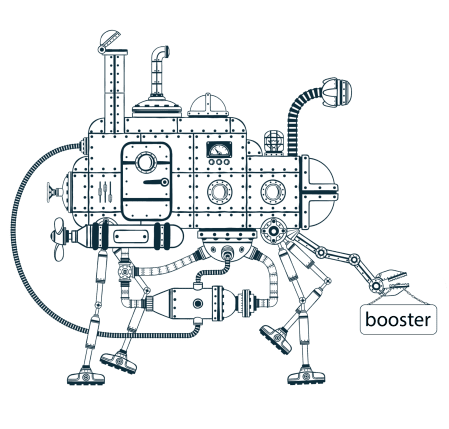
**Motivation**

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. 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 am a fan of implementing the algorithms in R by myself. Of course this means digging through the maths as well. However, you can challenge your understanding of the algorithm directly.



This blog post will introduce you to gradient boosted machines. Surely, there are tons of great articles out there which explain gradient boosting theoretically accompanied with a hands-on example. This is not the objective of this blog post.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.

**Gathering all puzzle pieces**

Gradient boosting is a very special machine learning algorithm, because it is rather a vehicle for machine learning algorithms rather than a machine learning algorithm itself. That is because you can incorporate any machine learning algorithm within gradient boosting. I admit that sounds quite confusing, but it will be clear by the end of this post.

Anyway, what do we need to boost those gradients? Well, we need to define our problem (a little maths is necessary ).

We have the following problem: \textbf{y} = \textbf{X} \boldsymbol{\theta} + \textbf{u}

This is just a simple regression equation, where we state that we want to map our features \textbf{X}onto a real valued target \textbf{y}with estimator \boldsymbol{\theta}and an error term (residuals) \textbf{u}. This is nothing out of the ordinary every machine learning algorithm starts here or somewhere close to this equation. However, unlike other machine learning algorithms, specifying a loss funciton is mandatory.

# this is our loss function

# y - the target

# yhat - the fitted values

loss <- function(y, yhat) return(mean(1/2\*(y-yhat)^2))

Our objective is to find the minimum of this loss function, which is an adaption of the mean squared error. The algorithm exploits the fact that in a minimum there is no slope in our loss function. The gradient of our function, which is the negative partial derivative with respect to yhat, describes this slope. So we can actually reformulate our objective to searching a gradient of zero or close to zero to ultimately fulfill our goal of minimzing the loss.

# the derivative of our loss function is the gradient

# y - the target

# yhat - the fitted values

gradient <- function(y, yhat) {return(y - yhat)}

Now comes the interesting part of the algorithm. In our case the gradient coincides with the residuals u = y – yhat. Remember, we want the gradient to be zero or close to zero. Or put differently, we want the state y = yhat. The algorithm exploits this fact by simply iteratively fitting (boosting) the residuals (the gradient) instead of y itself. This means we update our gradient in every iteration to improve our fit in general. This step describes our movement to the minimum of our loss function. We will clarify this observation with a little exmple once we see the theory in action.

**The algorithm**

Let us first have a look at the algorithm as a whole.

# grad\_boost

#' Fit a boosted linear model

#'

#' @param formula an object of class formula

#' @param data a data.frame or matrix

#' @param nu a numeric within the range of [0,1], the learning rate

#' @param stop a numeric value determining the total boosting iterations

#' @param loss.fun a convex loss function which is continuously differentiable

#' @param grad.fun a function which computes the gradient of the loss function

#' @param yhat.init a numeric value determining the starting point

#'

#' @return \itemize{

#' \item theta - this is our estimator

#' \item u - this is the last gradient value

#' \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

grad\_boost <- function(formula, data, nu = 0.01, stop,

grad.fun, loss.fun, yhat.init = 0) {

# 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]]

# initial fit

fit <- yhat.init

# initialize the gradient with yhat.init

u <- grad.fun(y = y, yhat = fit)

# initial working parameter (our theta)

# this is just an empty body

theta <- rep(0, ncol(X))

# track loss

loss <- c()

# boost from 1 to stop

for (i in 1:stop) {

# estimate working parameter (via OLS)

# theta\_i = (X'X)^-1 X'y

# This is the (base procedure) where you can literally place

# any optimization algorithm

base\_prod <- lm.fit(x = X, y = u)

theta\_i <- coef(base\_prod)

# update our theta

theta <- theta + nu\*as.vector(theta\_i)

# new fitted values

fit <- fit + nu \* fitted(base\_prod)

# update gradient

u <- grad.fun(y = y, yhat = fit)

# update loss

loss <- append(loss, loss.fun(y = y, yhat = fit))

}

# naming the estimator

names(theta) <- colnames(X)

# define return

return(list(theta = theta, u = u, fit = fit, loss = loss,

formula = formula, data = data))

}

All the magic happens within the for loop, which is coded in less than 30 lines of code. Everything else is just input/output handling and so on.

Let us breakdown what is happening.

# 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]]

# initial fit

fit <- f.init

# initialize the gradient with yhat.init

u <- grad.fun(y = y, yhat = fit)

# initial working parameter (our theta)

# this is just an empty body

theta <- rep(0, ncol(X))

# track loss

loss <- c()

The first part is just input handling, but we can see that we have to initialize the gradient, by starting with a best guess fit yhat.init, e.g. zero. After initializing, we need some containers to store our loss and our estimator theta.

The next part is all about our optimization, where we iteratively approaching the minimum of our prespecified loss function.

# boost from 1 to stop

for (i in 1:stop) {

# estimate working parameter (via OLS)

# theta\_i = (X'X)^-1 X'y

# This is the (base procedure) where you can literally place

# any optimization algorithm

base\_prod <- lm.fit(x = X, y = u)

theta\_i <- coef(base\_prod)

# update our theta

theta <- theta + nu\*as.vector(theta\_i)

# new fitted values

fit <- fit + nu \* fitted(base\_prod)

# update gradient

u <- grad.fun(y = y, yhat = fit)

# update loss

loss <- append(loss, loss.fun(y = y, yhat = fit))

}

**Unveiling the magic**

Alright, let’s get to the gravy. We will go through the first iteration step by step. The first thing we are doing is to estimate the gradient with a simple regression. Yes, a simple regression which you unfortunately have to accept here for a moment. Remember how we initialized the gradient. In the first iteration we are doing ordinary least squares with our features X and our target y, because the gradient coincides with our target, i.e. u = y - 0. What happens next is the last missing puzzle piece to gradient boosting. We are simply updating our estimator theta the fitted values fit and our gradient u. But what does this accomplish? Suppose, this multiplier nu is equal to one, then the following is quite straightforward. We are stating that our new fit – we initialized it with zero – is plain ordinary least squares and our estimator theta is the OLS estimate as well. After updating the gradient we observe that the gradient in iteration two is actually the residuals from the OLS fit of iteration one.

Let us recap that in some code real quick. In the first iteration we were actually fitting y, since u = y - 0 and in the second iteration we are fitting u = resid(base\_prod), since u = y – fitted(base\_prod). Thus, if we repeat this often enough u should converge to zero, since our fit will improve everytime. But why should it improve everytime? The algorithm allows us to fit the gradient – which again are the residuals – at every iteration with completely new estimates theta\_i. Naturally, this guess will be good enough to let the gradient converge towards zero or put differently minimize the distance between our target y and our fit yhat. And yes, this means that with gradient boosting we can theoretically reach the equilibrium of y = yhat, which we desire. But do we though? Remember we could theoretically reach that state in every problem we estimate with gradient boosting. Well, the important point to note here is that we can only do this for the data we have at hand. Chances are very high when observing new data, that our gradient boosted machine is garbage, because our estimator theta does not cover the underlying effect of our featuers on the target, but covers rather a combination which minimizes our loss function with the data we fed it. That is why we have to control the boosting iterations. We have to terminate the iterative fitting at the right time, to get a theta which explains the effect we are interested in rather than minimizing loss functions. There are so-called early-stopping techniques which terminate the learning by an exceeding of an out of sample risk to avoid this phenomenom called overfitting the data.

Anyway, let us come back to the parameter nu. This is the shrinkage (learning rate) parameter. It controls how big the steps towards the minimum of our loss function are. Suppose, the same scenario as above with nu = 0.1, that would mean we would not get the full OLS fit, but only 10% of it. As a result, we would need more iterations to converge to the minimum, however, empirically smaller learning rates have been found to yield better predictive accuracies.(2)

At the beginning I have stated that gradient boosting is rather a vehicle for machine learning algorithms than a machine learning algorithm itself. Essentially, this is because instead of lm.fit we could have used any other optimization function here. Most gradient boosted machines out there use tree-based algorithms. This makes the gradient boosted machine to a very unique machine learning algorithm.

**References**

1. Bühlmann, P., & Hothorn, T. (2007). Boosting algorithms: Regularization, prediction and model fitting. *Statistical Science*, *22*(4), 477-505
2. Friedman, J. H. (2001). Greedy function approximation: a gradient boosting machine. *Annals of statistics*, 1189-1232.