Random Forest and Boosting

36-600

Motivation

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

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
```

- Above is an example of bootstrapping...think of it as randomly selecting rows of a ten-row dataset, with replacement
  - in the first example, 3 and 9 are each repeated three times, while 2, 4, 5, and 8 don't appear at all
  - in the second example, we get two 1's and two 5's, but no 7 or 9
  - the probability that a number in the set $[1, 2, \ldots, n]$ is sampled at least once asymptotically approaches $\approx$ 63.2% as the sample size n 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

Bagging: Algorithm

- The bagging, or bootstrap aggregation, algorithm may be written down as follows:

```
Specify number of trees: k

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

Pass a test datum through all k trees:
  - if regression: the prediction is average of those observed for each tree
  - if classification: by default, predicted class for each tree is the
                       most-represented class in the leaf; overall prediction
                       is the majority vote (but we can override this!)
```

- Bagging improves prediction accuracy at the expense of interpretability
  - one can "read" a single tree, but if you have, say, 500 trees, what can you do?
- *Variable importance* is a metric that represents the average improvement in, e.g., RSS or the Gini index 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
- 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 thus acts to "decorrelate" the different trees

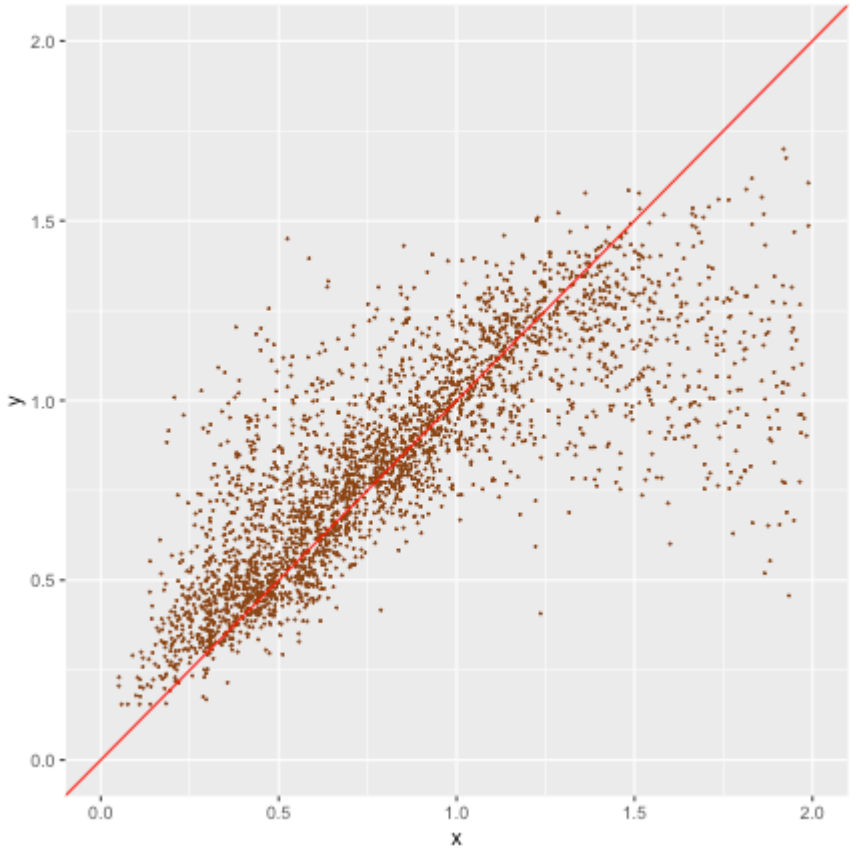
Random Forest: Variable Importance

- The preferred measures of variable importance are `%IncMSE` for regression, and `MeanDecreaseAccuracy` for classification
- To observe 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` denotes what happens if you take a 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...thus 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
  - (note that "accuracy" is simply the opposite of "misclassification")

Random Forest: Regression Example

- We load in a dataset with 10,000 rows, 6 predictor variables and a response (`redshift`) which denotes distance from the Earth, and we perform a 70-30 data split
- We learn a random-forest model:

```
suppressMessages(library(tidyverse))
suppressMessages(library(randomForest))
rf.out = randomForest(redshift~.,data=df.train,
                      importance=TRUE)
resp.pred = predict(rf.out,newdata=df.test)
ggplot(data=data.frame("x"=df.test$redshift,
                       "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")
```

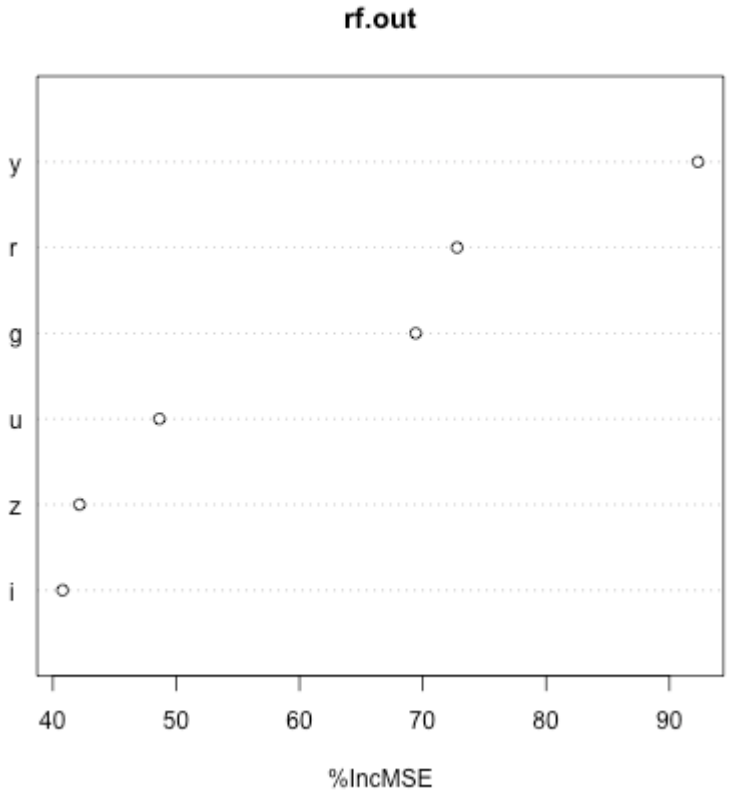


Random Forest: Regression Example

- Variable importance plots provide the extent to which we can do inference with a random forest model
- The plot to the right indicates that y-, r-, and g-band magnitudes are the more important variables for predicting redshift...however, note the x-axis limits!

```
round(mean((resp.pred-df.test$redshift)^2),3)
varImpPlot(rf.out,type=1)
```

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



Random Forest: Prediction in (Binary) Classification Models

- We will use this opportunity to remind you how to extract Class 1 probabilities for the models we've looked at thus far
- 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]
```

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"
  - through iteration, a strong learner is created
- *Boosting* is a so-called "meta-algorithm": it dictates how to repeatedly apply another algorithm
  - boosting can thus be applied to many models, like linear regression, but it is most associated with trees
- There are also many different kinds of boosting (i.e., many different ways to define the meta-algorithm)
  - the most popular boosting algorithm currently is *extreme-gradient boosting*

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:

$$\hat{f}(x) \leftarrow \hat{f}(x) + \lambda \hat{f}^b(x). \tag{8.10}$$

   (c) Update the residuals,

$$r_i \leftarrow r_i - \lambda \hat{f}^b(x_i). \tag{8.11}$$

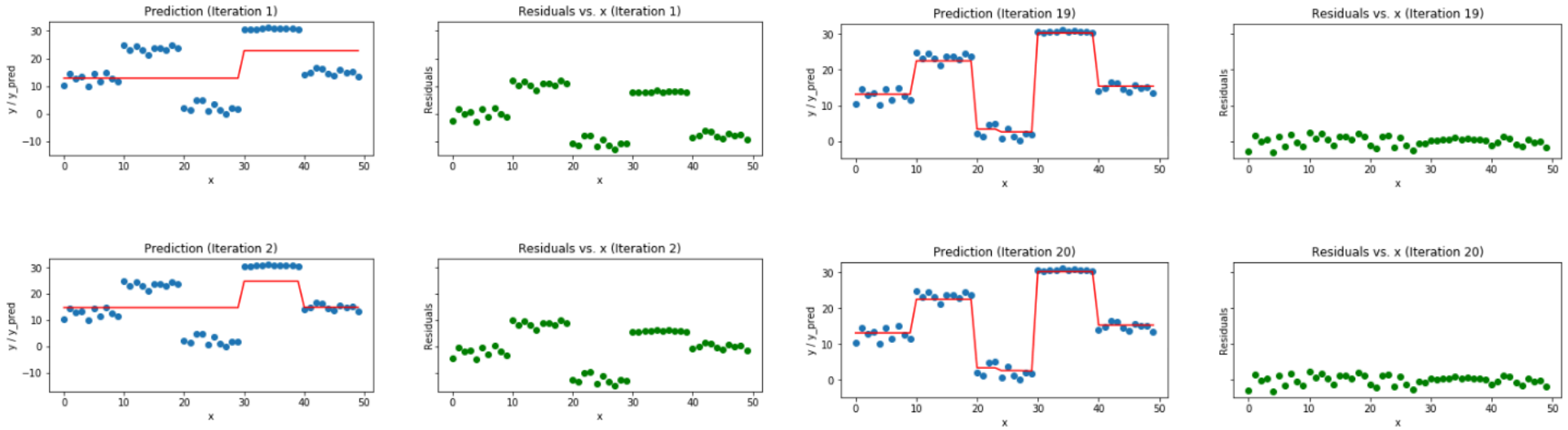
3. Output the boosted model,

$$\hat{f}(x) = \sum_{b=1}^{B} \lambda \hat{f}^b(x). \tag{8.12}$$

(Algorithm 8.2, *Introduction to Statistical Learning* by James et al. Note that this is *not* specifically extreme-gradient boosting.)

- The core idea of boosting is that it *slowly* learns a model by fitting the model *residuals* from the previous fit
  - 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 λ, 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
  - boosting is bonsai rather than a forest

Gradient Boosting



(See 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" because the `xgboost` package developers didn't bother to try to at least approximately match `R` modeling function syntax
- **IMPORTANT**...off-the-shelf `xgboost` currently works only with continuous (or quantitative) predictor variables!

```
set.seed(101)
suppressMessages(library(xgboost))
train      <- xgb.DMatrix(data=as.matrix(df.train[,1:6]),label=df.train$redshift)
test       <- xgb.DMatrix(data=as.matrix(df.test[,1:6]),label=df.test$redshift)
xgb.cv.out <- xgb.cv(params=list(objective="reg:squarederror"),train,nrounds=30,nfold=5,verbose=0)
rmse.min   <- xgb.cv.out$evaluation_log$test_rmse_mean
cat("The optimal number of trees is ",which.min(rmse.min),"\n")
```

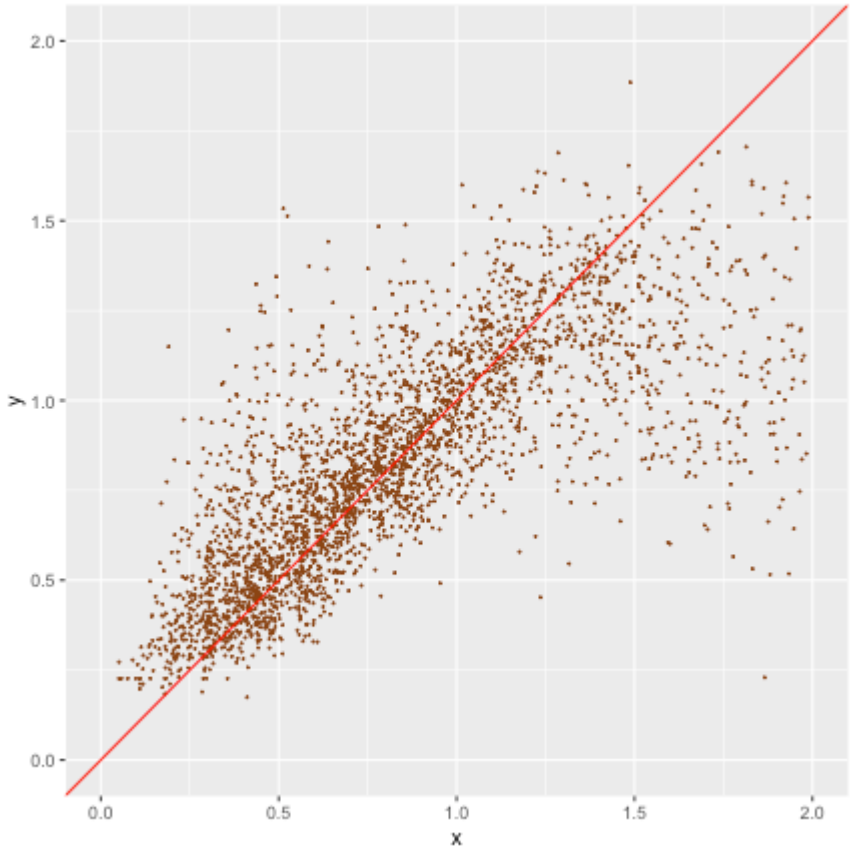
```
## The optimal number of trees is  26
```

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

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

Gradient Boosting: Regression Example

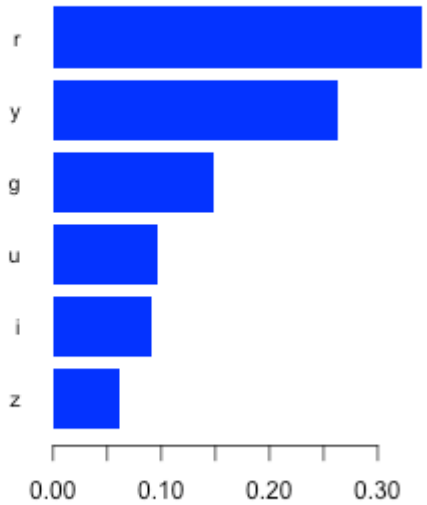
```
ggplot(data=data.frame("x"=df.test$redshift,
                       "y"=resp.pred),
       mapping=aes(x=x,y=y)) +
  geom_point(size=0.1,color="saddlebrown") +
  xlim(0,2) + ylim(0,2) +
  geom_abline(intercept=0,slope=1,color="red")
```



Gradient Boosting: Regression Example

- Like random forest, boosting allows one to measure variable importance
    - the *gain* is 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."

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



Gradient Boosting: Prediction in (Binary) Classification Models

- 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"`