**By Gabriel Vasconcelos**

I already discussed some of the parameters we have to tune to estimate a boosting model using the xgboost package. In this post I will discuss the two parameters that were left out in part I, which are the gamma and the min\_child\_weight. These two parameters are much less obvious to understand but they can significantly change the results. Unfortunately, the best way to set them changes from dataset to dataset and we have to test a few values to select the best model. Note that there are many other parameters in the xgboost package. I am only showing the ones I use more.

Turning xgboost in R Part I

# **Introduction**

Tuning a Boosting algorithm for the first time may be a very confusing task. There are so many parameters to choose and they all have different behaviour on the results. Also, the best choice may depends on the data. Every time I get a new dataset I learn something new. A good understanding of classification and regression trees (CART) is also helpful because we will be boosting trees, you can start here if you have no idea of what a CART is.

My favourite Boosting package is the xgboost, which will be used in all examples below. Before going to the data let’s talk about some of the parameters I believe to be the most important. These parameters mostly are used to control how much the model may fit to the data. We would like to have a fit that captures the structure of the data but only the real structure. In other words, we do not want the model to fit noise because this will be translated in a poor out-of-sample performance.

* **eta:** Learning (or shrinkage) parameter. It controls how much information from a new tree will be used in the Boosting. This parameter must be bigger than 0 and limited to 1. If it is close to zero we will use only a small piece of information from each new tree. If we set eta to 1 we will use all information from the new tree. Big values of eta result in a faster convergence and more over-fitting problems. Small values may need to many trees to converge.
* **colsample\_bylevel:** Just like Random Forests, some times it is good to look only at a few variables to grow each new node in a tree. If we look at all variables the algorithm needs less trees to converge, but looking at, for example, 2/3 of the variables may result in models more robust to over-fitting. There is a similar parameter called colsample\_bytree that re-sample the variables in each new tree instead of each new node.
* **max\_depth:** Controls the maximum depth of the trees. Deeper trees have more terminal nodes and fit more data. Convergence also requires less trees if we grow them deep. However, if the trees are to deep we will be using a lot of information from the first trees and the final trees of the algorithm will have less importance on the loss function. The Boosting benefits from using information from many trees. Therefore it is intuitive that huge trees are not desirable. Smaller trees also grow faster and because the Boosting grow new trees in a pseudo-residual and we do not require any amazing adjustment for an individual tree.
* **sub\_sample:** This parameter determines if we are estimating a Boosting or a Stochastic Boosting. If we use 1 we obtain the regular Boosting. Values between 0 and 1 are for the stochastic case. The stochastic Boosting uses only a fraction of the data to grow each tree. For example, if we use 0.5 each tree will sample 50% of the data to grow. Stochastic Boosting is very useful if we have outliers because it limits their influence on the final model because they are dropped on several sub-samples. Moreover, it is possible to have a significant improvement on smaller instances because they are more susceptible to over-fitting.
* **gamma**: Controls the minimum reduction in the loss function required to grow a new node in a tree. This parameter is sensitive to the scale of the loss function, which will be linked to the scale of your response variable. The main consequence of using a gamma different from 0 is to stop the algorithm from growing useless trees that barely reduce the in-sample error and are likely to result in over-fitting. I will leave this parameter to a future post to save some space here.
* **min\_child\_weigth:** Controls the minimum number of observations (instances) in a terminal node. The minimum value for this parameter is 1, which allows the tree to have terminal nodes with only one observation. If we use bigger values we limit a possible perfect fit on some observations. This parameter will also be left to part II.

## Example

The data will be generate from the following equation:

\displaystyle y_i = (x_i'\beta)^2 + \epsilon_i 

where x_i \sim N(0,1), \beta_k = (-0.5)^k,~~ k=1,\dots,K, e_i \sim N(0,1). The number of variables, K, will be set to 10 and the number of instances to 1000.

The experiment will be to change each Boosting parameter keeping all the others constant to try to isolate their effects. The standard model will have the following parameters:

* eta: 0.1
* colsample\_bylevel: 2/3
* max\_depth: 6
* sub\_sample: 0.5

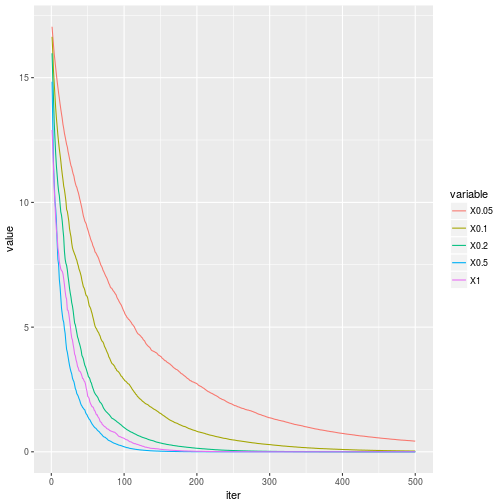
I will change each of these parameters to the values in the code. We will analyse the convergence and the root mean squared error (RMSE) in the test sample. The code below will prepare everything to run the models.

|  |  |
| --- | --- |
| 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  26  27  28  29  30  31  32  33  34  35  36 | library(xgboost)  library(ggplot2)  library(reshape2)  library(Ecdat)    set.seed(1)  N = 1000  k = 10  x = matrix(rnorm(N\*k),N,k)  b = (-1)^(1:k)  yaux=(x%\*%b)^2  e = rnorm(N)  y=yaux+e    # = select train and test indexes = #  train=sample(1:N,800)  test=setdiff(1:N,train)    # = parameters = #  # = eta candidates = #  eta=c(0.05,0.1,0.2,0.5,1)  # = colsample\_bylevel candidates = #  cs=c(1/3,2/3,1)  # = max\_depth candidates = #  md=c(2,4,6,10)  # = sub\_sample candidates = #  ss=c(0.25,0.5,0.75,1)    # = standard model is the second value  of each vector above = #  standard=c(2,2,3,2)    # = train and test data = #  xtrain = x[train,]  ytrain = y[train]  xtest = x[test,]  ytest = y[test] |

### eta

We will start analysing the eta parameter. The code below estimates Boosting models for each candidate eta. First we have the convergence plot, which shows that bigger values of eta converge faster. However, the train RMSE just below the plot shows that faster convergence does not translate into good out-of-sample performance. Smaller values of eta like 0.05 and 0.1 are the ones that produce smaller errors. My opinion is that in this case 0.1 gives us a good out-of-sample performance with acceptable convergence speed. Note that the result for eta=0.5 and 1 are really bad compared to the others.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | set.seed(1)  conv\_eta = matrix(**NA**,500,length(eta))  pred\_eta = matrix(**NA**,length(test), length(eta))  colnames(conv\_eta) = colnames(pred\_eta) = eta  **for**(i **in** 1:length(eta)){    params=list(eta = eta[i], colsample\_bylevel=cs[standard[2]],                subsample = ss[standard[4]], max\_depth = md[standard[3]],                min\_child\_weigth = 1)    xgb=xgboost(xtrain, label = ytrain, nrounds = 500, params = params)    conv\_eta[,i] = xgb$evaluation\_log$train\_rmse    pred\_eta[,i] = predict(xgb, xtest)  } |
| 1  2  3 | conv\_eta = data.frame(iter=1:500, conv\_eta)  conv\_eta = melt(conv\_eta, id.vars = "iter")  ggplot(data = conv\_eta) + geom\_line(aes(x = iter, y = value, color = variable)) | |

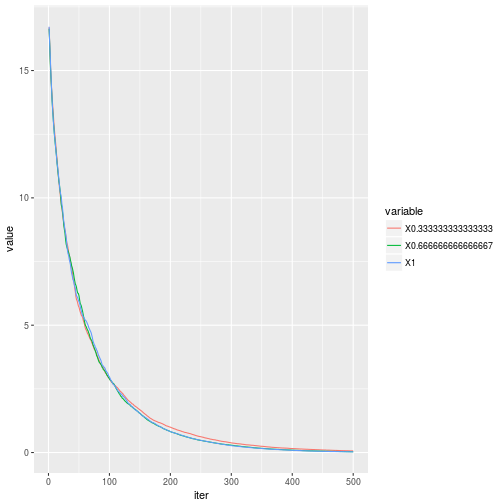


|  |  |
| --- | --- |
| 1 | (RMSE\_eta = sqrt(colMeans((ytest-pred\_eta)^2))) |
| 1  2 | ##      0.05       0.1       0.2       0.5         1  ##  9.964462 10.052367 10.223738 13.691344 20.929690 |

### colsample\_bylevel

The next parameter controls the fraction of variables (characteristics) tested in each new node. Recall that if we use 1 all variables are tested. Values smaller than 1 test only the correspondent fraction of variables. The convergence shows that the model is much less sensitive to the colsample\_bylevel. The curves with smaller values are just slightly above the curves with big values. The most accurate model seems to be the one that uses 25% of the sample in each tree. This result may change with different types of data. For example, the data we generated used the same distribution to create all response variables and the betas gave the variables a similar weight, which makes the sampling less relevant.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | set.seed(1)  conv\_cs = matrix(**NA**,500,length(cs))  pred\_cs = matrix(**NA**,length(test), length(cs))  colnames(conv\_cs) = colnames(pred\_cs) = cs  **for**(i **in** 1:length(cs)){    params = list(eta = eta[standard[1]], colsample\_bylevel = cs[i],                subsample = ss[standard[4]], max\_depth = md[standard[3]],                min\_child\_weigth = 1)    xgb=xgboost(xtrain, label = ytrain,nrounds = 500, params = params)    conv\_cs[,i] = xgb$evaluation\_log$train\_rmse    pred\_cs[,i] = predict(xgb, xtest)  } |
| 1  2  3 | conv\_cs = data.frame(iter=1:500, conv\_cs)  conv\_cs = melt(conv\_cs, id.vars = "iter")  ggplot(data = conv\_cs) + geom\_line(aes(x = iter, y = value, color = variable)) | |

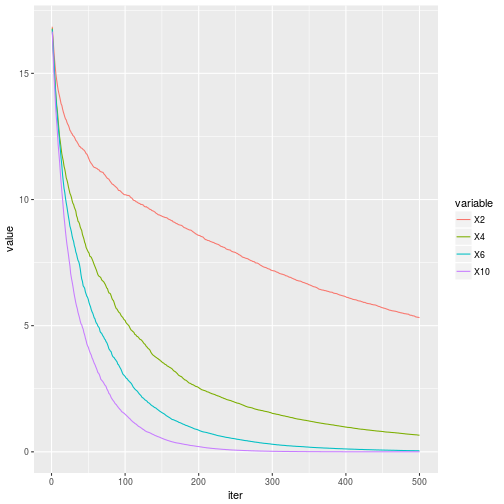


|  |  |
| --- | --- |
| 1 | (RMSE\_cs = sqrt(colMeans((ytest-pred\_cs)^2))) |
| 1  2 | ## 0.333333333333333 0.666666666666667                 1  ##          10.29836          10.05237          10.20938 |

### max\_depth

The third parameter is the maximum deepness allowed in each tree. Naturally, the model converges faster if we grow bigger trees (see the figure below). However, the best out-of-sample performance is for max\_depth=4. Remember that bigger trees are more likely to result in over-fitting. In my experience there is no need to use values bigger than 4 to 6, but there may be exceptions.

|  |  |  |
| --- | --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | set.seed(1)  conv\_md=matrix(**NA**,500,length(md))  pred\_md=matrix(**NA**,length(test),length(md))  colnames(conv\_md)=colnames(pred\_md)=md  **for**(i **in** 1:length(md)){    params=list(eta=eta[standard[1]],colsample\_bylevel=cs[standard[2]],                subsample=ss[standard[4]],max\_depth=md[i],                min\_child\_weigth=1)    xgb=xgboost(xtrain, label = ytrain,nrounds = 500,params=params)    conv\_md[,i] = xgb$evaluation\_log$train\_rmse    pred\_md[,i] = predict(xgb, xtest)  } | |
| 1  2  3 | conv\_md=data.frame(iter=1:500,conv\_md)  conv\_md=melt(conv\_md,id.vars = "iter")  ggplot(data=conv\_md)+geom\_line(aes(x=iter,y=value,color=variable)) |

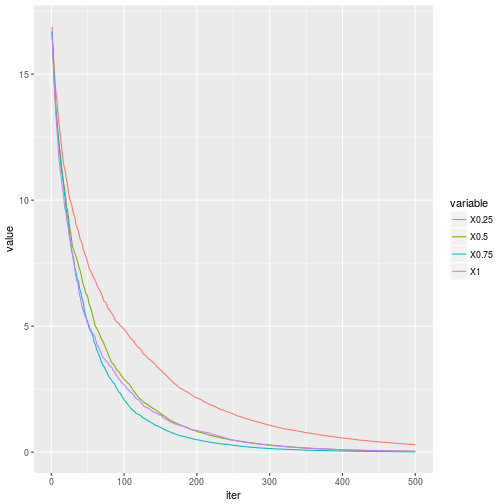


|  |  |
| --- | --- |
| 1 | (RMSE\_md=sqrt(colMeans((ytest-pred\_md)^2))) |
| 1  2 | ##         2         4         6        10  ## 12.502733  9.374257 10.134965 10.100691 |

### sub\_sample

The next parameter determines if we are estimating a Boosting or a Stochastic Boosting. Smaller values result in bigger errors in-sample but it may generate more robust out-of-sample estimates. However, as mentioned before, you will obtain bigger improvement on samples that have to many characteristics compares to the number of observations and when the data is very noisy. This is not the case here. Nevertheless, there is some improvement compared to the deterministic case, which I believed is mostly caused by outliers in the response variable (boxplot the variable y to see).

|  |  |  |
| --- | --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | set.seed(1)  conv\_ss=matrix(**NA**,500,length(ss))  pred\_ss=matrix(**NA**,length(test),length(ss))  colnames(conv\_ss)=colnames(pred\_ss)=ss  **for**(i **in** 1:length(ss)){    params=list(eta=eta[standard[1]],colsample\_bylevel=cs[standard[2]],                subsample=ss[i],max\_depth=md[standard[3]],                min\_child\_weigth=1)    xgb=xgboost(xtrain, label = ytrain,nrounds = 500,params=params)    conv\_ss[,i] = xgb$evaluation\_log$train\_rmse    pred\_ss[,i] = predict(xgb, xtest)  } | |
| 1  2  3 | conv\_ss=data.frame(iter=1:500,conv\_ss)  conv\_ss=melt(conv\_ss,id.vars = "iter")  ggplot(data=conv\_ss)+geom\_line(aes(x=iter,y=value,color=variable)) |



|  |  |
| --- | --- |
| 1 | (RMSE\_ss=sqrt(colMeans((ytest-pred\_ss)^2))) |
| 1  2 | ##      0.25       0.5      0.75         1  ##  9.731273 10.052367 11.119535 11.233855 |

The main conclusion here is that there is no unique rule to tune Boosting models. The best way is to test several configurations.

Stay tuned if you liked this article, we will be talking more about Boosting soon.

**Parameters**

* **min\_child\_weigth:** This parameter puts a lower bound on the number of instances in a terminal node. Bigger values require more instances in terminal nodes, which makes each tree in the boosting smaller and the algorithm more conservative.
* **gamma:** Controls the minimum reduction in the loss function required to grow a new node in a tree. This parameter is sensitive to the scale of the loss function, which will be linked to the scale of your response variable. The xgboost documentation defines general loss functions as L(\theta) = \sum_i \ell(\theta). In the quadratic loss, for example, we have \ell(\theta) = (y_i - \hat{y}_i)^2. If we want to use a gamma that requires a RMSE reduction in a split of at least mour gamma must be in the order of (m*N)^2, where Nis the sample size.

**Examples**

I will use a housing dataset from the Ecdat package, which includes 546 observations of sold houses with their prices and characteristics. The objective is to change the two parameters keeping everything else constant and see what happens.

library(xgboost)

library(ggplot2)

library(reshape2)

library(Ecdat)

data = Housing

N = nrow(data)

# = divided price by 10000 to use smaller values of gamma = #

y = data$price/10000

x = data[,-1]

# = Transform categorical variables into dummies = #

for(i in 1:ncol(x)){

if(is.factor(x[,i])){

x[,i] = ifelse(x[,i] == "yes",1,0)

}

}

x = as.matrix(x)

# = select train and test indexes = #

set.seed(1)

train=sample(1:N,456)

test=setdiff(1:N,train)

# = min\_child\_weights candidates = #

mcw=c(1,10,100,400)

# = gamma candidates = #

gamma=c(0.1,1,10,100)

# = train and test data = #

xtrain = x[train,]

ytrain = y[train]

xtest = x[test,]

ytest = y[test]

**min\_child\_weight**

The code below runs the boosting for the four min\_child\_weights in the mcw vector. You can see the convergence and the test RMSE for each setup below the code. The results show a significant change in the model as we move on the min\_child\_weight. Values of 1 and 10 seems to overfit the data a little while and a value of 400 ignores a lot of information and returns a poor predictive model. The best solution in this case was for min\_child\_weight = 100.

set.seed(1)

conv\_mcw = matrix(NA,500,length(mcw))

pred\_mcw = matrix(NA,length(test), length(mcw))

colnames(conv\_mcw) = colnames(pred\_mcw) = mcw

for(i in 1:length(mcw)){

params = list(eta = 0.1, colsample\_bylevel=2/3,

subsample = 1, max\_depth = 6,

min\_child\_weight = mcw[i], gamma = 0)

xgb = xgboost(xtrain, label = ytrain, nrounds = 500, params = params)

conv\_mcw[,i] = xgb$evaluation\_log$train\_rmse

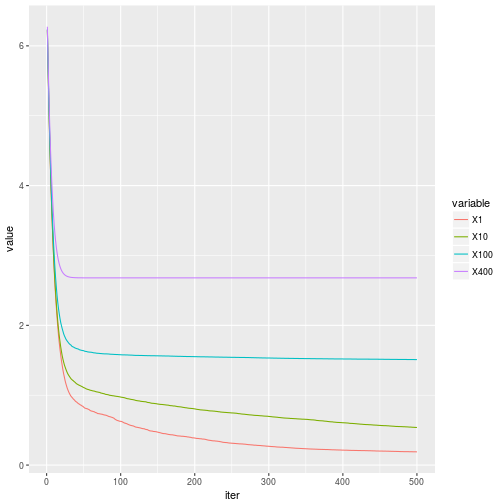
pred\_mcw[,i] = predict(xgb, xtest)

}

conv\_mcw = data.frame(iter=1:500, conv\_mcw)

conv\_mcw = melt(conv\_mcw, id.vars = "iter")

ggplot(data = conv\_mcw) + geom\_line(aes(x = iter, y = value, color = variable))



(RMSE\_mcw = sqrt(colMeans((ytest-pred\_mcw)^2)))

## 1 10 100 400

## 1.744679 1.751487 1.690150 2.607713

**gamma**

The results for gamma are similar to the min\_child\_weight. The models in the middle (gamma = 1 and gamma = 10) are superior in terms of predictive accuracy. Unfortunately, the convergence plot does not give us any clue on which model is the best. We have to test the model in a test sample or in a cross-validation scheme to select the most accurate. Both the min\_child\_weight and the gamma are applying shrinkage to the trees by limiting their size, bu they look at different measures to do so.

set.seed(1)

conv\_gamma = matrix(NA,500,length(gamma))

pred\_gamma = matrix(NA,length(test), length(gamma))

colnames(conv\_gamma) = colnames(pred\_gamma) = gamma

for(i in 1:length(gamma)){

params = list(eta = 0.1, colsample\_bylevel=2/3,

subsample = 1, max\_depth = 6, min\_child\_weight = 1,

gamma = gamma[i])

xgb = xgboost(xtrain, label = ytrain, nrounds = 500, params = params)

conv\_gamma[,i] = xgb$evaluation\_log$train\_rmse

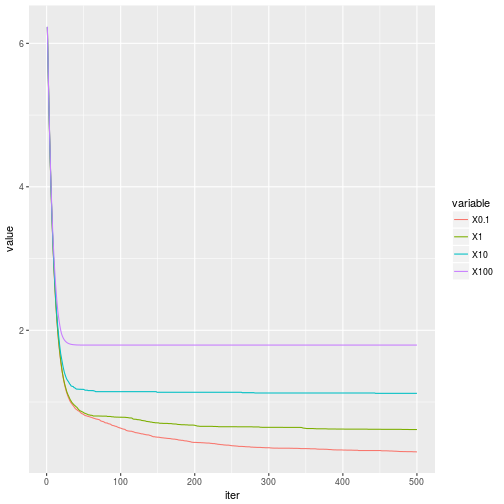
pred\_gamma[,i] = predict(xgb, xtest)

}

conv\_gamma = data.frame(iter=1:500, conv\_gamma)

conv\_gamma = melt(conv\_gamma, id.vars = "iter")

ggplot(data = conv\_gamma) + geom\_line(aes(x = iter, y = value, color = variable))



(RMSE\_gamma = sqrt(colMeans((ytest-pred\_gamma)^2)))

## 0.1 1 10 100

## 1.810786 1.737987 1.739655 1.909785