9 Random Forests

Random forests are a model averaging technique using trees as base models.

9.1 The process of building a random forest

1. Specify B, the number of bootstrap samples, and $m \leq p$, the number of features to consider at each split.
2. For $b = 1, \ldots, B$,
   - create a bootstrap sample of the training data;
   - build a very high depth tree $\hat{r}_b$ without pruning;
   - for every split, only m randomly selected features are considered.
3. The final RF model has prediction at x_0:
   - continuous outcome: $\hat{r}_{\text{rf}}(x_0) = \frac{1}{B}\sum_b \hat{r}_b(x_0)$
   - categorical outcome: majority vote from $\{\hat{r}_b(x_0) : b = 1, \cdots, B\}$.

When $m = p$ for every split, the method is called **bagging**.

Notes:

- m is a hyperparameter. Once B and m are specified, RF is an automatic procedure and no tuning is needed. The choice of m determines the complexity of the final model.
- B is not a tuning parameter because a large value of B will not lead to overfitting.
- The RF algorithm is non-deterministic due to the randomness in generating bootstrap samples and in selecting subsets of features.
- Majority vote is a type of average.
- Low correlation among the individual trees, because
  - randomness from one bootstrap sample to another;
  - a random subset of features is considered for every split;
  - the individual trees severely overfit data due to a high depth.
- Can be slow for very large n. In `randomForest()`, one can set `nodesize=` (minimum size of terminal nodes; default 1 for classification and 5 for regression) and `maxnodes=` (maximum number of terminal nodes).
- RFs are **adaptive nearest-neighbor estimators** with neighborhood-defining features selected adaptively. The value m implicitly determines the definition of neighborhood; the smaller m the larger neighborhood.
- Model averaging can be used with any base models besides trees.

Justification for model averaging: Let $\hat{a}_1, \ldots, \hat{a}_B$ be estimates of μ that have the same mean μ and variance σ^2. If their pairwise correlation is $cor(\hat{a}_i, \hat{a}_j) = \rho$ for any $i \neq j$, then their average $\frac{1}{B}\sum \hat{a}_i$ has mean μ and variance

$$\frac{\sigma^2}{B^2}[B + B(B-1)\rho] = \sigma^2\rho + \frac{1}{B}\sigma^2(1-\rho).$$

We can pick a large B to make $\frac{1}{B}\sigma^2(1-\rho)$ very small. If ρ is very small and σ^2 is not too high, then $\sigma^2\rho$ can be small.

Thus, we seek to define a base model (or classifier) such that:

(1) it is unbiased;
(2) it is fast to build (so that B can be large);
(3) it has very small correlation from one to another, and a variance not too large.

In R `randomForest` package, by default, m (`mtry`) is $m = \sqrt{p}$ for classification trees and $m = p/3$ for regression trees, B (`ntree`) is 500. The measure of impurity is MSE for regression trees and Gini index for classification trees. We use the `Heart` dataset as an example. We first prepare the data.

```
Heart = read.csv("Heart.csv", row.names=1)  ## first column is row name, not a variable
names(Heart); dim(Heart)  ## 303, 14

Heart$Thal = factor(Heart$Thal, c('normal', 'reversable', 'fixed'), ordered=T)
Heart$ChestPain = factor(Heart$ChestPain, levels(Heart$ChestPain), ordered=T)
```

```
apply(is.na(Heart), 2, sum)           ## check which variable has missing data
which(apply(is.na(Heart), 1, sum)>0)  ## check which observation has missing data
Heart2 = na.omit(Heart); dim(Heart2)  ## 297, 14
table(Heart2$AHD)   ## 160 No; 137 Yes
```

One can either create a "clean" version of data like what I just did above, or run `randomForest()` using `na.action=na.omit`. I use the former approach here.

```
library(randomForest)
rf1 = randomForest(AHD ~ ., data=Heart)  ## error due to missing data

set.seed(899); rf1 = randomForest(AHD ~ ., data=Heart2)
set.seed(899); rf2 = randomForest(AHD ~ ., data=Heart, na.action=na.omit)

all.equal(rf1$predicted, rf2$predicted, check.attributes=F)  ## True
rm(rf1, rf2)
```

Now we create training and test sets, and then fit a RF model using the training set.

```
set.seed(2018)
trainidx = sample(1:nrow(Heart2), 220)
Heart2.train = Heart2[trainidx, ]  ## training set
Heart2.test = Heart2[-trainidx, ]  ## test set

set.seed(2018)
rf = randomForest(AHD ~ ., data = Heart2.train)
rf                 ## print general results
names(rf)          ## all details are here
rf$mtry; rf$ntree  ## check what values were used: 3, 500
```

9.2 Out-of-bag (OOB) error estimation

RF allows CV-like error estimation without doing CV. For every tree, those in the bootstrap sample are training data and those left out can serve as validation data.

Let w_{bi} be the number of copies of observation i in the b-th bootstrap sample. Let $B_i = \#\{b : w_{bi} = 0\}$ be the number of bootstrap samples not containing observation i. (Note that $E(B_i) = e^{-1}B \approx 0.37B$.) For observation i, let $\hat{r}_{\text{rf}}^{(i)}(x_i)$ be the out-of-bag prediction for x_i.

- For regression problems, $\hat{r}_{\text{rf}}^{(i)}(x_i) = \frac{1}{B_i}\sum_{b:w_{bi}=0}\hat{r}_b(x_i)$;
- for classification problems, $\hat{r}_{\text{rf}}^{(i)}(x_i)$ is the majority vote from $\{\hat{r}_b(x_i) : w_{bi} = 0\}$.

Let $\text{OOB}_i = L(y_i, \hat{r}_{\text{rf}}^{(i)}(x_i))$. The **overall OOB error** is $\text{OOB} = \frac{1}{n}\sum_i \text{OOB}_i$. When B is sufficiently large, the OOB error estimate is equivalent to leave-one-out cross-validation error.

The number of trees not containing an observation is in the `oob.times` element. The average fraction of trees not containing an observation should approximately $e^{-1} = 0.368$.

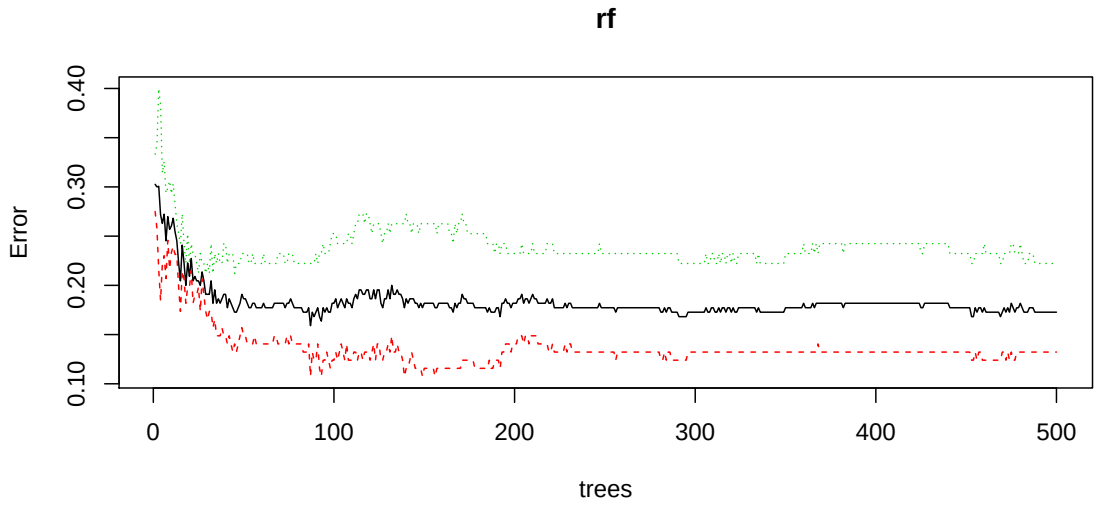
```
mean(rf$oob.times / rf$ntree); exp(-1)  ## they are close
```

The `err.rate` element contains the overall OOB error and the OOB errors stratified by the outcome categories, as more and more trees are built. The last row has the final OOB error based on all trees.

```
head(rf$err.rate); tail(rf$err.rate)
rf$err.rate[rf$ntree, ]  ## last row
```

The function `plot.randomForest()` displays these numbers in a plot (`col=1:3` for the three columns).

```
plot(rf)  ## same as the matplot code in next line
matplot(rf$err.rate, type='l', xlab='trees', ylab='Error')
matplot(rf$err.rate[,'OOB'], type='l', xlab='trees', ylab='Error')  ## overall OOB error only
```



Using `predict()` without providing a `newdata` will give the *fitted values*, which are the OOB predictions for the observation. Using `predict()` with a dataset specified for the `newdata=` argument will give the *predicted values*. For other models we have learned, if the `newdata` is the dataset we used to build the model, these two give the same results. However, for random forests, these give quite different results. Let us check.

```
table(Heart2.train$AHD, predict(rf))                ## fitted values (OOB)
table(Heart2.train$AHD, predict(rf, Heart2.train))  ## "predicted" values; wrong
```

In random forests, the individual trees severely overfit data. For every observation, because it is in ~63% of the bootstrap samples, ~63% of the trees "fit" the observation very well. As a result, the majority vote from *all* trees almost always give the "correct" prediction. This is what happened in `predict(rf, Heart2.train)`. The real performance is reflected in OOB errors. To obtain the fitted value for an observation, only those trees that did not use the observation will be used. This is what happened in `predict(rf)`.

The **apparent error rate** is the error rate based on the results from `predict(rf, Heart2.train)`. It is known for being wildly optimistic.

Thus, all measures of model performance are based on OOB errors: `predicted` contains the fitted values (the result of OOB majority vote); `votes` contains the distribution of OOB predictions from individual trees; `confusion` is the corresponding confusion matrix between the observed outcomes and the fitted values.

```
all.equal(rf$predicted, predict(rf))  ## TRUE
dim(rf$votes); head(rf$votes)
table(rf$predicted, rf$votes[,2] > 0.5)  ## same results

rf$confusion             ## same as table(Heart2.train$AHD, rf$predicted)
rf$err.rate[rf$ntree, ]  ## the `class.error` values are also here
```

We can make predictions on new data using `predict()`.

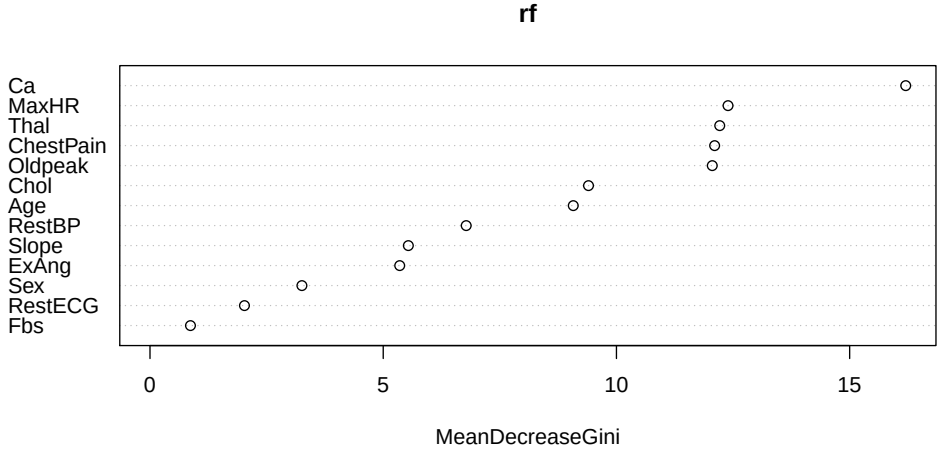
```
rfyhat.test = predict(rf, Heart2.test)
table(Heart2.test$AHD, rfyhat.test)                        ## test set confusion matrix
sum(Heart2.test$AHD != rfyhat.test) / length(rfyhat.test)  ## test set error rate
```

The test set error rate is very similar to the training set OOB error rate.

9.3 Variable importance

To measure the importance of a feature, we record the reduction of impurity (MSE, Gini, etc.) whenever the variable is used to split a node, compute the total reduction for every tree, and then average over all trees. The function `importance()` and `varImpPlot()` display the importance values stored in the `importance` element (named `MeanDecreaseGini`).

```
importance(rf)  ## show rf$importance
varImpPlot(rf)  ## same as dotchart(rf$importance[, 'MeanDecreaseGini']) except order
```

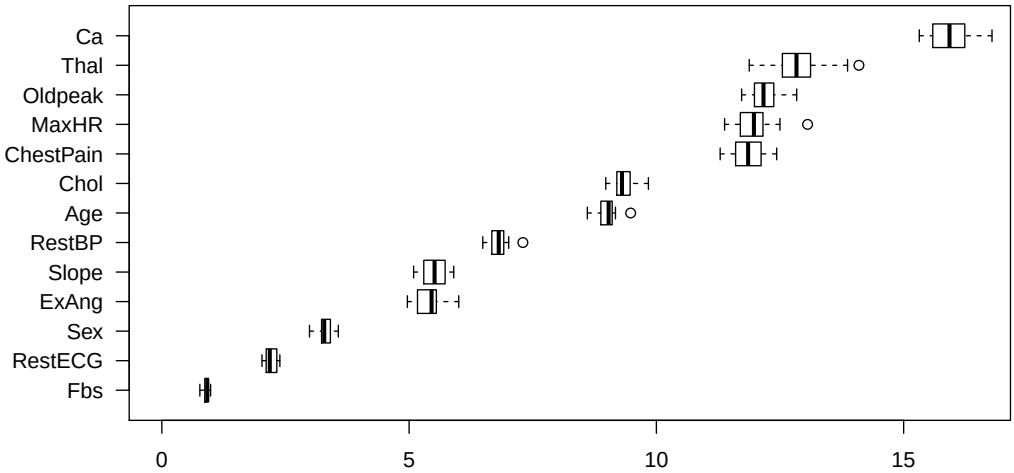


The results may vary quite a bit between one run and another because of the randomness introduced in (1) bootstrap sampling, and (2) selection of a random subset of features for each split.

```
varImpPlot(randomForest(AHD ~ ., data=Heart2.train))  ## repeat a few times
```

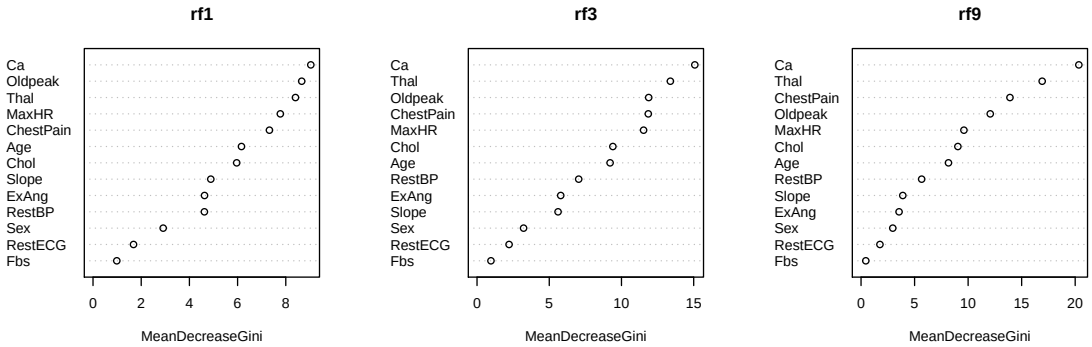
Below we summarize the variation of the results across 20 runs.

```
set.seed(2018)
importance.multirun = matrix(,20,13)  ## 13 predictors
for(i in 1:20)
    importance.multirun[i,] = randomForest(AHD ~ ., data=Heart2.train, ntree=500)$importance
colnames(importance.multirun) = rownames(rf$importance)

par(mar=c(3,5,1,1))
idx = order(apply(importance.multirun, 2, median))  ## order of features by median of importance
boxplot(importance.multirun[, idx], horizontal=T, las=1, ylim=c(0,16.5))
#dotchart(importance.multirun[, idx], las=1, ylim=c(0,20))
```



The smaller m the more spread-out of relative importance over the variables. Using a small m has some similarity to ridge regression, which tends to share the coefficients evenly among correlated variables.

```
set.seed(2019)
rf1 = randomForest(AHD ~ ., data=Heart2.train, mtry=1)
rf3 = randomForest(AHD ~ ., data=Heart2.train, mtry=3)
rf9 = randomForest(AHD ~ ., data=Heart2.train, mtry=9)

varImpPlot(rf1); varImpPlot(rf3); varImpPlot(rf9)
```



The test set performance for RF models using different `mtry` values:

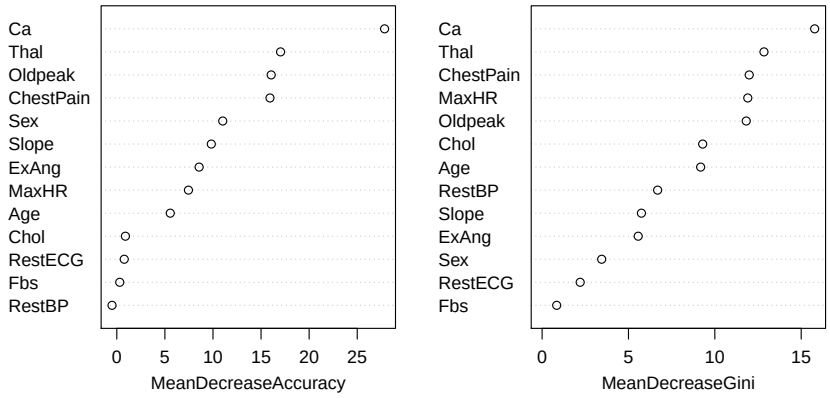
```
table(Heart2.test$AHD, predict(rf1, Heart2.test))  ## 13 errors / 77
table(Heart2.test$AHD, predict(rf3, Heart2.test))  ## 12 errors / 77
table(Heart2.test$AHD, predict(rf9, Heart2.test))  ## 16 errors / 77
```

Mean decrease in accuracy is calculated when `importance=T` is specified. The help page for `importance()` explains what the measure is: For each tree b, the prediction error e_b on the OOB portion of the data is recorded (error rate for classification, MSE for regression). Then for each predictor j, permute it in the OOB portion, calculate the prediction error again as e_{bj}. Their difference, $d_{bj} = e_{bj} - e_b$, is the amount of additional error caused by permuting predictor j (effectively making it noninformative). The mean of d_{bj} over all trees is stored as column `MeanDecreaseAccuracy` in the `importance` element, while the standard deviation is stored in the `importanceSD` element. The coefficient of variation (i.e., mean/SD) is reported when `importance()` is called.

```
set.seed(2018)
rf = randomForest(AHD ~ ., data=Heart2.train, importance=T)
importance(rf)
all.equal(importance(rf)[,1:3], rf$importance[,1:3] / rf$importanceSD)  ## TRUE
all.equal(importance(rf)[,4],   rf$importance[,4])  ## TRUE
```

In this case, `varImpPlot()` will show both criteria.

```
varImpPlot(rf)
```



Again, the results vary quite a bit from one run to another.

```
varImpPlot(randomForest(AHD ~ ., data=Heart2.train, importance=T))
```

Local variable importance is the mean decrease of accuracy of the predictors for each observation when it is OOB. To do this, set `localImp=T`. The results are stored in the element `localImportance`.

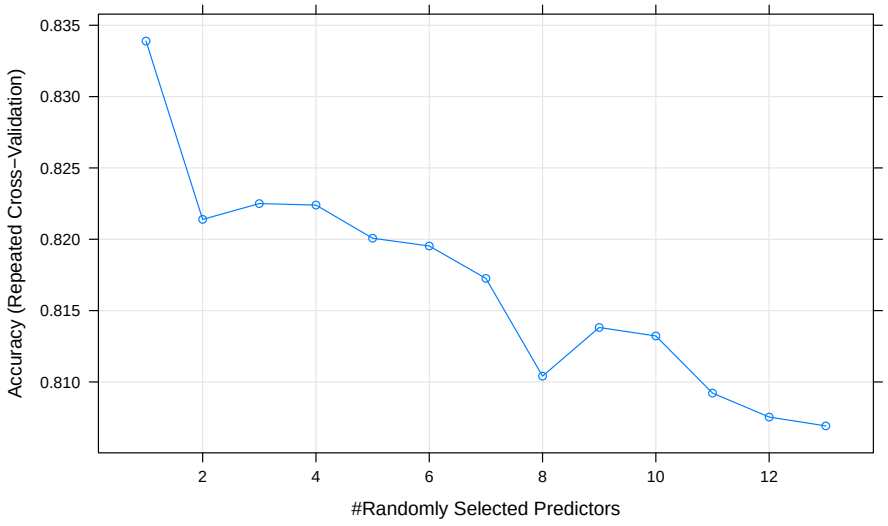
```
set.seed(2018)
rf = randomForest(AHD ~ ., data=Heart2.train, localImp=T)
dim(rf$localImportance)  ## 13 x 220
matplot(rf$localImportance[,1:5], type='l')  ## first 5 observations
#rf$localImportance %*% rf$oob.times
```

9.4 Cross-validation to select m

The R `caret` package can do cross-validation to select the hyperparameter m. By default, a grid over the hyperparameter is selected by the `train()` function. To change that, use either `tuneLength=` to specify the number of values in a grid or `tuneGrid=` to specify the values in a data frame. By default, `summaryFunction=defaultSummary` and `metric=Accuracy`. If `summaryFunction` is set to be `twoClassSummary`, `metric` can only be `ROC`.

When I ran 8-times 10-fold cross-validation on this dataset, it took 2 minutes on my laptop. The results show that `mtry=1` gives the best performance.

```
library(caret)
cvCtrl = trainControl(method="repeatedcv", number=10, repeats=8,  ## 10-fold CV repeated 8 times
                      #summaryFunction=twoClassSummary,
                      classProbs=TRUE)
set.seed(2018)
fitRFcaret = train(x=Heart2.train[,1:13], y=Heart2.train$AHD, trControl=cvCtrl,
                   tuneGrid=data.frame(mtry=1:13),
                   #tuneLength=4,
                   #metric="ROC",  ## when summaryFunction=twoClassSummary
                   method="rf", ntree=500)
plot(fitRFcaret)
```



The object obtained from the `train()` function contains other results.

```
fitRFcaret
names(fitRFcaret)
```

```
fitRFcaret$results
fitRFcaret$bestTune$mtry
```

It also contains the model using the 'optimal' hyperparameter selected from `train()`. This "final model" was fit using all data in the training set.

```
fitRFcaret$finalModel
fitRFcaret$finalModel$confusion  ## OOB confusion matrix
```

The following gives test set confusion matrix when applying the "final" model.

```
table(Heart2.test$AHD, predict(fitRFcaret$finalModel, Heart2.test))
```

Note 1: `train()` can be called in two ways. `train(x,y)` is recommended because it handles categorical predictors nicely. If you use `train(formula, data)`, dummy variables are created for categorical predictors with more than two categories, which can cause problems when using functions such as `predict()`. Here is an example.

```
cvCtrl2 = trainControl(method="repeatedcv", number=5, repeats=4, classProbs=TRUE)
fitRFcaret2 = train(AHD ~ ., data=Heart2.train, trControl=cvCtrl2,
                    method="rf", ntree=200)
predict(fitRFcaret2$finalModel, Heart2.test)  ## Error, due to reason below

fitRFcaret2$finalModel$importance  ## notice the variable names
fitRFcaret$finalModel$importance
```

Note 2: `caret` also has a function `tuneRF()`, which is not as useful as the `train()` approach above.

```
tuneRF(x=Heart2.train[,1:13], y=Heart2.train$AHD, stepFactor=1.5, improve=0.01, ntreeTry=50)
```

Other evaluations using caret: We can try several models under different settings, put them together into a list, and evaluate them using `resamples()`. As an example, I fit 4 RF models with different ntree values and compare them.

```
cvCtrl2 = trainControl(method="repeatedcv", number=5, repeats=4, classProbs=TRUE)
modellist = list()
for (ntree in c(200, 400, 600, 800))
    modellist[[toString(ntree)]] = train(x=Heart2.train[,1:13], y=Heart2.train$AHD,
                                         method="rf", trControl=cvCtrl2, ntree=ntree)

results = resamples(modellist)
names(results)
results$timings
summary(results)
dotplot(results)
```

`caret` allows users to define a new method for `train()`. An example is in https://machinelearningmastery.com/tune-machine-learning-algorithms-in-r/, under "Extend Caret" in Section 3, where the author defines a new RF method so that the grid search can be done over a grid of combinations of mtry and ntree.

9.5 An example with the NCI60 dataset

Classification RF on the `NCI60` dataset, which contains gene expression of 6830 genes for 64 tumor samples. 59 samples are classified into 9 types. 5 samples are labelled with other names.

```
library(ISLR)
str(NCI60)
table(NCI60$labs)  ## distribution of label
dd = data.frame(NCI60$data)
dd$labs = as.factor(NCI60$labs)  ## randomForest() requires categorical outcome to be a factor
```

A first run.

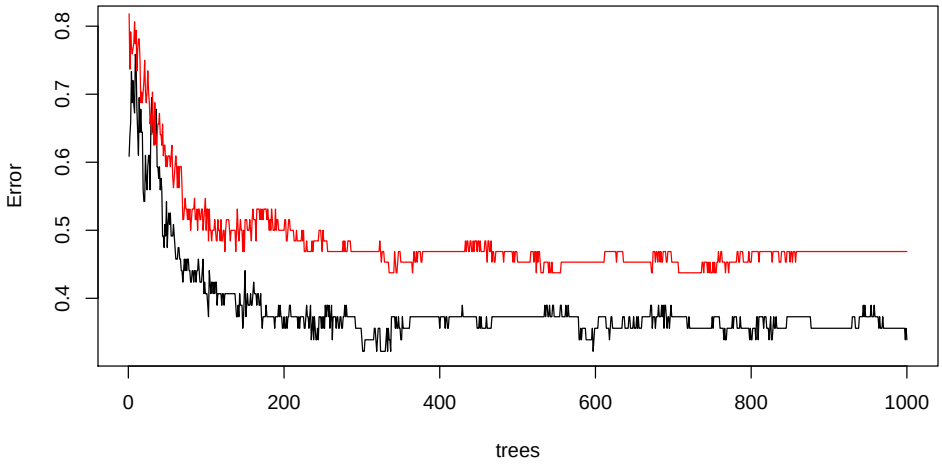
```
set.seed(2018)
rf = randomForest(labs ~ ., data=dd, ntree=1000)
plot(rf$err.rate[,1], type='l', xlab='trees', ylab='Error')
```

The 5 unique categories are problematic. I remove them and then re-run.

```
idx = dd$labs %in% names(table(NCI60$labs))[table(NCI60$labs) > 1]
dd2 = dd[idx,]
dd2$labs = as.factor(as.character(dd2$labs))  ## redefine the factor levels

rf2 = randomForest(labs ~ ., data=dd2, ntree=1000)
```

The classification error drops after removing the 5 categories.

```
plot(rf2$err.rate[,1], type='l', xlab='trees', ylab='Error')
points(rf$err.rate[,1], type='l', col=2)
```



9.6 An example of regression RF

We use the `Wage` dataset:

```
library(ISLR)
levels(Wage$education)
Wage$education = factor(Wage$education, levels(Wage$education), ordered=T)

rf = randomForest(wage ~ . - logwage, data=Wage)  ## all predictors except `logwage`
rf
names(rf)
rf$mtry; rf$ntree  ## values we did not specify; 3, 500
```

The `mse` element contains the overall OOB MSE as more and more trees are built. It is the counterpart of `err.rate`.

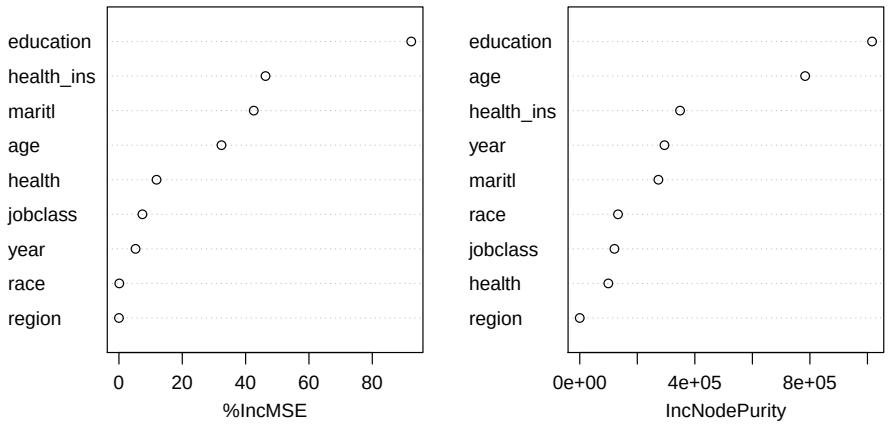
```
plot(rf)  ## same as plot(rf$mse, type='l', xlab='trees', ylab='Error')
plot(Wage$wage, rf$predicted)
cor(Wage$wage, rf$predicted, method='spearman')  ## 0.63
```

Again, below is the wrong. They are the "apparent MSEs", not OOB MSEs.

```
plot(Wage$wage, predict(rf, Wage))
cor(Wage$wage, predict(rf, Wage), method='spearman')  ## 0.81
```

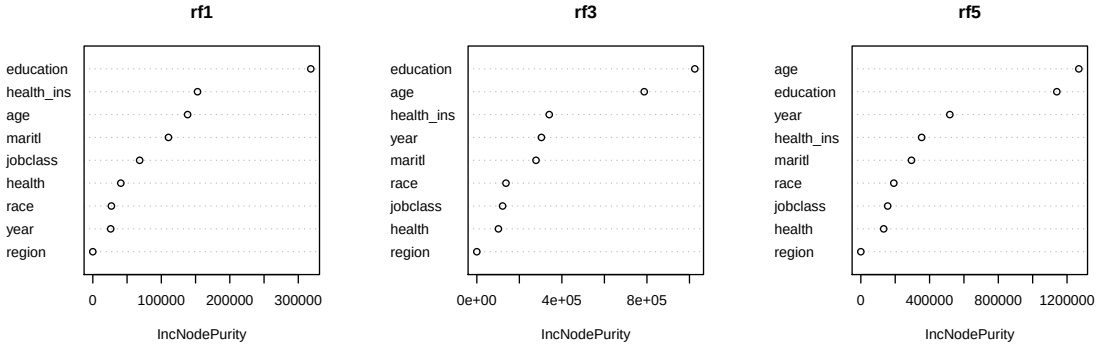
The `importance` element contains `IncNodePurity`. When `importance=T` is used, `%IncMSE` is the counterpart of `MeanDecreaseAccuracy`.

```
set.seed(2019)
rf = randomForest(wage ~ . - logwage, data=Wage, importance=T)
importance(rf)
varImpPlot(rf)
```



The smaller m the more spread-out of relative importance over the variables.

```
set.seed(2019)
rf1 = randomForest(wage ~ .-logwage, data=Wage, mtry=1)
rf3 = randomForest(wage ~ .-logwage, data=Wage, mtry=3)
rf5 = randomForest(wage ~ .-logwage, data=Wage, mtry=5)

varImpPlot(rf1); varImpPlot(rf3); varImpPlot(rf5)
```



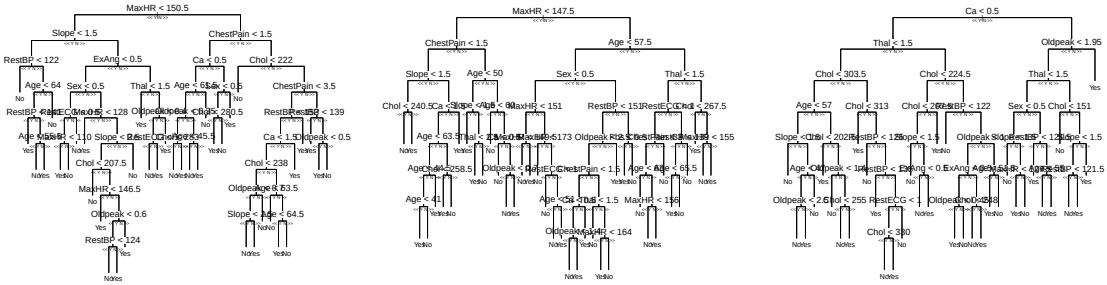
9.7 Individuals trees in RFs

Individuals trees can be obtained with `getTree()` in the `randomForest` package. For example, to get the 100th tree from the object `rf`, use `getTree(rf, 100)`. Unfortunately, the result of `getTree()` is in a format different from the frame format used by the `rpart` and `tree` packages, we cannot use the graphics functions from those packages.

```
set.seed(2018)
rf = randomForest(AHD ~ ., data=Heart2.train)
getTree(rf, 100)
```

An R package `reprtree` posted on the GitHub can be used to plot the individual trees. The plots are often not useful because the indivial trees often have a high depth. To install the `reprtree` package, use `devtools::install_github('araastat/reprtree')`.

```
reprtree:::plot.getTree(rf, k=20)
reprtree:::plot.getTree(rf, k=30)
reprtree:::plot.getTree(rf, k=50)
```



To redraw a tree without text, we use `as.tree()` to convert the result from `getTree()` to a `tree` class, and then plot it.

```
library(rpart)
tr100 = reprtree:::as.tree(getTree(rf, k=20, labelVar=T), rf)
plot(tr100, type='uniform')
#text(tr100, split=T, cex=.8)  ## adding text may make the plot look too crowded
```

The `reprtree` package also implements a method to generate "representative trees" for random forests (Banerjee, Ding, Noone (2012). Identifying representative trees from ensembles. *Statistics in Medicine*).

```
rep = reprtree::ReprTree(rf, Heart2.train)  ## representative tree
reprtree:::plot.reprtree(rep)
```

9.8 Notes on other R packages for random forests

- `caret` (document) provides an interface for using `randomForest` and a few other RF packages. Details are here. For bagging methods, details are here. `names(getModelInfo())` provides a list of all trainable model types.
- `party` privides functions to build "conditional inference trees" (ctree) and to fit random forest models with conditional inference trees as base models.