partykit: A Toolkit for Recursive Partytioning

Torsten Hothorn
Ludwig-Maximilians-Universität
München

Achim Zeileis
Universität Innsbruck

Abstract

This vignette is intended to be a short technical introduction to the **partykit** package. It is still unfinished but hopefully already helpful to some interested users.

The purpose of the package is to provide a toolkit with infrastructure for representing, summarizing, and visualizing tree-structured regression and classification models. Thus, the focus is not on *inferring* such a tree structure from data but to *represent* a given tree so that print/plotting and computing predictions can be performed in a standardized way. In particular, this unified infrastructure can be used for reading/coercing tree models from different sources (**rpart**, **RWeka**, PMML) yielding objects that share functionality for `print()`/`plot()`/`predict()` methods.

Keywords:~recursive partitioning, regression trees, classification trees, decision trees.

1. Motivating example and overview

To illustrate how **partykit** can be used to represent trees, a simple artificial data set from Witten & Frank's book *Data Mining: Practical Machine Learning Tools and Techniques* is used. It concerns the conditions suitable for playing some unspecified game.

```
> data("WeatherPlay", package = "partykit")
> WeatherPlay

    outlook temperature humidity windy play
1     sunny          85       85 false   no
2     sunny          80       90  true   no
3  overcast          83       86 false  yes
4     rainy          70       96 false  yes
5     rainy          68       80 false  yes
6     rainy          65       70  true   no
7  overcast          64       65  true  yes
8     sunny          72       95 false   no
9     sunny          69       70 false  yes
10    rainy          75       80 false  yes
11    sunny          75       70  true  yes
12 overcast          72       90  true  yes
13 overcast          81       75 false  yes
14    rainy          71       91  true   no
```

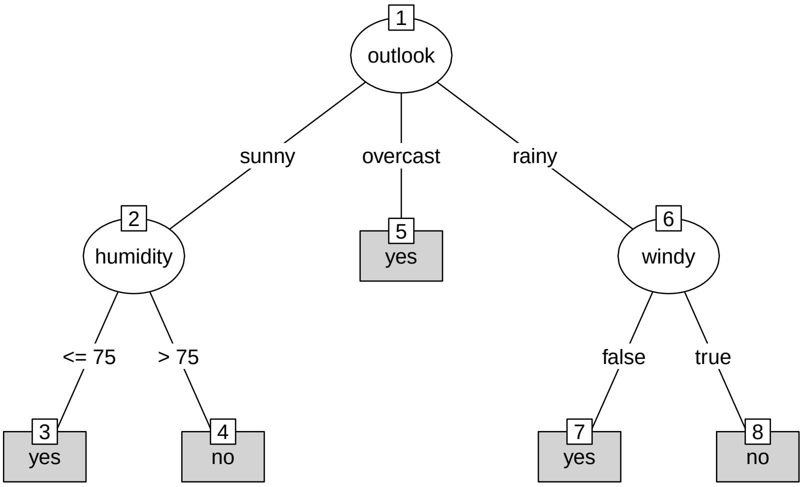

Figure 1: Decision tree for `play` decision based on weather conditions in `WeatherPlay` data.

To represent the `play` decision based on the corresponding weather condition variables one could use the tree displayed in Figure~1. For now, it is ignored how this tree was *inferred* and it is simply assumed to be given.

To represent this tree (also known as recursive partition) in **partykit**, two basic building blocks are used: splits of class "`partysplit`" and nodes of class "`partynode`". The resulting recursive partition can then be associated with a data set in an object of class "`party`".

Splits

First, we employ the `partysplit()` function to creat the three splits in the "weather tree" from Figure~1. The function takes the following arguments

```
partysplit(varid, breaks = NULL, index = NULL, ..., info = NULL)
```

where `varid` is an integer id of the variable used for splitting e.g., `1L` for `outlook`, `3L` for `humidity`, `4L` for `windy` etc. Then, `breaks` and `index` determine which observations are sent to which of the branches, e.g., `breaks = 75` for the humidity split. Apart from further arguments not shown above (and just comprised under '`...`'), some arbitrary information can be associated with a "`partysplit`" object by passing it to the `info` argument. The three splits from Figure~1 can then be created via

```
> sp_o <- partysplit(1L, index = 1:3)
> sp_h <- partysplit(3L, breaks = 75)
> sp_w <- partysplit(4L, index = 1:2)
```

For the numeric `humidity` variable the `breaks` are set while for the factor variables `outlook` and `windy` the information is supplied which of the levels should be associated with which of the branches of the tree.

Nodes

Second, we use these splits in the creation of the whole decision tree. In **partykit** a tree is represented by a "partynode" object which is recursive in that it may have "kids" that are again "partynode" objects. These can be created with the function

```
partynode(id, split = NULL, kids = NULL, ..., info = NULL)
```

where id is an integer identifier of the node number, split is a "partysplit" object, and kids is a list of "partynode" objects. Again, there are further arguments not shown (...) and arbitrary information can be supplied in info. The whole tree from Figure˜1 can then be created via

```
> pn <- partynode(1L, split = sp_o, kids = list(
+   partynode(2L, split = sp_h, kids = list(
+     partynode(3L, info = "yes"),
+     partynode(4L, info = "no"))),
+   partynode(5L, info = "yes"),
+   partynode(6L, split = sp_w, kids = list(
+     partynode(7L, info = "yes"),
+     partynode(8L, info = "no")))))
```

where the previously created "partysplit" objects are used as splits and the nodes are simply numbered from˜1 to˜8. For the terminal nodes of the tree there are no kids and the corresponding play decision is stored in the info argument. Printing the "partynode" object reflects the recursive structure stored.

```
> pn
```

```
[1] root
|   [2] V1 in (-Inf,1]
|   |   [3] V3 <= 75 *
|   |   [4] V3 > 75 *
|   [5] V1 in (1,2] *
|   [6] V1 in (2, Inf]
|   |   [7] V4 <= 1 *
|   |   [8] V4 > 1 *
```

However, the displayed information is still rather raw as it has not yet been associated with the WeatherPlay data set.

Trees (or recursive partitions)

Hence, in a third step the recursive tree structure stored in pn is coupled with the corresponding data in a "party" object.

```
> py <- party(pn, WeatherPlay)
> py
```

```
[1] root
|   [2] outlook in sunny
|   |   [3] humidity <= 75: yes
|   |   [4] humidity > 75: no
|   [5] outlook in overcast: yes
|   [6] outlook in rainy
|   |   [7] windy in false: yes
|   |   [8] windy in true: no
```

And Figure~1 can easily be created by

```
> plot(py)
```

In addition to `print()` and `plot()`, the `predict()` method can now be applied, yielding the predicted terminal node IDs.

```
> predict(py, newdata = WeatherPlay)
```

In addition to the "`partynode`" and the "`data.frame`", the function `party()` takes several further arguments

```
party(node, data, fitted = NULL, terms = NULL, ..., info = NULL)
```

i.e., `fitted` values, a `terms` object, arbitrary additional `info`, and again some further arguments comprised in `...`.

Further information

More detailed technical information is provided in the subsequent sections. Section~2, 3, and~4 discuss the `partysplit()`, `partynode()`, and `party()` functions, respectively. Section~5 illustrates how this infrastructure can be employed in a function that recursively *infers* a tree.

Design principles

To facilitate reading of the subsequent sections, two design principles employed in the creation of **partykit** are briefly explained.

(1)~Many helper utilities are encapsulated in functions that follow a simple naming convention. To extract/compute some information *foo* from splits, nodes, or trees, **partykit** provides *foo*`_split`, *foo*`_node`, *foo*`_party` functions (that are applicable to "`partysplit`", "`partynode`", and "`party`" objects, repectively). An example for the information *foo* might be `kidids`. Such functions are typically not to be called by the end-user but potentially by package designers that want to build functionality on top of **partykit**.

(2)~Printing and plotting relies on *panel functions* that visualize and/or format certain aspects of the resulting display, e.g., that of inner nodes, terminal nodes, headers, footers, etc. A simple example would be printing with a custom panel function for formatting the terminal node:

```
> print(py,
+   terminal_panel = function(node) paste(": play=", node$info, sep = ""))

[1] root
|   [2] outlook in sunny
|   |   [3] humidity <= 75: play=yes
|   |   [4] humidity > 75: play=no
|   [5] outlook in overcast: play=yes
|   [6] outlook in rainy
|   |   [7] windy in false: play=yes
|   |   [8] windy in true: play=no
```

Furthermore, arguments like `terminal_panel` can also take *panel-generating functions*, i.e., functions that produce a panel function when applied to the "`party`" object.

2. Splits

A split is basically a function that maps data, more specifically a partitioning variable, to daugther nodes. Objects of class "`partysplit`" are designed to represent such functions and are set-up by the `partysplit()` constructor:

```
> ## binary split in numeric variable `Sepal.Length'
> sl5 <- partysplit(which(names(iris) == "Sepal.Length"), breaks = 5)
> class(sl5)

[1] "partysplit"
```

The internal structure of class "`partysplit`" contains information about the partitioning variable, the split-points, the handling of split-points, the treatment of observations with missing values and the daughter nodes to send observations to:

```
> unclass(sl5)

$varid
[1] 1

$breaks
[1] 5

$index
NULL

$right
[1] TRUE

$prob
```

```
NULL

$info
NULL
```

Here, the split is defined in the first variable (corresponds to `Sepal.Length` in data frame `iris`) and the splitting rule is `Sepal.Length` ≤ 5:

```
> character_split(sl5, data = iris)

$name
[1] "Sepal.Length"

$levels
[1] "<= 5" "> 5"
```

This representation of splits is completely abstract and, most importantly, independent of any data. Now, data comes into play when we actually want to perform splits:

```
> kidids_split(sl5, data = iris)

  [1] 2 1 1 1 1 2 1 1 1 1 2 1 1 1 2 2 2 2 2 2 2 2 1 2 1 1 1 2 2 1 1 2
 [33] 2 2 1 1 2 1 1 2 1 1 1 1 2 1 2 1 2 1 2 2 2 2 2 2 2 1 2 2 1 2 2 2
 [65] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 2 2
 [97] 2 2 2 2 2 2 2 2 2 2 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
[129] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
```

For each observation in `iris` the split is performed and the number of the daughter node to send this observation to is returned. Of course, this is a very complicated way of saying

```
> (!with(iris, Sepal.Length <= 5)) + 1

  [1] 2 1 1 1 1 2 1 1 1 1 2 1 1 1 2 2 2 2 2 2 2 2 1 2 1 1 1 2 2 1 1 2
 [33] 2 2 1 1 2 1 1 2 1 1 1 1 2 1 2 1 2 1 2 2 2 2 2 2 2 1 2 2 1 2 2 2
 [65] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 2 2
 [97] 2 2 2 2 2 2 2 2 2 2 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
[129] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
```

Formally, a split is a function f mapping an element $x = (x_1, \dots, x_p)$ of a p-dimensional sample space $\mathcal{X}$ into a set of k daugther nodes $\mathcal{D} = \{d_1, \dots, d_k\}$. This mapping is defined as a composition $f = h \circ g$ of two functions $g : \mathcal{X} \rightarrow \mathcal{I}$ and $h : \mathcal{I} \rightarrow \mathcal{D}$ with index set $\mathcal{I} = \{1, \dots, l\}, l \geq k$.

Let $\mu = (-\infty, \mu_1, \dots, \mu_{l-1}, \infty)$ denote the split points ($(\mu_1, \dots, \mu_{l-1})$ = `breaks`). We are interested to split according to the information contained in the ith element of x (i = `varid`). For numeric x_i, the split points are also numeric. If x_i is a factor at levels $1, \dots, K$, the default split points are $\mu = (-\infty, 1, \dots, K - 1, \infty)$.

The function g essentially determines, which of the intervals (defined by μ) the value x_i is contained in (I denotes the indicator function here):

$$x \mapsto g(x) = \sum_{j=1}^{l} j I_{\mathcal{A}(j)}(x_i)$$

where $\mathcal{A}(j) = (\mu_{j-1}, \mu_j]$ for `right = TRUE` except $\mathcal{A}(l) = (\mu_{l-1}, \infty)$. If `right = FALSE`, then $\mathcal{A}(j) = [\mu_{j-1}, \mu_j)$ except $\mathcal{A}(1) = (-\infty, \mu_1)$. Note that with some categorical variable x_i and default split points, g is the identity.

Now, h maps from the index set $\mathcal{I}$ into the set of daugther nodes:

$$f(x) = h(g(x)) = d_{\sigma_{g(x)}}$$

where $\sigma = (\sigma_1, \dots, \sigma_l) \in \{1, \dots, k\}^l$ (`index`). By default, $\sigma = (1, \dots, l)$ and $k = l$.

If x_i is missing, then $f(x)$ is randomly drawn with $\mathbb{P}(f(x) = d_j) = p_j, j = 1, \dots, k$ for a discrete probability distribution $p = (p_1, \dots, p_k)$ over the k daugther nodes (`prob`).

In the simplest case of a binary split in a numeric variable x_i, there is only one split point μ_1 and, with $\sigma = (1, 2)$, observations with $x_i \le \mu_1$ are sent to daugther node d_1 and observations with $x_i > \mu_1$ to d_2. However, this representation of splits is general enough to deal with more complicated set-ups like surrogate splits, where typically the index needs modification, for example $\sigma = (2, 1)$, categorical splits, i.e., there is one data structure for both ordered and unordered splits, multiway splits, and functional splits. The latter can be implemented by defining a new artificial splitting variable x_{p+1} by means of a potentially very complex function of x later used for splitting.

As an example, consider a split in a categorical variable at three levels where the first two levels go to the left daugther node and the third one to the right daugther node:

```
> ## binary split in factor `Species'
> sp <- partysplit(which(names(iris) == "Species"), index = c(1L, 1L, 2L))
> character_split(sp, data = iris)

$name
[1] "Species"

$levels
[1] "setosa, versicolor" "virginica"

> table(kidids_split(sp, data = iris), iris$Species)

    setosa versicolor virginica
  1     50         50         0
  2      0          0        50
```

The internal structure of this object contains the `index` slot

```
> unclass(sp)
```

```
$varid
[1] 5

$breaks
NULL

$index
[1] 1 1 2

$right
[1] TRUE

$prob
NULL

$info
NULL
```

that maps levels to daughter nodes. This mapping is also useful with splits in ordered variables, for example when representing multiway splits:

```
> ## multiway split in numeric variable `Sepal.Width',
> ## higher values go to the first kid, smallest values
> ## to the last kid
> sw23 <- partysplit(which(names(iris) == "Sepal.Width"),
+   breaks = c(3, 3.5), index = 3:1)
> character_split(sw23, data = iris)

$name
[1] "Sepal.Width"

$levels
[1] "(3.5, Inf]" "(3,3.5]"    "(-Inf,3]"

> table(kidids_split(sw23, data = iris),
+   cut(iris$Sepal.Width, breaks = c(-Inf, 2, 3, Inf)))

    (-Inf,2] (2,3] (3, Inf]
  1        0     0       19
  2        0     0       48
  3        1    82        0
```

The mapping of classes of the categorized numeric variable to daugther nodes can be changed by modifying `index`:

```
> sw23 <- partysplit(which(names(iris) == "Sepal.Width"),
+   breaks = c(3, 3.5), index = c(1L, 3L, 2L))
> character_split(sw23, data = iris)
```

```
$name
[1] "Sepal.Width"

$levels
[1] "(-Inf,3]"   "(3.5, Inf]" "(3,3.5]"
```

The additional argument `prop` is used to specify a discrete probability distribution over the daugther nodes that is used to map observations with missing values to daugther nodes. Furthermore, the `info` argument and slot takes arbitrary objects to be stored with the split (for example split statistics) but is not structured at the moment.

The slots of "`partysplit`" objects shall be accessed by the corresponding accessor functions.

3. Nodes

Inner and terminal nodes are represented by objects of class "`partynode`". Each node has a unique identifier `id`. A node consisting only of such an identifier (and possibly additional information in `info`) is a terminal node:

```
> n1 <- partynode(id = 1L)
> is.terminal(n1)

[1] TRUE

> print(n1)

[1] root
```

Inner nodes have to have a primary split `split` and at least two daugther nodes. The daugther nodes are objects of class "`partynode`" itself and thus represent the recursive nature of this data structure. The daugther nodes are pooled in a list `kids`. In addition, a list of "`partysplit`" objects offering surrogate splits can be supplied; a list of "`partysplit`" objects in slot `surrogates` defines such additional splits (mostly used for handling missing values). Note that "`partynode`" objects aren't connected to the actual data.

Based on the binary split `sl5` defined in the previous section, we set-up an inner node with two terminal daugther nodes and print this stump (the data is needed because neither split nor nodes contain information about variable names or levels):

```
> n1 <- partynode(id = 1L, split = sl5, kids = sapply(2:3, partynode))
> print(n1, data = iris)

[1] root
|   [2] Sepal.Length <= 5 *
|   [3] Sepal.Length > 5 *
```

Now that we have defined our first simple tree, we want to assign observations to terminal nodes:

```
> fitted_node(n1, data = iris)

  [1] 3 2 2 2 2 3 2 2 2 2 3 2 2 2 3 3 3 3 3 3 3 3 2 3 2 2 2 3 3 2 2 3
 [33] 3 3 2 2 3 2 2 3 2 2 2 2 3 2 3 2 3 2 3 3 3 3 3 3 3 2 3 3 2 3 3 3
 [65] 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 2 3 3
 [97] 3 3 3 3 3 3 3 3 3 3 2 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
[129] 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
```

Here, the `ids` of the terminal node each observations falls into are returned. Alternatively, we could compute the position of these daugther nodes in the list `kids`:

```
> kidids_node(n1, data = iris)

  [1] 2 1 1 1 1 2 1 1 1 1 2 1 1 1 2 2 2 2 2 2 2 2 1 2 1 1 1 2 2 1 1 2
 [33] 2 2 1 1 2 1 1 2 1 1 1 1 2 1 2 1 2 1 2 2 2 2 2 2 2 1 2 2 1 2 2 2
 [65] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 2 2
 [97] 2 2 2 2 2 2 2 2 2 2 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
[129] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
```

Furthermore, the `info` argument and slot takes arbitrary objects to be stored with the node (predictions, for example, but we will handle this issue later). The slots can be extracted by means of the corresponding accessor functions.

A number of methods are defined for "`partynode`" objects. `is.partynode()` checks if the argument is a valid "`partynode`" object. `is.terminal()` is `TRUE` for terminal nodes and `FALSE` for inner nodes. The subset methods return the "`partynode`" object corresponding to the `i`th kid:

```
> n1[2]

[NA] root
```

The `as.partynode()` and `as.list()` methods can be used to convert flat list structures into recursive "`partynode`" objects and vice versa. `as.partynode()` applied to "`partynode`" objects renumbers the recursive nodes starting with root node identifier `from`.

`length()` gives the number of kid nodes of the root node, `depth()` the depth of the tree and `width()` the number of terminal nodes.

4. Trees

Although tree structures can be represented by "`partynode`" objects, a tree is more than a number of nodes and splits. More information about (parts of the) corresponding data is necessary for high-level computations on trees.

Objects of class "`party`" basically consist of a "`partynode`" object representing the tree structure in a recursive way and data. The `data` argument takes a "`data.frame`" which, however, might have zero columns. Optionally, a "`data.frame`" with at least one variable `(fitted)` containing the terminal node numbers of data used for fitting the tree may be specified along with a `terms` object or any additional (currently unstructured) information as `info`. Argument `names` defines names for all nodes in `node`.

```
> t1 <- party(n1,
+   data = iris,
+   fitted = data.frame(
+     "(fitted)" = fitted_node(n1, data = iris),
+     "(response)" = iris$Species,
+     check.names = FALSE)
+ )
> t1

[1] root
|   [2] Sepal.Length <= 5: *
|   [3] Sepal.Length > 5: *
```

5. My first tree

Package **partykit** does not offer unified infrastructure for growing trees. However, once you know how to estimate splits from data, it is fairly straightforward to implement trees. Consider a very simple tree algorithm. We assume that both response and features are numeric. We search for the binary best split by means of t-test p-values, i.e., we cycle through all variables and potential split points and assess the quality of the split by comparing the distributions of the response in the so-defined two groups. We select the feature/split point combination with lowest two-sided p-value, however only if this result is significant at level $\alpha = 0.05$.

This strategy can be implemented based on the data (response and features) and some case weights as follows (`response` is just the name of the response and `data` is a data frame with all variables):

```
> findsplit <- function(response, data, weights) {
+
+   ### extract response values from data
+   y <- data[[response]]
+
+   logpmin <- 0
+   xselect <- NULL
+
+   ### cycle through all features
+   for (i in which(names(data) != response)) {
+
+     ### expand data
+     x <- data[[i]]
+     xt <- rep(x, weights)
+     yt <- rep(y, weights)
+
+     ### potential split points (not too many)
+     qx <- unique(quantile(xt,
+                  prob = seq(from = 0.1, to = 0.9, by = 0.05)))
+
+     ### assess all potential splits by their t-test
+     ### log-p-value
+     logp <- sapply(qx, function(q) {
+       tt <- t.test(yt[xt <= q], yt[xt > q])
+       pt(-abs(tt$statistic), tt$parameter, log = TRUE) + log(2)
+     })
+
+     ### if the best split in variable i significant AND
+     ### better than what we already had store this information
+     if (min(logp) < logpmin & min(logp) < log(0.05)) {
+       logpmin <- min(logp)
+       xselect <- i
+       splitpoint <- qx[which.min(logp)]
+     }
+   }
+
+   ### no significant split found, give up
+   if (is.null(xselect)) return(NULL)
+
+   ### return split as partysplit object
+   return(partysplit(
+       varid = as.integer(xselect),         ### which variable?
+       breaks = as.numeric(splitpoint),   ### which split point?
+       info = list(pvalue = exp(logpmin)  ### save p-value in addition
+   )))
+ }
```

In order to actually grow a tree on data, we have to set-up the recursion for growing a recursive "partynode" structure:

```
> growtree <- function(id = 1L, response, data, weights) {
+
+   ### for less than 30 obs. stop here
+   if (sum(weights) < 30) return(partynode(id = id))
+
+   ### find best split
+   sp <- findsplit(response, data, weights)
+   ### no split found, stop here
+   if (is.null(sp)) return(partynode(id = id))
+
+   ### actually split the data
+   kidids <- kidids_split(sp, data = data)
+
+   ### set-up all daugther nodes
+   kids <- vector(mode = "list", length = max(kidids))
+   for (kidid in 1:max(kidids)) {
```

```
+   ### select obs for current node
+   w <- weights
+   w[kidids != kidid] <- 0
+   ### get next node id
+   if (kidid > 1) {
+     myid <- max(nodeids(kids[[kidid - 1]]))
+   } else {
+     myid <- id
+   }
+   ### start recursion on this daugther node
+   kids[[kidid]] <- growtree(id = as.integer(myid + 1), response, data, w)
+   }
+
+   ### return nodes
+   return(partynode(id = as.integer(id), split = sp, kids = kids))
+ }
```

A very rough sketch of formula-based user-interface needs to set-up the data and call `growtree()`:

```
> mytree <- function(formula, data, weights = NULL) {
+
+   ### name of the response variable
+   response <- all.vars(formula)[1]
+   ### data without missing values, response comes last
+   data <- data[complete.cases(data), c(all.vars(formula)[-1], response)]
+   ### data is numeric
+   stopifnot(all(sapply(data, is.numeric)))
+
+   if (is.null(weights)) weights <- rep(1, nrow(data))
+   ### weights are case weights, i.e., integers
+   stopifnot(length(weights) == nrow(data) &
+     max(abs(weights - floor(weights))) < .Machine$double.eps)
+
+   ### grow tree
+   nodes <- growtree(id = 1L, response, data, weights)
+
+   ### compute terminal node number for each obs.
+   fitted <- fitted_node(nodes, data = data)
+   ### return rich object
+   ret <- party(nodes,
+     data = data,
+     fitted = data.frame(
+       "(fitted)" = fitted,
+       "(response)" = data[[response]],
+       "(weights)" = weights,
+       check.names = FALSE),
+     terms = terms(formula))
+   as.constparty(ret)
+ }
```

We now can fit this tree, for example to the airquality data; the `print()` method provides us with a first overview on the resulting model

```
> aqt <- mytree(Ozone ~ Solar.R + Wind + Temp, data = airquality)
> aqt

Model formula:
Ozone ~ Solar.R + Wind + Temp

Fitted party:
[1] root
|   [2] Temp <= 87.5
|   |   [3] Solar.R <= 48.95: 11.929 (n = 14, err = 580.9)
|   |   [4] Solar.R > 48.95
|   |   |   [5] Temp <= 77
|   |   |   |   [6] Wind <= 8.84: 29.500 (n = 8, err = 450.0)
|   |   |   |   [7] Wind > 8.84
|   |   |   |   |   [8] Temp <= 76.1: 17.852 (n = 27, err = 1677.4)
|   |   |   |   |   [9] Temp > 76.1: 21.333 (n = 3, err = 0.7)
|   |   |   [10] Temp > 77
|   |   |   |   [11] Wind <= 9.7: 61.154 (n = 26, err = 33555.4)
|   |   |   |   [12] Wind > 9.7: 37.688 (n = 16, err = 1849.4)
|   [13] Temp > 87.5: 90.059 (n = 17, err = 3652.9)

Number of inner nodes:    6
Number of terminal nodes: 7
```

We depict the model graphically using `plot()` (see Figure~2) and compute predictions using

```
> predict(aqt, newdata = airquality[1:10,])

       1        2        3        4        5        6        7 
29.50000 29.50000 17.85185 17.85185 11.92857 17.85185 29.50000 
       8        9       10 
17.85185 11.92857 29.50000 
```

An interesting feature is the ability to extract subsets of trees:

```
> aqt4 <- aqt[4]
> aqt4

Model formula:
Ozone ~ Solar.R + Wind + Temp
```

```
> plot(aqt)
```

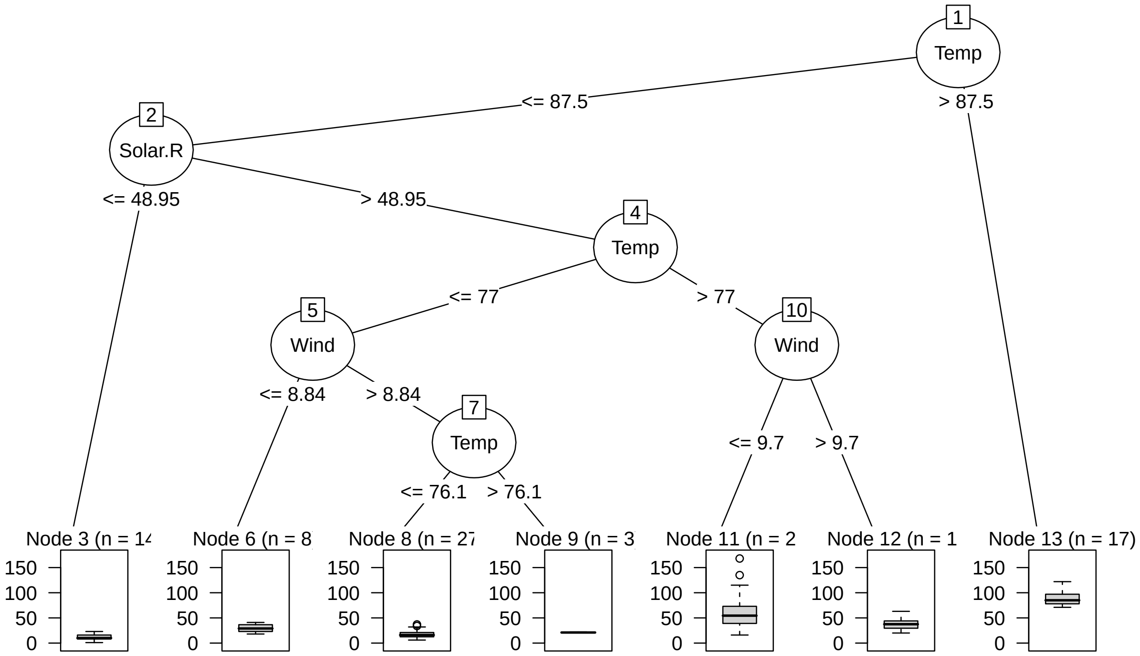

Figure 2: Tree.

```
Fitted party:
[4] root
|   [5] Temp <= 77
|   |   [6] Wind <= 8.84: 29.500 (n = 8, err = 450.0)
|   |   [7] Wind > 8.84
|   |   |   [8] Temp <= 76.1: 17.852 (n = 27, err = 1677.4)
|   |   |   [9] Temp > 76.1: 21.333 (n = 3, err = 0.7)
|   [10] Temp > 77
|   |   [11] Wind <= 9.7: 61.154 (n = 26, err = 33555.4)
|   |   [12] Wind > 9.7: 37.688 (n = 16, err = 1849.4)

Number of inner nodes:    4
Number of terminal nodes: 5
```

which again are objects inheriting from "`party`" and thus can be plotted easily (see Figure˜3). We also might be interested in extracting the *p*-values in the inner nodes in a nicely formatted way:

```
> fun <- function(x) format.pval(info_split(split_node(x))$pvalue,
+   digits = 3, eps = 0.001)
> nid <- nodeids(aqt)
> iid <- nid[!(nid %in% nodeids(aqt, terminal = TRUE))]
> unlist(nodeapply(aqt, ids = iid, FUN = fun))
> plot(aqt4)
```

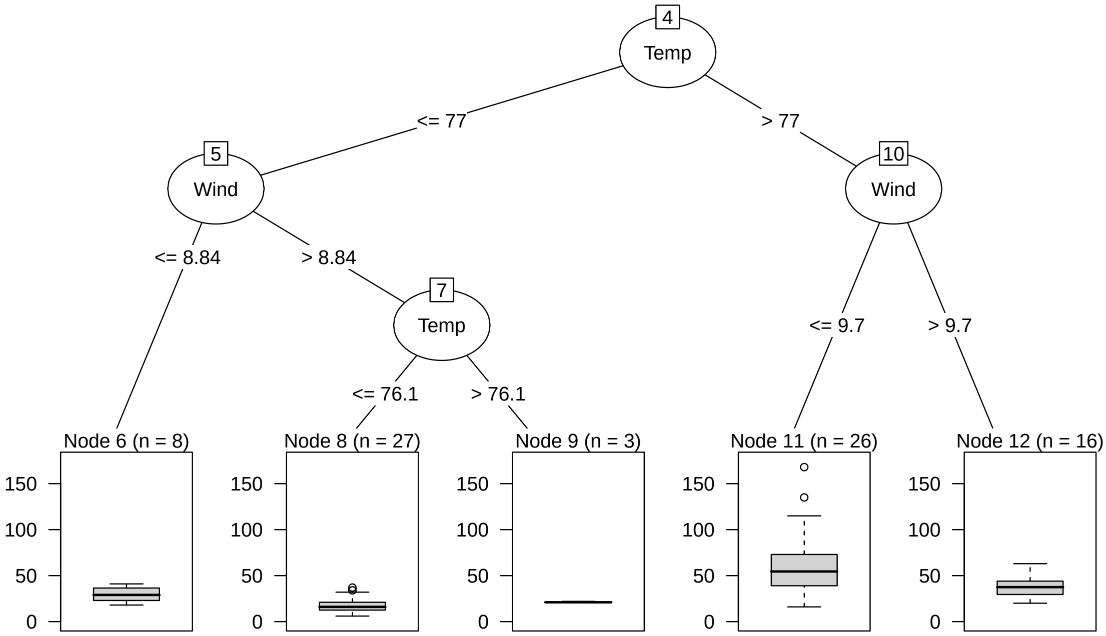

Figure 3: Subtree.

```
         1          2          4         5          7         10
  "<0.001"   "<0.001"   "<0.001" "0.00457"   "0.0362" "0.00462"
```

Affiliation:

Torsten Hothorn
Institut für Statistik
Ludwig-Maximilians-Universität München
Ludwigstr.~33
80539 München, Germany
E-mail: Torsten.Hothorn@R-project.org
URL: http://www.stat.uni-muenchen.de/~hothorn/

Achim Zeileis
Department of Statistics
Universität Innsbruck
Universitätsstr.~15
6020 Innsbruck, Austria
E-mail: Achim.Zeileis@R-project.org
URL: http://eeecon.uibk.ac.at/~zeileis/