Team 2 - Homework Two

Assignment 3: KJ 8.1-8.3; KJ 8.7

Sang Yoon (Andy) Hwang

DATE:2019-11-14

Dependencies

```
# Forecast libraries
libraries('mlbench', 'AppliedPredictiveModeling')

# Regression libraries
libraries('randomForest', 'caret', 'party', 'partykit', 'gbm', 'Cubist')

# Formatting Libraries
libraries('default', 'knitr', 'kableExtra', 'mice', 'party')

# Plotting Libraries
libraries('ggplot2', 'grid', 'ggfortify')
```

(1) Kuhn & Johnson 8.1

Recreate the simulated data from Exercise 7.2:

```
set.seed(200)
simulated <- mlbench.friedman1(200, sd = 1)
simulated <- cbind(simulated$x, simulated$y)
simulated <- as.data.frame(simulated)
colnames(simulated)[ncol(simulated)] <- "y"
```

(a). Fit a random forest model to all of the predictors, then estimate the variable importance scores. Did the random forest model significantly use the uninformative predictors (V6-V10)?

Based on the result, we know that feature importances of V6-V10 are much less than V1-V5 – except for V6, all of them are negative. This shows that the random forest model did not use the uninformative predictors significantly.

```
set.seed(200)
model1 <- randomForest(y ~ ., data = simulated,
                       importance = TRUE,
                       ntree = 1000)
rfImp1 <- varImp(model1, scale = FALSE)
```

	Overall
V1	8.6053659
V4	7.8833841
V2	6.8312592
V5	2.2447503
V3	0.7415349
V6	0.1360542
V7	0.0559509
V9	0.0031962
V10	-0.0547059
V8	-0.0681958

(b). Now add an additional predictor that is highly correlated with one of the informative predictors. Fit another random forest model to these data. Did the importance score for V1 change? What happens when you add another predictor that is also highly correlated with V1? For example:

Note that `V1` is no longer the most important variable. It looks like the importance score for `V1` was partly absorbed by new predictor which underestimates true importance of `V1` - the score sum of `V1` and `duplicate1` are similar to the V1 score in (a). It makes sense as `duplicate1` contains almost the same information as `V1`. Not only that, the importance score for some variables, such as `V9` and `V10`, have rather increased as a result of addition of new variable. The order of importance is changed.

```
set.seed(200)
simulated$duplicate1 <- simulated$V1 + rnorm(200) * .1
cor(simulated$duplicate1, simulated$V1)
```

```
[1] 0.9497025
```

	Overall
V4	6.7968990
V2	6.1965079
V1	6.0509601
duplicate1	4.2064179
V5	2.2679088
V3	0.5224913
V6	0.1917214
V7	0.0376683
V9	0.0204112
V10	-0.0448119
V8	-0.0840651

(c). Use the `cforest` function in the party package to fit a random forest model using conditional inference trees. The party package function `varimp` can calculate predictor importance. The `conditional` argument of that function toggles between the traditional importance measure and the modified version described in Strobl et al. (2007). Do these importances show the same pattern as the traditional random forest model?

We performed both `varimp(, conditional = T)` and `varimp(, conditional = F)` to compare `varimp` of `cforest` in terms of permutation importance and conditional permutation importance.

1. RF vs CF Given that no correlated term is added, the importance pattern is similar except for the fact that V4 is now the most important feature in CF.
2. RF vs CF (with correlated term added) Given that correlated term is added, the importance score for `duplicate1` is much smaller in CF. This is the pin point difference between importance based on Gini coefficient (decision tree) and permutation test using p-value (conditional inference tree).

3. CF conditional vs CF with correlated term added and conditional When `conditional = T`, we perform conditional permutation test for measuring feature importance instead. Note that `duplicate1` has even smaller importance in `CF.cor.cond` than in `CF.cor`. For `CF.cor.cond`, notice `V1` became 3rd most important feature when it was 2nd most important for `CF.cor`. This is because conditional permutation helps uncovering the spurious correlation between `V1` and `duplicate1`.

In summary, we learned that `CF` model surpresses the importance score of `duplicate1` which helps maintain the importance of `V1`. When `conditional = TRUE` in `varimp` for `CF` model, the importance score of `duplicate1` is even smaller.

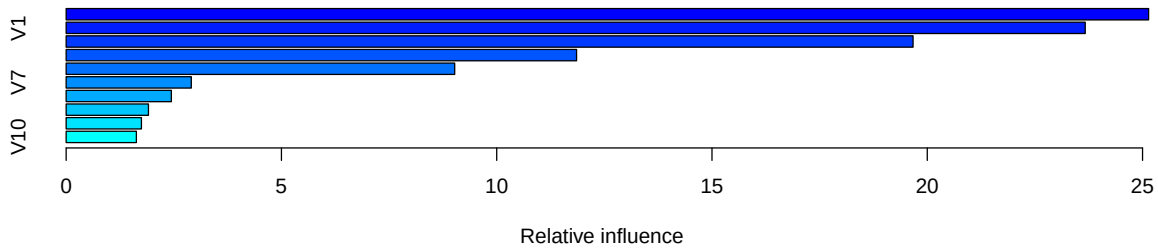
features	RF	CF	RF.cor	CF.cor	CF.cond	CF.cor.cond
duplicate1	NA	NA	4.2064179	0.3872163	NA	0.0349069
V1	8.6053659	10.0871749	6.0509601	9.5267417	3.4058876	2.3728223
V2	6.8312592	7.6547399	6.1965079	7.6972528	4.7488981	4.5369424
V3	0.7415349	0.0314208	0.5224913	0.0375929	0.0121964	0.0076374
V4	7.8833841	10.3995239	6.7968990	10.2764564	5.8314126	6.2393285
V5	2.2447503	2.1331052	2.2679088	2.3767165	0.7388594	0.8462725
V6	0.1360542	0.0236553	0.1917214	0.0358447	0.0120596	0.0206135
V7	0.0559509	0.1071825	0.0376683	0.0689175	0.0182668	0.0064646
V8	-0.0681958	-0.0457278	-0.0840651	-0.0349693	-0.0005354	-0.0047206
V9	0.0031962	-0.0605760	0.0204112	-0.0424421	-0.0080181	-0.0023240
V10	-0.0547059	-0.0021033	-0.0448119	0.0254297	-0.0030144	0.0061051

(d). Repeat this process with different tree models, such as boosted trees and Cubist. Does the same pattern occur?

For boosting method `GBM` without `duplicate1`, we see that `V4` is the most important followed by `V1`. Compared to `RF` without `duplicate1`, the general pattern in `GBM` is still similar as most important features still range from `V1` to `V5`. For `GBM` with `duplicate1`, pattern is still similar to `RF` with `duplicate1` where the importance score of `V1` shrinks substantially due to the presence of `duplicate1`.

For rule-based `Cubist`, the general pattern is still similar as it assigns the most of importance scores for `V1` to `V5` and just like for `CF`, `duplicate1` still abosorves the score from `V1` and other predictors.

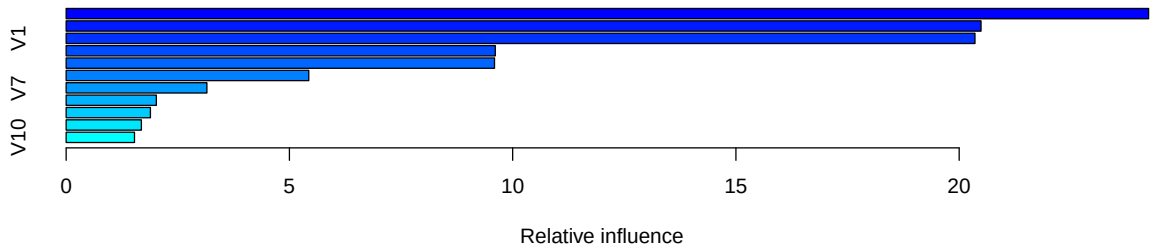
```
[1] "GBM"
```




```
    var   rel.inf
V4   V4 25.140661
V1   V1 23.668702
V2   V2 19.670210
V5   V5 11.854998
V3   V3  9.024607
V7   V7  2.906671
V6   V6  2.443473
V8   V8  1.911299
```

```
V9   V9  1.748566
V10 V10  1.630814
```

```
[1] "GBM - with duplicate1"
```




```
                  var   rel.inf
V4                 V4 24.242321
V2                 V2 20.489928
V1                 V1 20.353713
V5                 V5  9.611193
V3                 V3  9.593772
duplicate1 duplicate1  5.433687
V7                 V7  3.152069
V6                 V6  2.020182
V9                 V9  1.887429
V8                 V8  1.685039
V10               V10  1.530667
```

```
[1] "Cubist"
```

```
    Overall
V1     71.5
V3     47.0
V2     58.5
V4     48.0
V5     33.0
V6     13.0
V7      0.0
V8      0.0
V9      0.0
V10     0.0
```

```
[1] "Cubist - with duplicate1"
```

```
           Overall
V3            56.5
V1            62.0
V2            58.0
V5            26.0
V4            34.0
V6            12.5
duplicate1     9.0
V8             2.5
V7             0.0
```

```
V9             0.0
V10            0.0
```

(2) Kuhn & Johnson 8.2

Use a simulation to show tree bias with different granularities.

According to the text (pg.182), `Finally, these trees suffer from selection bias: predictors with a higher number of distinct values are favored over more granular predictors`, we know that there is a high probability that predictors with a higher number of distinct values are favored over the predictors with less number of distinct values.

Our equation is:

- `y <- (V1 + V2) + rnorm(200,mean=0,sd=4)`
- `V1 <- rep(1:2,each=100)`
- `V2 <- rnorm(200, mean=0, sd=2)`
- `V3 <- rep(1:100,each=2)`
- `V4 <- rnorm(200, mean=0, sd=3)`

Indeed, we have equal number of samples for both `V1` and `V2` but since `V2` has higher number of distinct values, `varImp` suggests `V2` is more important (and higher `cor` between `V2` and `y`) than `V1`. This confirms the hypothesis from pg.182.

```
            y          V1           V2           V3          V4
y  1.00000000  0.06889140  0.433709605  0.125707042  0.01182582
V1 0.06889140  1.00000000 -0.039289372  0.866068708 -0.02926059
V2 0.43370960 -0.03928937  1.000000000 -0.005339808 -0.03761656
V3 0.12570704  0.86606871 -0.005339808  1.000000000 -0.02766897
V4 0.01182582 -0.02926059 -0.037616564 -0.027668972  1.00000000
```

```
        Overall
V1  0.081520573
V2  8.510388045
V3 -0.004432213
V4 -0.072687915
```

(3) Kuhn & Johnson 8.3

> In stochastic gradient boosting the bagging fraction and learning rate will govern the construction of the trees as they are guided by the gradient. Although the optimal values of these parameters should be obtained through the tuning process, it is helpful to understand how the magnitudes of these parameters affect magnitudes of variable importance. Figure 8.24 provides the variable importance plots for boosting using two extreme values for the bagging fraction (0.1 and 0.9) and the learning rate (0.1 and 0.9) for the solubility data. The left-hand plot has both parameters set to 0.1, and the right-hand plot has both set to 0.9:
>
> > **(a). Why does the model on the right focus its importance on just the first few of predictors, whereas the model on the left spreads importance across more predictors?**

From the text (pg.206), we know that boosting employs "greedy" strategy of choosing the optimal weak learner at each stage. In other words, regularization (or shrinkage) strategy is used to find a fraction of the current predicted value is added to the previous iteration's predicted value. This fraction is called `learning rate` (between 0 and 1). As the fraction becomes larger, less iteration is required and vice-versa.

In order to furthur improve the bosting technique through reducing prediction variance for bagging (reducing error rate on test set), Friedman updated the technique with a random sampling sceme by randomly select a fraction of the traning data known as `bagging fraction`. The iteration then is based only on the sample of

data. This new technique is called `stochastic gradient boosting`. This technique is heavily depedent on previous tree - indeed, each tree is based on previous tree or correlated one another. Imagine one of the trees split on few features more often than the others. The next tree, which will be built upon the previous tree, will still be based on the previous error and again split on the same features again. In this technique, there is a chance that only few predictors (many of the same predictors) would be chosen in almost all trees and hence inflating feature importance for few predictors.

Having said that, let's think about what it means by when we have `learning rate` of 0.9 vs 0.1. Intuitvely, when `learning rate` is larger, it means larger fraction of the current predicted value is being added to the previous iteration's predicted value. In other words, it means that we use more of the same predictors will be selected among the trees. This is why you tend to favor few predictors (since they are selected multiple times) when you have higher `learning rate`.

Also, when you have larger `bagging fraction`, it means for each iteration, each tree will more likely see the same data samples, therefore choose the same predictor as before.

The smaller rate/fraction is self-explanatory as it is just an opposite case to the ones above.

(b). Which model do you think would be more predictive of other samples?

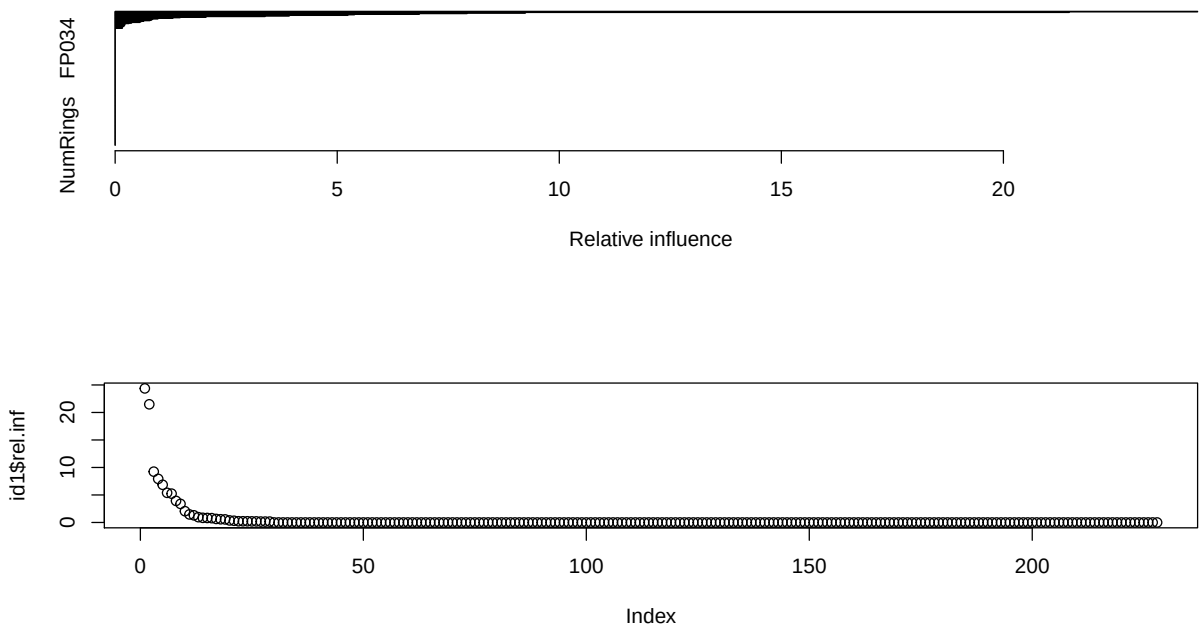
Given that lower `learning rate` usually produces higher predictive power and higher `bagging fraction` tend to reduce variance, we really have to test on test set in order to answer this question. From testing result using `solubility` data set, we confirmed that right one (0.9/0.9) gives lower RMSE on test set.

```
  RMSE_0.1_0.1 RMSE_0.9_0.1 RMSE_0.1_0.9 RMSE_0.9_0.9
1    0.7670641    0.8137618     1.546084    0.7196716
```

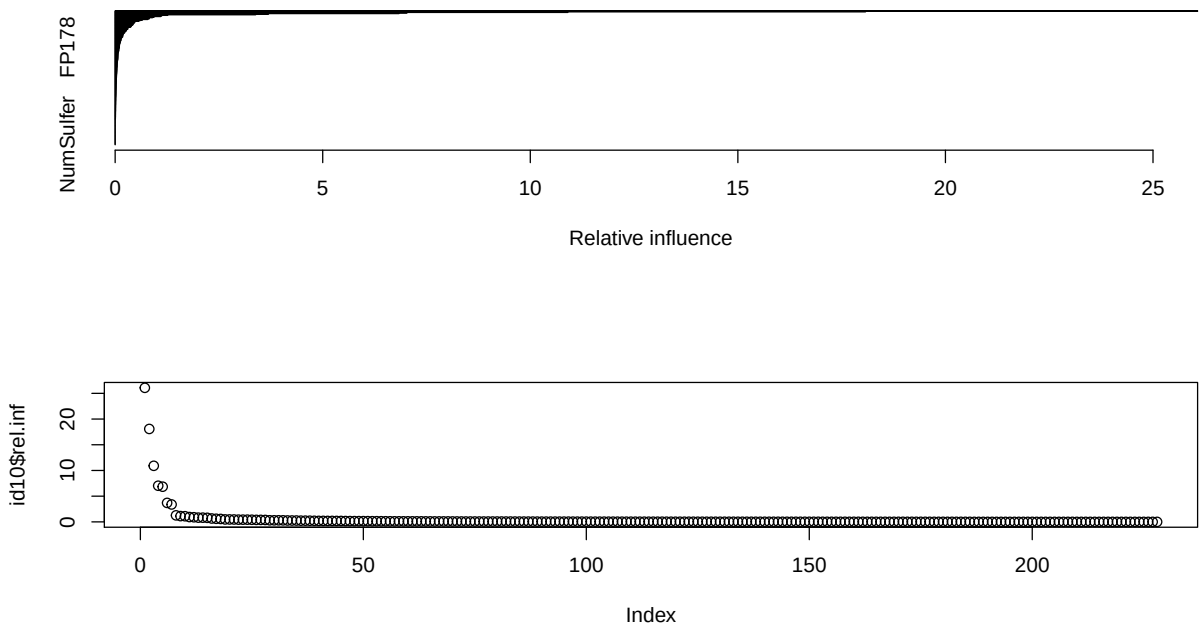
(c). How would increasing interaction depth affect the slope of predictor importance for either model in Fig.8.24?

As you can see, increasing `interaction.depth` decreases the slope of predictor importance since tree would grow in a more sophisticated way and hence more predictors are chosen in splitting process.

```
[1] "interaction.depth = 1 with shrinkage = 0.1"
```



```
[1] "interaction.depth = 10 with shrinkage = 0.1"
```

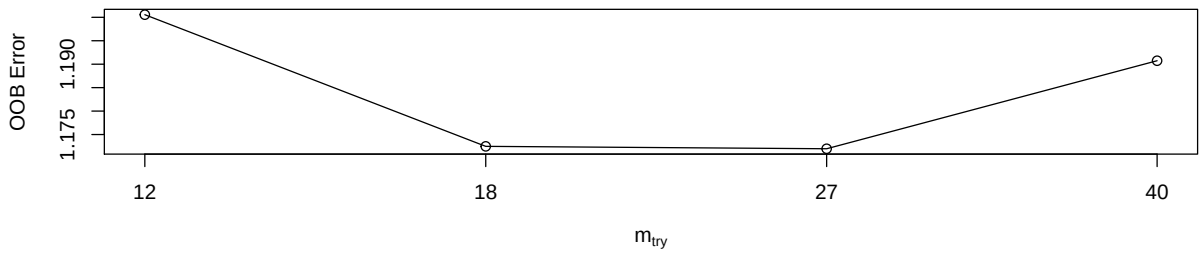


(4) Kuhn & Johnson 8.7

Refer to Exercises 6.3 and 7.5 which describe a chemical manufacturing process. Use the same data imputation, data splitting, and pre-processing steps as before and train several tree-based models:

(a). Which tree-based regression model gives the optimal resampling and test set performance?

```
mtry = 18  OOB error = 1.17248
Searching left ...
mtry = 12   OOB error = 1.200555
-0.02394523 1e-05
Searching right ...
mtry = 27   OOB error = 1.171969
0.0004355745 1e-05
mtry = 40   OOB error = 1.190739
-0.01601582 1e-05
```



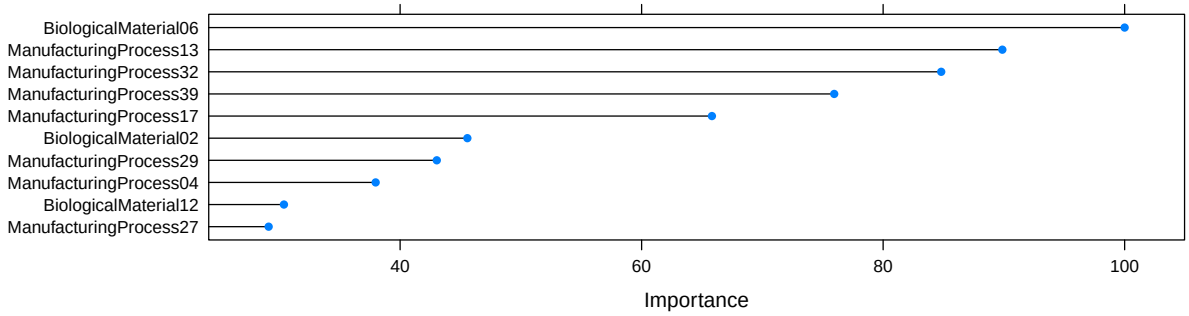

```
   RMSE_RF RMSE_GBM RMSE_CUBIST Rsquare_RF Rsquare_GBM Rsquare_CUBIST
1 1.383378 1.222199    1.134999  0.6057993   0.6727696      0.7320794
```

From hyperparameter tuning of RF, GBM and Cubist, we confirmed Cubist had the lowest RMSE on test set of 1.1349986.

(b). Which predictors are most important in the optimal tree-based regression model? Do either the biological or process variables dominate the list? How do the top 10 important predictors compare to the top 10 predictors from the optimal linear and nonlinear models?

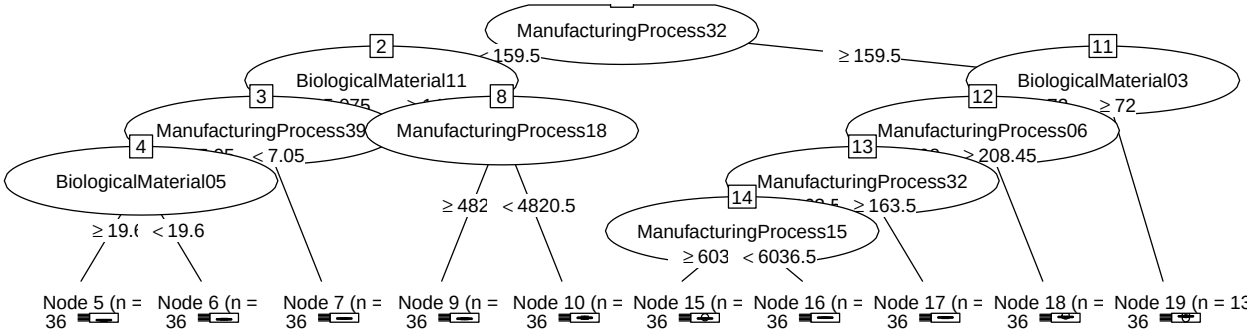
`BiologicalMaterial6` is the top predictor and `ManufacturingProcess32` is one of the top predictors for tree-based models and it had been one of the top predictors even for linear and non-linear model.

The general patterns of predictor importance ranking is similar to linear and non-linear models.



(c). Plot the optimal single tree with the distribution of yield in the terminal nodes. Does this view of the data provide additional knowledge about the biological or process predictors and their relationship with yield?

The optimal recursive partitioning tree shows that `ManufacturingProcess32` is at the top. It was one of the most important variables in GBM and Cubist models. Note that it appears twice in the plot. This proves why this variable had higher importance scores, thus highly associated with `Yield`, than most of other variables.



R Code

```
# insert code here

# (8.1)
set.seed(200)
simulated <- mlbench.friedman1(200, sd = 1)
simulated <- cbind(simulated$x, simulated$y)
simulated <- as.data.frame(simulated)
colnames(simulated)[ncol(simulated)] <- "y"

# (8.1a)
set.seed(200)
model1 <- randomForest(y ~ ., data = simulated,
                       importance = TRUE,
                       ntree = 1000)
rfImp1 <- varImp(model1, scale = FALSE)

d1 <- rfImp1[ order(-rfImp1), , drop=FALSE ]
kable(d1)

# (8.1b)
set.seed(200)
simulated$duplicate1 <- simulated$V1 + rnorm(200) * .1
cor(simulated$duplicate1, simulated$V1)

set.seed(200)
model2 <- randomForest(y ~ ., data = simulated,
                       importance = TRUE,
                       ntree = 1000)
rfImp2 <- varImp(model2, scale = FALSE)
#rfImp2 <- data.frame(rfImp2[0][order(rfImp2, decreasing = TRUE),], rfImp2[order(rfImp2, decreasing = T
#colnames(rfImp2) <- 'Importance Score'
d2 <- rfImp2[ order(-rfImp2), , drop=FALSE ]
kable(d2)

# (8.1c)
set.seed(200)
# Now remove correlated predictor
simulated$duplicate1 <- NULL
bagCtrl <- cforest_control(mtry = ncol(simulated) - 1)
```

```r
baggedTree <- cforest(y ~ ., data = simulated, controls = bagCtrl)

cfImp <- varimp(baggedTree, conditional = T)
#cfImp <- kable(sort(cfImp, decreasing = TRUE))

cfImp1 <- varimp(baggedTree, conditional = F)
#cfImp1 <- kable(sort(cfImp1, decreasing = TRUE))

# Keep correlated predictor
simulated$duplicate1 <- simulated$V1 + rnorm(200) * .1
bagCtrl <- cforest_control(mtry = ncol(simulated) - 1)
baggedTree <- cforest(y ~ ., data = simulated, controls = bagCtrl)

cfImp2 <- varimp(baggedTree, conditional = T)
#cfImp2 <- kable(sort(cfImp2, decreasing = TRUE))

cfImp22 <- varimp(baggedTree, conditional = F)
#cfImp22 <- kable(sort(cfImp22, decreasing = TRUE))

simulated$duplicate1 <- NULL

a <- data.frame(features = rownames(rfImp1), RF = rfImp1[,1])
b <- data.frame(features = rownames(rfImp2), RF.cor = rfImp2[,1])
c <- data.frame(features = names(cfImp), CF.cond = cfImp)
d <- data.frame(features = names(cfImp1), CF = cfImp1)
e <- data.frame(features = names(cfImp2), CF.cor.cond = cfImp2)
f <- data.frame(features = names(cfImp22), CF.cor = cfImp22)

aa <- merge(a,d, all=T)
bb <- merge(b,f,all=T)
cc <- merge(c,e,all=T)
dd <-merge(aa,bb,all=T)

final_df <- merge(dd,cc,all=T)
final_df <- rbind(final_df[-3,], final_df[3,])
rownames(final_df) <- c(1:11)

kable(final_df)

# (8.1d)
set.seed(200)
#GBM
gbmModel <- gbm(y ~ ., data = simulated, distribution = "gaussian", n.trees=1000)
print("GBM")
summary(gbmModel)

simulated$duplicate1 <- simulated$V1 + rnorm(200) * .1
gbmModel <- gbm(y ~ ., data = simulated, distribution = "gaussian", n.trees=1000)
print("GBM - with duplicate1")
summary(gbmModel)

simulated$duplicate1 <- NULL
```

```r
#Cubist
cubistMod <- cubist(simulated[-11], simulated$y, committees = 100)
print("Cubist")
varImp(cubistMod)

simulated$duplicate1 <- simulated$V1 + rnorm(200) * .1
cubistMod <- cubist(simulated[-11], simulated$y, committees = 100)
print("Cubist - with duplicate1")
varImp(cubistMod)

simulated$duplicate1 <- NULL

# (8.2)
set.seed(200)
V1 <- rep(1:2,each=100)
V2 <- rnorm(200, mean=0, sd=2)
V3 <- rep(1:100,each=2)
V4 <- rnorm(200, mean=0, sd=3)
y <- (V1 + V2) + rnorm(200,mean=0,sd=4)

simulated_df <- data.frame(y,V1,V2,V3,V4)

bagCtrl <- cforest_control(mtry = ncol(simulated_df) - 1)
simulated_RF <-  cforest(y ~ ., data = simulated_df, controls = bagCtrl)

cor(simulated_df)
varImp(simulated_RF)

# (8.3a)
## Answers in text only

# (8.3b)
set.seed(200)
data(solubility)

sol_df <- data.frame(solTrainXtrans, solTrainY)
training <- sol_df$solTrainY %>%
  createDataPartition(p = 0.8, list = FALSE)

df_train  <- sol_df[training, ]
df_test <- sol_df[-training, ]

gbm1 <- gbm(solTrainY ~ ., data = df_train, distribution = "gaussian", n.trees = 100, bag.fraction = 0.

gbm10 <- gbm(solTrainY ~ ., data = df_train, distribution = "gaussian", n.trees = 100, bag.fraction = 0

gbm91 <- gbm(solTrainY ~ ., data = df_train, distribution = "gaussian", n.trees = 100, bag.fraction = 0

gbm910 <- gbm(solTrainY ~ ., data = df_train, distribution = "gaussian", n.trees = 100, bag.fraction =

p1 <- gbm1 %>% predict(df_test, n.trees = 100)
p2 <- gbm10 %>% predict(df_test, n.trees = 100)
p3 <- gbm91 %>% predict(df_test, n.trees = 100)
```

```r
p4 <- gbm910 %>% predict(df_test, n.trees = 100)

sum_t <- data.frame(
  RMSE_0.1_0.1 = caret::RMSE(p1, df_test$solTrainY),
  RMSE_0.9_0.1 = caret::RMSE(p2, df_test$solTrainY),
  RMSE_0.1_0.9 = caret::RMSE(p3, df_test$solTrainY),
  RMSE_0.9_0.9 = caret::RMSE(p4, df_test$solTrainY)
)
print(sum_t)

# (8.3c)
gbm1 <- gbm(solTrainY ~ ., data = df_train, distribution = "gaussian", n.trees = 100, interaction.depth

gbm10 <- gbm(solTrainY ~ ., data = df_train, distribution = "gaussian", n.trees = 100, interaction.deptl

#gbm91 <- gbm(solTrainY ~ ., data = df_train, distribution = "gaussian", n.trees = 100, interaction.dep

#gbm910 <- gbm(solTrainY ~ ., data = df_train, distribution = "gaussian", n.trees = 100, interaction.de

summary(gbm1)
summary(gbm10)
#summary(gbm91)
#summary(gbm910)

#p1 <- gbm1 %>% predict(df_test, n.trees = 100)
#p2 <- gbm10 %>% predict(df_test, n.trees = 100)
#p3 <- gbm91 %>% predict(df_test, n.trees = 100)
#p4 <- gbm910 %>% predict(df_test, n.trees = 100)

#sum_t <- data.frame(
#  RMSE_0.1_1 = caret::RMSE(p1, df_test$solTrainY),
#  RMSE_0.1_10 = caret::RMSE(p2, df_test$solTrainY),
#  RMSE_0.9_1 = caret::RMSE(p3, df_test$solTrainY),
#  RMSE_0.9_10 = caret::RMSE(p4, df_test$solTrainY)
#)
#print(sum_t)

# (8.7a)
# save df
data("ChemicalManufacturingProcess")
df <- ChemicalManufacturingProcess

# set seed for split to allow for reproducibility
set.seed(200)
# use mice w/ default settings to impute missing data
miceImput <- mice(df, printFlag = FALSE)

# add imputed data to original data set
df_mice <- complete(miceImput)

# Look for any features with no variance:
zero_cols <- nearZeroVar( df_mice )
df_final <- df_mice[,-zero_cols] # drop these zero variance columns
```

```r
#df_final <- df_mice

# split data train/test
training <- df_final$Yield %>%
  createDataPartition(p = 0.8, list = FALSE)
df_train  <- df_final[training, ]
df_test <- df_final[-training, ]

# model1 - RF
# Algorithm Tune (tuneRF)
set.seed(200)
bestmtry <- tuneRF(df_train[,-1], df_train[,1], stepFactor=1.5, improve=1e-5, ntree=2500)
##mtry <- ( (ncol(df_train) -1) / 3 ) or sqrt(ncol(df_train) - 1) # By default, # of predictors / 3 for

# from above result, we got mtry= 27 and ntree=2500 as optimal parameters
rf <- randomForest(Yield~., data=df_train, method="rf", mtry= 27, importance = TRUE, ntree = 2500)

# model2 - GBM
set.seed(200)
Control <- trainControl(method="repeatedcv", number=5, repeats=2)

gbmGrid <- expand.grid(
  n.trees=c(1000, 1500, 2000, 2500),
  interaction.depth=seq(1, 10, by = 2),
  shrinkage = c(0.01, 0.1),
  n.minobsinnode=c(5,10) )

gbmModel <- caret::train(Yield~., data=df_train,
                         method = 'gbm',
                         trControl = Control,
                         tuneGrid=gbmGrid,
                         tuneLength = 5,
                         verbose = FALSE)

# model3 - Cubist
cubist <- caret::train(Yield~., data=df_train, method = "cubist")

# Make predictions
p1 <- rf %>% predict(df_test)
p2 <- gbmModel %>% predict(df_test)
p3 <- cubist %>% predict(df_test)

# Model performance metrics
sum_t <- data.frame(
  RMSE1 = caret::RMSE(p1, df_test$Yield),
  RMSE2 = caret::RMSE(p2, df_test$Yield),
  RMSE3 = caret::RMSE(p3, df_test$Yield),
  Rsquare1 = caret::R2(p1, df_test$Yield),
  Rsquare2 = caret::R2(p2, df_test$Yield),
  Rsquare3 = caret::R2(p3, df_test$Yield)
)
print(sum_t)
```

```r
# (8.7b)
#code
t < - varImp(gbmModel)
plot(t)

# (8.7c)
rpartGrid <- expand.grid(maxdepth= seq(5,30,by=1))
ctrl <- trainControl(method = "boot", number = 25)

rpartChemTune <- caret::train(Yield~.,
                        data = df_train,
                        method = "rpart2",
                        metric = "Rsquared",
                        tuneGrid = rpartGrid,
                        trControl = ctrl)

plot(as.party(rpartChemTune$finalModel),gp=gpar(fontsize=11))
```