**Introduction**

This is the first of a series of post on the BooST (Boosting Smooth Trees). The BooST is a model that uses Smooth Trees as base learners, which makes it possible to approximate the derivative of the underlying model. In this post, we will show some examples on generated data of how the BooST approximates the derivatives and we also will discuss how the BooST may be a good choice when dealing with smooth functions if compared to the usual discrete Regression Trees.

**Installation**

To install the BooST package in R you should run the following code:

library(devtools)

Library("BooST")

Note that this R implementation is suited just for small problems. If you want to use the BooST on larges instances with more speed we recommend the Julia implementation until the C++ is ready. The Julia package can be installed with:

Pkg.clone("[https://github.com/gabrielrvsc/BooSTjl.jl"](https://github.com/gabrielrvsc/BooSTjl.jl%22))

and then using BooSTjl in your Julia terminal. Both packages have documentation for all exported functions.

**Example 1: Cosine**

The first example is the one we briefly discussed in the previous post. We are going to generate data from:

\displaystyle y_i = \cos( \pi [x_{i,1}+x_{i,2}])+ \varepsilon_i

where x_{i,1} \sim N(0,1), x_{i,2}is a Bernoulli with p = 0.5and \varepsilon_i \sim N(0,1). The function used to generate the data is:

dgp = function(N,r2){

X = matrix(rnorm(N\*2,0,1),N,2)

X[,ncol(X)] = base::sample(c(0,1),N,replace=TRUE)

aux = X

yaux = cos(pi\*(rowSums(X)))

vyaux = var(yaux)

ve = vyaux\*(1-r2)/r2

e = rnorm(N,0,sqrt(ve))

y = yaux+e

var(yaux)/var(y)

return(list(y = y, X = X))

}

We are going to generate 1000 observations with an R2 of 0.3 (a lot of noise). The code below generates the data and runs the BooST and the Boosting using the xgboost package. We estimated 300 trees in each model with a step of 0.2. The last lines in the code just organize the results in a data.frame and generate values for the real function we want to recover.

library(BooST)

library(tidyverse)

library(xgboost)

library(reshape2)

set.seed(1)

data = dgp(N = 1000, r2 = 0.3)

y = data$y

x = data$X

set.seed(1)

BooST\_Model = BooST(x,y, v = 0.2, M = 300 ,display = TRUE)

xgboost\_Model = xgboost(x,label = y, nrounds = 300, params = list(eta = 0.2, max\_depth = 3))

x1r = rep(seq(-4,4,length.out = 1000), 2)

x2r = c(rep(0,1000), rep(1,1000))

yr = cos(pi\*(x1r+x2r))

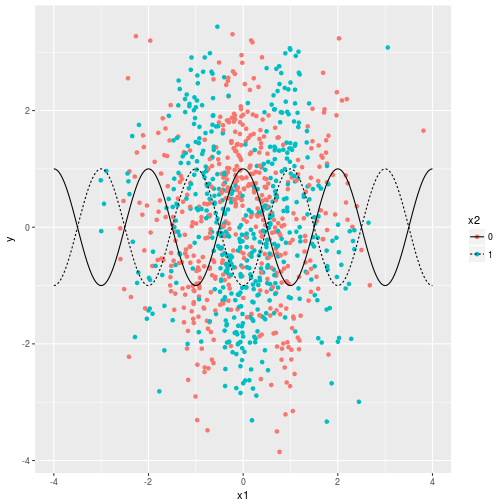
real\_function = data.frame(x1 = x1r, x2 = as.factor(x2r), y = yr)

fitted = data.frame(x1 = x[,1],x2 = as.factor(x[,2]), BooST = fitted(BooST\_Model),

xgboost = predict(xgboost\_Model,x), y = y)

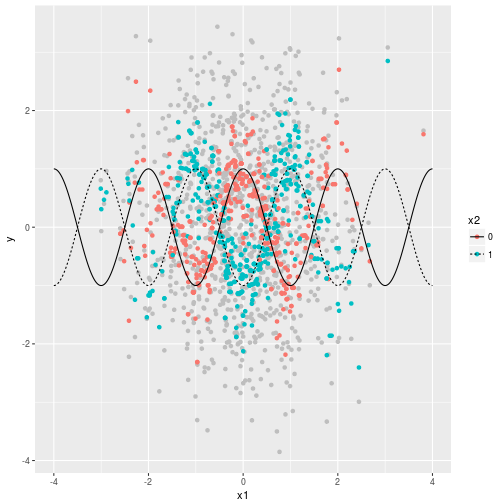
Before going into the results let’s have a look at the data in the figure below. The two black lines are the real cosine function we used and the dots are the data we generated. In this first look it seems hard to recover the real function from this data.

ggplot() + geom\_point(data = fitted, aes(x = x1, y = y, color = x2)) + geom\_line(data = real\_function, aes(x = x1, y = y, linetype = x2))



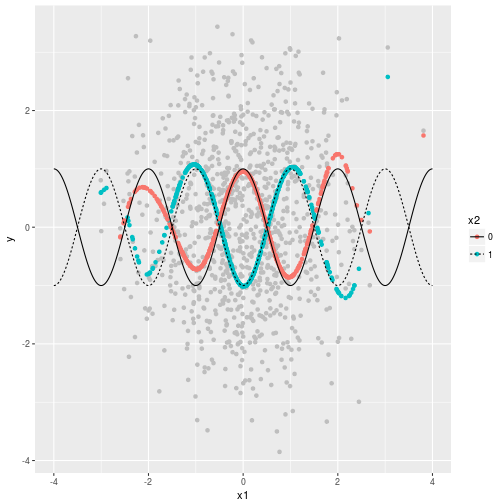
The next figure shows the result with regular Boosting using the xgboost package. The points in blue and red are what we fitted and the points in gray are the data from the previous plot. All the structure in the model comes from the cosine function represented by the two black lines. The main conclusion here is that we over-fitted the data.

ggplot() + geom\_point(data = fitted, aes(x = x1, y = y), color = "gray") + geom\_point(data = fitted, aes(x = x1, y = xgboost, color = x2)) + geom\_line(data = real\_function, aes(x = x1, y = y, linetype = x2))



The next plot shows what we obtained with the BooST. Again, red and blue points are fitted values and the data is in gray. The model fits the function very well with a few exceptions on extreme points where we have much less data.

ggplot() + geom\_point(data = fitted, aes(x = x1, y = y), color = "gray") + geom\_point(data = fitted, aes(x = x1, y = BooST, color = x2)) + geom\_line(data = real\_function, aes(x = x1, y = y, linetype = x2))



Next, let’s have a look at the derivatives. The code below estimates them and organizes the results for the plot.

BooST\_derivative = estimate\_derivatives(BooST\_Model, x, 1)

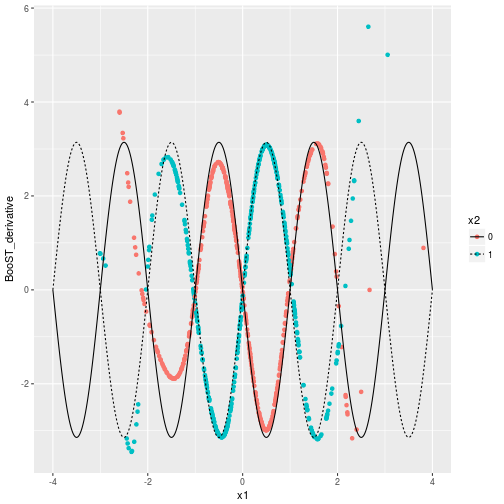
derivative = data.frame(x1 = x[,1],x2 = as.factor(x[,2]), derivative = BooST\_derivative)

dr = -1\*sin(pi\*(x1r+x2r))\*pi

real\_function$derivative = dr

The results are in the next plot. As we can see, the model also estimates the derivatives very well. However, the performance deteriorates as we go to the boarders where we have less data. This is a natural feature of many nonparametric models, which are more precise where the data are more dense.

ggplot() + geom\_point(data = fitted, aes(x = x1, y = BooST\_derivative, color = x2)) + geom\_line(data = real\_function, aes(x = x1, y = derivative, linetype = x2))



Finally, let’s generate new data from the same dgp and see how the BooST and the Boosting perform. The output of the following code is the BooST RMSE divided by the Boosting RMSE.

set.seed(2)

dataout=dgp(N=1000,r2=0.3)

yout = dataout$y

xout = dataout$X

p\_BooST = predict(BooST\_Model, xout)

p\_xgboost = predict(xgboost\_Model, xout)

sqrt(mean((p\_BooST - yout)^2))/sqrt(mean((p\_xgboost - yout)^2))

## [1] 0.9165293

**Example 2: More variables interacting**

In the previous example we only had two variables. What happens if we add more? In the next example we are going to generate data from a dgp with 10 variables, where the function we are interested in comes from all second order interactions with all variables:

\displaystyle y_i = f(x_i) + \varepsilon_i,  
where,

\displaystyle f(x_i) = \sum_{j=1}^{10}\sum_{k=i}^{10}x_{ij}x_{ik},

with all variables and \varepsilon_igenerated from a standard Normal distribution. The difficulty comes from the fact that we have to many interactions. Nevertheless, let’s see how the BooST recovers derivatives in this setup. Since we now have 10 variables interacting a plot like the one we did with the cosine is no longer possible. A good way to make this example more visual is to keep all variables fixed except one and see the derivative as we move on this one variable. However, this strategy needs a lot of data to work because we may be calculating derivatives in parts of the space that were poorly mapped. First, the dgp function:

dgp2 = function(N,k,r2){

yaux = rep(0,N)

x = matrix(rnorm(N\*k),N,k)

for(i in 1:k){

for(j in i:k)

yaux = yaux + x[,i]\*x[,j]

}

vyaux=var(yaux)

ve=vyaux\*(1-r2)/r2

e=rnorm(N,0,sqrt(ve))

y=yaux+e

var(yaux)/var(y)

return(list(y=y,x=x))

}

Now let’s estimate the model. Given the complexity of this model, we adopted a more conservative strategy with a step of 0.1 and smaller values for gamma (controls the transition on the logistic function). These adjustment may require more trees to converge. Therefore, we used M=1000 trees.

set.seed(1)

data = dgp2(1000,10,0.7)

x = data$x

y = data$y

set.seed(1)

BooSTmv = BooST(x,y, v = 0.1, M = 1000, display = TRUE, gamma = seq(0.5,1.5,0.01))

The next step is to put the data in the way we need to calculate the derivative. We are going to look at the derivative of y_iwith respect to the first variable x_{i1}keeping all other variables in the mean, which is 0. The solution in this case will be 2x_{i1}for the derivative.

xrep = x

xrep[,2:ncol(x)] = 0

derivative\_mv = estimate\_derivatives(BooSTmv,xrep,1)

df = data.frame(x1 = x[,1], derivative = derivative\_mv)

xr1 = seq(-4,4,0.01)

dfr = data.frame(x1 = xr1, derivative = 2\*xr1)

Finally, the results. The black line is the real derivative and the blue dots are what we estimated. Given the complexity of the problem the results are very good. The blue dots are always close to the black line except by some extreme values where we have less data.

ggplot() + geom\_point(data = df, aes(x = x1, y = derivative),color = "blue") + geom\_line(data = dfr, aes(x = x1, y = derivative)) + xlim(-4,4)

