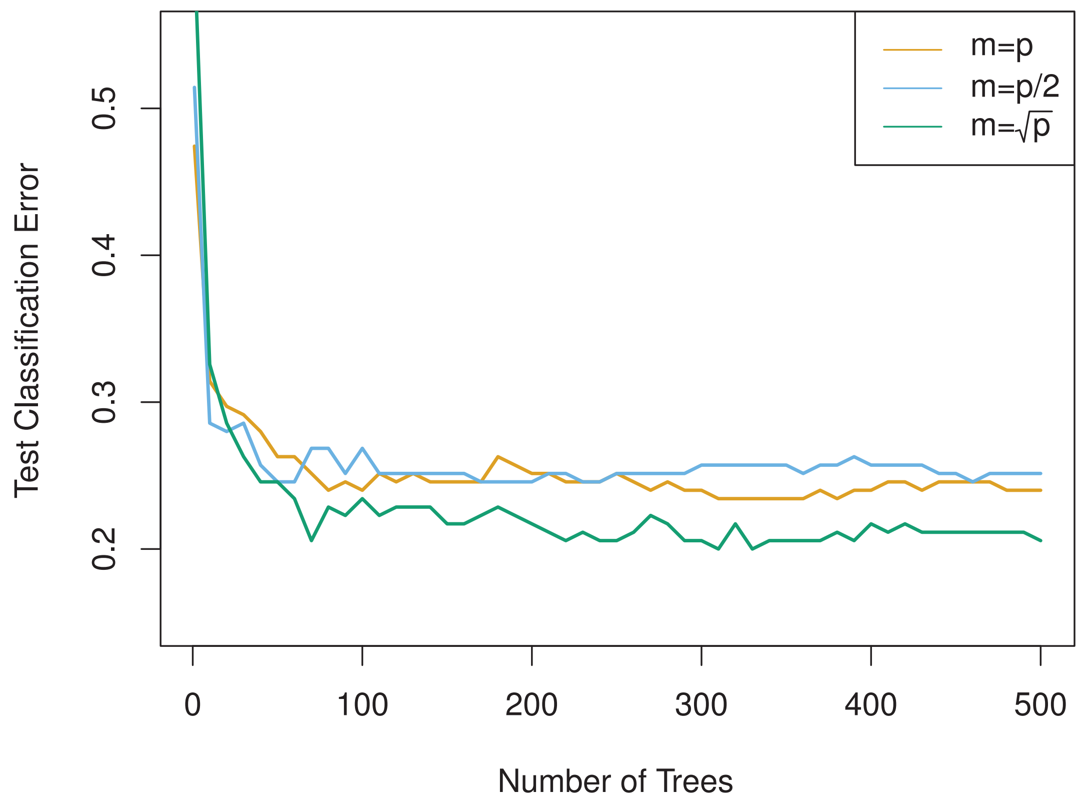
Random Forests

- Random forests provide an improvement over bagged trees by way of a small tweak that decorrelates the trees.
- As in bagging, we build a number of decision trees on bootstrapped training samples. But when building these decision trees, each time a split in a tree is considered, a random sample of m predictors is chosen as split candidates from the full set of p predictors.
- The split is allowed to use only one of those m predictors. A fresh sample of m predictors is taken at each split, and typically we choose $m \approx \sqrt{p}$.
- The main difference between bagging and random forests is the choice of predictor subset size m.

If a random forest is built using $m=p$, then this amounts simply to bagging.

- Using a small value of m in building a random forest will typically be helpful when we have a large number of correlated predictors.
- As with bagging, random forests will not overfit if we increase B, so in practice we use a value of B sufficiently large for the error rate to have settled down.



Results from random forests for the 15-class gene expression data set with $p = 500$ predictors. The test error is displayed as a function of the number of trees. Each colored line corresponds to a different value of m, the number of predictors available for splitting at each interior tree node. Random forests ($m < p$) lead to a slight improvement over bagging ($m = p$). A single classification tree has an error rate of 45.7%.

Performing Bagging and Random Forests in R

Here we apply bagging and random forests to the Hitters data, using the randomForest package in R. Recall that bagging is simply a special case of a random forest with $m = p$. Therefore, the randomForest() function can be used to perform both random forests and bagging. We perform bagging as follows:

```
> library(randomForest)
> set.seed(1)
> bag_hitters <- randomForest(Log_Salary ~ .- Salary, data = Hitters,
   subset = train, mtry = 19, importance = TRUE)
> bag_hitters

Call:
 randomForest(formula = Log_Salary ~ . - Salary, data = Hitters,
    mtry = 19, importance = TRUE, subset = train)
               Type of random forest: regression
                     Number of trees: 500
No. of variables tried at each split: 19

          Mean of squared residuals: 0.2052503
                    % Var explained: 73.54
```

The argument mtry = 19 indicates that all 19 predictors should be considered for each split of the tree—in other words, that bagging should be done. How well does this bagged model perform on the test set?

```
> yhat_bag <- predict(bag_hitters, newdata = Hitters[-train, ])
> mean((yhat_bag - Log_Salary_test)^2)
[1] 0.190633
```

The test set MSE associated with the bagged regression tree is 0.19. We could change the number of trees grown by randomForest() using the ntree argument:

```
> bag_hitters <- randomForest(Log_Salary ~ .- Salary, data = Hitters,
   subset = train, mtry = 19, ntree = 100)
> yhat_bag <- predict(bag_hitters, newdata = Hitters[-train, ])
> mean((yhat_bag - Log_Salary_test)^2)
[1] 0.1914903
```

Growing a random forest proceeds in exactly the same way, except that we use a smaller value of the mtry argument. By default, randomForest() uses $p/3$

variables when building a random forest of regression trees, and $\sqrt{p}$ variables when building a random forest of classification trees. Here we use mtry = 9.

```
> set.seed(1)
> rf_hitters <- randomForest(Log_Salary ~ .- Salary, data = Hitters,
   subset = train, mtry = 9, importance = TRUE)
> yhat_rf <- predict(bag_hitters, newdata = Hitters[-train, ])
> mean((yhat_rf - Log_Salary_test)^2)
[1] 0.1864585
```

The test set MSE is 0.1864; this indicates that random forests yielded an improvement over bagging in this case.

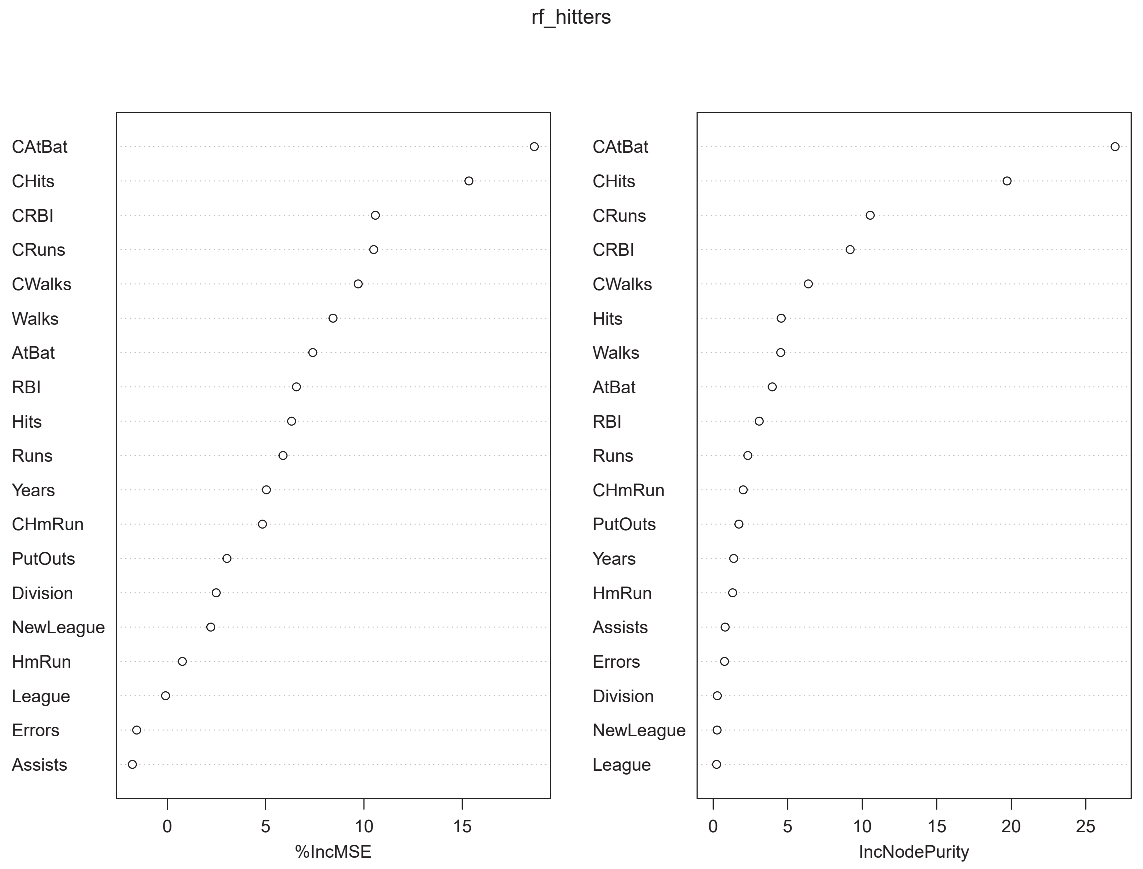
Using the importance() function, we can view the importance of each variable.

```
> importance(rf_hitters)
              %IncMSE IncNodePurity
AtBat       7.39691621     3.9659015
Hits        6.31787642     4.5657460
HmRun       0.75996667     1.3053628
Runs        5.88199191     2.3246396
RBI         6.56497811     3.0876107
Walks       8.42641065     4.5315977
Years       5.03743504     1.3763260
CAtBat     18.66803325    26.9664695
CHits      15.34037858    19.7202614
CHmRun      4.83630562     2.0156468
CRuns      10.49703146    10.5368887
CRBI       10.58221268     9.1893571
CWalks      9.70825005     6.3857389
League     -0.09445326     0.2290954
Division    2.48325541     0.2774326
PutOuts     3.02902964     1.7236172
Assists    -1.78189591     0.8043313
Errors     -1.56458058     0.7615783
NewLeague   2.20634407     0.2627862
```

Two measures of variable importance are reported. The first is based upon the mean decrease of accuracy in predictions on the out of bag samples. The second is a measure of the total decrease in node impurity that results from splits over that variable, averaged over all trees

Plots of these importance measures can be produced using the varImpPlot() function.

```
> varImpPlot(rf_hitters)
```



The results indicate that across all of the trees considered in the random forest, the number of times at bat during his career (CAtBat) and the number of hits during his career (CHits) are by far the two most important variables.

Boosting

- Like bagging, boosting is a general approach that can be applied to many statistical learning methods for regression or classification.
- Recall that bagging involves creating multiple copies of the original training data set using the bootstrap, fitting a separate decision tree to each copy, and then combining all of the trees in order to create a single predictive model.
- Notably, each tree is built on a bootstrap data set, independent of the other trees.
- Boosting works in a similar way, except that the trees are grown sequentially: each tree is grown using information from previously grown trees.
- Boosting does not involve bootstrap sampling; instead each tree is fit on a modified version of the original data set.

Boosting for Regression Trees:

1. Set $\hat{f}(x) = 0$ and $r_i = y_i$ for all i in the training set.
2. For $b = 1, 2, ..., B$, 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:
   $$\hat{f}(x) \leftarrow \hat{f}(x) + \lambda \hat{f}^b(x)$$
   (c) Update the residuals,
   $$r_i \leftarrow r_i - \lambda \hat{f}^b(x_i)$$

3. Output the boosted model,

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

What is the idea behind this procedure?

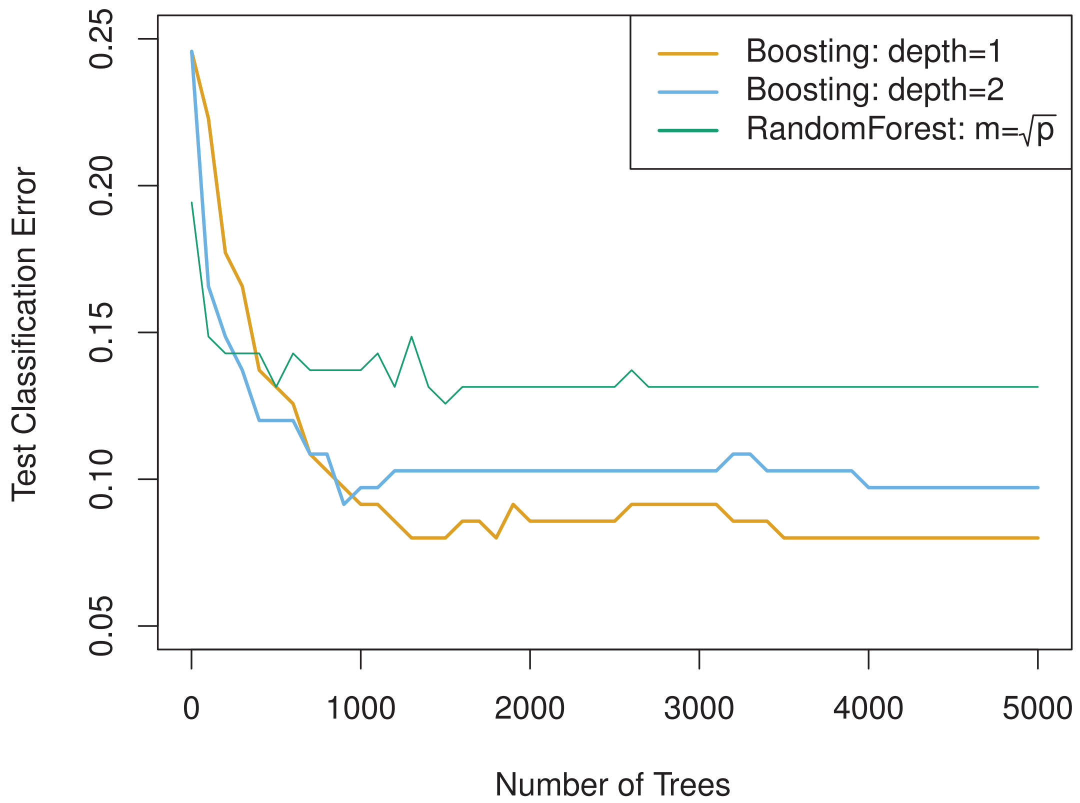
- Unlike fitting a single large decision tree to the data, which amounts to fitting the data hard and potentially overfitting, the boosting approach instead learns slowly.
- Given the current model, we fit a decision tree to the residuals from the model. We then add this new decision tree into the fitted function in order to update the residuals.
- Each of these trees can be rather small, with just a few terminal nodes, determined by the parameter d in the algorithm. By fitting small trees to the residuals, we slowly improve $\hat{f}$ in areas where it does not perform well.
- The shrinkage parameter λ slows the process down even further, allowing more and different shaped trees to attack the residuals.

Boosting has three tuning parameters:

1. The number of trees B. Unlike bagging and random forests, boosting can overfit if B is too large, although this overfitting tends to occur slowly if at all. We use cross-validation to select B.
2. The shrinkage parameter λ, a small positive number. This controls the rate at which boosting learns. Very small λ can require using a very large value of B in order to achieve good performance. Typical values are 0.01 or 0.001, and the right choice can depend on the problem.

3. The number d of splits in each tree, which controls the complexity of the boosted ensemble. Often $d = 1$ works well, in which case each tree is a stump, consisting of a single split. In this case, the boosted ensemble is fitting an additive model, since each term involves only a single variable. More generally d is the interaction depth, and controls the interaction order of the boosted model, since d splits can involve at most d variables.

Note: Boosting classification trees proceeds in a similar but slightly more complex way.



Results from performing boosting and random forests on the 15-class gene expression data set in order to predict cancer versus normal. The test error is displayed as a function of the number of trees. For the two boosted models, $\lambda = 0.01$. Depth-1 trees slightly outperform depth-2 trees, and both outperform the random forest. The test error rate for a single tree is 24%.

Performing Boosting in R

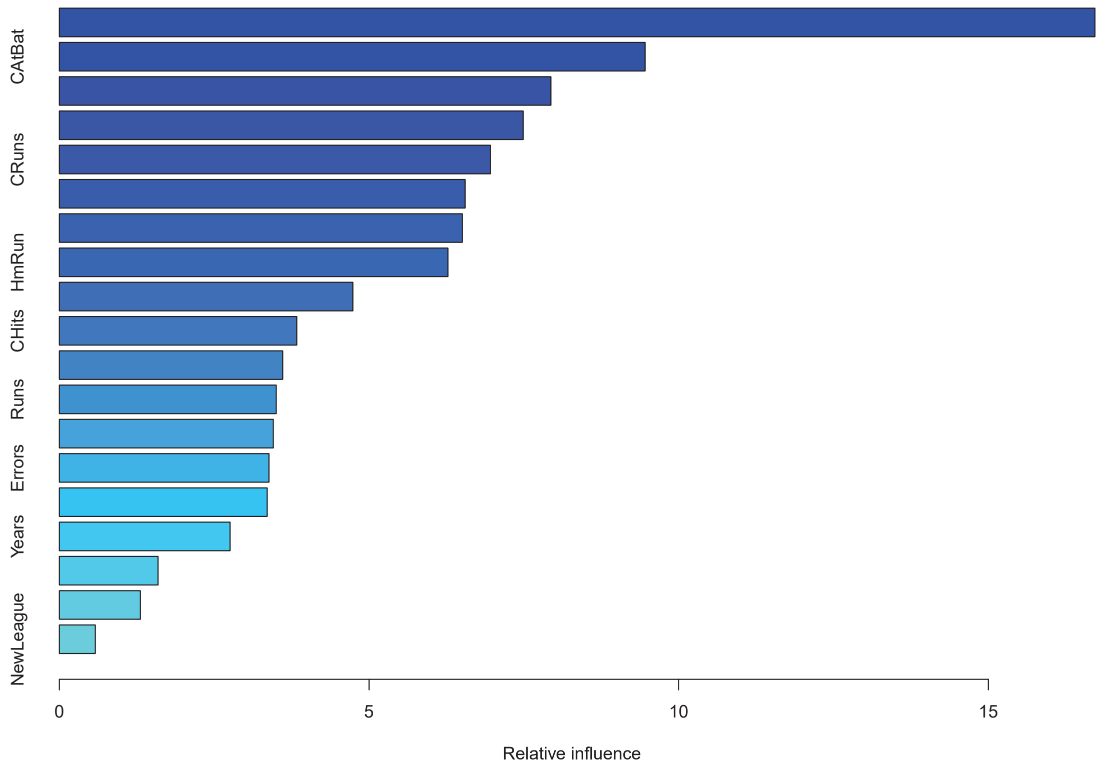
Here we use the gbm package, and within it the gbm() function, to fit boosted regression trees to the Hitters data set. We run gbm() with the option distribution = ”gaussian” since this is a regression problem; if it were a binary classification problem, we would use distribution = ”bernoulli”. The argument n.trees = 1000 indicates that we want 1000 trees, and the option interaction.depth = 4 limits the depth of each tree.

```
> set.seed(1)
> Hitters_1 <- data.frame(Hitters, Log_Salary)
   #adds Log_Salary to the Hitters dataset
> boost_hitters <- gbm(Log_Salary ~ . -Salary, data = Hitters_1[train,],
   distribution = "gaussian", n.trees = 1000, interaction.depth = 4)
```

The summary() function produces a relative influence plot and also outputs the relative influence statistics.

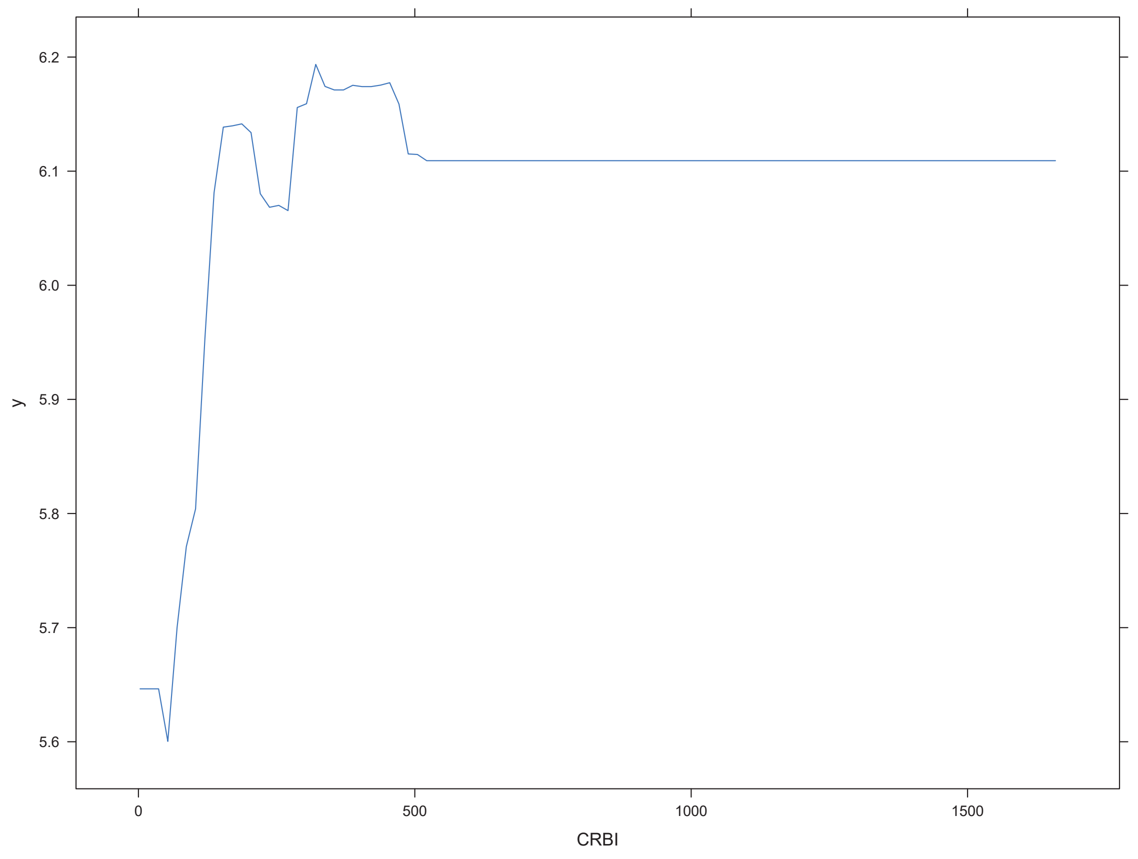
```
> summary(boost_hitters)
                var    rel.inf
CRBI           CRBI 16.7207391
CAtBat       CAtBat  9.4574716
PutOuts     PutOuts  7.9369647
Walks         Walks  7.4876860
CRuns         CRuns  6.9584283
Assists     Assists  6.5507739
CWalks       CWalks  6.5050494
HmRun         HmRun  6.2739462
CHmRun       CHmRun  4.7390139
CHits         CHits  3.8335119
Hits           Hits  3.6066876
Runs           Runs  3.5007935
RBI             RBI  3.4535948
Errors       Errors  3.3839989
AtBat         AtBat  3.3541106
Years         Years  2.7554308
League       League  1.5925953
Division   Division  1.3084273
```

```
NewLeague NewLeague  0.5807762
```



We see that the number of runs batted in during his career (CRBI) is by far the most important variable. We can also produce partial dependence plots for variables. These plots illustrate the marginal effect of the selected variables on the response after integrating out the other variables.

```
plot(boost_hitters, i = "CRBI")
```



We now use the boosted model to predict log Salary on the test set

```
> yhat_boost <- predict(boost_hitters, newdata = Hitters_1[-train,],
   n.trees = 1000)
> mean((yhat_boost - Log_Salary_test)^2)
[1] 0.2447526
```

If we want to, we can perform boosting with a different value of the shrinkage parameter λ. The default value is 0.001, but this is easily modified.

```
> boost_hitters <- gbm(Log_Salary ~ . -Salary, data = Hitters_1[train,],
   distribution = "gaussian", n.trees = 1000, shrinkage = 0.01)
> yhat_boost <- predict(boost_hitters, newdata = Hitters_1[-train,],
   n.trees = 1000)
```

```
> mean((yhat_boost - Log_Salary_test)^2)
[1] 0.209225
```

In this case, using $\lambda = 0.01$ leads to a lower test MSE than $\lambda = 0.001$.

Bayesian Additive Regression Trees

- *Bayesian additive regression trees* (BART), is another ensemble method that uses decision trees as its building blocks.
- BART is related to bagging, random forests, and boosting approaches: each tree is constructed in a random manner as in bagging and random forests, and each tree tries to capture signal not yet accounted for by the current model, as in boosting.
- The main novelty in BART is the way in which new trees are generated.
- We let K denote the number of regression trees, and B the number of iterations for which the BART algorithm will be run.
- The notation $\hat{f}_k^b(x)$ represents the prediction at x for the kth regression tree used in the bth iteration. At the end of each iteration, the K trees from that iteration will be summed, i.e. $\hat{f}^b(x) = \sum_{k=1}^K \hat{f}_k^b(x)$ for $b = 1, ..., B$.
- In the first iteration of the BART algorithm, all trees are initialized to have a single root node, with $\hat{f}_k^1(x) = \frac{1}{nK} \sum_{i=1}^n y_i$, the mean of the response values divided by the total number of trees. Thus, $\hat{f}^1(x) = \sum_{k=1}^K \hat{f}_k^1(x) = \frac{1}{n} \sum_{i=1}^n y_i$.
- In subsequent iterations, BART updates each of the K trees, one at a time. In the bth iteration, to update the kth tree, we subtract from each response value the predictions from all but the kth tree, in order to obtain a partial residual

$$r_i = y_i - \sum_{k'<k} \hat{f}_{k'}^b(x_i) - \sum_{k'>k} \hat{f}_{k'}^{b-1}(x_i)$$