Advanced Trees

Contents

Trees are flexible and easy to understand but tend to overfit and do not have the same level of predictive accuracy as other prediction models. In the following sections, we will examine a number of ways to improve tree models including tuning tree hyperparameters, and ensemble models such as bagging, forests, and boosting.

Load Data

The dataset, Credit for this exercise accompanies the `ISLR2` library. The dataset contains information on credit cards and demographics for a set of 400 customers. The goal is to predict credit card balance using other information in the dataset.

```
library(ISLR2)
data(Credit)
```

Split Data

Conduct a 75:25 stratified split of the data on the outcome variable, Balance.

```
library(caret)
set.seed(1031)
split = createDataPartition(y = Credit$Balance, p = 0.75, list = F,groups = 10)
train = Credit[split,]
test = Credit[-split,]
```

Explore

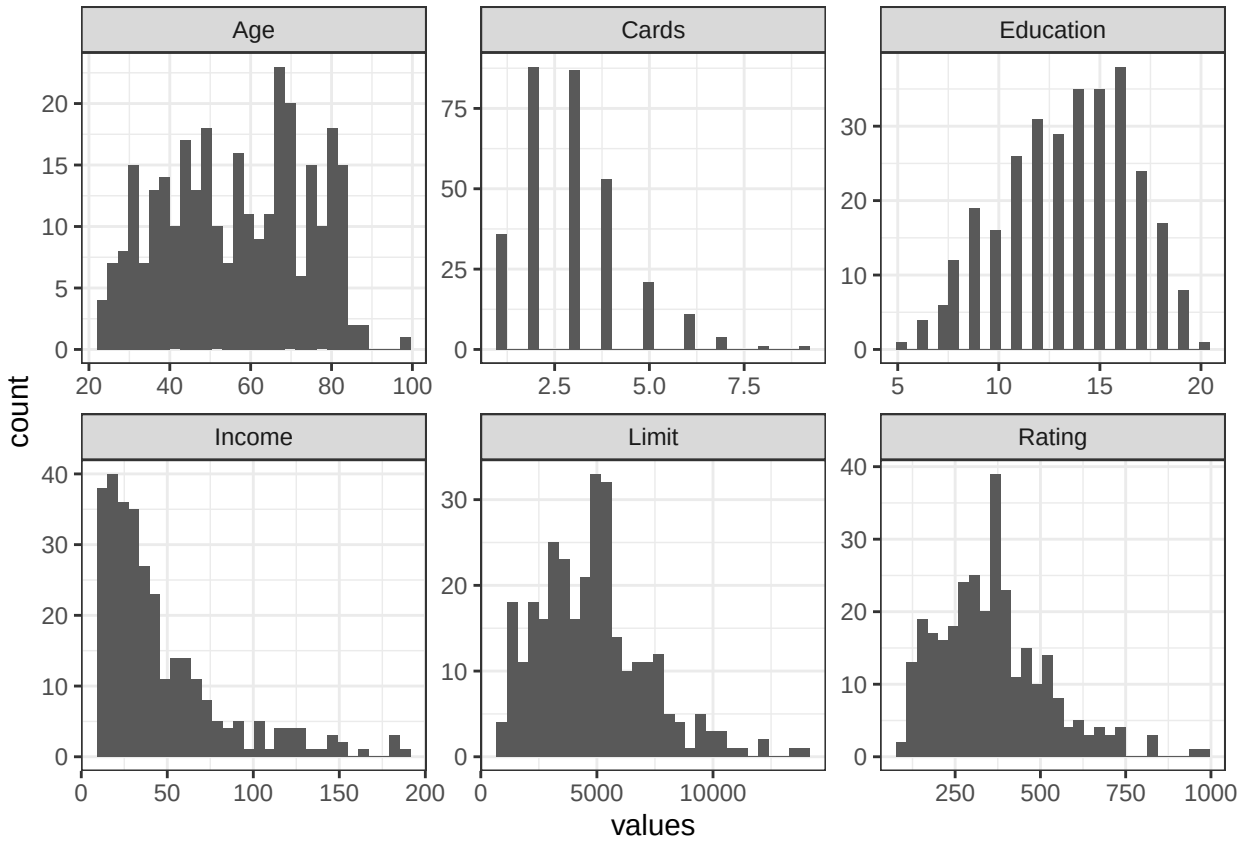
Summary of the data is displayed below. Here are a few highlights and their relevance to regression trees.
1. Outcome variable, Balance is continuous: Accordingly, we will use regression trees.
2. Predictors include categorical and continuous variables: Trees can handle both categorical and continuous predictors. Categorical variables can be handled automatically, without dummy coding.
3. Distribution of at least a few continuous variables appear to be skewed. Trees can effectively handle many type of predictors including those with skewed distributions. Predictors do not need to be transformed to make their distribution symmetric nor do they need to be standardized. Similarly, there is no need to create interaction terms to explore joint effect of two or more predictors.
4. There are no missing values. While there are no missing values here, it is worth noting that Trees can effectively handle missing data by constructing surrogate splits.

```
summary(Credit)
```

```
##      Income           Limit           Rating          Cards
##  Min.   : 10.35   Min.   :  855   Min.   : 93.0   Min.   :1.000
##  1st Qu.: 21.01   1st Qu.: 3088   1st Qu.:247.2   1st Qu.:2.000
##  Median : 33.12   Median : 4622   Median :344.0   Median :3.000
##  Mean   : 45.22   Mean   : 4736   Mean   :354.9   Mean   :2.958
##  3rd Qu.: 57.47   3rd Qu.: 5873   3rd Qu.:437.2   3rd Qu.:4.000
##  Max.   :186.63   Max.   :13913   Max.   :982.0   Max.   :9.000
##       Age          Education      Own      Student   Married     Region
##  Min.   :23.00   Min.   : 5.00   No :193   No :360   No :155   East : 99
##  1st Qu.:41.75   1st Qu.:11.00   Yes:207   Yes: 40   Yes:245   South:199
##  Median :56.00   Median :14.00                                 West :102
##  Mean   :55.67   Mean   :13.45
##  3rd Qu.:70.00   3rd Qu.:16.00
##  Max.   :98.00   Max.   :20.00
##     Balance
##  Min.   :   0.00
##  1st Qu.:  68.75
##  Median : 459.50
##  Mean   : 520.01
##  3rd Qu.: 863.00
##  Max.   :1999.00
```

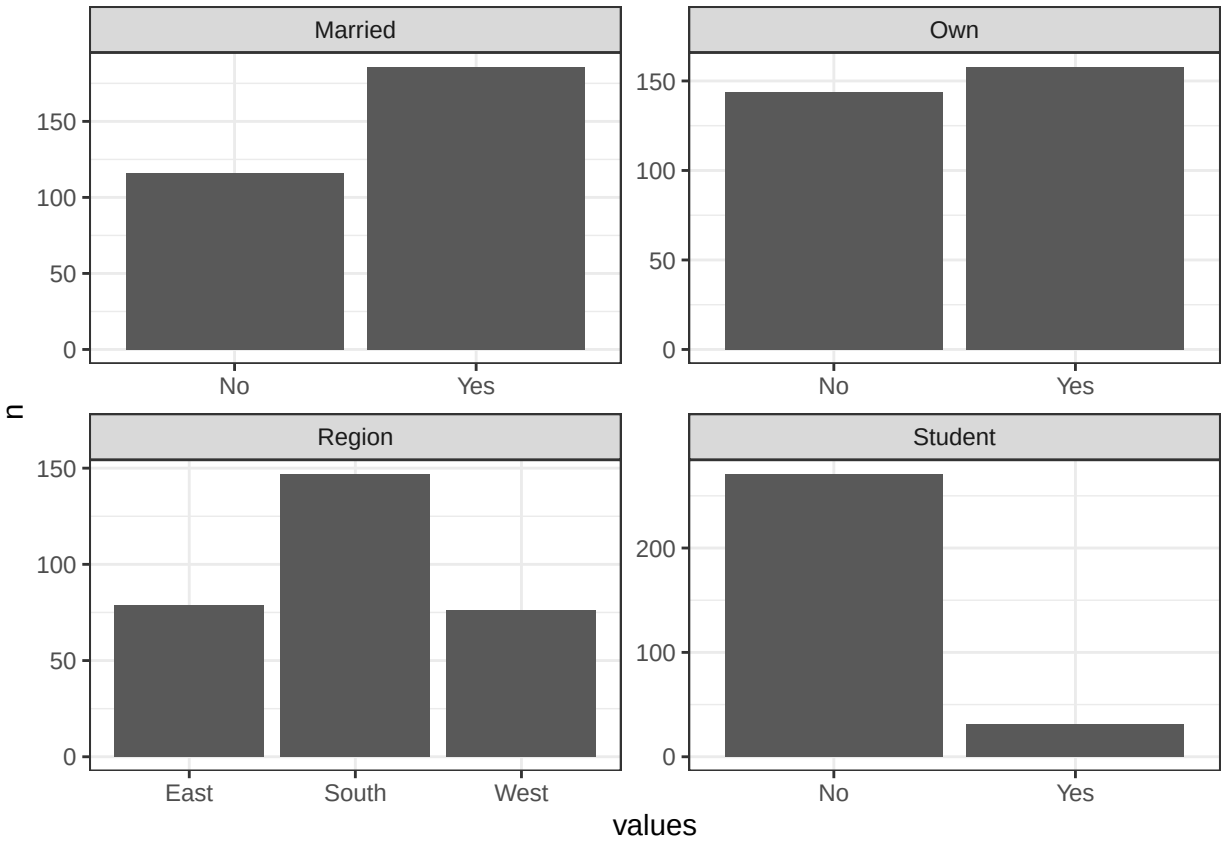
Examine distribution of numeric predictors

```
library(dplyr); library(tidyr)
train %>%
  select(-Balance)%>%
  select_if(is.numeric)%>%
  pivot_longer(cols = 1:6,names_to = 'numeric_predictor', values_to = 'values'  )%>%
  ggplot(aes(x = values))+
  geom_histogram()+
  facet_wrap(numeric_predictor~., scales = 'free')+
  theme_bw()
```



Examine frequency distribution for categorical predictors

```
library(dplyr); library(tidyr)
train %>%
  select_if(is.factor)%>%
  pivot_longer(cols = 1:4,names_to = 'categorical_predictor', values_to = 'values'  )%>%
  group_by(categorical_predictor, values)%>%
  count()%>%
  ungroup()%>%
  ggplot(aes(x = values, y = n))+
  geom_col()+
  facet_wrap(categorical_predictor~., scales = 'free')+
  theme_bw()
```



Regression Tree

In this section, we will examine a regression tree using the default value of cp and a tree tuned for cp

Default Tree

We will establish a baseline for performance by running a default regression tree.

```
library(rpart); library(rpart.plot)
tree = rpart(Balance~.,data = train, method = 'anova')
pred_train = predict(tree)
rmse_train_tree = sqrt(mean((pred_train - train$Balance)^2)); rmse_train_tree
```

```
## [1] 156.4753
```

```
pred = predict(tree, newdata = test)
rmse_tree = sqrt(mean((pred - test$Balance)^2)); rmse_tree
```

```
## [1] 170.4549
```

Tuned Tree

We can tune tree hyperparameters to address the threat of overfitting. The implementation of regression tress in `rpart` has a number of hyperparameters that can be accessed by `rpart.control()`

rpart.control(minsplit = 20, minbucket = round(minsplit/3), cp = 0.01,
maxcompete = 4, maxsurrogate = 5, usesurrogate = 2, xval = 10,
surrogatestyle = 0, maxdepth = 30, ...)

1. We will tune the model using `cp`. This is because we are using the caret framework which makes the tuning process easier but can only tune using cp. To use other hyperparameters, we can write loops to go through the grid.

2. Next, we will specify the range of hyperparameter values for cp. The values to evaluate may be informed by experience. Alternatively, one can use a trial and error approach. When using the latter, it is best to begin grid search with wide intervals to get a general idea of the optimal value of cp and then later focus the grid search with narrower intervals. For instance, one could begin with a range of `seq(0,0.4,0.001)` and then later narrow it down to `seq(0,0.1,0.0001)`

```
tuneGrid = expand.grid(cp = seq(0,0.1,0.0001))
```

3 & 4. Train the model on each value of cp and evaluate using k-fold cross-validation. The caret framework provides a simple interface to doing evaluating a model across a set of hyperparameter values using cross-validation. In the caret function, `trainControl`, method = 'cv' specifies cross-validation and number = 5 specifies the number of folds to use. `train` is a wrapper function that will implement algorithms from the `method` argument and evaluate hyperparameter values specified in the grid by using metrics specified in trControl.

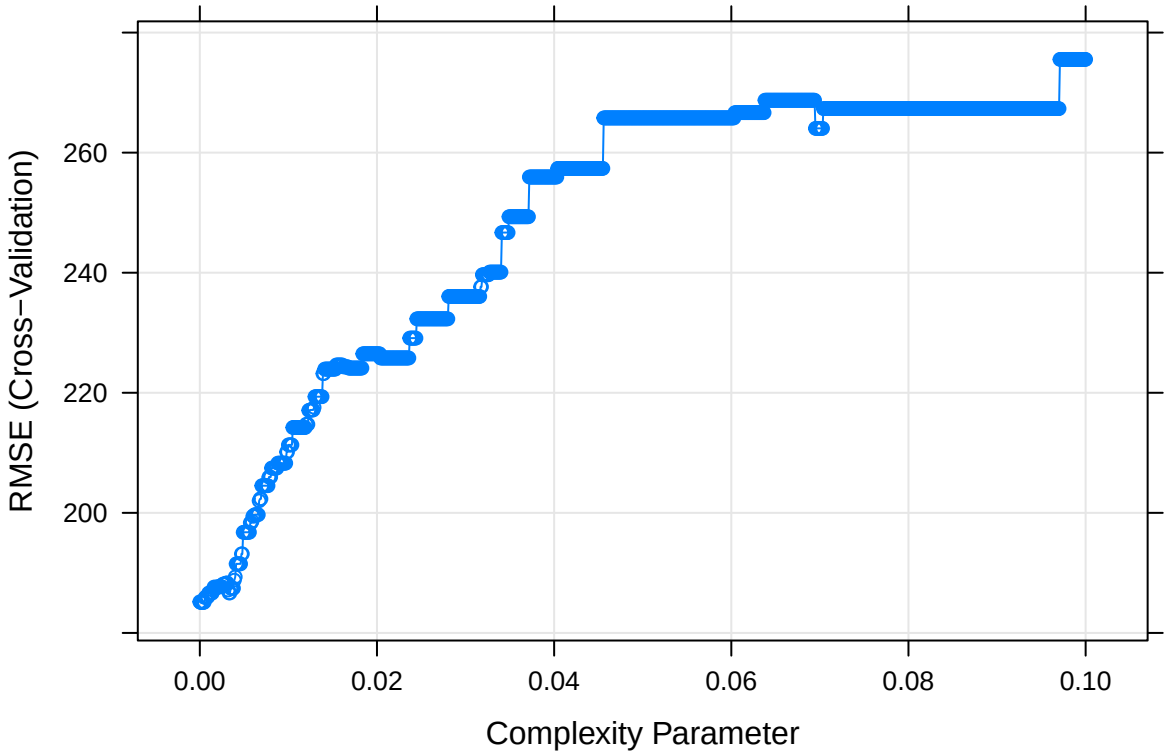
```
library(caret)
trControl = trainControl(method = 'cv',number = 5)
set.seed(1031)
tree_cv = train(Balance~.,
                data = train,
                method = 'rpart',
                trControl = trControl,
                tuneGrid = tuneGrid)
```

5. Examine the cross-validation errors for each value of cp and select the value of cp that yields the lowest cross-validation error.

```
head(tree_cv$results)
```

```
##      cp     RMSE  Rsquared      MAE   RMSESD RsquaredSD    MAESD
## 1 0e+00 185.1799 0.8484916 124.9289 37.11329 0.06255765 24.07179
## 2 1e-04 185.0671 0.8485286 125.1595 37.16187 0.06264025 24.15026
## 3 2e-04 185.0671 0.8485286 125.1595 37.16187 0.06264025 24.15026
## 4 3e-04 185.0671 0.8485286 125.1595 37.16187 0.06264025 24.15026
## 5 4e-04 185.0671 0.8485286 125.1595 37.16187 0.06264025 24.15026
## 6 5e-04 185.1338 0.8483796 125.1595 37.07568 0.06249549 24.15026
```

```
plot(tree_cv)
```



The optimal value for cp is much different from the default of 0.01.

```
tree_cv$bestTune
```

```
##      cp
## 5 4e-04
```

Now, that we have the optimal value of cp, we can use it in a regression tree and construct predictions.

```
cvtree = rpart(Balance~.,
               data = train,
               method = 'anova',
               cp = tree_cv$bestTune)

pred_train = predict(cvtree)
rmse_train_cv_tree = sqrt(mean((pred_train - train$Balance)^2)); rmse_train_cv_tree
```

```
## [1] 123.8085
```

```
pred = predict(cvtree, newdata = test)
rmse_cv_tree = sqrt(mean((pred - test$Balance)^2)); rmse_cv_tree
```

```
## [1] 140.9731
```

Bag

Bag or Bootstrap AGgregation is an ensemble model that aggregates predictions from tree models fitted to a set of bootstrapped samples. Averaging predictions has the benefit of reducing variance in tree models while leaving bias unchanged.

ipred

The `ipred` library is one of many R libraries for estimating bag models. The structure of the modeling function is similar to other R modeling functions. It differs in that one needs to specify the number of bootstrapped samples to fit. For bag models, it is common to use a large number of trees. A seed is specified to ensure reproducibility of results, since bootstrapped samples are generated randomly.

```
library(ipred)
set.seed(1031)
bag = bagging(Balance~.,
              data = train,
              nbagg = 1000)

pred_train = predict(bag)
rmse_train_bag_ipred = sqrt(mean((pred_train - train$Balance)^2)); rmse_train_bag_ipred
```

```
## [1] 166.0304
```

```
pred = predict(bag, newdata = test)
rmse_bag_ipred = sqrt(mean((pred - test$Balance)^2)); rmse_bag_ipred
```

```
## [1] 154.7436
```

randomForest

As the name suggests, this package was not designed for estimating bag models but one can adapt it with a little trick. By specifying mtry to include all predictors, random forest functions like a bag model.

```
library(randomForest)
set.seed(1031)
bag = randomForest(Balance~.,
                   train,
                   mtry = ncol(train)-1,
                   ntree = 1000)

pred_train = predict(bag)
rmse_train_bag_randomforest = sqrt(mean((pred_train - train$Balance)^2)); rmse_train_bag_randomforest
```

```
## [1] 113.2525
```

```
pred = predict(bag, newdata = test)
rmse_bag_randomforest = sqrt(mean((pred - test$Balance)^2)); rmse_bag_randomforest
```

```
## [1] 91.44994
```

Random Forest

Unlike bag models, random forest models use a random subset of features for each bootstrapped tree. This simple tweak helps decorrelate trees and reduces variance when we average trees.

randomForest

The number of predictors used in each tree is set by mtry. For regression problems, this is p/3 and for classification problems, sqrt(p), where p is number of predictors in the train sample.

```
library(randomForest)
set.seed(1031)
forest = randomForest(Balance~.,
                      train,
```

```
                      ntree = 1000)

pred_train = predict(forest)
rmse_train_forest = sqrt(mean((pred_train - train$Balance)^2)); rmse_train_forest
```

```
## [1] 150.1041
```

```
pred_forest = predict(forest, newdata= test)
rmse_forest = sqrt(mean((pred_forest - test$Balance)^2)); rmse_forest
```

```
## [1] 133.1547
```

Tuned randomForest

The default values of mtry may not be optimal. The model may be tuned using k-fold crossvalidation to pick the best value for mtry.

```
library(randomForest)
trControl = trainControl(method = 'cv', number = 5)
tuneGrid = expand.grid(mtry = 1:ncol(train)-1)
set.seed(1031)
forest_cv = train(Balance~.,
                  data = train,
                  method = 'rf',
                  trControl = trControl,
                  tuneGrid = tuneGrid,
                  ntree = 1000)
forest_cv$bestTune$mtry
```

```
## [1] 10
```

In this case, the optimal value for `mtry` is the same as p. This effectively reduces a random forest model to the bag model we ran above.

```
set.seed(1031)
cvforest = randomForest(Balance~.,
                        train,
                        mtry = forest_cv$bestTune$mtry,
                        ntree = 1000)

pred_train = predict(cvforest)
rmse_train_cv_forest = sqrt(mean((pred_train - train$Balance)^2)); rmse_train_cv_forest
```

```
## [1] 113.2525
```

```
pred_forest = predict(cvforest, newdata= test)
rmse_cv_forest = sqrt(mean((pred_forest - test$Balance)^2)); rmse_cv_forest
```

```
## [1] 91.44994
```

ranger

A fast implementation of Random Forests, particularly suited for high dimensional data.

```
library(ranger)
set.seed(1031)
forest_ranger = ranger(Balance~.,
                       data = train,
                       num.trees = 1000)
```

```
pred_train = predict(forest_ranger, data = train, num.trees = 1000)
rmse_train_forest_ranger = sqrt(mean((pred_train$predictions - train$Balance)^2)); rmse_train_forest_ra
```

```
## [1] 68.26044
```

```
pred = predict(forest_ranger, data = test, num.trees = 1000)
rmse_forest_ranger = sqrt(mean((pred$predictions - test$Balance)^2)); rmse_forest_ranger
```

```
## [1] 133.0266
```

tuned ranger

To derive value from `ranger`, it is important to tune model hyperparameters. Here we are going to tune mtry, splitrule and min.node.size with 5-fold cross-validation using the `caret` framework.

```
trControl=trainControl(method="cv",number=5)
tuneGrid = expand.grid(mtry=1:ncol(train)-1,
                       splitrule = c('variance','extratrees','maxstat'),
                       min.node.size = c(2,5,10,15,20,25))
set.seed(1031)
cvModel = train(Balance~.,
                data=train,
                method="ranger",
                num.trees=1000,
                trControl=trControl,
                tuneGrid=tuneGrid)
cvModel$bestTune
```

```
##     mtry  splitrule min.node.size
## 187   10 extratrees             2
```

Now, that we have the best combination of hyperparameters, we can use this to fit a random forest model and make predictions.

```
set.seed(1031)
cv_forest_ranger = ranger(Balance~.,
                          data=train,
                          num.trees = 1000,
                          mtry=cvModel$bestTune$mtry,
                          min.node.size = cvModel$bestTune$min.node.size,
                          splitrule = cvModel$bestTune$splitrule)

pred_train = predict(cv_forest_ranger, data = train, num.trees = 1000)
rmse_train_cv_forest_ranger = sqrt(mean((pred_train$predictions - train$Balance)^2)); rmse_train_cv_for
```

```
## [1] 40.29017
```

```
pred = predict(cv_forest_ranger, data = test, num.trees = 1000)
rmse_cv_forest_ranger = sqrt(mean((pred$predictions - test$Balance)^2)); rmse_cv_forest_ranger
```

```
## [1] 89.81588
```

Boost

Like bag and forest models, boosting models are ensemble models that derive predictions from a number of trees. The key difference is that in boosting, trees are grown sequentially, each tree is grown using information from previously grown trees. Thus, boosting can be seen as a slow learning evolutionary model. Since we are predicting a numerical variable, earn, the distribution is set to 'gaussian'. Had the goal been to predict a binary outcome, we would have set distribution to 'bernoulli'.

gbm

We will begin with a simple boosting model with reasonable values for model hyperparameters.

```
library(gbm)
set.seed(617)
boost = gbm(Balance~.,
            data=train,
            distribution="gaussian",
            n.trees = 500,
            interaction.depth = 2,
            shrinkage = 0.01)
pred_train = predict(boost, n.trees=500)
rmse_train_boost = sqrt(mean((pred_train - train$Balance)^2)); rmse_train_boost
```

```
## [1] 129.3212
```

```
pred = predict(boost, newdata = test, n.trees = 500)
rmse_boost = sqrt(mean((pred - test$Balance)^2)); rmse_boost
```

```
## [1] 132.621
```

tuned gbm

```
library(caret)
set.seed(1031)
trControl = trainControl(method="cv",number=5)
tuneGrid = expand.grid(n.trees = 500,
                       interaction.depth = c(1,2,3),
                       shrinkage = (1:100)*0.001,
                       n.minobsinnode=c(5,10,15))
garbage = capture.output(cvModel <- train(Balance~.,
                                          data=train,
                                          method="gbm",
                                          trControl=trControl,
                                          tuneGrid=tuneGrid))
set.seed(1031)
cvboost = gbm(Balance~.,
              data=train,
              distribution="gaussian",
              n.trees=500,
              interaction.depth=cvModel$bestTune$interaction.depth,
              shrinkage=cvModel$bestTune$shrinkage,
              n.minobsinnode = cvModel$bestTune$n.minobsinnode)

pred_train = predict(cvboost, n.trees=500)
rmse_train_cv_boost = sqrt(mean((pred_train - train$Balance)^2)); rmse_train_cv_boost
```

```
## [1] 38.55024
```

```
pred = predict(cvboost, newdata = test, n.trees = 500)
rmse_cv_boost = sqrt(mean((pred - test$Balance)^2)); rmse_cv_boost
```

```
## [1] 72.09727
```

xgboost

XGBoost is an optimized distributed gradient boosting library designed to be highly efficient, flexible and portable.

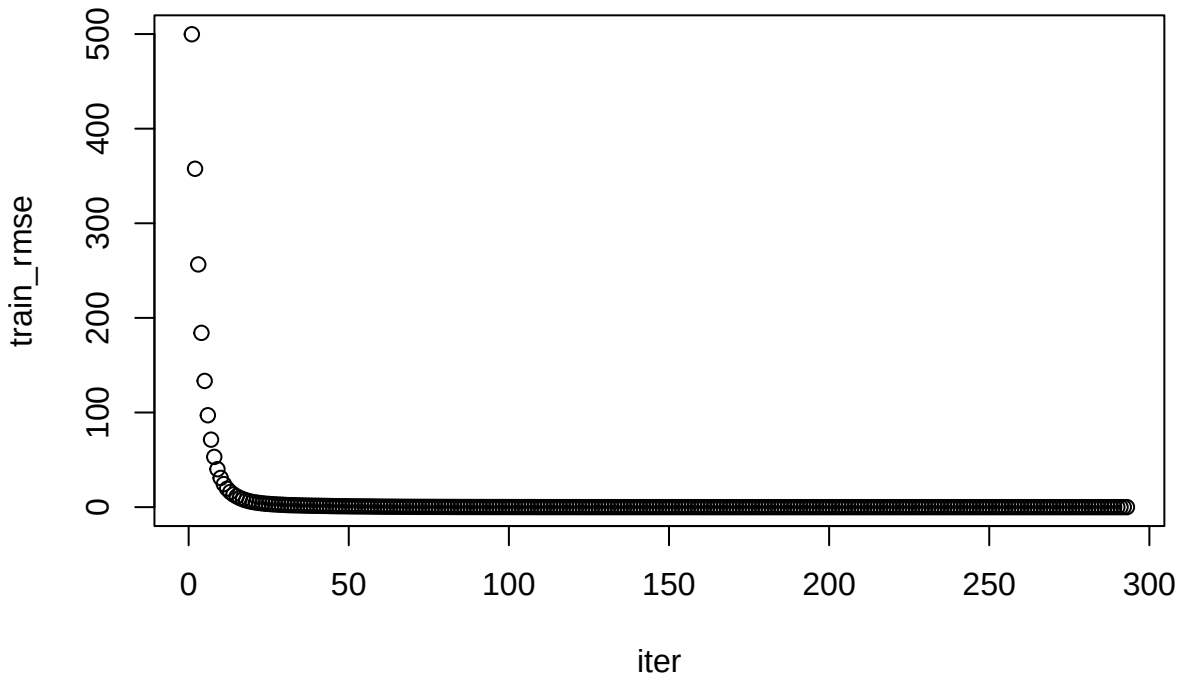
The algorithm is a bit picky about the format of variables used. All factor class variables need to be dummy coded and fed into the model as a matrix. To do this, we will dummy code using library(vtreat).

There are numerous hyperparameters to tune XGBoost, but in this illustration, we will only use defaults. The only additional argument is for early stopping which will stop boosting iterations when the train set RMSE does not improve for 100 boosting iterations. In other words, boosting iterations will stop when train set RMSE does not improve for 100 boosting iterations or when the value of 10000 for nrounds is reached.

```
library(xgboost)
xgboost = xgboost(data=as.matrix(train_input),
                  label = train$Balance,
                  nrounds=10000,
                  verbose = 0,
                  early_stopping_rounds = 100)
xgboost$best_iteration
```

```
## [1] 193
```

```
plot(xgboost$evaluation_log)
```



The above chart shows that the best result was achieved at 193 boosting iterations.

```
pred_train = predict(xgboost,
                     newdata=as.matrix(train_input))
rmse_train_xgboost = sqrt(mean((pred_train - train$Balance)^2)); rmse_train_xgboost
```

```
## [1] 0.001623366
```

```
pred = predict(xgboost,
               newdata=as.matrix(test_input))
rmse_xgboost = sqrt(mean((pred - test$Balance)^2)); rmse_xgboost
```

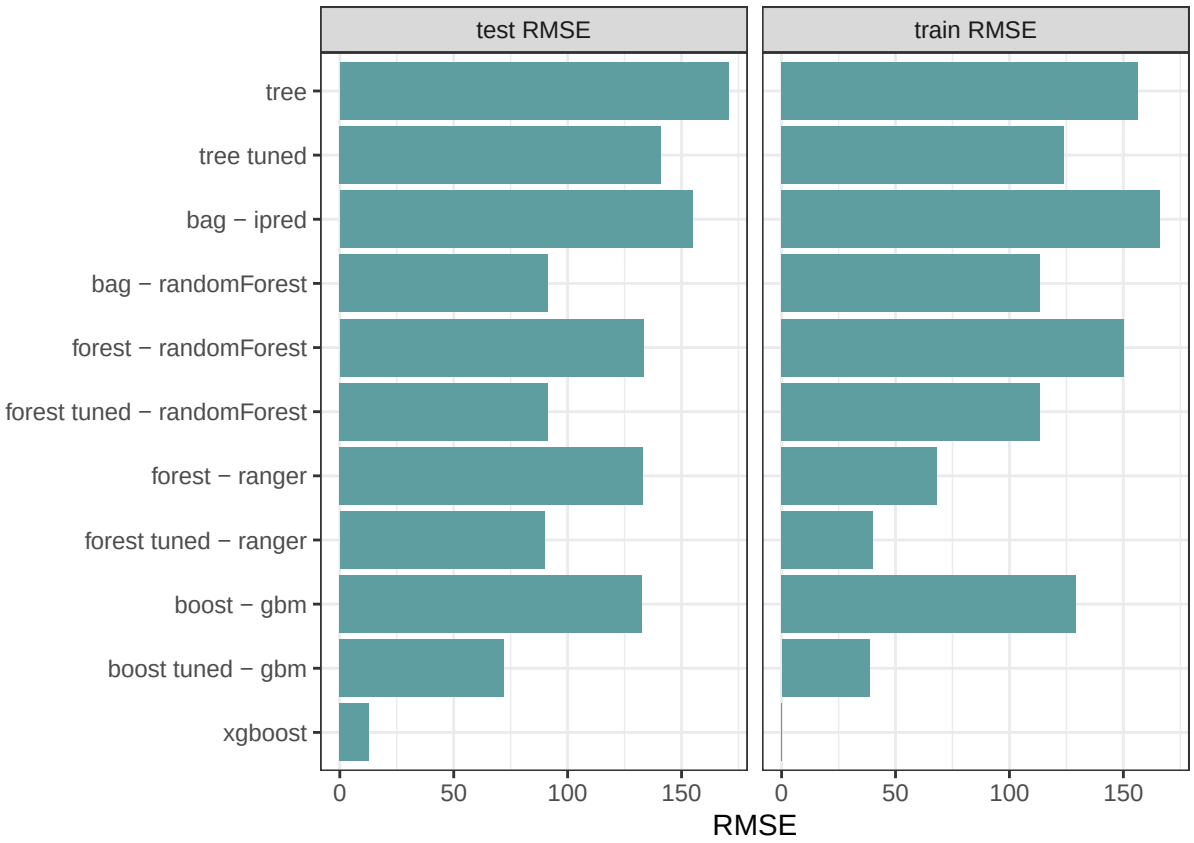
```
## [1] 12.75401
```

Results

```
data.frame(
  id = 1:11,
  model = c('tree','tree tuned','bag - ipred', 'bag - randomForest','forest - randomForest','forest tun
  rmse_train = c(rmse_train_tree, rmse_train_cv_tree, rmse_train_bag_ipred, rmse_train_bag_randomforest
  rmse = c(rmse_tree, rmse_cv_tree, rmse_bag_ipred, rmse_bag_randomforest, rmse_forest, rmse_cv_forest,
  mutate(rmse_train = round(rmse_train,3),
         rmse = round(rmse,3))%>%
  rename('train RMSE' = rmse_train, 'test RMSE' = rmse)
```

```
##    id                       model train RMSE test RMSE
## 1   1                        tree    156.475   170.455
## 2   2                  tree tuned    123.808   140.973
## 3   3                 bag - ipred    166.030   154.744
## 4   4          bag - randomForest    113.253    91.450
## 5   5       forest - randomForest    150.104   133.155
## 6   6 forest tuned - randomForest    113.253    91.450
## 7   7             forest - ranger     68.260   133.027
## 8   8       forest tuned - ranger     40.290    89.816
## 9   9                 boost - gbm    129.321   132.621
## 10 10           boost tuned - gbm     38.550    72.097
## 11 11                     xgboost      0.002    12.754
```

```
library(dplyr); library(tidyr); library(ggplot2)
data.frame(
  id = 1:11,
  model = c('tree','tree tuned','bag - ipred', 'bag - randomForest','forest - randomForest','forest tun
  rmse_train = c(rmse_train_tree, rmse_train_cv_tree, rmse_train_bag_ipred, rmse_train_bag_randomforest
  rmse = c(rmse_tree, rmse_cv_tree, rmse_bag_ipred, rmse_bag_randomforest, rmse_forest, rmse_cv_forest,
#  mutate(rmse_train = round(rmse_train,3),rmse = round(rmse,3))%>%
  rename('train RMSE' = rmse_train, 'test RMSE' = rmse)%>%
  pivot_longer(cols = 3:4,names_to = 'Sample', values_to = 'RMSE')%>%
  ggplot(aes(x=reorder(model,desc(id)), y = RMSE))+
  geom_col(fill = 'cadetblue')+
  xlab('')+
  coord_flip()+
  theme_bw()+
  facet_wrap(~Sample)
```



In conclusion,
1. Performance of trees can be greatly improved by aggregating trees
2. Tuning model hyperparameters can unlock a lot of potential

Based on these results, it may be tempting to declare a winner, however such a conclusion would be myopic. Recall the No Free Lunch Theorem states While certain models work with certain data characteristics, they may fail with different data characteristics.