Data Science Unplugged: Additive Boosting

Brent Johnson

2018-01-19

You can easily download the R Markdown version of this document and reproduce all my examples on your own. You may just need to install the rpart and rpart.plot packages.

Data Science Unplugged

When I want to really understand a machine learning or statistical tool, to really get a feel for how it works, I like to step away from the finished package and code the algorithm by hand. Sure, once in production mode I will use a package. Package authors often include helpful options and features that aren't worth my time to reconstruct. Finished packages are often optimized for speed. But to truly understand the nuts and bolts of what these packages are doing, I often need to first step away from them. I often like to code these algorithms in base R, Pandas (Python), or SAS/IML. Doing so forces me to understand exactly how the machine learning tool works.

In this document I'll explore additive boosting. As I do so, I'm going to tie my hands, so to speak, and just use base R as much as possible. As far as ground rules go, I will allow myself the use of plotting packages, the tidyverse, and other packages as long as they don't mask the fundamental algorithm that I'm trying most to make clear.

The benefits of this manual approach include not only an understanding of the algorithm, but greater knowledge of its assumptions and when it is or isn't likely to work well. Doing so also gives me great ideas for further improving the algorithm, possibly inventing unique enhancements on my own.

Boosting and ensemble predictive models

There are multiple flavors of boosting. Gradient boosting, for example, is a current favorite over at Kaggle. In this document, I'll explore the slightly older additive boosting cousin–called AdaBoost. Readers wanting to learn more ought to pursue Schapire and Freund's excellent book.

All boosting algorithms follow an ensemble modeling approach to generating predictions. Ensemble models are used almost exclusively for prediction (a limitation or caveat that I'll get to later) and generate their predictions only after averaging over dozens or hundreds (i.e., an ensemble) of separate ingredient models or predictions. Ensemble models algorithms are meta-models, in sense. They're "models of models."

Setting Things Up

I first need some dummy data to which I can apply my homegrown AdaBoost algorithm. The below generates the dummy data used extensively by Ryan Tibshirini and described in one of his lectures on page 8 and in Chapter 10 of the excellent book, The Elements of Statistical Learning. The following chunk generates a data frame, `X`, containing ten normally distributed independent random variables or predictors. I also generate a binary dependent variable, `Y`, that's only loosely correlated with the predictor variables:

Many AdaBoost implimentations require a binary outcome variable be coded as -1 or 1. My example requires 0 or 1 coded outcomes.

```
set.seed(415)
X <- data.frame(matrix(rnorm(10 * 1000, 0, 1),
    ncol = 10))
X$Y <- ifelse(rowSums(data.frame(lapply(X, function(x) x^2))) >
    qchisq(0.5, 10), 1, 0)
```

The above data will support my development of a "weak learner"–a classification that, on its own, enables an only slightly better-than-chance prediction. In words, my `Y` variable is built from the sum of my 10 squared independent variables. If the sum exceeds the median of a chi-square distribution having 10 degrees of freedom (which yields a value of 9.3), then `Y=1`; otherwise `Y=0` In general, boosting algorithms show the greatest prediction success when applied using a weak learner like this one. Further below in this article I'll contrast this with both a strong learner and even a learner containing just random noise.

I'm working in R, so, to further prepare my work environment I need to load some libraries. I first load rpart, a package for fitting classification and regression trees and also rattle and rpart.plot which is helpful for generating nice plots of classification trees like those rpart produces.

```
library(rpart)
library(rattle)
library(rpart.plot)
```

So, right at the start here I'm assuming the reader is somewhat familiar with classification via decision or regression trees and I won't code a classification tree from scratch in base R. That would be exhausting and detract from my purpose here. For what it's worth, I could just as easily demonstrate boosting if I employed logistic regression instead of a classification tree. Also, one can apply boosting to standard linear regression models as well as to classification trees. But boosting appears to be most often applied in tree-based settings, so I'll stick with rpart.

As an experiment, the reader could create a boosted logistic regression by inserting the code, "model <- glm(Y ~ X1 + X2 + X3 + X4 + + X5 + X6 + X7 + X8 + X9 + X10, weights=weight, data=X, family='quasibinomial')" in place of the rpart function below. Just recall, however, that the predicted values from such a model are on the logistic scale. So, one would need to modify the predict() function accordingly in order to append the 0/1 predicted values to yhat[[i]].

Boosting described in words–and in simple code

AdaBoost is perhaps the simplest boosting algorithm. Following Schapire and Freund's excellent description, AdaBoost consists of the following steps:

1. Fit a model to one's data
2. Compute the predicted outcome
3. Increase the relative weight given to the poorly predicted observations; reduce the relative weight given to successfully predicted outcomes.
4. Wash, rinse, and repeat for n iterations, saving the predicted outcomes from each iteration
5. Compute the winning prediction using a weighted average of the predictions from the n iterations

I implement these steps in the following code chunks. This is the AdaBoost algorithm unplugged. I begin by initializing a few objects for collecting the results. And I initialize a weight variable. This weight variable starts of with identical values for each observation or record in `X`.

```
# create placeholders for the forecast
# ensemble
alpha <- vector("list")
yhat <- vector("list")
models <- list()
success.vec <- mean(X$Y)

X$weight <- 1/nrow(X)  # Intitialize a weight vector
```

I next code AdaBoost. I first specify the number of boosted trees or iterations (`niter`) that I want in my ensemble. I then repeatedly estimate a classification tree (using the rpart function), saving the predicted values from each member of the ensemble into `yhat`. As for the construction of my classification tree and the rpart options, I won't go into a deep explanation. The function options below merely specify that each tree in my ensemble contains a single split with a minimum size of 100 observations per split. A classification or decision tree with a single split like this is called a decision "stump."

The AdaBoost algorithm inside the loop below might be best understood by working from the bottom up. Near the end of the algorithm, the final predicted probability (`predicted.prob`) of each observation is a weighted average of the predictions from all 100 trees or members of the ensemble. The relative weight (`alpha.weight`) given each ingredient prediction is determined by `alpha`. And the

`alpha` for each tree is in turn constructed from each model's logged relative prediction error (`.5*log((1-error)/error)`. Note that this prediction error is simply 1 minus the sum of the weights for all successfully predicted outcomes (`1- sum(as.numeric(X$Y == yhat[[i]])*X$weight)`).

By experimenting with different `error` values one can see that an ingredient tree's prediction is given exponentially more weight the higher its prediction success. For example, if a tree predicts no better than chance (error = .5) then its alpha would be zero. Conversely if a given tree in the ensemble predicts with 100% success (error = 0) then the weight given its prediction would be infinitely high (since `.5*log((1-0)/0) = Inf`). This is an exponential loss function and is the default option in many AdaBoost package implementations. Following this rule, highly predictive models in the ensemble get exponentially more weight than poorly predictive ones.

```
set.seed(415)
niter <- 100

for (i in seq(niter)) {
    model <- rpart(Y ~ X1 + X2 + X3 + X4 + +X5 +
        X6 + X7 + X8 + X9 + X10, data = X, maxdepth = 1,
        minbucket = 100, cp = -1, xval = 0, weights = weight,
        method = "class")
    models[[i]] <- model
    yhat[[i]] <- ifelse(predict(model)[, 2] >
        0.5, 1, 0)
    error <- 1 - sum(as.numeric(X$Y == yhat[[i]]) *
        X$weight)
    alpha[[i]] <- 0.5 * log((1 - error)/error)
    weight.adjustment <- ifelse(yhat[[i]] == X$Y,
        exp(-alpha[[i]]), exp(alpha[[i]]))
    new.unscaled.weight <- X$weight * weight.adjustment
    X$weight <- new.unscaled.weight/sum(new.unscaled.weight)

    # create weights for combining iterations in
    # the ensemble
    alpha.weight <- lapply(alpha, function(x) x/sum(unlist(alpha)))

    # created a weighted predicted probability
    # from the ensemble, weighted by alpha
    predicted.prob <- rowSums(data.frame(Map("*",
        yhat, alpha.weight)))
```