rForest: A Basic Implementation of the CART an Random Forest Algorithm

Stefan Klocke (matriculation number: 4836825)

2018-04-23

Contents

1 Introduction **2**

2 Theoretical Framework **2**
2.1 Classification and Regression Trees: CART Algorithm 2
2.2 Bootstrap Aggregating: Bagging Algorithm 3
2.3 Random Forest Algorithm 3

3 Implementation in R **4**
3.1 gini 4
3.2 setSplit 4
3.3 gain 5
3.4 rule 5
3.5 setNode and setLeaf 5
3.6 grow (internal within tree) 7
3.7 traverse.node 10
3.8 tree 10
3.9 importance.tree 11
3.10 parenthesize.table 11
3.11 print.leaf 11
3.12 print.node 11
3.13 print.tree 12
3.14 singlePred (internal within predict.tree) 12
3.15 predict.tree 13
3.16 summary.tree 13
3.17 bootstrap 14
3.18 forest 14
3.19 predict.forest 15
3.20 importance.forest 17
3.21 print.forest 17
3.22 summary.forest 18

4 Discussion **19**

References **19**

1 Introduction

The package `rForest` introduces tree-based methods as tools for analyzing high dimensional classification problems. In particular, the package provides a basic implementation of the classification and regression trees (CART) algorithm (where as of now, the emphasis here is put on categorial response variables, i.e. on classification problems). In combination with bootstrap resampling, the algorithm's functionality will be extended by *Bagging* to support the construction of tree ensembles or *bagged trees*, of which the *Random Forest* algorithm is introduced as a special case.

2 Theoretical Framework

Before turning to the implementation of the three algorithms, it is this section's purpose to discuss the theoretical underpinnings on which this package was built.

2.1 Classification and Regression Trees: CART Algorithm

With their book, Breiman *et al.* (1984) introduced the classification and regression trees algorithm as a new approach to assessing the explanatory relationship between response and predictor variables.

Binary Splitting

Unlike e.g. regression analysis, the CART algorithm's key approach is to stratify some predictor space (i.e. the space spanned by a set of predictors and their respective values) into a number of smaller and more homogeneous regions by employing binary splits (see James *et al.* (2013, p. 303)).

In this sense, binary splitting means that a predictor variable and value will be chosen to partition data into two subsets. The goals is to achieve a binary split that results in the highest possible degree of intra-subset homogeneity (respectively, inter-subset heterogeneity) with respect to the data's response values.

The homogeneity of observations or *node purity* can be assessed using the Gini index, which attains high values for heterogeneous data:

$$gini_m = 1 - \sum_{i=1}^{K} \hat{p}_{mk}^2$$

where $\hat{p}_{mk}$ is the probability (relative frequency) of variable m's class or value k. The goodness of a split is then calculated by the *information gain* as the weighted reduction of the Gini from splitting the dataset. To find the best split, the algorithm calculates the information gain for all values of all predictors and returns the predictor and value with the highest gain as the best possible spplit rule.

Recursive Partitioning

The main rationale behind CART is to repeatedly use binary splits in order to successively obtain smaller subsets that increase in purity. In this manner, a splitting rule is derived for every (sub)set of observations, further splitting the data into two subsets. This process will then be recursively applied to both subsets, ultimately resulting in a sequence of binary splits.

The result of this process, following Breiman *et al.* (1984), is referred to as a classification tree. The starting point of a tree is referred to as the `root`, split points within the tree are called `node`s, while endpoints are `leaf`s. Both, `leaf`s and the `root` are special cases of `node`s.

In extreme cases, this process continues until each leaf contains only one observation (i.e. a perfectly pure subset). This, however, should not be the aim of tree construction, since the resulting classification model

would be highly susceptible to overfitting (see James *et al.* (2013, p. 312)). Therefore some stopping criteria need to be defined in order to constrain the tree growing process at the cost of some impurities in the leafs (see also Kuhn and Johnson (2013, p. 372)).

Greediness

Following James *et al.* (2013, p. 306), a special property of the CART algorithm is that of *greediness*. At every node, the algorithm always chooses the split rule that increases node purity the most at this point in the process, rather than lookin forward aiming at the predictive accuracy of the overall tree. Since this can be problematic if some particularly strong predictors are used, greediness will be addressed with the introduction of *Random Forests*.

2.2 Bootstrap Aggregating: Bagging Algorithm

As pointed out by James *et al.* (2013, p. 315), classification trees can be very non-robust to changes in data. That is, slight changes in the data may result in significant changes in tree structure. Therefore, classification trees may be susceptible to high variance and, therefore low predictive power when built on different data.

Bootstrap Resampling

An approach for tackling this issue was developed by Breiman (1996), who introduced the concept of bootstrap aggregating or *Bagging*. The idea behind this approach is to randomly draw b bootstrap samples *with replacement* from some original training dataset and then to construct one individual classification tree per sample. Tree variance can then be accounted for by deriving an aggregate prediction over all trees.

Out-of-Bag (OOB) Estimation

Since resampling is performed with replacement, some observations will happen to enter a bootstrap sample multiple times, while other observation may not be drawn at all. These observations are referred to as *out-of-bag* (OOB) observations. Since these observations will not be used for tree construction, they are not correlated with the constructed trees and can, therefore, be used for *out-of-bag estimation* (see James *et al.* (2013, p. 317)). Even though this is a key concept to Bagging, it goes beyond the constraints of this project. As for now, this version of `rForest` only supports the use of separate testing datasets.

Tree Aggregation

Now given an *ensemble* of b clasification trees, the simplest approach to obtaining a prediction (as in James *et al.* (2013, p. 317)) is to use a majority vote or *mode* over the predictions of all trees in the ensemble. The *mode* will then be used as an aggregate prediction.

This approach, however, may not always be capable of addressing issues arising from the greediness of the CART algorithm. Under the presence of some particularly strong predictors, greediness may actually cause trees to be correlated, having a similar structure and yielding similar predictions.

2.3 Random Forest Algorithm

The problem with strong predictors is that they will be chosen by the CART algorithm mostly in early stages of the recursive partitioning process (greediness), therefore somewhat mitigating the relative importance of other predictors in early split stages, causing correlation between bagged trees.

As an extension to the Bagging approach, Breiman (2001) introduced a concept for decorrelating bagged trees, referred to as *Random Forests*. The rationale behind this approach is to allow only a random subset of $m \leq p$ predictors to be used for binary splitting - As a rule of thumb, Breiman (2001) suggests $m = \sqrt{p}$. At each stage of the partitioning process, a *new random subset* of predictors will be used. In this manner, it is possible that at some split points, particularly strong predictors may randomly drop out of consideration, in turn promoting the relative importance of those (less strong) predictors that remained in consideration.

3 Implementation in R

This section provids details on the implementation of the introduced concepts in R. The implemented functions are introduced in order of their usage within the three algorithms. The `iris` dataset with its response variable `Species` will be used for demonstration.

```
library(rForest)

# Data import and sample split
data(iris)
data <- iris
set.seed(1337)
rows <- sample.int(nrow(data), size = floor(0.7*nrow(data)), replace = FALSE)
trainData <- data[rows,]
testData <- data[-rows,]
```

3.1 gini

As a starting point, a node purity index is needed to evaluate results from the splitting mechanism introduced soon. The function `gini()` calculates the purity for a given variable (which in our case is the response variable `Species`):

```
purity <- gini(trainData, "Species")
purity
```

```
## [1] 0.6643084
```

3.2 setSplit

In order to support both, numerical and categorial predictor variables, the function `setSplit()` evaluates the class of a split variable `splitVar` before partitioning the data. If `splitVar` is numeric, observations with values smaller than `splitVal` will be assigned to the `left` group, else `right`. For categorial variables, observations with values other than `splitVal` will be assigned `left`.

```
# categorial
groups <- setSplit(trainData, "Species", "setosa")
unique(groups$left$Species)
```

```
## [1] versicolor virginica
## Levels: setosa versicolor virginica
```

```
unique(groups$right$Species)
```

```
## [1] setosa
## Levels: setosa versicolor virginica
```

```
# numeric
groups <- setSplit(trainData, "Sepal.Width", 3.1)
max(groups$left$Sepal.Width)
```

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

```
min(groups$right$Sepal.Width)
```

```
## [1] 3.1
```

3.3 gain

Given some binary split and the pre-split Gini value, the function `gain()` is then used to calculate the *information gain* as the weighted Gini-reduction (or weighted increase in purity) from a binary split. The desirability of a split increases with its information gain.

```
gain(groups$left, groups$right, "Species", purity)
```

```
## [1] 0.1295706
```

3.4 rule

The search for the best possible split is implemented in the function `rule()`, which is passed some `data`, a `response` variable name and a list of variable names used as `predictors`. The variable and value along with the resulting gain are returned in a list:

```
# Use all variables other than "Species" as predictors
response <- "Species"
predictors <- setdiff(colnames(iris), "Species")
bestRule <- rule(trainData, response, predictors)
bestRule
```

```
## $var
## [1] "Petal.Length"
##
## $val
## [1] 3
##
## $type
## [1] "numeric"
##
## $gain
## [1] 0.3183201
```

Given the set of `predictors`, the function loops over all predictor variables. For each predictor, the variable's unique values (or factor levels) will be stored in a vector. For each value, a second loop (within the first one) splits `data` and calculates the resulting information gain. The pair of predictor variable and value with the highest gain is returned as the best possible split rule for `data`. Internally, the list containing the rule is carried through the loops and updated every time a better rule is found.

3.5 setNode and setLeaf

Before being able to finally implement the tree growing process, constructors for the creation of `node` and `leaf` objects are needed. In the recursive growing process, a `node` is set whenever a further binary split of the

current (sub)set is possible. If, instead, no further splits are possible, a `leaf` will be created. Whether further splits are possible is evaluated using stopping criteria, which will be introduced along with implemention of recursive tree growing later.

The function `setNode()` is the constructor for `node` objects:

```
# Create a new split using best possible split rule from above
groups <- setSplit(trainData, bestRule$var, bestRule$val)
node <- setNode(bestRule, groups$left, groups$right, "Species")
str(node)
```

```
## List of 5
##  $ rule :List of 4
##   ..$ var : chr "Petal.Length"
##   ..$ val : num 3
##   ..$ type: chr "numeric"
##   ..$ gain: num 0.318
##  $ depth: num 0
##  $ left :'data.frame':   32 obs. of  5 variables:
##   ..$ Sepal.Length: num [1:32] 5.4 5.3 5 4.9 5.4 5 4.6 5.1 5.2 5.4 ...
##   ..$ Sepal.Width : num [1:32] 3.7 3.7 3.5 3.1 3.4 3.4 3.1 3.8 4.1 3.9 ...
##   ..$ Petal.Length: num [1:32] 1.5 1.5 1.3 1.5 1.7 1.6 1.5 1.9 1.5 1.7 ...
##   ..$ Petal.Width : num [1:32] 0.2 0.2 0.3 0.2 0.2 0.4 0.2 0.4 0.1 0.4 ...
##   ..$ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
##  $ right:'data.frame':   73 obs. of  5 variables:
##   ..$ Sepal.Length: num [1:73] 6.7 5.4 5.6 6.5 6.3 6.4 6 5.8 6.3 7.4 ...
##   ..$ Sepal.Width : num [1:73] 3.1 3 3 2.8 3.4 3.1 3 2.8 2.8 2.8 ...
##   ..$ Petal.Length: num [1:73] 4.7 4.5 4.5 4.6 5.6 5.5 4.8 5.1 5.1 6.1 ...
##   ..$ Petal.Width : num [1:73] 1.5 1.5 1.5 1.5 2.4 1.8 1.8 2.4 1.5 1.9 ...
##   ..$ Species     : Factor w/ 3 levels "setosa","versicolor",..: 2 2 2 2 3 3 3 3 3 3 ...
##  $ names:List of 1
##   ..$ response: chr "Species"
##  - attr(*, "class")= chr "node"
```

Note that in a final tree, `node` objects do not actually contain raw data inside their `left` and `right` elements. Instead, as the recursive tree construction process proceeds, either a `node` or a `leaf` object is created inside `left` and `right`. `leaf` objects however do indeed contain subsets of the original dataset.

To set a `leaf` object, the constructor `setLeaf()` is called. Suppose for the `node` object created above, no further splits (in both, `left` and `right` groups) were possible. Then both elements would be set as `leaf`s:

```
node$left <- setLeaf(node$left, response)
node$right <- setLeaf(node$right, response)
# Look at node's structure again
str(node)
```

```
## List of 5
##  $ rule :List of 4
##   ..$ var : chr "Petal.Length"
##   ..$ val : num 3
##   ..$ type: chr "numeric"
##   ..$ gain: num 0.318
##  $ depth: num 0
##  $ left :List of 5
##   ..$ depth        : num 0
##   ..$ obs          :'data.frame':    32 obs. of  5 variables:
##   .. ..$ Sepal.Length: num [1:32] 5.4 5.3 5 4.9 5.4 5 4.6 5.1 5.2 5.4 ...
```

```
##   .. ..$ Sepal.Width : num [1:32] 3.7 3.7 3.5 3.1 3.4 3.4 3.1 3.8 4.1 3.9 ...
##   .. ..$ Petal.Length: num [1:32] 1.5 1.5 1.3 1.5 1.7 1.6 1.5 1.9 1.5 1.7 ...
##   .. ..$ Petal.Width : num [1:32] 0.2 0.2 0.3 0.2 0.2 0.4 0.2 0.4 0.1 0.4 ...
##   .. ..$ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 ...
##   ..$ prediction   : Factor w/ 3 levels "setosa","versicolor",..: 1
##   ..$ predictionErr: num 0
##   ..$ names        :List of 1
##   .. ..$ response: chr "Species"
##   ..- attr(*, "class")= chr "leaf"
##  $ right:List of 5
##   ..$ depth        : num 0
##   ..$ obs          :'data.frame':    73 obs. of  5 variables:
##   .. ..$ Sepal.Length: num [1:73] 6.7 5.4 5.6 6.5 6.3 6.4 6 5.8 6.3 7.4 ...
##   .. ..$ Sepal.Width : num [1:73] 3.1 3 3 2.8 3.4 3.1 3 2.8 2.8 2.8 ...
##   .. ..$ Petal.Length: num [1:73] 4.7 4.5 4.5 4.6 5.6 5.5 4.8 5.1 5.1 6.1 ...
##   .. ..$ Petal.Width : num [1:73] 1.5 1.5 1.5 1.5 2.4 1.8 1.8 2.4 1.5 1.9 ...
##   .. ..$ Species     : Factor w/ 3 levels "setosa","versicolor",..: 2 2 2 2 3 3 3 3 3 3 ...
##   ..$ prediction   : Factor w/ 3 levels "setosa","versicolor",..: 3
##   ..$ predictionErr: num 0.466
##   ..$ names        :List of 1
##   .. ..$ response: chr "Species"
##   ..- attr(*, "class")= chr "leaf"
##  $ names:List of 1
##   ..$ response: chr "Species"
##  - attr(*, "class")= chr "node"
```

A `leaf` is an endpoint of a tree, carrying a subset of observations from the original dataset. From these observations, the mode (see `?getMode`) of response classes is used as a prediction value. A prediction error is included as the relative number of observations with response values other than the predicted class.

Different to popular implementations of the CART algorithms such as the `rpart` package, `leaf` objects actually contain observations from the original dataset. This, of course, can be extremely inefficient in terms of memory usage, especially for large datasets. This property will be addressed in the final section

3.6 grow (internal within tree)

Now, the core function of the CART implementation can be introduced. Provided a `group` of observations, `grow()` retrieves the best split `rule()` and performs a binary split. Given the binary split result, `grow()` is called recursively on both `left` and `right` groups. The results of both recursive calls will be combined to a `node` object afterwards (*post-order* traversal, see Morris (1979)) . Note that `grow()` is defined locally within the `tree()` constructor.

```
# Function definition within tree()
grow <- function(group, depth = 0) {
  if(nrow(group) < minsplit) {
    setLeaf(group, response, depth)
  }
  # Get best binary split rule
  rule <- rule(group, response, predictors)
  # Do not split, if split does not yield info. gain or maxdepth reached
  if(rule$gain == 0 || depth > maxdepth) {
    setLeaf(group, response, depth)
  } else {
    split <- setSplit(group, rule$var, rule$val)
```

```
    # Do not split, if not too few obs. or if resulting leafs/nodes too small
    if(nrow(split$left) < minbucket || nrow(split$right) < minbucket) {
      setLeaf(group, response, depth)
    } else {
      # Recursively process left/right groups resulting from previous split
      left <- grow(split$left, depth = depth + 1)
      right <- grow(split$right, depth = depth + 1)
      # Combine subtrees into node
      setNode(rule, left, right, response, depth = depth)
    }
  }
}
```

A recursion (not necessarily the entire process) terminates conditional on the following stopping criteria:

1. Current 'group' of observations does not contain enough observations (less than specified in 'minsplit').
2. Next binary split does not yield information gain.
3. Maximum number of levels 'maxdepth' is reached.
4. At least one group from next split does not contain enough observations (less than specified in 'minbucket').

If one of these criteria is met, the current recursion ends with a `leaf` being created. If none of the criteria is met, a `node` will be created and recursion proceeds.

Even though `grow()` is defined locally within `tree()`, a function call outside of `tree()` can be demonstrated if some of `tree()`'s arguments are defined manually on `.GlobalEnv`:

```
# Manually define some option parameters of tree()
minsplit <- 15
minbucket <- 5
# Only grow 2 levels so vignette is not overloaded with output
maxdepth <- 2

# Grow tree, show its structure
irisTree <- list(root = grow(trainData))
str(irisTree)
```

```
## List of 1
##  $ root:List of 5
##   ..$ rule :List of 4
##   .. ..$ var : chr "Petal.Length"
##   .. ..$ val : num 3
##   .. ..$ type: chr "numeric"
##   .. ..$ gain: num 0.318
##   ..$ depth: num 0
##   ..$ left :List of 5
##   .. ..$ depth        : num 1
##   .. ..$ obs          :'data.frame': 32 obs. of  5 variables:
##   .. .. ..$ Sepal.Length: num [1:32] 5.4 5.3 5 4.9 5.4 5 4.6 5.1 5.2 5.4 ...
##   .. .. ..$ Sepal.Width : num [1:32] 3.7 3.7 3.5 3.1 3.4 3.4 3.1 3.8 4.1 3.9 ...
##   .. .. ..$ Petal.Length: num [1:32] 1.5 1.5 1.3 1.5 1.7 1.6 1.5 1.9 1.5 1.7 ...
##   .. .. ..$ Petal.Width : num [1:32] 0.2 0.2 0.3 0.2 0.2 0.4 0.2 0.4 0.1 0.4 ...
##   .. .. ..$ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
##   .. ..$ prediction   : Factor w/ 3 levels "setosa","versicolor",..: 1
##   .. ..$ predictionErr: num 0
```

```
##   .. ..$ names        :List of 1
##   .. .. ..$ response: chr "Species"
##   .. ..- attr(*, "class")= chr "leaf"
##   ..$ right:List of 5
##   .. ..$ rule :List of 4
##   .. .. ..$ var : chr "Petal.Width"
##   .. .. ..$ val : num 1.8
##   .. .. ..$ type: chr "numeric"
##   .. .. ..$ gain: num 0.373
##   .. ..$ depth: num 1
##   .. ..$ left :List of 5
##   .. .. ..$ rule :List of 4
##   .. .. .. ..$ var : chr "Petal.Length"
##   .. .. .. ..$ val : num 5
##   .. .. .. ..$ type: chr "numeric"
##   .. .. .. ..$ gain: num 0.0756
##   .. .. ..$ depth: num 2
##   .. .. ..$ left :List of 5
##   .. .. .. ..$ depth        : num 3
##   .. .. .. ..$ obs          :'data.frame':   32 obs. of  5 variables:
##   .. .. .. .. ..$ Sepal.Length: num [1:32] 6.7 5.4 5.6 6.5 5 4.9 6.3 6.1 6.1 5.5 ...
##   .. .. .. .. ..$ Sepal.Width : num [1:32] 3.1 3 3 2.8 2.3 2.4 2.3 3 2.8 2.4 ...
##   .. .. .. .. ..$ Petal.Length: num [1:32] 4.7 4.5 4.5 4.6 3.3 3.3 4.4 4.6 4.7 3.8 ...
##   .. .. .. .. ..$ Petal.Width : num [1:32] 1.5 1.5 1.5 1.5 1 1 1.3 1.4 1.2 1.1 ...
##   .. .. .. .. ..$ Species     : Factor w/ 3 levels "setosa","versicolor",..: 2 2 2 2 2 2 2 2 2 2 ...
##   .. .. .. ..$ prediction   : Factor w/ 3 levels "setosa","versicolor",..: 2
##   .. .. .. ..$ predictionErr: num 0.0312
##   .. .. .. ..$ names        :List of 1
##   .. .. .. .. ..$ response: chr "Species"
##   .. .. .. ..- attr(*, "class")= chr "leaf"
##   .. .. ..$ right:List of 5
##   .. .. .. ..$ depth        : num 3
##   .. .. .. ..$ obs          :'data.frame':   5 obs. of  5 variables:
##   .. .. .. .. ..$ Sepal.Length: num [1:5] 6.3 7.2 6 6.7 6
##   .. .. .. .. ..$ Sepal.Width : num [1:5] 2.8 3 2.7 3 2.2
##   .. .. .. .. ..$ Petal.Length: num [1:5] 5.1 5.8 5.1 5 5
##   .. .. .. .. ..$ Petal.Width : num [1:5] 1.5 1.6 1.6 1.7 1.5
##   .. .. .. .. ..$ Species     : Factor w/ 3 levels "setosa","versicolor",..: 3 3 2 2 3
##   .. .. .. ..$ prediction   : Factor w/ 3 levels "setosa","versicolor",..: 3
##   .. .. .. ..$ predictionErr: num 0.4
##   .. .. .. ..$ names        :List of 1
##   .. .. .. .. ..$ response: chr "Species"
##   .. .. .. ..- attr(*, "class")= chr "leaf"
##   .. .. ..$ names:List of 1
##   .. .. .. ..$ response: chr "Species"
##   .. .. ..- attr(*, "class")= chr "node"
##   .. ..$ right:List of 5
##   .. .. ..$ depth        : num 2
##   .. .. ..$ obs          :'data.frame':  36 obs. of  5 variables:
##   .. .. .. ..$ Sepal.Length: num [1:36] 6.3 6.4 6 5.8 7.4 7.7 6.5 5.9 6.4 6.4 ...
##   .. .. .. ..$ Sepal.Width : num [1:36] 3.4 3.1 3 2.8 2.8 2.8 3.2 3.2 2.8 3.2 ...
##   .. .. .. ..$ Petal.Length: num [1:36] 5.6 5.5 4.8 5.1 6.1 6.7 5.1 4.8 5.6 5.3 ...
##   .. .. .. ..$ Petal.Width : num [1:36] 2.4 1.8 1.8 2.4 1.9 2 2 1.8 2.2 2.3 ...
##   .. .. .. ..$ Species     : Factor w/ 3 levels "setosa","versicolor",..: 3 3 3 3 3 3 3 2 3 3 ...
```

```
##   .. .. ..$ prediction   : Factor w/ 3 levels "setosa","versicolor",..: 3
##   .. .. ..$ predictionErr: num 0.0278
##   .. .. ..$ names        :List of 1
##   .. .. .. ..$ response: chr "Species"
##   .. .. ..- attr(*, "class")= chr "leaf"
##   .. ..$ names:List of 1
##   .. .. ..$ response: chr "Species"
##   .. ..- attr(*, "class")= chr "node"
##   ..$ names:List of 1
##   .. ..$ response: chr "Species"
##   ..- attr(*, "class")= chr "node"
```

3.7 traverse.node

Provided a complete tree structure, the concept of recursive *pre-order* traversal (see Morris (1979)) can be used to extract e.g. split information from leafs and nodes. This is implemented in `traverse.node()`:

```
# The argument 'ans' is never passed to manual function calls (recursives only)
traverse(irisTree$root)
```

```
##            var val    type       gain
## 1 Petal.Length 3.0 numeric 0.31832013
## 2  Petal.Width 1.8 numeric 0.37327654
## 3 Petal.Length 5.0 numeric 0.07561176
```

Each recursion of `traverse.node()` is passed a data.frame `ans` of previously collected split rules such that at the end of a recursion, a data.frame of split rules for all nodes is returned (or passed to the next recursive call). The function will later be used to generate variable importance rankings using `importance.tree()`.

3.8 tree

While all tools for growing a tree are already in place, the `tree()` constructor can now be implemented. The function is passed a `formula` to describe the explainatory relationship between `data`'s variables. A period can be specified in order to use all predictors.

`m` specifies the size of a random predictor subset as discussed in section 2. The functionality for random predictor subsets is already included here, so the function `forest()` (introduced later) can make use of `tree()`. The classical CART approach, however, uses the full set of predictors `m = p`.

For the sake of comparability between packages, the argument names `minsplit`, `minbucket` and `maxdepth` (as well as their defaults) are adapted from the `rpart` package. They are used for control structures and as stopping criteria (see above) to control for overfitting.

Since `grow()` is defined within `tree()`, `grow()` can access the above described option parameters from `tree()`'s environment (so they do not have to be passed to `grow()` again). Finally, the function assembles a `tree` object as a list contraining the tree, information on variables and their importances (using `importance.tree()`) as well as on control parameters (see `?tree` for details).

```
# Generate a fitted tree model using all 4 predictors
fit <- tree(Species ~ .,
            data = trainData,
            m = 4,
            minsplit = 15,
            minbucket = 5,
            maxdepth = 10)
```

3.9 importance.tree

The variable importance ranking included in a `tree` object is generated by the `importance.tree()` function. It uses `traverse.node()` to extract split rules from all nodes and then aggregates the data.frame (of split rules) returned. Similar to James *et al.* (2013, p. 319), aggregation is performed by calculating the mean information gain relative to the predictor with the highest mean information gain.

```
importance(fit, predictors)
```

```
##            var relMeanGain
## 1  Petal.Width   1.0000000
## 2 Petal.Length   0.5276676
## 3 Sepal.Length          NA
## 4  Sepal.Width          NA
```

A vector with variable names as `predictors` is passed: If some predictors were not used for splitting, these predictors will be added to the data.frame with `relMeanGain` being set `NA`.

3.10 parenthesize.table

Auxiliary function for printing (see `?parenthesize.table`). Omitted to save space.

3.11 print.leaf

Given a complete tree structure, we are now interested in visualizing it. Starting from bottom to top, leafs are printed using the generic `print.leaf()`:

```
# root's left branch leads to a leaf
class(fit$root$left)
```

```
## [1] "leaf"
```

```
fit$root$left
```

```
## Classes:(32\0\0) Prediction: setosa (Error: 0%)
```

3.12 print.node

Printing `node` objects is more complicated, since it requires printing all contained subtrees down to the `leaf`s. For this purpose, pre-order traversal is used again, since left child `node`s and left `leaf`s are needed to be processed first. This recursive traversal is implemented in `print.node()`, which can be used to print either entire trees or only subtrees. If a `leaf` is visited, the current recursion stops and `print.leaf()` is called. Otherwise the function is recursively called first for the `left` and then for the `right` child:

```
# root's right branch leads to another node
class(fit$root$right)
```

```
## [1] "node"
```

```
fit$root$right
```

```
## --> Left: Petal.Width < 1.8
##     --> Left: Petal.Length < 5
##         Classes:(0\31\1) Prediction: versicolor (Error: 3.12%)
##     --> Right: Petal.Length >= 5
##         Classes:(0\2\3) Prediction: virginica (Error: 40%)
```

```
## --> Right: Petal.Width >= 1.8
##     Classes:(0\1\35) Prediction: virginica (Error: 2.78%)
```

3.13 print.tree

Since the printing entire trees is already accomplished by `print.node()`, this function will only be wrapped by `print.tree()`, adding some additional information on the root's response class frequencies:

```
fit
```

```
## Root frequencies: (32\34\39) of classes (setosa\versicolor\virginica)
## --> Left: Petal.Length < 3
##     Classes:(32\0\0) Prediction: setosa (Error: 0%)
## --> Right: Petal.Length >= 3
##     --> Left: Petal.Width < 1.8
##         --> Left: Petal.Length < 5
##             Classes:(0\31\1) Prediction: versicolor (Error: 3.12%)
##         --> Right: Petal.Length >= 5
##             Classes:(0\2\3) Prediction: virginica (Error: 40%)
##     --> Right: Petal.Width >= 1.8
##         Classes:(0\1\35) Prediction: virginica (Error: 2.78%)
```

3.14 singlePred (internal within predict.tree)

Before being able to predict an entire dataset, a function needs to be implemented that handles the prediction of a single observation. The recursive function `singlePred()` defined within the generic `predict.tree()` passes an observation down a tree structure, recursively calling the function when a node is visited. Recursion stops once a leaf is reached, returning the `leaf`'s `prediction` element.

```
# Recursive tree walk to generate single prediction
singlePred <- function(object, data) {
  # Stop recursion if leaf is reached
  if(class(object) == "leaf") {
    object$prediction
  } else {
    splitVar <- object$rule$var
    splitVal <- object$rule$val
    # Construct dynamic relation string for numeric/categorial split cases
    if(object$rule$type == "numeric") {
      rel <- paste(data[[splitVar]],
                   "<",
                   splitVal, sep = "")
    } else {
      rel <- paste("\"", data[[splitVar]], "\"",
                   "!=",
                   "\"", splitVal, "\"", sep = "")
    }
    # Turn the character 'rel' into an expression and evaluate it: Return Bool
    isLeft <- eval(parse(text = rel))
    # Recursively walk through tree until leaf is reached
    if(isLeft) {
      singlePred(object$left, data)
    } else {
      singlePred(object$right, data)
```

```
    }
  }
}
```

The below chunk shows the classifcation of some observation from `testData`.

```
singlePred(fit$root, testData[42,])
```

```
## [1] virginica
## Levels: setosa versicolor virginica
```

3.15 predict.tree

The functionality of `singlePred()` can then be extended to support entire datasets by looping over it, calling `singlePred()` for every observation and finally returning an atomic/factor vector of predictions. The generic `predict.tree()` does exactly this:

```
predict(fit, testData)
```

```
##  [1] setosa     setosa     setosa     setosa     setosa     setosa
##  [7] setosa     setosa     setosa     setosa     setosa     setosa
## [13] setosa     setosa     setosa     setosa     setosa     setosa
## [19] versicolor versicolor versicolor versicolor versicolor versicolor
## [25] versicolor versicolor versicolor versicolor versicolor versicolor
## [31] versicolor versicolor versicolor versicolor virginica  virginica
## [37] virginica  virginica  virginica  virginica  virginica  virginica
## [43] virginica  virginica  virginica
## Levels: setosa versicolor virginica
```

3.16 summary.tree

Provided functionaly for prediction, the key properties of a `tree` object can be summarized by `summary.tree()`. This function has an optional argument to pass a `test` dataset, printing additional prediction summaries:

```
summary(fit, testData)
```

```
## Call:
## tree(formula = Species ~ ., data = trainData, m = 4, minsplit = 15,
##     minbucket = 5, maxdepth = 10)
##
## Number of observations: n = 105
## Root response class distribution:
##
##     setosa versicolor  virginica
##         32         34         39
##
## Random predictor subset: FALSE
##   Predictors (model spec.): p = 4
## [1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"
##
## Nodes and split rules:
##            var val    type       gain
## 1 Petal.Length 3.0 numeric 0.31832013
## 2  Petal.Width 1.8 numeric 0.37327654
```

```
## 3 Petal.Length 5.0 numeric 0.07561176
##
## Predictor variable importance:
##            var relMeanGain
## 1  Petal.Width   1.0000000
## 2 Petal.Length   0.5276676
## 3 Sepal.Length          NA
## 4  Sepal.Width          NA
##
## Prediction confusion matrix:
##             obs
## pred         setosa versicolor virginica
##   setosa         18          0         0
##   versicolor      0         16         0
##   virginica       0          0        11
##
## Misclassification error rate: 0%
```

3.17 bootstrap

Before being able to construct forests, a function for generating multiple training samples is needed. The function `bootstrap()` implements simple functionality for random sampling *with replacement*. The below code chunk demonstrates the creation of 5 bootstrap samples:

```
set.seed(1337)
for(i in 1:5) {
  bs <- bootstrap(trainData)
  tbl <- table(bs$Species)
  if(i == 1) cat("Class distribution of",parenthesize(tbl, type = "names"),"\n")
  cat(parenthesize(tbl, type = "counts"), "\n", sep = "")
}
```

```
## Class distribution of (setosa\versicolor\virginica)
## (34\25\46)
## (39\29\37)
## (37\32\36)
## (37\33\35)
## (36\32\37)
```

3.18 forest

Being able to generate multipe training samples, the constructor `forest()` extends the function `tree()` to the creation of *bagged tree ensembles* or *Random Forests* (for $m < p$). For a specified number of `b` bootstrap samples, the function uses a loop to iteratively resample `data`. Subsequent to resampling, a `tree` is generated using the current bootstrap sample. The trees generated in the loop are stored in the list element `ensemble`.

```
# Create Random Forest with m = 3 < 4 = p
set.seed(1337)
start <- Sys.time()
forestFit <- forest(Species ~ .,
                    data = trainData,
                    m = 3,
                    b = 500,
                    minsplit = 15,
```

```
                    minbucket = 5,
                    maxdepth = 10,
                    bsinclude = FALSE)
end <- Sys.time()
end - start
```

```
## Time difference of 38.11755 secs
```

In order to show the effect of bootstrap resampling on the structural variation of trees, the below chunk prints some trees included in the `ensemble`:

```
forestFit$ensemble[[1]]
```

```
## Root frequencies: (34\25\46) of classes (setosa\versicolor\virginica)
## --> Left: Petal.Length < 4.9
##     --> Left: Petal.Width < 1
##         Classes:(34\0\0) Prediction: setosa (Error: 0%)
##     --> Right: Petal.Width >= 1
##         Classes:(0\25\1) Prediction: versicolor (Error: 3.85%)
## --> Right: Petal.Length >= 4.9
##     Classes:(0\0\45) Prediction: virginica (Error: 0%)
```

```
forestFit$ensemble[[2]]
```

```
## Root frequencies: (37\30\38) of classes (setosa\versicolor\virginica)
## --> Left: Petal.Length < 3
##     Classes:(37\0\0) Prediction: setosa (Error: 0%)
## --> Right: Petal.Length >= 3
##     --> Left: Petal.Length < 4.8
##         Classes:(0\26\1) Prediction: versicolor (Error: 3.7%)
##     --> Right: Petal.Length >= 4.8
##         --> Left: Petal.Length < 5.1
##             Classes:(0\3\5) Prediction: virginica (Error: 37.5%)
##         --> Right: Petal.Length >= 5.1
##             Classes:(0\1\32) Prediction: virginica (Error: 3.03%)
```

Obsiously, the changes in the data caused by resampling had some significant effects on the tree structure.

3.19 predict.forest

To assess the predictive performance of the generated `forest` object, the function `predict.forest()` provides two types of predictions. Depending on the `type` argument passed, the function can either return an atomic/factor prediction vector containing the modal prediction over all trees (`type = aggr`) . For `type = prob`, the relative prediction probabilities for each response class are returned as a matrix. Internally, the function loops over all trees inside the `ensemble` and stores their predictions inside a data.frame. The data.frame is then aggregated according to the `type` argument specified.

```
# Aggregate/modal prediction
predMode <- predict(forestFit, testData, type = "aggr")
predMode
```

```
##  [1] setosa     setosa     setosa     setosa     setosa     setosa
##  [7] setosa     setosa     setosa     setosa     setosa     setosa
## [13] setosa     setosa     setosa     setosa     setosa     setosa
## [19] versicolor versicolor versicolor versicolor versicolor versicolor
## [25] versicolor versicolor versicolor versicolor versicolor versicolor
```

```
## [31] versicolor versicolor versicolor versicolor virginica  virginica
## [37] virginica  virginica  virginica  virginica  virginica  virginica
## [43] virginica  virginica  virginica
## Levels: setosa versicolor virginica
```

```
# Prediction probabilites
predProbs <- predict(forestFit, testData, type = "prob")
predProbs
```

```
##       setosa versicolor virginica
##  [1,]  1.000      0.000     0.000
##  [2,]  0.998      0.002     0.000
##  [3,]  1.000      0.000     0.000
##  [4,]  1.000      0.000     0.000
##  [5,]  0.998      0.002     0.000
##  [6,]  1.000      0.000     0.000
##  [7,]  1.000      0.000     0.000
##  [8,]  1.000      0.000     0.000
##  [9,]  1.000      0.000     0.000
## [10,]  0.998      0.002     0.000
## [11,]  1.000      0.000     0.000
## [12,]  0.998      0.002     0.000
## [13,]  1.000      0.000     0.000
## [14,]  1.000      0.000     0.000
## [15,]  1.000      0.000     0.000
## [16,]  0.998      0.002     0.000
## [17,]  0.998      0.002     0.000
## [18,]  1.000      0.000     0.000
## [19,]  0.000      0.974     0.026
## [20,]  0.000      0.538     0.462
## [21,]  0.000      0.998     0.002
## [22,]  0.000      0.916     0.084
## [23,]  0.000      0.980     0.020
## [24,]  0.000      1.000     0.000
## [25,]  0.000      1.000     0.000
## [26,]  0.000      1.000     0.000
## [27,]  0.000      0.948     0.052
## [28,]  0.000      0.502     0.498
## [29,]  0.000      1.000     0.000
## [30,]  0.000      1.000     0.000
## [31,]  0.000      1.000     0.000
## [32,]  0.000      1.000     0.000
## [33,]  0.000      1.000     0.000
## [34,]  0.000      1.000     0.000
## [35,]  0.000      0.020     0.980
## [36,]  0.000      0.002     0.998
## [37,]  0.000      0.000     1.000
## [38,]  0.000      0.004     0.996
## [39,]  0.000      0.000     1.000
## [40,]  0.000      0.044     0.956
## [41,]  0.000      0.490     0.510
## [42,]  0.000      0.000     1.000
## [43,]  0.000      0.000     1.000
## [44,]  0.000      0.020     0.980
## [45,]  0.000      0.022     0.978
```

While the first prediction type is easy to interpret, prediction probabilites should never be neglected as they provide information on a prediction's reliability. Take e.g. the following two observations:

```
# High variance prediction
predProbs[20,]
```

```
##     setosa versicolor  virginica
##      0.000      0.538      0.462
```

```
# Low variance prediction
predProbs[23,]
```

```
##     setosa versicolor  virginica
##       0.00       0.98       0.02
```

3.20 importance.forest

Almost analogous to `importance.tree()`, variable importance rankings can be generated for tree ensembles using `importance.forest()`. In a first step, importance rankings for all trees are gathered inside a single data.frame using a loop. Then, again as in `importance.tree()`, all rows of this data.frame will be aggregated as relative mean gain. The below chunk compares importance rankings between the single CART tree `fit` and the random forest `forestFit`:

```
# Single Tree
importance(fit, predictors)
```

```
##            var relMeanGain
## 1  Petal.Width   1.0000000
## 2 Petal.Length   0.5276676
## 3 Sepal.Length          NA
## 4  Sepal.Width          NA
```

```
# Random Forest
importance(forestFit, predictors)
```

```
##            var relMeanGain
## 1 Petal.Length   1.0000000
## 2  Petal.Width   0.9920090
## 3  Sepal.Width   0.4946496
## 4 Sepal.Length   0.2565853
```

As apparent, relative variable importance has changed dramatically for all predictors. For a single tree, `Sepal.Length` and `Sepal.Width` were not used at all and `Petal.Width` was roughly 50% as important as `Petal.Length`. With `b = 500` trees, instead, the latter two variables lie almost equal in importance, while `Sepal.Width` and `Sepal.Length` attain relative importance of about 50% and 25%, respectively.

3.21 print.forest

After creating a `forest` object, some basic information can be displayed using its generic `print.forest()` function. Unlike `print.tree()`, only textual information will be displayed, since printing the structures of every tree in the ensemble will be rather confusing and hardly interpretable:

```
print(forestFit)
```

```
## Call:
## forest(formula = Species ~ ., data = trainData, m = 3, b = 500,
##     minsplit = 15, minbucket = 5, maxdepth = 10, bsinclude = FALSE)
```

```
##
## Type of forest: Random Forest
##
## Number of observations: n = 105
## Number of bootstrap samples and bagged trees: b = 500
##
## Predictors (model spec.): 4
## [1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"
##
## Predictor variable importance:
##            var relMeanGain
## 1 Petal.Length   1.0000000
## 2  Petal.Width   0.9920090
## 3  Sepal.Width   0.4946496
## 4 Sepal.Length   0.2565853
```

3.22 summary.forest

To provide a more detailed overview of the previously presented contents of a `forest` object, `summary.forest()` concludes the model properties similar summary function for `tree`. Again, prediction is conducted optionally:

```
summary(forestFit, testData)
```

```
## Call:
## forest(formula = Species ~ ., data = trainData, m = 3, b = 500,
##     minsplit = 15, minbucket = 5, maxdepth = 10, bsinclude = FALSE)
##
## Number of observations: n = 105
## Training data response class distribution:
##
##     setosa versicolor  virginica
##         32         34         39
##
## Mean relative bootstrap response class distribution:
##
##     setosa versicolor  virginica
##    0.31812    0.34170    0.39018
##
## Random predictor subset: TRUE
##   Predictors (model spec.): 4
## [1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"
##
## Predictor variable importance:
##            var relMeanGain
## 1 Petal.Length   1.0000000
## 2  Petal.Width   0.9920090
## 3  Sepal.Width   0.4946496
## 4 Sepal.Length   0.2565853
##
## Prediction confusion matrix:
##             obs
## pred         setosa versicolor virginica
##   setosa         18          0         0
```

```
##   versicolor      0         16         0
##   virginica       0          0        11
##
## Misclassification error rate: 0%
```

4 Discussion

Even for small datasets with a limited number of predictors, the use of Bagging or Random Forests seems to already be capable of changing conclusions drawn on the importance of some predictors.

For datasets large in both, number of observations and predictors, however, the presented implementation of the three algorithms may be expected to be rather inefficient. For instance, the storage of data inside `tree`/`forest` objects is to memory intensive for large datasets, especially when `forest()` is used with a high number of bootstraps. A solution to this would be to create a "dictionary" storing observation-rownames from the original dataset and leaf-IDs to which the observations will be assigned during tree construction.

In terms of extended functionaliy, possible package extensions could be regression trees, additional purity indices like e.g. cross-entropy, functions for plotting trees, ROC curves (predictive performance measurement) and added support for OOB estimation.

References

Breiman, L., 1996. Bagging Predictors. *Machine Learning*, 24 (2), 123–140.

Breiman, L., 2001. Random Forests. *Machine Learning*, 45 (1), 5–32.

Breiman, L., Friedman, J., Stone, C.J., and Olshen, R.A., 1984. *Classification and Regression Trees*. 1st ed. Taylor & Francis.

James, G., Witten, D., Hastie, T., and Tibshirani, R., 2013. *An Introduction to Statistical Learning with Applications in R*. 1st ed. New York, NY: Springer Science+Business Media.

Kuhn, M. and Johnson, K., 2013. *Applied Predictive Modeling*. 5th ed. New York, NY: Springer Science+Business Media.

Morris, J.M., 1979. Traversing Binary Trees Simply and Cheaply. *Information Processing Letters*, 9 (5).