Random Forests

Code

Ruoqing Zhu

Last Updated: October 20, 2023

Random Forests

Roughly speaking, random forests (Breiman, 2001) are parallelly fitted many CART models with some randomness. There are several main components:

- Bootstrapping of data for each tree using the Bagging idea (Breiman, 1996), and use the averaged result (for regression) or majority voting (for classification) of all trees as the prediction.
- At each internal node, we may not consider all variables. Instead, we consider a randomly selected `mtry` variables to search for the best split. This idea was inspired by by Ho (1998).
- For each tree, we will not perform pruning. Instead, we simply stop when the internal node contains no more than `nodesize` number of observations.

Bagging Predictors

CART models may be difficult when dealing with non-axis-aligned decision boundaries. This can be seen from the example below, in a two-dimensional case. The idea of Bagging is that we can fit many CART models, each from a Bootstrap sample, i.e., sample with replacement from the original n observations. The reason that Breiman considered bootstrap samples is because it can approximate the original distribution that generates the data. But the end result is that since each tree may be slightly different from each other, when we stack them, the decision bound can be more "smooth".

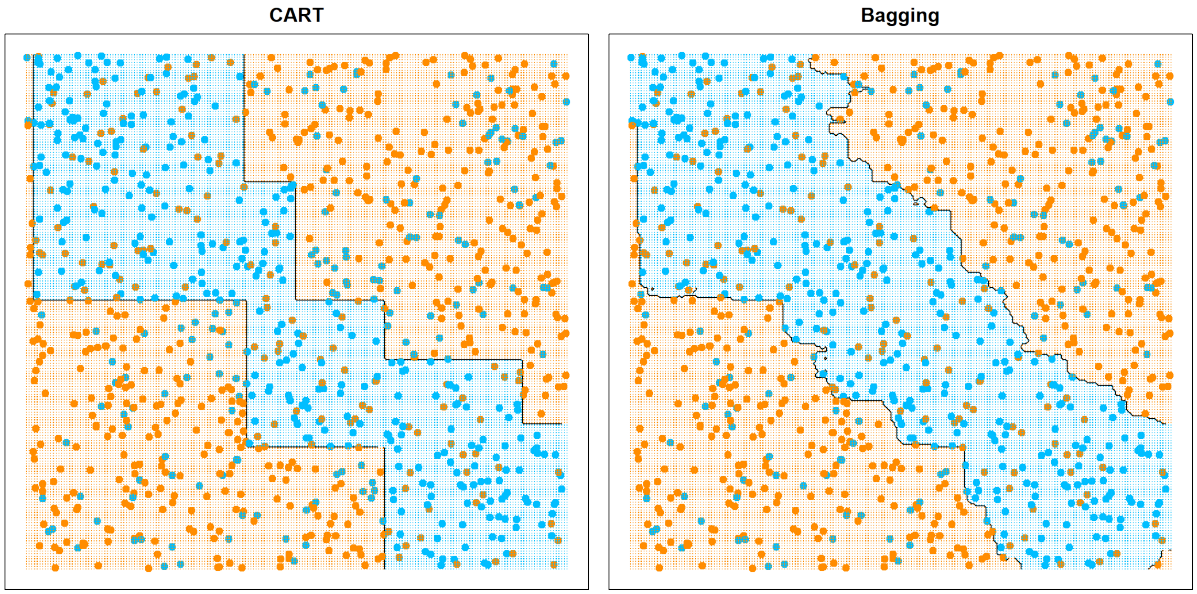
Hide

```
  # generate some data
  set.seed(2)
  n = 1000
  x1 = runif(n, -1, 1)
  x2 = runif(n, -1, 1)
  y = rbinom(n, size = 1, prob = ifelse((x1 + x2 > -0.5) & (x1 + x2 < 0.5) , 0.8,
0.2))
  xgrid = expand.grid(x1 = seq(-1, 1, 0.01), x2 = seq(-1, 1, 0.01))
```

Let's compare the decision rule of CART and Bagging. For CART, the decision line has to be aligned to axis. For Bagging, we use a total of 200 trees, specified by `nbagg` in the `ipred` package.

Hide

```
  # fit CART
  library(rpart)
  rpart.fit = rpart(as.factor(y)~x1+x2, data = data.frame(x1, x2, y))

  # we could fit a different tree using a bootstrap sample
  # rpart.fit = rpart(as.factor(y)~x1+x2, data = data.frame(x1, x2, y)[sample(1:n,
n, replace = TRUE), ])

  pred = matrix(predict(rpart.fit, xgrid, type = "class") == 1, 201, 201)
  contour(seq(-1, 1, 0.01), seq(-1, 1, 0.01), pred, levels=0.5, labels="",axes=FALS
E)
  points(x1, x2, col = ifelse(y == 1, "deepskyblue", "darkorange"), pch = 19, yaxt
="n", xaxt = "n")
  points(xgrid, pch=".", cex=1.2, col=ifelse(pred, "deepskyblue", "darkorange"))
  box()
  title("CART")

  # fit Bagging
  library(ipred)
  bag.fit = bagging(as.factor(y)~x1+x2, data = data.frame(x1, x2, y), nbagg = 200,
ns = 400)
  pred = matrix(predict(prune(bag.fit), xgrid) == 1, 201, 201)
  contour(seq(-1, 1, 0.01), seq(-1, 1, 0.01), pred, levels=0.5, labels="",axes=FALS
E)
  points(x1, x2, col = ifelse(y == 1, "deepskyblue", "darkorange"), pch = 19, yaxt
="n", xaxt = "n")
  points(xgrid, pch=".", cex=1.2, col=ifelse(pred, "deepskyblue", "darkorange"))
  box()
  title("Bagging")
```



Random Forests

Random forests are equipped with this Bootstrapping strategy, but also with other things, which are mentioned previously. They are controlled by several key parameters:

- `ntree` : number of trees
- `sampsize` : how many samples to use when fitting each tree
- `mtry` : number of randomly sampled variable to consider at each internal node
- `nodesize` : stop splitting when the node sample size is no larger than `nodesize`

Using the `randomForest` package, we can fit the model. It is difficult to visualize this when `p > 2`. But we can look at the testing error.

Hide

```
  # generate some data with larger p
  set.seed(2)
  n = 1000
  p = 10
  X = matrix(runif(n*p, -1, 1), n, p)
  x1 = X[, 1]
  x2 = X[, 2]
  y = rbinom(n, size = 1, prob = ifelse((x1 + x2 > -0.5) & (x1 + x2 < 0.5), 0.8, 0.
2))
  xgrid = expand.grid(x1 = seq(-1, 1, 0.01), x2 = seq(-1, 1, 0.01))

  # fit random forests with a selected tuning
  library(randomForest)
## randomForest 4.7-1.1
## Type rfNews() to see new features/changes/bug fixes.
  rf.fit = randomForest(X, as.factor(y), ntree = 1000,
                        mtry = 7, nodesize = 10, sampsize = 800)
```

Instead of generating a set of testing samples labels, let's directly compare with the "true" decision rule, the Bayes rule.

Hide

```
  # the testing data
  Xtest = matrix(runif(n*p, -1, 1), n, p)

  # the Bayes rule
  BayesRule = ifelse((Xtest[, 1] + Xtest[, 2] > -0.5) &
                     (Xtest[, 1] + Xtest[, 2] < 0.5), 1, 0)

  mean( (predict(rf.fit, Xtest) == "1") == BayesRule )
## [1] 0.785
```

Effect of `mtry`

In the two dimensional setting, we probably won't see much difference by using random forests, since the only effective change is `mtry = 1`, which is not really different than `mtry = 2` (the CART choice). You can try this by yourself.

However, the difference would be significant in higher dimensional settings, in our case $p = 10$. This is again an issue of bias-variance trade-off. The intuition is that, when we use a small `mtry`, and when p is large, we may by chance randomly select some irrelevant variables that has nothing to do with the

outcome. Then this particular split would be wasted. Missing the true variable may cause larger bias. On the other hand, when we use a large `mtry`, we will be greedy for signals since we compare many different variables and pick the best one. But this is also as the risk of over-fitting. Hence, tuning is necessary.

Just as an example, let's try a small `mtry`:

Hide

```
  rf.fit = randomForest(X, as.factor(y), ntree = 1000,
                        mtry = 1, nodesize = 10, sampsize = 800)

  mean( (predict(rf.fit, Xtest) == "1") == BayesRule )
## [1] 0.634
```

Effect of `nodesize`

When we use a small `nodesize`, we are at the risk of over-fitting. This is similar to the 1NN example. When we use large `nodesize`, there could be under-fitting. We will further understand this through the kernel view.

Categorical Predictors

Random forests can directly handle categorical predictors without coding them into dummy variables, thanks to its mechanics of splitting. For example, if we have a variable with K categories, then we can generate some candidate split by randomly putting some categories to the left node and some to the right node. We can then exhaust all such constructions and pick the one with the best reduction of impurity.

Variable Importance

Random forests model provides a way to evaluate the importance of each variable. This can be done by specifying the `importance` argument. We usually use the `MeanDecreaseAccuracy` or `MeanDecreaseGini` column as the summary of the importance of each variable.

Hide

```
  rf.fit = randomForest(X, as.factor(y), ntree = 1000,
                        mtry = 7, nodesize = 10, sampsize = 800,
                        importance=TRUE)

  importance(rf.fit)
##             0         1 MeanDecreaseAccuracy MeanDecreaseGini
## 1  39.1053057 39.823786           45.4232897         47.79065
## 2  38.1820764 40.119387           45.1964485         54.89580
## 3   3.2719270  1.461298            3.2274895         28.44828
## 4  -0.2777943 -6.287430           -4.8470758         22.09006
## 5   2.0937973  1.654224            2.5256400         28.57575
## 6   2.2354984 -2.435663           -0.1297796         25.29836
## 7   0.2083020  2.724449            2.0679184         24.28751
## 8   0.2018946  3.350897            2.3962745         25.51630
## 9  -1.6159803  2.150674            0.3912234         23.41498
## 10  2.6081961  4.417256            4.8004480         27.80399
```

Kernel view of Random Forests

For this part, we need to install a new package `RLT`. This is a package in development. But you can install the current version from GitHub (ver ≥ 4.2.6). Please note that the current CRAN version (ver. 3.2.5) does not work for this part. Use the following code to install the package.

Hide

```
  # install.packages("devtools")
  devtools::install_github("teazrq/RLT")
```

If you are using MacOS, then you need to follow this guild (https://teazrq.github.io/random-forests-tutorial/rlab/basics/packages.html) to install the package.

Kernel Function Induced from Random Forest

Similar as a tree model, random forest can also be viewed as kernel estimator. Essentially its a stacking of kernels induced from all trees. The idea has been illustrated in @scornet2016random. However, since random forest is a random algorithm, each tree can be slightly different from each other. To incorporate this, denote the randomness of a tree estimator by η, which can affect how the tree is constructed. Suppose we fit B trees in a random forest, with each tree denoted as $\square_b$, then a random forest estimator can be expressed as

$$\hat{f}(x) = \frac{1}{B}\sum_{b=1}^{B} \frac{\sum_i K_{\square_b}(x, x_i; \eta_b) y_i}{\sum_i K_{\square_b}(x, x_i; \eta_b)}.$$

Note that in this expression, the denominators in each tree estimator are different. However, by the design of the algorithm, each terminal nodes are roughly the same size. Hence, we could also consider an alternative kernel induced from random forest, which aligns with traditional kernel estimators.

$$\hat{f}(x) = \frac{\sum_i y_i \sum_{b=1}^{B} K_{\square_b}(x, x_i)}{\sum_i \sum_{b=1}^{B} K_{\square_b}(x, x_i)}.$$

In this case, the kernel function

$$K_{\text{RF}}(x, x_i) = \sum_{b=1}^{B} K_{\square_b}(x, x_i),$$

which counts how many times x falls into the same terminal node as observation x_i. Hence, this kernel representation can be incorporated into many machine learning algorithms. For example, if we are interested in a supervised clustering setting, we can first fit a random forest model and perform spectral clustering using the induced kernel matrix. We can also use the kernel ridge regression with the kernel induced. Let's generate a new dataset with 5 continuous variables. The true model depends on just the first two variables.

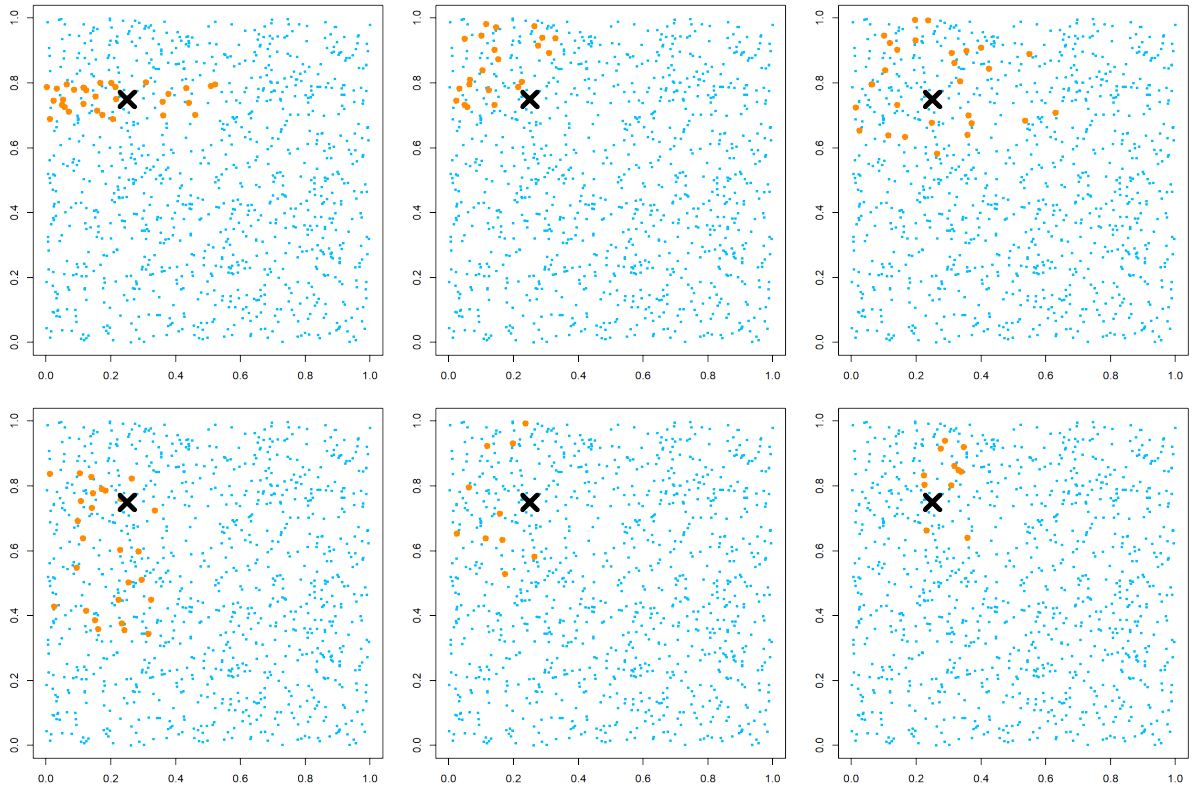
Hide

```
  # generate data
  n = 1000; p = 5
  X = matrix(runif(n*p), n, p)
  y = X[, 1] + X[, 2] + rnorm(n)
```

If we fit just one tree, there could be different variations based on the randomness.

Hide

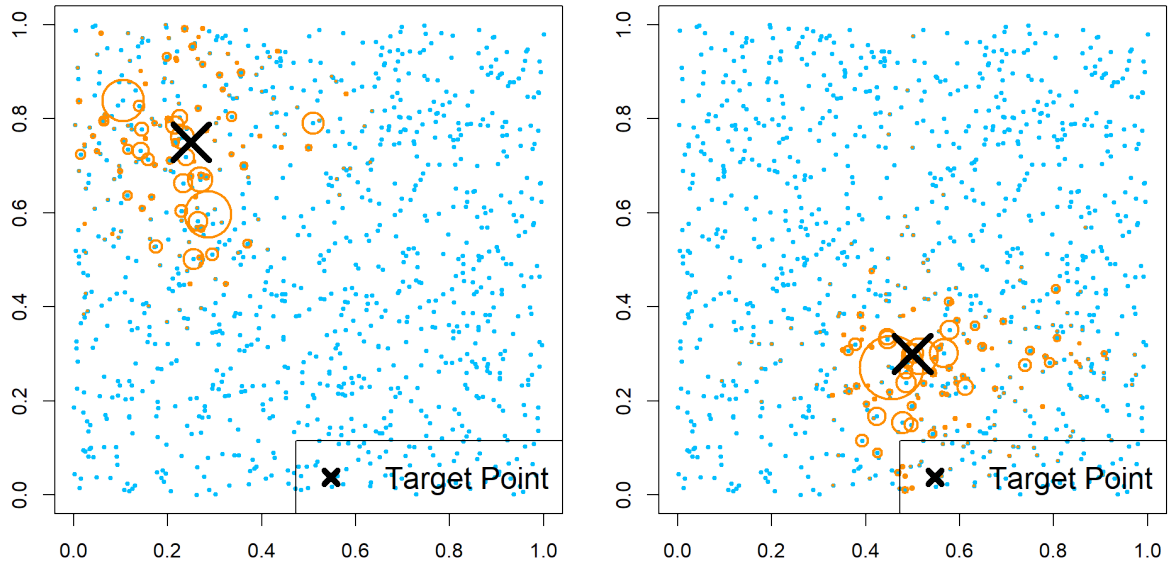
```
  library(RLT)
## RLT and Random Forests v4.2.6
## pre-release at github.com/teazrq/RLT
  par(mfrow=c(2, 3))

  for (i in 1:6)
  {

    # fit a model with one tree
    RLTfit <- RLT(X, y, ntrees = 1, nmin = 30, mtry = 5,
                  split.gen = "best", resample.prob = 0.9,
                  resample.replace = FALSE,
                  param.control = list("resample.track" = TRUE))

    # target point 1
    newX = matrix(c(0.25, 0.75, 0.5, 0.5, 0.5),
                  1, 5)

    KernelW = forest.kernel(RLTfit, X1 = newX, X2 = X, vs.train = TRUE)$Kernel

    par(mar = c(2, 2, 2, 2))
    plot(X[, 1], X[, 2], col = "deepskyblue", pch = 19, cex = 0.5)
    points(X[, 1], X[, 2], col = "darkorange", pch = 19, cex = KernelW>0, lwd = 2)
    points(newX[1], newX[2], col = "black", pch = 4, cex = 3, lwd = 5)
  }
```



If we stack all of them, we obtain a random forest kernel.

Hide

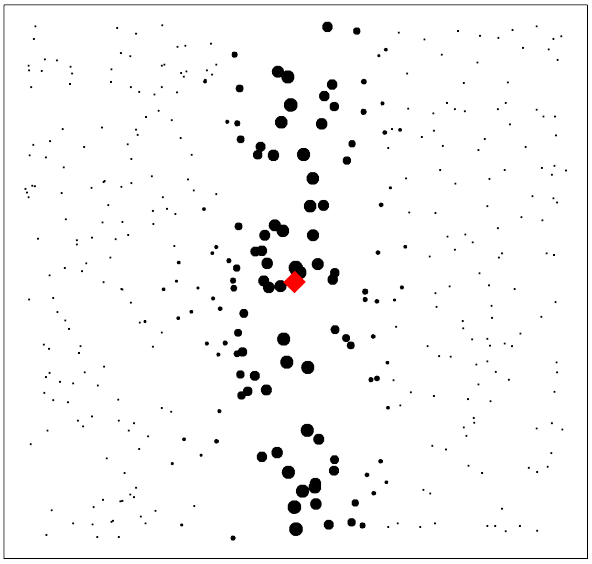
```
RLTfit <- RLT(X, y, ntrees = 500, nmin = 4, mtry = 5,
              split.gen = "best", resample.prob = 0.8,
              resample.replace = FALSE,
              param.control = list("resample.track" = TRUE))

par(mfrow=c(1, 2))

# target point 1
newX = matrix(c(0.25, 0.75, 0.5, 0.5, 0.5),
              1, 5)

KernelW = forest.kernel(RLTfit, X1 = newX, X2 = X, vs.train = TRUE)$Kernel
w = KernelW / 500

par(mar = c(2, 2, 2, 2))
plot(X[, 1], X[, 2], col = "deepskyblue", pch = 19, cex = 0.5)
points(X[, 1], X[, 2], col = "darkorange", cex = w*30, lwd = 2)
points(newX[1], newX[2], col = "black", pch = 4, cex = 4, lwd = 5)
legend("bottomright", "Target Point", pch = 4, col = "black",
       lwd = 5, lty = NA, cex = 1.5)


# target point 2
newX = matrix(c(0.5, 0.3, 0.5, 0.5, 0.5),
              1, 5)

KernelW = forest.kernel(RLTfit, X1 = newX, X2 = X, vs.train = TRUE)$Kernel
w = KernelW / 500

par(mar = c(2, 2, 2, 2))
plot(X[, 1], X[, 2], col = "deepskyblue", pch = 19, cex = 0.5)
points(X[, 1], X[, 2], col = "darkorange", cex = w*30, lwd = 2)
points(newX[1], newX[2], col = "black", pch = 4, cex = 4, lwd = 5)
legend("bottomright", "Target Point", pch = 4, col = "black",
       lwd = 5, lty = NA, cex = 1.5)
```



Adaptiveness of Random Forest Kernel

However, random forest can adapt pretty well in a high-dimensional, especially sparse setting. This is because of the greedy splitting rule selection. The adaptiveness works in a way that, it tends to ignore covariates that are not effective on explaining the variation of Y. Hence, making the model similar to a kernel method on a low-dimensional space. The following example illustrate this effect in a two-dimensional case. We can see that the outcome is only related to the first dimension. Hence, when setting `mtry = 2`, we will almost always perfer to split on the first variable, making its neighbors very close to the target prediction point $(0, 0)^T$.

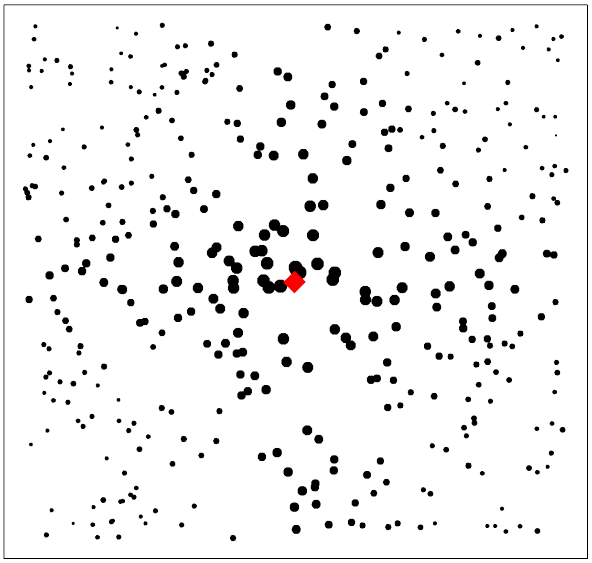
Show



The tuning parameter `mtry` has a very strong effect on controlling this greediness. When `mtry` is large, we will be very greedy on selecting the true signal variable to split. In a high-dimensional setting, we may only use a few variables before reaching a terminal node, making the model only rely a few dimensions. When we use a very small `mtry`, the model behaves similarly to a regular kernel estimator with good

smoothing (small variance) property. However, since it is effectively randomly selecting a dimension to split, the bandwidth on each dimension would also be similar but large since we can only afford a few splits before the node size becomes too small. This can be seen from the following example, with `mtry = 1`.

Show



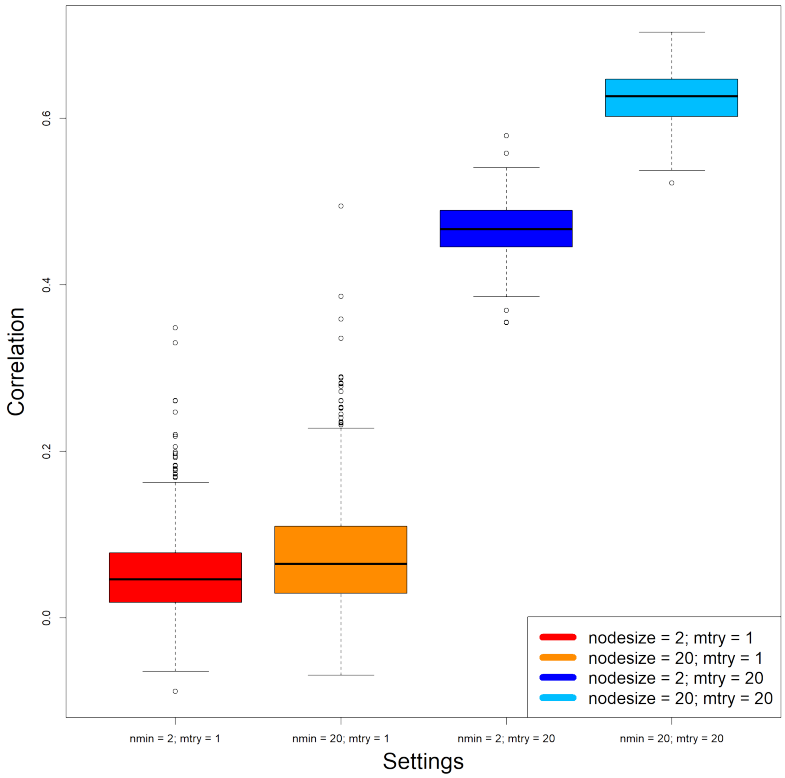
Correlation of Tree Predictions

`nodesize` is not the only tuning parameter that would affect the bias-variance trade-off. `mtry` can determine how greedy the search algorithm is. This serves two purposes: make each individual tree more accurate, but also this makes trees highly correlated, hence the variance of the average model would be relatively high. We generate 500 pairs of trees, and calculate the correlation of predictions on 1000 testing samples. The correlation result is very different using different tuning parameters.

Hide

```
  par(mfrow=c(1, 1))
  n = 1000
  p = 20

  X = matrix(rnorm(n*p), n, p)
  y = as.vector(X[, 1:3] %*% c(0.2, 0.5, 1)) + rnorm(n, 0.5)

  # fit 4 models
  RLT.fit1 <- RLT(X, y, ntree = 1000, mtry = 1, nmin = 3)
  RLT.fit2 <- RLT(X, y, ntree = 1000, mtry = 1, nmin = 20)
  RLT.fit3 <- RLT(X, y, ntree = 1000, mtry = 20, nmin = 3)
  RLT.fit4 <- RLT(X, y, ntree = 1000, mtry = 20, nmin = 20, importance = TRUE)

  # calculate predictions
  XNew = matrix(rnorm(n*p), n, p)
  yNew = as.vector(XNew[, 1:3] %*% c(0.2, 0.5, 1)) + rnorm(n, 0.5)

  pred1 = predict(RLT.fit1, XNew, keep.all = TRUE)
  mean((pred1$Prediction - yNew)^2)
## [1] 1.757041

  pred2 = predict(RLT.fit2, XNew, keep.all = TRUE)
  mean((pred2$Prediction - yNew)^2)
## [1] 1.822809

  pred3 = predict(RLT.fit3, XNew, keep.all = TRUE)
  mean((pred3$Prediction - yNew)^2)
## [1] 1.05167

  pred4 = predict(RLT.fit4, XNew, keep.all = TRUE)
  mean((pred4$Prediction - yNew)^2)
## [1] 1.039693

  # correlation among pairs of trees
  allcor = matrix(NA, 500, 4)

  for (i in 1:500)
  {
    allcor[i, 1] = cor(pred1$PredictionAll[, i], pred1$PredictionAll[, 500+i])
    allcor[i, 2] = cor(pred2$PredictionAll[, i], pred2$PredictionAll[, 500+i])
    allcor[i, 3] = cor(pred3$PredictionAll[, i], pred3$PredictionAll[, 500+i])
    allcor[i, 4] = cor(pred4$PredictionAll[, i], pred4$PredictionAll[, 500+i])
  }
## Warning in cor(pred1$PredictionAll[, i], pred1$PredictionAll[, 500 + i]): the st
andard deviation is
## zero

## Warning in cor(pred1$PredictionAll[, i], pred1$PredictionAll[, 500 + i]): the st
andard deviation is
## zero

## Warning in cor(pred1$PredictionAll[, i], pred1$PredictionAll[, 500 + i]): the st
```

```
andard deviation is
## zero

  colnames(allcor) = c("nmin = 2; mtry = 1",
                       "nmin = 20; mtry = 1",
                       "nmin = 2; mtry = 20",
                       "nmin = 20; mtry = 20")
  par(mar = c(5,5,2,2))
  boxplot(allcor, cex.lab = 2, col = c("red", "darkorange", "blue", "deepskyblue"),
          xlab = "Settings", ylab = "Correlation")
  legend("bottomright", c("nodesize = 2; mtry = 1",
                          "nodesize = 20; mtry = 1",
                          "nodesize = 2; mtry = 20",
                          "nodesize = 20; mtry = 20"),
         col = c("red", "darkorange", "blue", "deepskyblue"),
         lty = 1, lwd = 10, cex = 1.5)
```



Out-of-bag Error and Parameter Tuning

The Out-of-Bag (OOB) prediction error is a built-in cross-validation method in random forests. Each decision tree in a random forest is trained on a bootstrap sample, which is a randomly selected subset of the training data with replacement. The remaining data points, not used in constructing a particular tree, are called “out-of-bag” samples. The OOB prediction errors are often used to select tuning parameters. The procedure works as follows:

- Train All Trees: While training each tree, keep track of the OOB samples—those not included in the bootstrap sample for that particular tree.
- OOB Prediction: For each observation, collect all trees that do not include this sample as the training data. Treat this collection of trees as a (smaller size) forest and obtain the predicted value of this observation.
- OOB Error: For each observation, obtain the OOB prediction and average their errors.

The following code uses the `randomForest` package for selecting the best tuning parameter.

Hide

```
library(randomForest)
n = 1000
p = 20

X = matrix(rnorm(n*p), n, p)
y = as.vector(X[, 1:3] %*% c(0.2, 0.5, 1)) + rnorm(n, 0.5)

alltune = expand.grid("nodesize" = c(3, 10), "mtry" = c(5, 20))
allerror = rep(NA, nrow(alltune))

for (k in seq_along(allerror))
{
  rf.fit = randomForest(x = X, y = y, ntree = 1000,
                        nodesize = alltune$nodesize[k],
                        mtry = alltune$mtry[k] )
  # OOB prediction error
  allerror[k] = mean( (rf.fit$predicted - y)^2 )
}

cbind(alltune, allerror)
```

nodesize <dbl>	mtry <dbl>	allerror <dbl>
3	5	1.194541
10	5	1.188658
3	20	1.126162
10	20	1.127482

4 rows

Variable Importance

The idea of variable importance is to identify which features (or variables) are most influential in predicting the outcome based on the fitted model. Variable importance is typically assessed through techniques like mean decrease accuracy, which measures the decrease in model accuracy when a variable's values are permuted across the out-of-bag samples, thereby disrupting the relationship between that variable and the target. Alternatively, it can also be measured using the mean decrease impurity, which calculates the total reduction in the criterion (Gini impurity, entropy, or mean squared error) that each variable provides when used in trees, averaged over all trees in the forest. The calculation can be summarized by the following steps:

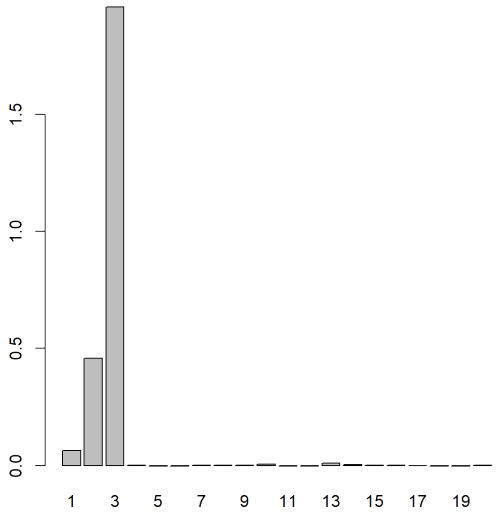
- Train the Random Forest: Fit a random forest model to your data using all available variables.
- Out-of-Bag Evaluation: For each tree in the forest, predict the outcome for the out-of-bag (OOB) samples—these are the samples not used in the construction of that particular tree. Compute the

OOB accuracy (or another relevant metric like AUC for classification, MSE for regression) for these predictions.

- Permute Variable & Re-evaluate: For each variable of interest, randomly permute its values among the OOB samples. Then, use the same tree to make predictions on these "shuffled" data and compute the accuracy (or other metrics) again.
- Calculate Decrease in Accuracy: Compare the accuracy obtained from the permuted data to the original OOB accuracy for each tree. The difference is a measure of the importance of the variable for that specific tree.
- Average Over All Trees: Aggregate these importance measures across all trees in the forest to get a single importance score for each variable.

Hide

```
    rf.fit = randomForest(x = X, y = y, ntree = 1000,
                      nodesize = 10, mtry = 20, importance = TRUE)

    # variable importance for %IncMSE (1st column)
    rf.fit$importance
##          %IncMSE IncNodePurity
## 1   6.544543e-02     100.50559
## 2   4.576953e-01     329.75841
## 3   1.957402e+00    1120.10653
## 4  -1.511316e-03      41.39130
## 5  -2.084699e-03      38.65617
## 6  -2.854162e-03      40.09969
## 7  -1.166074e-03      43.61276
## 8  -1.195933e-03      39.78524
## 9   7.045932e-04      41.92282
## 10  5.193246e-03      43.05239
## 11 -1.753814e-03      36.93066
## 12 -1.902518e-03      45.37310
## 13  1.005804e-02      46.67606
## 14  4.727713e-03      46.20895
## 15  1.512402e-03      42.02358
## 16 -1.182866e-03      39.84742
## 17 -9.769127e-05      36.61692
## 18 -2.467652e-03      36.72770
## 19 -2.122681e-03      42.97624
## 20  8.160158e-04      40.38117

    # plot
    barplot( rf.fit$importance[, 1] )
```


0.0
0.5
1.0
1.5
1
3
5
7
9
11
13
15
17
19