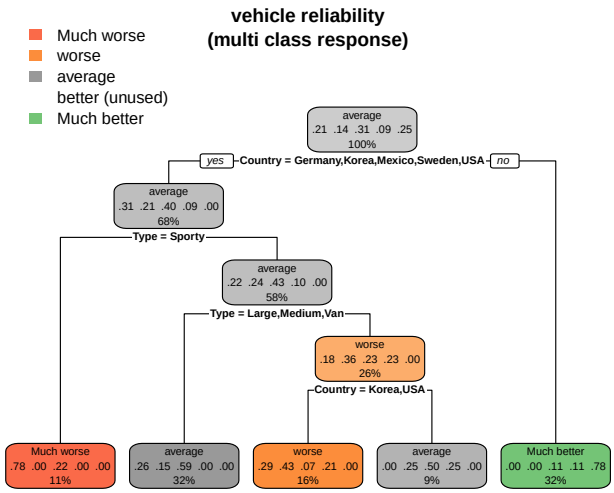
Each node shows
- the predicted class (died or survived),
- the predicted probability of survival,
- the percentage of observations in the node.



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

```
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.

```
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 thus is marked `unused` in the legend.
Use `trace = 1` to see the auto-positioned x,y coordinates of the legend. This position can be adjusted with the `legend.x` and `legend.y` arguments.

Figure 1: *`rpart.plot` with default arguments and different kinds of model*

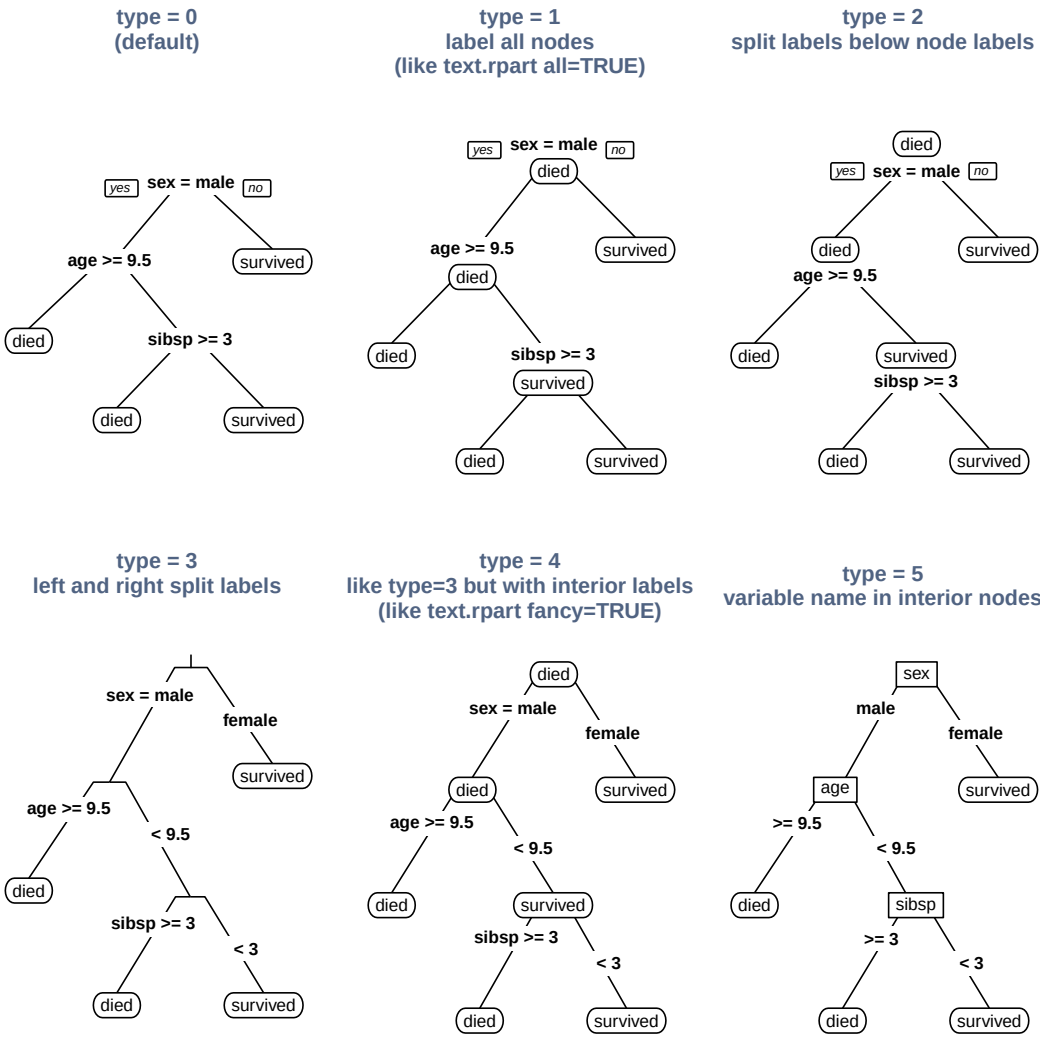


Figure 2: *The* `type` *argument.*

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), and `shadow` (add shadows to the node boxes). Section 4.1 illustrates the `roundint` and `clip.facs` arguments.

When dealing with the many arguments of `prp`, 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.

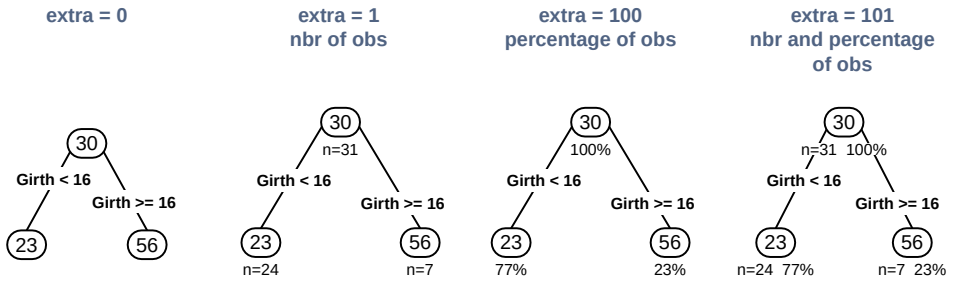


Figure 3: *The* `extra` *argument with an* `anova` *model. Percentages are included by adding* `100` *to* `extra`*.*

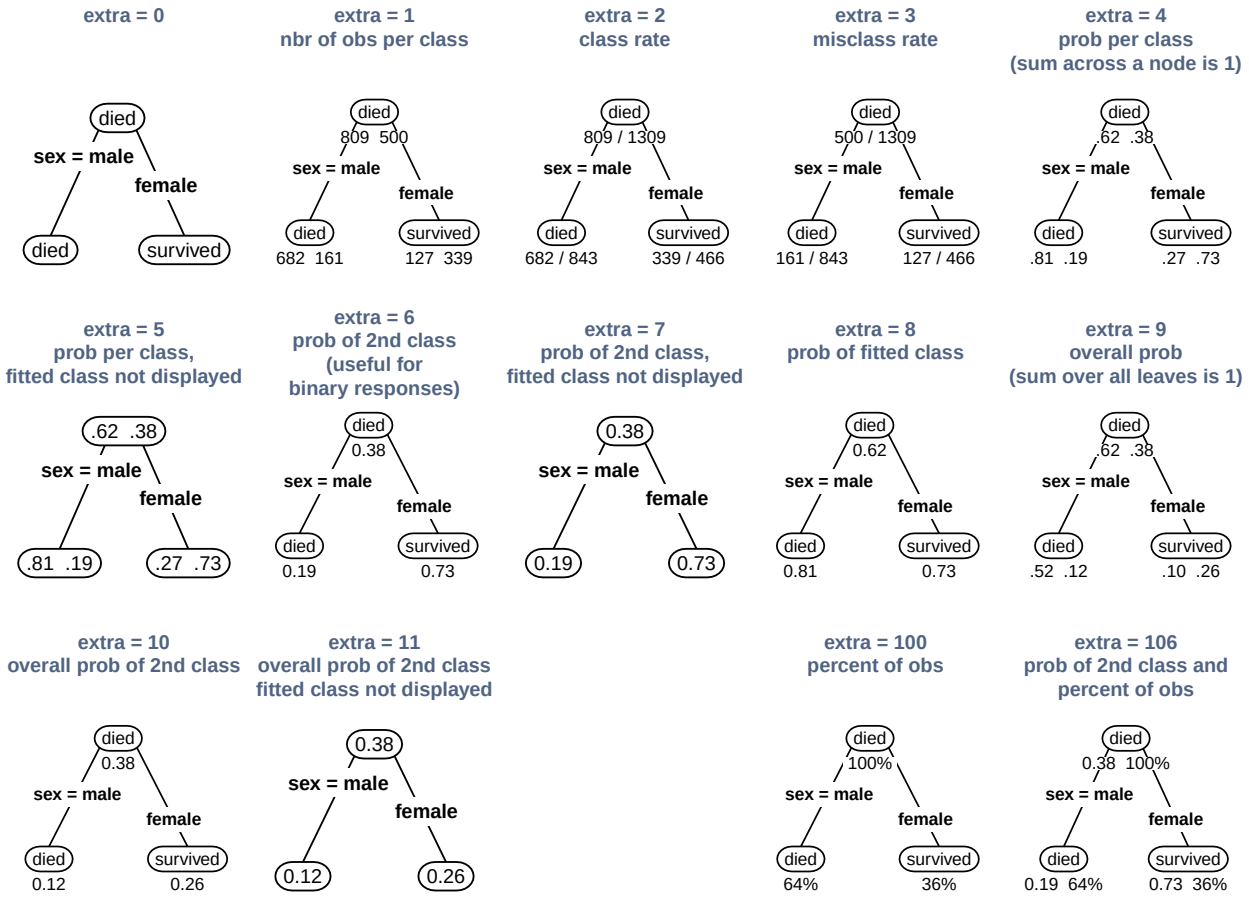


Figure 4: *The* `extra` *argument with a* `class` *model. This figure also illustrates* `under = TRUE` *which puts the extra data under the box.*

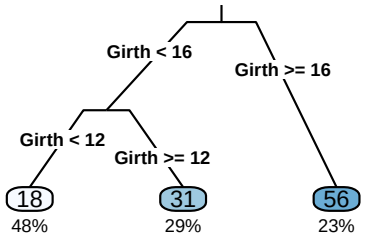
4 Printing rules with `rpart.rules`

An `rpart` tree can be printed as a set of rules using the function `rpart.rules`. The rules are sometimes clearer or more convenient than the plotted tree.

For example, we build a model to predict the volume of usable timber from cherry trees:

```
data(trees)
volume <- rpart(Volume ~ ., data = trees)
rpart.plot(volume, type = 3, clip.right.labs = FALSE, branch = .3, under = TRUE)
rpart.rules(volume)
```

The resulting tree and rules (shown in blue) are:



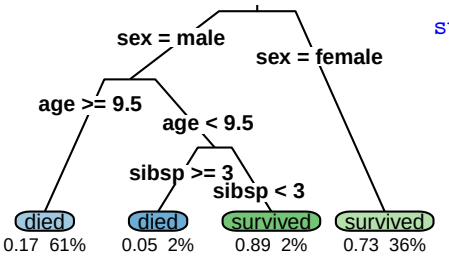

```
Volume
    18 when Girth <  12
    31 when Girth is 12 to 16
    56 when Girth >=       16
```

We can see that the `rpart` algorithm discards the `Height` variable in the `trees` data, and estimates the `Volume` by separating the `Girth` into three partitions. Notice how the two conditions along the left side of the tree (`Girth < 16` and `Girth < 12`) are collapsed into the single rule `Girth < 12`.

Another example is the probability of survival of Titanic passengers:

```
data(ptitanic)
survived <- rpart(survived ~ ., data = ptitanic, cp = .02)
rpart.plot(survived, type = 3, clip.right.labs = FALSE, branch = .3, under = TRUE)
rpart.rules(survived, cover = TRUE)
```

The tree and rules are:




```
survived                                                    cover
    0.05 when sex is   male & age <  9.5 & sibsp >= 3       2%
    0.17 when sex is   male & age >= 9.5                   61%
    0.73 when sex is female                                36%
    0.89 when sex is   male & age <  9.5 & sibsp <  3       2%
```

The left column of the rules gives the `survived` probability. The rules are sorted on this column. The rightmost column gives the percentage of observations in each rule (printed because we used the optional `cover = TRUE` argument).

Starting at the bottom, the last rule says that young boys without many siblings had a survival probability of `0.89`.[1] Presumably (and poignantly) boys with more siblings stayed with their families rather than board a lifeboat. The cover column on the right shows that only `2%` of the passengers fell into this rule.

The second rule from the bottom says simply that females had a `0.73` survival probability, and the entry in the cover column show that `36%` of the passengers were female.

By building trees with different combinations of variables and complexity `cp`, it's possible to see how younger females were more likely to survive, and so were people in higher classes. (Not shown here.)

[1] `sibsp` is the number of siblings and spouses aboard.

4.1 Arguments for rules

This section discusses a couple of arguments that are useful for rules. Actually these arguments can also be used for trees plotted by `rpart.plot` and `prp`, but are perhaps more clearly illustrated in the rules.

The `roundint` argument

With `roundint = TRUE` (default) the splits for a predictor are rounded to integer, if all values of that predictor in the training data are integers. This is helpful, because for integer predictors `rpart` usually creates splits ending in `.5`. Compare the following two sets of rules, focusing on the red text:

```
> rpart.rules(survived)   # default roundint = TRUE
survived
    0.05 when sex is   male & age <  9.5 & sibsp >= 3
    0.17 when sex is   male & age >= 9.5
    0.73 when sex is female
    0.89 when sex is   male & age <  9.5 & sibsp <  3

> rpart.rules(survived, roundint = FALSE)
survived
    0.05 when sex is   male & age <  9.5 & sibsp >= 2.5
    0.17 when sex is   male & age >= 9.5
    0.73 when sex is female
    0.89 when sex is   male & age <  9.5 & sibsp <  2.5
```

When `roundint = TRUE` (default), the splits on `sibsp` are printed as integers, because all values of `sibsp` in the input data are integers. The number of a passenger's siblings and parents is always an integer. Given this, the two sets of rules behave identically.

When `roundint = FALSE`, the splits on `sibsp` are printed as they are modeled in the tree (and printed by `print.rpart`).

In this example, the `roundint` setting doesn't affect `age`, because there are non-integer ages in the input data (some infants had ages less than a year).

Note that `roundint = TRUE` requires access to the data used to build the model, which isn't always available (in which case a warning will be issued and the rules will be printed as if `roundint = FALSE`).

The `clip.facs` argument

Use the `clip.facs` argument (by default `FALSE`) to reduce the amount of text by dropping predictor names for factors. Compare the following two sets of rules, focusing on the red text:

```
> rpart.rules(survived)   # default clip.facs = FALSE
survived
    0.05 when sex is   male & age <  9.5 & sibsp >= 3
    0.17 when sex is   male & age >= 9.5
    0.73 when sex is female
    0.89 when sex is   male & age <  9.5 & sibsp <  3

> rpart.rules(survived, clip.facs = TRUE)
survived
    0.05 when   male & age <  9.5 & sibsp >= 3
    0.17 when   male & age >= 9.5
    0.73 when female
    0.89 when   male & age <  9.5 & sibsp <  3
```

The extra argument

The default for the `extra` argument is the same as for `rpart.plot`, i.e., a sensible default is automatically chosen based on the type of model. For binomial models like the Titanic survival model, this means that the probability of the second class is shown (`extra = 6`, illustrated in Figure 4). To show the fitted class and probabilities for *both* classes use `extra = 4`:

```
> rpart.rules(survived, extra = 4, cover = TRUE)
  survived  die sur                                                  cover
      died [.95 .05] when sex is   male & age <  9.5 & sibsp >= 3     2%
      died [.83 .17] when sex is   male & age >= 9.5                 61%
  survived [.27 .73] when sex is female                              36%
  survived [.11 .89] when sex is   male & age <  9.5 & sibsp <  3     2%
```

Using `extra = 9`, we can show the overall probability at each rule (instead of the probability conditioned on the rule):

```
> rpart.rules(survived, extra = 9, cover = TRUE)
survived  die sur                                                  cover
    died [.01 .00] when sex is   male & age <  9.5 & sibsp >= 3      2%
    died [.50 .10] when sex is   male & age >= 9.5                  61%
survived [.00 .02] when sex is   male & age <  9.5 & sibsp <  3      2%
survived [.10 .26] when sex is female                               36%
```

Notice first that the rules are (somewhat) reordered because the probabilities used for sorting are different with the change to `extra`.

The sum of probabilities across all rules and classes is 1, up to rounding. In the last rule, we see that a fraction 0.26 of all who survived is covered by the rule.

In contrast, in the same rule on the top of the page we see that a fraction 0.73 of the passengers covered by the rule survived. In that set of rules, the sum of probabilities for each rule is 1.

The digits argument

The default `digits = 2` is quite low, to reduce noise. Change to a higher value if necessary. Specify the special value `digits = 0` to use `getOption("digits")`.

The `digits` argument for `rpart.rules` is handled a little differently from the same argument for `rpart.plot`. Numbers in rules are printed using R standard `data.frame` processing for `digits` (meaning that `digits` is adjusted internally for optimum formatting taking into account all the numbers in a column). Engineering notation isn't used in the rules, since lining up numbers vertically in mixed engineering and standard notation isn't really feasible.

The rule used to make a prediction

The function `rpart.predict` shows the rules used to make predictions. For example:

```
> rpart.predict(survived, newdata=ptitanic[1:3,], rules=TRUE)

     died survived
1 0.27253  0.72747 because sex is female
2 0.11111  0.88889 because sex is   male & age <  9.5 & sibsp <  3
3 0.27253  0.72747 because sex is female
```

5 FAQ

5.1 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`.

5.2 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 27 on page 33 illustrates the effect of `compress` and `ycompress`.

5.3 What arguments should I use for the clearest tree?

For audiences unfamiliar with trees, one approach is to use is to use `yesno = 2` to print `yes` and `no` at every split (Figure 5 second tree).

But perhaps the clearest trees are obtained with `type = 3` and `clip.right.labs = FALSE` (Figure 5 third tree). This explicitly labels both left and right hand branches of each split. In the example tree we also use `branch = .3` to draw the branch lines at an angle—I think this makes the tree structure a bit more clear. You could also try `under = TRUE` to put details under the boxes instead of in them.

If you need labels on the interior nodes, use `type = 4` instead of `type = 3`, but this often results in text that is very small (Figure 5 rightmost tree).

A problem with using `type 3` or `4` with larger trees is that the diplayed text can be too small. This is one reason these values aren't the defaults—the defaults try to strike a balance between clarity, redundancy,

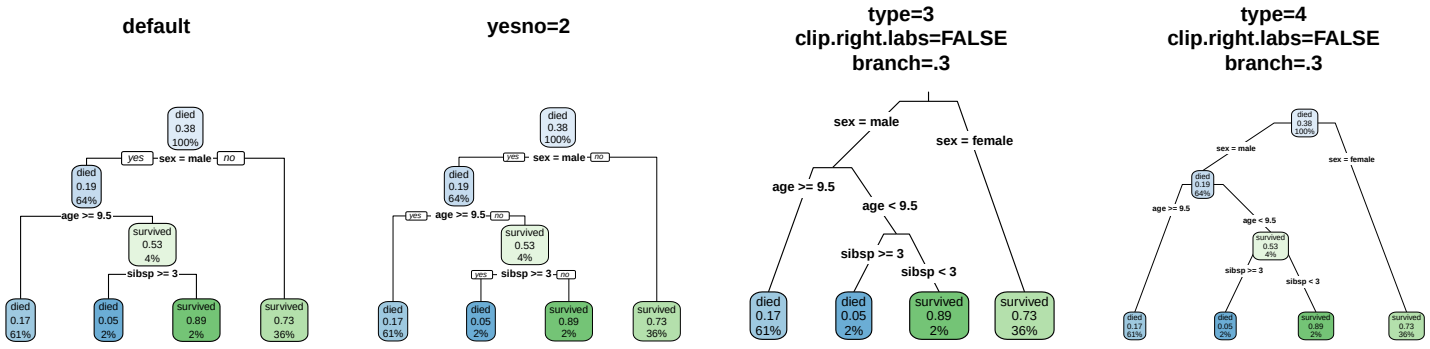


Figure 5: *Using the* `type` *and other arguments to make interpretation of the splits more obvious.*

clutter, and text size in the available screen space.

5.4 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/vingette.R` included in the source code of `rpart.plot`.

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

```
pdf("front.pdf")
library(earth) # for the ozone1 data
data(ozone1)
a <- rpart(O3 ~ ., data = ozone1, cp = .024)
prp(a,
    type = 4,                # left and right split labels (see Figure 2)
    clip.right.labs = FALSE, # full right split labels
    extra = 101,             # show nbr of obs and percentages (see Figure 3)
    under = TRUE,            # position extra info _under_ the boxes
    under.cex = 1,           # size of text under the boxes (default is .8)
    fallen.leaves = TRUE,    # put leaves at the bottom of plot
    box.palette = "GnYlRd",  # color of the boxes
    branch = .3,             # branch lines with narrow shoulders and down slopes
    round = 0,               # no rounding of node corners i.e. use rectangles
    leaf.round = 9,          # round leaf nodes (for leaves, this supersedes the round arg)
    prefix = "ozone\n",      # prepend this string to the node labels
    main = "An Example",     # main title
    cex.main = 1.5,          # use big text for main title
    branch.col = "gray",     # color of branch lines
    branch.lwd = 2)          # line width of branch lines
dev.off() # close pdf file
```

5.5 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:

```
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, use this:

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

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

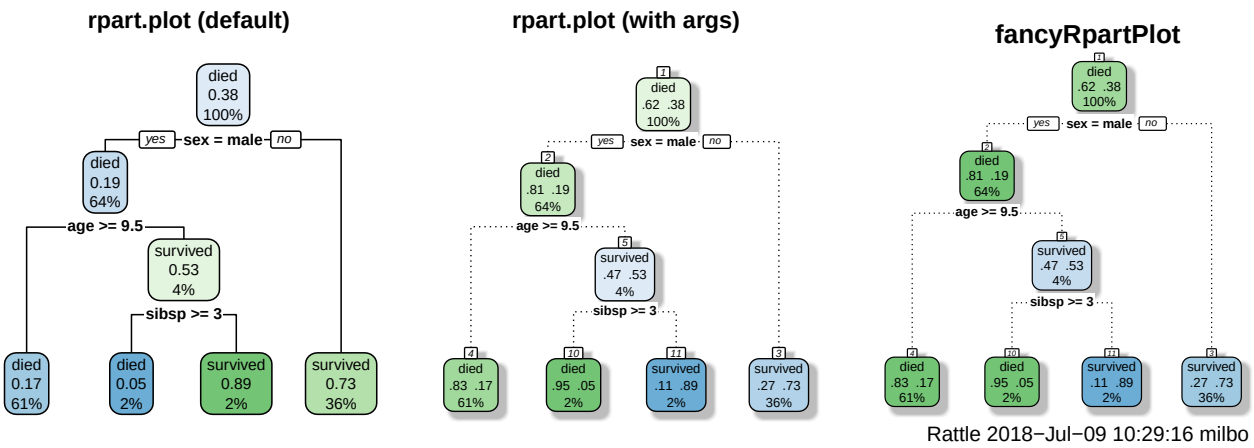


Figure 6: *Using* `rpart.plot` *to get the same graphs as* `fancyRpartPlot`*.*

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

The popular `fancyRpartPlot` [8] function calls `rpart.plot` internally.

Invoking `rpart.plot` with the default arguments gives results similar to `fancyRpartPlot`, but with slightly fewer display elements (Figure 6 left side):

```
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 6 middle):

```
rpart.plot(binary.model,                    # middle graph
           extra = 104,           # show fitted class, probs, percentages
           box.palette = "GnBu", # color scheme
           branch.lty = 3,        # dotted branch lines
           shadow.col = "gray",  # shadows under the node boxes
           nn = TRUE)             # display the node numbers
```

5.7 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`, or if your `rpart` model has user splits.

Details: The `prp` node-labeling code recognizes the following values for the `method` field of a tree: `"anova"`, `"class"`, `"poisson"`, `"exp"`, and `"mrt"`. 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"`.

5.8 Citing the package

Stephen Milborrow. *rpart.plot: Plot rpart Models. An Enhanced Version of plot.rpart.*, 2016. R Package.

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

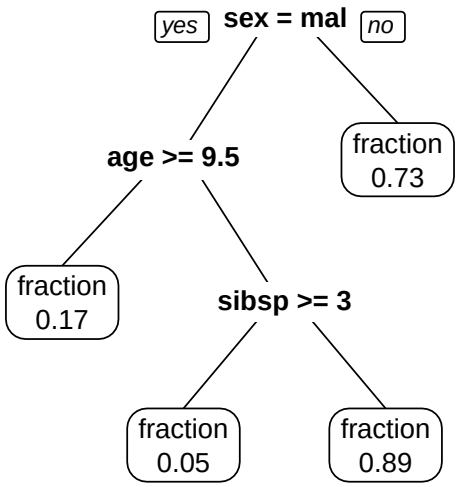


Figure 7: *Adding a constant prefix "`fraction`" to the node labels using `prefix = "fraction"`.*

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 7):

```
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 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 8):

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

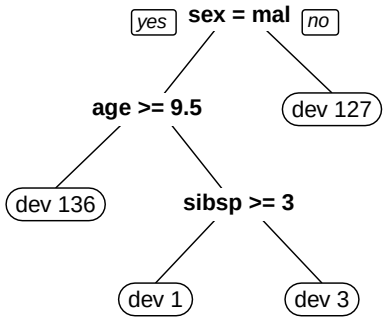


Figure 8: *Printing text at the nodes with* `node.fun`.

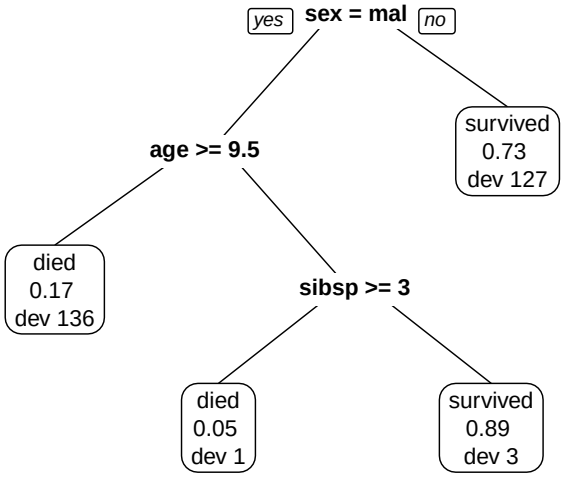


Figure 9: *Adding extra text to the node labels with* `node.fun`.

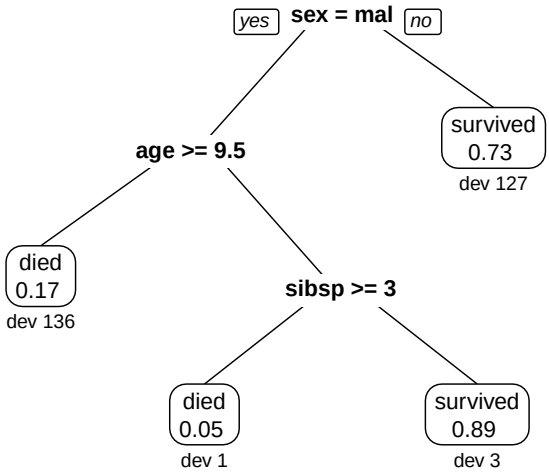


Figure 10: *Same as Figure 9, but with double newlines* `\n\n` *in the labels to move text below the boxes.*

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 9):

```
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 10):

```
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 11 is an example that wraps long split labels over multiple lines. The middle plot was produced by the following code. The maximum length of each line is 15 characters. Change the 15 to suit your purposes.

```
tree <- rpart(Price/1000 ~ Mileage + Type + Country, cu.summary)
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 = 15), collapse = "\n")
    }
    labs
}
prp(tree, split.fun = split.fun)
```

We can also reduce the amount of text with `clip.facs = TRUE`. This will remove the prefixes `"Country ="` and `"Type ="` (Section 4.1 has examples.) Also bear in mind that we can use `faclen` to control how the factors are abbreviated. All of these can be used together (right figure):

```
prp(tree, faclen = 0, clip.facs = TRUE, split.fun = split.fun)
```

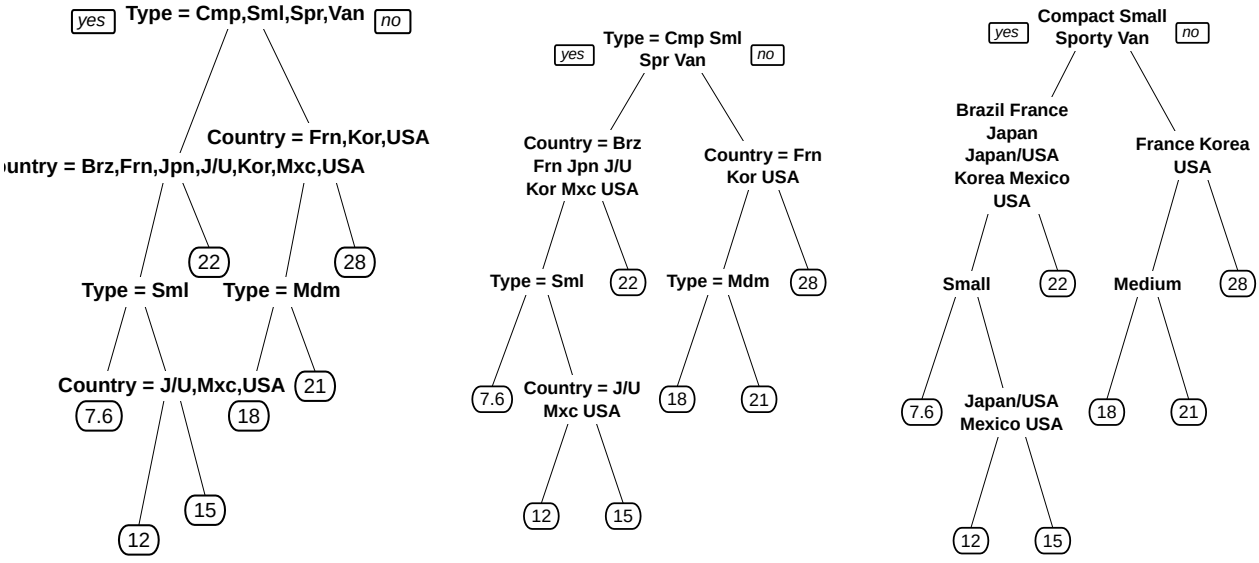


Figure 11: *Default* *Long labels split with a* `split.fun` *Various arguments, see code above*

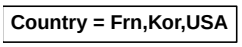
Note that we can generate labels of the form

```
"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



becomes narrower:

.

Figure 12 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(" = ", ":\n", labs)
}
prp(tree, extra = 100, under = T, yesno = F, split.fun = split.fun)
```

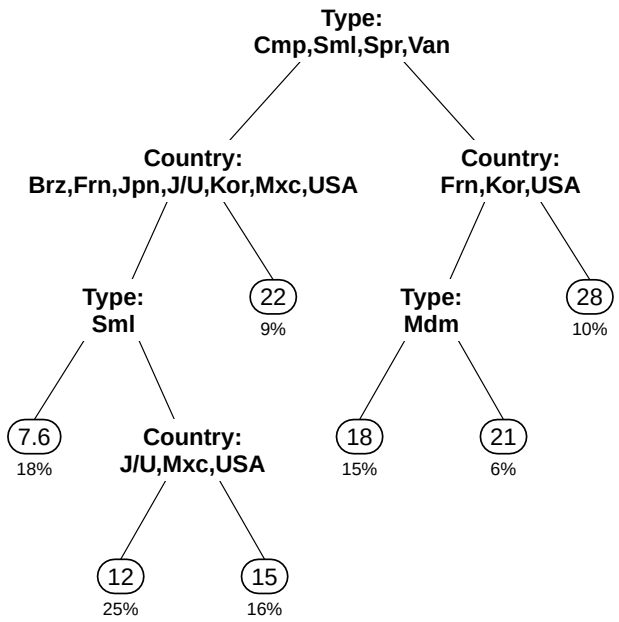


Figure 12: *Inserting a newline into the split labels with* ***split.fun****.*

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 significantly 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.

model response	auto `extra`	auto `box.palette`
continuous	100	`"Blues"`
two class (binary)	106	`"BuGn"`
three class	104	`list("Reds", "Grays", "Greens")`
four class	104	`list("Reds", "Oranges", "Grays", "Greens")`
five class	104	`list("Reds", "Oranges", "Grays", "Blues", "Greens")`
etc.		

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).

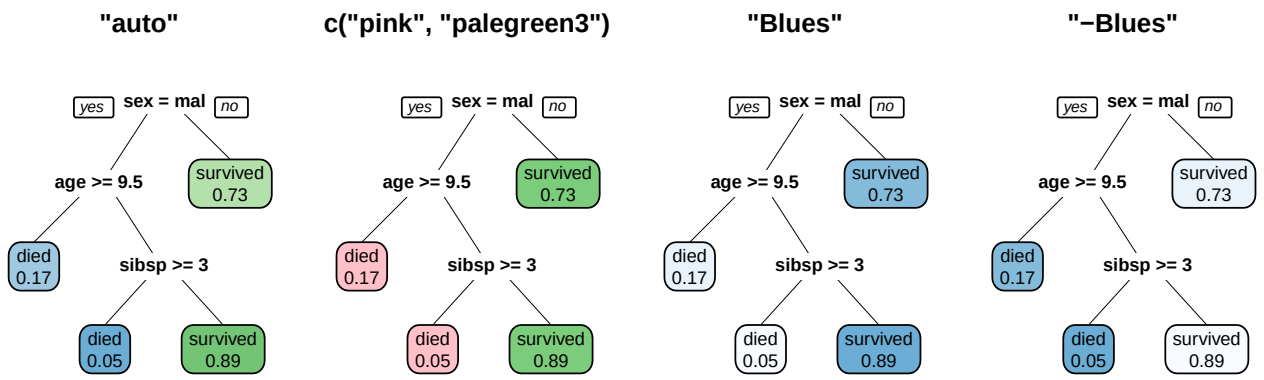


Figure 13: *The* `box.palette` *argument with a binary model.*

The code is (Figure13):

```
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 (Figure14):

```
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
```

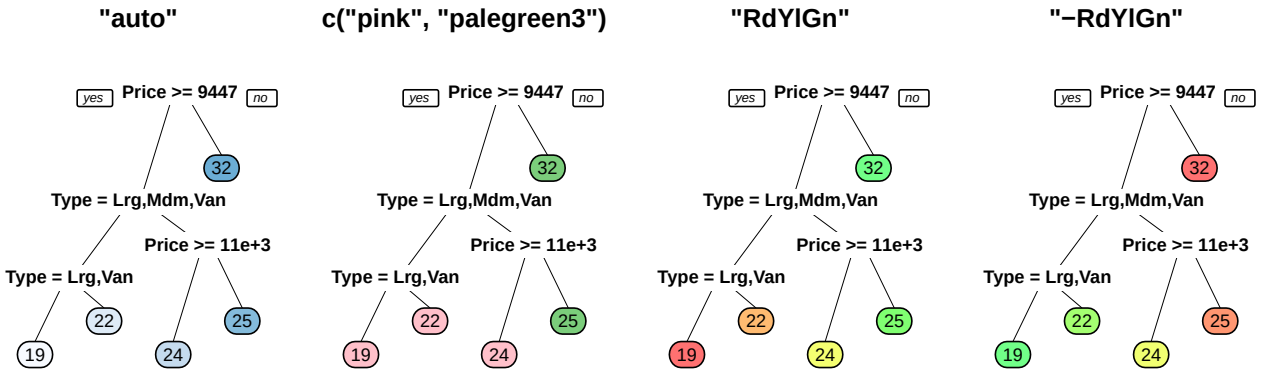


Figure 14: *The* `box.palette` *argument with a continuous response.*

7.2 The `col` and related arguments

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

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 ncompete 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 15). 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])
```

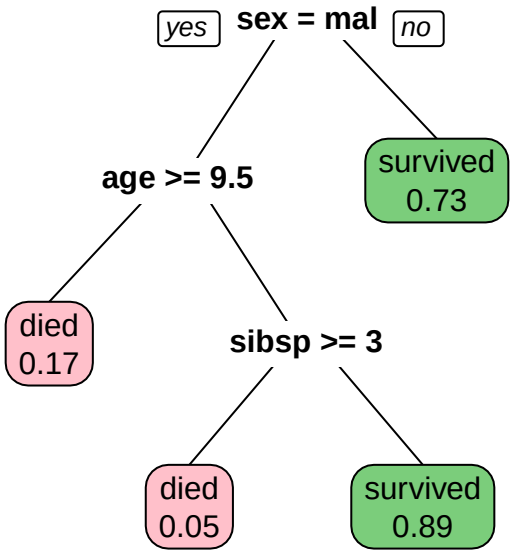


Figure 15: *Using the fitted value and the* `box.col` *argument to determine the color of the boxes.*

Figure 16 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:

```
heat.tree <- function(tree, low.is.green = FALSE, ...) { # dots args passed to prp
    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)
```

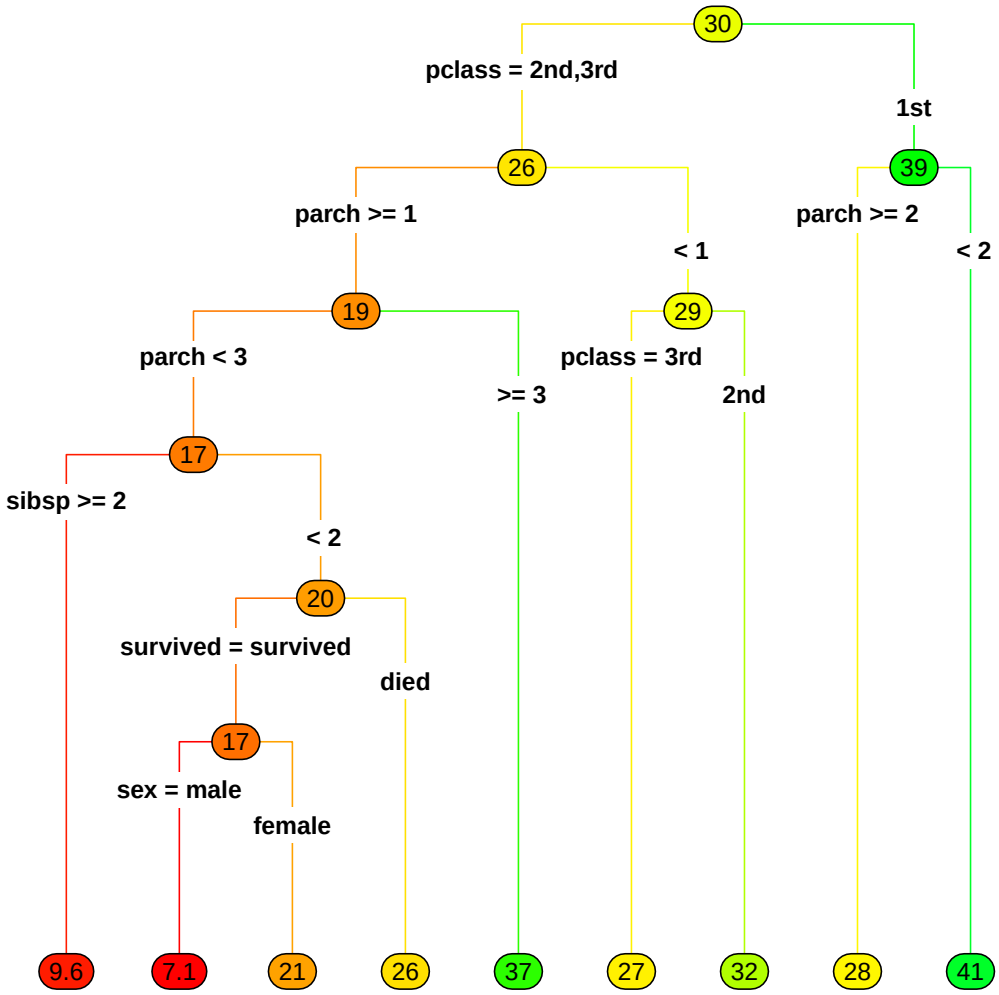


Figure 16: *Using the fitted value and the* `box.col` *argument to determine the color of regression nodes.*

The effect of the example on the previous page can be achieved more simply with the `box.palette` argument (Figure 17):

```
data(ptitanic)
tree <- rpart(age ~ ., data = ptitanic)
rpart.plot(tree, type = 4, extra = 0, branch.lty = 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.)

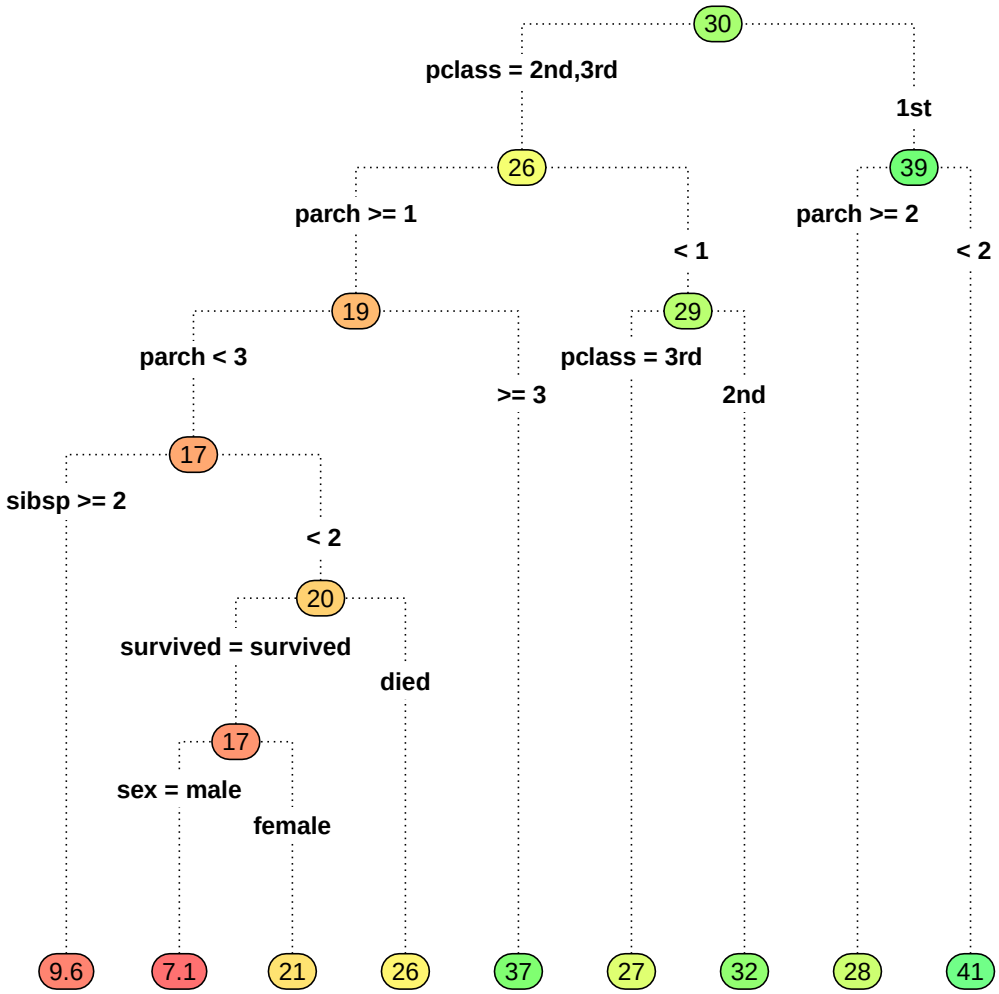


Figure 17: *Using the fitted value and the* `box.palette` *argument to determine the color of regression nodes.*

The following code creates a series of images – a movie – which shows how the tree is pruned on node complexity. Figure 18 is one of the plots produced by this code.

```
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
}
```

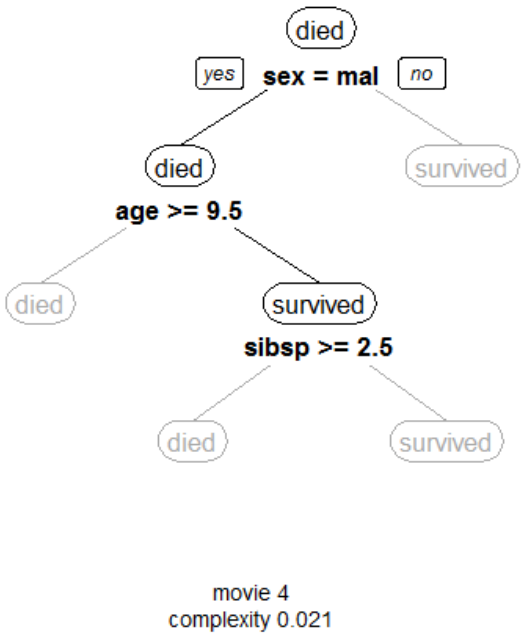


Figure 18: *Using the color arguments to indicate a nodes's complexity.*
Nodes with a complexity greater than a certain value (0.021) are grayed out in this example.

The following code shows how a tree is constructed in depth-first fashion, node by node (Figure 19):

```
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)
}
```

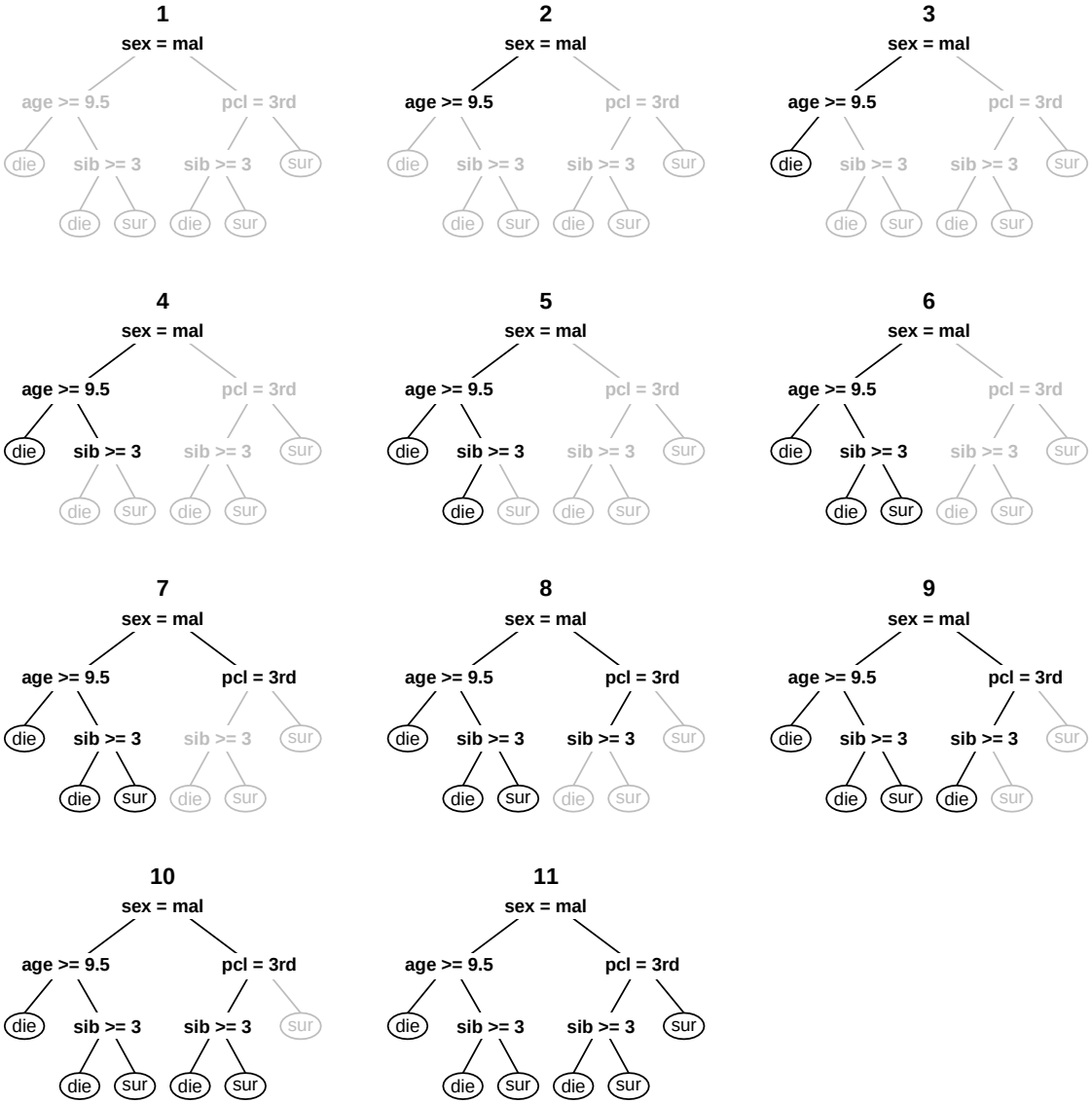


Figure 19: *Depth first tree construction.*

The following code highlights a node and all its ancestors (Figure 20):

```
# 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)
```

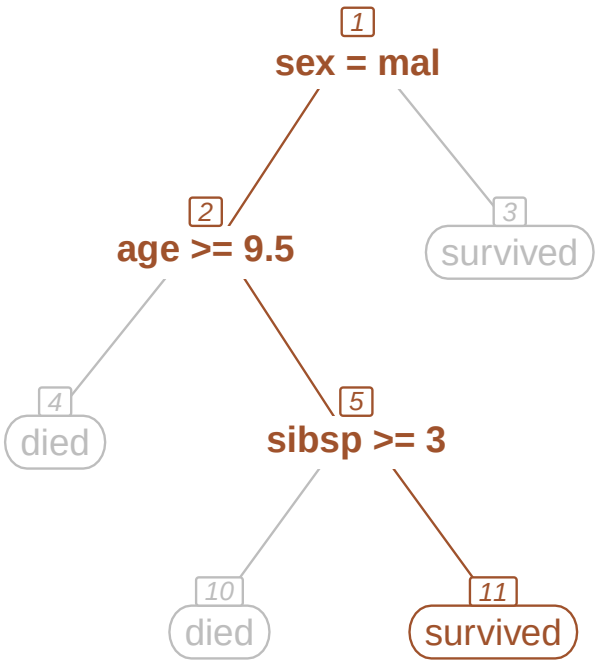


Figure 20: *Node 11 and its ancestors are highlighted.*

7.3 Working with `rpart` model code: some hints

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`:

```
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 21, 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 22 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).

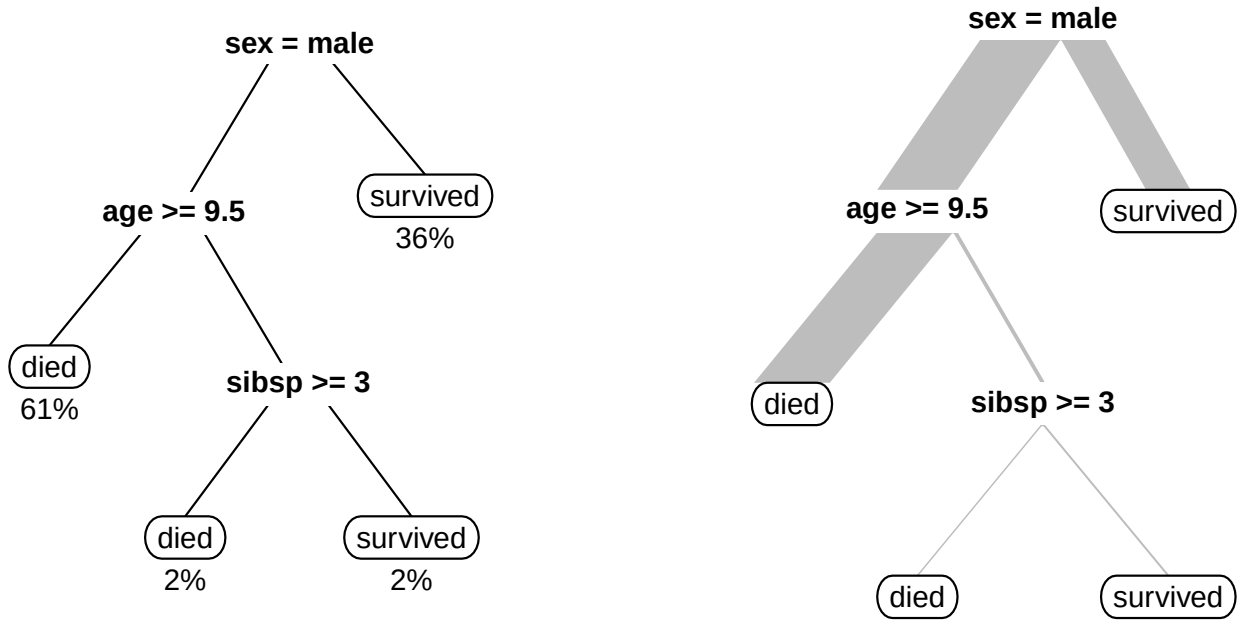


Figure 21: ***left*** *The percentage of observations in a node.*
right *That information represented by the width of the branches.*

Figure 22: *Misleading branch widths. The two branches have the same width, measured horizontally.*

9 Trimming a tree with the mouse

Set `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 `QUIT` button or right click, and `prp` will return the trimmed tree (in the `obj` field). Example (Figure 23):

```
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.

Additionally, we can use the `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.

```
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)
```

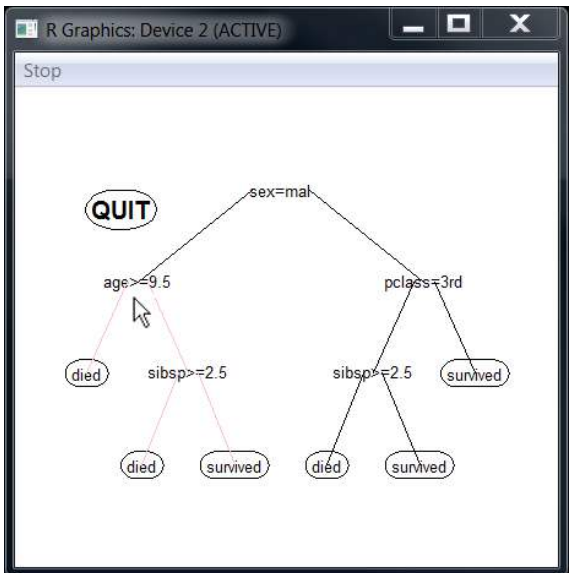


Figure 23: *Interactively trimming a tree with* ***`snip = TRUE`****.*

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 24 illustrates this on the `kyphosis` data:

```
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` [4] 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"`.

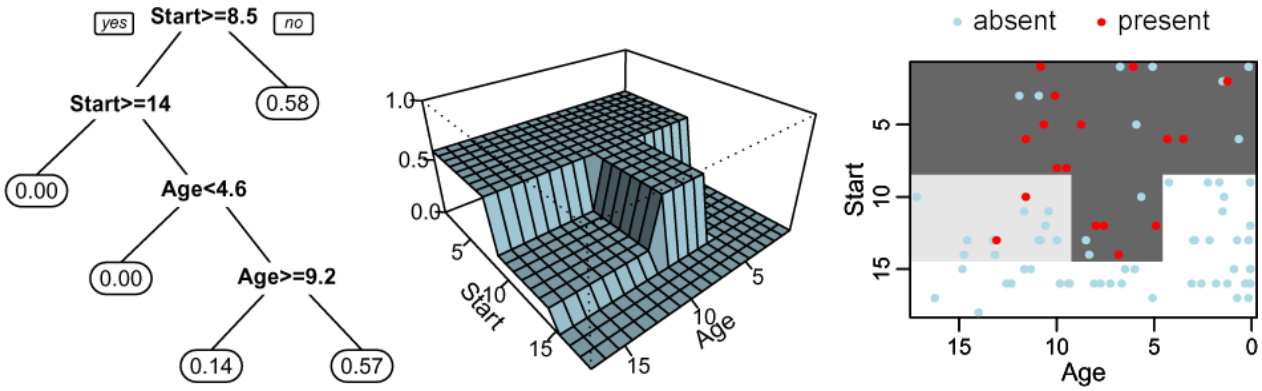


Figure 24: *The same tree represented in three different ways. The middle and right graphs show the predicted probability as a function of the predictors. The right graph is an aerial view of the middle graph.*

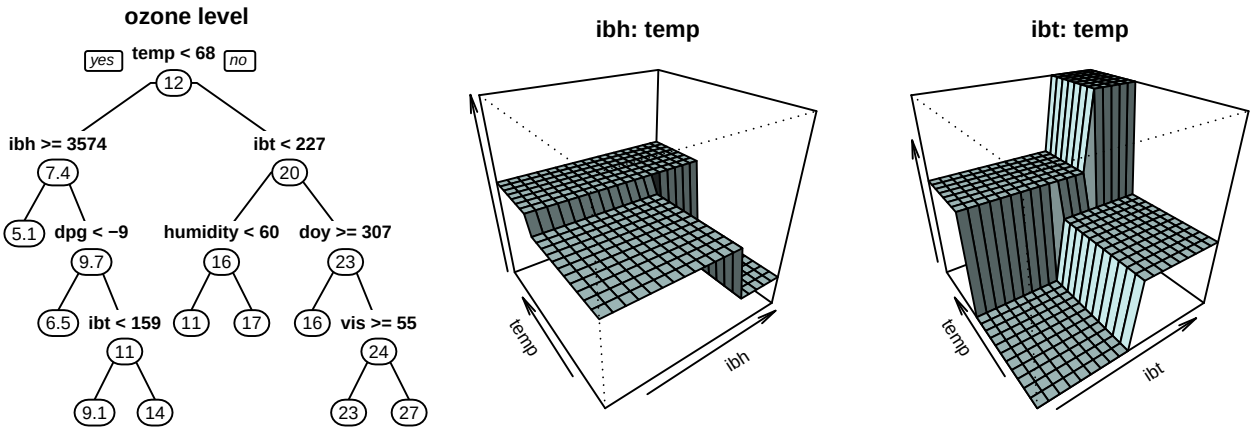


Figure 25: *A tree built from the* `ozone` *data, and regression surfaces for the predictors at the upper splits.*

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 25 shows a tree built from the `ozone` data:

```
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.

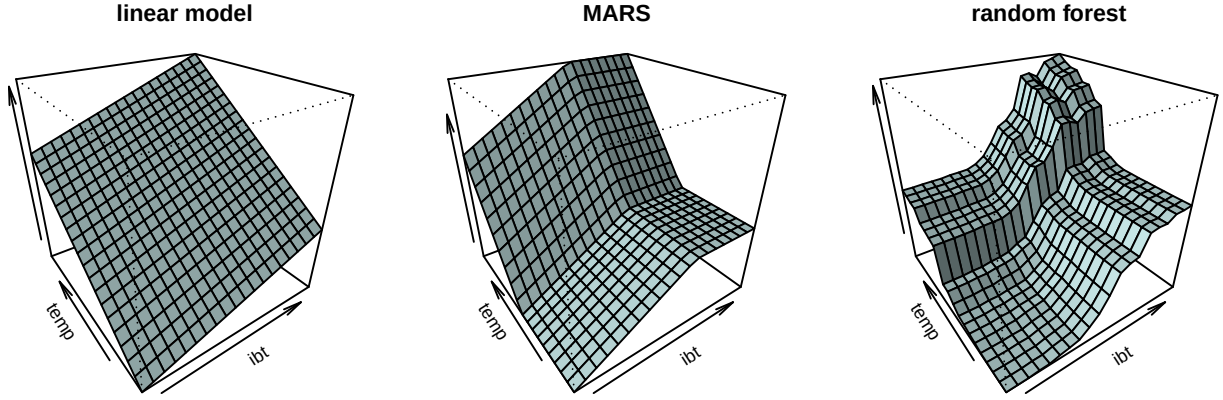


Figure 26: *Surfaces for other models using the* `ozone` *data. Compare to the right graph of Figure 25.*

It is interesting to compare the `rpart` tree to other models (Figure 26). The linear model gives a flat surface. MARS [5] generates a surface by combining hinge functions (see `https://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 displayed, and use `persp.theta` to force the same orientation on all `persp` plots. See the `plotmo` help page for details. The code is:

```
a <- lm(O3 ~ ., data = ozone1)                              # left graph, linear model
plotmo(a, degree1 = NA, degree2 = c("temp", "ibh"), persp.theta = -35)

library(earth) # earth is an implementation of MARS (MARS is a trademarked term)
a <- earth(O3 ~ ., data = ozone1, degree = 2)               # middle graph, MARS
plotmo(a, degree1 = NA, degree2 = c("temp", "ibh"), persp.theta = -35)

library(randomForest)
a <- randomForest(O3 ~ ., data = ozone1)                    # right graph, random forest
plotmo(a, degree1 = NA, degree2 = c("temp", "ibh"), persp.theta = -35)
```

11 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.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 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`.

11.1 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`:

```
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 = "/")
```

12 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 27 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 >= 17` 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

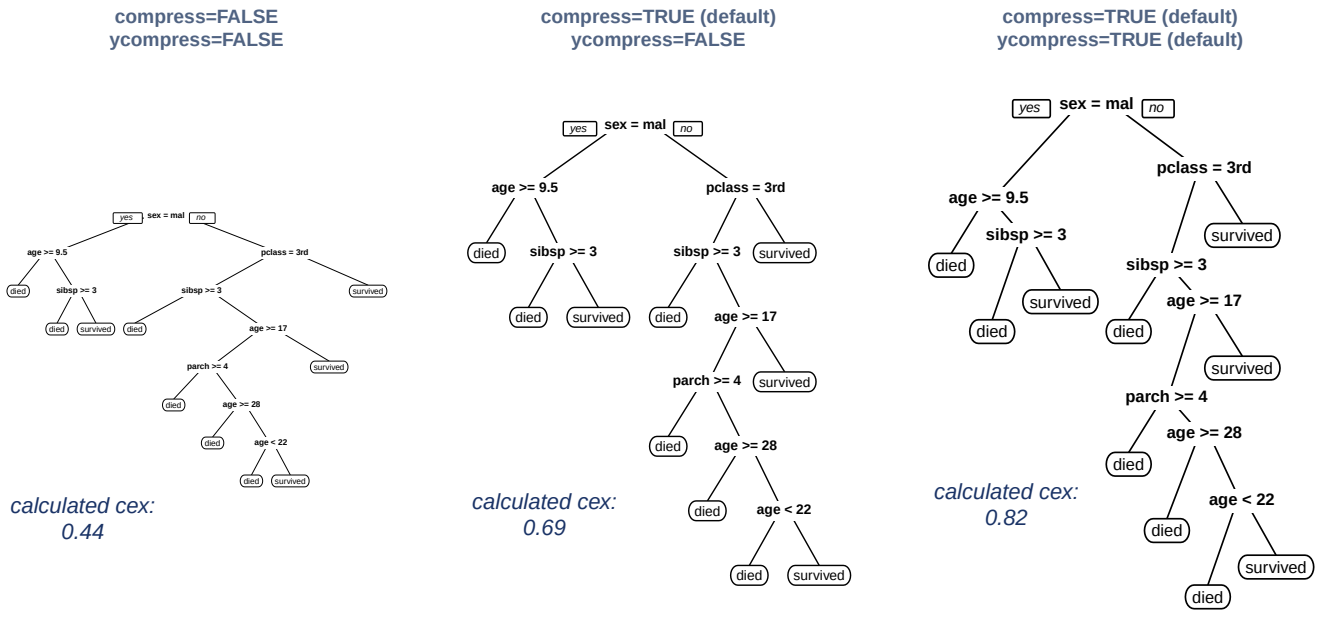


Figure 27: *The* `compress` *and* `ycompress` *arguments*

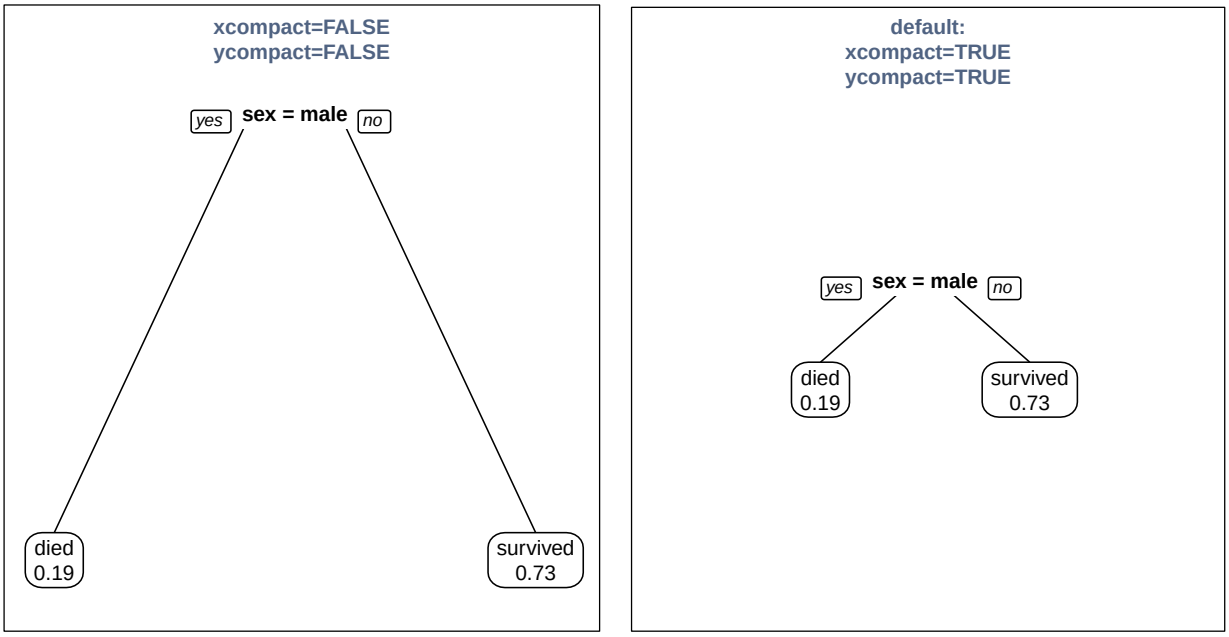


Figure 28: *Small trees are compacted by default, as shown on the right.*

for 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 27.

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 28). 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:

```
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)
```

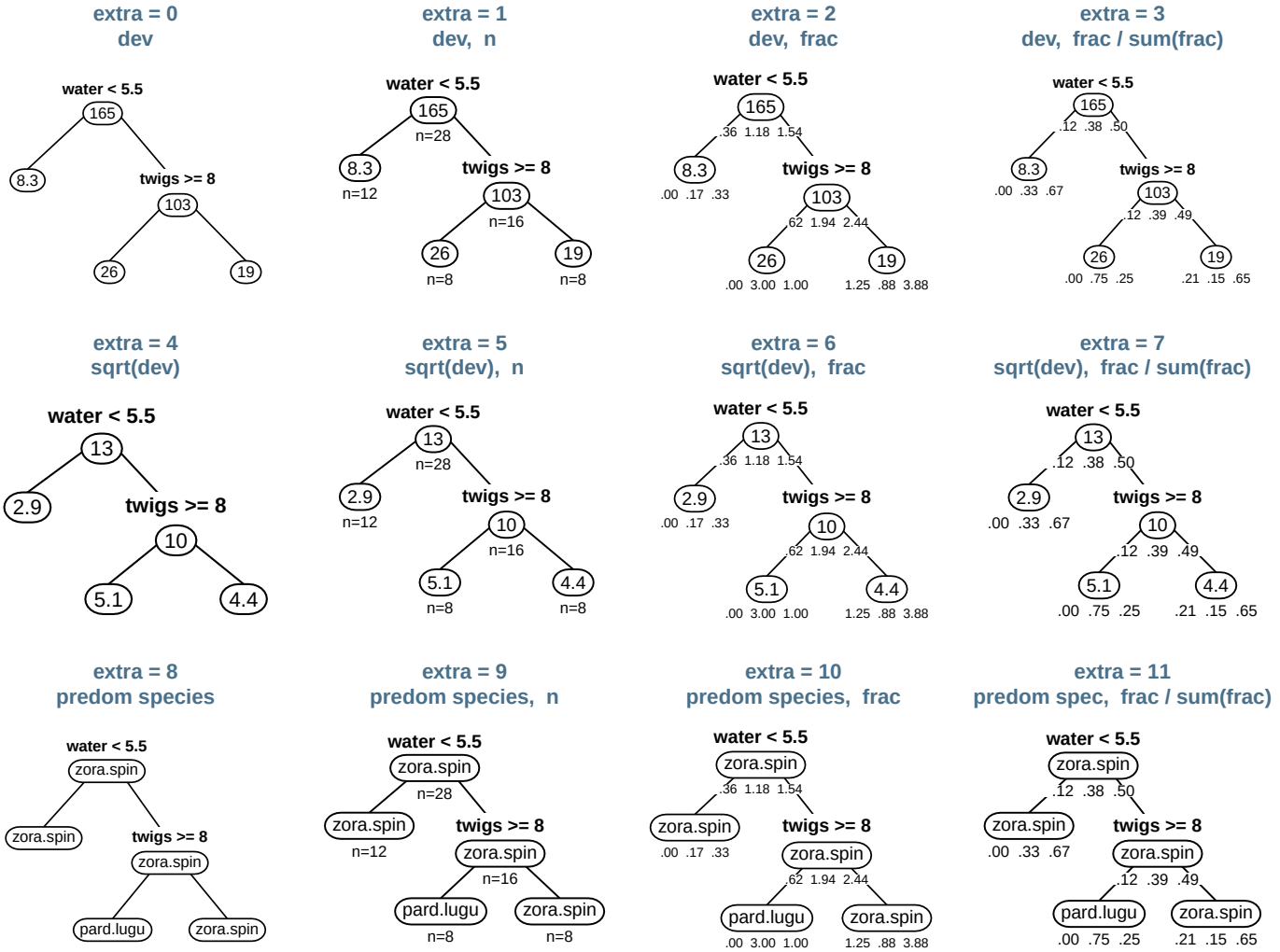


Figure 29: *The* `extra` *argument with an* `mrt` *model.*

References

[1] Cynthia A. Brewer. *www.ColorBrewer.org*. available online, Accessed Jan 2016. https://colorbrewer2.org/. Cited on page 18.

[2] Glenn De'ath. *mvpart: Multivariate partitioning*, 2014. R package, https://CRAN.R-project.org/package=mvpart. Cited on page 35.

[3] Andy Liaw, Mathew Weiner; Fortran original by Leo Breiman, and Adele Cutler. *randomForest: Breiman and Cutler's random forests for regression and classification*, 2014. R package, https://CRAN.R-project.org/package=randomForest. Cited on page 31.

[4] S. Milborrow. *plotmo: Plot a Model's Residuals, Response, and Partial Dependence Plots*, 2018. R package, https://CRAN.R-project.org/package=plotmo. Cited on page 29.

[5] S. Milborrow. Derived from mda:mars by T. Hastie and R. Tibshirani. *earth: Multivariate Adaptive Regression Splines*, 2025. R package, http://www.milbo.org/earth/. Cited on page 31.

[6] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, 2014. https://www.R-project.org/. Cited on page 2.

[7] Terry Therneau and Beth Atkinson. *rpart: Recursive Partitioning and Regression Trees*, 2014. R package, https://CRAN.R-project.org/package=rpart. Cited on page 2.

[8] Graham J. Williams. *Data Mining with Rattle and R: The art of excavating data for knowledge discovery*. Springer, 2011. https://togaware.com/index.html. Cited on page 11.