While reading [Dr. Nina Zumel](http://www.win-vector.com/site/staff/nina-zumel/)’s [excellent note on bias in common ensemble methods](http://www.win-vector.com/blog/2019/07/common-ensemble-models-can-be-biased/), I ran the examples to see the effects she described (and I think it is *very* important that she is establishing the issue, prior to discussing mitigation).

In doing that We ran into one more avoidable but strange issue in using [xgboost](https://github.com/dmlc/xgboost): when run for a *small* number of rounds it at first *appears* that xgboost doesn’t get the unconditional average or grand average right (let alone the conditional averages Nina was working with)!

Let’s take a look at that by running a trivial example in [R](https://www.r-project.org/).

Let’s take a trivial data set with only one explanatory variable “x” that is attempting to model a single response variable “y.”

library(xgboost)

library(wrapr)

d <- data.frame(

x = c(1 , 1 , 1 , 1 , 1 ),

y = c(10, 10, 10, 10, 10))

knitr::kable(d)

| **x** | **y** |
| --- | --- |
| 1 | 10 |
| 1 | 10 |
| 1 | 10 |
| 1 | 10 |
| 1 | 10 |

Now let’s do a bad job modeling y as a function of x.

model\_xgb\_bad <- xgboost(

data = as.matrix(d[, "x", drop = FALSE]),

label = d$y,

nrounds = 2, # part of the problem

verbose = 0,

params = list(

objective = "reg:linear"

#, base\_score = mean(d$y) # would fix issue

#, eta = 1 # would nearly fix issue

))

d$pred\_xgboost\_bad <- predict(

model\_xgb\_bad,

newdata = as.matrix(d[, "x", drop = FALSE]))

aggregate(cbind(y, pred\_xgboost\_bad) ~ 1,

data = d,

FUN = mean) %.>%

knitr::kable(.)

| **y** | **pred\_xgboost\_bad** |
| --- | --- |
| 10 | 4.65625 |

The above is odd: the training value of y has a mean of 10, yet the prediction averages to the very different value 4.66.

The issue is hidden in the usual value of [the “learning rate” eta](https://xgboost.readthedocs.io/en/latest/parameter.html). In gradient boosting we fit sub-models (in this case regression trees), and then use a linear combination of the sub-models predictions as our overall model. However: eta defaults to 0.3 ([ref](https://xgboost.readthedocs.io/en/latest/parameter.html)), which roughly means each sub-model is used to move about 30% of the way from the current estimate to the suggested next estimate. Thus with a small number of trees the model deliberately can’t model the unconditional average as it hasn’t been allowed to fully use the sum-model estimates.

The low learning rate is thought to help fix over-fit driven by depending too much on any one sub-learner. The issue goes away as we build larger models with more rounds, as a systematic issue (such as getting the grand-mean wrong) is quickly corrected as each sub-learner suggests related adjustments. This is part of the idea of boosting: some of the generalization performance comes from smoothing over behaviors unique to one sub-learner and concentrating on behaviors that aggregate across sub-learners (which may be important features of the problem). This idea can’t fight systematic model bias (errors that re-occur again and again) but does help with some model variance issues.

We can fix this by running xgboost closer to how we would see it run in production (which was in fact how Nina ran it in the first place!). Run for a larger number of rounds, and determine the number of rounds by cross-validation.

cvobj <- [xgb.cv](http://xgb.cv)(params = list(objective="reg:linear"),

as.matrix(d[, "x", drop = FALSE]),

label = d$y,

verbose = 0,

nfold = 5,

nrounds = 50)

evallog <- cvobj$evaluation\_log

( ntrees <- which.min(evallog$test\_rmse\_mean) )

## [1] 50

model\_xgb\_good <- xgboost(

data = as.matrix(d[, "x", drop = FALSE]),

label = d$y,

nrounds = ntrees,

verbose = 0,

params = list(

objective = "reg:linear"

))

d$pred\_xgboost\_good <- predict(

model\_xgb\_good,

newdata = as.matrix(d[, "x", drop = FALSE]))

aggregate(cbind(y, pred\_xgboost\_good) ~ 1,

data = d,

FUN = mean) %.>%

knitr::kable(.)

| **y** | **pred\_xgboost\_good** |
| --- | --- |
| 10 | 9.999994 |

Or we can fix this by returning to the [documentation](https://xgboost.readthedocs.io/en/latest/parameter.html), and noticing the somewhat odd parameter “base\_score”.

model\_xgb\_base <- xgboost(

data = as.matrix(d[, "x", drop = FALSE]),

label = d$y,

nrounds = 1,

verbose = 0,

params = list(

objective = "reg:linear",

base\_score = mean(d$y)

))

d$pred\_xgboost\_base <- predict(

model\_xgb\_base,

newdata = as.matrix(d[, "x", drop = FALSE]))

aggregate(cbind(y, pred\_xgboost\_base) ~ 1,

data = d,

FUN = mean) %.>%

knitr::kable(.)

| **y** | **pred\_xgboost\_base** |
| --- | --- |
| 10 | 10 |

base\_score is [documented as](https://xgboost.readthedocs.io/en/latest/parameter.html):

base\_score [default=0.5]

* The initial prediction score of all instances, global bias
* For sufficient number of iterations, changing this value will not have too much effect.

Frankly this parameter (and its default value) violate the [principle of least astonishment](https://en.wikipedia.org/wiki/Principle_of_least_astonishment). Most users coming to xgboost-regression from other forms of regression would expect the grand average to be quickly modeled, and not something the user has to specify (especially if there is in explicit constant column in the list of explanatory variables). It is a somewhat minor [“footgun”](https://en.wiktionary.org/wiki/footgun), but a needless footgun all the same.

We (at Win-Vector LLC) think this is an issue as we always teach: try a method on simple problems you know the answer to before trying it on large or complex problems you don’t have a solution for. We think one should build up intuition and confidence about a method by seeing how it works on small simple problems (even if its forte is large complex problems). The mathematics principle is: concepts that are correct or correct in the extremes. It is better to not have a problem in the first place than to have a problem with remedy at hand.

Why does this issue live on? Because, as the documentation says, it rarely matters in practice. However it may be a good practice to try setting base\_score = mean(d$y) (especially if your model is having problems and you are seeing a small number of trees in your xgboost model).

* Should one use xgboost? Heck yes! It is a very good implementation of an important machine learning algorithm.
* Should we always set the base\_score? That isn’t clear. The initial wrong setting of base\_score also biases the number of trees fit in cross-validation up, which may be a feature that other xgboost parameters may be tuned with respect to or counting on. In careful work (such as our [book](http://www.practicaldatascience.com/)) we do set the base\_score. In practical terms it often does not make a difference (as we saw above), so over-emphasizing this parameter can also give the student a strange impression of how boosting works.