Boosting (and bagging)

Carsten F. Dormann

November 15, 2018

Boosting is a way to combine several a not-so-good algorithms into a magnificent, stupendous model. This is achieved by repeating the fitting over different "stages", giving more weights to the data with high residuals, and then refitting the algorithm *on top of* the previous stages. Thus, each new stage improves on those data that the previous stages have not been able to represent well. Boosting can be applied to any modelling algorithm (e.g. linear regression, CARTs, asf.), but it seems to work particularly well for "weak learners", i.e. algorithms that don't do so well by themselves (e.g. stumpy CARTs). **Bagging** is short for "bootstrap aggregation" and simply means averaging bootstraps.

1 Introduction

"Boosting" is the summary term for a variety of algorithms implementing the same idea: combine stage-wise statistical analyses of the same data, in which each next stage re-weights the data points based on the previous stage's model's residuals (Hastie *et al.*, 2009). This idea emerged out of the (successful) challenge to turn a "weak leaner", i.e. a poorly fitting algorithm, into a "strong learner". This idea has (at least) two elements:

1. Weighted regression (or alike); and
2. stage-wise modelling.

1.1 Weighted Regression

We first need to understand what a weighted regression is. Essentially, it gives each data point a different contribution to the overall model. For example, if we knew for each data point how accurately it was measured, then we would give more weight to those data points that are more accurate (Fig. 1). In the case of a (multiple) linear model ($Y = X\beta + \varepsilon$), the slope is estimated as

$$\hat{\beta} = (X^T X)^{-1} X^T Y.$$

In the case of the *weighted* linear model, we add a vector W, which contains a value for each data point (typically this is $1/(\text{standard error}_i^2$) for data point i). The algebraic solution for the weighted linear model is now:

$$\hat{\beta}_W = (X^T W X)^{-1} X^T W Y.$$

In boosting, the weights W vary from one stage to the next.

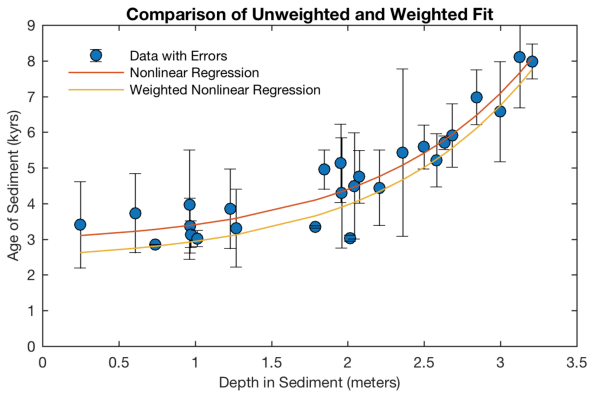

Figure 1: Example of weighted (non-linear) regression. In the weighted regression, the weight of each data point is proportional to its precision (i.e. $1/\text{variance} = 1/(\text{standard error}^2)$). Figure taken from `http://mres.uni-potsdam.de/index.php/2017/02/09/create-publishable-graphics-with-matlab/`.

1.2 Stage-wise Regression

In stage-wise modelling, a data set is analysed in several steps, each taking some form of output from the previous as input.

The reason to use stage-wise regression is that each stage is a poor model. Imagine, for example, a non-linear relationship between X and Y (e.g. $y = a + bx2 + \varepsilon$). If you fit a *linear* model, then your predictions will be poor. Also, the model will have a pattern in the residuals. Now, we can use this residual pa ern, by, in the next model stage, giving more weight to points that were poorly fit in the previous stage.

So, the algorithm (roughly) is:

1. Set W equal for all data points, i.e. $W = 1/n$.
2. Fit model: $F_0 = W_0 X \hat{\beta}_0$. Store initial model F_0 and model residuals R_0.
3. Compute new weights, W_1, as $W_1 = R_0^2$ (or similar).
4. Fit next stage's model's coeffcients $\hat{\beta}_1$ as $F_1 = X\hat{\beta}_0 + \gamma_1 W_1 X \hat{\beta}_1$. The parameter γ is optimised to get the best balance between the previous stages and the new one (for more see next section).
5. Repeat the last two steps m times, leading, recursively, to model $F_m = F_{m-1} + \gamma_m W_{m-1} X \hat{\beta}_m$.

(Note that I used the example of a linear regression because we are most familiar with it. However, the idea applies to any "learner", i.e. to any model type.)

As we see, each stage contains the results of all previous stages as part of the model. So what the first stage gets wrong, the next stage(s) will focus on (by having higher weights there). Figure 2 provides an example of the initial improvements over only 3 stages of a boosted CART.

Typically this system of stages will level off after some stages. To avoid overfitting, typically the entire process is wrapped into a cross-validation, which thus assesses whether the improvement in fit at a stage is maintained in prediction.

Another measure taken to prevent overfitting is to shrink parameter estimates. This is done by multiplying the contribution of a stage to the new model by a value ν (the "learning rate", typically $\nu < 0.1$) during the whole algorithm, i.e. $F_m = F_{m-1} + \nu\gamma_m W_{m-1} X \hat{\beta}_m$. The lower the learning rate, the less each stage contributes to the model and thereby cannot have dramatic effects on the fit.

1.3 Stage Aggregation

This section is only for clarification. Stage aggregation occurs during the above algorithm already. Each stage has a weight, γ_m, with which it contributes to the overall model. We thus do not have to implement any averaging step as we do in bagging (see below).

1.4 Variations on Boosting

There are plenty of variations on the above idea, particularly with respect to the "learner" used (i.e. CARTs instead of a linear model), the "loss function" (i.e. the weight-generating function, which we above took to be the residuals of the model), and particular adaptive tuning parameters or optimised to high-performance computing infrastructure. A list of boosting variants is listed e.g. on `https://en.wikipedia.org/wiki/Boosting(machinelearning)` under "See also" (e.g. AdaBoost, gradient boosting, xgboost).

When using CARTs (or "trees" for short), each stage can also comprise several new trees that are then combined. That is, the boosting proceeds in stages of several trees whose weight γ is the same. The reasoning behind this approach is that each tree is a rather poor model, especially when allowing only for a few splits. Thus, one can use several of them in any one stage. This idea neatly leads to the next topic: bagging.

2 Bagging

Bagging is such simple (although surprisingly "powerful") idea that it requires little space to explain (for a full exposition see Breiman, 1996):

1. Draw 500 different bootstrap samples of your data.

2. Fit your favourite model type (e.g. linear regression or kNN or CART or whatever) to each of them.

3. Predict to a new data point with each bootstrapped model, then average predictions (take majority vote for classification). Done.

Bagging is a good thing because it shrinks (in a sense) predictions from all bootstraps to their mean, thereby providing a more robust (i.e. less affected by outliers) version than the single fit on the full data set. In a sense, by bootstrapping we generate a wider distribution of predictions, and by averaging we pull them back in.

While bagging can be used for any type of model, Breiman (2001) added another twist to when using it with Classification And Regression Trees: instead of examining all predictors at each split of a tree, only use a random subset of them (typically $\sqrt{p}$ or $p/3$ for p predictors in classification or regression, respectively).

Why should this "random splitting of random data in trees", or "Random Forest" for short, be useful, on top of the effect of bagging? Well, the point of bagging is that its averaging pulls back spurious effects of single predictors. So if a predictor X_1 has a strong effect on response Y, then of course all (bootstrapped) trees will strongly feature X_1 and hence be (highly) correlated in their predictions. That severely reduces the usefulness of bagging, as essentially we would be averaging very similar values. That problem is (partially) solved by excluding X_1 from $2/3$ of the models: now the models differ much more, and yield more independent predictions to be averaged (see Dormann *et al.*, 2018, for a full exposition of the logic behind averaging model predictions).

2.1 Out-of-bag Error

Random Forest comes with a new type of cross-validation(-like) estimate of prediction error: the out-of-bag error. For each tree in a random forest, we have omitted some data points (to be precise, we omitted $1/e = 0.38$ of the data). We can now predict the value of the omitted y_i using *only those trees that did not use* row i. That is, we predict each data point in the data set only with those roughly 40% of the trees that did not include it. This OOB-error (computed as RMSE) is very close to an k-fold cross-validation error, but slightly more conservative (as it uses a smaller set of trees).

NOTE that in R's **randomForest**, the `predict`-function uses this OOB if no newdata are provided! That is a value possibly rather different to a "full" randomForest prediction.

2.2 Prediction Confidence Intervals for Random Forests

Since the random forest contains a lot of trees, one can compute an (asymptotically correct in infinitesimal jackknife) confidence interval (Wager *et al.*, 2014). Of course, one could simply bootstrap a random forest, but that is computationally wasteful, as all relevant information is already present in the random forest algorithm itself.[1]

2.3 Variable Importance

People seem to be interested in the importance of a predictor for reasons I cannot quite understand. We are using a fancy tool to generate predictions, so why should I care how the black box does it?

There are several ideas of how to measure variable importance, and this is not the point to delve too deeply into them. One logical way would be to compare all trees in a random forest that do **not** contain X_1 and compare their OOB-error with all those trees that **do** contain X_1. This is (somehow) never done. Another way would be to measure, at each tree's split which uses X_1, how much the node impurity decreases (averaged across all trees and measured as Gini coefficient in classification or as sum of squares in regression). That measure is indeed used and referred to as **mean decrease in node impurity**. Finally, we can shuffle each predictor in the OOB in turn and compare how much worse the prediction becomes (averaged across all trees). This measure is referred to as **mean decrease in accuracy**.

Imagine two highly correlated predictors X_1 and X_2 (e.g. $r = 0.8$). In trees where both preditors are present, in on half of the splits X_1 will be used, and in the other half X_2, every time with some positive effect on purity and accuracy. Since they are making the same statement, the overall importance allocated to this statement is now split over two variables, reducing their importance! This does not happen when only one of the correlated predictors is used. Since the probability of using X_i (in regression) is 1/3, the probability of having both predictors is $1/3^2 \approx 0.1$. We would thus expect the importance of correlated predictors to be severely underestimated in 10% of the trees, which I guess is a moderate proportion.[2]

When people claim that random forest is unaffected by collinearity (typically in the form of wishful thinking rather than by providing evidence Oppel *et al.*, 2009), they may refer to two different reasonings: (1) it doesn't affect prediction (which is true for all other models types in exactly the same way: being right for the wrong reason is still being right); and (2) importance is fine (which is probably a bit optimistic a statement, but reasonable). In extensive simulations of various model complexities and degrees of collinarity, randomForest performed better than GLM for high correlations, but no better than GAMs, lasso, ridge, BRTs, SVMs or MARS (Dormann *et al.*, 2013).

3 Examples

3.1 Boosting Regression

Time to show my true colours: You cannot boost a linear regression model. Sorry. What I did in the previous section was a didactic example, because you are familiar with linear regression and the analytical solution to it. The problem is, however, that a boosted linear regression is, de facto, still a linear regression: $X\beta_0 + \gamma_1 X\beta_1 + \cdots + \gamma_n X\beta_n = X(\beta 0 + \gamma_1\beta_1 + \cdots + \gamma_n\beta_n) = X\beta_{666}$.[3] Thus, you cannot improve on it using boosting![4]

[1] The R-package **randomForestCI** does that for us. Or, if you use **ranger** instead of **randomForest**, simply use `predict(., type='se', se.method='infjack')` to get standard errors.

[2] If we fiddle with the setting which proportion of predictors is used per tree, then this number will obviously change. Using half of the predictors will lead to a bias in 25%, and always using all will make the importance quantification highly dubious.

[3] See, e.g., `https://stats.stackexchange.com/questions/186966/gradient-boosting-for-linear-regression-why-does-it-not-work`; 666 is of course the number of the beast, although in China it is considered a lucky number (yeah right: I could not come up with a nice symbol here and used a ridiculous number instead).

[4] See also the other answer at the stackexchange-page for a nice and simple proof of this by *kirtap*.

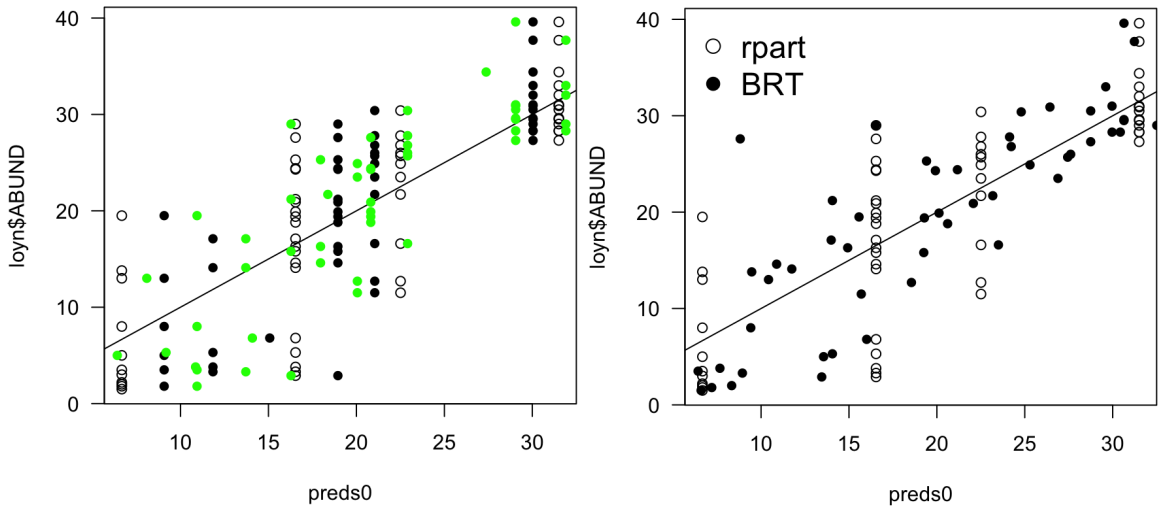

Figure 2: *Left*: Example of manually boosting the original CART (in black) twice (first white, then green dots). As a result, the observed data points (on the y-axis) become better aligned with the fitted values (on the x-axis). In this particular case, the correlation between fitted and observed improved from $r = 0.80$ (black) to $r = 86$ (green). *Right*: Eventually, and not manually, boosting a regression tree may lead to the right panel ($r = 0.87$).

As Matthew Drury put it (in the footnoted reference): "Boosting shines when there is no terse functional form around. Boosting decision trees lets the functional form of the regressor/classifier evolve slowly to fit the data, often resulting in complex shapes one could not have dreamed up by hand and eye. When a simple functional form is desired, boosting is not going to help you find it (or at least is probably a rather inefficient way to find it)." So let's turn to Classification And Regression Trees instead.

3.2 Boosting a Classification And Regression Tree

There are plenty of flavours of BRT (boosted regression trees) implemented in R and Python (and alike). Too many to review here. Thus, just a short example to illustrate the tuning parameters and the things to look out for when running BRTs.

Figure 2 depicts the improvements achieved by boosting a CART (for illustration, the left panel shows a manually boosted CART). In this case the settings for the boosted regression tree (BRT) were an tree complexity (a.k.a. interaction depth) of 2 and a learning rate (a.k.a. shrinkage) of 0.01. We can depict how the cross-validation error improves as we add more and more boosted stages (Fig. 3).

We can now try to tune these two most important parameters, preferably automatically.[5] The result may look like this:

```
Stochastic Gradient Boosting

No pre-processing
Resampling: Cross-Validated (10 fold)
Summary of sample sizes: 51, 51, 50, 49, 52, 51, ...
Resampling results across tuning parameters:

  shrinkage  interaction.depth  RMSE      Rsquared   MAE
  0.0010     0                       NaN        NaN        NaN
  0.0010     1                  7.277497   0.5668687  6.064850
  0.0010     2                  7.314708   0.5613965  6.103794
  0.0010     3                  7.296797   0.5632505  6.083630
```

[5] The package **caret** allows us to do so.

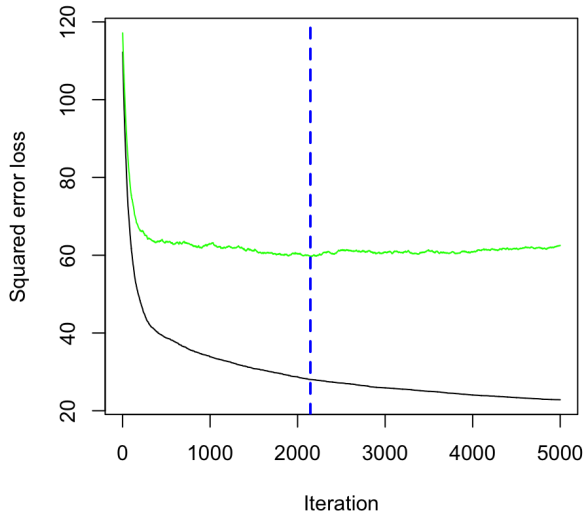

Figure 3: Improvement on (5-fold) cross-validation error as the number of trees (= stages) increases. Black line depicts error on fit, while the green line depicts error on unused data (cross-validation). Horizontal line indicates the optimal (= best comprise of runtime and precision) number of trees (which will vary if we re-run the analysis with a different seed).

```
0.0010      4                7.303803   0.5640958   6.090758
0.0010      5                7.285114   0.5656577   6.064230
0.0025      0                     NaN         NaN        NaN
0.0025      1                7.294638   0.5774026   6.038509
0.0025      2                7.296601   0.5763065   6.042342
0.0025      3                7.301065   0.5757505   6.049976
0.0025      4                7.313419   0.5713754   6.031497
0.0025      5                7.285400   0.5762384   6.013173
0.0050      0                     NaN         NaN        NaN
0.0050      1                7.454343   0.5738181   6.213360
0.0050      2                7.447417   0.5757828   6.194141
0.0050      3                7.426699   0.5751545   6.175022
0.0050      4                7.456768   0.5755451   6.184638
0.0050      5                7.434653   0.5747946   6.180953
0.0100      0                     NaN         NaN        NaN
0.0100      1                7.729304   0.5576387   6.426976
0.0100      2                7.685537   0.5656762   6.391547
0.0100      3                7.784959   0.5507302   6.481599
0.0100      4                7.685112   0.5643546   6.407061
0.0100      5                7.704323   0.5555653   6.418403
0.0500      0                     NaN         NaN        NaN
0.0500      1                8.747006   0.5020349   7.351464
0.0500      2                9.018933   0.4617087   7.599196
0.0500      3                8.857064   0.4822591   7.506450
0.0500      4                9.064949   0.4704230   7.649412
0.0500      5                8.873356   0.4902884   7.421030

Tuning parameter 'n.trees' was held constant at a value of 5000
Tuning parameter 'n.minobsinnode' was held constant at a value of 10
RMSE was used to select the optimal model using the smallest value.
The final values used for the model were n.trees = 5000, interaction.depth = 1,
shrinkage = 0.001 and n.minobsinnode = 10.
```

In this case, it suggests to use stumps (i.e. only 1 additional split) and very slow learning. Both

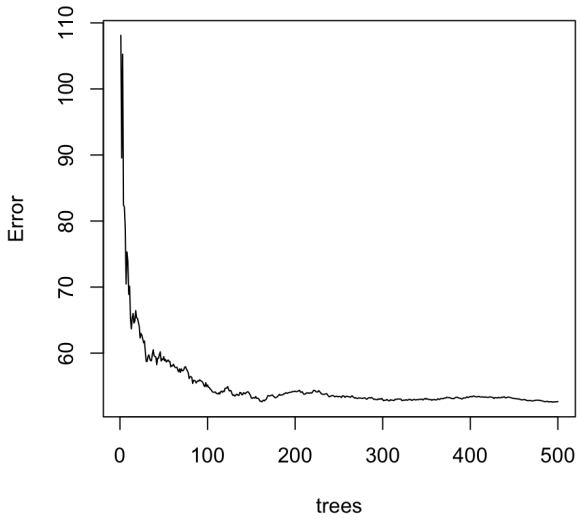

Figure 4: Improvement on out-of-bag error of random forest as the number of trees increases. Note that here no boosting takes place, i.e. improvement is merely due to averaging more and more trees fitted on a bootstrapped data set (with random predictors). Total error is lower than in the cross-validated BRT, although correlation with observed is virtually identical. The reason is that OOB-estimates are not exactly the same as actual cross-validation estimates.

`n.trees` and `n.minobsinnode` were kept constant in this example. But the output also shows that many other setting fared similarly well (in terms of RMSE). We should avoid very high learning rates (of 0.05), but even our 0.01 is not much worse than the proposed settings.

3.3 Bagging: a random forest example

When using random forest (and there are several implementations of it), we have a similiar improvement of prediction error as the number of trees grows (Fig. 4). In this case, we need far fewer trees than the BRT before, but that is an effect of the BRTs settings (increasing the learning rate would also lead to fewer trees there).

To get confidence intervals for the model predictions, we can employ an implementation of random forest that is both faster and more convinient (for error bars) than the original **randomForest** R-package: **ranger**. The result (Fig. 5) may at first glance surprise. But that is mainly because we have no culture of looking at confidence intervals of fitted data as much as we should.

References

Breiman, L. (2001) Random forests. *Machine Learning Journal*, **45**, 5–32.

Breiman, L. (1996) Bagging predictors. *Machine Learning*, **24**, 123–140.

Dormann, C.F., Elith, J., Bacher, S., Buchmann, C.M., Carl, G., Carré, G., García Marquéz, J.R., Gruber, B., Lafourcade, B., Leitão, P.J. & et al. (2013) Collinearity: a review of methods to deal with it and a simulation study evaluating their performance. *Ecography*, **36**, 27–46.

Dormann, C.F., Guillera-Arroita, G., Calabrese, J.M., Matechou, E., Bahn, V., Bartón, K., Beale, C.M., Ciuti, S., Elith, J., Gerstner, K. & et al. (2018) Model averaging in ecology:a review of bayesian, information-theoretic, machine-learning and other approaches. *Ecological Monographs*, **in review**.

Hastie, T., Tibshirani, R.J. & Friedman, J.H. (2009) *The Elements of Statistical Learning: Data Mining, Inference, and Prediction.* Springer, Berlin, 2nd edition.

Oppel, S., Powell, A.N. & Dickson, D.L. (2009) Using an algorithmic model to reveal individually variable movement decisions in a wintering sea duck. *Journal of Animal Ecology*, **78**, 524–31.

Wager, S., Hastie, T. & Efron, B. (2014) Confidence intervals for random forests: the jackknife and the infinitesimal jackknife. *Journal of Machine Learning Research*, **15**, 1625–1651.

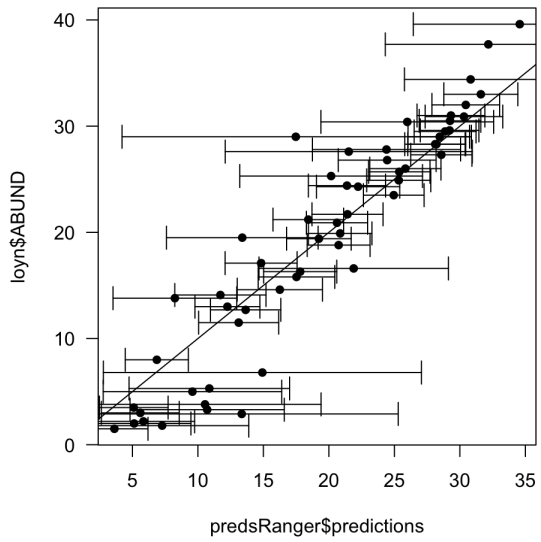

Figure 5: Random forest fits compared to observation (on y-axis). The confidence interval of each prediction is substantial, of course it is: we have only 56 data points and thus we'd expect very uncertain predictions. Thirdly, notice that almost all (95%) of the confidence intervals cross the 1:1-line. That's good! If the error bars were smaller, they would not cross the 1:1-line in the nominal 95% of cases, and then they would be wrong. Thus, when a point deviates far from the 1:1 line, it should also have a large CI, otherwise there is something wrong with the model (e.g. it is systematically biased).