DATA 624: HOMEWORK 2

Kann & Johnson Problems: 6.3, 7.2, 7.5,8.1, 8.2, 8.3, 8.7

Angrand, Burke, Deboch, Groysman, Karr

Contents

Data 624: Week 8 Homework

Angrand, Burke, Deboch, Groysman, Karr

November 29, 2019

Week 8 Assignment

```
library(AppliedPredictiveModeling)
library(caret)
library(dplyr)
library(RANN)
library(knitr)
```

Exercise 6.3

6.3. A chemical manufacturing process for a pharmaceutical product was discussed in Sect. 1.4. In this problem, the objective is to understand the relationship between biological measurements of the raw materials (predictors), measurements of the manufacturing process (predictors), and the response of product yield. Biological predictors cannot be changed but can be used to assess the quality of the raw material before processing. On the other hand, manufacturing process predictors can be changed in the manufacturing process. Improving product yield by 1% will boost revenue by approximately one hundred thousand dollars per batch:

a. Start R and use these commands to load the data.

```
data(ChemicalManufacturingProcess)
```

The matrix processPredictors contains the 57 predictors (12 describing the input biological material and 45 describing the process predictors) for the 176 manufacturing runs. yield contains the percent yield for each run.

b. A small percentage of cells in the predictor set contain missing values. Use an imputation function to fill in these missing values (e.g., see Sect. 3.8).

- Find missing values with sapply. THe total dataframe only contains 175=6 rows and there are quite a few columns that are missing over 5% of their values. The values need to be imputted not removed.
- The mentioned section 3.8 highlights the impute.knn function from the impute library and the preprocess function from the caret library. The impute.knn function uses K-nearest neighbors to estimate the missing data and can be called as a subcomponent in the preprocess function.

- After calling the prerpocess function, the predict method applies the results to the set of data
- Check to see if all nulls have been removed with sapply

```
#dim
dim(ChemicalManufacturingProcess)

## [1] 176  58

#check for NaNs
sapply(ChemicalManufacturingProcess, function(x) sum(is.na(x)))

##                   Yield   BiologicalMaterial01   BiologicalMaterial02
##                       0                      0                      0
##   BiologicalMaterial03   BiologicalMaterial04   BiologicalMaterial05
##                       0                      0                      0
##   BiologicalMaterial06   BiologicalMaterial07   BiologicalMaterial08
##                       0                      0                      0
##   BiologicalMaterial09   BiologicalMaterial10   BiologicalMaterial11
##                       0                      0                      0
##   BiologicalMaterial12 ManufacturingProcess01 ManufacturingProcess02
##                       0                      1                      3
## ManufacturingProcess03 ManufacturingProcess04 ManufacturingProcess05
##                      15                      1                      1
## ManufacturingProcess06 ManufacturingProcess07 ManufacturingProcess08
##                       2                      1                      1
## ManufacturingProcess09 ManufacturingProcess10 ManufacturingProcess11
##                       0                      9                     10
## ManufacturingProcess12 ManufacturingProcess13 ManufacturingProcess14
##                       1                      0                      1
## ManufacturingProcess15 ManufacturingProcess16 ManufacturingProcess17
##                       0                      0                      0
## ManufacturingProcess18 ManufacturingProcess19 ManufacturingProcess20
##                       0                      0                      0
## ManufacturingProcess21 ManufacturingProcess22 ManufacturingProcess23
##                       0                      1                      1
## ManufacturingProcess24 ManufacturingProcess25 ManufacturingProcess26
##                       1                      5                      5
## ManufacturingProcess27 ManufacturingProcess28 ManufacturingProcess29
##                       5                      5                      5
## ManufacturingProcess30 ManufacturingProcess31 ManufacturingProcess32
##                       5                      5                      0
## ManufacturingProcess33 ManufacturingProcess34 ManufacturingProcess35
##                       5                      5                      5
## ManufacturingProcess36 ManufacturingProcess37 ManufacturingProcess38
##                       5                      0                      0
## ManufacturingProcess39 ManufacturingProcess40 ManufacturingProcess41
##                       0                      1                      1
## ManufacturingProcess42 ManufacturingProcess43 ManufacturingProcess44
##                       0                      0                      0
```

```
## ManufacturingProcess45
##                       0

#impute with preProcess, apply with predict
impute <- preProcess(as.matrix(ChemicalManufacturingProcess), method=c("knnIm
pute"))
impute.chem <-  as.data.frame(predict(impute, as.matrix(ChemicalManufacturing
Process)))
#check again for nulls after applying
sapply(impute.chem, function(x) sum(is.na(x)))

##                  Yield   BiologicalMaterial01   BiologicalMaterial02
##                      0                      0                      0
##   BiologicalMaterial03   BiologicalMaterial04   BiologicalMaterial05
##                      0                      0                      0
##   BiologicalMaterial06   BiologicalMaterial07   BiologicalMaterial08
##                      0                      0                      0
##   BiologicalMaterial09   BiologicalMaterial10   BiologicalMaterial11
##                      0                      0                      0
##   BiologicalMaterial12 ManufacturingProcess01 ManufacturingProcess02
##                      0                      0                      0
## ManufacturingProcess03 ManufacturingProcess04 ManufacturingProcess05
##                      0                      0                      0
## ManufacturingProcess06 ManufacturingProcess07 ManufacturingProcess08
##                      0                      0                      0
## ManufacturingProcess09 ManufacturingProcess10 ManufacturingProcess11
##                      0                      0                      0
## ManufacturingProcess12 ManufacturingProcess13 ManufacturingProcess14
##                      0                      0                      0
## ManufacturingProcess15 ManufacturingProcess16 ManufacturingProcess17
##                      0                      0                      0
## ManufacturingProcess18 ManufacturingProcess19 ManufacturingProcess20
##                      0                      0                      0
## ManufacturingProcess21 ManufacturingProcess22 ManufacturingProcess23
##                      0                      0                      0
## ManufacturingProcess24 ManufacturingProcess25 ManufacturingProcess26
##                      0                      0                      0
## ManufacturingProcess27 ManufacturingProcess28 ManufacturingProcess29
##                      0                      0                      0
## ManufacturingProcess30 ManufacturingProcess31 ManufacturingProcess32
##                      0                      0                      0
## ManufacturingProcess33 ManufacturingProcess34 ManufacturingProcess35
##                      0                      0                      0
## ManufacturingProcess36 ManufacturingProcess37 ManufacturingProcess38
##                      0                      0                      0
## ManufacturingProcess39 ManufacturingProcess40 ManufacturingProcess41
##                      0                      0                      0
## ManufacturingProcess42 ManufacturingProcess43 ManufacturingProcess44
##                      0                      0                      0
```

```
## ManufacturingProcess45
##                      0
```

c. Split the data into a training and a test set, pre-process the data, and tune a model of your choice from this chapter. What is the optimal value of the performance metric?

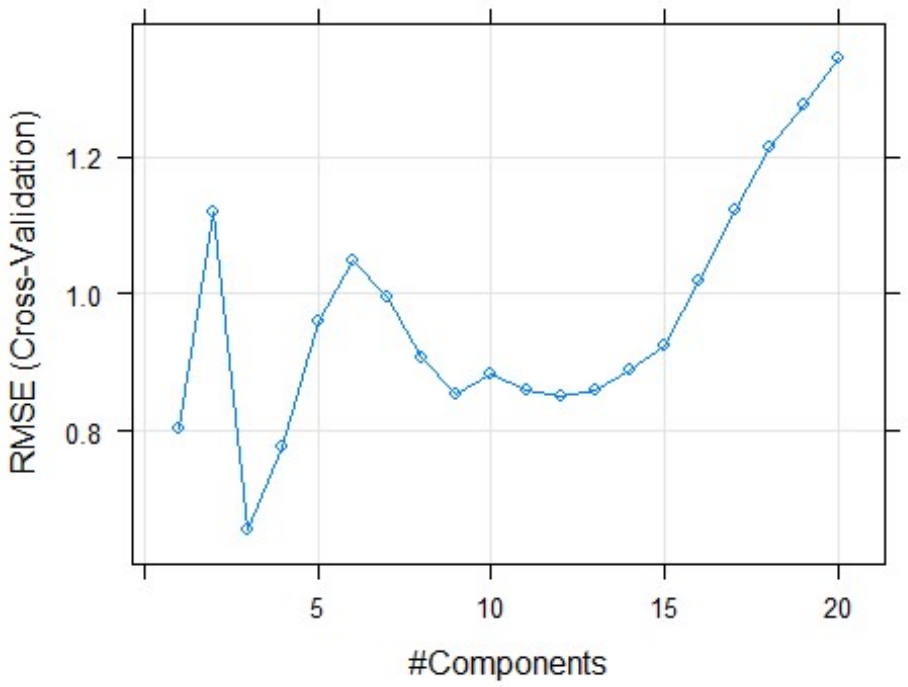
- Yield c(1) is the response of the other columns (predictors)
- Prepare the data: split the data into train/test samples. Train (75% for building a predictive model) and Test (15% for evaluating the model)
- Use partial least squares method with a tested 20 different values for the tuning parameter ncomp
- As seen below, the most optimal value is ncomp = 3 with the smallest RSME of 0.6554035 and a R^2 of 0.6096468

```
## set the seed to make the partition reproducible
set.seed(123)
train.chem <- createDataPartition(ChemicalManufacturingProcess$Yield, p=0.75,
list=FALSE)

#apply to the predictors
chem.Train <- impute.chem[train.chem,-1]
chem.Test <- impute.chem[-train.chem,-1]
#apply to yield
yield.Train <- impute.chem[train.chem,1]
yield.Test <- impute.chem[-train.chem,1]

#partial least squares w/ train data

pls.chem <- train(chem.Train, yield.Train,
                  method = "pls",
                  tuneLength = 20, trControl = trainControl(method = "cv", num
ber = 10),
                  preProc = c("center", "scale"))
#print outcomes of the pls
pls.chem
```

```
## Partial Least Squares
##
## 132 samples
##  57 predictor
##
## Pre-processing: centered (57), scaled (57)
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 119, 118, 120, 120, 118, 118, ...
## Resampling results across tuning parameters:
##
##   ncomp  RMSE       Rsquared   MAE
##    1     0.8047230  0.4775122  0.6339437
##    2     1.1187475  0.4818566  0.6862102
##    3     0.6554035  0.6096468  0.5318995
```

```
##     4      0.7774620  0.5555155  0.5816168
##     5      0.9598662  0.4876206  0.6366717
##     6      1.0485623  0.4743592  0.6653206
##     7      0.9953023  0.4820678  0.6540929
##     8      0.9072158  0.5017679  0.6331196
##     9      0.8528902  0.5063925  0.6226068
##    10      0.8838340  0.4967802  0.6336124
##    11      0.8606428  0.4919373  0.6392547
##    12      0.8513129  0.4905676  0.6490674
##    13      0.8600490  0.4801921  0.6694921
##    14      0.8886411  0.4663450  0.6848575
##    15      0.9253709  0.4541527  0.7022949
##    16      1.0199562  0.4323120  0.7366779
##    17      1.1213147  0.4124365  0.7769600
##    18      1.2121685  0.4016808  0.8083622
##    19      1.2764454  0.3933288  0.8276471
##    20      1.3445415  0.3906539  0.8448760
##
## RMSE was used to select the optimal model using the smallest value.
## The final value used for the model was ncomp = 3.
```

```
# Plot model RMSE vs different values of components
plot(pls.chem)
```




```
# Print the best tuning parameter ncomp that
# minimize the cross-validation error, RMSE
pls.chem$bestTune
```

```
##   ncomp
## 3     3
```

```
# Summarize the final model
summary(pls.chem$finalModel)
```

```
## Data:    X dimension: 132 57
##  Y dimension: 132 1
## Fit method: oscorespls
## Number of components considered: 3
## TRAINING: % variance explained
##           1 comps  2 comps  3 comps
## X           17.17    25.62    31.08
## .outcome    52.84    67.27    72.34
```

d. Predict the response for the test set. What is the value of the performance metric and how does this compare with the resampled performance metric on the training set?

- Make predictions with the predict() function with the inputted chem.test data
- Compare the predicted values to the actual valyes "yield.Test"
- The RMSE is very close to the train data RSME. The Rsquare value is lower than the train values.

```
# Make predictions

predictions <- predict(pls.chem, newdata = chem.Test)

data.frame(
  RMSE = caret::RMSE(predictions, yield.Test),
  Rsquare = caret::R2(predictions, yield.Test)
)
```

```
##       RMSE   Rsquare
## 1 0.694739 0.4295074
```

e. Which predictors are most important in the model you have trained? Do either the biological or process predictors dominate the list?

- Find the absolute value from the mean contributions for each coefficient
- ManufacturingProcess32, ManufacturingProcess13,ManufacturingProcess17 & ManufacturingProcess09 appear to be the most significant by a good margin.
- In general, the manufacturing process variables appear to be more significant than any other grouping of variables

```
predictors.pls <- as.data.frame(pls.chem$finalModel$coefficients)
predictors.pls<- tibble::rownames_to_column(predictors.pls, "coefficients")
predictors.pls%>%
  mutate(meancol= rowMeans(.[, 2:4]))%>%
  mutate(absmeancol =abs(meancol))%>%
  arrange(-absmeancol)
```

```
##              coefficients .outcome.1 comps .outcome.2 comps
## 1  ManufacturingProcess32     0.0712238757     1.109397e-01
## 2  ManufacturingProcess13    -0.0661158030    -1.395337e-01
## 3  ManufacturingProcess17    -0.0577815025    -1.395057e-01
## 4  ManufacturingProcess09     0.0654051170     1.303278e-01
## 5  ManufacturingProcess36    -0.0629479000    -9.425630e-02
## 6  ManufacturingProcess11     0.0507167348     9.468503e-02
## 7  ManufacturingProcess12     0.0470604710     8.639382e-02
## 8  ManufacturingProcess06     0.0485912907     8.009020e-02
## 9  ManufacturingProcess33     0.0514050466     6.220852e-02
## 10 ManufacturingProcess37    -0.0291606515    -6.601438e-02
## 11 ManufacturingProcess34     0.0178812807     6.052770e-02
## 12 ManufacturingProcess15     0.0324619266     3.801886e-02
## 13 ManufacturingProcess30     0.0324426471     5.579544e-02
## 14   BiologicalMaterial06     0.0584312750     4.495394e-02
## 15   BiologicalMaterial03     0.0539347628     4.805170e-02
## 16   BiologicalMaterial02     0.0596292461     4.351645e-02
## 17 ManufacturingProcess24    -0.0279548342    -3.764810e-02
## 18   BiologicalMaterial07    -0.0106260492    -3.533573e-02
## 19 ManufacturingProcess21    -0.0074744865    -4.431938e-02
## 20 ManufacturingProcess39     0.0060988025     3.620146e-02
## 21 ManufacturingProcess44     0.0094662555     3.807731e-02
## 22 ManufacturingProcess10     0.0291995589     4.993165e-02
## 23 ManufacturingProcess43     0.0233307462     3.665961e-02
## 24   BiologicalMaterial08     0.0518001888     2.836059e-02
## 25   BiologicalMaterial12     0.0490325444     2.972720e-02
## 26   BiologicalMaterial04     0.0468070169     2.374136e-02
## 27 ManufacturingProcess45     0.0013979169     2.482118e-02
## 28   BiologicalMaterial11     0.0463832457     2.363290e-02
## 29   BiologicalMaterial05     0.0210190639     1.644061e-02
## 30 ManufacturingProcess08     0.0095328221     2.820146e-02
## 31   BiologicalMaterial01     0.0430725927     1.508968e-02
## 32 ManufacturingProcess22     0.0072933469     2.560602e-02
## 33 ManufacturingProcess19     0.0219409221     3.318482e-03
## 34 ManufacturingProcess35    -0.0185679782    -2.204654e-02
## 35 ManufacturingProcess42    -0.0021437692     1.800772e-02
## 36 ManufacturingProcess29     0.0190983507     1.086223e-02
## 37 ManufacturingProcess31    -0.0084488434    -2.342044e-02
## 38 ManufacturingProcess41    -0.0055720930    -1.793995e-02
## 39 ManufacturingProcess40    -0.0067586547    -1.612848e-02
## 40 ManufacturingProcess28     0.0354945629     1.093106e-02
## 41 ManufacturingProcess20    -0.0071633382     8.146575e-03
## 42 ManufacturingProcess18    -0.0074149492     7.527277e-03
## 43 ManufacturingProcess05     0.0128651292    -7.162564e-03
## 44 ManufacturingProcess38    -0.0103386629    -8.397222e-03
## 45   BiologicalMaterial09     0.0151941295    -6.127552e-03
## 46 ManufacturingProcess16    -0.0032859018    -5.494742e-03
## 47 ManufacturingProcess25     0.0009501445    -1.361350e-02
## 48 ManufacturingProcess01    -0.0087233596     9.917543e-03
## 49 ManufacturingProcess27     0.0011863844    -1.229260e-02
```

```
## 50 ManufacturingProcess04    -0.0344634603    -1.484713e-02
## 51 ManufacturingProcess23    -0.0066616847     9.688644e-05
## 52 ManufacturingProcess14     0.0011316252    -1.252973e-02
## 53 ManufacturingProcess07     0.0027703386     4.052646e-03
## 54   BiologicalMaterial10     0.0293414934    -5.953754e-03
## 55 ManufacturingProcess26     0.0048294389    -6.316201e-03
## 56 ManufacturingProcess02    -0.0243838427     1.151700e-02
## 57 ManufacturingProcess03    -0.0025420455    -5.934078e-03
##    .outcome.3 comps       meancol   absmeancol
## 1       0.171668817  0.1179441344 0.1179441344
## 2      -0.141447945 -0.1156991341 0.1156991341
## 3      -0.149495753 -0.1155943036 0.1155943036
## 4       0.137650718  0.1111278698 0.1111278698
## 5      -0.134944299 -0.0973828316 0.0973828316
## 6       0.084715814  0.0767058595 0.0767058595
## 7       0.076668653  0.0700409813 0.0700409813
## 8       0.078469904  0.0690504646 0.0690504646
## 9       0.090165950  0.0679265063 0.0679265063
## 10     -0.095111378 -0.0634288041 0.0634288041
## 11      0.099304472  0.0592378170 0.0592378170
## 12      0.081504796  0.0506618599 0.0506618599
## 13      0.058618280  0.0489521214 0.0489521214
## 14      0.041960341  0.0484485186 0.0484485186
## 15      0.042659663  0.0482153756 0.0482153756
## 16      0.038507706  0.0472177999 0.0472177999
## 17     -0.054699035 -0.0401006549 0.0401006549
## 18     -0.073436749 -0.0397995086 0.0397995086
## 19     -0.058054979 -0.0366162835 0.0366162835
## 20      0.067164453  0.0364882386 0.0364882386
## 21      0.058157601  0.0352337210 0.0352337210
## 22      0.023711632  0.0342809464 0.0342809464
## 23      0.040475152  0.0334885034 0.0334885034
## 24      0.010848688  0.0303364897 0.0303364897
## 25      0.011644640  0.0301347949 0.0301347949
## 26      0.018201184  0.0295831877 0.0295831877
## 27      0.053864928  0.0266946753 0.0266946753
## 28      0.007878601  0.0259649170 0.0259649170
## 29      0.035645329  0.0243683345 0.0243683345
## 30      0.033862102  0.0238654608 0.0238654608
## 31      0.012873518  0.0236785964 0.0236785964
## 32      0.035039319  0.0226462274 0.0226462274
## 33      0.041016579  0.0220919943 0.0220919943
## 34     -0.020080093 -0.0202315381 0.0202315381
## 35      0.042963320  0.0196090904 0.0196090904
## 36      0.027711629  0.0192240708 0.0192240708
## 37     -0.015868135 -0.0159124722 0.0159124722
## 38     -0.021370578 -0.0149608744 0.0149608744
## 39     -0.020313656 -0.0144002632 0.0144002632
## 40     -0.010835085  0.0118635134 0.0118635134
## 41      0.033078216  0.0113538174 0.0113538174
```

```
## 42      0.033395149  0.0111691589 0.0111691589
## 43     -0.033968050 -0.0094218283 0.0094218283
## 44     -0.009295092 -0.0093436589 0.0093436589
## 45     -0.036104949 -0.0090127906 0.0090127906
## 46     -0.014100772 -0.0076271384 0.0076271384
## 47     -0.005385365 -0.0060162385 0.0060162385
## 48      0.015380878  0.0055250205 0.0055250205
## 49     -0.004295679 -0.0051339650 0.0051339650
## 50      0.034398893 -0.0049705663 0.0049705663
## 51     -0.007221722 -0.0045955066 0.0045955066
## 52      0.024393990  0.0043319626 0.0043319626
## 53     -0.013624735 -0.0022672502 0.0022672502
## 54     -0.019834614  0.0011843750 0.0011843750
## 55      0.003344658  0.0006192985 0.0006192985
## 56      0.011073945 -0.0005976317 0.0005976317
## 57      0.009477843  0.0003339065 0.0003339065
```

f. Explore the relationships between each of the top predictors and the response. How could this information be helpful in improving yield in future runs of the manufacturing process?

- For the manufacturing processes with negative coefficients, the facility could alter their processes to decrease the associated impact to yields
- For the manufacturing processes with positibe coefficients, the facility to could their processes to increase the associasted impact to yields
- Given that Biological materials do not have a significant impact, the facility could alter the ingrediants/materials to increase the associated yields

Data 624: Week 9 Homework

Angrand, Burke, Deboch, Groysman, Karr

December 8, 2019

Week 9 Assignment

Chapter 7 KJ 7.2, 7.5

```
library(AppliedPredictiveModeling)
library(caret)
library(mlbench)
library(tidyverse)
library(pracma)

library(tidyverse)

library(gridExtra)
library(ggcorrplot)
```

Exercise 7.2

7.2 Friedman (1991) introduced several benchmark data sets created by simulation. One of these simulations used the following nonlinear equation to create data:

$$y = 10\sin(\pi x_1 x_2) + 20(x_3 - 0.5)^2 + 10x_4 + 5x_5 + N(0, \sigma^2)$$

where the x values are random variables uniformly distributed between [0,1] (there are also 5 other non-informative variables created in the simulation). The package *mlbench* contains a function called mlbench.friedman1 that simulates these data:

Read Data & EDA

a. Creating Training and Testing Data

```
set.seed(100)
trainingData <- mlbench.friedman1(200, sd = 1)
## We convert th 'x' data from a matrix to data frame
## One reason is that this will give the columns names.
trainingData$x <- data.frame(trainingData$x)



## This creates a list with a vector 'y' and a matrix
## of predictors 'x'.  Also simulate a large test set to
## estimate the truee error rate with good precisions:
testData <- mlbench.friedman1(5000, sd = 1)
testData$x <- data.frame(testData$x)
```

The relationship between the predictors x1-x10 and the response y

```
head(trainingData$x)
```

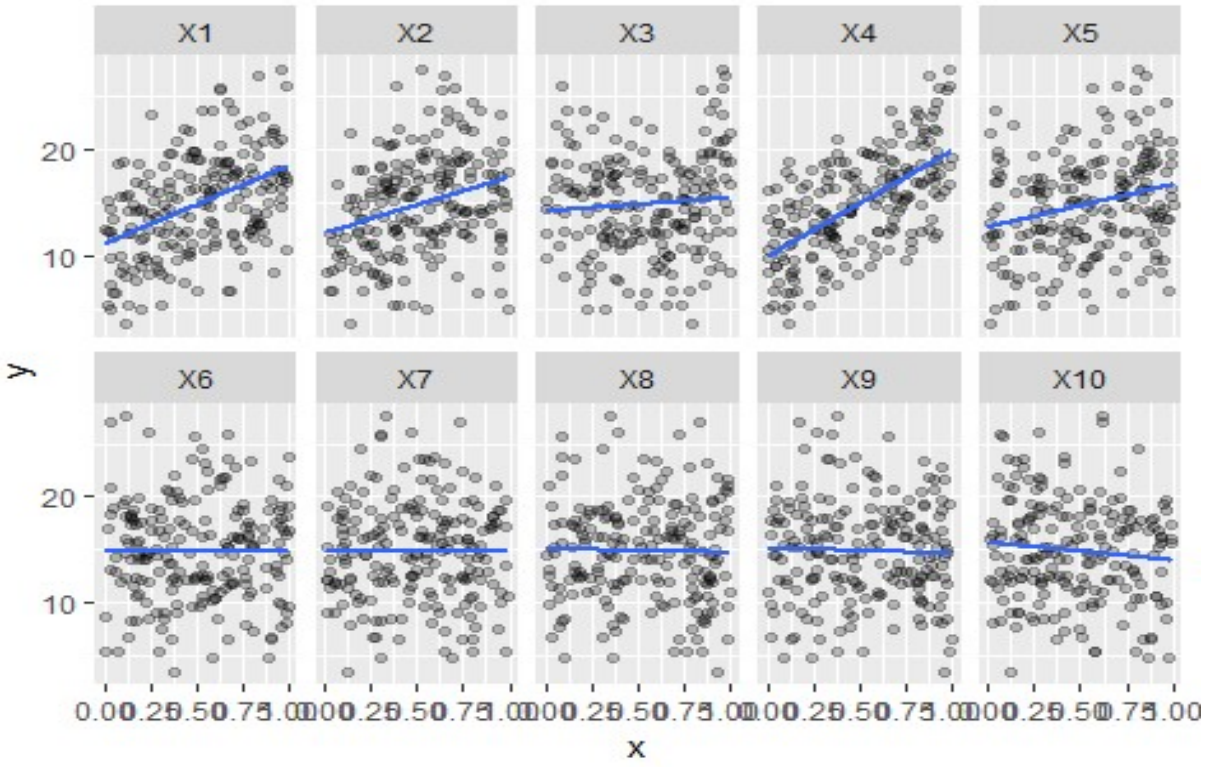
```
##           X1        X2        X3         X4         X5        X6        X7
## 1 0.30776611 0.3695961 0.5112374 0.03176634 0.09942609 0.0740483 0.9734581
## 2 0.25767250 0.9563228 0.2777107 0.57970549 0.22993408 0.1118664 0.7717425
## 3 0.55232243 0.9135767 0.3606569 0.15420484 0.44362621 0.6239440 0.4949800
## 4 0.05638315 0.8233363 0.4375279 0.12527050 0.51570490 0.6710818 0.3926061
## 5 0.46854928 0.3194822 0.8030667 0.14798581 0.92489425 0.3658942 0.1053059
## 6 0.48377074 0.8777003 0.5206097 0.91334263 0.04445684 0.1831814 0.3560706
##          X8        X9       X10
## 1 0.6939725 0.3761842 0.5704419
## 2 0.2022793 0.2294918 0.4786617
## 3 0.5222016 0.9693079 0.3072272
## 4 0.8133963 0.1315974 0.8855426
## 5 0.9161582 0.6090285 0.5418244
## 6 0.9463884 0.2840458 0.4221332
```

```
head(testData$x)
```

```
##           X1         X2        X3        X4         X5         X6
## 1 0.9511717 0.84353248 0.8396137 0.3112072 0.02490033 0.42544544
## 2 0.9223041 0.58380180 0.5400547 0.6214827 0.29711901 0.02439103
## 3 0.3012870 0.85267526 0.3482427 0.8339510 0.60000986 0.79394431
## 4 0.5742679 0.42253889 0.9993041 0.2476972 0.96193464 0.99934132
## 5 0.7283813 0.67991180 0.7148379 0.7698435 0.68527210 0.38128356
## 6 0.5430245 0.07116613 0.4131925 0.7132922 0.01547487 0.93686149
##           X7        X8        X9        X10
## 1 0.27018771 0.9954754 0.7930478 0.53310722
## 2 0.53027543 0.3503692 0.9809181 0.67506119
## 3 0.03388816 0.4602326 0.2053274 0.58461319
## 4 0.01742161 0.1520394 0.8638043 0.06242131
## 5 0.69713633 0.8956982 0.6678364 0.09462883
## 6 0.89782147 0.5493450 0.7149965 0.72507148
```

b. Determine correlation between x and y

```
## Look at the data using featurePlot
## or other methods.

trainingData$x %>%
  #gather x and y
  mutate(y=trainingData$y) %>%
  # tidy data frame for easier manipulating & plotting
  gather(var, x,-y) %>%
  #factor x variable and change factors so X10 is last
  mutate(var= forcats::fct_relevel(factor(var), "X10", after=Inf)) %>%
  ggplot(aes(x,y)) +
  geom_point(alpha=0.25) +
  stat_smooth(method="glm", se = FALSE) +
  facet_wrap(~ var, nrow = 2)
```



Training Models

Tune several models on these data. For example:

a. K-Nearest Neighboor Model (KNN)

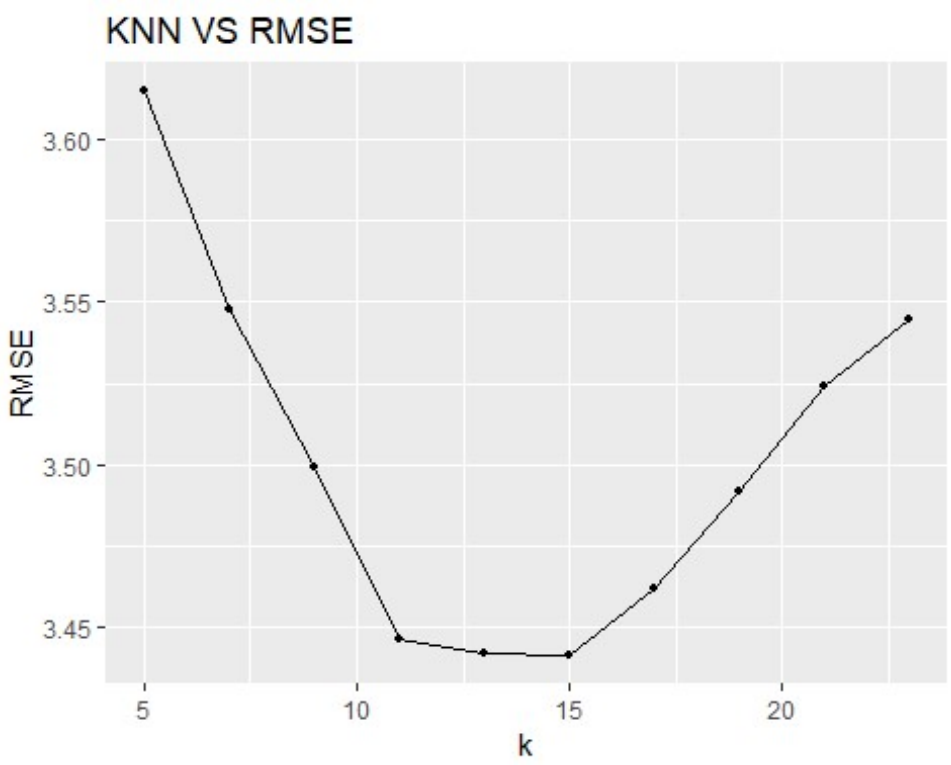
```
library(caret)
set.seed(921)
knnModel <- train(x = trainingData$x,
                  y = trainingData$y,
                  method = "knn",
                  preProc = c("center","scale"),
                  tuneLength = 10)
knnModel
```

```
## k-Nearest Neighbors
##
## 200 samples
##  10 predictor
##
## Pre-processing: centered (10), scaled (10)
## Resampling: Bootstrapped (25 reps)
## Summary of sample sizes: 200, 200, 200, 200, 200, 200, ...
## Resampling results across tuning parameters:
##
##   k   RMSE      Rsquared   MAE
##    5  3.614818  0.4443500  2.959250
##    7  3.547913  0.4721382  2.918330
```

```
##    9  3.499002  0.4998668  2.879042
##   11  3.446309  0.5310551  2.825429
##   13  3.441987  0.5462941  2.822529
##   15  3.441374  0.5645635  2.815643
##   17  3.462089  0.5718349  2.824125
##   19  3.491635  0.5728072  2.843148
##   21  3.524142  0.5697454  2.879433
##   23  3.544471  0.5755214  2.894378
##
## RMSE was used to select the optimal model using the smallest value.
## The final value used for the model was k = 15.
```

RMSE with varying tuning parameters

```
knnModel$results %>%
  ggplot(aes(x=k, y=RMSE)) +
  geom_line() + geom_point(size=1) +
  labs(title="KNN VS RMSE")
```



- k=15 is the optimal model

```
knnPred <- predict(knnModel, newdata = testData$x)

## The function 'postResample' can be used to get test set
## performance values
knn.pred <- postResample(pred = knnPred, obs = testData$y)
knn.pred
```

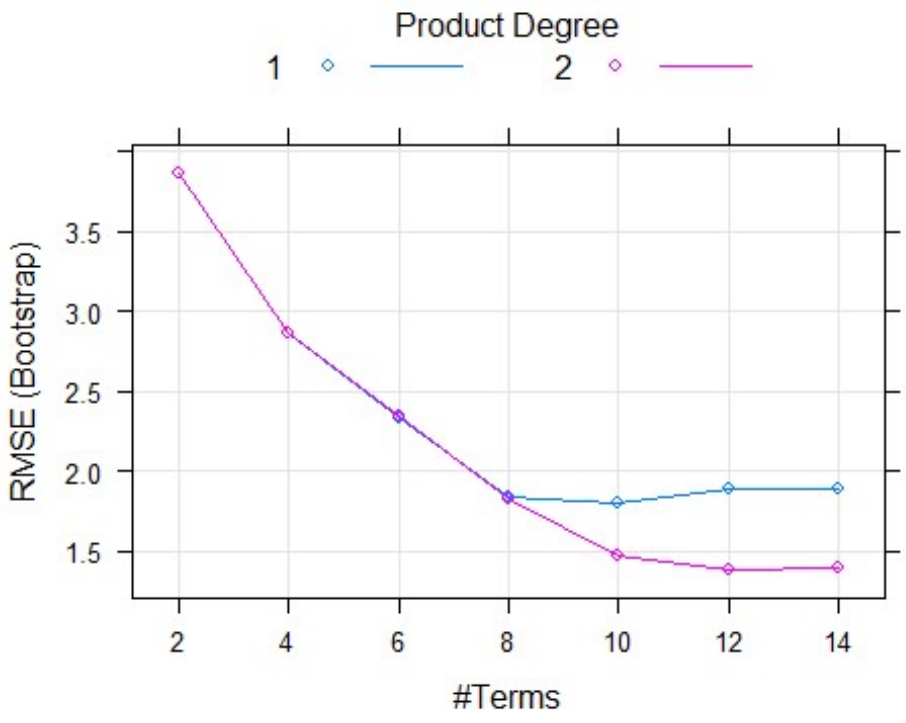
```
##      RMSE  Rsquared       MAE
## 3.3709432 0.6630201 2.7279373
```

Which model appears to get the best performance? Does MARS select the informative predictors (those named X1-15)

K-nearest neighbors models perform better when predictor and response relationships have a locational dependency. The simulation data is not related in this way so other models are expected perform better. In fact MARS and SVM have lower RMSE values and thus a better fit.

b. MARS Model

```
marsGrid <- expand.grid(degree = 1:2, nprune = seq(2,14,by=2))
set.seed(921)
marsModel <- train(x = trainingData$x,
                   y = trainingData$y,
                   method = "earth",
                   preProc = c("center","scale"),
                   tuneGrid = marsGrid)

marsPred <- predict(marsModel, newdata = testData$x)
plot(marsModel)
```




```
mars.pred <- postResample(pred = marsPred, obs = testData$y)
mars.pred
```

```
##      RMSE  Rsquared       MAE
## 1.1772309 0.9430908 0.9386423
```

- The MARS model is the optimal one of those tested with the lowest RMSE or fit. The optimal RMSE is achieved with a second-degree. We can further investigate variable importance and see that only the top 5 predictors have significant influence on the response variable with the following ranking . . . V4, V1, V3, V5, V3.

```
varImp(marsModel)
```

```
## earth variable importance
##
##     Overall
## X4   100.00
## X1    78.53
## X2    67.42
## X5    52.24
## X3    40.90
## X7     0.00
## X10    0.00
## X8     0.00
## X9     0.00
## X6     0.00
```

- A summary model can also be generated using the earth function

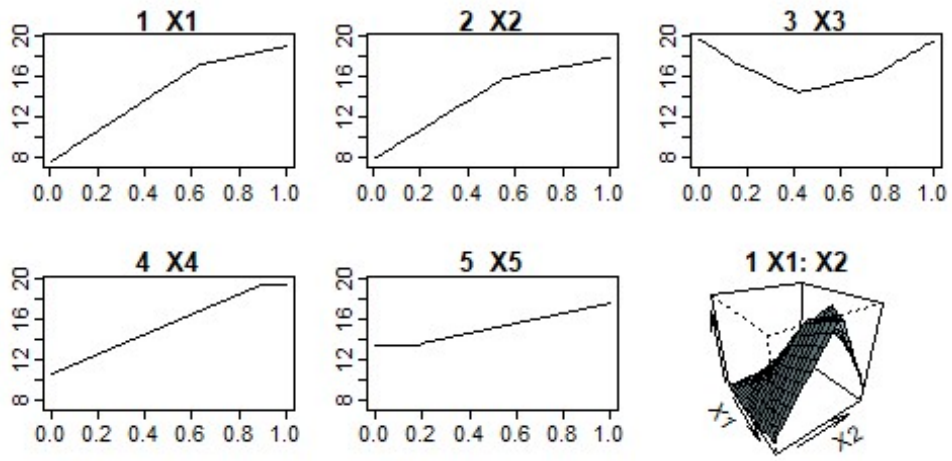
```
marsFit <- earth(x = trainingData$x,
                 y = trainingData$y,
                 nprune = 12, degree = 2)
summary(marsFit)
```

```
## Call: earth(x=trainingData$x, y=trainingData$y, degree=2, nprune=12)
##
##                                 coefficients
## (Intercept)                        18.273216
## h(0.629995-X1)                    -16.509679
## h(X1-0.629995)                      4.989265
## h(0.55336-X2)                     -19.474961
## h(X2-0.55336)                       6.462382
## h(X3-0.162376)                      4.970899
## h(0.422184-X3)                     15.463906
## h(X3-0.741199)                      8.489787
## h(0.902811-X4)                     -9.745742
## h(X5-0.161512)                      5.006716
## h(X1-0.468549) * h(X2-0.55336)    -55.417177
## h(0.673249-X1) * h(0.55336-X2)     28.647676
##
## Selected 12 of 19 terms, and 5 of 10 predictors
## Termination condition: Reached nk 21
## Importance: X4, X1, X2, X5, X3, X6-unused, X7-unused, X8-unused, ...
## Number of terms at each degree of interaction: 1 9 2
## GCV 1.381629    RSS 203.1841    GRSq 0.940204    RSq 0.9555886
```

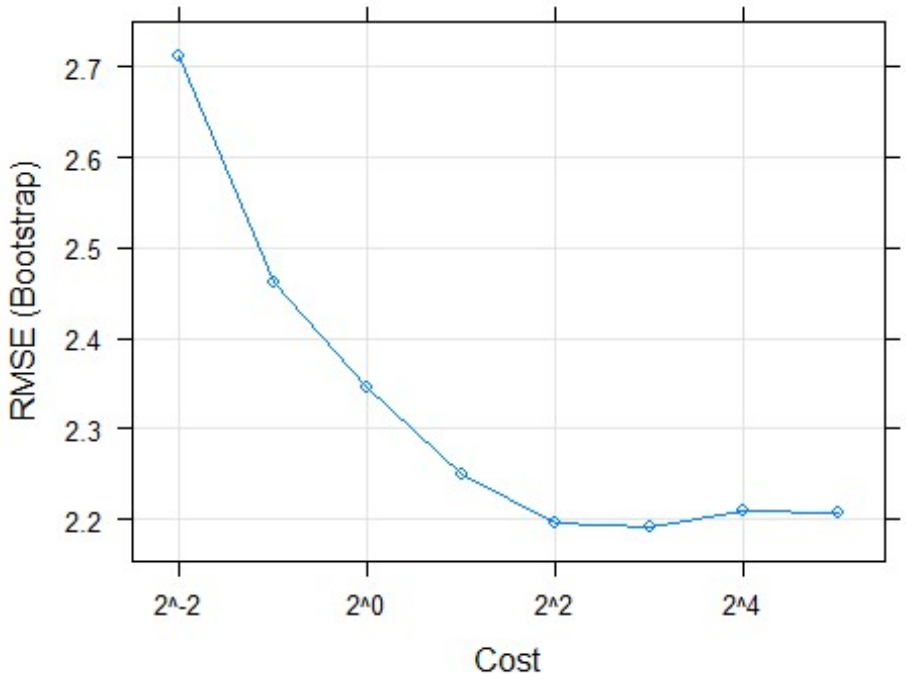
c. SVM Model

```
plotmo(marsFit, caption = "")
```

```
##  plotmo grid:    X1        X2        X3        X4        X5        X6
##             0.5011993 0.5111177 0.5632154 0.4727849 0.5352105 0.5118565
##          X7        X8       X9       X10
##  0.5070687 0.5548462 0.509134 0.4232667
```




```
set.seed(921)
svmRModel <- train(x = trainingData$x,
                   y = trainingData$y,
                   method = "svmRadial",
                   preProc = c("center","scale"),
                   tuneLength = 8)
svmRPred <- predict(svmRModel, newdata = testData$x)

svm.pred <- postResample(pred = svmRPred, obs = testData$y)
plot(svmRModel, scales = list(x = list(log = 2)))
```



- The Cost to RMSE(Bootstrap) plot shows the SVM tuning parameter profile. The optimal model has a cost value of 16 and an RMSE of ~2.0%

```
rbind(knn.pred, mars.pred, svm.pred)

##                RMSE  Rsquared       MAE
## knn.pred  3.370943 0.6630201 2.7279373
## mars.pred 1.177231 0.9430908 0.9386423
## svm.pred  1.962450 0.8418429 1.5185282
```

- Overall, the MARS model performs best, the radial basis function SVM coming in next and K-NN has the worst performance for this problem.

Exercise 7.5

```
7.5 Exercise 6.3 describes data for a chemical manufacturing process.  Use th
e smae data imputation, data splitting and pre-processing steps as before and
train several nonlinear regressions models.

a) Which nonlinear regression model gives the optimal resampling and test set
performance?

b) Which predictors are most important in the optimal nonlinear regression mo
del?
   Do either the biological or process variables dominate the list?
   How do the top ten important predictors compare to the top ten predictors
from the optimal linear model?
```

c) Explore the relationships between the top predictors and the response for the predictors that are unique to the optimal nonlinear regression model. Do these plots reveal intuition about the bioglogical or process predictors and their relationship yield?

Data Pre-Work

a. Read in Data & EDA

```
set.seed(100)
data(ChemicalManufacturingProcess)

processPredictors = ChemicalManufacturingProcess[,2:58]
yield = ChemicalManufacturingProcess[,1]

n_samples = dim(processPredictors)[1]
n_features = dim(processPredictors)[2]
n_samples
```

```
## [1] 176
```

```
n_features
```

```
## [1] 57
```

b. Impute missing values

```
null.values <-as.data.frame(sapply(processPredictors, function(x) sum(is.na(x
))), col.names = "null_values")%>%
  tibble::rownames_to_column("Predictors")%>%
  rename(nulls = 2)%>%
  arrange(-nulls)%>%
  filter(nulls >0)
null.values
```

```
##                Predictors nulls
## 1  ManufacturingProcess03    15
## 2  ManufacturingProcess11    10
## 3  ManufacturingProcess10     9
## 4  ManufacturingProcess25     5
## 5  ManufacturingProcess26     5
## 6  ManufacturingProcess27     5
## 7  ManufacturingProcess28     5
## 8  ManufacturingProcess29     5
## 9  ManufacturingProcess30     5
## 10 ManufacturingProcess31     5
## 11 ManufacturingProcess33     5
## 12 ManufacturingProcess34     5
## 13 ManufacturingProcess35     5
## 14 ManufacturingProcess36     5
## 15 ManufacturingProcess02     3
## 16 ManufacturingProcess06     2
## 17 ManufacturingProcess01     1
```

```
## 18 ManufacturingProcess04     1
## 19 ManufacturingProcess05     1
## 20 ManufacturingProcess07     1
## 21 ManufacturingProcess08     1
## 22 ManufacturingProcess12     1
## 23 ManufacturingProcess14     1
## 24 ManufacturingProcess22     1
## 25 ManufacturingProcess23     1
## 26 ManufacturingProcess24     1
## 27 ManufacturingProcess40     1
## 28 ManufacturingProcess41     1
```

```
# Fill in missing values where we have NAs with the median over the non-NA va
lues:
replacements = sapply( processPredictors, median, na.rm=TRUE )
as.data.frame(replacements)
```

```
##                         replacements
## BiologicalMaterial01           6.305
## BiologicalMaterial02          55.090
## BiologicalMaterial03          67.220
## BiologicalMaterial04          12.100
## BiologicalMaterial05          18.490
## BiologicalMaterial06          48.460
## BiologicalMaterial07         100.000
## BiologicalMaterial08          17.510
## BiologicalMaterial09          12.835
## BiologicalMaterial10           2.710
## BiologicalMaterial11         146.080
## BiologicalMaterial12          20.120
## ManufacturingProcess01        11.400
## ManufacturingProcess02        21.000
## ManufacturingProcess03         1.540
## ManufacturingProcess04       934.000
## ManufacturingProcess05       999.200
## ManufacturingProcess06       206.800
## ManufacturingProcess07       177.000
## ManufacturingProcess08       178.000
## ManufacturingProcess09        45.730
## ManufacturingProcess10         9.100
## ManufacturingProcess11         9.400
## ManufacturingProcess12         0.000
## ManufacturingProcess13        34.600
## ManufacturingProcess14      4856.000
## ManufacturingProcess15      6031.500
## ManufacturingProcess16      4588.000
## ManufacturingProcess17        34.400
## ManufacturingProcess18      4835.000
## ManufacturingProcess19      6022.000
## ManufacturingProcess20      4582.000
```

```
## ManufacturingProcess21       -0.300
## ManufacturingProcess22        5.000
## ManufacturingProcess23        3.000
## ManufacturingProcess24        8.000
## ManufacturingProcess25     4855.000
## ManufacturingProcess26     6047.000
## ManufacturingProcess27     4587.000
## ManufacturingProcess28       10.400
## ManufacturingProcess29       19.900
## ManufacturingProcess30        9.100
## ManufacturingProcess31       70.800
## ManufacturingProcess32      158.000
## ManufacturingProcess33       64.000
## ManufacturingProcess34        2.500
## ManufacturingProcess35      495.000
## ManufacturingProcess36        0.020
## ManufacturingProcess37        1.000
## ManufacturingProcess38        3.000
## ManufacturingProcess39        7.200
## ManufacturingProcess40        0.000
## ManufacturingProcess41        0.000
## ManufacturingProcess42       11.600
## ManufacturingProcess43        0.800
## ManufacturingProcess44        1.900
## ManufacturingProcess45        2.200
```

```
for( ci in 1:n_features ){
  bad_inds = is.na( processPredictors[,ci] )
  processPredictors[bad_inds,ci] = replacements[ci]
}
```

c. No Variance Predictors Removal

```
# Look for any features with no variance:
zero_cols = nearZeroVar( processPredictors )
zero_cols
```

```
## [1] 7
```

```
processPredictors = processPredictors[,-zero_cols] # drop these zero variance
columns
```

d. Train/Test Split

```
# Split this data into training and testing sets:
# We set aside 20% of the observations to be the test dataset.
training = createDataPartition( yield, p=0.8 )

processPredictors_training = processPredictors[training$Resample1,]
yield_training = yield[training$Resample1]

processPredictors_testing = processPredictors[-training$Resample1,]
```

```
yield_testing = yield[-training$Resample1]
preProc_Arguments = c("center","scale")
```

Data Modeling

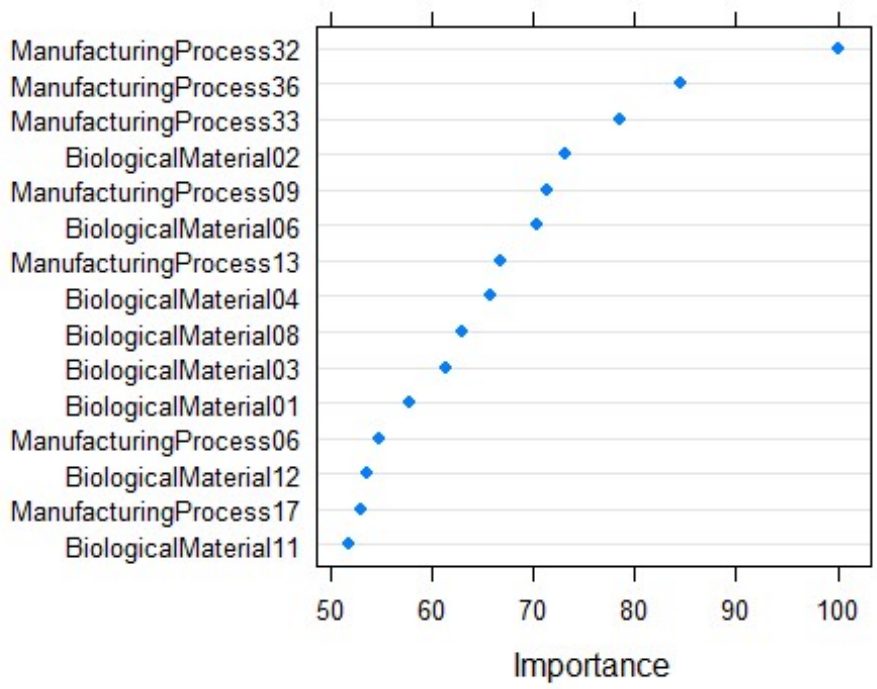
a. PLS

- adding thepls for comp. purposes

```
set.seed(100)
plsModel<-train(x=processPredictors_training, y=yield_training, method="pls",
tuneLength = 10,preProcess=preProc_Arguments)
plsModel
```

```
## Partial Least Squares
##
## 144 samples
##  56 predictor
##
## Pre-processing: centered (56), scaled (56)
## Resampling: Bootstrapped (25 reps)
## Summary of sample sizes: 144, 144, 144, 144, 144, 144, ...
## Resampling results across tuning parameters:
##
##   ncomp  RMSE      Rsquared   MAE
##    1     1.606147  0.3348312  1.192398
##    2     2.616416  0.2525362  1.344721
##    3     2.261047  0.3026285  1.276079
##    4     2.249021  0.3164123  1.281933
##    5     2.543660  0.2914832  1.354735
##    6     2.838645  0.2619764  1.432249
##    7     3.249180  0.2450920  1.530636
##    8     3.534123  0.2352648  1.606542
##    9     3.788171  0.2258456  1.668000
##   10     4.015881  0.2112639  1.722710
##
## RMSE was used to select the optimal model using the smallest value.
## The final value used for the model was ncomp = 1.
```

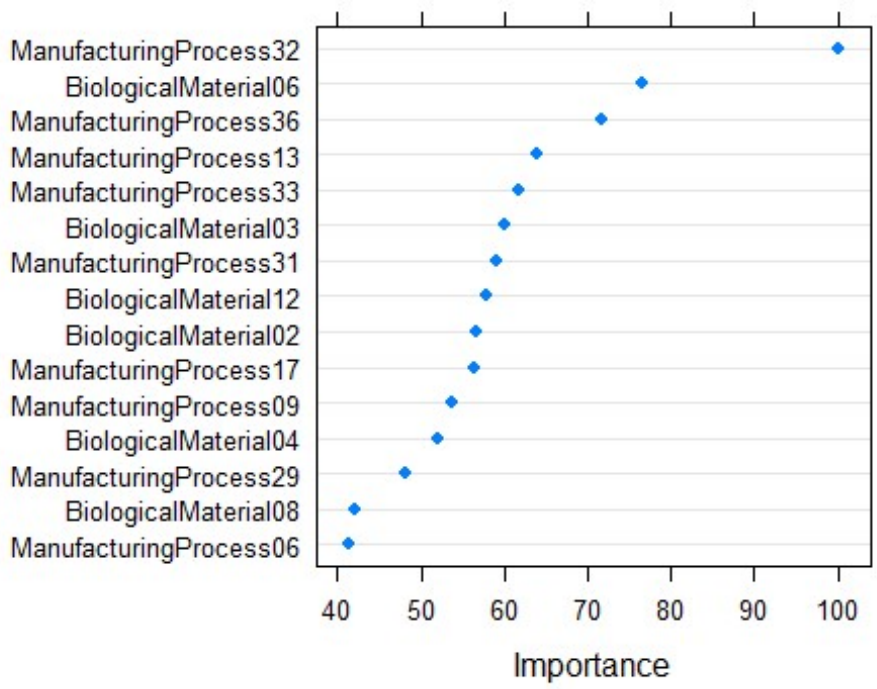
```
# Lets see what variables are most important in the pls model:
dotPlot(varImp(plsModel), top=15)
```



b. KNN

```
# A K-NN model:
set.seed(100)
knnModel = train(x=processPredictors_training, y=yield_training, method="knn"
, preProc=preProc_Arguments, tuneLength=10)

# predict on training/testing sets
knnPred = predict(knnModel, newdata=processPredictors_training)
knnPR = postResample(pred=knnPred, obs=yield_training)
rmses_training = c(knnPR[1])
r2s_training = c(knnPR[2])
methods = c("KNN")

pred.train.knn<- data.frame(cbind(rmses_training, r2s_training))

knnPred = predict(knnModel, newdata=processPredictors_testing)
knnPR = postResample(pred=knnPred, obs=yield_testing)
rmses_testing = c(knnPR[1])
r2s_testing = c(knnPR[2])

pred.test.knn<- data.frame(cbind(rmses_testing, r2s_testing))
knnModel

## k-Nearest Neighbors
##
## 144 samples
```

```
##  56 predictor
##
## Pre-processing: centered (56), scaled (56)
## Resampling: Bootstrapped (25 reps)
## Summary of sample sizes: 144, 144, 144, 144, 144, 144, ...
## Resampling results across tuning parameters:
##
##   k   RMSE      Rsquared   MAE
##    5  1.470315  0.3687432  1.145282
##    7  1.441013  0.3916406  1.131379
##    9  1.439163  0.3954489  1.136648
##   11  1.436249  0.4026296  1.141582
##   13  1.434620  0.4080690  1.141594
##   15  1.438548  0.4121259  1.143584
##   17  1.444909  0.4126914  1.149710
##   19  1.450598  0.4140712  1.149589
##   21  1.460501  0.4116571  1.157581
##   23  1.472091  0.4033518  1.166578
##
## RMSE was used to select the optimal model using the smallest value.
## The final value used for the model was k = 13.

pred.train.knn

##      rmses_training r2s_training
## RMSE       1.241575     0.566462

pred.test.knn

##      rmses_testing r2s_testing
## RMSE      1.596003   0.4502603

# Lets see what variables are most important in the MARS model:
dotPlot(varImp(knnModel), top=15)
```



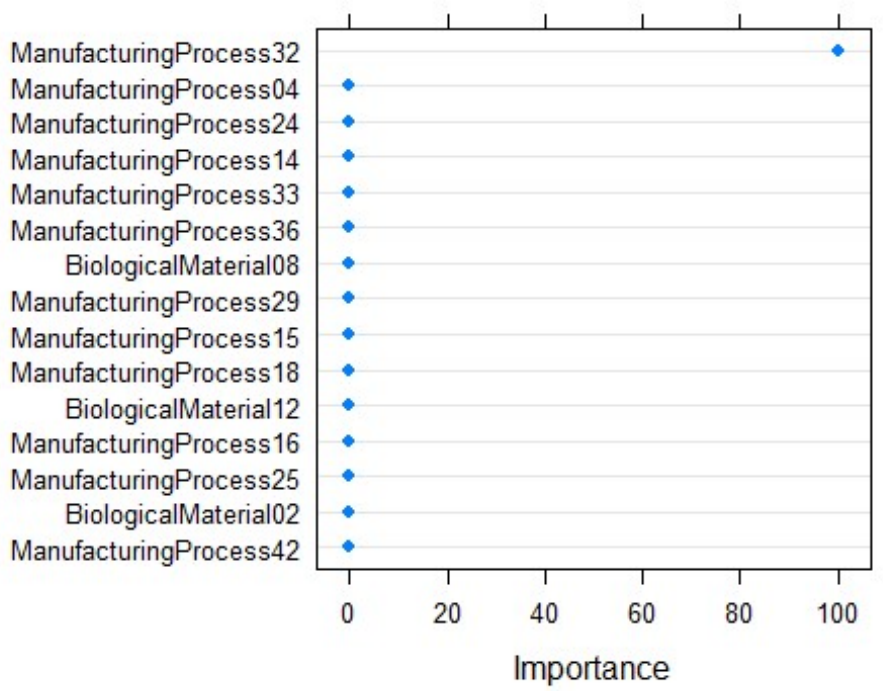
c. MARS

```
# MARS model:
marsGrid = expand.grid(.degree=1:2, .nprune=2:38)
set.seed(100)
marsModel = train(x=processPredictors_training, y=yield_training, method="ear
th", preProc=preProc_Arguments, tuneGrid=marsGrid)

marsPred = predict(marsModel, newdata=processPredictors_training)
marsPR = postResample(pred=marsPred, obs=yield_training)
rmses_training = c(rmses_training,marsPR[1])
r2s_training = c(r2s_training,marsPR[2])
methods = c(methods,"MARS")

pred.train.mars<- data.frame(cbind(rmses_training, r2s_training))

marsPred = predict(marsModel, newdata=processPredictors_testing)
marsPR = postResample(pred=marsPred, obs=yield_testing)
rmses_testing = c(rmses_testing,marsPR[1])
r2s_testing = c(r2s_testing,marsPR[2])

pred.test.mars<- data.frame(cbind(rmses_testing, r2s_testing))


marsModel
```

```
## Multivariate Adaptive Regression Spline
##
## 144 samples
##  56 predictor
##
## Pre-processing: centered (56), scaled (56)
## Resampling: Bootstrapped (25 reps)
## Summary of sample sizes: 144, 144, 144, 144, 144, 144, ...
## Resampling results across tuning parameters:
##
##   degree  nprune  RMSE      Rsquared   MAE
##   1        2      1.337151  0.4790123  1.044371
##   1        3      2.451967  0.5246877  1.163454
##   1        4      2.659078  0.4983851  1.202188
##   1        5      2.427845  0.4858812  1.186132
##   1        6      2.844089  0.4433150  1.266668
##   1        7      3.310084  0.4204201  1.351856
##   1        8      2.700904  0.4379416  1.269914
##   1        9      3.495957  0.3916653  1.402524
##   1       10      3.792364  0.3805513  1.446673
##   1       11      4.177186  0.3593005  1.516468
##   1       12      3.775477  0.3688420  1.478919
##   1       13      4.680886  0.3275937  1.618349
##   1       14      4.383436  0.3115230  1.594634
##   1       15      4.582657  0.3090088  1.626380
##   1       16      4.236303  0.3081437  1.588525
##   1       17      4.302603  0.3088288  1.597112
##   1       18      4.286381  0.3097487  1.595393
##   1       19      4.289130  0.3094598  1.596080
##   1       20      4.291710  0.3093852  1.597559
##   1       21      4.296745  0.3087382  1.598127
##   1       22      4.296745  0.3087382  1.598127
##   1       23      4.302942  0.3082131  1.603612
##   1       24      4.305441  0.3077901  1.604965
##   1       25      4.304930  0.3071020  1.605520
##   1       26      4.305669  0.3075048  1.605704
##   1       27      4.307095  0.3069246  1.606610
##   1       28      4.307332  0.3066939  1.608166
##   1       29      4.309387  0.3063270  1.608482
##   1       30      4.315124  0.3057524  1.612510
##   1       31      4.311237  0.3053641  1.611400
##   1       32      4.309146  0.3046149  1.609631
##   1       33      4.310194  0.3037503  1.608994
##   1       34      4.311315  0.3048814  1.610440
##   1       35      4.307734  0.3048285  1.608950
##   1       36      4.330227  0.3007319  1.614174
##   1       37      4.334214  0.2998526  1.613614
##   1       38      4.322701  0.3010904  1.610889
##   2        2      1.345555  0.4714679  1.053690
##   2        3      2.471280  0.5081173  1.179458
```

```
##   2        4      2.598490  0.4778338  1.209053
##   2        5      2.696246  0.4326667  1.259740
##   2        6      2.697818  0.4231996  1.271407
##   2        7      3.113208  0.3916284  1.340831
##   2        8      3.067344  0.3824652  1.332873
##   2        9      3.164055  0.3850711  1.358673
##   2       10      3.013927  0.3840193  1.342080
##   2       11      3.865343  0.3364183  1.581944
##   2       12      4.166328  0.3073726  1.631589
##   2       13      4.160213  0.2927092  1.572803
##   2       14      4.674119  0.2697273  1.755725
##   2       15      4.774054  0.2683552  1.766710
##   2       16      4.870083  0.2608880  1.793716
##   2       17      4.688700  0.2343572  1.778118
##   2       18      5.180577  0.2367059  1.861901
##   2       19      5.421072  0.2232073  1.910054
##   2       20      5.309574  0.1980600  1.911964
##   2       21      5.792738  0.1861919  2.000826
##   2       22      5.899997  0.1830459  2.076372
##   2       23      5.915569  0.1812768  2.106135
##   2       24      6.542384  0.1742493  2.232929
##   2       25      6.620992  0.1711038  2.261212
##   2       26      6.605927  0.1700223  2.260263
##   2       27      6.739054  0.1719858  2.279209
##   2       28      6.962719  0.1643088  2.334231
##   2       29      6.948969  0.1620855  2.336345
##   2       30      6.881643  0.1664310  2.339031
##   2       31      7.028239  0.1667145  2.364606
##   2       32      7.083725  0.1643350  2.387525
##   2       33      7.100343  0.1636304  2.390179
##   2       34      7.019171  0.1631205  2.377913
##   2       35      7.238098  0.1623559  2.434382
##   2       36      7.242257  0.1605259  2.437154
##   2       37      7.247018  0.1610770  2.440052
##   2       38      7.300230  0.1609540  2.455346
##
## RMSE was used to select the optimal model using the smallest value.
## The final values used for the model were nprune = 2 and degree = 1.

pred.train.mars

##        rmses_training r2s_training
## RMSE         1.241575    0.5664620
## RMSE.1       1.301557    0.4705536

pred.test.mars

##        rmses_testing r2s_testing
## RMSE        1.596003   0.4502603
## RMSE.1      1.823847   0.2278359
```

```
# Lets see what variables are most important in the MARS model:
dotPlot(varImp(marsModel), top=15)
```



d. SVM

```
# A Support Vector Machine (SVM):
set.seed(100)
svmModel = train(x=processPredictors_training, y=yield_training, method="svmR
adial", preProc=preProc_Arguments, tuneLength=20)

svmPred = predict(svmModel, newdata=processPredictors_training)
svmPR = postResample(pred=svmPred, obs=yield_training)
rmses_training = c(rmses_training,svmPR[1])
r2s_training = c(r2s_training,svmPR[2])
methods = c(methods,"SVM")

pred.train.svm<- data.frame(cbind(rmses_training, r2s_training))

svmPred = predict(svmModel, newdata=processPredictors_testing)
svmPR = postResample(pred=svmPred, obs=yield_testing)
rmses_testing = c(rmses_testing,svmPR[1])
r2s_testing = c(r2s_testing,svmPR[2])



pred.test.svm<- data.frame(cbind(rmses_testing, r2s_testing))
```

```
svmModel
```

```
## Support Vector Machines with Radial Basis Function Kernel
##
## 144 samples
##  56 predictor
##
## Pre-processing: centered (56), scaled (56)
## Resampling: Bootstrapped (25 reps)
## Summary of sample sizes: 144, 144, 144, 144, 144, 144, ...
## Resampling results across tuning parameters:
##
##   C          RMSE      Rsquared   MAE
##        0.25  1.464937  0.4214672  1.158680
##        0.50  1.390638  0.4539078  1.094862
##        1.00  1.344224  0.4783981  1.059965
##        2.00  1.312820  0.4974055  1.035729
##        4.00  1.295658  0.5088374  1.018830
##        8.00  1.290419  0.5125148  1.012466
##       16.00  1.290462  0.5124509  1.012496
##       32.00  1.290462  0.5124509  1.012496
##       64.00  1.290462  0.5124509  1.012496
##      128.00  1.290462  0.5124509  1.012496
##      256.00  1.290462  0.5124509  1.012496
##      512.00  1.290462  0.5124509  1.012496
##     1024.00  1.290462  0.5124509  1.012496
##     2048.00  1.290462  0.5124509  1.012496
##     4096.00  1.290462  0.5124509  1.012496
##     8192.00  1.290462  0.5124509  1.012496
##    16384.00  1.290462  0.5124509  1.012496
##    32768.00  1.290462  0.5124509  1.012496
##    65536.00  1.290462  0.5124509  1.012496
##   131072.00  1.290462  0.5124509  1.012496
##
## Tuning parameter 'sigma' was held constant at a value of 0.01491264
## RMSE was used to select the optimal model using the smallest value.
## The final values used for the model were sigma = 0.01491264 and C = 8.
```

```
pred.train.svm
```

```
##        rmses_training r2s_training
## RMSE        1.2415750    0.5664620
## RMSE.1      1.3015567    0.4705536
## RMSE.2      0.1773248    0.9924706
```

```
pred.test.svm
```

```
##        rmses_testing r2s_testing
## RMSE        1.596003   0.4502603
```

```
## RMSE.1      1.823847   0.2278359
## RMSE.2      1.216252   0.6565869
```

Questions - Answered

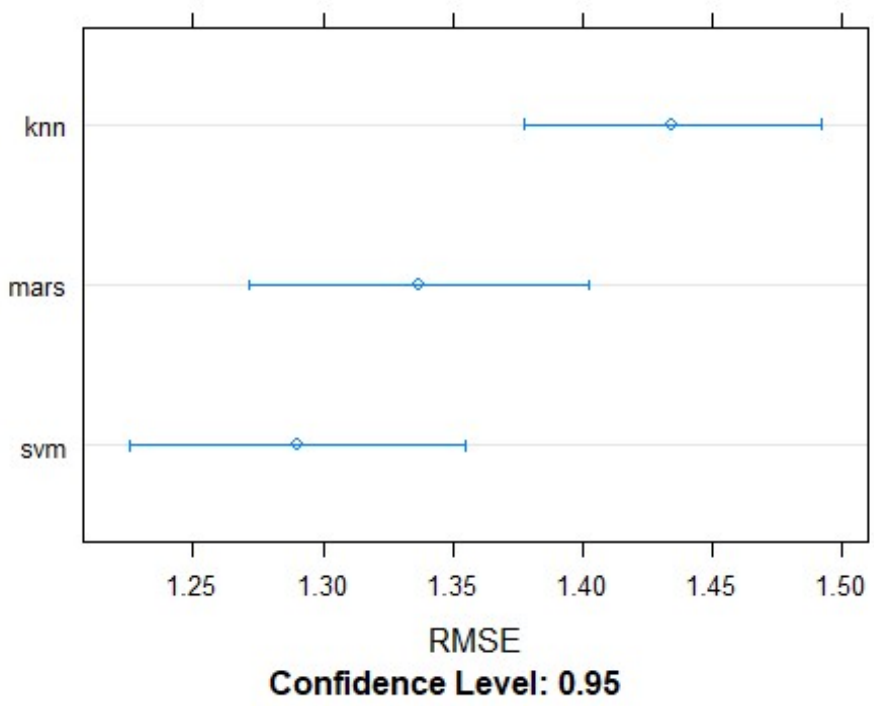
a. Which nonlinear regression model gives the optimal resampling and test set performance?

- The test data used for predictions for KNN, MARS and SVM had RMSE values of 1.59, 1.82 and 1.21 respectively. The SVM model achieved this fit and appears to be the optimal model of those attempted.

```
# Package the results up:
res_training = data.frame( rmse=rmses_training, r2=r2s_training )
rownames(res_training) <- methods

training_order = order( -res_training$rmse )

res_training = res_training[ training_order, ] # Order the dataframe so that
the best results are at the bottom:
print("Final Training Results")

## [1] "Final Training Results"

res_training

##           rmse        r2
## MARS 1.3015567 0.4705536
## KNN  1.2415750 0.5664620
## SVM  0.1773248 0.9924706

res_testing = data.frame( rmse=rmses_testing, r2=r2s_testing )
rownames(res_testing) = methods

res_testing = res_testing[ training_order, ] # Order the dataframe so that th
e best results for the training set are at the bottom:
print("Final Testing Results")

## [1] "Final Testing Results"

res_testing

##          rmse        r2
## MARS 1.823847 0.2278359
## KNN  1.596003 0.4502603
## SVM  1.216252 0.6565869

resamp = resamples( list(knn=knnModel,svm=svmModel,mars=marsModel) )
summary(resamp)

##
## Call:
## summary.resamples(object = resamp)
```

```
##
## Models: knn, svm, mars
## Number of resamples: 25
##
## MAE
##           Min.   1st Qu.    Median     Mean  3rd Qu.     Max. NA's
## knn  0.9789189 1.0674709 1.1443701 1.141594 1.215663 1.365669    0
## svm  0.8362991 0.9339826 0.9990398 1.012466 1.091015 1.278862    0
## mars 0.8135195 0.9580927 1.0438588 1.044371 1.124134 1.303500    0
##
## RMSE
##           Min.  1st Qu.   Median     Mean  3rd Qu.     Max. NA's
## knn  1.1488511 1.311098 1.439503 1.434620 1.554134 1.644363    0
## svm  1.0351677 1.168407 1.234980 1.290419 1.424835 1.603327    0
## mars 0.9904353 1.228108 1.345768 1.337151 1.453645 1.599669    0
##
## Rsquared
##           Min.   1st Qu.    Median      Mean   3rd Qu.      Max. NA's
## knn  0.2375618 0.3604480 0.4134720 0.4080690 0.4597024 0.5372964    0
## svm  0.2900938 0.4737095 0.5370702 0.5125148 0.5813574 0.6268504    0
## mars 0.2710663 0.4423782 0.4777335 0.4790123 0.5385581 0.6461168    0
```

```
dotplot( resamp, metric="RMSE" )
```

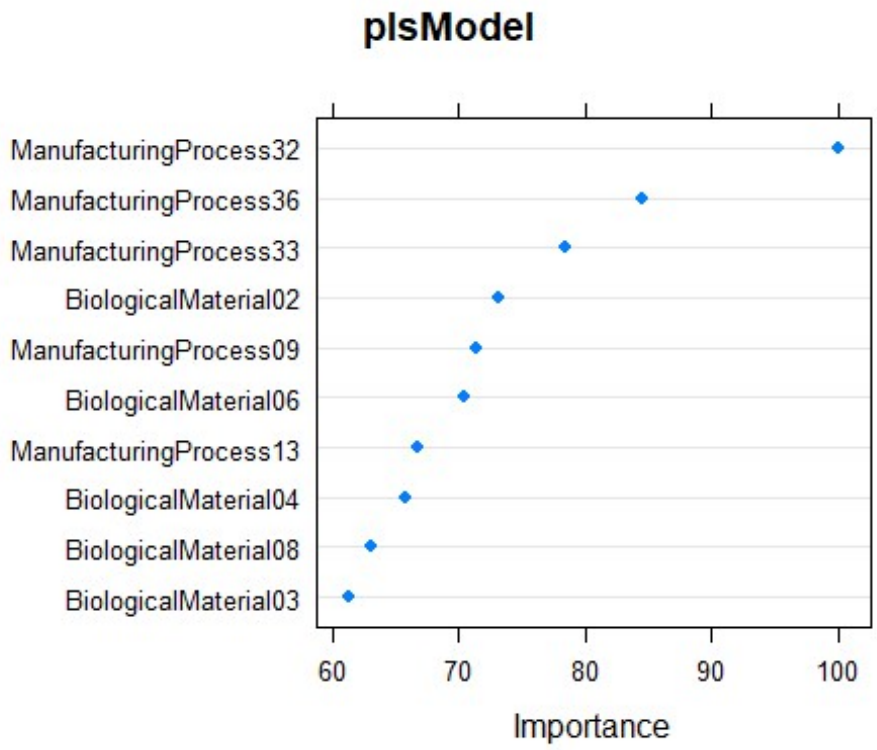



```
summary(diff(resamp))
```

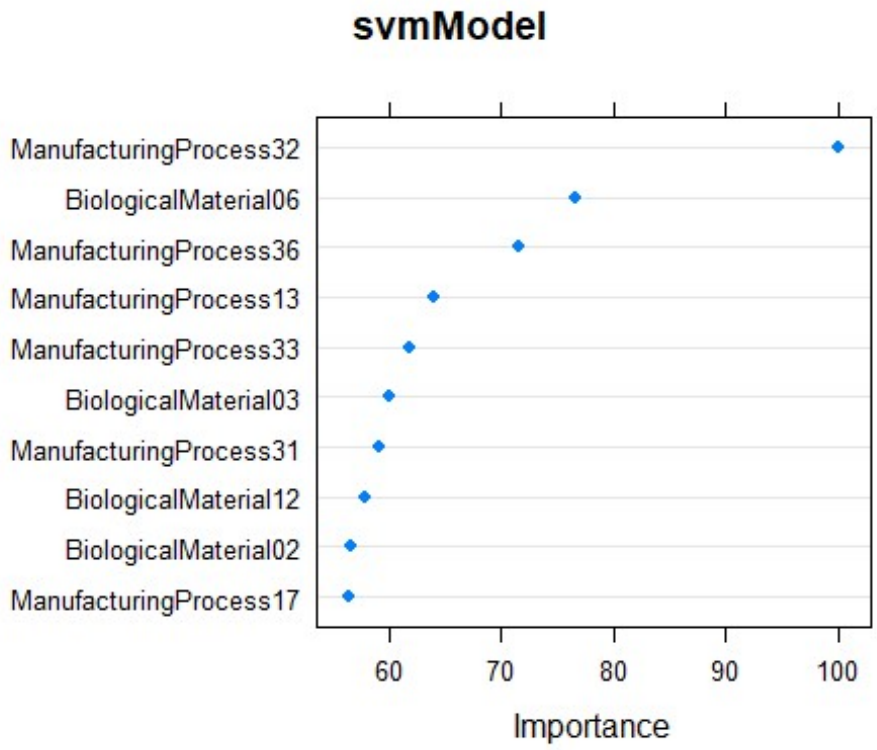
```
##
## Call:
## summary.diff.resamples(object = diff(resamp))
##
## p-value adjustment: bonferroni
## Upper diagonal: estimates of the difference
## Lower diagonal: p-value for H0: difference = 0
##
## MAE
##      knn       svm      mars
## knn             0.12913  0.09722
## svm  5.720e-07          -0.03190
## mars 3.194e-05 0.7201
##
## RMSE
##      knn       svm      mars
## knn             0.14420  0.09747
## svm  9.119e-07          -0.04673
## mars 2.690e-05 0.3419
##
## Rsquared
##      knn       svm       mars
## knn             -0.10445  -0.07094
## svm  2.125e-06             0.03350
## mars 0.0006429 0.5008691
```

b. The variable importance, Which predictors are most important in the optimal nonlinear regression model? Do either the biological or process variables dominate the list? How do the top ten important predictors compare to the top ten predictors from the optimal linear model? - Yes, ManufacturingProcessXX dominates the list. There are 4 BiologicalMaterials that rank 2,6,8,9. - ManufacturingProcess32 is the most important predictor in both pls and svm. The remaining dominant predictors are also very similar and maintain similar order of importance.

```
dotPlot(varImp(plsModel),main="plsModel", top=10)
```



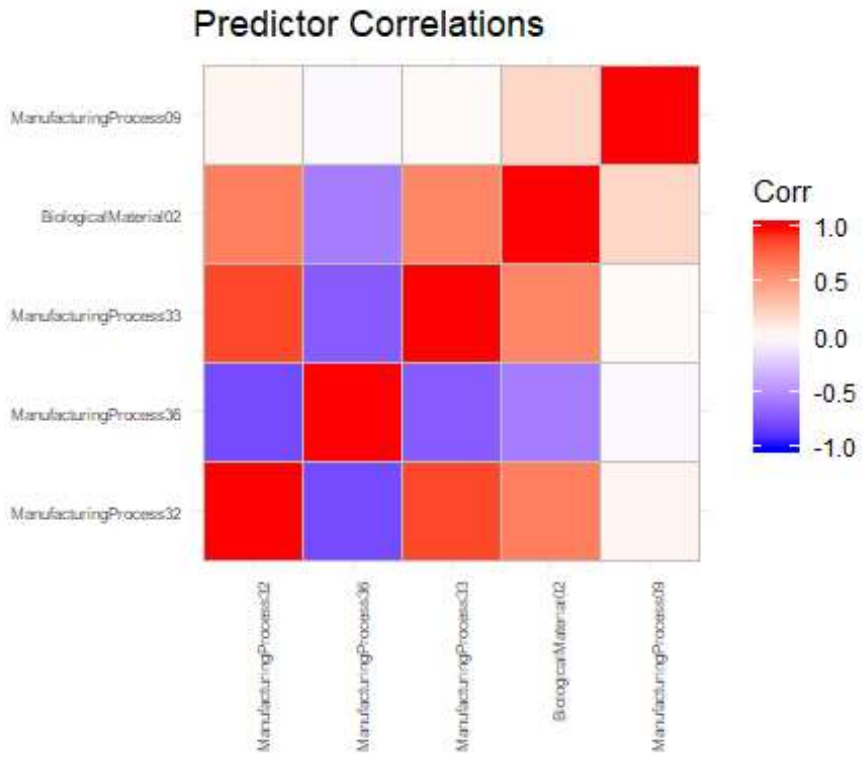

```
dotPlot(varImp(svmModel),main="svmModel", top=10)
```



c. Explore the relationships between the top predictors and the response for the predictors that are unique to the optimal nonlinear regression model.Do these plots reveal intuition about the biological or process predictors and their relationship with yield?

- Explore the correlations between the top svm predictors ("ManufacturingProcess32","ManufacturingProcess36", "ManufacturingProcess33", "BiologicalMaterial02","ManufacturingProcess09")
- "ManufacturingProcess32" and "ManufacturingProcess33" have a correlation value of .87 which means they are likely providing same information and are redundant in the model
- The four of the five top predictors show low to moderate correlation with the response variable
- Next predictor and plot how the response varies as a function of this value. Build a dataframe with variation by each of the five dataframes

```
#y=yield_training
top.pred.svm <- processPredictors_training %>%
  select(c("ManufacturingProcess32","ManufacturingProcess36", "ManufacturingP
rocess33", "BiologicalMaterial02","ManufacturingProcess09"))

print("Cor() TOP Pred")

## [1] "Cor() TOP Pred"

ggcorrplot(cor(top.pred.svm))+
  theme(axis.text.x=element_text(size=rel(.7), angle=90, hjust=1),
        axis.text.y = element_text(size=rel(.7), hjust=1))+
  ggtitle("Predictor Correlations")
```




```
reshape2::melt(cor(top.pred.svm))%>%
  rename(Predictor1 = Var1, Predictor2 = Var2, CorrelationValue = value)%>%
  filter(CorrelationValue != 1)%>%
 filter (! duplicated(CorrelationValue))
```

```
##                Predictor1            Predictor2 CorrelationValue
## 1  ManufacturingProcess36 ManufacturingProcess32      -0.77629083
## 2  ManufacturingProcess33 ManufacturingProcess32       0.87294630
## 3    BiologicalMaterial02 ManufacturingProcess32       0.64617672
## 4  ManufacturingProcess09 ManufacturingProcess32       0.04626312
## 5  ManufacturingProcess33 ManufacturingProcess36      -0.71488392
## 6    BiologicalMaterial02 ManufacturingProcess36      -0.55764138
## 7  ManufacturingProcess09 ManufacturingProcess36      -0.02562218
## 8    BiologicalMaterial02 ManufacturingProcess33       0.61165716
## 9  ManufacturingProcess09 ManufacturingProcess33       0.02846168
## 10 ManufacturingProcess09   BiologicalMaterial02       0.20623385
```

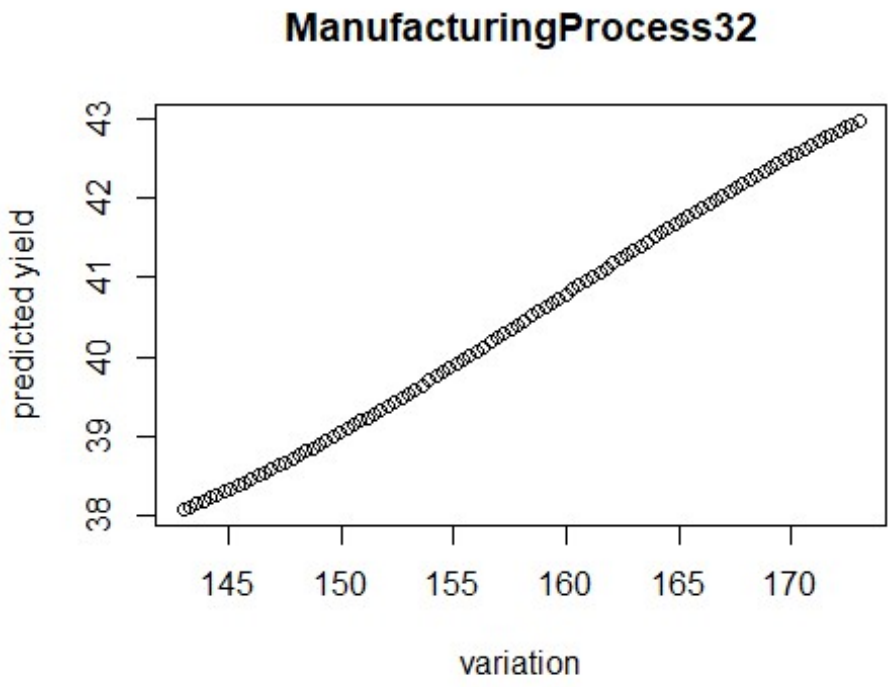
```
print("Cor() Against Yield")
```

```
## [1] "Cor() Against Yield"
```

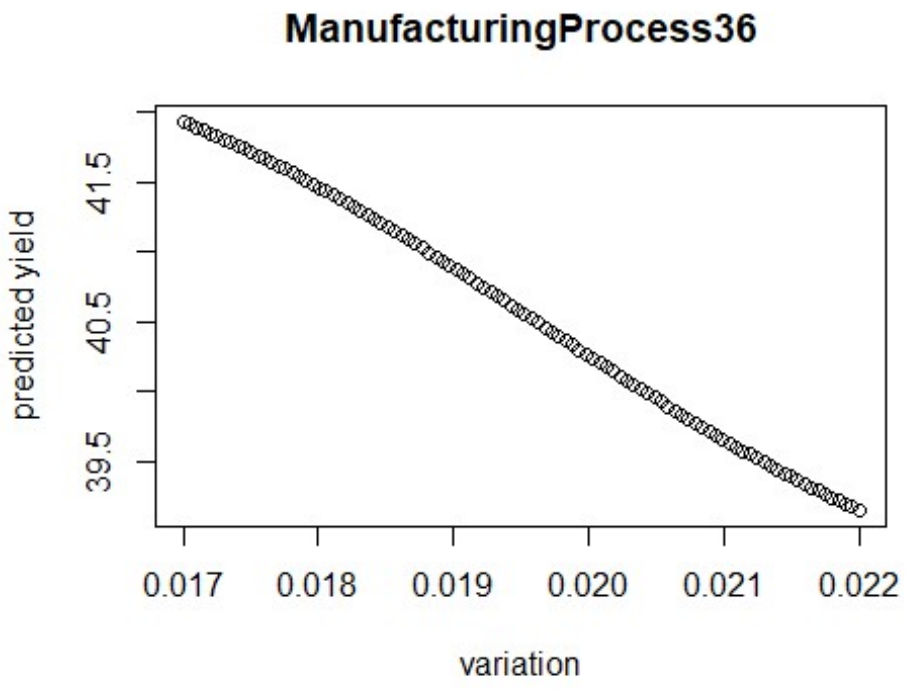
```
cor(top.pred.svm, yield_training)
```

```
##                              [,1]
## ManufacturingProcess32  0.6490227
## ManufacturingProcess36 -0.5494493
## ManufacturingProcess33  0.5100480
```

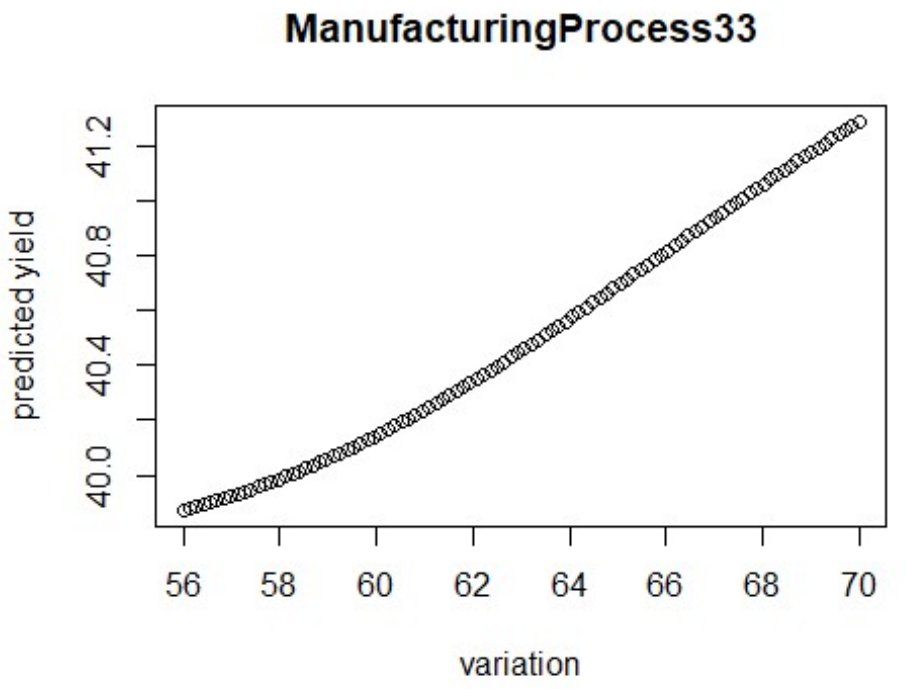
```
## BiologicalMaterial02    0.4752546
## ManufacturingProcess09  0.4640182

plot.funct <- function(processPredictors, predictor){
  p_range = range( processPredictors[,predictor] )

  variation = seq( from=p_range[1], to=p_range[2], length.out=100 )
  mean_predictor_values = apply( processPredictors, 2, mean )

  # build a dataframe with variation in only one dimension (for this part we
pick ManufacturingProcess32)

  newdata = repmat( as.double(mean_predictor_values), length(variation), 1 )
  newdata = data.frame( newdata )

  colnames( newdata ) = colnames( processPredictors )
  newdata[,predictor] = variation
  xs = variation
  y_hat = predict( svmModel, newdata=as.matrix(newdata) )
  return(plot( xs, y_hat, xlab='variation', ylab='predicted yield' , main = p
redictor))


}
plot.funct(processPredictors, "ManufacturingProcess32")
```



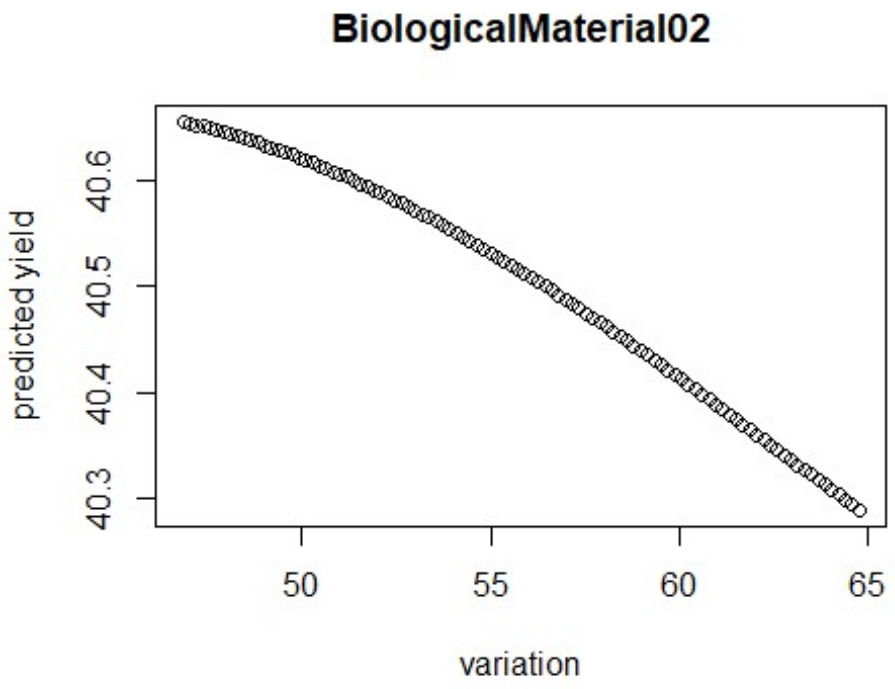
```
plot.funct(processPredictors, "ManufacturingProcess36")
```



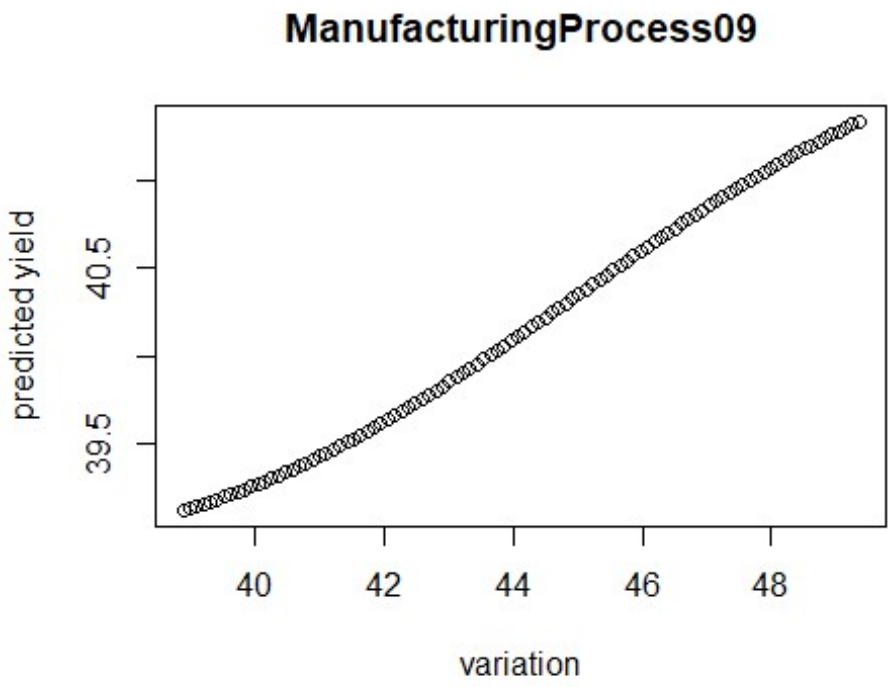

```
plot.funct(processPredictors, "ManufacturingProcess33")
```



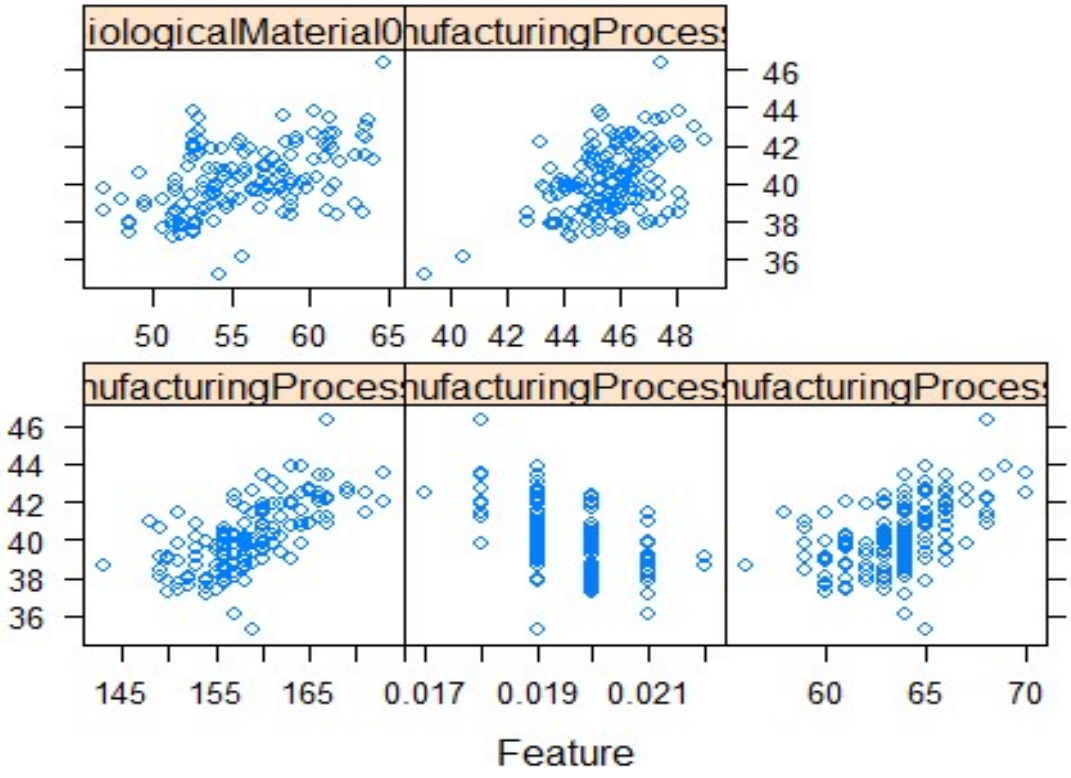
```
plot.funct(processPredictors, "BiologicalMaterial02")
```




```
plot.funct(processPredictors, "ManufacturingProcess09")
```



```
featurePlot(top.pred.svm, yield_training)
```



Data 624: Week 10 Homework

Angrand, Burke, Deboch, Groysman, Karr

December 10, 2019

Week 10 Assignment

KJ 8.1, 8.2, 8.3, 8.7

```
### Load packages
suppressMessages(library("AppliedPredictiveModeling"))
suppressMessages(library("caret"))
suppressMessages(library("ipred"))
suppressMessages(library("mlbench"))
suppressMessages(library("party"))
suppressMessages(library("randomForest"))
suppressMessages(library("gbm"))
suppressMessages(library("rpart"))
suppressMessages(library("Cubist"))
suppressMessages(library("dplyr"))
suppressMessages(library("gridExtra"))
suppressMessages(library("partykit"))
suppressMessages(library("xgboost"))
```

Exercise 8.1

Recreate the simulated data from Exercise 7.2:

Per 7.2, Friedman (1991) introduced several benchmark data sets create by simulation. One of these simulations used the following nonlinear equation to create data:

$$y = 10sin(??x1x2) + 20(x3???0.5)2 + 10x4 + 5x5 + N(0,??2)$$

where the x values are random variables uniformly distributed between [0, 1] (there are also 5 other non-informative variables also created in the simulation).

```
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:

```
model1 = randomForest( y ~ ., data=simulated, importance=TRUE, ntree=1000 )
rfImp1 = varImp(model1, scale=FALSE)
rfImp1 = rfImp1[ order(-rfImp1), , drop=FALSE ]
print("randomForest (no correlated predictor)")
```

```
## [1] "randomForest (no correlated predictor)"
```

```
print("Table 1: Variable importance scores for part (a) simulation.")
```

```
## [1] "Table 1: Variable importance scores for part (a) simulation."
```

```
print(rfImp1)
```

```
##          Overall
## V1   8.732235404
## V4   7.615118809
## V2   6.415369387
## V5   2.023524577
## V3   0.763591825
## V6   0.165111172
## V7  -0.005961659
## V10 -0.074944788
## V9  -0.095292651
## V8  -0.166362581
```

Q. Did the random forest model significantly use the uninformative predictors (V6 - V10)?

R. The predictor significance for the simulated data set in this model can be seen in Table 1. The model weights predictors V1,V4,V2,V5,V3 in order of diminishing significance and trailing off after that.

b. Fit another random forest model to these data. Did the importance score for V1 change? What happens when you add another predictors that is also highly correlated with V1?

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

```
## [1] 0.9460206
```

- After adding a highly correlated, predictors are ordered V4,V1,V2,duplicate1,V5 in order of diminishing significance and trailing off after that. V1 moved down to 3rd place in ranking.

```
model2 = randomForest( y ~ ., data=simulated, importance=TRUE, ntree=1000 )
rfImp2 = varImp(model2, scale=FALSE)
rfImp2 = rfImp2[ order(-rfImp2), , drop=FALSE ]
print("randomForest (one correlated predictor)")
```

```
## [1] "randomForest (one correlated predictor)"
```

```
print("Table 2: Variable importance scores for part (b) simulation.")
```

```
## [1] "Table 2: Variable importance scores for part (b) simulation."
```

```
print(rfImp2)
```

```
##                Overall
## V4          7.04752238
## V2          6.06896061
## V1          5.69119973
## duplicate1  4.28331581
## V5          1.87238438
## V3          0.62970218
## V6          0.13569065
## V10         0.02894814
## V9          0.00840438
## V7         -0.01345645
## V8         -0.04370565
```

```
simulated$duplicate2 = simulated$V1 + rnorm(200) * .1
cor(simulated$duplicate2,simulated$V1)
```

```
## [1] 0.9408631
```

- Adding a 2nd highly correlated variable, predictors are ordered V2,V2,V1,duplicate2,duplicate1 moving V1 to 3rd rank.

```
model3 = randomForest( y ~ ., data=simulated, importance=TRUE, ntree=1000 )
rfImp3 = varImp(model3, scale=FALSE)
rfImp3 = rfImp3[ order(-rfImp3), , drop=FALSE ]
print("randomForest (two correlated predictors)")
```

```
## [1] "randomForest (two correlated predictors)"
```

```
print(rfImp3)
```

```
##                Overall
## V4          7.04870917
## V2          6.52816504
## V1          4.91687329
## duplicate1  3.80068234
## V5          2.03115561
## duplicate2  1.87721959
## V3          0.58711552
## V6          0.14213148
## V7          0.10991985
## V10         0.09230576
## V9         -0.01075028
## V8         -0.08405687
```

c. Study this when fitting conditional inference trees:

-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?

- Yes, the conditional inference model has a similar pattern of importance as the random forest model from Part (a). Predictor's rank importance scores for the conditional inference random forests are shown as follows :

no correlated predictor V1,V4,V2,V5,V7 one correlated predictor and V4,V1,V2,duplicate1,V5 two correlated predictor V4,V1,V2,duplicate2,duplicate1

- So once again, adding highly correlated predictors reduces the value other predictors and reduces the rank of V1.

```
simulated$duplicate1 = NULL
simulated$duplicate2 = NULL
model1 = cforest( y ~ ., data=simulated )
cfImp1 = as.data.frame(varimp(model1),conditional=TRUE)
cfImp1 = cfImp1[ order(-cfImp1), , drop=FALSE ]
print(sprintf("cforest (no correlated predictor);varimp(*,conditional=%s)",TR
UE))
```

```
## [1] "cforest (no correlated predictor);varimp(*,conditional=TRUE)"
```

```
print(cfImp1)
```

```
##     varimp(model1)
## V1      8.06914297
## V4      7.17043007
## V2      6.10899476
## V5      2.15748730
## V3      0.38216895
## V9     -0.03067555
## V7     -0.08552060
## V8     -0.11126857
## V6     -0.16569917
## V10    -0.25947380
```

```
# Now we add correlated predictors one at a time
simulated$duplicate1 = simulated$V1 + rnorm(200) * 0.1
model2 = cforest( y ~ ., data=simulated )
cfImp2 = as.data.frame(varimp(model2),conditional=use_conditional_true)
cfImp2 = cfImp2[ order(-cfImp2), , drop=FALSE ]
print(sprintf("cforest (one correlated predictor);varimp(*,conditional=%s)",T
RUE))
```

```
## [1] "cforest (one correlated predictor);varimp(*,conditional=TRUE)"
```

```
print(cfImp2)
```

```
##            varimp(model2)
## V1             7.03783456
```

```
## V4             6.86760421
## V2             6.10664414
## duplicate1     4.19225992
## V5             1.94009208
## V3             0.18086871
## V6            -0.01608504
## V7            -0.02984121
## V8            -0.05042838
## V10           -0.05595558
## V9            -0.20738687
```

```
simulated$duplicate2 = simulated$V1 + rnorm(200) * 0.1
model3 = cforest( y ~ ., data=simulated )
cfImp3 = as.data.frame(varimp(model3),conditional=TRUE)
cfImp3 = cfImp3[ order(-cfImp3), , drop=FALSE ]
print(sprintf("cforest (two correlated predictor); varimp(*,conditional=%s)",
TRUE))
```

```
## [1] "cforest (two correlated predictor); varimp(*,conditional=TRUE)"
```

```
print(cfImp3)
```

```
##            varimp(model3)
## V4             6.37489054
## V1             6.35477591
## V2             5.54686894
## duplicate1     3.74174008
## duplicate2     3.48900969
## V5             2.00427921
## V3             0.35797772
## V7             0.01581870
## V6            -0.08873844
## V10           -0.09035362
## V9            -0.09220024
## V8            -0.20995828
```

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

The gbm pattern does re-rank the importance of predictors but with less change in rank. Adding an extra highly correlated predictor with has a lesser impact on the overall importance of predictors compared to that of random forest.

no correlated predictor V4,V1,V2,V5,V3 one correlated predictor and V4,V2,V1,duplicate1,V5 two correlated predictor V4,V2,V1,V5,V3

```
simulated$duplicate1 = NULL
simulated$duplicate2 = NULL

model1 = gbm( y ~ ., data=simulated, distribution="gaussian", n.trees=1000 )
print(sprintf("gbm (no correlated predictor)"))
```

```
## [1] "gbm (no correlated predictor)"
```

```
print(summary(model1,plotit=F)) # the summary method gives variable importanc
e ...
```

```
##     var   rel.inf
## V4   V4 24.983999
## V1   V1 22.067384
## V2   V2 21.419174
## V5   V5 11.023160
## V3   V3  8.915081
## V7   V7  3.191040
## V8   V8  2.349239
## V6   V6  2.161459
## V9   V9  2.084007
## V10 V10  1.805458
```

```
# Now we add correlated predictors one at a time
simulated$duplicate1 = simulated$V1 + rnorm(200) * 0.1
model2 = gbm( y ~ ., data=simulated, distribution="gaussian", n.trees=1000 )
print(sprintf("gbm (one correlated predictor)"))
```

```
## [1] "gbm (one correlated predictor)"
```

```
print(summary(model2,plotit=F))
```

```
##                   var   rel.inf
## V4                 V4 24.478223
## V1                 V1 22.905602
## V2                 V2 19.576695
## V5                 V5 11.320659
## V3                 V3  8.929942
## V6                 V6  2.623043
## V7                 V7  2.601706
## V9                 V9  2.100246
## duplicate1 duplicate1  1.943571
## V10               V10  1.818640
## V8                 V8  1.701675
```

```
simulated$duplicate2 = simulated$V1 + rnorm(200) * 0.1
model3 = gbm( y ~ ., data=simulated, distribution="gaussian", n.trees=1000 )
print(sprintf("gbm (two correlated predictor)"))
```

```
## [1] "gbm (two correlated predictor)"
```

```
print(summary(model3,plotit=F))
```

```
##                   var   rel.inf
## V4                 V4 26.147234
## V1                 V1 19.636669
## V2                 V2 18.302074
## V5                 V5 10.923626
```

```
## V3                 V3  9.235543
## duplicate2 duplicate2  3.783240
## V6                 V6  2.705703
## V7                 V7  2.588724
## V8                 V8  2.081522
## V9                 V9  1.973067
## V10               V10  1.320825
## duplicate1 duplicate1  1.301773
```

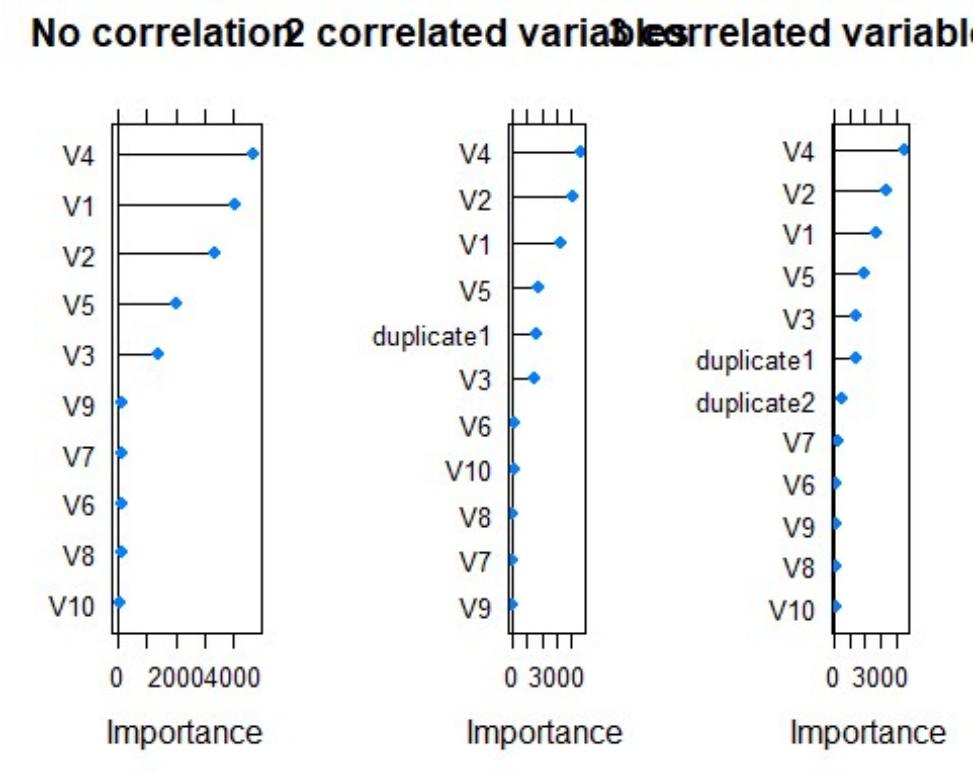
```
set.seed(200)
simulated1 <- mlbench.friedman1(200, sd = 1)
simulated1 <- cbind(simulated1$x, simulated1$y)
simulated1 <- as.data.frame(simulated1)
colnames(simulated1)[ncol(simulated1)] <- "y"
set.seed(200)
simulated2 <-
  simulated1 %>%
  mutate(duplicate1 = V1 + rnorm(200) * .1)
cor(simulated2$duplicate1, simulated2$V1)
```

```
## [1] 0.9497025
```

```
# add another correlated variable
set.seed(5)
simulated3 <-
  simulated2 %>%
  mutate(duplicate2 = V1 + rnorm(200) * .1)
cor(simulated3$duplicate2, simulated3$V1)
```

```
## [1] 0.9412195
```

```
gbm1 <- train(y ~ ., data = simulated1, method = "gbm", verbose = F)
gbm2 <- train(y ~ ., data = simulated2, method = "gbm", verbose = F)
gbm3 <- train(y ~ ., data = simulated3, method = "gbm", verbose = F)
gridExtra::grid.arrange(
  plot(varImp(gbm1, scale = F), main = "No correlation"),
  plot(varImp(gbm2, scale = F), main = "2 correlated variables"),
  plot(varImp(gbm3, scale = F), main = "3 correlated variables"),
  ncol = 3
)
```



- For Cubist indicates that predictors V1-V5 are at the top of the importance ranking. Adding an extra highly correlated predictor with V1 has very little impact on the importance scores when using Cubist.

```
vnames <- c('V1', 'V2', 'V3', 'V4', 'V5', 'V6', 'V7', 'V8', 'V9', 'V10')
cbFit1 <- cubist(x = simulated[, 1:10],
                 y = simulated$y,
                 committees = 100)
cbImp1 <- varImp(cbFit1)
names(cbImp1) <- "Original"
cbImp1$Variable <- factor(rownames(cbImp1), levels = vnames)
cbFit2 <- cubist(x = simulated[, names(simulated) != "y"],
                 y = simulated$y, committees = 100)
cbImp2 <- varImp(cbFit2)
names(cbImp2) <- "Extra"
cbImp2$Variable <- factor(rownames(cbImp2), levels = vnames)
cbImp <- merge(cbImp1, cbImp2, all = TRUE)
#rownames(cbImp) <- cbImp$Variable #this won't knit, commenting out
#cbImp$Variable <- NULL
print(cbImp)
```

```
##    Variable Original Extra
## 1        V1     71.5  69.0
## 2        V2     58.5  58.5
## 3        V3     47.0  47.5
## 4        V4     48.0  48.5
## 5        V5     33.0  31.5
```

```
## 6         V6     13.0  10.5
## 7         V7      0.0   0.0
## 8         V8      0.0   2.5
## 9         V9      0.0   0.0
## 10       V10      0.0   0.0
## 11      <NA>       NA   3.5
## 12      <NA>       NA   3.5
```

Exercise 8.2

a. Use a simulation to show tree bias with different granularities.

- Intuitively, predictors that appear higher in the tree (i.e., earlier splits) or those that appear multiple times in the tree will be more important than predictors that occur lower in the tree or not at all. even if the predictor has little-to-no relationship with the response.
- This simulation uses one categorical predictor splitting the response into two groups. As a comparison, a similar simulation uses a continuous predictor that doesn't split the response into two groups.
- simulations where X1 is categorical and X2 is continuous

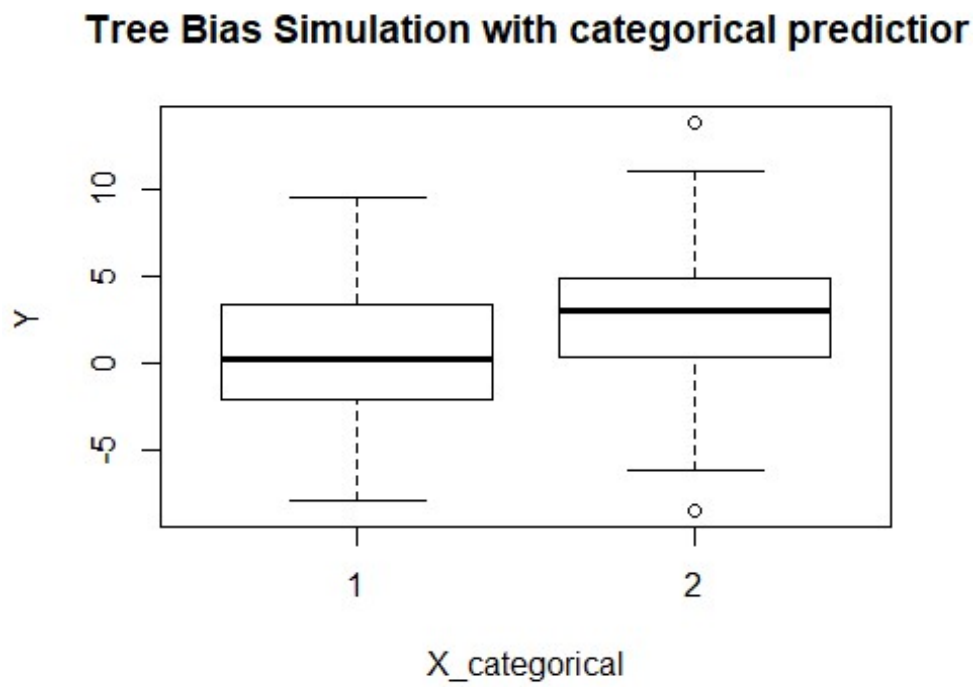
```
X_categorical <- rep(1:2,each=100)
Y <- X_categorical + rnorm(200,mean=0,sd=4)
set.seed(103)
X_continuous <- rnorm(200,mean=0,sd=2)
simData <- data.frame(Y=Y,X_categorical=X_categorical,X_continuous=X_continuo
us)
```

Note predictor X1 splits the response into two response groups and predictor X2, is independent of response.

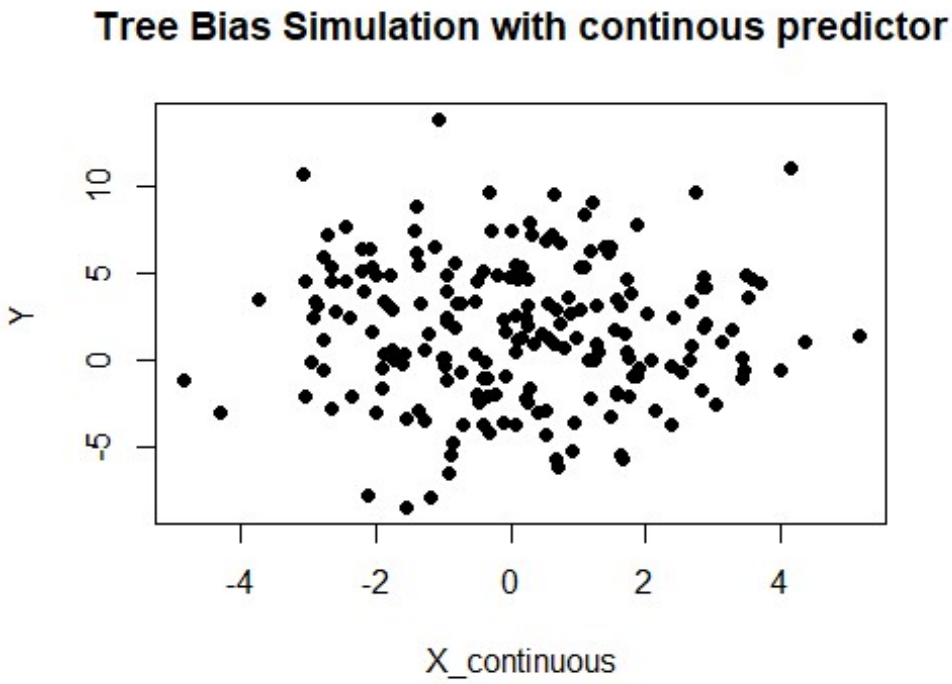
b) Plot frequency of predictor selection for tree bias simulation.

|————————| |X_categorical 52 |
|X_continuous 44 | |————————|

```
boxplot(
  Y~X_categorical,
  #data=Y,
  main="Tree Bias Simulation with categorical predictior",
  xlab="X_categorical",
  ylab="Y"
)
```




```r
plot(
  X_continuous,
  Y,
  main="Tree Bias Simulation with continous predictor",
  xlab="X_continuous",
  ylab="Y",
  pch=19
)
```



Exercise 8.3

8.3. In stochastic gradient boosting the bagging fraction and learning ratewill 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: fig824.

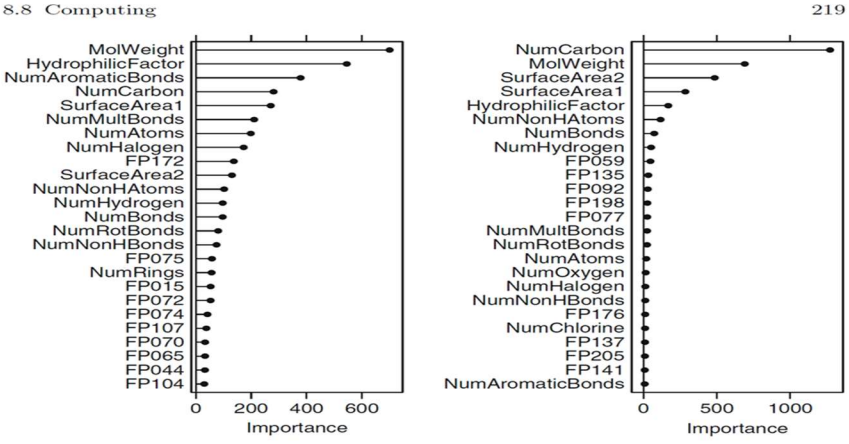


Fig. 8.24: A comparison of variable importance magnitudes for differing values of the bagging fraction and shrinkage parameters. Both tuning parameters are set to 0.1 in the *left* figure. Both are set to 0.9 in the *right* figure

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?

- To understand the justification it would be wise to give some background information on the concept of tuning parameters. A tuning parameter (λ), sometimes called a penalty parameter, controls the strength of the penalty term in ridge regression and lasso regression. It is basically the amount of shrinkage, where data values are shrunk towards a central point, like the mean. Shrinkage results in simple, sparse models which are easier to analyze than high-dimensional data models with large numbers of parameters. As we can observe from Figure 8.24 the right model (with both parameters set to 0.9) is focused on only the first few predictors for reasons related to each of the tuning parameters: the higher learning rate yields a "greedier" model (likely to identify fewer predictors); the higher bagging fraction means that a larger portion of the data is used in model building (yielding more concentrated importance on few variables).

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

- As it can be observed from the figure , the left model (with both tuning parameters set to 0.1) will likely have better predictive performance. On the other hand , the concentration of the model importance on fewer predictors in the right model will likely cause lower performance, as it does provide importance across the full range of predictors.

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

- Similar to the behavior of an increased bagging fraction, increased interaction depth (i.e. tree depth) is likely to spread importance more evenly across predictors. This in turn will lead to a decreased slope in predictor importance for both the left and right models.

Exercise 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:

Restructure the data

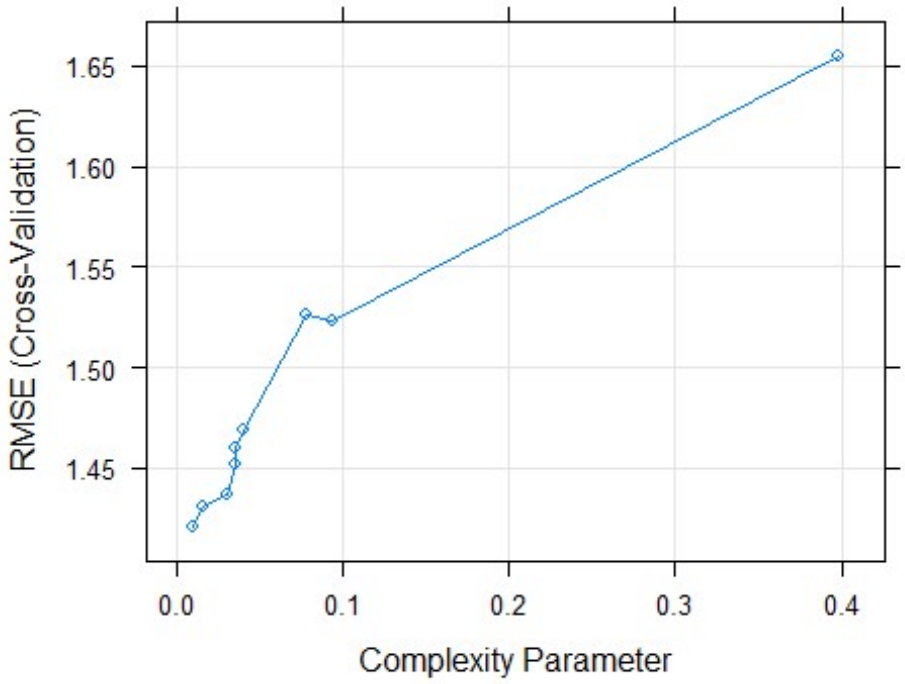
- Based on the instruction of the exercise, the imputed, split and pre-processed data from exercise 6.3 and 7.5 are used here. Both exercises, 6.3 and 7.5 are prepared in a similar way . So we will be using to answer this question accordingly.For effective reply the same training controls chem_ctrl are used as in exercises 6.3 & 7.5.

```
data(ChemicalManufacturingProcess) # mlbench
set.seed(123)
# preprocess for trees, impute missing
chem_preprocess <- preProcess(ChemicalManufacturingProcess, method = c("bagIm
pute"))
chem_df <- predict(chem_preprocess, ChemicalManufacturingProcess)
# train-test partition

training_rows <- createDataPartition(chem_df$Yield, p =.8, list=FALSE)

x_train <- chem_df[training_rows,]
x_test <- chem_df[-training_rows,]
y_test <- chem_df[-training_rows, "Yield"]
## parms for all models . . .
ctrl <- trainControl(method="cv", number=5, allowParallel=T, savePredictions=
"final")
```

Build Models

a. CART Model

```
#### CART Model ####
set.seed(123)
mcart <- train(Yield ~., data = x_train, method='rpart',metric="RMSE", trCont
rol=ctrl, tuneLength=10)
(mcart$bestTune)

##            cp
## 1 0.009077131

data.frame(model="CART", mcart$bestTune, RMSE=min(mcart$results$RMSE), row.na
mes="")

##  model          cp     RMSE
##   CART 0.009077131 1.420234

plot(mcart)
```




```
(mcart$results)
```

```
##             cp     RMSE  Rsquared      MAE    RMSESD RsquaredSD     MAESD
## 1  0.009077131 1.420234 0.4701538 1.076364 0.2431920 0.16169232 0.2119601
## 2  0.015694066 1.430581 0.4569493 1.088763 0.2291442 0.15411296 0.1935696
## 3  0.015713889 1.430581 0.4569493 1.088763 0.2291442 0.15411296 0.1935696
## 4  0.030217175 1.436403 0.4425863 1.114653 0.1928027 0.11071137 0.1765903
## 5  0.035229890 1.451428 0.4380820 1.116904 0.1849398 0.10185824 0.1750377
## 6  0.035757471 1.459647 0.4410795 1.123100 0.1683389 0.10260022 0.1639948
## 7  0.040346896 1.469066 0.4282490 1.125345 0.1858776 0.10507292 0.1509463
## 8  0.077621368 1.525884 0.3699531 1.186994 0.1816544 0.08751792 0.1571966
## 9  0.093282110 1.522987 0.3756902 1.205723 0.2042485 0.09105582 0.1529299
## 10 0.398636379 1.655397 0.3875512 1.333440 0.3192996 0.08264490 0.2839377
```

- The optimal resampled RMSE is 1.417223, is achieved with a parameter maxdepth = 7. We have utilized the same model to predict the test set:

```
chem_tree_pred <- predict(mcart, x_test)

chem_tree_perf <- defaultSummary(data.frame(obs = y_test, pred = chem_tree_pr
ed))
chem_tree_perf
```

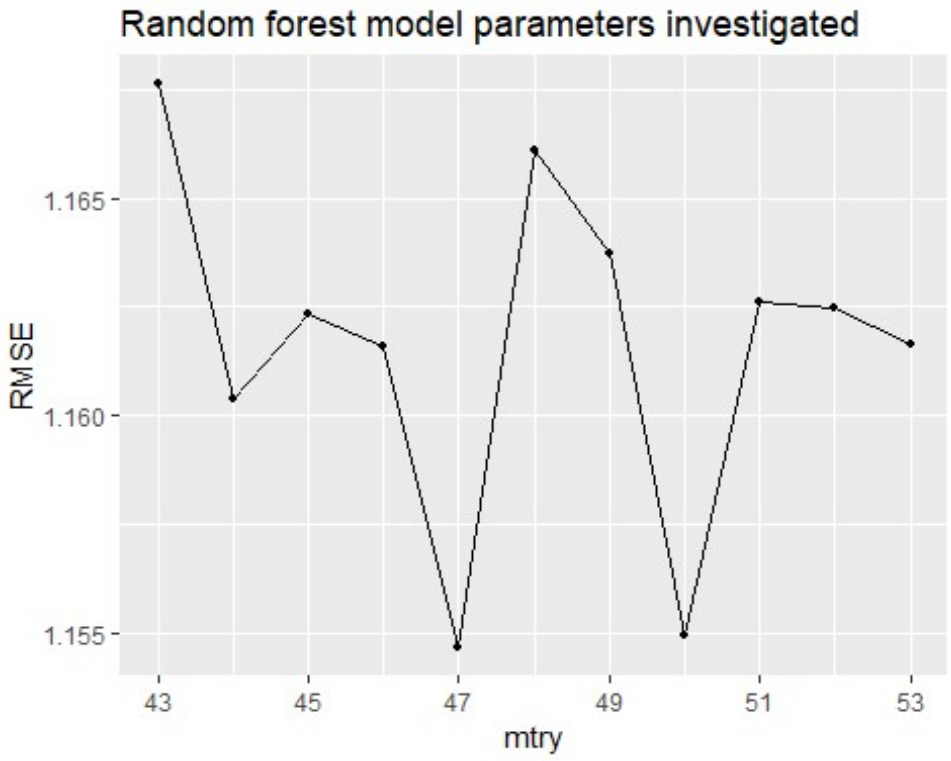
```
##      RMSE  Rsquared       MAE
## 1.7447743 0.3293661 1.4957603
```

b. Random Forest Model

```
set.seed(100)
chem_rf <- train(Yield ~., data = x_train,
                 method = "rf", trControl = ctrl,
                 tuneGrid = data.frame(mtry = nrow(x_train) / 3 + -5:5),
                 ntrees = 1000, importance = TRUE)
chem_rf
```

```
## Random Forest
##
## 144 samples
##  57 predictor
##
## No pre-processing
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 116, 114, 115, 116, 115
## Resampling results across tuning parameters:
##
##   mtry  RMSE      Rsquared   MAE
##   43    1.167621  0.6017570  0.9159220
##   44    1.160397  0.6081941  0.9074889
##   45    1.162357  0.6090829  0.9055400
##   46    1.161576  0.6093665  0.9058600
##   47    1.154677  0.6117039  0.9024386
##   48    1.166083  0.6039483  0.9152202
##   49    1.163724  0.6040586  0.9139077
##   50    1.154961  0.6112007  0.8998563
##   51    1.162608  0.6055907  0.9090759
##   52    1.162475  0.6051890  0.9093491
##   53    1.161623  0.6054888  0.9097813
##
## RMSE was used to select the optimal model using the smallest value.
## The final value used for the model was mtry = 47.
```

```
chem_rf$results %>%
  ggplot(aes(x = mtry, y = RMSE)) +
  geom_line() + geom_point(size = 1) +
  scale_x_continuous(breaks = chem_rf$results$mtry[seq(1, 11, 2)]) +
  labs(title = "Random forest model parameters investigated")
```



- The best RMSE of 1.250688 occurs at mtry = 47, equivalent to the default value mtry = nrow(x) / 3. As observed , the performance shows an interesting pattern, with fluctuations of increasing & decreasing values as mtry increases. Here is the test set is predicted with the random forest model:

```
chem_rf_pred  <- predict(chem_rf, x_test)

chem_rf_perf  <- defaultSummary(data.frame(obs = y_test, pred = chem_rf_pred)
)
chem_rf_perf

##      RMSE  Rsquared       MAE
## 1.1178421 0.6967837 0.8846789
```

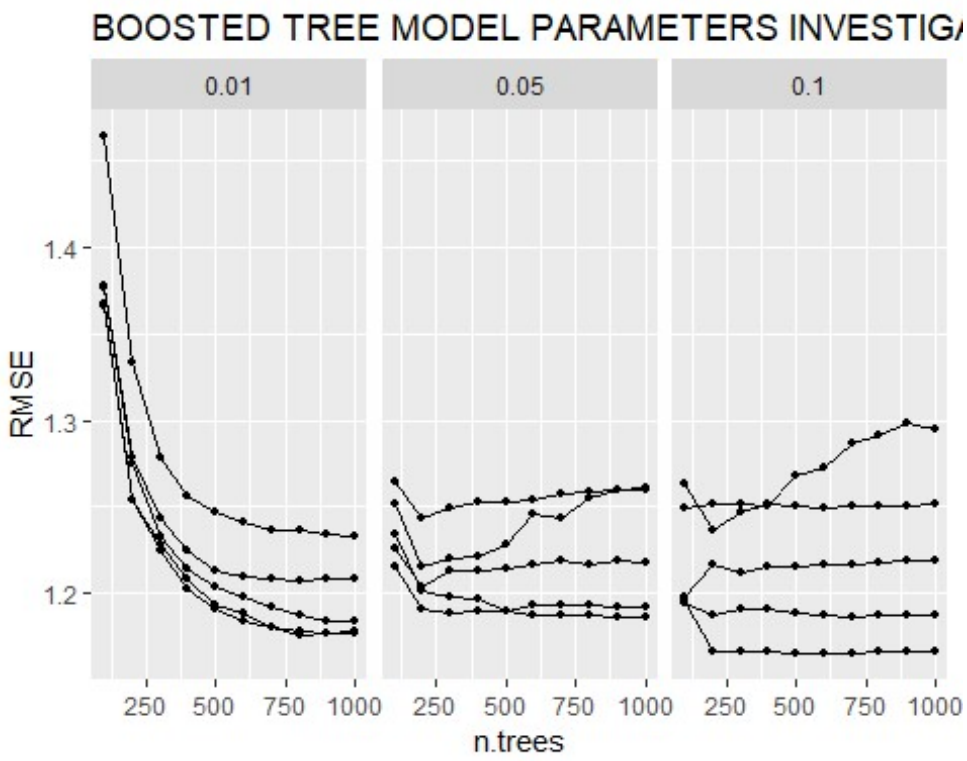
- From our observation we can learn that test performance of this model is 1.0167747 It shows the best resampled & test RMSE of the tree-based models investigated.

c. Extreme Gradient Boosting Trees

- Gradient boosting is a machine learning technique for regression and classification problems, which produces a prediction model in the form of an ensemble of weak prediction models, typically decision trees.

- From our observation we can see that gradient-boosted model is fitted, training across three shrinkage values, five interaction depths, and ten number of tree values, keeping the minimum number of observations per node set to the default of 10:

```
set.seed(100)
chem_boost <- train(Yield ~., data = x_train,
                    method = "gbm", trControl = ctrl,
                    tuneGrid = expand.grid(shrinkage = c(0.01, 0.05, 0.1),
                                           interaction.depth = seq(1, 9, 2),
                                           n.trees = seq(100, 1000, 100),
                                           n.minobsinnode = 10),
                    verbose = FALSE)

chem_boost$finalModel

## A gradient boosted model with gaussian loss function.
## 600 iterations were performed.
## There were 57 predictors of which 55 had non-zero influence.

(chem_boost$bestTune)

##     n.trees interaction.depth shrinkage n.minobsinnode
## 136     600                 7       0.1             10

min(chem_boost$results$RMSE)

## [1] 1.164757

chem_boost$results %>%
  ggplot(aes(x = n.trees, y = RMSE, COL = factor(interaction.depth))) +
  geom_line() + geom_point(size = 1) +
  facet_wrap(~ shrinkage, nrow  = 1) +
  labs(title = "BOOSTED TREE MODEL PARAMETERS INVESTIGATED", col = "INTERACTI
ON DEPTH") +
  theme(legend.position  = "TOP")
```



- The optimal model, with n.trees = 400, interaction.depth = 3, and shrinkage = 0.05, yields an optimal resampled RMSE of 1.194245 this is the lowest resampled RMSE observed. The boosted model is used to predict the test set:

```
chem_boost_pred <- predict(chem_boost, x_test)

chem_boost_perf <- defaultSummary(data.frame(obs = y_test, pred = chem_boost_
pred))
chem_boost_perf

##      RMSE  Rsquared       MAE
## 0.8075581 0.8487829 0.6696008
```

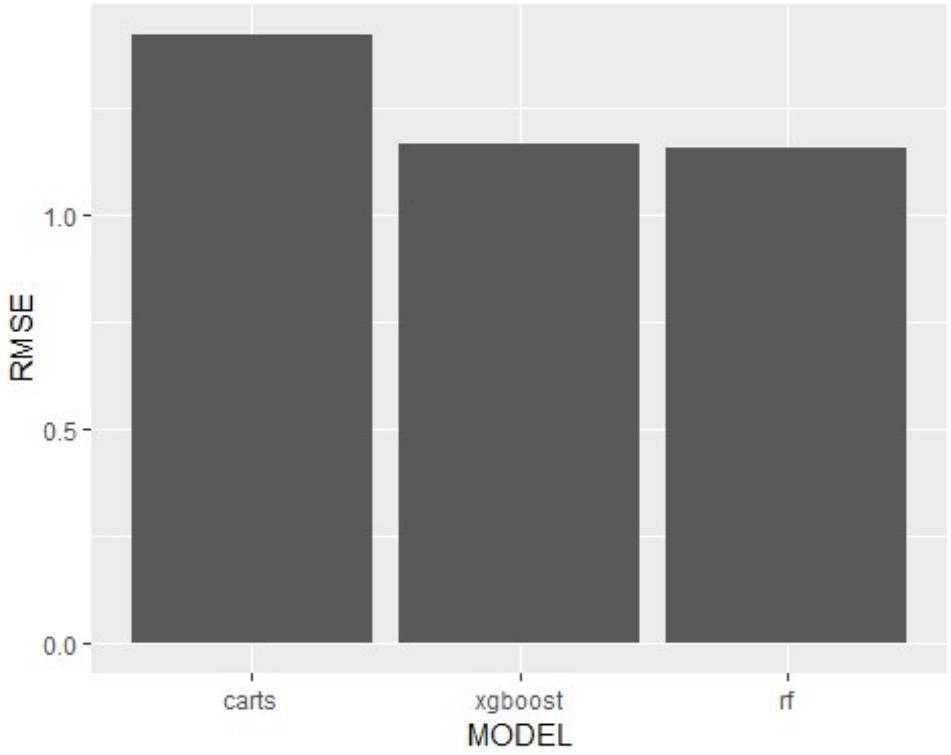
- The RMSE of this model against the test set is 0.9904002 - lowest from the observations so far .

Answer Questions

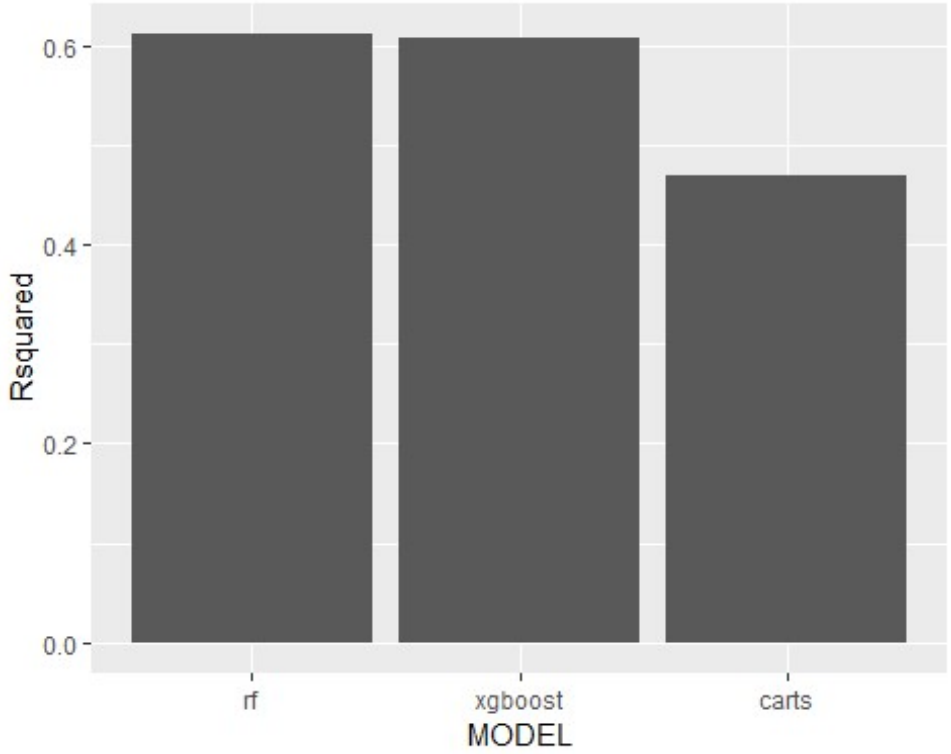
a. Which tree-based regression model gives the optimal resampling and test set performance?

```
#test data
rbind(chem_tree_perf, chem_rf_perf, chem_boost_perf)

##                       RMSE  Rsquared       MAE
## chem_tree_perf  1.7447743 0.3293661 1.4957603
```

```
## chem_rf_perf    1.1178421 0.6967837 0.8846789
## chem_boost_perf 0.8075581 0.8487829 0.6696008

# compare models on RMSE
df_mcarts <- data.frame(Model="carts",RMSE=mcart$results$RMSE, Rsquared=mcart
$results$Rsquared)
df_mcarts <- df_mcarts[df_mcarts$RMSE == min(df_mcarts$RMSE),]
df_results <- df_mcarts


df_mrf <- data.frame(Model="rf",RMSE=chem_rf$results$RMSE, Rsquared=chem_rf$r
esults$Rsquared)
df_mrf<- df_mrf[df_mrf$RMSE == min(df_mrf$RMSE),]
df_results <- rbind(df_results, df_mrf)


df_mxgb <- data.frame(Model="xgboost",RMSE=chem_boost$results$RMSE, Rsquared=
chem_boost$results$Rsquared)
df_mxgb <- df_mxgb[df_mxgb$RMSE == min(df_mxgb$RMSE),]
df_results <- rbind(df_results, df_mxgb)

df_results

##      Model     RMSE  Rsquared
## 1    carts 1.420234 0.4701538
## 5       rf 1.154677 0.6117039
## 87 xgboost 1.164757 0.6069876

ggplot() + geom_col(aes(x = reorder(df_results$Model, -df_results$RMSE), y =
df_results$RMSE)) + xlab("MODEL") + ylab("RMSE")
```



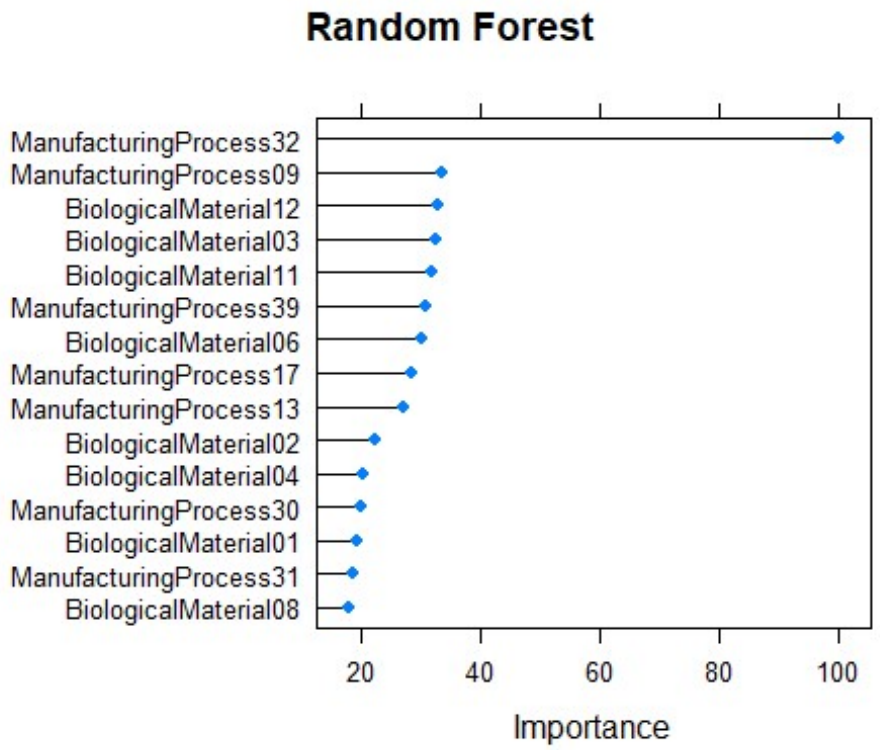

```
ggplot() + geom_col(aes(x = reorder(df_results$Model, -df_results$Rsquared),
y = df_results$Rsquared)) + xlab("MODEL") + ylab("Rsquared")
```



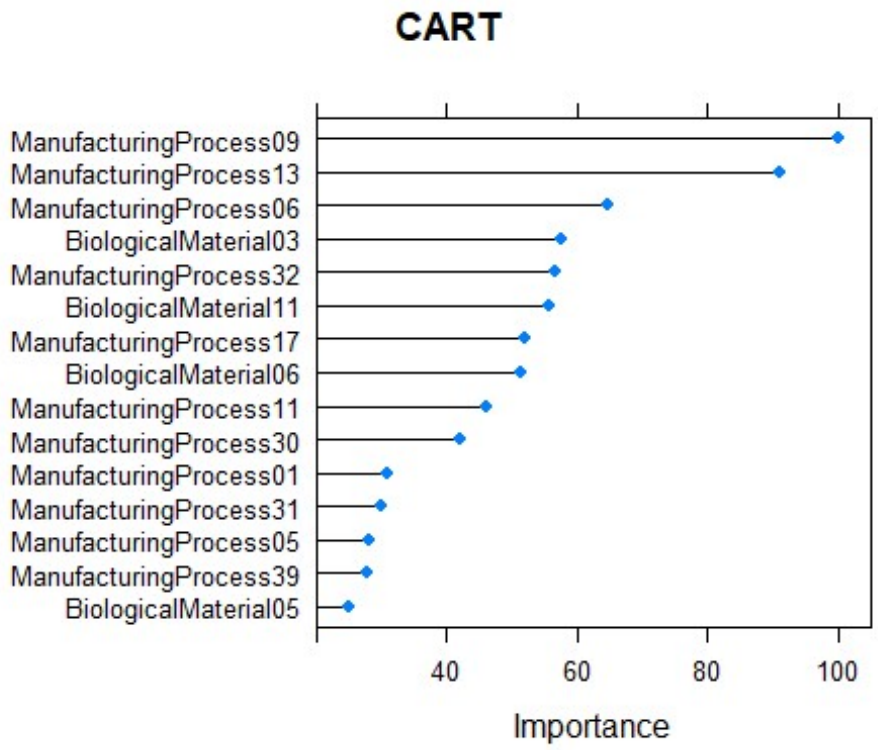
- The resampled data used for predictions for CARTS, Random Forest and Gradient Boosting had RMSE values of 1.41, 1.25 and 1.19 respectively.For the test data, the Gradient boosting model also performed the best with an RSME of 0.99.
- The optimal model, with n.trees = 400, interaction.depth = 3, and shrinkage = 0.05, yields the lowest resampled and test RMSE observed.

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?

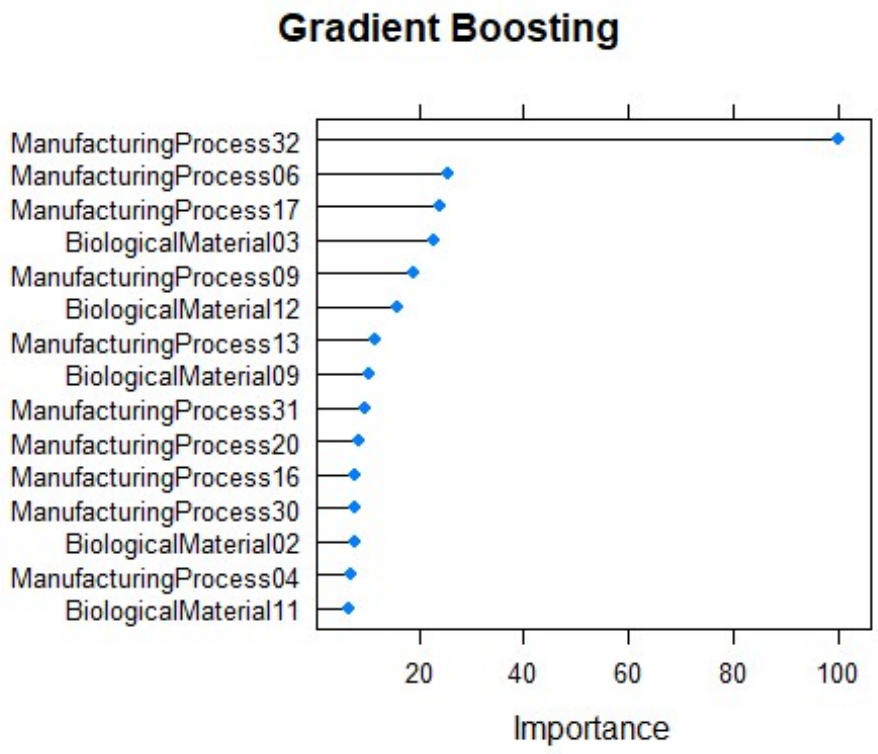
```
plot(varImp(chem_rf), top=15,main = "Random Forest")
```




```
plot(varImp(mcart), top=15,main = "CART")
```




```
plot(varImp(chem_boost), top=15,main = "Gradient Boosting")
```



The most important predictors in the gradient boosting model are :

```
1. ManufacturingProcess32
2. BiologicalMaterial12
2. ManufacturingProcess17
3. ManufacturingProcess06
4. ManufacturingProcess13
5. ManufacturingProcess09
```

- Manufacturing process dominate the top of the list. The random forest has a more balanced mixture of biological and process predictors. Although in both gradient boosting and random forest, ManufacturingProcess32 plays on outsized role-by more than a factor of 2. Also, the CART model shows a steep drop-off in variable importance after the 10th variable. Other models diminish more gradually with the exception of ManfuacturingProcess32–the top entry.

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?

- Kindly please see the optimal single tree visualization below together g with distribution of the response variable Yield in each terminal node. There's no new information about predictor names, but more detail about how they are evaluated The tree view shows the principal split point and percentages for each side of the split but does not provide any additional information about the predictors. As observed from the visualization manufacturing predictors are used more commonly than biological predictors.