# Trees Gone Wild: Random Forest and Boosting

36-600

Fall 2021

## Motivation

We have previously learned about decision trees. While trees are interpretable, they do have issues:

- They are highly variable. For instance, if you split a dataset in half and grow trees for each half, they can look very different.
- (Related:) They do not generalize as well as other models, i.e., they tend to have higher test-set MSE values.

To counteract these issues, one can utilize *bootstrap aggregation*, or *bagging*. Let's break this down:

- bootstrap: sample the training data *with replacement*
- aggregation: aggregate many trees, each constructed with a different bootstrap sample of the original training set

# Bootstrap

Here is an example of bootstrapping:

```
set.seed(101)
sample(10,10,replace=TRUE)
```

```
##  [1]  9  9  7  1 10  6  3  3  9  3
```

```
sample(10,10,replace=TRUE)
```

```
##  [1]  3  2  4  5  1  1  6  8 10  5
```

- In the first bootstrapped set, 3 and 9 are each repeated three times, while 2, 4, 5, and 8 don't appear at all.
- In the second bootstrapped set, we get two 1's and two 5's, but no 7 or 9.

The probability that a number is sampled at least once asymptotically approaches $\approx$ 63.2% as the sample size goes to infinity.

Bootstrapping is a clever way to repeat an experiment that you cannot actually repeat, and it has been shown theoretically to provide useful results. For us, that means that we can trust the output of an algorithm that utilizes bootstrapping (like, e.g., random forest).

## Bagging: Algorithm

```
Specify number of trees: n

For each tree 1->n:
  - construct a bootstrap sample from the training data
  - grow a deep and unpruned tree (i.e., overfit!)

Pass test datum through all n trees:
  - if regression: overall prediction is average of those for each tree
  - if classification: by default, predicted class for each tree is the
                       most represented class in the leaf; overall prediction
                       is majority vote
```

Bagging improves prediction accuracy at the expense of interpretability: one can "read" a single tree, but if you have 500 trees, what can you do?

$\Rightarrow$ You can use a metric dubbed *variable importance*, which is the average improvement in, e.g., RSS or Gini when splits are made on a particular predictor variable. The larger the average improvement, the more important the predictor variable.

# Random Forest: Algorithm

The random forest algorithm *is* the bagging algorithm, with a tweak.

For each bootstrapped sampled dataset, we randomly select a subset of the predictor variables, and we build the tree using only those variables. By default, $m = \sqrt{p}$. (If $m = p$, then we recover the bagging algorithm.)

$\Rightarrow$ Selecting a variable subset for each tree allows us to get around the issue that if there is a dominant predictor variable, the first split is (almost always) made on that predictor variable. (Subsetting acts to "decorrelate" the different trees.)

## Random Forest: Variable Importance

| | Regression | Classification | Preferred? |
|---|---|---|---|
| %IncMSE | YES (if importance = TRUE) | NO | ✓ |
| IncNodePurity | YES | NO | |
| MeanDecreaseGini | NO | YES | |
| MeanDecreaseAccuracy | NO | YES (if importance = TRUE) | ✓ |

To get the preferred output:

- specify `importance=TRUE` as an argument in your call to `randomForest()`
- specify `type=1` as an argument in your call to `varImpPlot()`

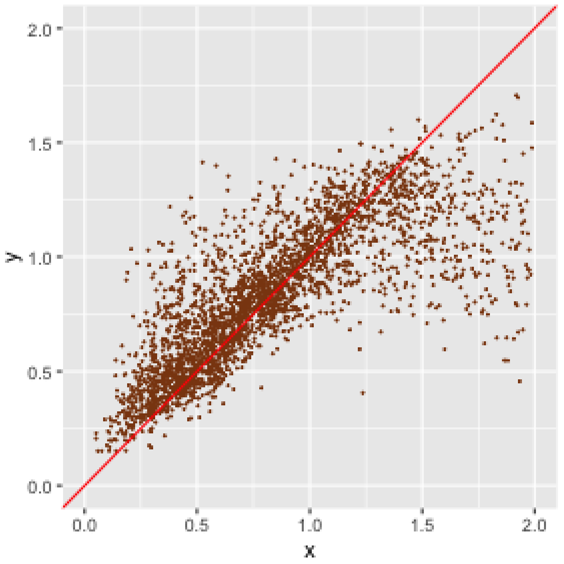
`%IncMSE` basically denotes what happens if you take a particular column of data and randomize its values (while leaving everything else the same). The more statistically informative a column's data are, the more randomization of the data will impact fitting and the more the MSE will rise. The most important variables have the highest `%IncMSE` values.

`MeanDecreaseAccuracy` is computed similarly, but for classification models. The most important variables have the highest `MeanDecreaseAccuracy` values.

# Random Forest: Regression Example

We load in 10,000 data, with 6 predictor variables and a response which denotes distance from the Earth. We then perform a 70-30 data split.

```
suppressMessages(library(tidyverse)) ; suppressMessages(library(randomForest))
rf.out = randomForest(resp.train~.,data=pred.train,importance=TRUE)
resp.pred = predict(rf.out,newdata=pred.test)
ggplot(data=data.frame("x"=resp.test,"y"=resp.pred),mapping=aes(x=x,y=y)) +
  geom_point(size=0.1,color="saddlebrown") + xlim(0,2) + ylim(0,2) +
  geom_abline(intercept=0,slope=1,color="red")
```

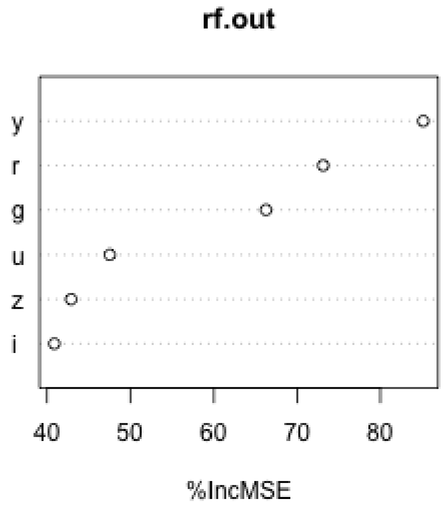


## Random Forest: Regression Example

```
round(mean((resp.pred-resp.test)^2),3)
```

```
## [1] 0.069
```

```
varImpPlot(rf.out,type=1)
```



Variable importance plots provide the extent to which we can do inference with a random forest model. The plot above indicates that y-, r-, and g-band magnitudes are the more important variables for predicting redshift…however, note the extent of the $x$ axis (the minimum value is not zero, so the i band contains far more information than you would initially think).

## Random Forest: Classification Example

Actually, no. We will use this opportunity to remind you how to extract Class 1 probabilities for the models we've looked at thus far. With this information, e.g., you can code a full random forest classifier.

- logistic regression

```
out.pred = predict(out.mod,newdata=df.test,type="response")
```

- classification tree

```
out.pred = predict(out.mod,newdata=df.test,type="prob")[,2]
```

- random forest

```
out.pred = predict(out.mod,newdata=df.test,type="prob")[,2]
```

The difference is due to one thing: logistic regression, by definition, has two response classes. Hence if you have the Class 1 probability, you know the Class 0 probability as well…so `R` just outputs a vector. The other algorithms allow for an arbitrary number of response classes. Hence `R` outputs matrices, and you have to specify the column you want (`[,1]` for Class 0, `[,2]` for Class 1, etc., etc.)

Will extreme gradient boosting have similar code? Of course not. This is `R` we're talking about.

## Boosting: Context

"Can a set of weak learners create a single strong learner?"

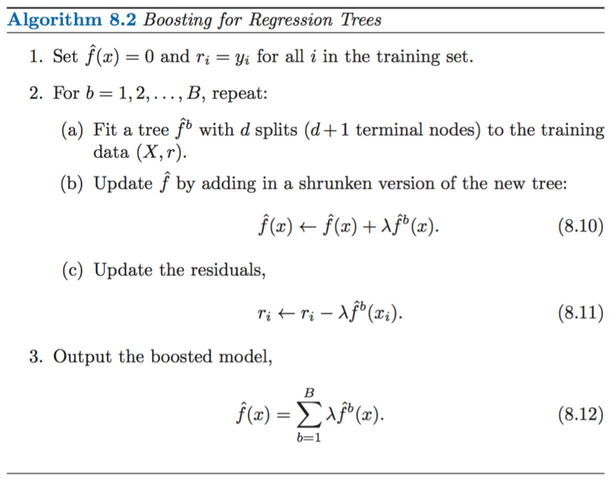
— Kearns & Valiant

An example of a "weak learner" is, e.g., a decision tree with a single split (i.e., a "decision stump"). The "set of weak learners" is, e.g., the repeated generation of stumps given some iterative rule, such as "let's upweight the currently misclassified observations next time around." (This is the core of the AdaBoost algorithm.) Through iteration, a strong learner is created.

Boosting is a so-called "meta-algorithm": it is an algorithm that dictates how to repeatedly apply another algorithm. As such, boosting can be applied with many models, like linear regression. However, boosting is most associated with trees.

There are also many different kinds of boosting (i.e., many different ways to define the meta-algorithm: upweight observations the next time, etc.). The most oft-used boosting algorithm currently is *gradient boosting*, which we describe on the next slide.

# Gradient Boosting



**Algorithm 8.2** *Boosting for Regression Trees*

```
1. Set $\hat{f}(x) = 0$ and $r_i = y_i$ for all $i$ in the training set.
2. For $b = 1, 2, \ldots, B$, repeat:
    (a) Fit a tree $\hat{f}^b$ with $d$ splits ($d+1$ terminal nodes) to the training
        data $(X, r)$.
    (b) Update $\hat{f}$ by adding in a shrunken version of the new tree:
            $\hat{f}(x) \leftarrow \hat{f}(x) + \lambda \hat{f}^b(x).$        (8.10)
    (c) Update the residuals,
            $r_i \leftarrow r_i - \lambda \hat{f}^b(x_i).$        (8.11)
3. Output the boosted model,
            $\hat{f}(x) = \sum_{b=1}^{B} \lambda \hat{f}^b(x).$        (8.12)
```

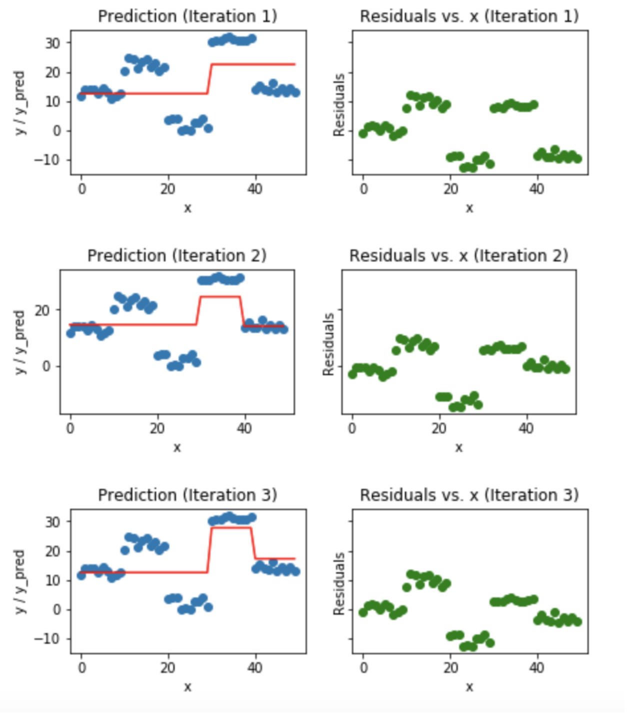
(Algorithm 8.2, *Introduction to Statistical Learning* by James et al.)

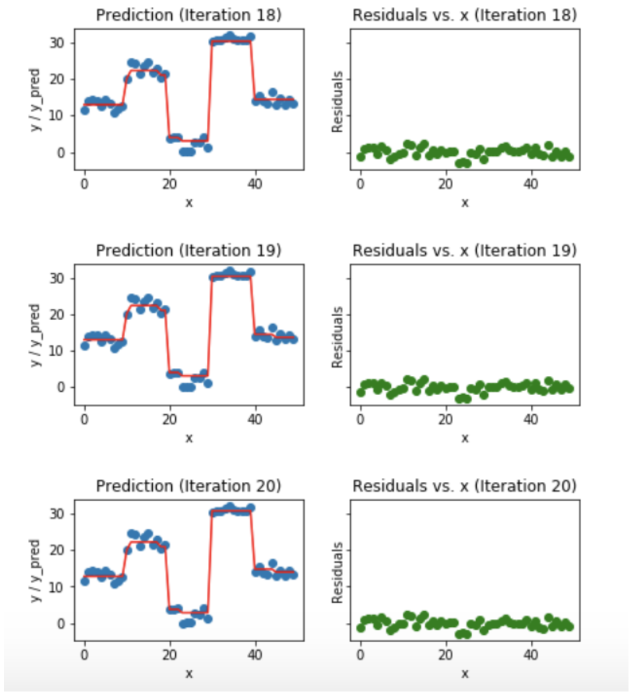
The core idea of regression tree boosting: it *slowly* learns a model by fitting the *residuals* of the previous fit. In other words, it fits a stumpy tree to the original data, shrinks that tree (note the $\lambda$ parameter), updates the residuals, sets the data to be those residuals, and repeats. Hence each iteration of boosting attempts *to hone in on those data that were not well fit previously*, i.e., those data for which the residual values $r_i$ continue to be large.

The smaller the value of $\lambda$, the more slowly and conservatively the final tree is grown.

The contrast with bagging is that bagging involves growing many separate deep trees that are aggregated, while boosting grows one tree sequentially by adding a weighted series of stumps: it is bonsai rather than a forest.

# Gradient Boosting





(These pictures were taken from this web page.)

# Gradient Boosting: Regression Example

We apply extreme gradient boosting, or "xgboost," to the same dataset as above. Note: the function calls given below "look weird." That's because the `xgboost` package developers didn't bother to try to at least approximately match `R` modeling function syntax. Also...**IMPORTANT**...`xgboost` works only with continuous (or quantitative) predictor variables!

```r
library(xgboost)
train = xgb.DMatrix(data=as.matrix(pred.train),label=resp.train)
test  = xgb.DMatrix(data=as.matrix(pred.test),label=resp.test)
set.seed(101)
xgb.cv.out = xgb.cv(params=list(objective="reg:squarederror"),train,nrounds=30,nfold=5,verbose=0)
cat("The optimal number of trees is ",which.min(xgb.cv.out$evaluation_log$test_rmse_mean),"\n")
```

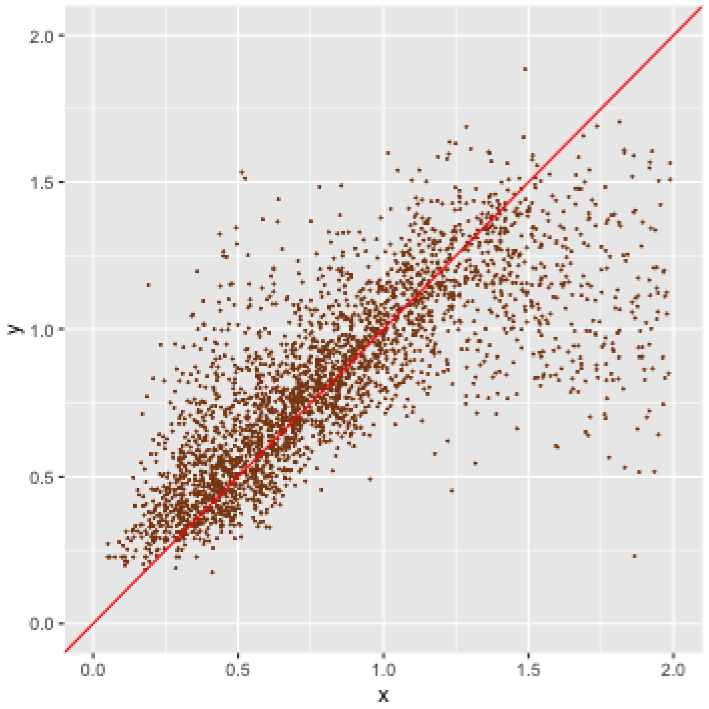
```
## The optimal number of trees is  26
```

```r
xgb.out = xgboost(train,nrounds=which.min(xgb.cv.out$evaluation_log$test_rmse_mean),
                  params=list(objective="reg:squarederror"),verbose=0)
resp.pred = predict(xgb.out,newdata=test)
round(mean((resp.pred-resp.test)^2),3)
```

```
## [1] 0.075
```

# Gradient Boosting: Regression Example

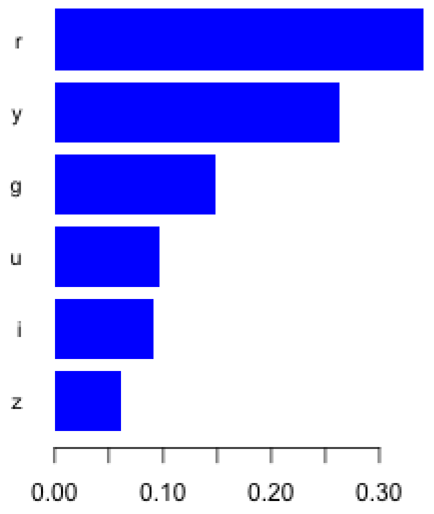
```r
ggplot(data=data.frame("x"=resp.test,"y"=resp.pred),mapping=aes(x=x,y=y)) +
  geom_point(size=0.1,color="saddlebrown") + xlim(0,2) + ylim(0,2) +
  geom_abline(intercept=0,slope=1,color="red")
```



## Gradient Boosting: Regression Example

Like random forest, boosting allows one to measure variable importance. This importance measure is called the "gain," defined as the "fractional contribution of each feature to the model based on the total gain of this feature’s splits. Higher percentage means a more important predictive feature." (Definition from `xgboost` documentation.)

```
imp.out = xgb.importance(model=xgb.out)
xgb.plot.importance(importance_matrix=imp.out,col="blue")
```



## Gradient Boosting: Classification Example

How to alter your code for classification:

- When calling `xgb.cv()` and `xgboost()`, change `objective="reg:squarederror"` to `objective="binary:logistic"`
- When calling `xgboost()`, change the string `evaluation_log$test_rmse_mean` to `evaluation_log$test_error_mean`
- When calling `xgb.cv()` and `xgboost()`, add the argument `eval_metric="error"`