Regression Trees and Ensemble Methods

```r
library(ISLR)
library(caret)
library(rpart) # Make plot of the decision trees
library(rpart.plot)
library(party) # Conditional inference trees
library(partykit) # Visualize the CIT
library(randomForest) # implement the original RF. Was originally written in Fortran. Can be slow
library(ranger) # This is also for RF but was written in C++ and must faster
library(gbm) # For boosting models
library(plotmo)
library(pdp) # to create partial dependence plot
library(lime) # to provide interpreation on why a model gives a prediction.
```

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

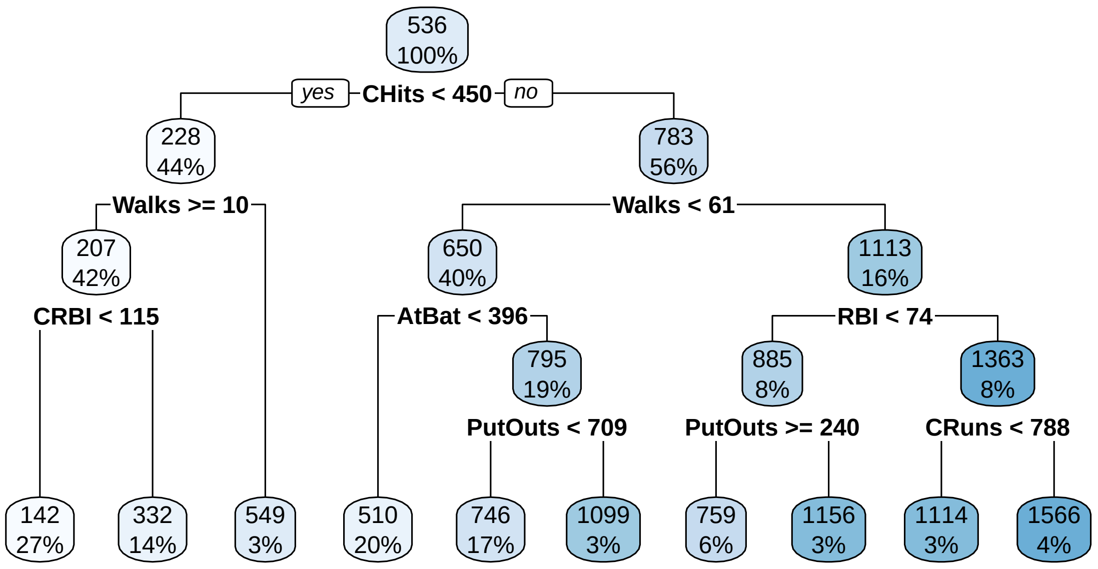
```r
data(Hitters)
Hitters2 <- Hitters[is.na(Hitters$Salary),] # players with missing outcome
Hitters <- na.omit(Hitters)
```

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.

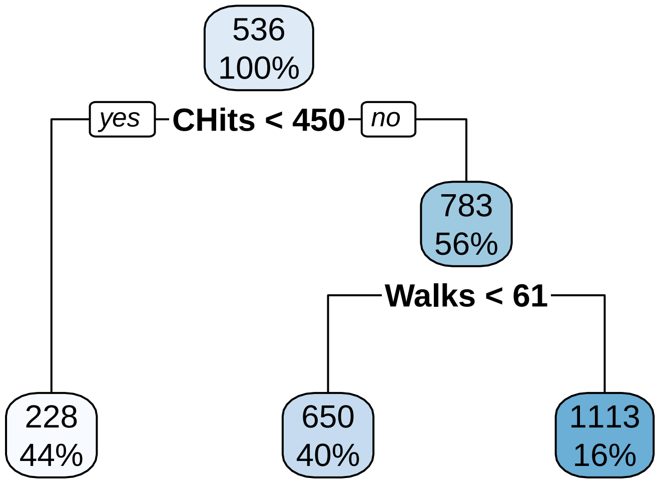
```r
set.seed(1)
tree1 <- rpart(formula = Salary~., data = Hitters)
rpart.plot(tree1)
```



cp of 0.01 gives a large tree so we might consider prunning it further. In the terminal nodes, we have the mean values and the proportion of the data in the terminal nodes. So we have 142 as the mean and 27% of the data are there.

We get a smaller tree by increasing the complexity parameter. When we increase alpha the size of the tree decreases because of a larger penalty term.

```
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 $1\breve{}R^2$. 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.

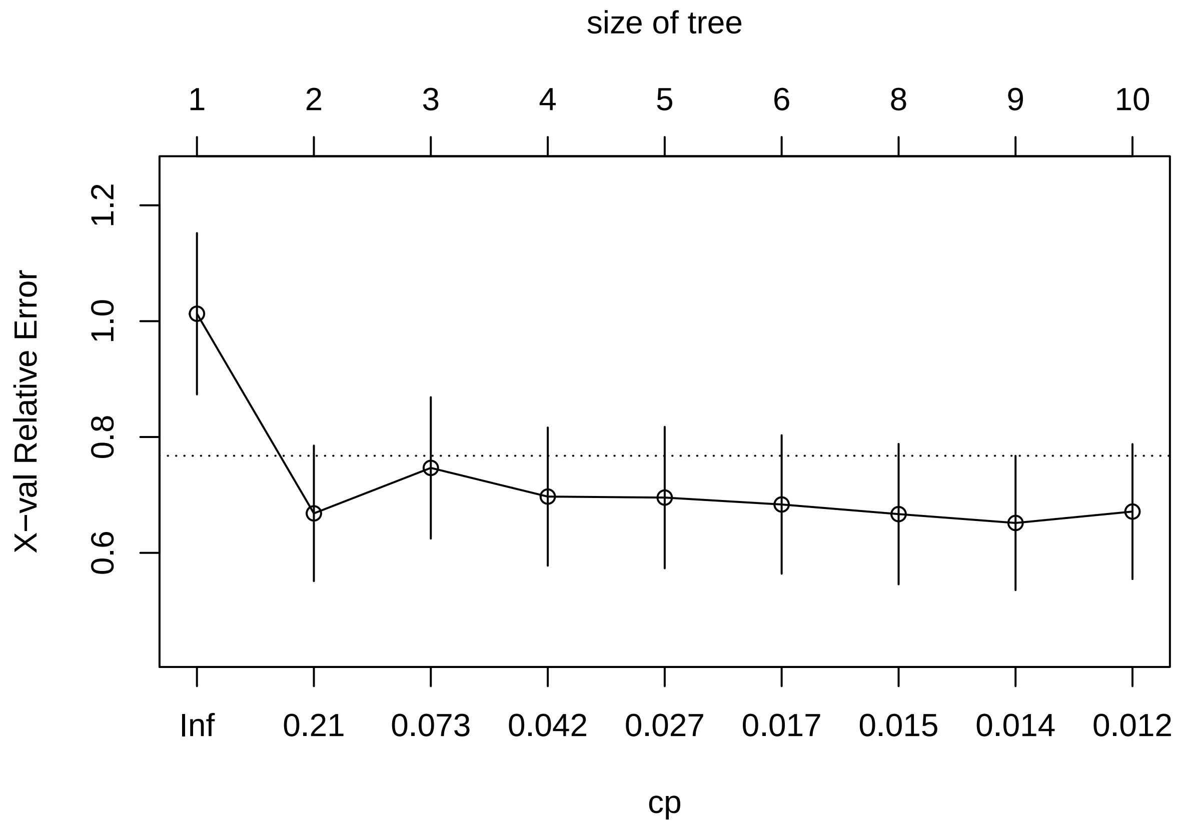
```
cpTable <- printcp(tree1) # This shows the cp values and the size of the tree ( the size is n split + 1
```

```
##
## 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 1.01280 0.13927
## 2 0.120266      1   0.62485 0.66820 0.11700
## 3 0.044776      2   0.50458 0.74669 0.12210
## 4 0.039507      3   0.45981 0.69712 0.11926
## 5 0.018906      4   0.42030 0.69543 0.12211
## 6 0.015646      5   0.40139 0.68351 0.11957
## 7 0.014121      7   0.37010 0.66688 0.12124
## 8 0.014051      8   0.35598 0.65163 0.11597
## 9 0.010000      9   0.34193 0.67125 0.11654
```

```r
cpTable
```

```
##           CP nsplit rel error    xerror      xstd
## 1 0.37515262      0 1.0000000 1.0128006 0.1392745
## 2 0.12026601      1 0.6248474 0.6682047 0.1170041
## 3 0.04477601      2 0.5045814 0.7466870 0.1221034
## 4 0.03950693      3 0.4598054 0.6971185 0.1192557
## 5 0.01890585      4 0.4202984 0.6954315 0.1221117
## 6 0.01564595      5 0.4013926 0.6835126 0.1195739
## 7 0.01412095      7 0.3701007 0.6668765 0.1212356
## 8 0.01405067      8 0.3559797 0.6516281 0.1159719
## 9 0.01000000      9 0.3419291 0.6712474 0.1165445
```

```r
plotcp(tree1)
```

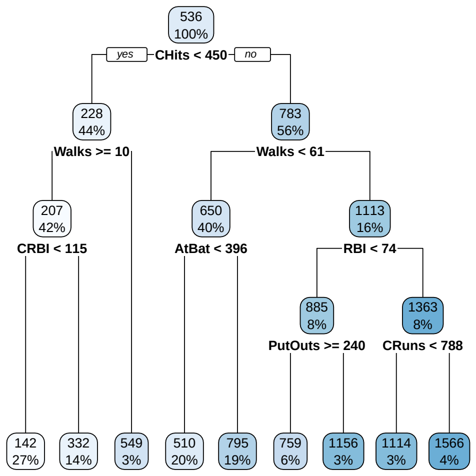


From the plot the minimun is 0.014. We also use the one standard error rule. Based on that we can choose 0.21.

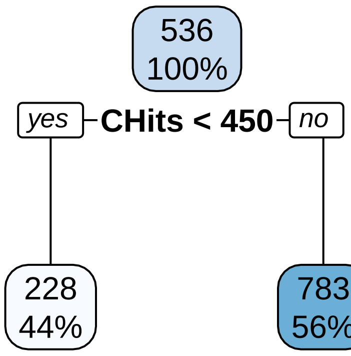
Prune the tree based on the `cp` table.

```r
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])
```

```r
rpart.plot(tree3)
```



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



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

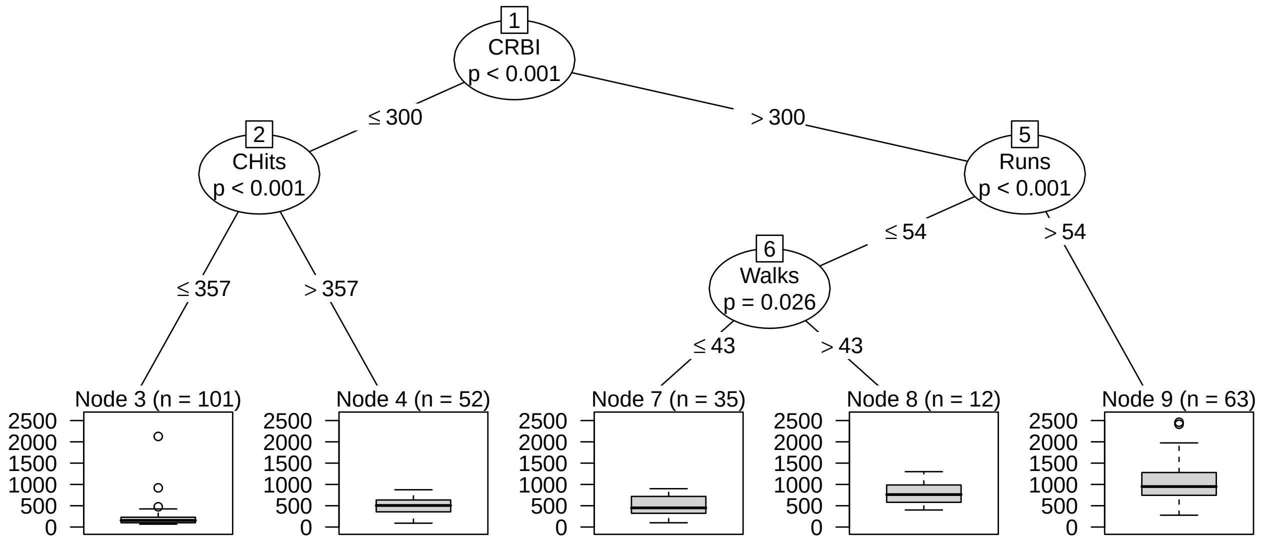
```r
predict(tree3, newdata = Hitters2[1:5,])
```

```
## -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.

```r
tree5 <- ctree(Salary~., Hitters) # This gives you are party object.
plot(tree5)
```



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

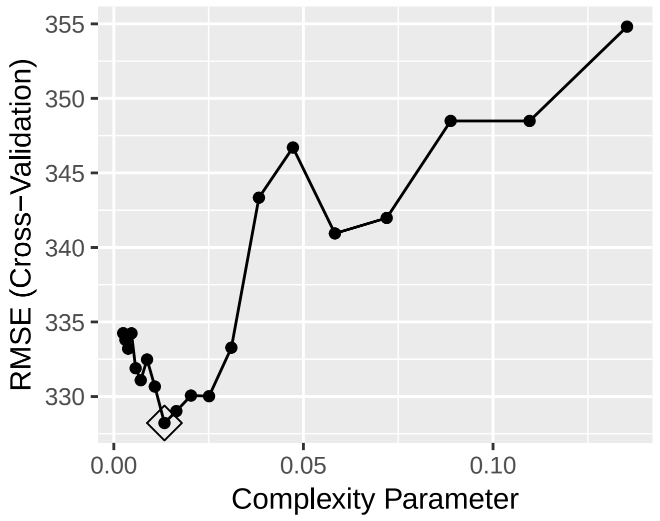
```
predict(tree5, newdata = Hitters2[1:5,])
```

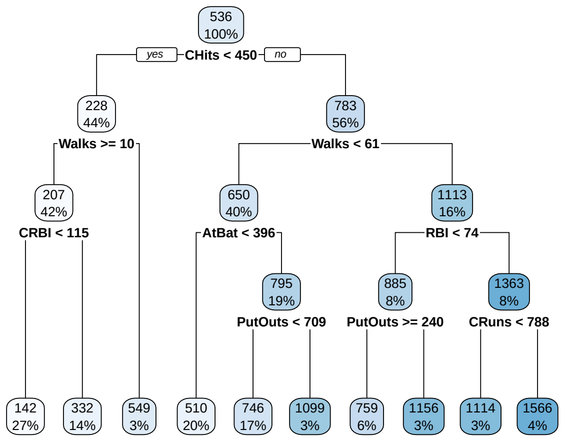
```
## -Andy Allanson    -Billy Beane  -Bruce Bochte      -Bob Boone    -Bobby Grich
##        202.2525        202.2525      1062.9419        202.2525        505.7619
```

Using caret

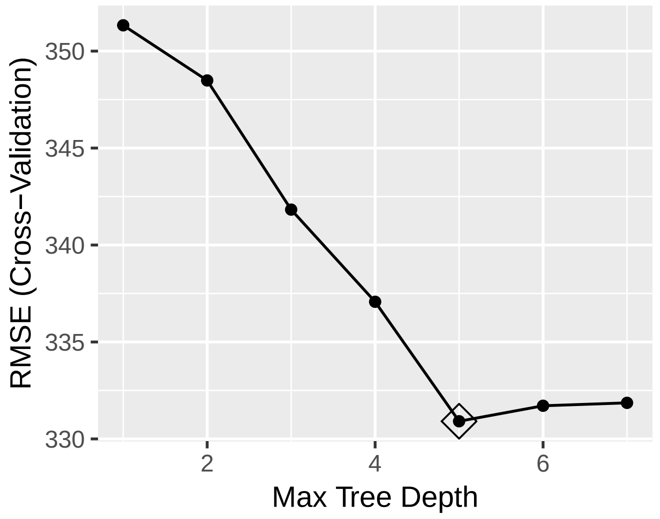
There are two options for CART: tuning over `cp` and tuning over `maxdepth`.

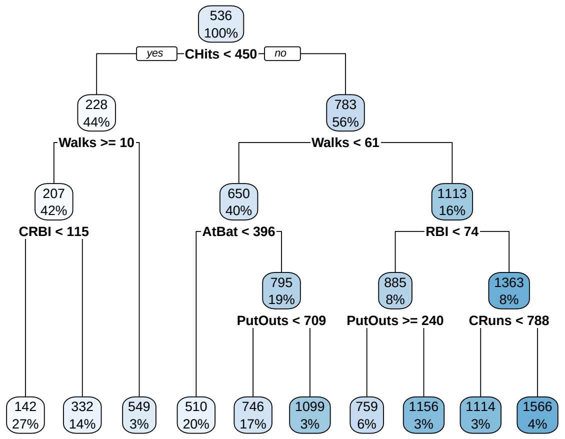
```
ctrl <- trainControl(method = "cv") # Also good to do repeated cv. Can be slow

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)
```





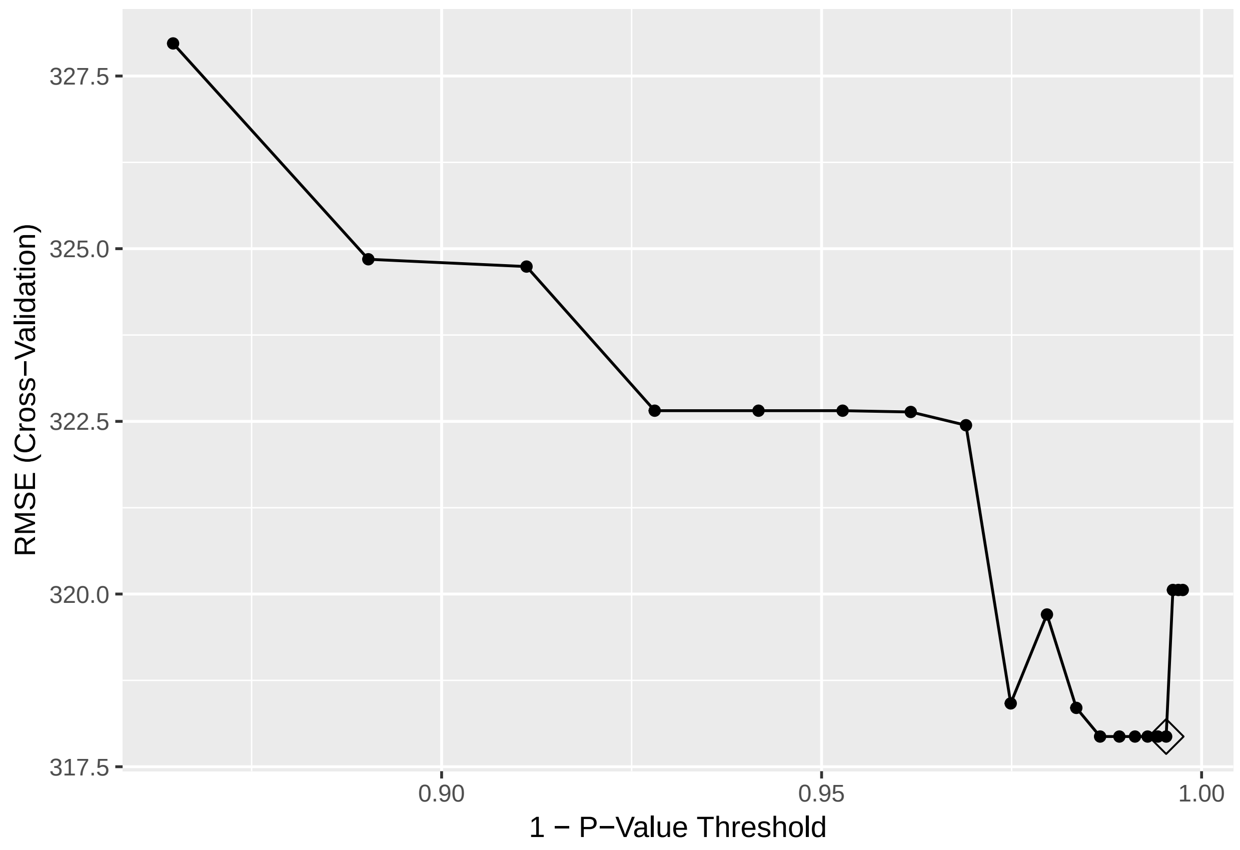
```r
set.seed(1)
# tune over maximum depth, method = "rpart2"
rpart2.fit <- train(Salary~., Hitters,
                   method = "rpart2",
                   tuneGrid = data.frame(maxdepth = 1:7),
                   trControl = ctrl)
ggplot(rpart2.fit, highlight = TRUE)
rpart.plot(rpart2.fit$finalModel)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



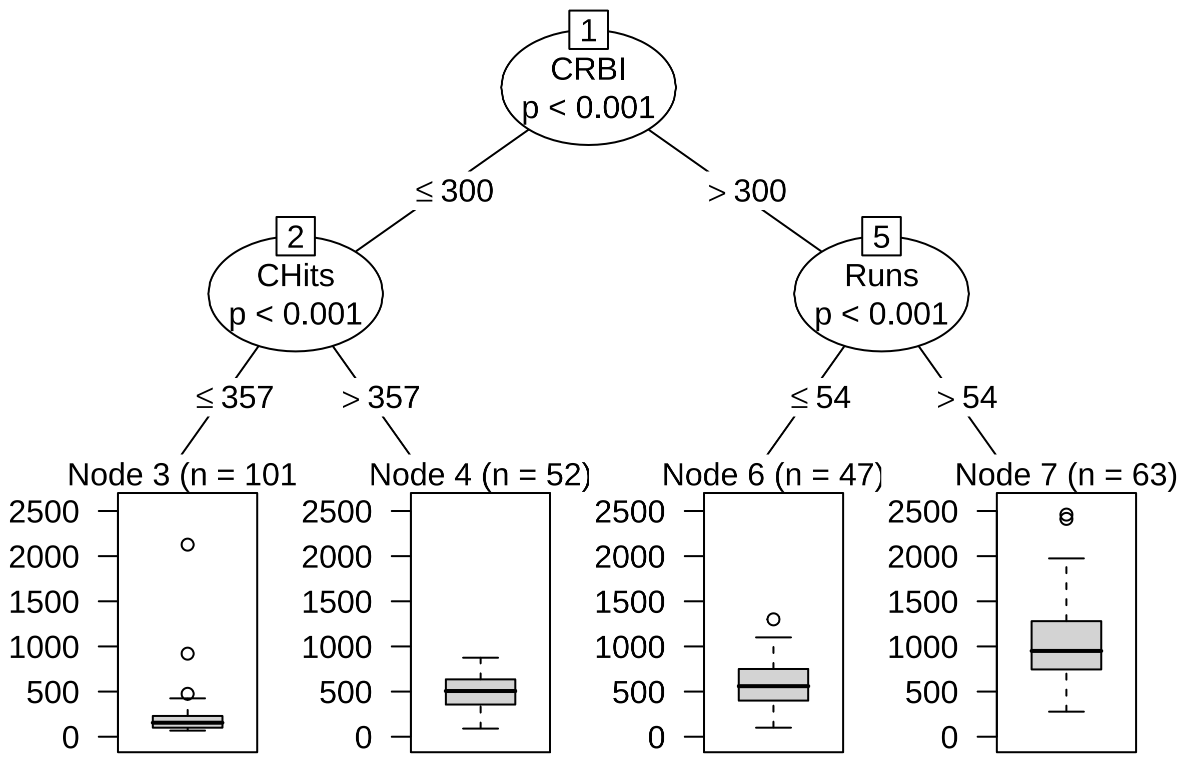


We can also fit a conditional inference tree model. The tuning parameter is `mincriterion`. Here the p value is the tuning parameter.

```r
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.

```r
set.seed(1)
bagging <- randomForest(Salary~., Hitters,
                   mtry = 19) # We have 19 predictors. when it is equal to the number of predictors. Th

set.seed(1)
rf <- randomForest(Salary~., Hitters,
                   mtry = 6) # Rule of thumb for m is p/3

# 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
```

```r
predict(rf2, data = Hitters2[1:5,])$predictions # in ranger there's a subobject called predictions
```

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

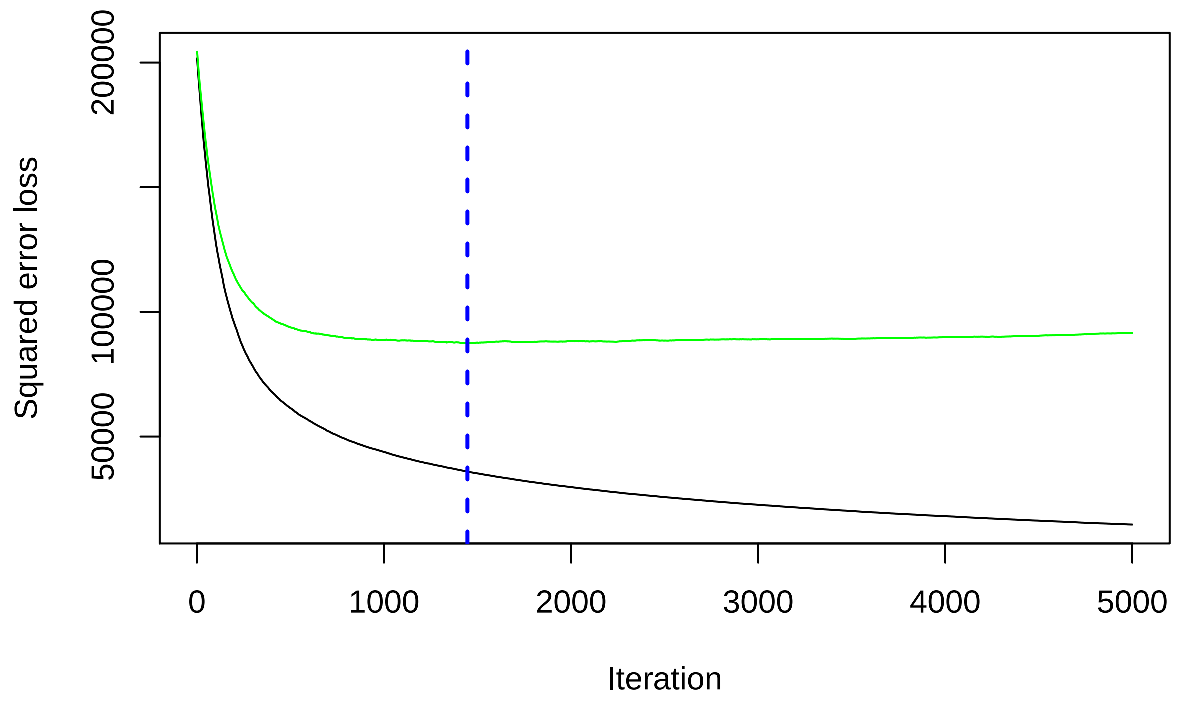
Boosting

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.

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



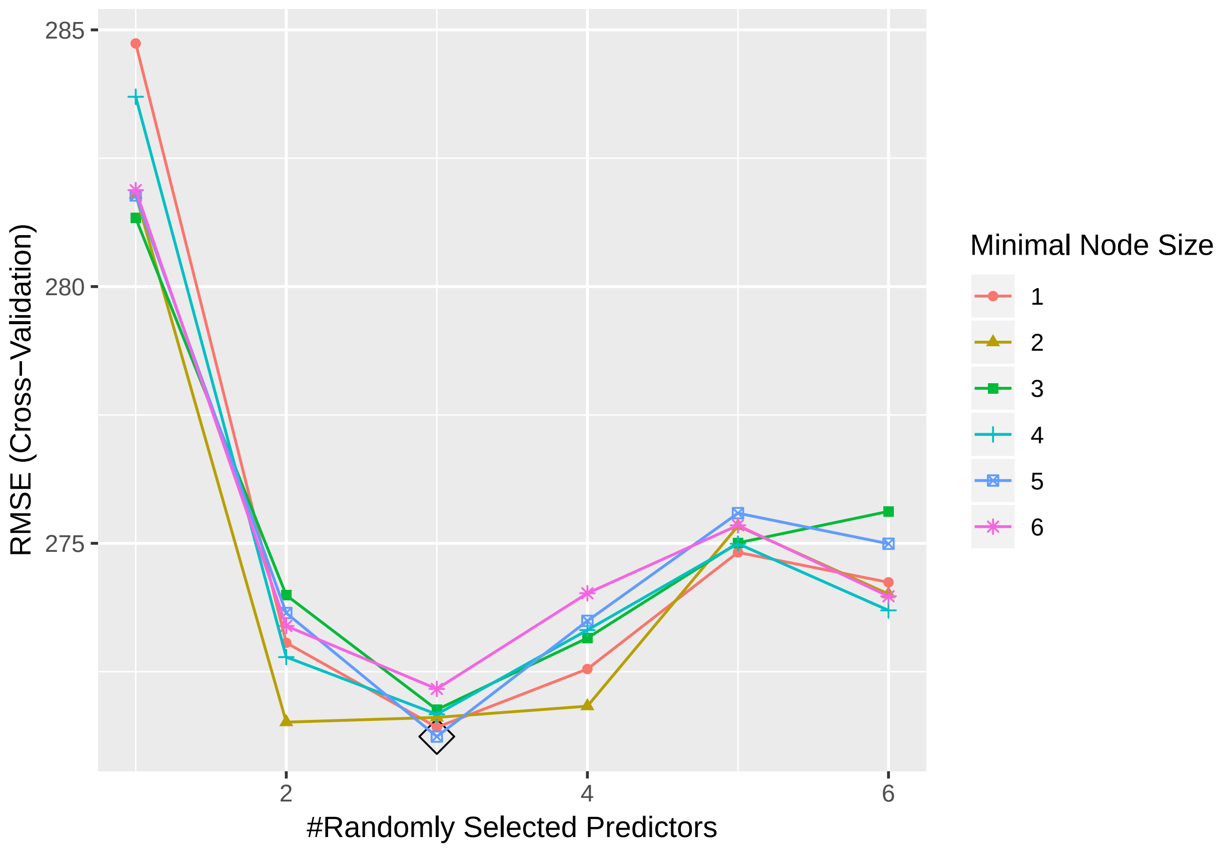
```
nt
```

```
## [1] 1446
```

Grid search using caret

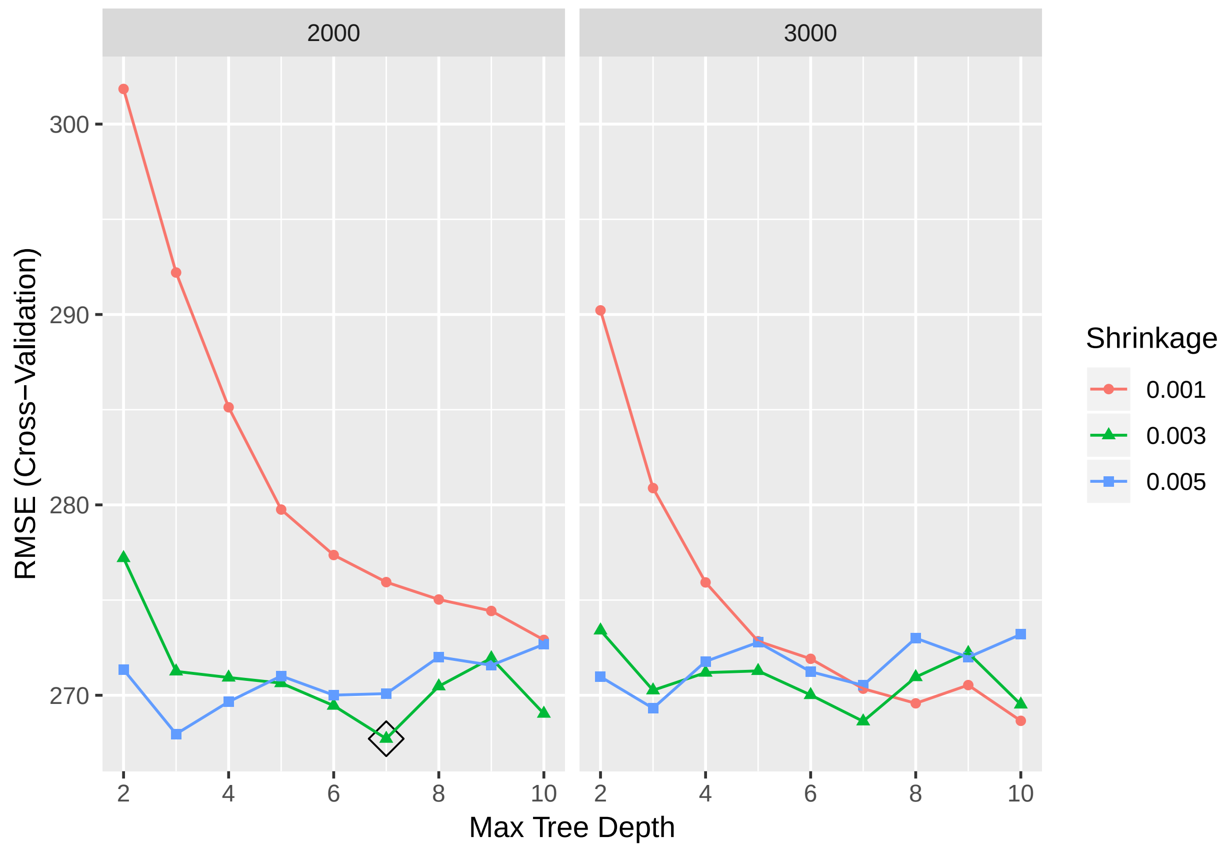
We use the fast implementation of random forest when tuning the model.

```
# 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)
```

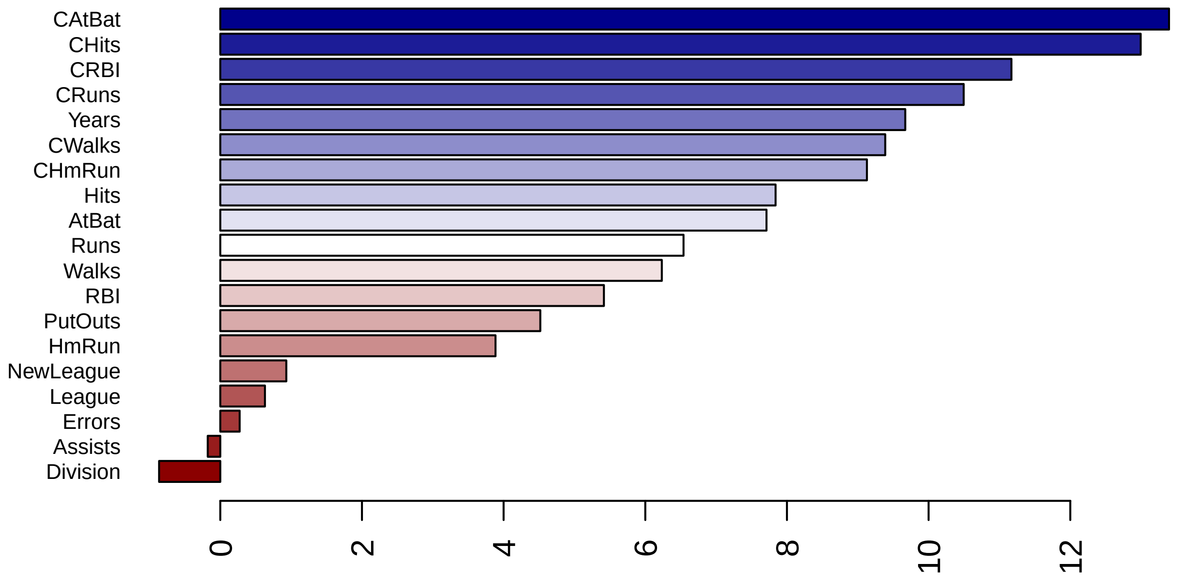
```
##
## Call:
## summary.resamples(object = resamp)
##
## Models: rf, gbm
## Number of resamples: 10
##
## MAE
##         Min.  1st Qu.   Median     Mean  3rd Qu.     Max. NA's
## rf  114.6328 138.1973 173.0966 168.6524 194.1539 226.4704    0
## gbm 122.3465 138.8980 170.6656 170.2506 201.6230 217.6166    0
##
## RMSE
##         Min.  1st Qu.   Median     Mean  3rd Qu.     Max. NA's
## rf  196.4885 211.9055 256.5444 271.2365 306.2881 396.8066    0
## gbm 196.2234 218.0976 252.6263 267.7173 318.6250 380.6951    0
##
## Rsquared
##           Min.   1st Qu.    Median      Mean   3rd Qu.      Max. NA's
## rf  0.4869971 0.5697425 0.6459258 0.6478927 0.7266503 0.8006483    0
## gbm 0.4931606 0.5401446 0.6206455 0.6552016 0.7743091 0.8510196    0
```

Explain the black-box models

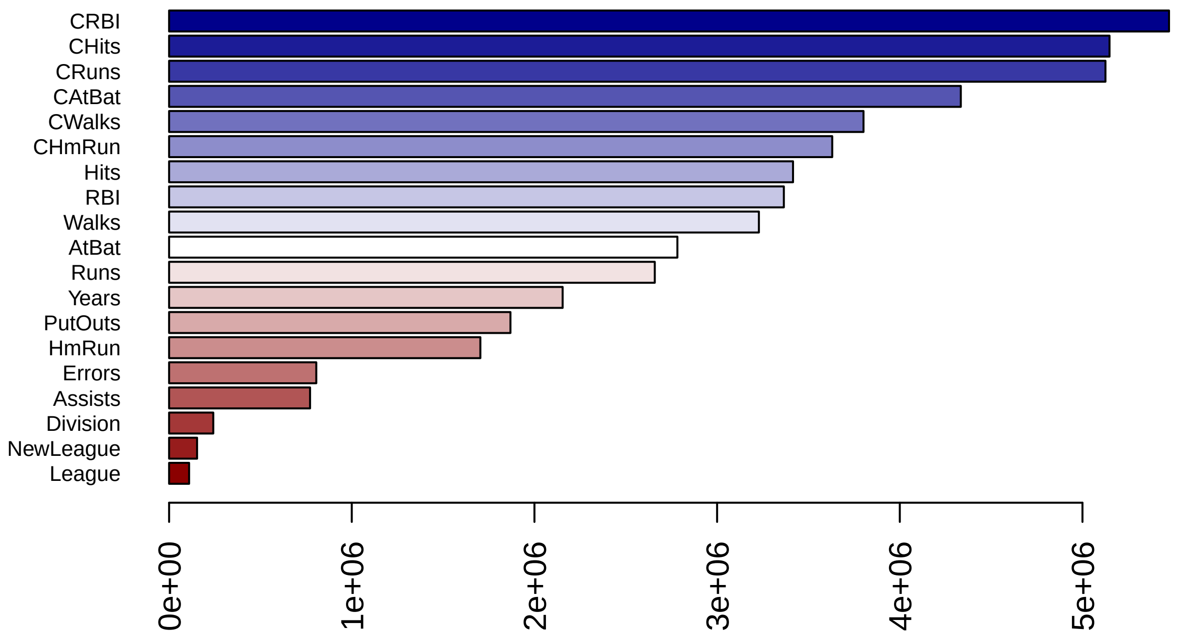
Variable importance

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))
```

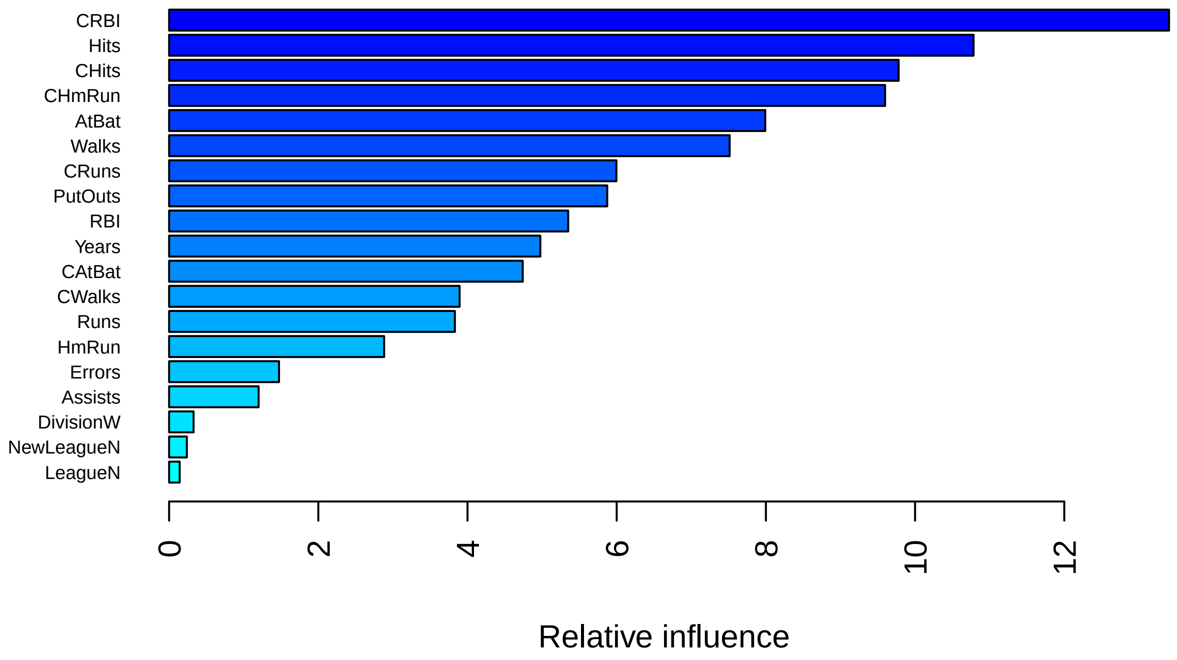


```
set.seed(1)
rf2.final.imp <- ranger(Salary~., Hitters,
                        mtry = 3, splitrule = "variance",
                        min.node.size = 5,
                        importance = "impurity")

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)
```



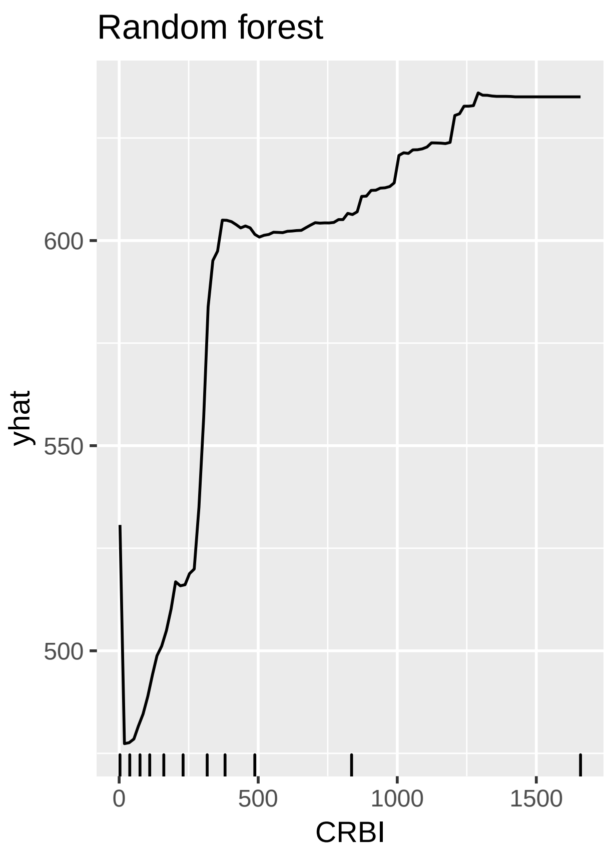
```
##                 var    rel.inf
## CRBI           CRBI 13.4045856
## Hits           Hits 10.7840042
## CHits         CHits  9.7799023
## CHmRun       CHmRun  9.5983338
## AtBat         AtBat  7.9918577
## Walks         Walks  7.5147601
## CRuns         CRuns  5.9964133
## PutOuts     PutOuts  5.8731574
## RBI                 RBI  5.3499249
## Years             Years  4.9765388
## CAtBat           CAtBat  4.7405095
## CWalks           CWalks  3.8945628
## Runs               Runs  3.8311327
## HmRun             HmRun  2.8823520
## Errors           Errors  1.4736805
## Assists         Assists  1.2001552
## DivisionW     DivisionW  0.3282774
## NewLeagueN NewLeagueN  0.2380353
## LeagueN         LeagueN  0.1418165
```

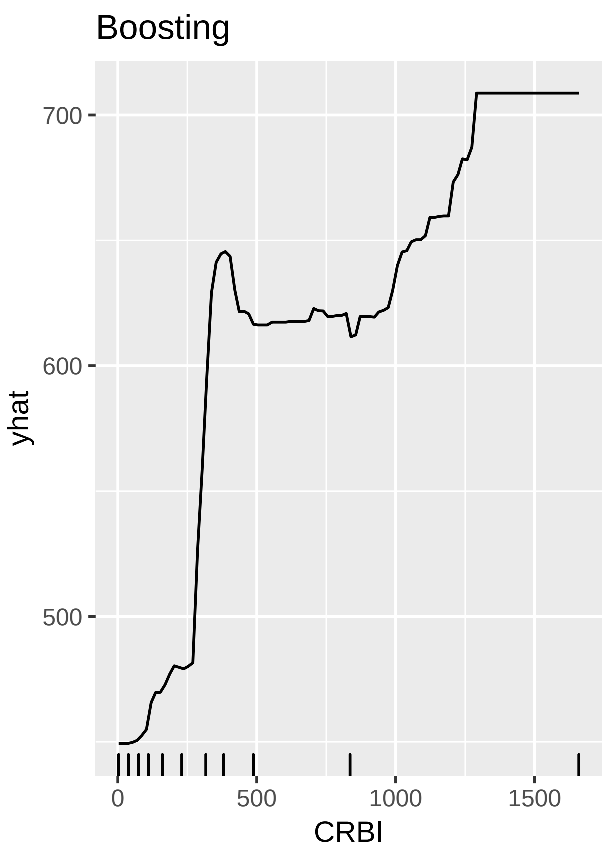
Partial dependence plots and individual conditional expectation curves

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.

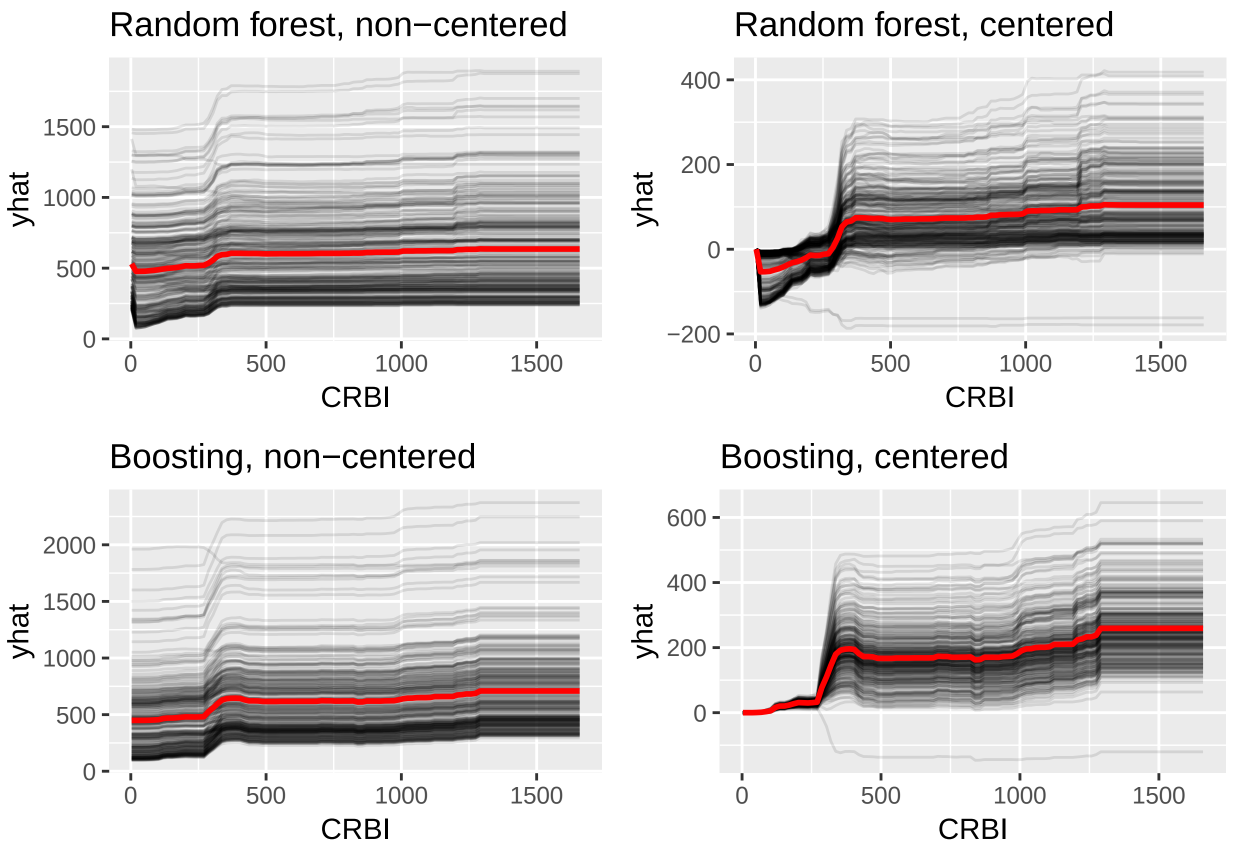
```r
pdp.rf <- rf.fit %>%
  partial(pred.var = "CRBI",
          grid.resolution = 100) %>%
  autoplot(rug = TRUE, train = Hitters) +
  ggtitle("Random forest")

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.

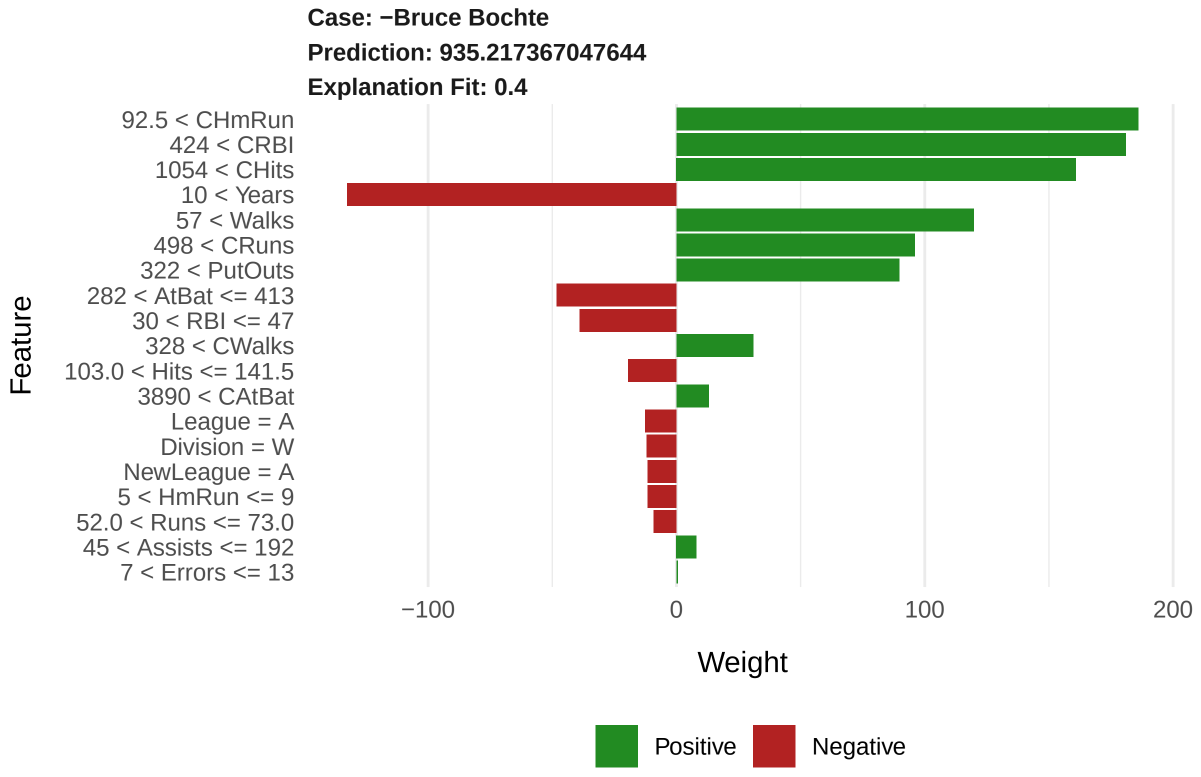
```r
ice1.rf <- rf.fit %>%
  partial(pred.var = "CRBI",
          grid.resolution = 100,
          ice = TRUE) %>%
  autoplot(train = Hitters, alpha = .1) +
  ggtitle("Random forest, non-centered")

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)
```



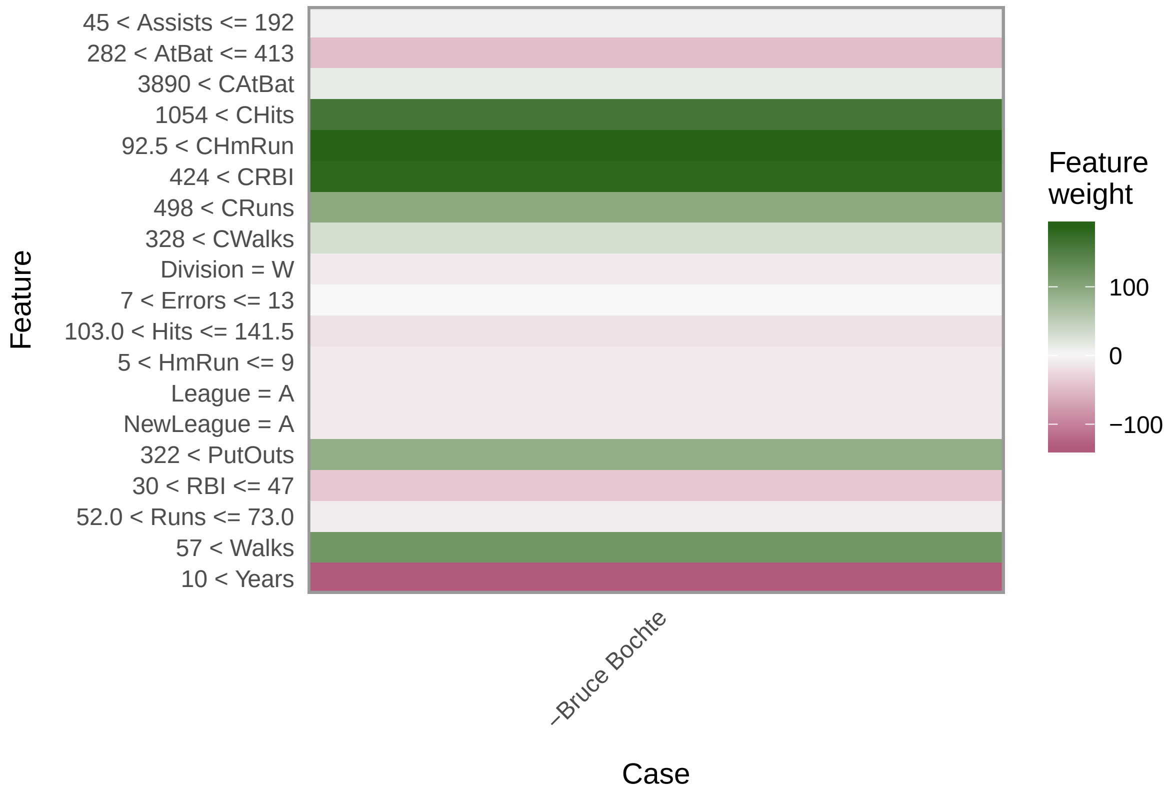
Plot the features in an explanation

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)
```

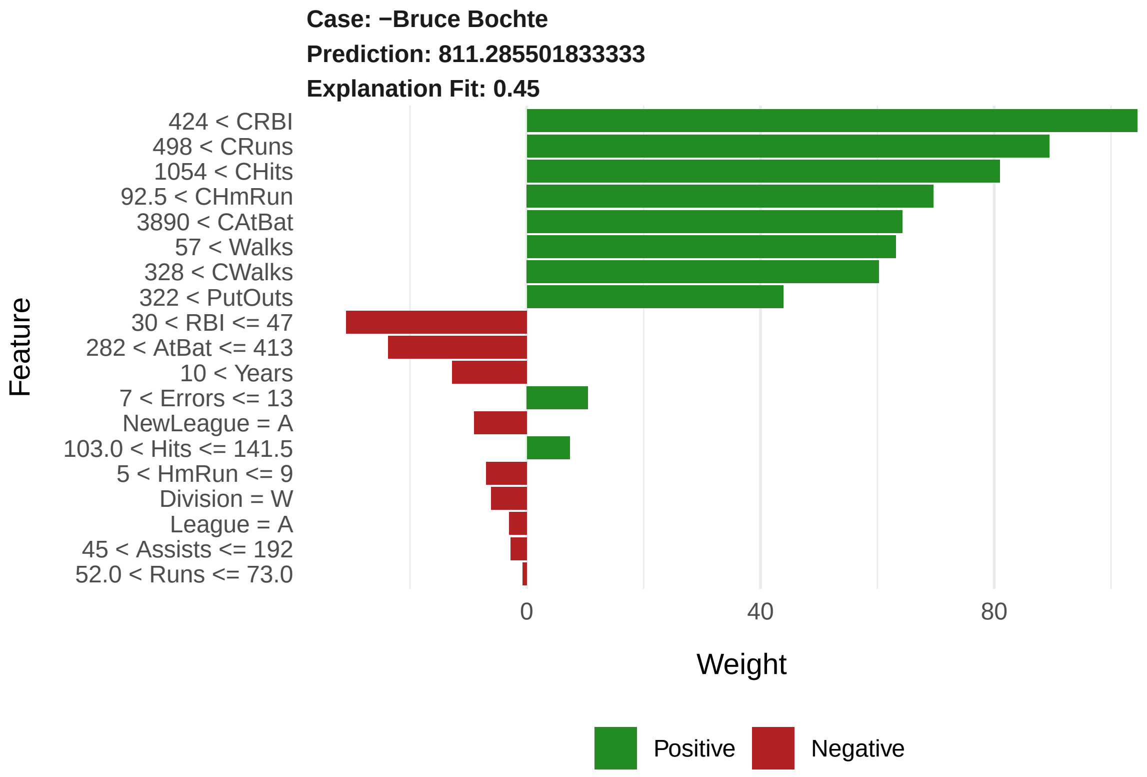


```
plot_explanations(explanation.gbm)
```



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

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


Case: −Bruce Bochte
Prediction: 811.285501833333
Explanation Fit: 0.45
Feature
424 < CRBI
498 < CRuns
1054 < CHits
92.5 < CHmRun
3890 < CAtBat
57 < Walks
328 < CWalks
322 < PutOuts
30 < RBI <= 47
282 < AtBat <= 413
10 < Years
7 < Errors <= 13
NewLeague = A
103.0 < Hits <= 141.5
5 < HmRun <= 9
Division = W
League = A
45 < Assists <= 192
52.0 < Runs <= 73.0
Weight
Positive
Negative