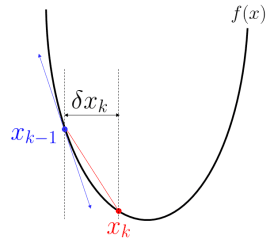
**An econometrician perspective**

I might start with a non-conventional introduction. But that’s actually how I understood what boosting was about. And I am quite sure it has to do with my background in econometrics.

The goal here is to solve something which looks likefor some loss function , and for some set of predictors . This is an optimization problem. Well, optimization is here in a function space, but still, that’s simply an optimization problem. And from a numerical perspective, optimization is solve using gradient descent (this is why this technique is also called [gradient boosting](https://en.wikipedia.org/wiki/Gradient_boosting)). And the gradient descent can be visualized like below



Again, the optimum is not some some real value , but some function . Thus, here we will have something like(as they write it is serious articles) where the term on the right can also be writtenI prefer the later, because we see clearly that is some model we fit on the remaining residuals.

We can rewrite it like that: definefor all . The goal is to fit a model so that , and when we have that optimal function, set (yes, we can include some shrinkage here).

Two important comments here. First of all, the idea should be weird to any econometrician. First, we fit a model to explain by some covariates . Then consider the residuals , and to explain them with the same covariate . If you try that with a linear regression, you’d done at the end of step 1, since residuals are orthogonal to covariates : no way that we can learn from them. Here it works because we consider simple non linear model. And actually, something that can be used is to add a shrinkage parameter. Do not consider but . The idea of *weak* learners is extremely important here. The more we shrink, the longer it will take, but that’s not (too) important.

I should also mention that it’s nice to keep learning from our mistakes. But somehow, we should stop, someday. I said that I will not mention this part in this series of posts, maybe later on. But heuristically, we should stop when we start to overfit. And this can be observed either using a split training/validation of the initial dataset or to use cross validation. I will get back on that issue later one in this post, but again, those ideas should probably be dedicated to another series of posts.

**Learning with splines**

Just to make sure we get it, let’s try to learn with splines. Because [standard splines](https://freakonometrics.hypotheses.org/52771) have fixed knots, actually, we do not really “learn” here (and after a few iterations we get to what we would have with a standard spline regression). So here, we will (somehow) optimize knots locations. There is a package to do so. And just to illustrate, use a Gaussian regression here, not a classification (we will do that later on). Consider the following dataset (with only one covariate)

|  |  |
| --- | --- |
| 1  2  3  4  5 | n=300  **set.seed**(1)  u=**sort**(**runif**(n)\*2\***pi**)  y=**sin**(u)+**rnorm**(n)/4  **df**=**data.frame**(x=u,y=y) |

For an optimal choice of knot locations, we can use

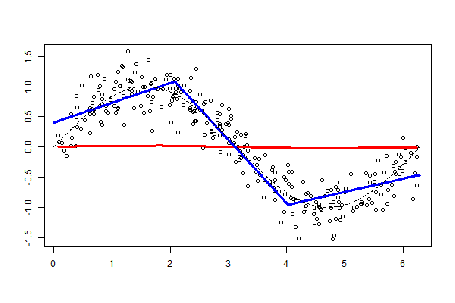
|  |  |
| --- | --- |
| 1  2 | **library**(freeknotsplines)  xy.freekt=freelsgen(**df**$x, **df**$y, degree = 1, numknot = 2, 555) |

With 5% shrinkage, the code it simply the following

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | v=.05  **library**(splines)  xy.freekt=freelsgen(**df**$x, **df**$y, degree = 1, numknot = 2, 555)  fit=**lm**(y~bs(x,degree=1,**knots**=xy.freekt@optknot),**data**=**df**)  yp=**predict**(fit,newdata=**df**)  **df**$yr=**df**$y - v\*yp  YP=v\*yp  **for**(**t** **in** 1:200){  xy.freekt=freelsgen(**df**$x, **df**$yr, degree = 1, numknot = 2, 555)  fit=**lm**(yr~bs(x,degree=1,**knots**=xy.freekt@optknot),**data**=**df**)  yp=**predict**(fit,newdata=**df**)  **df**$yr=**df**$yr - v\*yp  YP=**cbind**(YP,v\*yp)}  nd=**data.frame**(x=**seq**(0,2\***pi**,**by**=.01))  viz=**function**(M){  **if**(M==1) y=YP[,1]  **if**(M&gt;1) y=**apply**(YP[,1:M],1,**sum**)  **plot**(**df**$x,**df**$y,ylab="",xlab="")  **lines**(**df**$x,y,type="l",**col**="red",lwd=3)  fit=**lm**(y~bs(x,degree=1,**df**=3),**data**=**df**)  yp=**predict**(fit,newdata=nd)  **lines**(nd$x,yp,type="l",**col**="blue",lwd=3)  **lines**(nd$x,**sin**(nd$x),lty=2)} |

To visualize the ouput after 100 iterations, use

|  |  |
| --- | --- |
| 1 | viz(100) |

  
Clearly, we see that we learn from the data here… Cool, isn’t it?

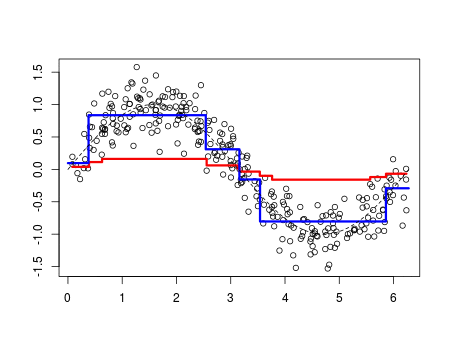
**Learning with stumps (and trees)**

Let us try something else. What if we consider at each step a regression tree, instead of a linear-by-parts regression (that was considered with linear splines).

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | **library**(rpart)  v=.1  fit=rpart(y~x,**data**=**df**)  yp=**predict**(fit)  **df**$yr=**df**$y - v\*yp  YP=v\*yp  **for**(**t** **in** 1:100){  fit=rpart(yr~x,**data**=**df**)  yp=**predict**(fit,newdata=**df**)  **df**$yr=**df**$yr - v\*yp  YP=**cbind**(YP,v\*yp)} |

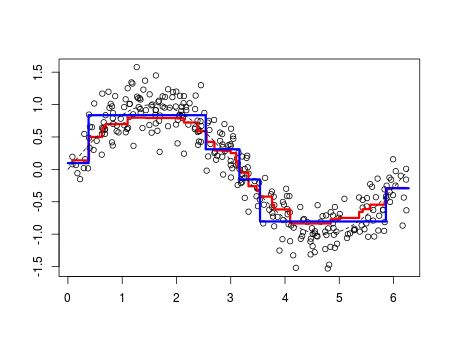
Again, to visualise the learning process, use

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | viz=**function**(M){  y=**apply**(YP[,1:M],1,**sum**)  **plot**(**df**$x,**df**$y,ylab="",xlab="")  **lines**(**df**$x,y,type="s",**col**="red",lwd=3)  fit=rpart(y~x,**data**=**df**)  yp=**predict**(fit,newdata=nd)  **lines**(nd$x,yp,type="s",**col**="blue",lwd=3)  **lines**(nd$x,**sin**(nd$x),lty=2)} |

  
This time, with those trees, it looks like not only we have a good model, but also a different model from the one we can get using a single regression tree.

What if we change the shrinkage parameter?

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | viz=**function**(v=0.05){  fit=rpart(y~x,**data**=**df**)  yp=**predict**(fit)  **df**$yr=**df**$y - v\*yp  YP=v\*yp  **for**(**t** **in** 1:100){  fit=rpart(yr~x,**data**=**df**)  yp=**predict**(fit,newdata=**df**)  **df**$yr=**df**$yr - v\*yp  YP=**cbind**(YP,v\*yp)}  y=**apply**(YP,1,**sum**)  **plot**(**df**$x,**df**$y,xlab="",ylab="")  **lines**(**df**$x,y,type="s",**col**="red",lwd=3)  fit=rpart(y~x,**data**=**df**)  yp=**predict**(fit,newdata=nd)  **lines**(nd$x,yp,type="s",**col**="blue",lwd=3)  **lines**(nd$x,**sin**(nd$x),lty=2)} |

  
There is clearly an impact of that shrinkage parameter. It has to be small to get a good model. This is the idea of using *weak learners* to get a good prediction.

**Classification and Adaboost**

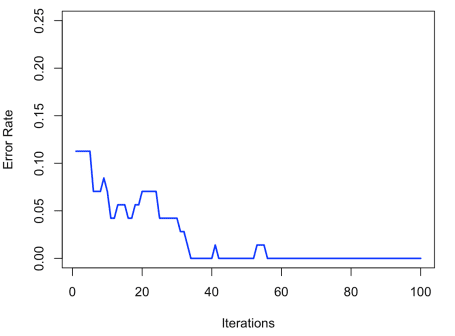
Now that we understand how bootsting works, let’s try to adapt it to classification. It will be more complicated because residuals are usually not very informative in a classification. And it will be hard to shrink. So let’s try something slightly different, to introduce the [adaboost](https://en.wikipedia.org/wiki/AdaBoost) algorithm.

In our initial discussion, the goal was to minimize a convex loss function. Here, if we express classes as , the loss function we consider is (this product ) was already discussed when we’ve seen the SVM algorithm. Note that the loss function related to the logistic model would be .

What we do here is related to gradient descent (or Newton algorithm). Previously, we were learning from our errors. At each iteration, the residuals are computed and a (weak) model is fitted to these residuals. The the contribution of this weak model is used in a gradient descent optimization process. Here things will be different, because (from my understanding) it is more difficult to play with residuals, because null residuals never exist in classifications. So we will add weights. Initially, all the observations will have the same weights. But iteratively, we ill change them. We will increase the weights of the wrongly predicted individuals and decrease the ones of the correctly predicted individuals. Somehow, we want to focus more on the difficult predictions. That’s the trick. And I guess that’s why it performs so well. This algorithm is well described in [wikipedia](https://en.wikipedia.org/wiki/AdaBoost#Example_algorithm_(Discrete_AdaBoost)), so we will use it.

We start with , then at each step fit a model (a classification tree) with weights (we did not discuss weights in the algorithms of trees, but it is straigtforward in the formula actually). Let denote that model (i.e. the probability in each leaves). Then consider the classifier which returns a value in . Then set where is the set of misclassified individuals,Then set and update finally the model usingas well as the weights(of course, devide by the sum to insure that the total sum is then 1). And as previously, one can include some shrinkage. To visualize the convergence of the process, we will plot the total error on our dataset.

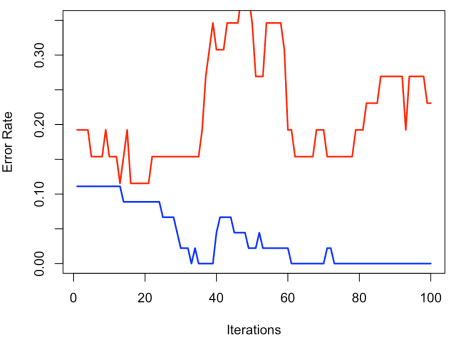
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | n\_iter = 100  y = (myocarde[,"PRONO"]==1)\*2-1  x = myocarde[,1:7]  error = **rep**(0,n\_iter)  f = **rep**(0,**length**(y))  w = **rep**(1,**length**(y)) #  alpha = 1  **library**(rpart)  **for**(i **in** 1:n\_iter){  w = **exp**(-alpha\*y\*f) \*w  w = w/**sum**(w)  rfit = rpart(y~., x, w, method="class")  g = -1 + 2\*(**predict**(rfit,x)[,2]&gt;.5)  e = **sum**(w\*(y\*g&lt;0))  alpha = .5\***log** ( (1-e) / e )  alpha = 0.1\*alpha  f = f + alpha\*g  error[i] = **mean**(1\*f\*y&lt;0)  }  **plot**(**seq**(1,n\_iter),error,type=&quot;l&quot;,  ylim=**c**(0,.25),**col**=&quot;blue&quot;,  ylab=&quot;Error Rate&quot;,xlab=&quot;Iterations&quot;,lwd=2) |

  
Here we face a classical problem in machine learning: we have a perfect model. With zero error. That is nice, but not interesting. It is also possible in econometrics, with polynomial fits: with 10 observations, and a polynomial of degree 9, we have a perfect fit. But a poor model. Here it is the same. So the trick is to split our dataset in two, a training dataset, and a validation one

|  |  |
| --- | --- |
| 1  2  3  4 | **set.seed**(123)  id\_train = **sample**(1:**nrow**(myocarde), size=45, **replace**=FALSE)  train\_myocarde = myocarde[id\_train,]  test\_myocarde = myocarde[-id\_train,] |

We construct the model on the first one, and we check on the second one that it’s not that bad…

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | y\_train = (train\_myocarde[,"PRONO"]==1)\*2-1  x\_train = train\_myocarde[,1:7]  y\_test = (test\_myocarde[,"PRONO"]==1)\*2-1  x\_test = test\_myocarde[,1:7]  train\_error = **rep**(0,n\_iter)  test\_error = **rep**(0,n\_iter)  f\_train = **rep**(0,**length**(y\_train))  f\_test = **rep**(0,**length**(y\_test))  w\_train = **rep**(1,**length**(y\_train))  alpha = 1  **for**(i **in** 1:n\_iter){  w\_train = w\_train\***exp**(-alpha\*y\_train\*f\_train)  w\_train = w\_train/**sum**(w\_train)  rfit = rpart(y\_train~., x\_train, w\_train, method="class")  g\_train = -1 + 2\*(**predict**(rfit,x\_train)[,2]&gt;.5)  g\_test = -1 + 2\*(**predict**(rfit,x\_test)[,2]&gt;.5)  e\_train = **sum**(w\_train\*(y\_train\*g\_train&lt;0))  alpha = .5\***log** ( (1-e\_train) / e\_train )  alpha = 0.1\*alpha  f\_train = f\_train + alpha\*g\_train  f\_test = f\_test + alpha\*g\_test  train\_error[i] = **mean**(1\*f\_train\*y\_train&lt;0)  test\_error[i] = **mean**(1\*f\_test\*y\_test&lt;0)}  **plot**(**seq**(1,n\_iter),test\_error,**col**='red')  **lines**(train\_error,lwd=2,**col**='blue') |

  
Here, as previously, after 80 iterations, we have a perfect model on the training dataset, but it behaves badly on the validation dataset. But with 20 iterations, it seems to be ok…

**R function**

Of course, it’s possible to use R functions,

|  |  |
| --- | --- |
| 1  2  3  4 | **library**(gbm)  gbmWithCrossValidation = gbm(PRONO ~ .,distribution = "bernoulli",  **data** = myocarde,n.trees = 2000,shrinkage = .01,cv.folds = 5,n.cores = 1)  bestTreeForPrediction = gbm.perf(gbmWithCrossValidation) |

Here cross-validation is considered, and not training/validation, as well as forests instead of single trees, but overall, the idea is the same… Off course, the output is much nicer (here the shrinkage is a very small parameter, and learning is extremely slow)  
