Regression Trees and Ensemble Methods

Yifei Sun

Contents

```r
library(ISLR)
library(caret)
library(rpart)
library(rpart.plot)
library(party)
library(partykit)
library(randomForest)
library(ranger)
library(gbm)
library(plotmo)
library(pdp)
library(lime)
```

Predict a baseball player's salary on the basis of various statistics associated with performance in the previous year. Use `?Hitters` for more details.

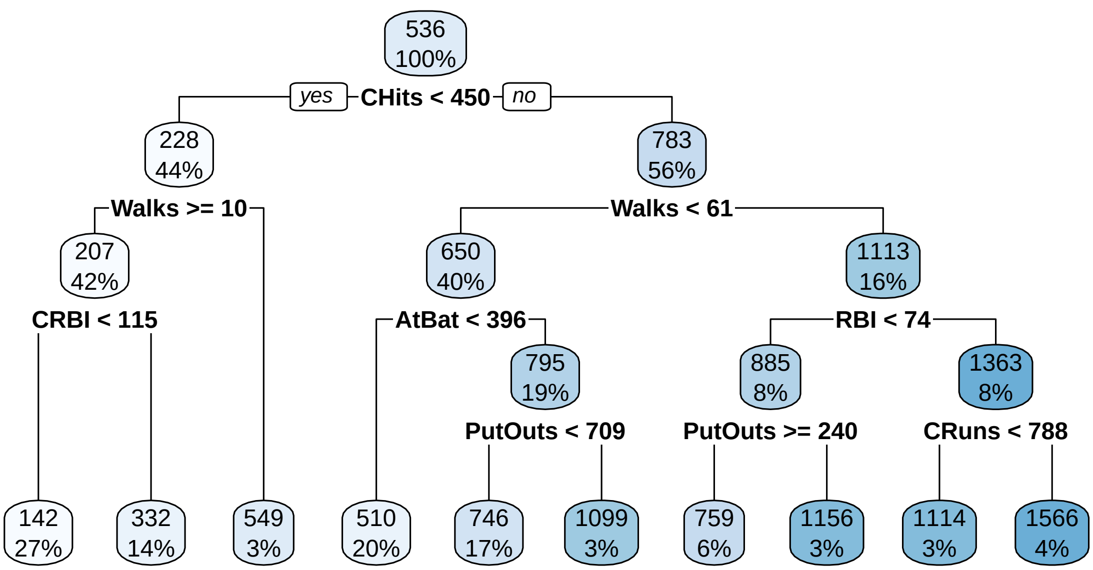
```r
data(Hitters)
Hitters2 <- Hitters[is.na(Hitters$Salary),] # players with missing outcome
Hitters <- na.omit(Hitters)
```

Regression Trees

The CART approach

We first apply the regression tree method to the Hitters data. `cp` is the complexity parameter. The default value for `cp` is 0.01.

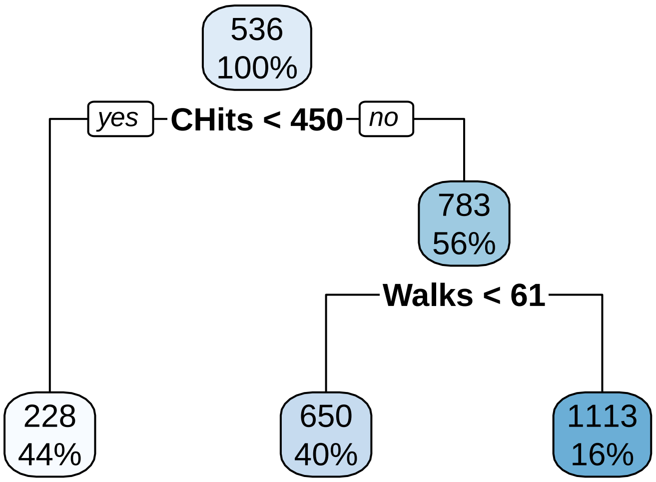
```r
set.seed(1)
tree1 <- rpart(formula = Salary~., data = Hitters)
rpart.plot(tree1)
```



We get a smaller tree by increasing the complexity parameter.

```r
set.seed(1)
tree2 <- rpart(Salary~., Hitters,
```

```
                control = rpart.control(cp = 0.1))
rpart.plot(tree2)
```



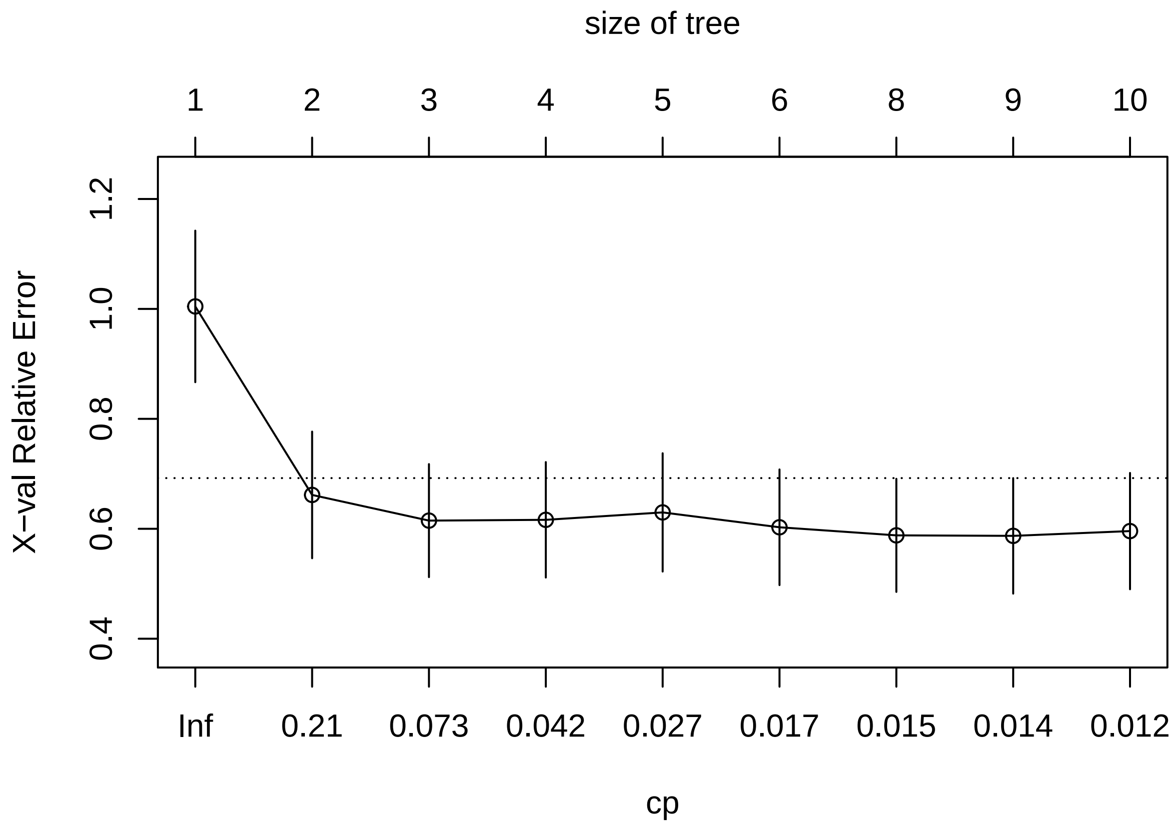
We next apply cost complexity pruning to obtain a tree with the right size. The functions `printcp()` and `plotcp()` give the set of possible cost-complexity prunings of a tree from a nested set. For the geometric means of the intervals of values of `cp` for which a pruning is optimal, a cross-validation has been done in the initial construction by `rpart()`.

The `cptable` in the fit contains the mean and standard deviation of the errors in the cross-validated prediction against each of the geometric means, and these are plotted by `plotcp()`. `Rel error` (relative error) is 1˘R^2. The x-error is the cross-validation error generated by built-in cross validation. A good choice of `cp` for pruning is often the leftmost value for which the mean lies below the horizontal line.

```
cpTable <- printcp(tree1)
```

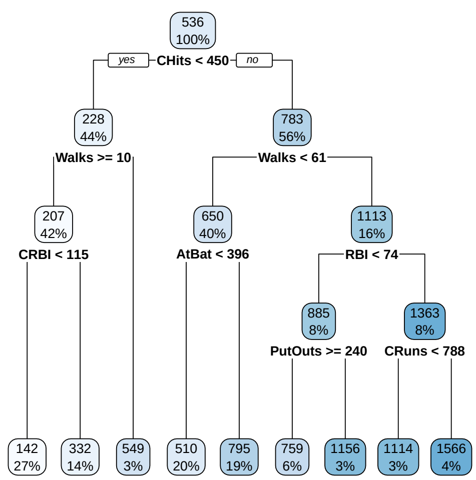
```
##
## Regression tree:
## rpart(formula = Salary ~ ., data = Hitters)
##
## Variables actually used in tree construction:
## [1] AtBat   CHits   CRBI    CRuns   PutOuts RBI     Walks
##
## Root node error: 53319113/263 = 202734
##
## n= 263
##
##         CP nsplit rel error  xerror    xstd
## 1 0.375153      0   1.00000 1.00454 0.13784
## 2 0.120266      1   0.62485 0.66160 0.11523
## 3 0.044776      2   0.50458 0.61487 0.10282
## 4 0.039507      3   0.45981 0.61631 0.10506
## 5 0.018906      4   0.42030 0.62980 0.10763
## 6 0.015646      5   0.40139 0.60274 0.10523
## 7 0.014121      7   0.37010 0.58805 0.10291
## 8 0.014051      8   0.35598 0.58711 0.10509
## 9 0.010000      9   0.34193 0.59581 0.10584
```

```
plotcp(tree1)
```

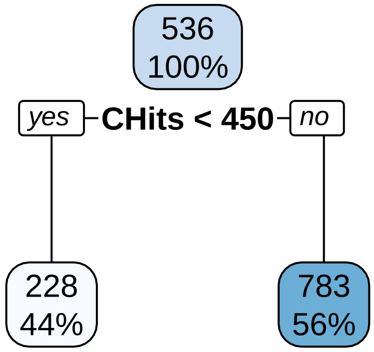


Prune the tree based on the `cp` table.

```
minErr <- which.min(cpTable[,4])
# minimum cross-validation error
tree3 <- prune(tree1, cp = cpTable[minErr,1])
# 1SE rule
tree4 <- prune(tree1, cp = cpTable[cpTable[,4]<cpTable[minErr,4]+cpTable[minErr,5],1][1])

rpart.plot(tree3)
```




```
rpart.plot(tree4)
```



Finally, the function `predict()` can be used for prediction from a fitted `rpart` object.

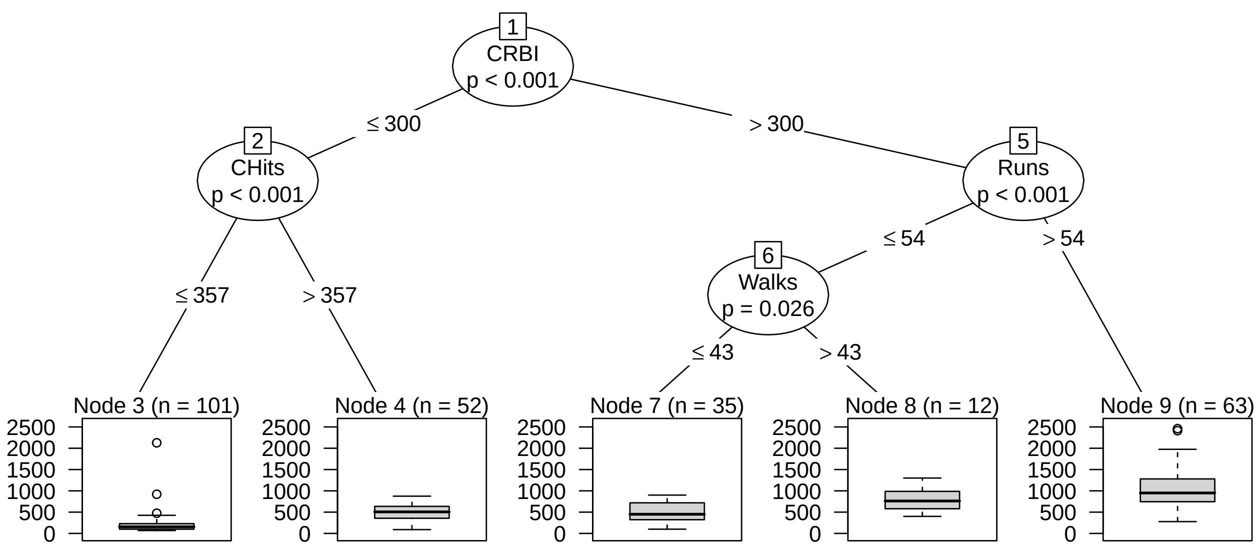
```
predict(tree3, newdata = Hitters2[1:5,])
```

```
## -Andy Allanson   -Billy Beane  -Bruce Bochte     -Bob Boone    -Bobby Grich
##        141.6343       141.6343       758.8889       548.5476       510.0157
```

Conditional inference trees

The implementation utilizes a unified framework for conditional inference, or permutation tests. Unlike CART, the stopping criterion is based on p-values. A split is implemented when (1 - p-value) exceeds the value given by `mincriterion` as specified in `ctree_control()`. This approach ensures that the right-sized tree is grown without additional pruning or cross-validation, but can stop early. At each step, the splitting variable is selected as the input variable with strongest association to the response (measured by a p-value corresponding to a test for the partial null hypothesis of a single input variable and the response). Such a splitting procedure can avoid a variable selection bias towards predictors with many possible cutpoints.

```
tree5 <- ctree(Salary~., Hitters)
plot(tree5)
```



Note that `tree5` is a `party` object. The function `predict()` can be used for prediction from a fitted `party` object.

```
predict(tree5, newdata = Hitters2[1:5,])
```

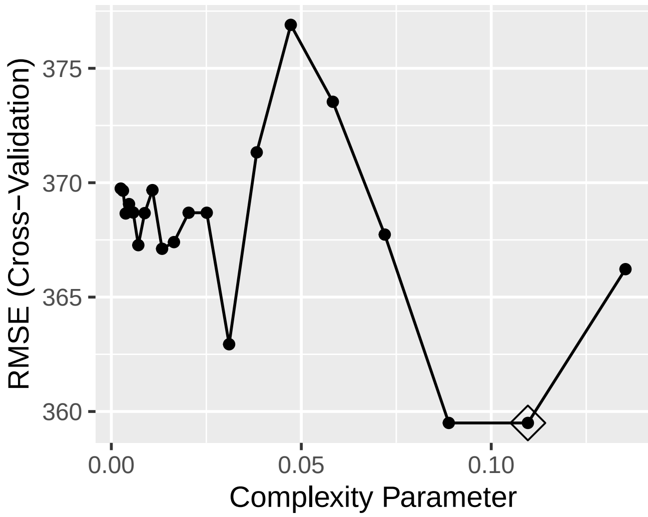
```
## -Andy Allanson   -Billy Beane  -Bruce Bochte     -Bob Boone    -Bobby Grich
```

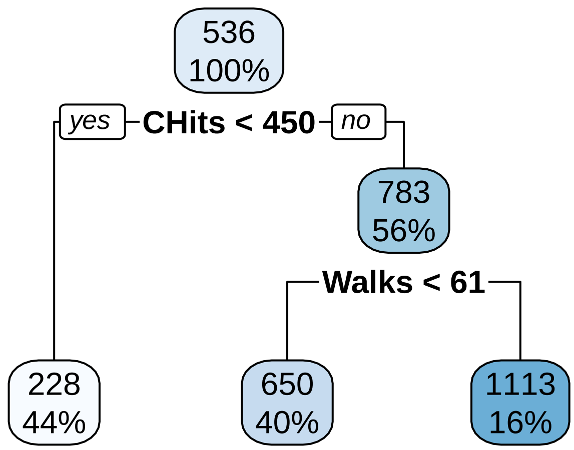
```
##        202.2525        202.2525       1062.9419        202.2525        505.7619
```

Using caret

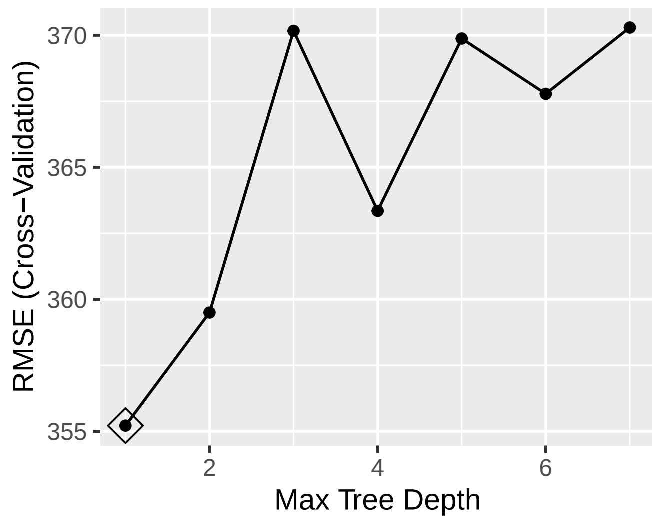
There are two options for CART: tuning over `cp` and tuning over `maxdepth`.

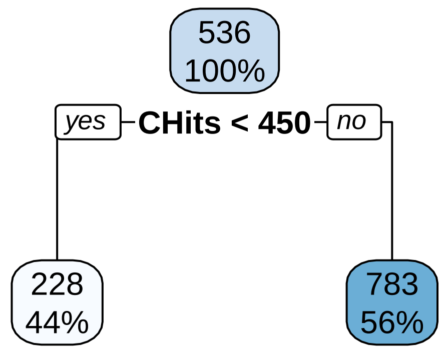
```
ctrl <- trainControl(method = "cv")

set.seed(1)
# tune over cp, method = "rpart"
rpart.fit <- train(Salary~., Hitters,
                   method = "rpart",
                   tuneGrid = data.frame(cp = exp(seq(-6,-2, length = 20))),
                   trControl = ctrl)
ggplot(rpart.fit, highlight = TRUE)
rpart.plot(rpart.fit$finalModel)
```





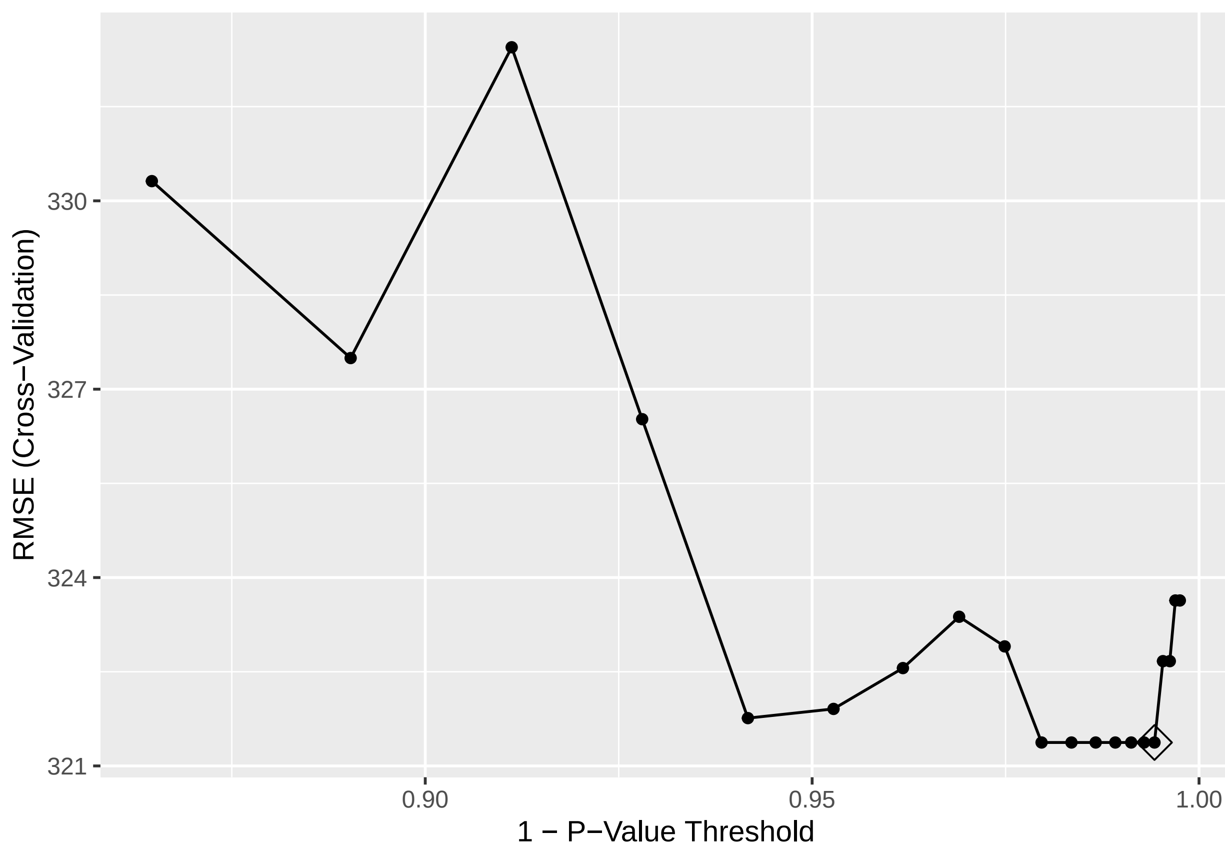

```
set.seed(1)
# tune over maximum depth, method = "rpart2"
rpart2.fit <- train(Salary~., Hitters,
                   method = "rpart2",
                   tuneGrid = data.frame(maxdepth = 1:7),
                   trControl = ctrl)
ggplot(rpart2.fit, highlight = TRUE)
rpart.plot(rpart2.fit$finalModel)
```



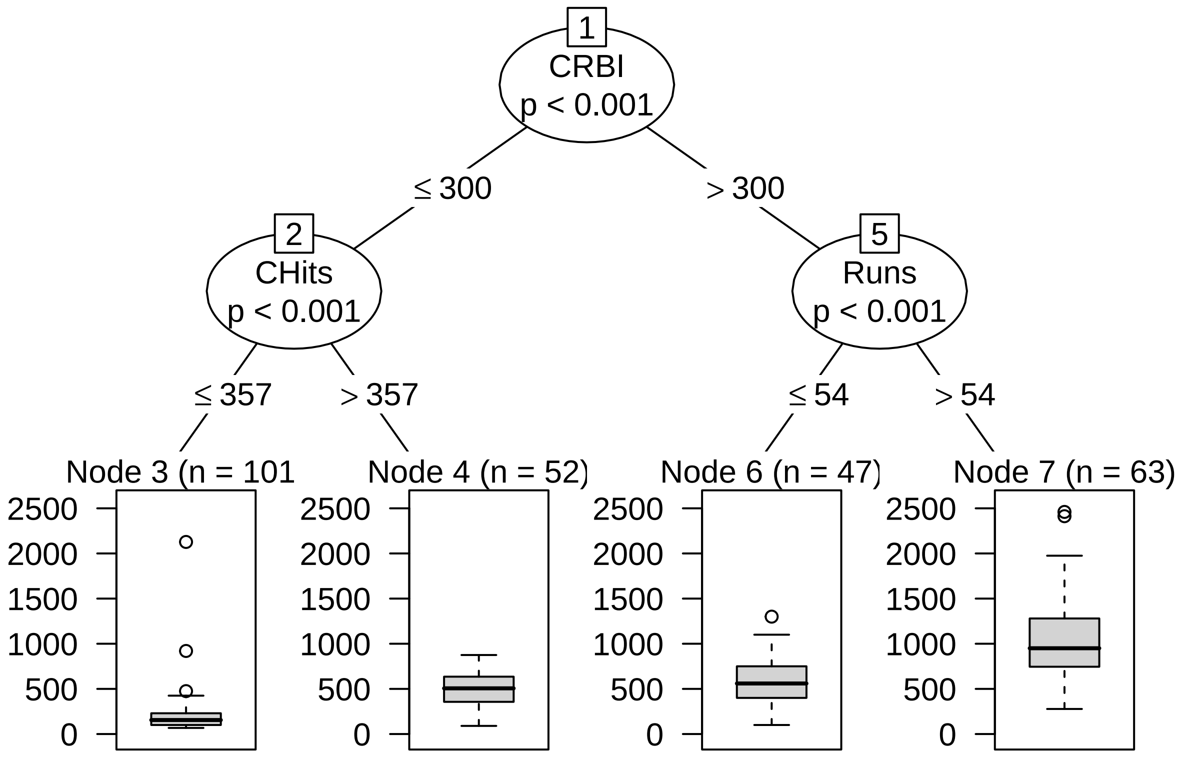


We can also fit a conditional inference tree model. The tuning parameter is `mincriterion`.

```
set.seed(1)
ctree.fit <- train(Salary~., Hitters,
                   method = "ctree",
                   tuneGrid = data.frame(mincriterion = 1-exp(seq(-6, -2, length = 20))),
                   trControl = ctrl)
ggplot(ctree.fit, highlight = TRUE)
```




```
plot(ctree.fit$finalModel)
```



Ensemble methods

Bagging and Random forests

The function `randomForest()` implements Breiman's random forest algorithm (based on Breiman and Cutler's original Fortran code) for classification and regression. `ranger()` is a fast implementation of Breiman's random forests, particularly suited for high dimensional data.

```
set.seed(1)
bagging <- randomForest(Salary~., Hitters,
                        mtry = 19)

set.seed(1)
rf <- randomForest(Salary~., Hitters,
                        mtry = 6)

# fast implementation
set.seed(1)
rf2 <- ranger(Salary~., Hitters,
              mtry = 6)
# scale permutation importance by standard error

predict(rf, newdata = Hitters2[1:5,])
```

```
## -Andy Allanson    -Billy Beane  -Bruce Bochte      -Bob Boone    -Bobby Grich
##        77.65082        79.15122      839.73987      1014.36042       583.71912
```

```
predict(rf2, data = Hitters2[1:5,])$predictions
```

```
## [1]  78.13878  83.02668 879.65169 981.18513 587.66885
```

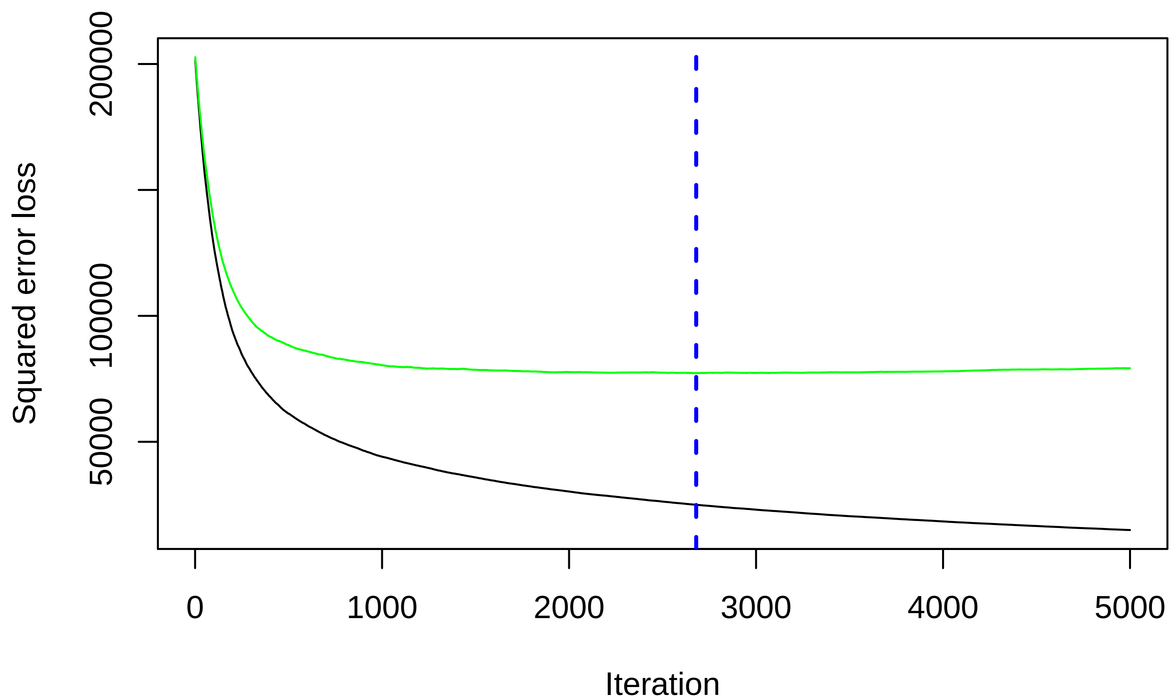
Boosting

We first fit a gradient boosting model with Gaussian loss function with 10000 iterations.

```r
set.seed(1)
bst <- gbm(Salary~., Hitters,
           distribution = "gaussian",
           n.trees = 5000,
           interaction.depth = 3,
           shrinkage = 0.005,
           cv.folds = 10)
```

We plot loss function as a result of number of trees added to the ensemble.

```r
nt <- gbm.perf(bst, method = "cv")
```




```r
nt
```

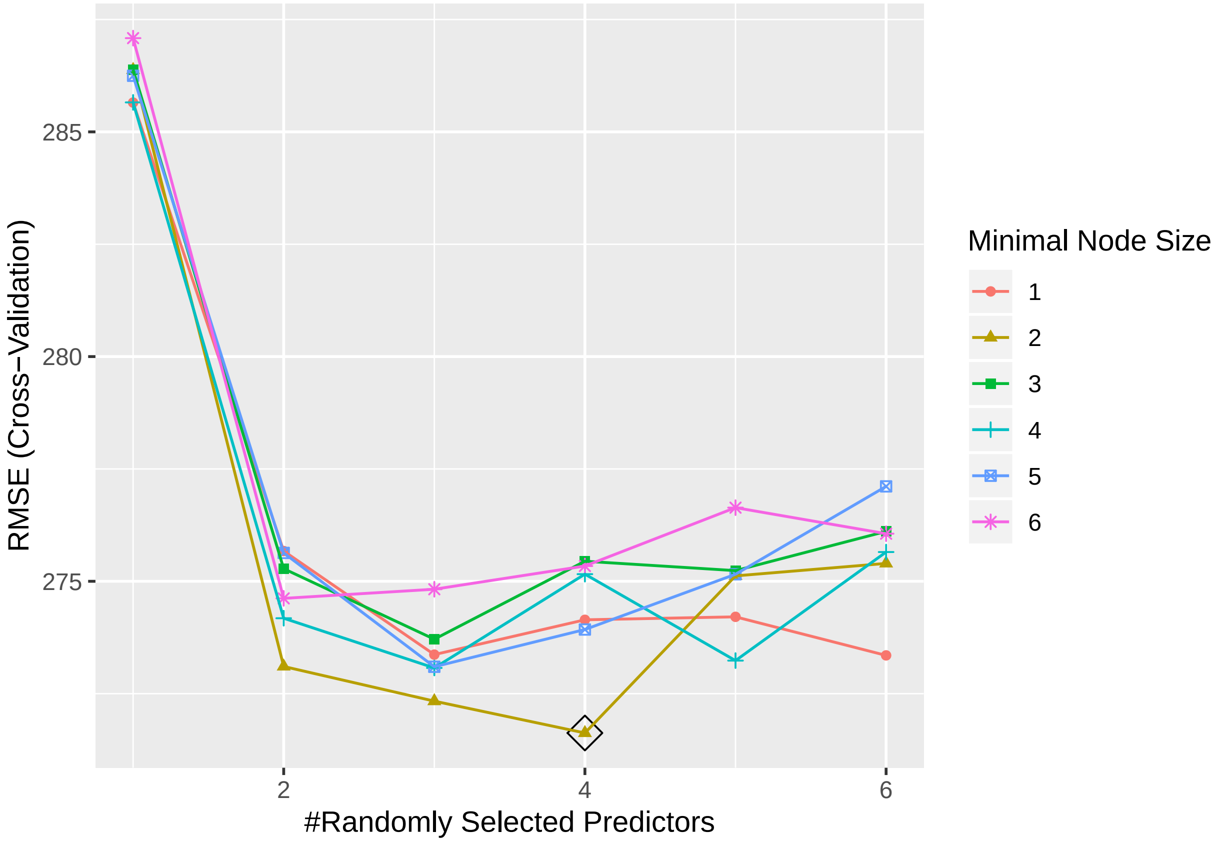
```
## [1] 2680
```

Grid search using caret

We use the fast implementation of random forest when tuning the model.

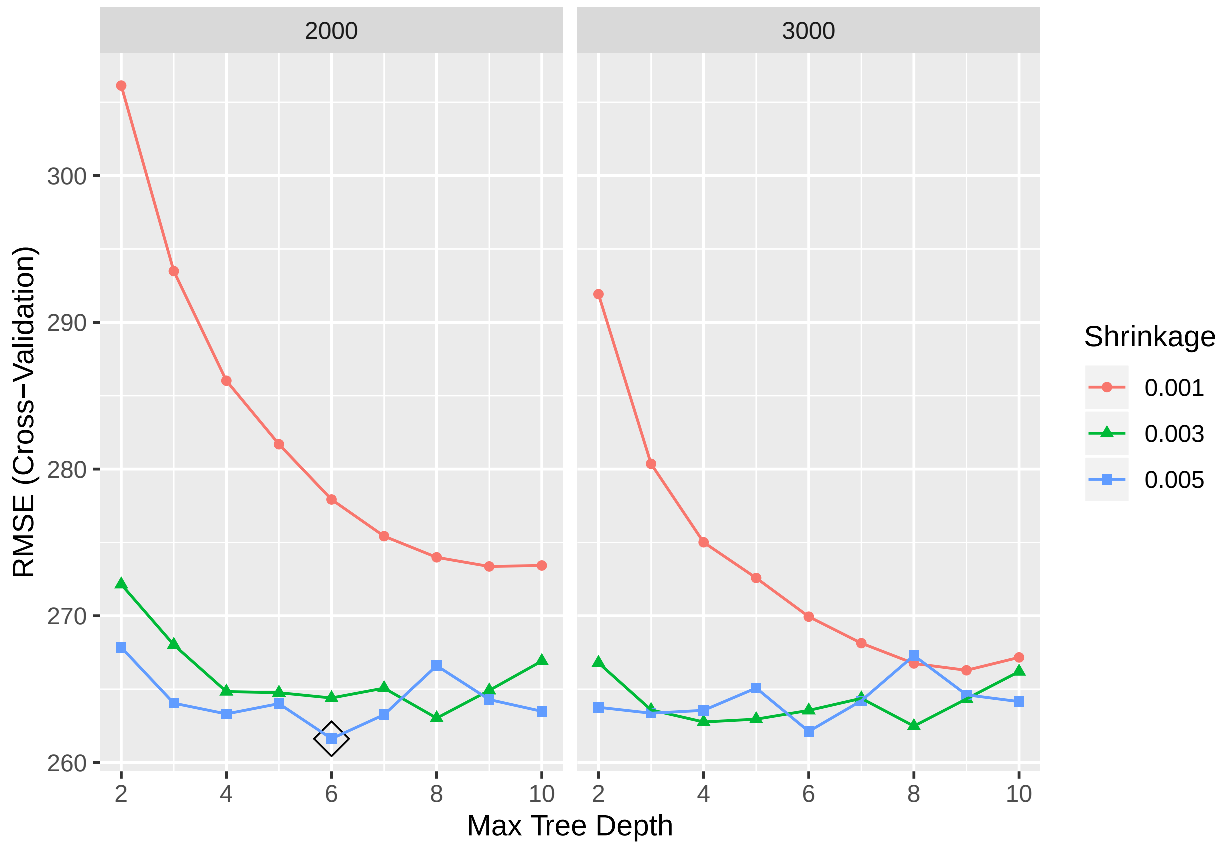
```r
# Try more if possible
rf.grid <- expand.grid(mtry = 1:6,
                       splitrule = "variance",
                       min.node.size = 1:6)
set.seed(1)
rf.fit <- train(Salary~., Hitters,
                method = "ranger",
```

```
                 tuneGrid = rf.grid,
                 trControl = ctrl)

ggplot(rf.fit, highlight = TRUE)
```



We then tune the `gbm` model.

```
# Try more
gbm.grid <- expand.grid(n.trees = c(2000,3000),
                        interaction.depth = 2:10,
                        shrinkage = c(0.001,0.003,0.005),
                        n.minobsinnode = 1)
set.seed(1)
gbm.fit <- train(Salary~., Hitters,
                 method = "gbm",
                 tuneGrid = gbm.grid,
                 trControl = ctrl,
                 verbose = FALSE)

ggplot(gbm.fit, highlight = TRUE)
```



As you can see, it takes a while to train the `gbm` even with a rough tuning grid. The `xgboost` package provides an efficient implementation of gradient boosting framework (apprx 10x faster than `gbm`). You can find much useful information here: https://github.com/dmlc/xgboost/tree/master/demo.

Compare the cross-validation performance. You can also compare with other models that we fitted before.

```r
resamp <- resamples(list(rf = rf.fit, gbm = gbm.fit))
summary(resamp)
```

```
##
## Call:
## summary.resamples(object = resamp)
##
## Models: rf, gbm
## Number of resamples: 10
##
## MAE
##         Min.  1st Qu.   Median     Mean  3rd Qu.     Max. NA's
## rf  131.2957 152.5255 156.7333 164.7457 162.5672 229.4511    0
## gbm 140.7966 146.8811 155.5835 163.6629 173.5095 225.0195    0
##
## RMSE
##         Min.  1st Qu.   Median     Mean  3rd Qu.     Max. NA's
## rf  211.1594 230.7905 263.9181 271.6251 313.8305 338.5279    0
## gbm 185.1412 233.5030 246.9259 261.6234 306.3777 337.9614    0
##
## Rsquared
##          Min.  1st Qu.    Median      Mean   3rd Qu.      Max. NA's
```

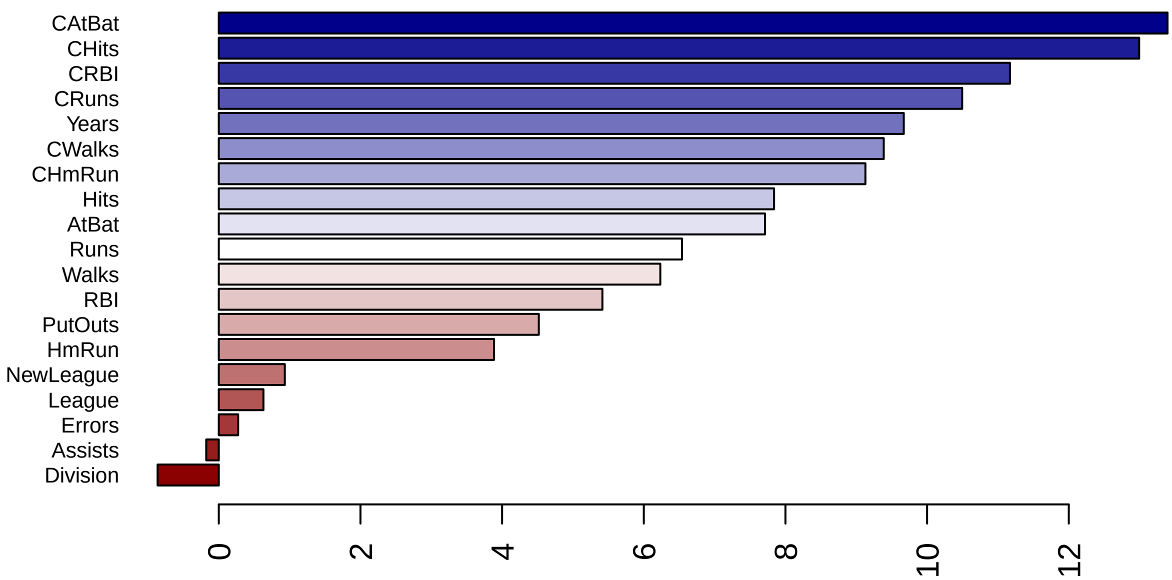
```
## rf  0.4608947 0.583214 0.7091786 0.6602780 0.7428620 0.8057854    0
## gbm 0.4475096 0.600445 0.6993854 0.6892037 0.8088044 0.8515948    0
```

Explain the black-box models

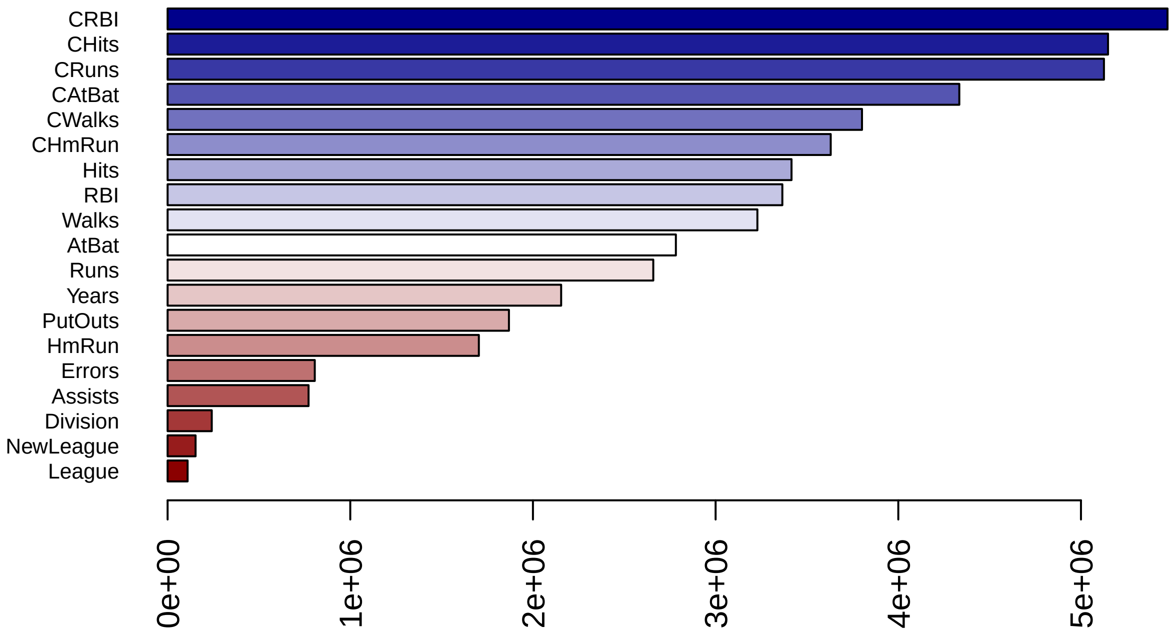
Variable importance

We can extract the variable importance from the fitted models. In what follows, the first measure is computed from permuting OOB data. The second measure is the total decrease in node impurities from splitting on the variable, averaged over all trees. For regression, node impurity is measured by residual sum of squares.

```
set.seed(1)
rf2.final.per <- ranger(Salary~., Hitters,
                        mtry = 3, splitrule = "variance",
                        min.node.size = 5,
                        importance = "permutation",
                        scale.permutation.importance = TRUE)

barplot(sort(ranger::importance(rf2.final.per), decreasing = FALSE),
        las = 2, horiz = TRUE, cex.names = 0.7,
        col = colorRampPalette(colors = c("darkred","white","darkblue"))(19))
```

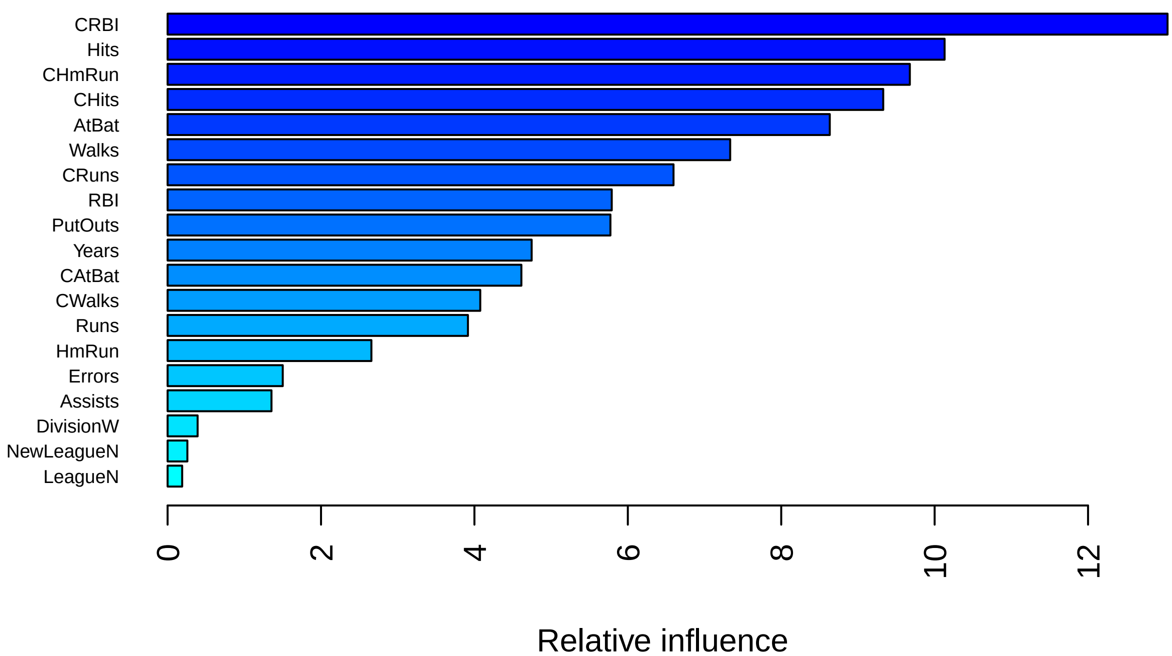



```
set.seed(1)
rf2.final.imp <- ranger(Salary~., Hitters,
                        mtry = 3, splitrule = "variance",
                        min.node.size = 5,
                        importance = "impurity")

barplot(sort(ranger::importance(rf2.final.imp), decreasing = FALSE),
        las = 2, horiz = TRUE, cex.names = 0.7,
        col = colorRampPalette(colors = c("darkred","white","darkblue"))(19))
```



Variable importance from boosting can be obtained using the `summary()` function.

```
summary(gbm.fit$finalModel, las = 2, cBars = 19, cex.names = 0.6)
```




```
##                   var    rel.inf
## CRBI             CRBI 13.0365354
## Hits             Hits 10.1299048
## CHmRun         CHmRun  9.6766261
## CHits           CHits  9.3295588
## AtBat           AtBat  8.6331212
## Walks           Walks  7.3345302
## CRuns           CRuns  6.5956747
## RBI               RBI  5.7902837
```

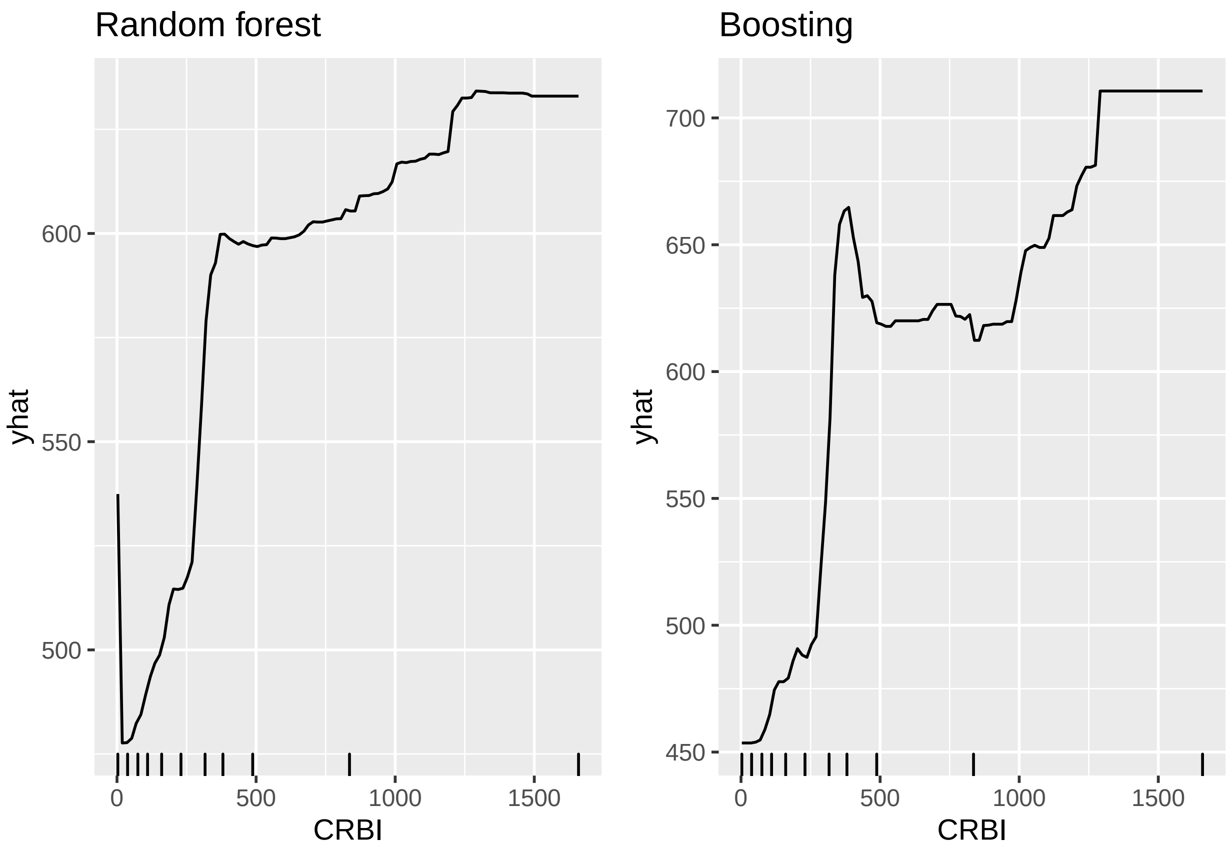
```
## PutOuts       PutOuts  5.7729475
## Years           Years  4.7463003
## CAtBat         CAtBat  4.6120259
## CWalks         CWalks  4.0760442
## Runs             Runs  3.9153142
## HmRun           HmRun  2.6570484
## Errors         Errors  1.5010809
## Assists       Assists  1.3541036
## DivisionW   DivisionW  0.3916078
## NewLeagueN NewLeagueN  0.2576222
## LeagueN       LeagueN  0.1896703
```

Partial dependence plots and individual conditional expectation curves

After the most relevant variables have been identified, the next step is to attempt to understand how the response variable changes based on these variables. For this we can use partial dependence plots (PDPs) and individual conditional expectation (ICE) curves.

PDPs plot the change in the average predicted value as specified feature(s) vary over their marginal distribution. The PDP plot below displays the average change in predicted `Salary` as we vary `CRBI` while holding all other variables constant. This is done by holding all variables constant for each observation in our training data set but then apply the unique values of `CRBI` for each observation. We then average the `Salary` across all the observations.

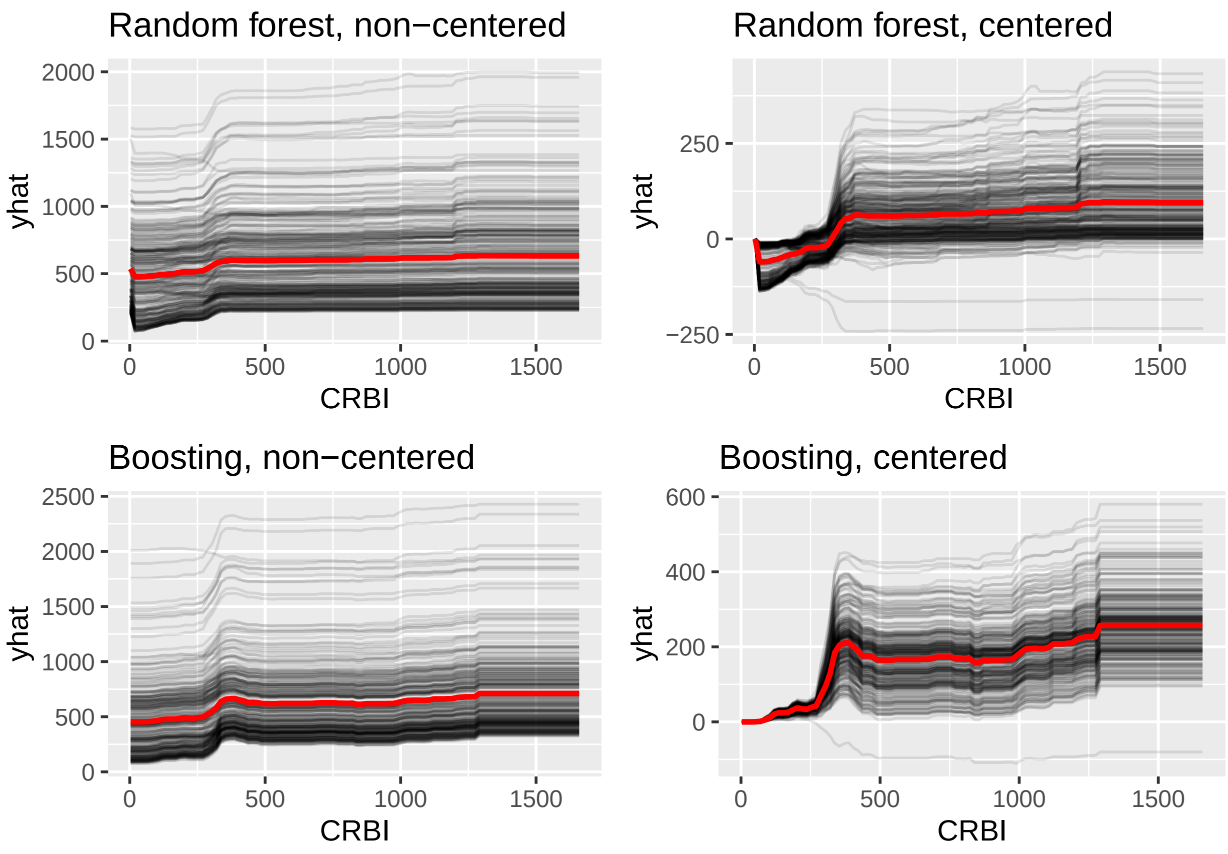
```r
pdp.rf <- rf.fit %>%
  partial(pred.var = "CRBI",
          grid.resolution = 100) %>%
  autoplot(rug = TRUE, train = Hitters) +
  ggtitle("Random forest")

pdp.gbm <- gbm.fit %>%
  partial(pred.var = "CRBI",
          grid.resolution = 100) %>%
  autoplot(rug = TRUE, train = Hitters) +
  ggtitle("Boosting")

grid.arrange(pdp.rf, pdp.gbm, nrow = 1)
```



ICE curves are an extension of PDP plots but, rather than plot the average marginal effect on the response variable, we plot the change in the predicted response variable for each observation as we vary each predictor variable.

```
ice1.rf <- rf.fit %>%
  partial(pred.var = "CRBI",
          grid.resolution = 100,
          ice = TRUE) %>%
  autoplot(train = Hitters, alpha = .1) +
  ggtitle("Random forest, non-centered")

ice2.rf <- rf.fit %>%
  partial(pred.var = "CRBI",
          grid.resolution = 100,
          ice = TRUE) %>%
  autoplot(train = Hitters, alpha = .1,
           center = TRUE) +
  ggtitle("Random forest, centered")

ice1.gbm <- gbm.fit %>%
  partial(pred.var = "CRBI",
          grid.resolution = 100,
          ice = TRUE) %>%
  autoplot(train = Hitters, alpha = .1) +
  ggtitle("Boosting, non-centered")

ice2.gbm <- gbm.fit %>%
  partial(pred.var = "CRBI",
```

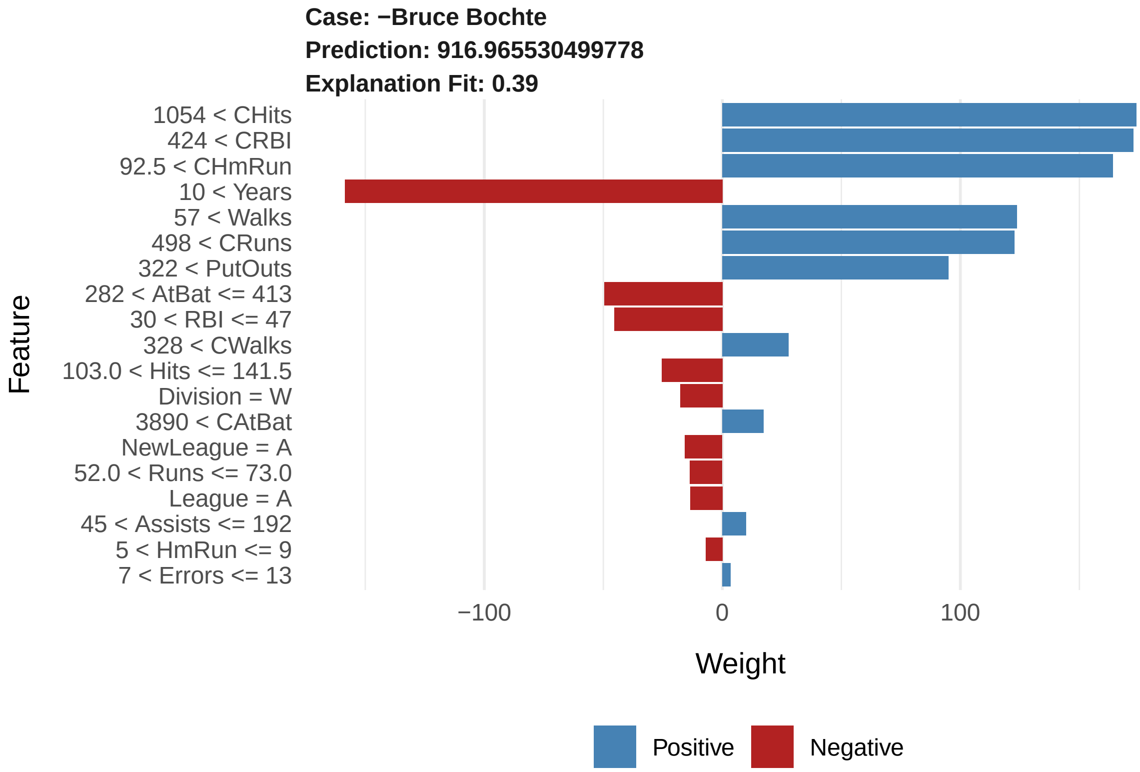
```
grid.resolution = 100,
          ice = TRUE) %>%
  autoplot(train = Hitters, alpha = .1,
           center = TRUE) +
  ggtitle("Boosting, centered")

grid.arrange(ice1.rf, ice2.rf, ice1.gbm, ice2.gbm,
             nrow = 2, ncol = 2)
```



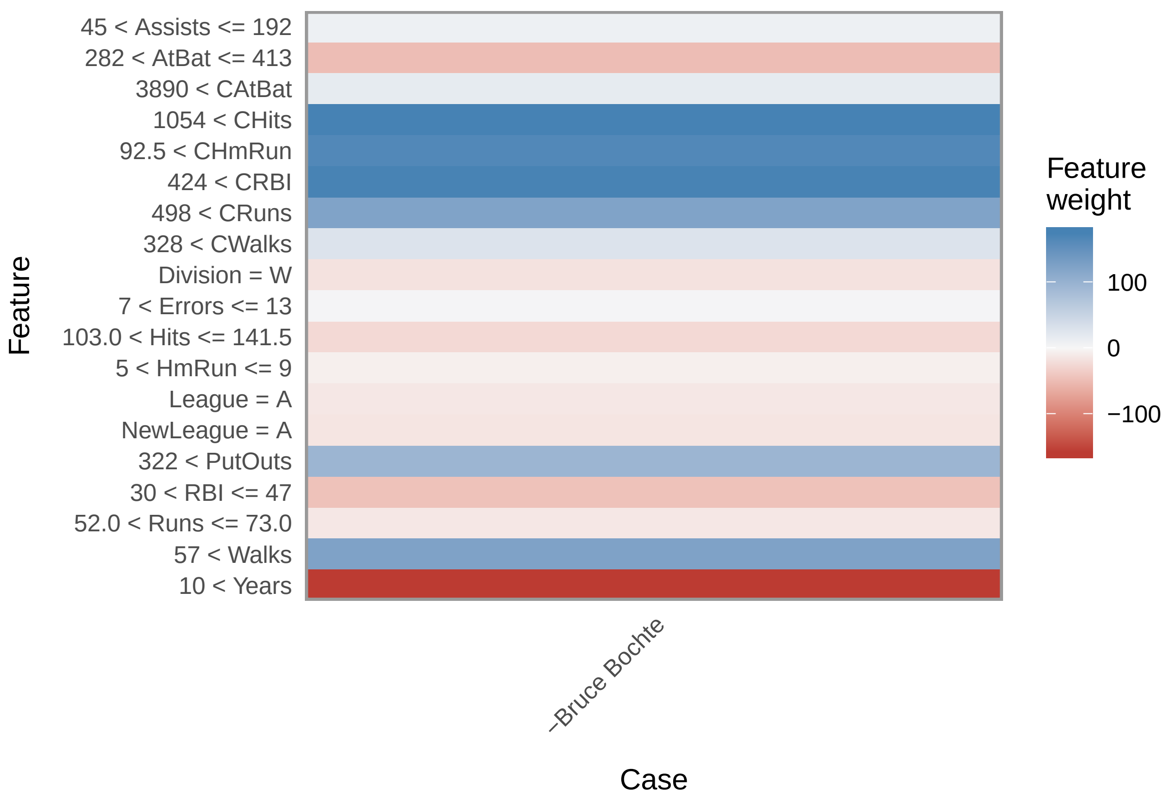
Plot the features in an explanation

The function `plot_features()` creates a compact visual representation of the explanations for each case and label combination in an explanation. Each extracted feature is shown with its weight, thus giving the importance of the feature in the label prediction.

```
new_obs <- Hitters2[3,-19]
explainer.gbm <- lime(Hitters[,-19], gbm.fit)
explanation.gbm <- explain(new_obs, explainer.gbm, n_features = 19)
plot_features(explanation.gbm)
```




```
plot_explanations(explanation.gbm)
```




```
explainer.rf <- lime(Hitters[,-19], rf.fit)
explanation.rf <- explain(new_obs, explainer.rf, n_features = 19)
```

```r
plot_features(explanation.rf)
```