Random Forests and Boosted Trees

Prof Wells

STA 295: Stat Learning

April 23rd, 2024

Outline

- Discuss random forests and boosted trees as methods for reducing variance in decision trees
- Implement random forests and boosted trees in R

Section 1

Random Forests

Review: Bagging

- To create a bagged model, first create many bootstrap samples from the original training set (i.e. sample with replacement to create a sample of same size as original)
    - Then fit a decision tree to each bootstrap sample. Average the resulting predictions to get the bagged prediction.
    - Unlike a single tree model, we do not prune trees in bagged models.

Review: Bagging

- To create a bagged model, first create many bootstrap samples from the original training set (i.e. sample with replacement to create a sample of same size as original)
  - Then fit a decision tree to each bootstrap sample. Average the resulting predictions to get the bagged prediction.
  - Unlike a single tree model, we do not prune trees in bagged models.
- Single (full) decision trees tend to have high variance but low bias. While single (pruned) decision trees tend to have lower variance but higher bias.

Review: Bagging

- To create a bagged model, first create many bootstrap samples from the original training set (i.e. sample with replacement to create a sample of same size as original)
  - Then fit a decision tree to each bootstrap sample. Average the resulting predictions to get the bagged prediction.
  - Unlike a single tree model, we do not prune trees in bagged models.
- Single (full) decision trees tend to have high variance but low bias. While single (pruned) decision trees tend to have lower variance but higher bias.
- However, in bagged trees, since we average large number of (full) decision trees, we reduce variance
  - Additionally, averaging models with low bias will produce a model with low bias

Review: Bagging

- To create a bagged model, first create many bootstrap samples from the original training set (i.e. sample with replacement to create a sample of same size as original)
  - Then fit a decision tree to each bootstrap sample. Average the resulting predictions to get the bagged prediction.
  - Unlike a single tree model, we do not prune trees in bagged models.
- Single (full) decision trees tend to have high variance but low bias. While single (pruned) decision trees tend to have lower variance but higher bias.
- However, in bagged trees, since we average large number of (full) decision trees, we reduce variance
  - Additionally, averaging models with low bias will produce a model with low bias
- This a a rare case in stat learning in which there is no bias-variance trade-off. Bagged trees allow us to reduce variance with no increase in bias!

Further Performance Improvements

Suppose we have *m* ensemble models built from the same data set and that it turns out that all *m* models are very similar.

Further Performance Improvements

Suppose we have m ensemble models built from the same data set and that it turns out that all m models are very similar.

- Do we expect the ensemble model to have high or low variance?

Further Performance Improvements

Suppose we have m ensemble models built from the same data set and that it turns out that all m models are very similar.

- Do we expect the ensemble model to have high or low variance?
  - High variance (since the models are very correlated)

Further Performance Improvements

Suppose we have *m* ensemble models built from the same data set and that it turns out that all *m* models are very similar.

- Do we expect the ensemble model to have high or low variance?
    - High variance (since the models are very correlated)
- When bagging trees, if one predictor accounts for large amount of deviation in the response, it will usually be selected as the first split (regardless of the bootstrap sample used)

Further Performance Improvements

Suppose we have m ensemble models built from the same data set and that it turns out that all m models are very similar.

- Do we expect the ensemble model to have high or low variance?
  - High variance (since the models are very correlated)
- When bagging trees, if one predictor accounts for large amount of deviation in the response, it will usually be selected as the first split (regardless of the bootstrap sample used)
- To artificially increase the variety among trees, we randomly restrict which predictors can be used at each split point.

Further Performance Improvements

Suppose we have *m* ensemble models built from the same data set and that it turns out that all *m* models are very similar.

- Do we expect the ensemble model to have high or low variance?
    - High variance (since the models are very correlated)
- When bagging trees, if one predictor accounts for large amount of deviation in the response, it will usually be selected as the first split (regardless of the bootstrap sample used)
- To artificially increase the variety among trees, we randomly restrict which predictors can be used at each split point.
- Although counterintuitive, this restriction tends to increase accuracy of the ensemble by breaking correlations among the participant trees

Random Forests

To create a random forest:

1. Select the number of models m to build and a number of predictors k to use at each step t
2. Generate a bootstrap sample for each model
3. Build a tree on the bootstrap sample where at each step, a random selection of k of the p predictors can be used (independent of prior predictors selected)
4. Aggregate the models to create an ensemble model.

Random Forests

To create a random forest:

1. Select the number of models m to build and a number of predictors k to use at each step t
2. Generate a bootstrap sample for each model
3. Build a tree on the bootstrap sample where at each step, a random selection of k of the p predictors can be used (independent of prior predictors selected)
4. Aggregate the models to create an ensemble model.

Advantages of the random forest?

Random Forests

To create a random forest:

1. Select the number of models m to build and a number of predictors k to use at each step t
2. Generate a bootstrap sample for each model
3. Build a tree on the bootstrap sample where at each step, a random selection of k of the p predictors can be used (independent of prior predictors selected)
4. Aggregate the models to create an ensemble model.

Advantages of the random forest?

- Individual models are less correlated, so ensemble has lower variance
- Each tree is quicker to build (why?)

Random Forests

To create a random forest:

1. Select the number of models m to build and a number of predictors k to use at each step t
2. Generate a bootstrap sample for each model
3. Build a tree on the bootstrap sample where at each step, a random selection of k of the p predictors can be used (independent of prior predictors selected)
4. Aggregate the models to create an ensemble model.

Advantages of the random forest?

- Individual models are less correlated, so ensemble has lower variance
- Each tree is quicker to build (why?)

Disadvantages?

Random Forests

To create a random forest:

1. Select the number of models m to build and a number of predictors k to use at each step t
2. Generate a bootstrap sample for each model
3. Build a tree on the bootstrap sample where at each step, a random selection of k of the p predictors can be used (independent of prior predictors selected)
4. Aggregate the models to create an ensemble model.

Advantages of the random forest?

- Individual models are less correlated, so ensemble has lower variance
- Each tree is quicker to build (why?)

Disadvantages?

- Difficult to interpret
- Theoretically properties less well-studied (possible MAP project!)

Hand-made Random Forests

I have a data set of 50 observations on a binary response Y and 3 quantitative predictors.

- Our goal is to build, as a class, a random forest for predicting Y.
- Each table will be tasked with building (by hand) a single decision tree for predicting Y.
- Each table will be randomly assigned 2 of the 3 predictors, and will have a bootstrap sample of the 50 observations.
- Each table will be given a scatterplot showing the relationship between their 2 predictors and the response, on their bootstrap sample.
- Each table should work together to decide where to make cuts in the scatterplot to create a decision tree with between 3 and 6 leaves (group's choice)
- I will give each group the same 10 test points to classify. And as a class, we will average the predictions to create a random forest prediction.

Section 2

Bagging and Random Forests in R

A Forest of Trees

We return to the pdxTrees data set, this time expanding both our data set size and number of predictors:

```r
names(my_pdxTrees)
```

```
## [1] "DBH"                        "Condition"
## [3] "Tree_Height"                "Crown_Width_NS"
## [5] "Crown_Width_EW"             "Crown_Base_Height"
## [7] "Functional_Type"            "Mature_Size"
## [9] "Carbon_Sequestration_lb"
```

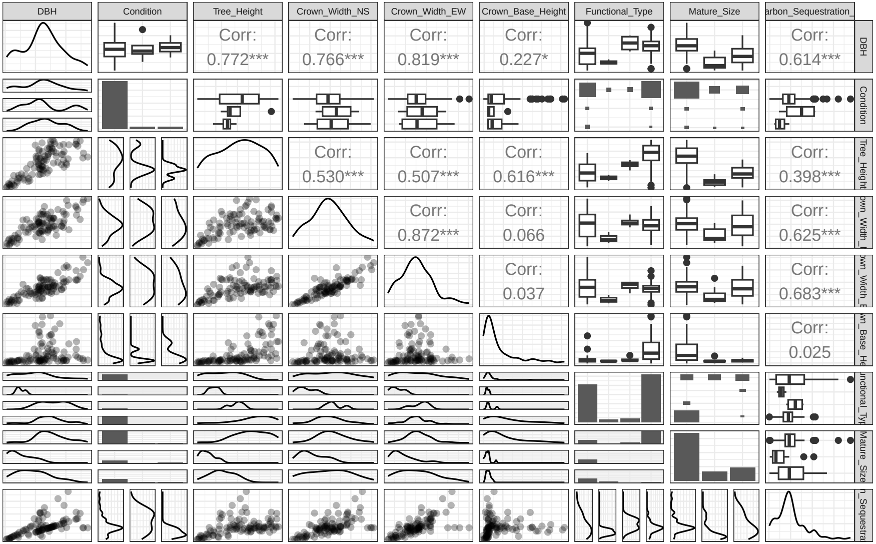
```r
dim(my_pdxTrees)
```

```
## [1] 3015    9
```

```r
set.seed(1)
library(rsample)
my_pdxTrees_split <- initial_split(my_pdxTrees )
my_pdxTrees_train <- training(my_pdxTrees_split)
my_pdxTrees_test <- testing(my_pdxTrees_split)
```

```r
library(GGally)
ggpairs(my_pdxTrees_train)
```

Exploratory Analysis



Random Forest in R

- To create both bagged trees and random forests, we use the `randomForest` function in the `randomForest` package in R:

```
library(randomForest)
rfmodel <- randomForest(Carbon_Sequestration_lb ~ ., data = my_pdxTrees_train)
rfmodel
```

```
##
## Call:
##  randomForest(formula = Carbon_Sequestration_lb ~ ., data = my_pdxTrees_train)
##               Type of random forest: regression
##                     Number of trees: 500
## No. of variables tried at each split: 2
##
##           Mean of squared residuals: 111.5371
##                     % Var explained: 85.84
```

Modifications

We can control how many trees are generated with `ntree` and the number of predictors at each split with `mtry`

Modifications

We can control how many trees are generated with `ntree` and the number of predictors at each split with `mtry`

- By default, `randomForest` uses $p/3$ predictors for regression and $\sqrt{p}$ predictors for classification

Modifications

We can control how many trees are generated with `ntree` and the number of predictors at each split with `mtry`

- By default, `randomForest` uses $p/3$ predictors for regression and $\sqrt{p}$ predictors for classification

```r
set.seed(1)
rfmodel2 <- randomForest(Carbon_Sequestration_lb ~ ., data = my_pdxTrees_train,
                         ntree = 10, mtry = 5)
rfmodel2
```

```
##
## Call:
##  randomForest(formula = Carbon_Sequestration_lb ~ ., data = my_pdxTrees_train,      ntree = 1
##                Type of random forest: regression
##                      Number of trees: 10
## No. of variables tried at each split: 5
##
##           Mean of squared residuals: 106.4475
##                     % Var explained: 86.48
```

Modifications

We can control how many trees are generated with `ntree` and the number of predictors at each split with `mtry`

- By default, `randomForest` uses $p/3$ predictors for regression and $\sqrt{p}$ predictors for classification

```
set.seed(1)
rfmodel2 <- randomForest(Carbon_Sequestration_lb ~ ., data = my_pdxTrees_train,
                         ntree = 10, mtry = 5)
rfmodel2
```

```
##
## Call:
##  randomForest(formula = Carbon_Sequestration_lb ~ ., data = my_pdxTrees_train,      ntree = 1
##                Type of random forest: regression
##                      Number of trees: 10
## No. of variables tried at each split: 5
##
##           Mean of squared residuals: 106.4475
##                     % Var explained: 86.48
```

How can we create a bagged model using the `randomForest` function?

Modifications

We can control how many trees are generated with `ntree` and the number of predictors at each split with `mtry`

- By default, `randomForest` uses $p/3$ predictors for regression and $\sqrt{p}$ predictors for classification

```r
set.seed(1)
rfmodel2 <- randomForest(Carbon_Sequestration_lb ~ ., data = my_pdxTrees_train,
                         ntree = 10, mtry = 5)
rfmodel2
```

```
##
## Call:
##  randomForest(formula = Carbon_Sequestration_lb ~ ., data = my_pdxTrees_train,      ntree = 1
##                Type of random forest: regression
##                      Number of trees: 10
## No. of variables tried at each split: 5
##
##           Mean of squared residuals: 106.4475
##                     % Var explained: 86.48
```

How can we create a bagged model using the `randomForest` function?

- Set `mtry=` `p`, where `p` is the total number predictors available

Making predictions

- So you have your `randomForest` model. How do you make predictions?

```r
my_preds<- predict(rfmodel, my_pdxTrees_test)
results <- data.frame(obs =  my_pdxTrees_test$Carbon_Sequestration_lb, preds = my_preds)

results %>% head()
```

```
##     obs    preds
## 1  39.0 38.26301
## 2 110.2 66.90372
## 3  61.2 76.66064
## 4  34.0 33.92686
## 5  75.4 52.68092
## 6  96.1 83.09862
```

Making predictions

- So you have your `randomForest` model. How do you make predictions?

```
my_preds<- predict(rfmodel, my_pdxTrees_test)
results <- data.frame(obs =  my_pdxTrees_test$Carbon_Sequestration_lb, preds = my_preds)

results %>% head()
```

```
##     obs    preds
## 1  39.0 38.26301
## 2 110.2 66.90372
## 3  61.2 76.66064
## 4  34.0 33.92686
## 5  75.4 52.68092
## 6  96.1 83.09862
```

Let's compute test rMSE

Making predictions

- So you have your `randomForest` model. How do you make predictions?

```
my_preds<- predict(rfmodel, my_pdxTrees_test)
results <- data.frame(obs =  my_pdxTrees_test$Carbon_Sequestration_lb, preds = my_preds)

results %>% head()
```

```
##     obs    preds
## 1  39.0 38.26301
## 2 110.2 66.90372
## 3  61.2 76.66064
## 4  34.0 33.92686
## 5  75.4 52.68092
## 6  96.1 83.09862
```

Let's compute test rMSE

```
library(yardstick)
results %>% rmse(truth = obs, estimate = preds)
```

```
## # A tibble: 1 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 rmse    standard        11.3
```

Making predictions

- So you have your `randomForest` model. How do you make predictions?

```
my_preds<- predict(rfmodel, my_pdxTrees_test)
results <- data.frame(obs =  my_pdxTrees_test$Carbon_Sequestration_lb, preds = my_preds)

results %>% head()
```

```
##     obs    preds
## 1  39.0 38.26301
## 2 110.2 66.90372
## 3  61.2 76.66064
## 4  34.0 33.92686
## 5  75.4 52.68092
## 6  96.1 83.09862
```

Let's compute test rMSE

```
library(yardstick)
results %>% rmse(truth = obs, estimate = preds)
```

```
## # A tibble: 1 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 rmse    standard        11.3
```

- For reference, the bagged model had rMSE of 12.3, while the average rMSE for single trees was 13.9

Variable Importance

Bagging and Random Forests increase prediction accuracy by reducing variance of the model.

Variable Importance

Bagging and Random Forests increase prediction accuracy by reducing variance of the model.

- But the cost comes in interpretability We no longer have a single decision tree to follow to reach our prediction.

Variable Importance

Bagging and Random Forests increase prediction accuracy by reducing variance of the model.

- But the cost comes in interpretability We no longer have a single decision tree to follow to reach our prediction.
- How can we determine which predictors are most influential?

Variable Importance

Bagging and Random Forests increase prediction accuracy by reducing variance of the model.

- But the cost comes in interpretability We no longer have a single decision tree to follow to reach our prediction.
- How can we determine which predictors are most influential?

One possibility is to record the total amount of RSS/Purity that is decreased due to splits of the given predictor, averaged across all trees in the random forest.

Importance in R

```
importance(rfmodel)
```

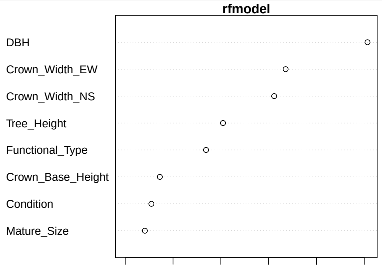
```
##                    IncNodePurity
## DBH                    506807.58
## Condition               54752.15
## Tree_Height            204541.39
## Crown_Width_NS         311571.85
## Crown_Width_EW         335526.52
## Crown_Base_Height       72446.30
## Functional_Type        169066.91
## Mature_Size             41094.65
```

Importance in R

```
importance(rfmodel)
```

```
##                   IncNodePurity
## DBH                   506807.58
## Condition              54752.15
## Tree_Height           204541.39
## Crown_Width_NS        311571.85
## Crown_Width_EW        335526.52
## Crown_Base_Height      72446.30
## Functional_Type       169066.91
## Mature_Size            41094.65
```

```
varImpPlot(rfmodel)
```

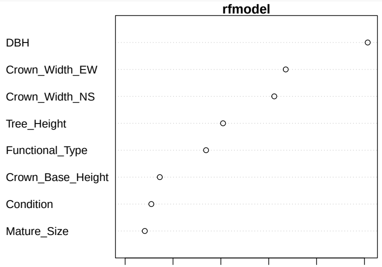


Importance in R

```r
importance(rfmodel)
```

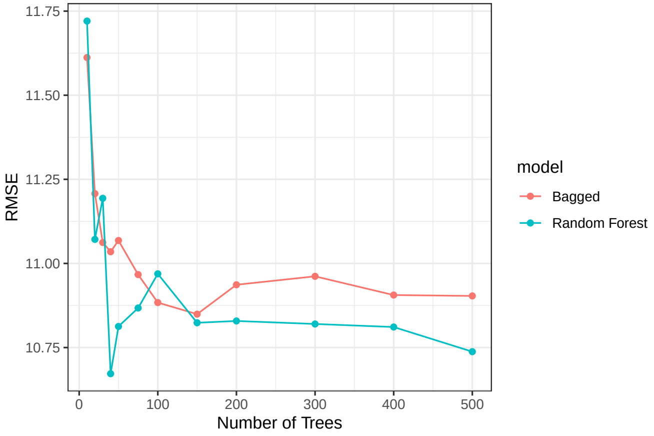
```
##                  IncNodePurity
## DBH                  506807.58
## Condition             54752.15
## Tree_Height          204541.39
## Crown_Width_NS       311571.85
## Crown_Width_EW       335526.52
## Crown_Base_Height     72446.30
## Functional_Type      169066.91
## Mature_Size           41094.65
```

```r
varImpPlot(rfmodel)
```



- For regression trees, node impurity is calculated using RSS.
- For classification trees, node impurity is calculated using Gini Index.

Comparison of Bagged Trees versus Random Forests



Section 3

Boosting

Motivation

Suppose you have a model which, given a binary classification dataset, always returned a classifier with training error strictly lower than 50%.

Motivation

Suppose you have a model which, given a binary classification dataset, always returned a classifier with training error strictly lower than 50%.

- Can one use it to build a strong classifier that has error close to 0?

Motivation

Suppose you have a model which, given a binary classification dataset, always returned a classifier with training error strictly lower than 50%.

- Can one use it to build a strong classifier that has error close to 0?

AdaBoost

In the 1990s, Shapire and Freund developed algorithms to do just that.

AdaBoost

In the 1990s, Shapire and Freund developed algorithms to do just that.

- Their algorithm (AdaBoost) generates a sequence of weak classifiers, where at each iteration the algorithm finds the best classifier based on the current sample weights.

AdaBoost

In the 1990s, Shapire and Freund developed algorithms to do just that.

- Their algorithm (AdaBoost) generates a sequence of weak classifiers, where at each iteration the algorithm finds the best classifier based on the current sample weights.
  - Observations that are incorrectly classifed in the kth iteration recieve more weight in the $(k+1)$th iteration.

AdaBoost

In the 1990s, Shapire and Freund developed algorithms to do just that.

- Their algorithm (AdaBoost) generates a sequence of weak classifiers, where at each iteration the algorithm finds the best classifier based on the current sample weights.
  - Observations that are incorrectly classifed in the kth iteration recieve more weight in the $(k+1)$th iteration.
- The overall sequence of classifiers are combined into an ensemble which as high chance of classifying more accurately than any individaul model in the list.

AdaBoost

In the 1990s, Shapire and Freund developed algorithms to do just that.

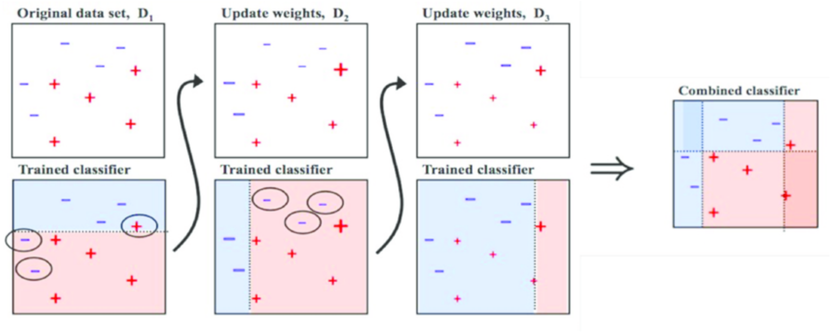
- Their algorithm (AdaBoost) generates a sequence of weak classifiers, where at each iteration the algorithm finds the best classifier based on the current sample weights.
  - Observations that are incorrectly classifed in the kth iteration recieve more weight in the $(k+1)$th iteration.
- The overall sequence of classifiers are combined into an ensemble which as high chance of classifying more accurately than any individaul model in the list.
- The algorithm relies on using a sequence of **weak** learners (low variance, high bias)

AdaBoost

In the 1990s, Shapire and Freund developed algorithms to do just that.

- Their algorithm (AdaBoost) generates a sequence of weak classifiers, where at each iteration the algorithm finds the best classifier based on the current sample weights.
  - Observations that are incorrectly classifed in the kth iteration recieve more weight in the $(k+1)$th iteration.
- The overall sequence of classifiers are combined into an ensemble which as high chance of classifying more accurately than any individaul model in the list.
- The algorithm relies on using a sequence of **weak** learners (low variance, high bias)
  - In the tree setting, we can create weak learners by restricting the depth of the tree.

AdaBoost Graphic



Boosting for regression

Boosting also works in the regression setting. The **gradient boosting machine** is a boosting algorithm that works as follows:

1. Select tree depth D and number of iterations K.
2. Compute the average response $\hat{y}$ and use this as the initial predicted value for each observation
3. Compute the residual for each observation.
4. Fit a regression tree of depth D, using the **residuals** as the response.
5. Predict each observation using the regression tree from the previous step.
6. Update the predicted value of each observation by adding the previous iteration's predicted value to the predicted value generated in the previous step.
7. Repeat at total of K times.

Brief Example

Compute the mean:

```
mu <- mean(my_pdxTrees_train$Carbon_Sequestration_lb)
mu
```

```
## [1] 34.49668
```

Brief Example

Compute the mean:

```r
mu <- mean(my_pdxTrees_train$Carbon_Sequestration_lb)
mu
```

```
## [1] 34.49668
```

Compute residuals:

```r
my_pdxTrees_train_boost <- my_pdxTrees_train %>%
  mutate(residuals1 = Carbon_Sequestration_lb - mu)
```

Brief Example

Compute the mean:

```
mu <- mean(my_pdxTrees_train$Carbon_Sequestration_lb)
mu
```

```
## [1] 34.49668
```

Compute residuals:

```
my_pdxTrees_train_boost <- my_pdxTrees_train %>%
  mutate(residuals1 = Carbon_Sequestration_lb - mu)
```

Fit a new tree

```
boost_tree_model<- rpart(residuals1 ~ Crown_Base_Height,
                        data = my_pdxTrees_train_boost,
                        control = rpart.control(maxdepth = 2))
```

Brief Example

Compute the mean:

```
mu <- mean(my_pdxTrees_train$Carbon_Sequestration_lb)
mu
```

```
## [1] 34.49668
```

Compute residuals:

```
my_pdxTrees_train_boost <- my_pdxTrees_train %>%
  mutate(residuals1 = Carbon_Sequestration_lb - mu)
```

Fit a new tree

```
boost_tree_model<- rpart(residuals1 ~ Crown_Base_Height,
                   data = my_pdxTrees_train_boost,
                   control = rpart.control(maxdepth = 2))
```

Predict

```
predictions<- predict(boost_tree_model, data = my_pdxTrees_test)+mu
```

Brief Example

Compute the mean:

```
mu <- mean(my_pdxTrees_train$Carbon_Sequestration_lb)
mu
```

```
## [1] 34.49668
```

Compute residuals:

```
my_pdxTrees_train_boost <- my_pdxTrees_train %>%
  mutate(residuals1 = Carbon_Sequestration_lb - mu)
```

Fit a new tree

```
boost_tree_model<- rpart(residuals1 ~ Crown_Base_Height,
                   data = my_pdxTrees_train_boost,
                   control = rpart.control(maxdepth = 2))
```

Predict

```
predictions<- predict(boost_tree_model, data = my_pdxTrees_test)+mu
```

And so on. . .

Boosting Properties

Boosting is similar to random forests: the final prediction is sum of predictions from an ensemble of models.

Boosting Properties

Boosting is similar to random forests: the final prediction is sum of predictions from an ensemble of models.

- But in Random Forests, all trees are created independently, are of maximum depth, and contribute equally to the final model.

Boosting Properties

Boosting is similar to random forests: the final prediction is sum of predictions from an ensemble of models.

- But in Random Forests, all trees are created independently, are of maximum depth, and contribute equally to the final model.
- In boosting, subsequent trees are are highly dependent on past trees, have minimal depth, and contribute unequally.

Boosting Properties

Boosting is similar to random forests: the final prediction is sum of predictions from an ensemble of models.

- But in Random Forests, all trees are created independently, are of maximum depth, and contribute equally to the final model.
- In boosting, subsequent trees are are highly dependent on past trees, have minimal depth, and contribute unequally.

Unlike random forests, boosting is susceptible to over-fitting (since it uses a greedy algorithm to maximize gradient at each step).

Boosting Properties

Boosting is similar to random forests: the final prediction is sum of predictions from an ensemble of models.

- But in Random Forests, all trees are created independently, are of maximum depth, and contribute equally to the final model.
- In boosting, subsequent trees are are highly dependent on past trees, have minimal depth, and contribute unequally.

Unlike random forests, boosting is susceptible to over-fitting (since it uses a greedy algorithm to maximize gradient at each step).

- To remedy, we introduce a shrinkage penalty (like in Ridge Regression/LASSO)

Boosting Properties

Boosting is similar to random forests: the final prediction is sum of predictions from an ensemble of models.

- But in Random Forests, all trees are created independently, are of maximum depth, and contribute equally to the final model.
- In boosting, subsequent trees are are highly dependent on past trees, have minimal depth, and contribute unequally.

Unlike random forests, boosting is susceptible to over-fitting (since it uses a greedy algorithm to maximize gradient at each step).

- To remedy, we introduce a shrinkage penalty (like in Ridge Regression/LASSO)
  - Instead of adding the full value for a sample to the previous iteration's predicted value, only a fraction of the current predicted value is added.

Boosting Properties

Boosting is similar to random forests: the final prediction is sum of predictions from an ensemble of models.

- But in Random Forests, all trees are created independently, are of maximum depth, and contribute equally to the final model.
- In boosting, subsequent trees are are highly dependent on past trees, have minimal depth, and contribute unequally.

Unlike random forests, boosting is susceptible to over-fitting (since it uses a greedy algorithm to maximize gradient at each step).

- To remedy, we introduce a shrinkage penalty (like in Ridge Regression/LASSO)
  - Instead of adding the full value for a sample to the previous iteration's predicted value, only a fraction of the current predicted value is added.
  - This fraction is called the *learning rate* λ, with $0 < \lambda < 1$. (Typical values range from 0.001 to 0.01)

Boosting in R

We use the `gbm` function in the `gmb` package to create Boosted Trees

Boosting in R

We use the `gbm` function in the `gmb` package to create Boosted Trees

- For regression problems, we use the argument `distribution = "gaussian"` and for classification problems, we use `distribution = "bernoulli"`

Boosting in R

We use the `gbm` function in the `gmb` package to create Boosted Trees

- For regression problems, we use the argument `distribution = "gaussian"` and for classification problems, we use `distribution = "bernoulli"`
- The argument `n.trees` controls the number of iterations
- The argument `interaction.depth` controls the depth of each tree
- The argument `shrinkage` controlls the learning rate λ

Boosting in R

We use the gbm function in the gmb package to create Boosted Trees

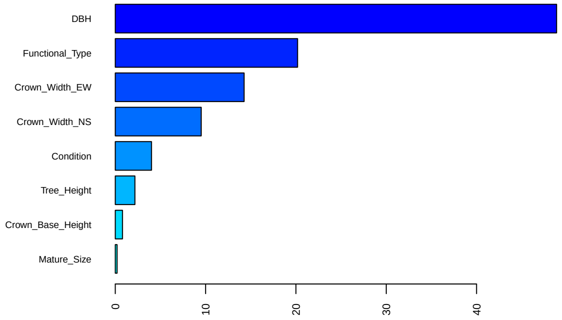
- For regression problems, we use the argument distribution = "gaussian" and for classification problems, we use distribution = "bernoulli"
- The argument n.trees controls the number of iterations
- The argument interaction.depth controls the depth of each tree
- The argument shrinkage controlls the learning rate λ

```r
library(gbm)
set.seed(10101)
boosted_tree<-gbm(Carbon_Sequestration_lb ~., my_pdxTrees_train,
                  distribution = "gaussian",
                  n.trees=5000,
                  interaction.depth = 3,
                  shrinkage = .0025)
```

Summary Information

```
summary(boosted_tree )
```

```
##                               var    rel.inf
## DBH                           DBH 48.8607778
## Functional_Type     Functional_Type 20.1833428
## Crown_Width_EW       Crown_Width_EW 14.2618538
## Crown_Width_NS       Crown_Width_NS  9.5128157
## Condition                 Condition  4.0105335
## Tree_Height             Tree_Height  2.1739086
## Crown_Base_Height Crown_Base_Height  0.7980455
## Mature_Size             Mature_Size  0.1987224
```



Boosted Tree Performance

- How does the boosted tree do vs Random Forest? A pruned tree? A linear model?

Boosted Tree Performance

- How does the boosted tree do vs Random Forest? A pruned tree? A linear model?

```
results %>% group_by(model) %>% rmse(truth = obs, estimate = preds) %>% arrange(.estimate)
```

```
## # A tibble: 4 x 4
##   model         .metric .estimator .estimate
##   <chr>         <chr>   <chr>          <dbl>
## 1 random_forest rmse    standard        10.8
## 2 boosted_tree  rmse    standard        11.2
## 3 pruned_tree   rmse    standard        13.7
## 4 linear_model  rmse    standard        17.7
```

Boosted Tree Performance

- How does the boosted tree do vs Random Forest? A pruned tree? A linear model?

```
results %>% group_by(model) %>% rmse(truth = obs, estimate = preds) %>% arrange(.estimate)
```

```
## # A tibble: 4 x 4
##   model         .metric .estimator .estimate
##   <chr>         <chr>   <chr>          <dbl>
## 1 random_forest rmse    standard        10.8
## 2 boosted_tree  rmse    standard        11.2
## 3 pruned_tree   rmse    standard        13.7
## 4 linear_model  rmse    standard        17.7
```

- This behavior is typical. Boosted trees and Random Forests often have comparable performance, and both tend to be more accurate than other model types

Boosted Tree Performance

- How does the boosted tree do vs Random Forest? A pruned tree? A linear model?

```
results %>% group_by(model) %>% rmse(truth = obs, estimate = preds) %>% arrange(.estimate)
```

```
## # A tibble: 4 x 4
##   model         .metric .estimator .estimate
##   <chr>         <chr>   <chr>          <dbl>
## 1 random_forest rmse    standard        10.8
## 2 boosted_tree  rmse    standard        11.2
## 3 pruned_tree   rmse    standard        13.7
## 4 linear_model  rmse    standard        17.7
```

- This behavior is typical. Boosted trees and Random Forests often have comparable performance, and both tend to be more accurate than other model types

- However, this performance comes at significant cost of interpretability.

Boosted Tree Performance

- How does the boosted tree do vs Random Forest? A pruned tree? A linear model?

```
results %>% group_by(model) %>% rmse(truth = obs, estimate = preds) %>% arrange(.estimate)
```

```
## # A tibble: 4 x 4
##   model         .metric .estimator .estimate
##   <chr>         <chr>   <chr>          <dbl>
## 1 random_forest rmse    standard        10.8
## 2 boosted_tree  rmse    standard        11.2
## 3 pruned_tree   rmse    standard        13.7
## 4 linear_model  rmse    standard        17.7
```

- This behavior is typical. Boosted trees and Random Forests often have comparable performance, and both tend to be more accurate than other model types
- However, this performance comes at significant cost of interpretability.
- Note that boosted trees have a number of important parameters: `n.trees`, `interaction.depth`, `shrinkage`.

Boosted Tree Performance

- How does the boosted tree do vs Random Forest? A pruned tree? A linear model?

```
results %>% group_by(model) %>% rmse(truth = obs, estimate = preds) %>% arrange(.estimate)
```

```
## # A tibble: 4 x 4
##   model         .metric .estimator .estimate
##   <chr>         <chr>   <chr>          <dbl>
## 1 random_forest rmse    standard        10.8
## 2 boosted_tree  rmse    standard        11.2
## 3 pruned_tree   rmse    standard        13.7
## 4 linear_model  rmse    standard        17.7
```

- This behavior is typical. Boosted trees and Random Forests often have comparable performance, and both tend to be more accurate than other model types
- However, this performance comes at significant cost of interpretability.
- Note that boosted trees have a number of important parameters: `n.trees`, `interaction.depth`, `shrinkage`.
  - How do we find the best values of these hyperparameters?

Boosted Tree Performance

- How does the boosted tree do vs Random Forest? A pruned tree? A linear model?

```
results %>% group_by(model) %>% rmse(truth = obs, estimate = preds) %>% arrange(.estimate)
```

```
## # A tibble: 4 x 4
##   model         .metric .estimator .estimate
##   <chr>         <chr>   <chr>          <dbl>
## 1 random_forest rmse    standard        10.8
## 2 boosted_tree  rmse    standard        11.2
## 3 pruned_tree   rmse    standard        13.7
## 4 linear_model  rmse    standard        17.7
```

- This behavior is typical. Boosted trees and Random Forests often have comparable performance, and both tend to be more accurate than other model types
- However, this performance comes at significant cost of interpretability.
- Note that boosted trees have a number of important parameters: `n.trees`, `interaction.depth`, `shrinkage`.
  - How do we find the best values of these hyperparameters?
  - Cross-validation!

Cross-Validating gbm

Warning! fitting a single `gbm` models can be time and computing intensive.

- Using cross-validation to compare multiple models can be VERY time and computing intensive
- Cross-validation for `gbm` models is NOT RECOMMENDED if using the RStudio Server

Cross-Validating gbm

Warning! fitting a single gbm models can be time and computing intensive.

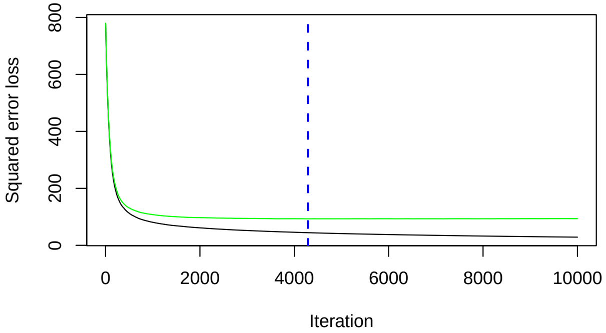
- Using cross-validation to compare multiple models can be VERY time and computing intensive
- Cross-validation for gbm models is NOT RECOMMENDED if using the RStudio Server
- We can include an additional cross-validation term in our boosted tree model.
    - It may be helpful to include a number of CPU cores as well. First verify your number of available cores using `parallel::decectCores()`

```
library(gbm)
set.seed(10101)
cv_boosted_tree<-gbm(Carbon_Sequestration_lb ~., my_pdxTrees_train,
                  distribution = "gaussian",
                  n.trees=10000,
                  interaction.depth = 3,
                  shrinkage = .01,
                  cv.folds = 10,
                  n.cores = 8)
```

CV Results

- We can plot cross-validated performance using `gbm.perf()`

```
gbm.perf(cv_boosted_tree, method = "cv")
```

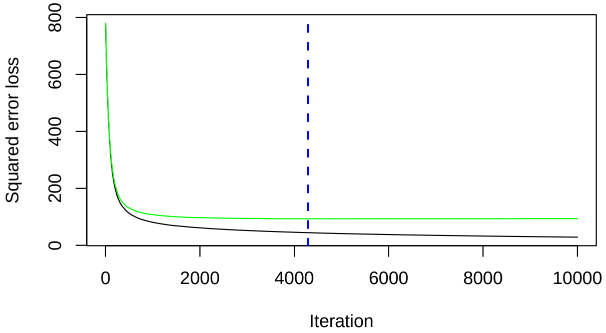


```
## [1] 4290
```

CV Results

- We can plot cross-validated performance using `gbm.perf()`

```
gbm.perf(cv_boosted_tree, method = "cv")
```



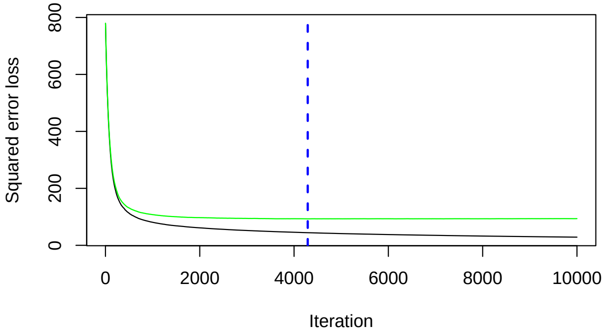
```
## [1] 4290
```

- The green curve is the cross-validated error, while the black curve is the training error.

CV Results

- We can plot cross-validated performance using `gbm.perf()`

```
gbm.perf(cv_boosted_tree, method = "cv")
```



```
## [1] 4290
```

- The green curve is the cross-validated error, while the black curve is the training error.
- The blue vertical line is the optimal value of the cross-validated error

Recording CV Error

- The `gbm` object also stores the values of the cross-validated errors for each number of trees used, accessible via `$cv.errors`

Recording CV Error

- The gbm object also stores the values of the cross-validated errors for each number of trees used, accessible via $cv.errors

```
my_errors <- cv_boosted_tree$cv.error
best_n <- which.min(cv_boosted_tree$cv.error)
data.frame(best_n, cv_error = my_errors[best_n])
```

```
##   best_n cv_error
## 1   4290 93.01164
```

Recording CV Error

- The gbm object also stores the values of the cross-validated errors for each number of trees used, accessible via $cv.errors

```
my_errors <- cv_boosted_tree$cv.error
best_n <- which.min(cv_boosted_tree$cv.error)
data.frame(best_n, cv_error = my_errors[best_n])
```

```
##   best_n cv_error
## 1   4290 93.01164
```

- This is particularly useful if we want to record the error for a model with certain parameters

General Strategy for finding best Parameters

1. Choose a relatively high initial learning rate. A rate of 0.1 is a reasonable starting point.
2. Determine the optimal number of trees for this learning rate using cross-validation.
3. Fix other tree-specific parameters and tune the learning rate, assessed by computation speed and model accuracy.
4. Tune tree-specific parameters for fixed learning rate.
5. Once tree-specific parameters have been found, lower learning rate and increase number of trees to assess improvements in accuracy.

Warning! This search can take considerable time (minutes to hours), depending on computing power, number of variables in model, and number of observations. DO NOT ATTEMPT ON RSTUDIO SERVER!!

Cross-Validation along a Grid

- In order to cross-validate a large number of parameters, we create a parameter grid:

Cross-Validation along a Grid

- In order to cross-validate a large number of parameters, we create a parameter grid:

```
my_grid <- expand.grid(
  n.trees = 5000,
  shrinkage = 0.01,
  interaction.depth = c(3,5,7),
  n.minobsinnode = c(5,10,15)
)
```

Cross-Validation along a Grid

- In order to cross-validate a large number of parameters, we create a parameter grid:

```
my_grid <- expand.grid(
  n.trees = 5000,
  shrinkage = 0.01,
  interaction.depth = c(3,5,7),
  n.minobsinnode = c(5,10,15)
)
```

- Then we create a model fitting function:

Cross-Validation along a Grid

- In order to cross-validate a large number of parameters, we create a parameter grid:

```
my_grid <- expand.grid(
  n.trees = 5000,
  shrinkage = 0.01,
  interaction.depth = c(3,5,7),
  n.minobsinnode = c(5,10,15)
)
```

- Then we create a model fitting function:

```
model_fit <- function(n.trees, shrinkage, interaction.depth, n.minobsinnode){
  set.seed(40)
  library(gbm)
  gbm_mod <- bm(Carbon_Sequestration_lb ~., my_pdxTrees_train,
                distribution = "gaussian",
                n.trees=n.trees,
                interaction.depth = interaction.depth,
                shrinkage = shrinkage,
                cv.folds = 10,
                n.cores = 8,
              n.minobsinnode = n.minobsinnode)
  rMSE <- sqrt(min(gbm_mod$cv.error))
  rMSE
}
```

Cross-Validation along a Grid

- In order to cross-validate a large number of parameters, we create a parameter grid:

Cross-Validation along a Grid

- In order to cross-validate a large number of parameters, we create a parameter grid:

```r
my_grid <- expand.grid(
  n.trees = 5000,
  shrinkage = 0.01,
  interaction.depth = c(3,5,7),
  n.minobsinnode = c(5,10,15)
)
```

Cross-Validation along a Grid

- In order to cross-validate a large number of parameters, we create a parameter grid:

```
my_grid <- expand.grid(
  n.trees = 5000,
  shrinkage = 0.01,
  interaction.depth = c(3,5,7),
  n.minobsinnode = c(5,10,15)
)
```

- Then we create a model fitting function:

Cross-Validation along a Grid

- In order to cross-validate a large number of parameters, we create a parameter grid:

```r
my_grid <- expand.grid(
  n.trees = 5000,
  shrinkage = 0.01,
  interaction.depth = c(3,5,7),
  n.minobsinnode = c(5,10,15)
)
```

- Then we create a model fitting function:

```r
model_fit <- function(n.trees, shrinkage, interaction.depth, n.minobsinnode){
  set.seed(40)
  library(gbm)
  gbm_mod <- gbm(Carbon_Sequestration_lb ~., my_pdxTrees_train,
                 distribution = "gaussian",
                 n.trees=n.trees,
                 interaction.depth = interaction.depth,
                 shrinkage = shrinkage,
                 cv.folds = 10,
                 n.cores = 8,
               n.minobsinnode = n.minobsinnode)
  rMSE <- sqrt(min(gbm_mod$cv.error))
  rMSE
}
```

Implementing the Grid Search

We now use the `pmap_dbl` function from `purrr`:

Implementing the Grid Search

We now use the `pmap_dbl` function from `purrr`:

```
library(purrr)
my_grid$rmse <- pmap_dbl(
  my_grid,
  ~ model_fit(
    n.trees = ..1,
    shrinkage = ..2,
    interaction =..3,
    n.minobsinnode = ..4
  )
)
```

Implementing the Grid Search

We now use the pmap_dbl function from purrr:

```
library(purrr)
my_grid$rmse <- pmap_dbl(
  my_grid,
  ~ model_fit(
    n.trees = ..1,
    shrinkage = ..2,
    interaction =..3,
    n.minobsinnode = ..4
  )
)
```

We can then view results of the exhuastive search:

Implementing the Grid Search

We now use the pmap_dbl function from purrr:

```
library(purrr)
my_grid$rmse <- pmap_dbl(
  my_grid,
  ~ model_fit(
    n.trees = ..1,
    shrinkage = ..2,
    interaction =..3,
    n.minobsinnode = ..4
  )
)
```

We can then view results of the exhuastive search:

```
head(arrange(my_grid, rmse))
```

```
##   n.trees shrinkage interaction.depth n.minobsinnode     rmse
## 1    5000      0.01                 7              5 9.389038
## 2    5000      0.01                 7             10 9.486088
## 3    5000      0.01                 5              5 9.557491
## 4    5000      0.01                 7             15 9.610702
## 5    5000      0.01                 5             10 9.708710
## 6    5000      0.01                 5             15 9.708723
```