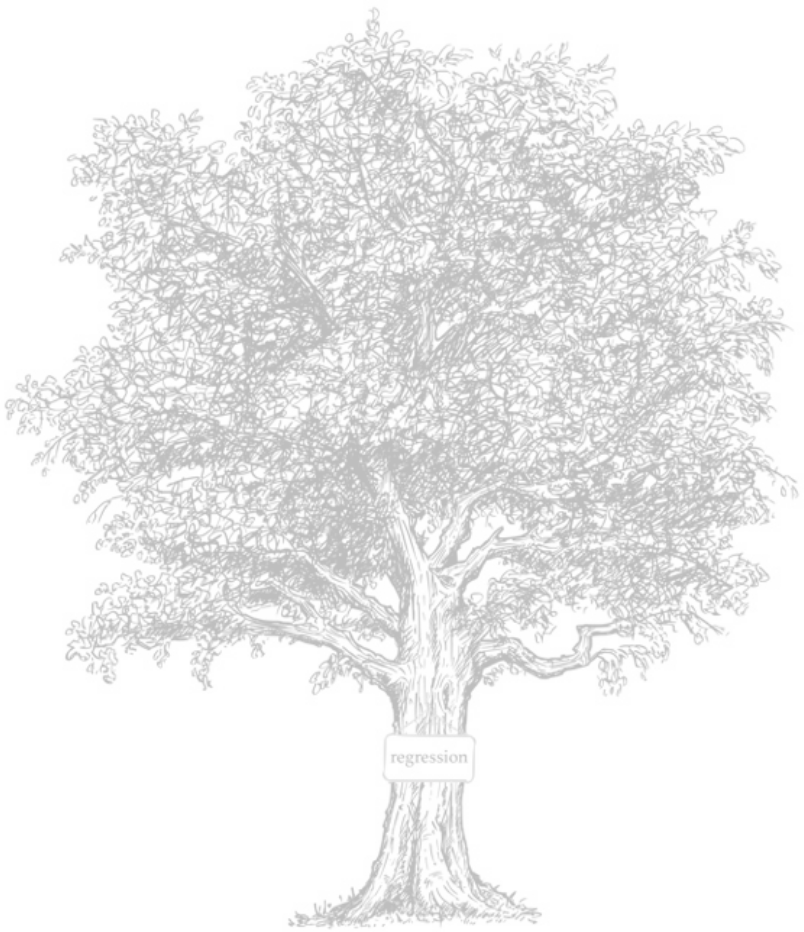
STATWORX *Blog*

Data Science, Machine Learning & AI.



Coding Regression trees in 150 lines of R code

ANDRÉ BLEIER / 9. NOVEMBER 2018 / BLOG, DATA SCIENCE

Motivation

There are dozens of machine learning algorithms out there. It is impossible to learn all their mechanics, however, many algorithms sprout from the most established algorithms, e.g. ordinary least squares, gradient boosting, support vector machines, tree-based algorithms and neural networks. At STATWORX we

STATWORX Blog

Willkommen auf unserem Blog. Hier dreht sich alles um Data Science, Statistik und Machine Learning. In regelmäßigen Abständen veröffentlichen wir Artikel, Posts und andere interessante Beiträge. Stay tuned!

Contributor to R-Bloggers

Tag Cloud

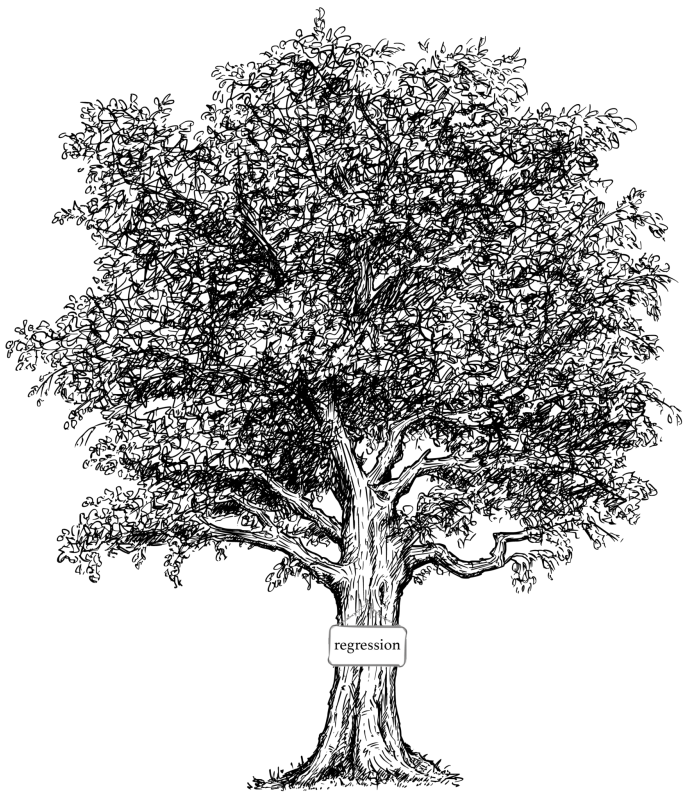
ACADEMY AI API BUSINESS CODING DASHBOARD DATA ENGINEERING DATA SCIENCE DATA STRATEGY DATA UNIVERSITY DEBUGGING DEEP LEARNING DESKRIPTIVE DOCKER ELASTIZITÄT EXPLAINABLE AI FEHLERBEHANDLUNG FRAGEBOGEN FUN GGPLOT HELFRLEIN HYPOTHESEN KERAS KORRELATION MACHINE LEARNING

discuss algorithms daily to evaluate their usefulness for a specific project. In any case, understanding these core algorithms is key to most machine learning algorithms in the literature.

While I like reading machine learning research papers, the maths is sometimes hard to follow. That is why I like implementing the algorithms in R by myself. Of course this means digging through the maths and the algorithms as well. However, you can challenge your understanding of the algorithm directly.

In my two subsequent blog post I will introduce two machine learning algorithms in 150 lines of R Code. This blog post will be about regression trees, which are the foundation of most tree-based algorithms. You can find the other blog post about coding gradient boosted machines from scratch on our blog. The algorithms will cover all core mechanics, while being very generic. You can find all code on my GitHub.

Gathering all puzzle pieces



Surely, there are tons of great articles out there which explain regression trees theoretically accompanied with a hands-on example. This is not the objective of this blog post. If you are interested in a hands-on tutorial with all necessary theory I strongly recommend this tutorial. The objective of this blog post is to establish the theory of the algorithm by writing simple R code. You do not need any prior knowledge of the algorithm to follow. The only thing you need to know is our objective: We want to estimate our real-valued target (`y`) with a set of real-valued features (`X`).

Most probably you are already familiar with decision trees, which is a machine learning algorithm to solve classification tasks. As the name itself states regression trees solve regressions, i.e. estimation with continuous scaled targets. These kind of trees are the key part of every tree-based method, since the way you grow a tree is more of the same really. The differing parts among the implementations are mostly about the splitting rule. In this tutorial we will program a very simple, but generic implementation of a regression tree.

Fortunately, we do not have to cover much maths in this tutorial, because the algorithm itself is rather a technical than a mathematical challenge. With that said, the technical path I have chosen might not be the most efficient way, but I tried to trade-off efficiency with simplicity.

Anyway, as most of you might know decision or regression trees are rule-based approaches. Meaning, we are trying to split the data into partitions conditional to our feature space. The data partitioning is done with the help of a splitting criterion. There is no common ground on how to do those splits, there are rather multiple different splitting criteria with different pros and cons. We will focus on a rather simple criterion in this tutorial. Bear with me, here comes some maths.

$$SSE = \sum_{i \in S_1} (y_i - \bar{(y)}_1)^2 + \sum_{i \in S_2} (y_i - \bar{(y)}_2)^2$$

Ok, so what does this state? This is the sum of squared errors determined in two different subsets (S_1 and S_2). As the name suggests, that should be something we want to minimize. In fact, it is the squared distance between the mean and the target within this data subset. In every node of our regression tree we calculate the SSE for every potential split we could do in our data for every feature we have to figure out the best split we can achieve.

Let us have a look at the R Code:

```
# This is the splitting criterion we minimize (SSE [Sum Of Squared Errors]):
# SSE = \sum{i \in S_1} (y_i - \bar(y)_1)^2 + \sum{i \in S_2} (y_i - \bar(y)_2)^2
sse_var <- function(x, y) {
  splits <- sort(unique(x))
  sse <- c()
  for (i in seq_along(splits)) {
    sp <- splits[i]
    sse[i] <- sum((y[x < sp] - mean(y[x < sp]))^2) +
              sum((y[x >= sp] - mean(y[x >= sp]))^2)
  }
  split_at <- splits[which.min(sse)]
  return(c(sse = min(sse), split = split_at))
}
```

The function takes two inputs our numeric feature `x` and our target real-valued `y`. We then go ahead and calculate the SSE for every unique value of our `x`. This means we calculate the SSE for every possible data subset we could obtain conditional on the feature. Often we want to cover more than one feature in our problem, which means that we have to run this function for every feature. As a result, the best splitting rule has the lowest SSE among all possible splits of all features. Once we have determined the best splitting rule we can split our data into these two subsets according to our criterion, which is nothing else than feature `x <= split_at` and `x > split_at`. We call these two subsets children and they again can be split into subsets again.

Let us lose some more words on the SSE though, because it reveals our estimator. In this implementation our estimator in the leaf is simply the avarage value of our target within this data subset. This is the simplest version of a regression tree. However, with some additional work you can apply more sophisticated models, e.g. an ordinary least squares fit.

The Algorithm

Enough with the talking, let's get to the juice. In the following you will see the algorithm in all of its beauty. Afterwards we will breakdown the algorithm in easy-to-digest code chunks.

```
#' reg_tree
#' Fits a simple regression tree with SSE splitting criterion. The estimator functio
#' is the mean.
#'
#' @param formula an object of class formula
#' @param data a data.frame or matrix
#' @param minsize a numeric value indicating the minimum size of observations
#'                in a leaf
#'
#' @return \itemize{
#' \item tree - the tree object containing all splitting rules and observations
#' \item fit - our fitted values, i.e. X %*% theta
#' \item formula - the underlying formula
#' \item data - the underlying data
#' }
#' @export
#'
#' @examples # Complete runthrough see: www.github.com/andrebleier/cheapml
reg_tree <- function(formula, data, minsize) {

  # coerce to data.frame
  data <- as.data.frame(data)

  # handle formula
  formula <- terms.formula(formula)

  # get the design matrix
  X <- model.matrix(formula, data)

  # extract target
  y <- data[, as.character(formula)[2]]

  # initialize while loop
  do_splits <- TRUE
```

```
# create output data.frame with splitting rules and observations
tree_info <- data.frame(NODE = 1, NOBS = nrow(data), FILTER = NA,
                        TERMINAL = "SPLIT",
                        stringsAsFactors = FALSE)

# keep splitting until there are only leafs left
while(do_splits) {

  # which parents have to be splitted
  to_calculate <- which(tree_info$TERMINAL == "SPLIT")

  for (j in to_calculate) {

    # handle root node
    if (!is.na(tree_info[j, "FILTER"])) {
      # subset data according to the filter
      this_data <- subset(data, eval(parse(text = tree_info[j, "FILTER"])))
      # get the design matrix
      X <- model.matrix(formula, this_data)
    } else {
      this_data <- data
    }

    # estimate splitting criteria
    splitting <- apply(X,  MARGIN = 2, FUN = sse_var, y = y)

    # get the min SSE
    tmp_splitter <- which.min(splitting[1,])

    # define maxnode
    mn <- max(tree_info$NODE)

    # paste filter rules
    tmp_filter <- c(paste(names(tmp_splitter), ">=",
                          splitting[2,tmp_splitter]),
                    paste(names(tmp_splitter), "<",
                          splitting[2,tmp_splitter]))

    # Error handling! check if the splitting rule has already been invoked
    split_here  <- !sapply(tmp_filter,
                           FUN = function(x,y) any(grepl(x, x = y)),
                           y = tree_info$FILTER)

    # append the splitting rules
    if (!is.na(tree_info[j, "FILTER"])) {
      tmp_filter  <- paste(tree_info[j, "FILTER"],
                           tmp_filter, sep = " & ")
    }

    # get the number of observations in current node
    tmp_nobs <- sapply(tmp_filter,
                       FUN = function(i, x) {
                         nrow(subset(x = x, subset = eval(parse(text = i))))
                       },
                       x = this_data)

    # insufficient minsize for split
    if (any(tmp_nobs <= minsize)) {
      split_here <- rep(FALSE, 2)
    }

    # create children data frame
    children <- data.frame(NODE = c(mn+1, mn+2),
                           NOBS = tmp_nobs,
                           FILTER = tmp_filter,
                           TERMINAL = rep("SPLIT", 2),
                           row.names = NULL)[split_here,]

    # overwrite state of current node
    tree_info[j, "TERMINAL"] <- ifelse(all(!split_here), "LEAF", "PARENT")

    # bind everything
    tree_info <- rbind(tree_info, children)
```

```
    # check if there are any open splits left
    do_splits <- !all(tree_info$TERMINAL != "SPLIT")
  } # end for
} # end while

# calculate fitted values
leafs <- tree_info[tree_info$TERMINAL == "LEAF", ]
fitted <- c()
for (i in seq_len(nrow(leafs))) {
  # extract index
  ind <- as.numeric(rownames(subset(data, eval(parse(text = leafs[i, "FILTER"])))))
  # estimator is the mean y value of the leaf
  fitted[ind] <- mean(y[ind])
}

# return everything
return(list(tree = tree_info, fit = fitted, formula = formula, data = data))
}
```

Uff, that was a lot of scrolling. Ok, so what do we have here. At first glance we see two loops a while loop and a for loop, which are essential to the algorithm. But let us start bit by bit. Let's have a look at the first code chunk.

```
# coerce to data.frame
data <- as.data.frame(data)

# handle formula
formula <- terms.formula(formula)

# get the design matrix
X <- model.matrix(formula, data)

# extract target
y <- data[, as.character(formula)[2]]

# initialize while loop
do_splits <- TRUE

# create output data.frame with splitting rules and observations
tree_info <- data.frame(NODE = 1, NOBS = nrow(data), FILTER = NA,
                        TERMINAL = "SPLIT",
                        stringsAsFactors = FALSE)
```

Essentially, this is everything before the while loop. Here, we see some input handling in the beginning and the extraction of our design matrix `X` and our target `y`. All our featuers are within this design matrix. `do_splits` is the condition for our while loop, which we will cover in a bit. The data frame `tree_info` is the key storage element in our algorithm, because it will contain every information we need about our tree. The essential piece of this object is the filter column. This column saves the paths (filter) we have to take through (to apply to) our data to get to a leaf (a terminal node) in our regression tree. We initiate this `data.frame` with `NODE = 1`, since at the beginning we are in the root node of our tree with the whole data set at our expense. Furthermore, there is a column called `TERMINAL`, which controls the state of the node. We have three different states `SPLIT`, `LEAF` and `PARENT`. When we describe a node with the `SPLIT` state, we mark it for a potential split. The state `PARENT` indicates, that we have already split this node. Lastly, the state `LEAF` marks terminal nodes of our regression tree.

Reaching the treetop

When do we reach such a terminal node? In many implementations there is a minimum size parameter, where we determine valid splits by the amount of observations of its children. If the children have lower data points than the minimum size the split is invalid and will not be done. Imagine not having this parameter in our case, we could end up with leafs covering only one observation. Another termination rule is if the lowest SSE is at a split we have already invoked within the branch and hence has already been invoked in this branch. The split at such a point would be nonsense, since we would end up with the same subset over and over again.

Basically, this is everything there is to the algorithm. The while and for loop just ensure, that we estimate and create every node in our `tree_info`. Thus, you can perceive the `tree_info` as sort of a job list, since we create new jobs within this data frame. Let us walk through the actual R code.

```
# keep splitting until there are only leafs left
while(do_splits) {

  # which parents have to be splitted
  to_calculate <- which(tree_info$TERMINAL == "SPLIT")
```

```
  for (j in to_calculate) {

    # handle root node
    if (!is.na(tree_info[j, "FILTER"])) {
      # subset data according to the filter
      this_data <- subset(data, eval(parse(text = tree_info[j, "FILTER"])))
      # get the design matrix
      X <- model.matrix(formula, this_data)
    } else {
      this_data <- data
    }

    # estimate splitting criteria
    splitting <- apply(X,  MARGIN = 2, FUN = sse_var, y = y)

    # get the min SSE
    tmp_splitter <- which.min(splitting[1,])

    # define maxnode
    mn <- max(tree_info$NODE)

    # paste filter rules
    tmp_filter <- c(paste(names(tmp_splitter), ">=",
                          splitting[2,tmp_splitter]),
                    paste(names(tmp_splitter), "<",
                          splitting[2,tmp_splitter]))

    # Error handling! check if the splitting rule has already been invoked
    split_here  <- !sapply(tmp_filter,
                           FUN = function(x,y) any(grepl(x, x = y)),
                           y = tree_info$FILTER)

    # append the splitting rules
    if (!is.na(tree_info[j, "FILTER"])) {
      tmp_filter  <- paste(tree_info[j, "FILTER"],
                           tmp_filter, sep = " & ")
    }

    # get the number of observations in current node
    tmp_nobs <- sapply(tmp_filter,
                       FUN = function(i, x) {
                         nrow(subset(x = x, subset = eval(parse(text = i))))
                       },
                       x = this_data)

    # insufficient minsize for split
    if (any(tmp_nobs <= minsize)) {
      split_here <- rep(FALSE, 2)
    }

    # create children data frame
    children <- data.frame(NODE = c(mn+1, mn+2),
                           NOBS = tmp_nobs,
                           FILTER = tmp_filter,
                           TERMINAL = rep("SPLIT", 2),
                           row.names = NULL)[split_here,]

    # overwrite state of current node
    tree_info[j, "TERMINAL"] <- ifelse(all(!split_here), "LEAF", "PARENT")

    # bind everything
    tree_info <- rbind(tree_info, children)

    # check if there are any open splits left
    do_splits <- !all(tree_info$TERMINAL != "SPLIT")
  } # end for
} # end while
```

The while loop covers our tree depth and our for loop all splits within this certain depth. Within the while loop we seek every row in our `tree_info`, which we still have to estimate, i.e. all nodes in the `"SPLIT"` state. In the first iteration it would be the first row, our root node. The for loop iterates over all potential splitting nodes. The first if condition ensures that we filter our data according to the tree depth. Of course, there is no filter in the root node that is why we take the data as is. But imagine calculating possible splits for a parent in depth level two. The filter would look similar to this one: `"feature_1 > 5.33 & feature_2 <= 3.22"`.

How do we apply splitting?

Afterwards we seek the minimum SSE by applying the `sse_var` function to every feature. Note, that in this version we can only handle numeric features. Once we have found the best splitting variable, here named `tmp_splitter`, we build the according filter rule in object `tmp_filter`.

We still have to check if this is a valid split, i.e. we have not invoked this split for this branch yet and we have sufficient observations in our children. Our indicator `split_here` rules whether we split our node. Well, that is about it. In the last passage of the loop we prepare the output for this node. That is, handling the state of the calculated node and adding the children information to our job list `tree_info`. After the for loop we have to check whether our tree is fully grown. The variable `do_splits` checks whether there are any nodes left we have to calculate. We terminate the calculation if there are no `"SPLIT"` nodes left in our `tree_info`.

Extracting our estimates

```
# calculate fitted values
 leafs <- tree_info[tree_info$TERMINAL == "LEAF", ]
 fitted <- c()
 for (i in seq_len(nrow(leafs))) {
   # extract index
   ind <- as.numeric(rownames(subset(data, eval(parse(
                                                text = leafs[i, "FILTER"])))))
   # estimator is the mean y value of the leaf
   fitted[ind] <- mean(y[ind])
 }
```

At the end of our calculation we have a filter rule for every leaf in our tree. With the help of these filters we can easily calculate the fitted values by simply applying the filter on our data and calculating our fit, i.e. the mean of our target in this leaf. I am sure by now you can think of a way to implement more sophisticated estimators, which I would leave up to you.

Well, that's a regression tree with minimum size restriction. I have created a little runthrough with data from my simulation package on my GitHub, which you can checkout and try everything on your own. Make sure to checkout my other blog post about coding gradient boosted machines from scratch.

ÜBER DEN AUTOR

André Bleier

The most exciting part of being a data scientist at STATWORX is to find this unique solution to a problem by fusing machine learning, statistics, and business knowledge.

CODING METHODEN R R-BLOGGERS TREE