8 PQHS 471 Notes Week 8

8.1 Week 8 Day 1

8.1.1 ISLR 8.2

8.1.2 Random Forests

Random forests are a model averaging technique using trees as base models. Bagging is a special case of random forests.

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 maximal-depth tree $\hat{r}_b$ *without pruning*. For every split during tree building, only m randomly selected features are considered.
3. The final RF model is $\hat{r}_{\text{rf}} = \frac{1}{B}\hat{r}_b$ for continuous outcome, or majority vote over $\hat{r}_b$ for categorical outcomes.

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

Notes:

- m is a hyperparameter. Once m is 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.
- This algorithm is non-deterministic.
- 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.
- Maximal-depth reduces correlation from one tree to another. Compared to bagging, the additional randomization due to a small $m < p$ results in trees with a higher variance, but are a lot less correlated.
- Maximal-depth can be slow for very large n. In `randomForest()`, one can set `nodesize=` (minimum size of terminal nodes) and `maxnodes=` (maximum number of terminal nodes).
- Majority vote is a type of average.
- 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

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

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

```r
library(randomForest)

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)

rf = randomForest(AHD ~ ., data=Heart)  ## error due to missing data

apply(is.na(Heart), 2, sum)  ## check which variable has missing data
apply(is.na(Heart), 1, sum)  ## check which observation has missing data

Heart2 = na.omit(Heart); dim(Heart2)  ## 297, 14
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
```

Now we fit a RF model:

```
rf = randomForest(AHD ~ ., data=Heart2)
rf
names(rf)
rf$mtry; rf$ntree
```

Out-of-bag (OOB) error estimation: RF allows CV-like error estimation without CV. For every tree, those in the bootstrap sample are training data and those left out are test data. Let w_{bi} be the number of copies of observation i is in bootstrap sample b. 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 $\hat{r}_{\text{rf}}^{(i)}(x_i) = \frac{1}{B_i}\sum_{b:w_{bi}=0}\hat{r}_b(x_i)$, or majority vote. Then $\text{OOB}_i = L(y_i, \hat{r}_{\text{rf}}^{(i)}(x_i))$, and $\text{OOB} = \frac{1}{n}\sum_i \text{OOB}_i$. When B is sufficiently large, the OOB error estimate is equivalent to leave-one-out cross-validation error.

```
mean(rf$oob.times/rf$ntree); exp(-1)
```

For a model fitted with `randomForest()`, its component `err.rate` contains the overall OOB error and the OOB errors stratified by the outcome categories as the number of trees increases. The function `plot.randomForest()` displays these errors in a plot.

```
plot(rf)  ## same as 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
```

All measures of model performance are based on the OOB samples: `votes` contains the distribution of OOB predictions, `predicted` is the result of majority vote, `confusion` is the corresponding confusion matrix.

```
table(rf$predicted, rf$votes[,2] >= 0.5)  ## "predicted" is based on "votes"
rf$confusion  ## same as table(Heart2$AHD, rf$predicted)
rf$err.rate[rf$ntree, ]
```

We can make predictions on new data using `predict()`. BUT we should not re-predict the training set because it would give near perfect accuracy. This is called the "apparent error rate" and is known for being wildly optimistic.

```
#rfyhat = predict(rf, testdata)
rfyhat = predict(rf)  ## without newdata, this returns rf$predicted

table(Heart2$AHD, predict(rf))
table(Heart2$AHD, predict(rf, Heart2))  ## This is wrong!
```

Variable importance: To measure the importance of a variable, 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.

For a classification model fitted with `randomForest()`, its component `importance` contains `MeanDecreaseGini`. The function `importance()` and `varImpPlot()` display the importance values.

```
rf = randomForest(AHD ~ ., data=Heart2)
importance(rf)  ## show rf$importance
varImpPlot(rf)  ## same as dotchart(rf$importance[, 'MeanDecreaseGini']) except order
```

The results can vary quite a bit between runs because of the randomness introduced during model fitting: (1) bootstrap sampling, and (2) a random subset of features are considered at each split.

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

Below we evaluate the variation of results across runs.

```
importance.multirun = matrix(,20,13)
for(i in 1:20)
```

```r
  importance.multirun[i,] = randomForest(AHD ~ ., data=Heart2, ntree=500)$importance
colnames(importance.multirun) = rownames(rf$importance)
idx = order(apply(importance.multirun, 2, median))

par(mar=c(4,6,1,1))
boxplot(importance.multirun[, idx], horizontal=T, las=1, ylim=c(0,20))
#dotchart(importance.multirun[, idx], las=1, ylim=c(0,20))
```

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

```r
varImpPlot(randomForest(AHD ~ ., data=Heart2, mtry=1))
varImpPlot(randomForest(AHD ~ ., data=Heart2, mtry=3))
varImpPlot(randomForest(AHD ~ ., data=Heart2, mtry=10))
```

Another measure of **variable importance** is the *mean decrease in accuracy*, which is calculated by `randomForest()` when `importance=T` is specified.

```r
rf = randomForest(AHD ~ ., data=Heart2, importance=T)
importance(rf)  ## different from rf$importance except the last column
varImpPlot(rf)
```

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}, and $d_{bj} = e_{bj} - e_b$, which 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 in the component `importance` while the standard deviation is stored in `importanceSD`. The coefficient of variation (i.e., mean/SD) is reported when `importance()` is called.

```r
all.equal(importance(rf)[,'MeanDecreaseAccuracy'],  ## True
  rf$importance[,'MeanDecreaseAccuracy']/rf$importanceSD[,'MeanDecreaseAccuracy'])
```

One can also get **local variable importance**, which 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 component `localImportance`.

```r
rf = randomForest(AHD ~ ., data=Heart2, localImp=T)
dim(rf$localImportance)  ## 13 x 297
matplot(rf$localImportance[,1:5], type='l')  ## first 5 observations
#rf$localImportance %*% rf$oob.times
```

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.

```r
library(caret)
cvCtrl = trainControl(method="repeatedcv", number=5, repeats=4,  ## 5-fold CV repeated 4 times
                      classProbs=TRUE, summaryFunction=twoClassSummary)
fitRFcaret = train(x=Heart2[,1:13], y=Heart2$AHD, trControl=cvCtrl,
                   #tuneGrid=data.frame(mtry=c(2,3,4,5)),
                   #tuneLength=4,
                   method="rf", metric="ROC", ntree=50)  ## small ntree for class demonstration
fitRFcaret
names(fitRFcaret)
fitRFcaret$bestTune$mtry
fitRFcaret$finalModel$confusion  ## same as table(Heart2$AHD, fitRFcaret$finalModel$predicted)
```

Again, the following generate “apparent error rates” that are too good.

```r
table(Heart2$AHD, predict(fitRFcaret))
table(Heart2$AHD, predict(fitRFcaret, Heart2))
table(Heart2$AHD, predict(fitRFcaret$finalModel, Heart2))
```

`train(x, y)` is recommended because it does not create dummy variables for categorical predictors with more than two categories. If you use `train(formula, data)`, dummy variables are created for categorical predictors with more than two categories. This may cause problems when using functions such as `predict()`.

```
fitRFcaret2 = train(AHD ~ ., data=Heart2, trControl=cvCtrl,
                    method="rf", metric="ROC", ntree=20)
predict(fitRFcaret2$finalModel, Heart2)  ## Error
fitRFcaret2$finalModel$importance
fitRFcaret$finalModel$importance
```

Example: Classification RF on the `NCI60` dataset, which contains gene expression of 6830 genes for 64 tumor samples. The tumors are classified into 14 types.

```
library(ISLR)
str(NCI60)
table(NCI60$labs)
dd = data.frame(NCI60$data)
dd$labs = as.factor(NCI60$labs)  ## randomForest() requires categorical outcome to be a factor
rf = randomForest(labs ~ ., data=dd, ntree=1000)
plot(rf$err.rate[,1], type='l', xlab='trees', ylab='Error')

## Remove those rare/unknown tumor types (those with only one obs)
idx = dd$labs %in% names(table(NCI60$labs))[table(NCI60$labs)>1]
dd2 = dd[idx,]
dd2$labs = as.factor(as.character(dd2$labs))  ## redefine the factor levels
rf2 = randomForest(labs ~ ., data=dd2, ntree=1000)
plot(rf2$err.rate[,1], type='l', xlab='trees', ylab='Error')
points(rf$err.rate[,1], type='l', col=2)
```

Example: Regression RF on the `Wage` dataset:

```
library(ISLR)
str(Wage)
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
```

The counterpart of `err.rate` is `mse`.

```
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')

## Again, below is wrong.
plot(Wage$wage, predict(rf, Wage))
cor(Wage$wage, predict(rf, Wage), method='spearman')
```

For a regression model fitted with `randomForest()`, its component `importance` contains `IncNodePurity`. When `importance=T` is used, `%IncMSE` is the counterpart of `MeanDecreaseAccuracy` in classification RF.

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

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

```
varImpPlot(randomForest(wage ~ .-logwage, data=Wage, mtry=1))
varImpPlot(randomForest(wage ~ .-logwage, data=Wage, mtry=3))
varImpPlot(randomForest(wage ~ .-logwage, data=Wage, mtry=5))
```

Individuals trees can be obtained with `getTree()`. 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. An R package `reprtree` posted on the GitHub can 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')`.

```
rf = randomForest(AHD ~ ., data=Heart2)
reprtree:::plot.getTree(rf, k=100)
## redraw the tree without text
library(rpart)
tr100 = reprtree:::as.tree(getTree(rf, k=100, 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)  ## representative tree
reprtree:::plot.reprtree(rep)
```

Notes on other R packages for bagging and random forests:

- `caret` (document) provides a unified interface (and a wrapper) for using `randomForest` and many other R packages. For example, it offers bagging of base models of type: lda, pls, nb (naive Bayes), ctree, svm, nnet. `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.

8.1.3 Assignment

1. Reading for next lecture: ISLR 8.2, HOML 7.
2. ISLR Chapter 8 R Labs

8.2 Week 8 Day 2 (off for midterm)