Simulation

To illustrate, we generate a fake dataset according to the “Friedman 1” model (Friedman 1991).

```r
f <- function(x)
  10 * sin(pi * x[,1] * x[,2]) + 20 * (x[,3] - 0.5)^2 +
  10 * x[,4] + 5 * x[,5]
```

```r
set.seed(99)
sigma <- 1.0
n <- 100
x <- matrix(runif(n * 10), n, 10)
y <- rnorm(n, f(x), sigma)
data <- data.frame(x, y)
```

Model Fitting

In order to interrogate the trees, they must be saved when the model is fit. This is accomplished by setting:

- For `bart`: `keeptrees = TRUE`
- For `bart2`: `keepTrees = TRUE`
- For a custom `dbartsSampler`, `control = dbartsControl(keepTrees = TRUE)`

In the context of our fake data, with small sample numbers for illustrative purposes, a model can be fit thusly:

```r
library(dbarts, quietly = TRUE)
bartFit <- bart(
  y ~ ., data,
  ndpost = 4, # number of posterior samples
  nskip = 1000, # number of "warmup" samples to discard
  nchain = 2, # number of independent, parallel chains
  nthread = 1, # units of parallel execution
  ntree = 3, # number of trees per chain
  seed = 2, # chosen to generate a deep tree
  keeptrees = TRUE,
  verbose = FALSE)
```
Extracting Trees

The extract function accepts as a type the value "trees". If present, the arguments chainNums, sampleNums, and/or treeNums can be used to extract only a subset of trees.

trees <- extract(bartFit, "trees")

Flattened Trees

The trees data frame corresponds to a depth-first, left-hand-side tree traversal.

print(head(trees, n = 10))

<table>
<thead>
<tr>
<th>#</th>
<th>chain</th>
<th>sample</th>
<th>tree</th>
<th>n</th>
<th>var</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>100</td>
<td>2</td>
<td>0.22796229</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>18</td>
<td>1</td>
<td>0.28581841</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>7</td>
<td>-1</td>
<td>-0.02959142</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>11</td>
<td>-1</td>
<td>-0.20336110</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>82</td>
<td>5</td>
<td>0.10434842</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>-1</td>
<td>-0.02468813</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>74</td>
<td>2</td>
<td>0.77999259</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>56</td>
<td>-1</td>
<td>0.09481149</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>18</td>
<td>4</td>
<td>0.76320697</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>9</td>
<td>-1</td>
<td>0.02619319</td>
</tr>
</tbody>
</table>

The columns refer to:

- chain, sample, tree - index variables
- n - number of observations in node
- var - either the index of the variable used for splitting or -1 if the node is a leaf
- value - either the value such that observations less than or equal to it are sent down the left path of the tree or the predicted value for a leaf node

The mapping between the values of var and the variable names can be looked up in the internal copy of the data that the sampler stores. This can be found in a fitted model as the element fit$data@x, as seen below.

Tree Traversal

A useful technique for processing trees is recursion. By having the function return the number of nodes it has processed, it is possible to advance from the left-hand-side to the right by skipping ahead the appropriate number of rows. For example:

```
# Turns a flatted tree data frame into a list of lists, or a "natural" tree structure.
rebuildTree <- function(tree, object) {
  # Define a worker function that will be recursively called on every node.
  rebuildTreeRecurse <- function(tree) {
    node <- list(
      value = tree$value[1],
      n   = tree$n[1]
    )
    # Check node if is a leaf, and if so return early.
    if (tree$var[1] == -1) {
      node$n_nodes <- 1
      return(node)
    }
    node$var <- variableNames[tree$var[1]]
    # Process node and return its children.
    node$left <- rebuildTreeRecurse(tree$left)
    node$right <- rebuildTreeRecurse(tree$right)
    node
  }
  # ...
# By removing the current row, we can recurse down the left branch.
headOfLeftBranch <- tree[-1,]
left <- rebuildTreeRecurse(headOfLeftBranch)
n_nodes.left <- left$n_nodes
left$n_nodes <- NULL
node$left <- left

# The right branch is obtained by advancing past the left nodes.
headOfRightBranch <- tree[seq.int(2 + n_nodes.left, nrow(tree)),]
right <- rebuildTreeRecurse(headOfRightBranch)
n_nodes.right <- right$n_nodes
right$n_nodes <- NULL
node$right <- right

node$n_nodes <- 1L + n_nodes.left + n_nodes.right

return(node)

variableNames <- colnames(object$fit$data@x)

result <- rebuildTreeRecurse(tree)
result$n_nodes <- NULL
return(result)

treeOfInterest <- subset(trees, chain == 1 & sample == 3 & tree == 1)
print(rebuildTree(treeOfInterest, bartFit))

## $value
## [1] 0.2279623

## $n
## [1] 100

## $var
## [1] "X2"

## $left
## $left$value
## [1] 0.2858184

## $left$n
## [1] 18

## $left$var
## [1] "X1"

## $left$left
## $left$left$value
## [1] -0.03007205

## $left$left$n
## [1] 7
## $left$right
## $left$right$value
## [1] 0.7964085
## $left$right$n
## [1] 11
## $left$right$var
## [1] "X10"
## $left$right$left
## $left$right$left$value
## [1] -0.2228073
## $left$right$left$n
## [1] 9
## $left$right$right
## $left$right$right$value
## [1] -0.2207859
## $left$right$right$n
## [1] 2
## $right$
## $right$value
## [1] 0.1043484
## $right$n
## [1] 82
## $right$var
## [1] "X5"
## $right$left
## $right$left$value
## [1] -0.0003996113
## $right$left$n
## [1] 8
## $right$right
## $right$right$value
## [1] 0.7799926
## $right$right$n
## [1] 74
Using a by statement, it is possible to “rebuild” all trees at once:

```r
allTrees <- by(
  data = trees,
  INDICES = trees[,c("chain", "sample", "tree")],
  FUN = rebuildTree,
  object = bartFit)
```

# One way to index the result of this:
# allTrees[chain = "1", sample = "2", tree = "3"]

Plotting Trees

dbartsSampler objects have a plotTree method that can be used to visualize single trees:
The following function traverses a flattened tree, splits observations while going the branches, and populates a vector giving the predicted value of that tree on input data. It requires a data in the same format as the fitted bart model so that it can evaluate the splits.

```r
getPredictionsForTree <- function(tree, x) {
  predictions <- rep(NA_real_, nrow(x))
  getPredictionsForTreeRecursive <- function(tree, indices) {
    if (tree$var[1] == -1) {
      # Assigns in the calling environment by using "<<-
      predictions[indices] <<- tree$value[1]
      return(1)
    }
    goesLeft <- x[indices, tree$var[1]] <= tree$value[1]
    headOfLeftBranch <- tree[-1,]
    n_nodes.left <- getPredictionsForTreeRecursive(
      headOfLeftBranch, indices[goesLeft])
    headOfRightBranch <- tree[seq.int(2 + n_nodes.left, nrow(tree)),]
    n_nodes.right <- getPredictionsForTreeRecursive(
      headOfRightBranch, indices[!goesLeft])
    return(1 + n_nodes.left + n_nodes.right)
  }

  getPredictionsForTreeRecursive(tree, seq_len(nrow(x))[-1])
}
```
getPredictionsForTreeRecursive(tree, seq_len(nrow(x)))

return(predictions)
}

getPredictionsForTree(treeOfInterest, bartFit$fit$data@x[1:5,])

## [1] -0.2228073037  0.2617128852  0.1331738132 -0.2228073037 -0.0003996113

A by statement can be used to obtain all predictions for all trees.

Advanced Traversal

The following function can be used to map an arbitrary function over a tree.

mapOverNodes <- function(tree, f, ...) {
  mapOverNodesRecurse <- function(tree, depth, f, ...) {
    node <- list(
      value = tree$value[1],
      n = tree$n[1],
      depth = depth
    )
    if (tree$var[1] == -1) {
      node$n_nodes <- 1
      node$f.x <- f(node, ...)
      return(node)
    }
    node$var <- tree$var[1]
    node$f.x <- f(node, ...)
    headOfLeftBranch <- tree[-1,]
    left <- mapOverNodesRecurse(headOfLeftBranch, depth + 1, f, ...)
    n_nodes.left <- left$n_nodes
    left$n_nodes <- NULL
    node$left <- left
    n_nodes.left
    headOfRightBranch <- tree[seq.int(2 + n_nodes.left, nrow(tree)),]
    right <- mapOverNodesRecurse(headOfRightBranch, depth + 1, f, ...)
    n_nodes.right <- right$n_nodes
    right$n_nodes <- NULL
    node$right <- right
    n_nodes.right
    node$n_nodes <- 1 + n_nodes.left + n_nodes.right
    return(node)
  }
  result <- mapOverNodesRecurse(tree, 1, f, ...)
  result$n_nodes <- NULL
  return(result)
}

As an example of its usage, the following function aggregates all ancestor/descendant relationships in a tree.

In a data object - here an R environment - it keeps track of the current state of traversal. This includes the variables that have been used for splits above the current node and also includes the current node’s depth, which is used to detect backtracking.
observeInteractions <- function(node, ...) {
  if (is.null(node$var)) return(NULL)

  interactionData <- list(...)$interactionData
  # Make the current node visible inside the environment.
  interactionData$node <- node
  with(interactionData, {
    if (node$depth <= currentDepth) {
      # If true, we have backtracked to go down the right branch, so we
      # remove the variables from the left branch.
      currentVariables <- currentVariables[seq_len(node$depth - 1)]
    }
    if (length(interactionData$currentVariables) > 0) {
      # This is a brute-force way of updating the following indices,
      # relying on the column-major storage order that R uses:
      # hasInteraction[currentVariables,,drop = FALSE][,node$var]
      updateIndices <- currentVariables +
          (node$var - 1) * nrow(hasInteraction)
      hasInteraction[updateIndices] <- TRUE
    }
    currentVariables <- c(currentVariables, node$var)
    currentDepth <- node$depth
  })
  rm("node", envir = interactionData)

  # Since the function is used for its side effects, there isn't a return
  # value.
  return(NULL)
}

numVariables <- ncol(bartFit$fit$data@x)
variableNames <- colnames(bartFit$fit$data@x)

# Define this as an environment as they are mutable
interactionData <- list2env(list(
  currentDepth = 0,
  currentVariables = integer(),
  hasInteraction = matrix(
    data = FALSE,
    ncol = numVariables, nrow = numVariables,
    dimnames = list(ancestor = variableNames, descendant = variableNames)
  )
))

invisible(mapOverNodes(
  treeOfInterest,
  observeInteractions,
  interactionData = interactionData
))

After execution, the boolean matrix in the interaction data environment will contain all ancestor/descendant relationships in this tree.
print(interactionData$hasInteraction)

##
## descendant
## ancestor  X1  X2  X3  X4  X5  X6  X7  X8  X9  X10
## X1  FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE
## X2  TRUE  TRUE FALSE  TRUE  TRUE FALSE FALSE FALSE FALSE TRUE
## X3  FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## X4  FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## X5  FALSE  TRUE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
## X6  FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## X7  FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## X8  FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## X9  FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## X10 FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE

Finally, the entire process can be wrapped in a `by` statement and the results aggregated across all trees in order to count the number of times variables have specific relationships.

**References**