Data624 - Homework9

Amit Kapoor

4/26/2021

Contents

Exercise 8.1 **1**
(a) 2
(b) 2
(c) 3
(d) 3

Exercise 8.2 **4**

Exercise 8.3 **5**
(a) 5
(b) 5
(c) 5

Exercise 8.7 **6**
Tree-based models 9
(a) 17
(b) 18
(c) 20

```r
library(mlbench)
library(randomForest)
library(caret)
library(party)
library(gbm)
library(Cubist)
library(rpart)
library(AppliedPredictiveModeling)
library(tidyverse)
library(naniar)
library(rpart.plot)
```

Exercise 8.1

Recreate the simulated data from Exercise 7.2

```r
set.seed(317)
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)
```

Did the random forest model significantly use the uninformative predictors (V6 – V10)?

```
rfImp1
```

```
##          Overall
## V1   6.017023817
## V2   7.197326850
## V3   1.969544508
## V4   9.875194405
## V5   2.087946997
## V6   0.060239433
## V7  -0.051620797
## V8  -0.024680384
## V9  -0.003664196
## V10 -0.082197496
```

Based on the above results, the random forest model doesnt significantly use the uninformative predictors (V6 – V10).

(b)

Now add an additional predictor that is highly correlated with one of the informative predictors. For example:

```
set.seed(317)
simulated$duplicate1 = simulated$V1 + rnorm(200) * .1
cor(simulated$duplicate1, simulated$V1)
```

```
## [1] 0.9400187
```

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?

```
model2 <- randomForest(y ~ ., data = simulated, importance = TRUE, ntree = 1000)
rfImp2 <- varImp(model2, scale = FALSE)
```

```
rfImp2
```

```
##                Overall
## V1          4.92452320
## V2          6.75435224
## V3          1.76154183
## V4          9.29279187
## V5          2.19868128
## V6          0.16614783
## V7         -0.04778316
## V8          0.01070291
## V9         -0.01098350
## V10        -0.15881709
## duplicate1  2.59912860
```

We see here after adding another predictor that is highly correlated with V1, its importance got reduced. The importance now is splitted between V1 and duplicate1 after adding the highly correlated duplicate1.

(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). Does this importance show the same pattern as the traditional random forest model?

```r
model3 <- cforest(y ~ ., data = simulated)
# conditional = TRUE
ctrue <- varimp(model3, conditional = TRUE)
# conditional = FALSE
cfalse <- varimp(model3, conditional = FALSE)
```

```r
cbind(model2=rfImp2, cforest_con=ctrue,cforest_uncon=cfalse )
```

```
##                  Overall cforest_con cforest_uncon
## V1            4.92452320  2.06175787  4.4446682522
## V2            6.75435224  5.06766860  6.8801452007
## V3            1.76154183  0.15834716  0.2782253511
## V4            9.29279187  6.67572503 10.2896859310
## V5            2.19868128  1.20148004  1.7910397555
## V6            0.16614783  0.09955285  0.1839316106
## V7           -0.04778316 -0.01022655  0.0151452375
## V8            0.01070291  0.01982557 -0.0004442511
## V9           -0.01098350  0.06097283  0.0166033367
## V10          -0.15881709 -0.06822672 -0.0523436249
## duplicate1    2.59912860  0.91810824  2.0665125520
```

We can see here that importance shows the different pattern as compared to traditional random forest model. V4, remains the most important variable in all the 3 cases. Also we can see that uninformative predictors remains low in all 3 cases.

(d)

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

```r
set.seed(317)

# boosting regression trees via stochastic gradient boosting machines

gbmGrid <- expand.grid(interaction.depth = seq(1, 7, by = 2),
                       n.trees = seq(100, 1000, by = 50),
                       shrinkage = 0.1,
                       n.minobsinnode = 5)

gbm_model <- train(y ~ ., data = simulated,
                  method = "gbm",
                  tuneGrid = gbmGrid,
                  verbose = FALSE)

gbm_imp <- varImp(gbm_model)
gbm_imp
```

```
## gbm variable importance
##
##               Overall
## V4         100.0000
## V2          80.1464
## V1          58.5491
## V3          48.3656
## V5          39.4555
## duplicate1  19.3401
## V6           6.1123
## V8           1.9197
## V7           0.4404
## V10          0.1300
## V9           0.0000
```

```
set.seed(317)

# cubist
cubist_model <- cubist(x = simulated[, names(simulated)[names(simulated) != 'y']],
                 y = simulated[,c('y')])
cubist_imp <- varImp(cubist_model)
cubist_imp
```

```
##            Overall
## V2            91.5
## V1            69.0
## V4            69.0
## V3            33.0
## V5            50.0
## V8             2.5
## V6             0.0
## V7             0.0
## V9             0.0
## V10            0.0
## duplicate1     0.0
```

Comparing the results with cforest, the uninformative predictors (V6-V10) still appear as lowest in ranking. V4 still appear as highest for boosted trees but cubist shows V2 as highest in ranking.

Exercise 8.2

Use a simulation to show tree bias with different granularities.

We will do the simulation here with 4 variables having different granularities. The response variable we will choose, would be a function of random selection and some noise.

```
set.seed(317)

df <- data.frame(x1 = sample(0:10000/10000, 250, replace = TRUE),
                 x2 = sample(0:100/100, 250, replace = TRUE),
                 x3 = sample(0:1000/1000, 250, replace = TRUE),
                 x4 = sample(0:10/10, 250, replace = TRUE))

df$y <- df$x1 + df$x4 + rnorm(250)

str(df)
```

```
## 'data.frame':    250 obs. of  5 variables:
##  $ x1: num  0.461 0.404 0.522 0.896 0.51 ...
##  $ x2: num  0.49 0.22 0.99 0.31 0.5 0.87 0.32 0.36 0.22 0.94 ...
##  $ x3: num  0.113 0.353 0.711 0.232 0.825 0.906 0.558 0.448 0.603 0.784 ...
##  $ x4: num  0.1 0.6 0.9 0.2 0.6 0.8 0.3 1 0.6 0.7 ...
##  $ y : num  0.824 0.444 2.531 1.344 2.635 ...
```

```r
# rpart
rp_model <- rpart(y~., data=df)
varImp(rp_model)
```

```
##      Overall
## x1 1.1982394
## x2 1.3662639
## x3 0.9649745
## x4 0.8960316
```

We can see the tree mostly uses x1 to split and x4 the least. Though x2 and x3 are not used to generate target but they are also used by tree to split. With this simulation, it is evident here that there is a selection bias in the tree model where favored predictors have more distinct values.

Exercise 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 the 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 predictors, whereas the model on the left spreads importance across more predictors?

Seeing the graphs, the one on the right has higher bagging fraction and higher learning rate that means it used larger chunk of data and increases the correlation in every iteration. Thus only less number of variables are considered important. Not the plot on the left has lower bagging fraction and lower learning rate that means it uses small chunk of data for model training and less dependent in each iteration. Thus the model on the left spreads importance across more predictors.

(b)

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

Bagging fraction and learning rate are considered important params to control overfitting. Based on above explanation, the model with smaller bagging fraction and learning rate will lead to better generalization over the test/new data. Given that, the model with smaller learning rate and bagging fraction would be more predictive over other samples.

(c)

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

Increasing the interaction depth would include more predictors and result to spread out importance. This would result the slope of predictor importance become flatten.

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:

```
data(ChemicalManufacturingProcess)
```

```
glimpse(ChemicalManufacturingProcess)
```

```
## Rows: 176
## Columns: 58
## $ Yield                  <dbl> 38.00, 42.44, 42.03, 41.42, 42.49, 43.57, 43.12~
## $ BiologicalMaterial01   <dbl> 6.25, 8.01, 8.01, 8.01, 7.47, 6.12, 7.48, 6.94,~
## $ BiologicalMaterial02   <dbl> 49.58, 60.97, 60.97, 60.97, 63.33, 58.36, 64.47~
## $ BiologicalMaterial03   <dbl> 56.97, 67.48, 67.48, 67.48, 72.25, 65.31, 72.41~
## $ BiologicalMaterial04   <dbl> 12.74, 14.65, 14.65, 14.65, 14.02, 15.17, 13.82~
## $ BiologicalMaterial05   <dbl> 19.51, 19.36, 19.36, 19.36, 17.91, 21.79, 17.71~
## $ BiologicalMaterial06   <dbl> 43.73, 53.14, 53.14, 53.14, 54.66, 51.23, 54.45~
## $ BiologicalMaterial07   <dbl> 100, 100, 100, 100, 100, 100, 100, 100, 100, 10~
## $ BiologicalMaterial08   <dbl> 16.66, 19.04, 19.04, 19.04, 18.22, 18.30, 18.72~
## $ BiologicalMaterial09   <dbl> 11.44, 12.55, 12.55, 12.55, 12.80, 12.13, 12.95~
## $ BiologicalMaterial10   <dbl> 3.46, 3.46, 3.46, 3.46, 3.05, 3.78, 3.04, 3.85,~
## $ BiologicalMaterial11   <dbl> 138.09, 153.67, 153.67, 153.67, 147.61, 151.88,~
## $ BiologicalMaterial12   <dbl> 18.83, 21.05, 21.05, 21.05, 21.05, 20.76, 20.75~
## $ ManufacturingProcess01 <dbl> NA, 0.0, 0.0, 0.0, 10.7, 12.0, 11.5, 12.0, 12.0~
## $ ManufacturingProcess02 <dbl> NA, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0~
## $ ManufacturingProcess03 <dbl> NA, NA, NA, NA, NA, NA, 1.56, 1.55, 1.56, 1.55,~
## $ ManufacturingProcess04 <dbl> NA, 917, 912, 911, 918, 924, 933, 929, 928, 938~
## $ ManufacturingProcess05 <dbl> NA, 1032.2, 1003.6, 1014.6, 1027.5, 1016.8, 988~
## $ ManufacturingProcess06 <dbl> NA, 210.0, 207.1, 213.3, 205.7, 208.9, 210.0, 2~
## $ ManufacturingProcess07 <dbl> NA, 177, 178, 177, 178, 178, 177, 178, 177, 177~
## $ ManufacturingProcess08 <dbl> NA, 178, 178, 177, 178, 178, 178, 178, 177, 177~
## $ ManufacturingProcess09 <dbl> 43.00, 46.57, 45.07, 44.92, 44.96, 45.32, 49.36~
## $ ManufacturingProcess10 <dbl> NA, NA, NA, NA, NA, NA, 11.6, 10.2, 9.7, 10.1, ~
## $ ManufacturingProcess11 <dbl> NA, NA, NA, NA, NA, NA, 11.5, 11.3, 11.1, 10.2,~
## $ ManufacturingProcess12 <dbl> NA, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0~
## $ ManufacturingProcess13 <dbl> 35.5, 34.0, 34.8, 34.8, 34.6, 34.0, 32.4, 33.6,~
## $ ManufacturingProcess14 <dbl> 4898, 4869, 4878, 4897, 4992, 4985, 4745, 4854,~
## $ ManufacturingProcess15 <dbl> 6108, 6095, 6087, 6102, 6233, 6222, 5999, 6105,~
## $ ManufacturingProcess16 <dbl> 4682, 4617, 4617, 4635, 4733, 4786, 4486, 4626,~
## $ ManufacturingProcess17 <dbl> 35.5, 34.0, 34.8, 34.8, 33.9, 33.4, 33.8, 33.6,~
## $ ManufacturingProcess18 <dbl> 4865, 4867, 4877, 4872, 4886, 4862, 4758, 4766,~
## $ ManufacturingProcess19 <dbl> 6049, 6097, 6078, 6073, 6102, 6115, 6013, 6022,~
## $ ManufacturingProcess20 <dbl> 4665, 4621, 4621, 4611, 4659, 4696, 4522, 4552,~
## $ ManufacturingProcess21 <dbl> 0.0, 0.0, 0.0, 0.0, -0.7, -0.6, 1.4, 0.0, 0.0, ~
## $ ManufacturingProcess22 <dbl> NA, 3, 4, 5, 8, 9, 1, 2, 3, 4, 6, 7, 8, 10, 11,~
## $ ManufacturingProcess23 <dbl> NA, 0, 1, 2, 4, 1, 1, 2, 3, 1, 3, 4, 1, 2, 3, 4~
## $ ManufacturingProcess24 <dbl> NA, 3, 4, 5, 18, 1, 1, 2, 3, 4, 6, 7, 8, 2, 15,~
## $ ManufacturingProcess25 <dbl> 4873, 4869, 4897, 4892, 4930, 4871, 4795, 4806,~
## $ ManufacturingProcess26 <dbl> 6074, 6107, 6116, 6111, 6151, 6128, 6057, 6059,~
## $ ManufacturingProcess27 <dbl> 4685, 4630, 4637, 4630, 4684, 4687, 4572, 4586,~
## $ ManufacturingProcess28 <dbl> 10.7, 11.2, 11.1, 11.1, 11.3, 11.4, 11.2, 11.1,~
## $ ManufacturingProcess29 <dbl> 21.0, 21.4, 21.3, 21.3, 21.6, 21.7, 21.2, 21.2,~
## $ ManufacturingProcess30 <dbl> 9.9, 9.9, 9.4, 9.4, 9.0, 10.1, 11.2, 10.9, 10.5~
## $ ManufacturingProcess31 <dbl> 69.1, 68.7, 69.3, 69.3, 69.4, 68.2, 67.6, 67.9,~
## $ ManufacturingProcess32 <dbl> 156, 169, 173, 171, 171, 173, 159, 161, 160, 16~
## $ ManufacturingProcess33 <dbl> 66, 66, 66, 68, 70, 70, 65, 65, 65, 66, 67, 67,~
## $ ManufacturingProcess34 <dbl> 2.4, 2.6, 2.6, 2.5, 2.5, 2.5, 2.5, 2.5, 2.5, 2.~
## $ ManufacturingProcess35 <dbl> 486, 508, 509, 496, 468, 490, 475, 478, 491, 48~
## $ ManufacturingProcess36 <dbl> 0.019, 0.019, 0.018, 0.018, 0.017, 0.018, 0.019~
## $ ManufacturingProcess37 <dbl> 0.5, 2.0, 0.7, 1.2, 0.2, 0.4, 0.8, 1.0, 1.2, 1.~
## $ ManufacturingProcess38 <dbl> 3, 2, 2, 2, 2, 2, 2, 2, 3, 3, 2, 3, 3, 3, 3, 3,~
## $ ManufacturingProcess39 <dbl> 7.2, 7.2, 7.2, 7.2, 7.3, 7.2, 7.3, 7.3, 7.4, 7.~
## $ ManufacturingProcess40 <dbl> NA, 0.1, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0~
## $ ManufacturingProcess41 <dbl> NA, 0.15, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0~
## $ ManufacturingProcess42 <dbl> 11.6, 11.1, 12.0, 10.6, 11.0, 11.5, 11.7, 11.4,~
## $ ManufacturingProcess43 <dbl> 3.0, 0.9, 1.0, 1.1, 1.1, 2.2, 0.7, 0.8, 0.9, 0.~
## $ ManufacturingProcess44 <dbl> 1.8, 1.9, 1.8, 1.8, 1.7, 1.8, 2.0, 2.0, 1.9, 1.~
## $ ManufacturingProcess45 <dbl> 2.4, 2.2, 2.3, 2.1, 2.1, 2.0, 2.2, 2.2, 2.1, 2.~
```

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.

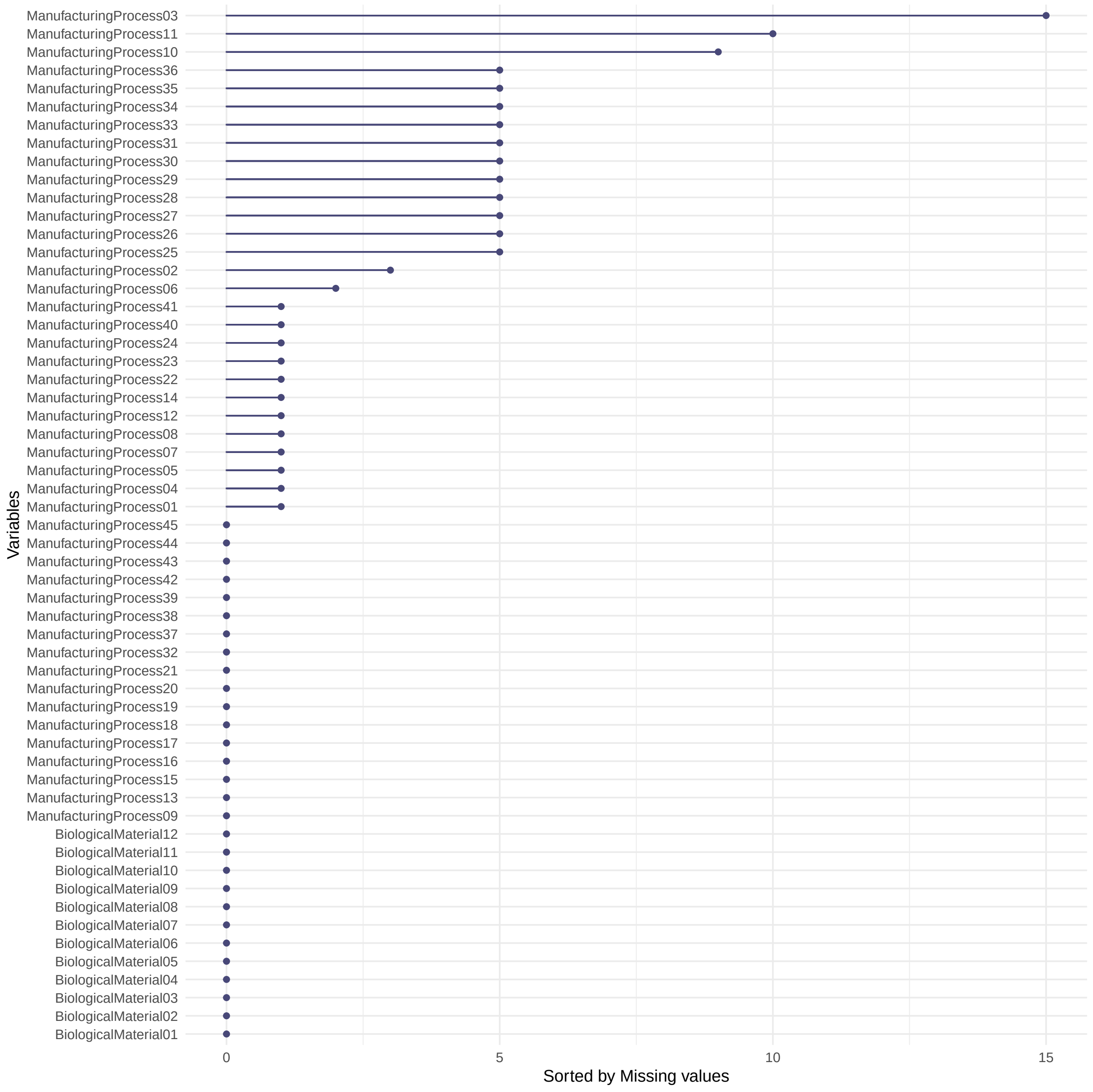
We will first see all the variables having any of the missing values. We have used below complete.cases() function to find the the missing values.

```
# columns having missing values
colnames(ChemicalManufacturingProcess)[!complete.cases(t(ChemicalManufacturingProcess))]
```

```
##  [1] "ManufacturingProcess01" "ManufacturingProcess02" "ManufacturingProcess03"
##  [4] "ManufacturingProcess04" "ManufacturingProcess05" "ManufacturingProcess06"
##  [7] "ManufacturingProcess07" "ManufacturingProcess08" "ManufacturingProcess10"
## [10] "ManufacturingProcess11" "ManufacturingProcess12" "ManufacturingProcess14"
## [13] "ManufacturingProcess22" "ManufacturingProcess23" "ManufacturingProcess24"
## [16] "ManufacturingProcess25" "ManufacturingProcess26" "ManufacturingProcess27"
## [19] "ManufacturingProcess28" "ManufacturingProcess29" "ManufacturingProcess30"
## [22] "ManufacturingProcess31" "ManufacturingProcess33" "ManufacturingProcess34"
## [25] "ManufacturingProcess35" "ManufacturingProcess36" "ManufacturingProcess40"
## [28] "ManufacturingProcess41"
```

So there are 28 columns having missing values. Here is the plot for missing values of all the predictors.

```
gg_miss_var(ChemicalManufacturingProcess[,-c(1)]) + labs(y = "Sorted by Missing values")
```



We will next use preProcess() method to impute the missing values using knnImpute (K nearest neighbor).

```
pre.proc <- preProcess(ChemicalManufacturingProcess[,c(-1)], method = "knnImpute")
chem_df <- predict(pre.proc, ChemicalManufacturingProcess[,c(-1)])
```

```
# columns having missing values
colnames(chem_df)[!complete.cases(t(chem_df))]
```

```
## character(0)
```

We will first filter out the predictors that have low frequencies using the `nearZeroVar` function from the caret package. After applying this function we see 1 column is removed and 56 predictors are left for modeling.

```
chem.remove.pred <- nearZeroVar(chem_df)
chem_df <- chem_df[,-chem.remove.pred]
length(chem.remove.pred) %>% paste('columns are removed. ', dim(chem_df)[2], ' predictors are left for m
## [1] "1 columns are removed.  56  predictors are left for modeling."
```

We will now look into pairwise correlation above 0.90 and remove the predictors having correlation with cutoff 0.90.

```
chem.corr.90 <- findCorrelation(cor(chem_df), cutoff=0.90)
chem_df <- chem_df[,-chem.corr.90]
length(chem.corr.90) %>% paste('columns having correlation 0.90 or more are removed. ', dim(chem_df)[2]
```

```
## [1] "10 columns having correlation 0.90 or more are removed.  46  predictors are left for modeling."
```

Next step is to split the data in training and testing set. We reserve 70% for training and 30% for testing. After split we will fit elastic net model.

```
set.seed(786)

pre.proc <- preProcess(chem_df, method = c("center", "scale"))
chem_df <- predict(pre.proc, chem_df)

# partition
chem.part <- createDataPartition(ChemicalManufacturingProcess$Yield, p=0.80, list = FALSE)

# predictor
X.train <- chem_df[chem.part,]
X.test <- chem_df[-chem.part,]

# response
y.train <- ChemicalManufacturingProcess$Yield[chem.part]
y.test <- ChemicalManufacturingProcess$Yield[-chem.part]
```

Tree-based models

Single tree

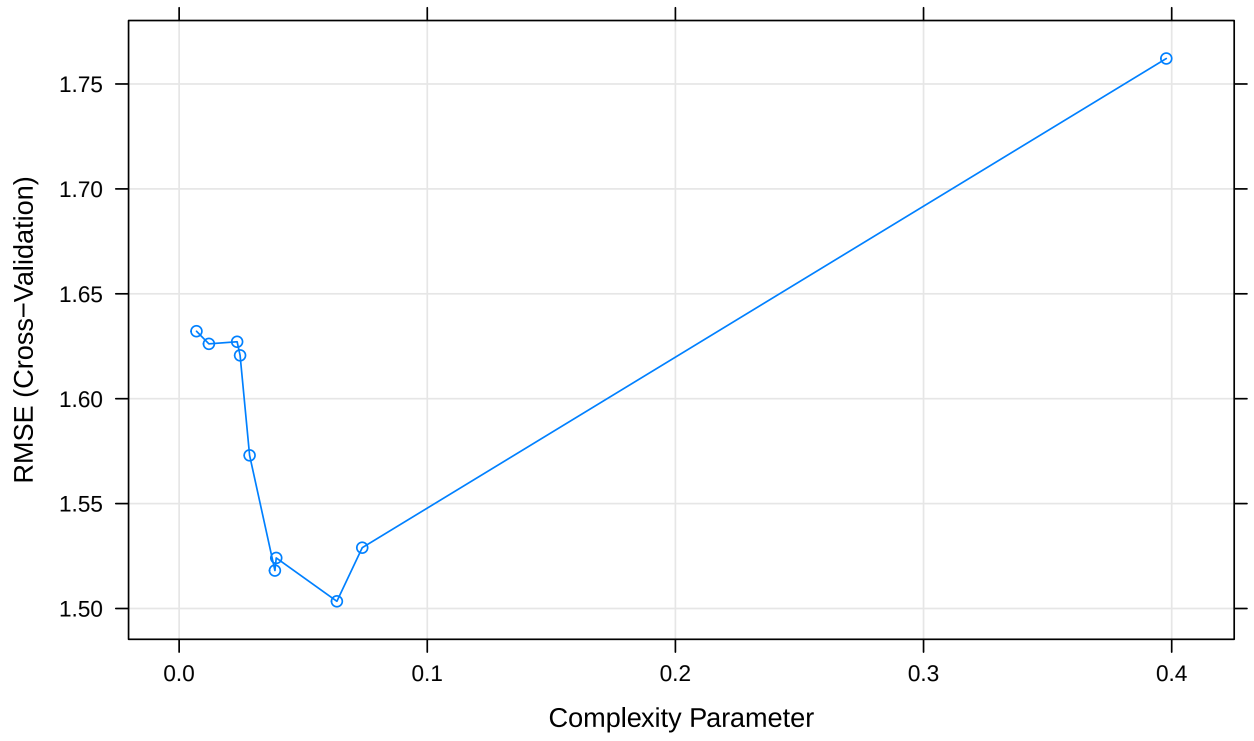
```
set.seed(317)

singletree.model <- train(x=X.train,
                          y=y.train,
                          method = "rpart",
                          tuneLength = 10,
                          trControl = trainControl(method = "cv"))

singletree.model
```

```
## CART
##
## 144 samples
##  46 predictor
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 130, 129, 130, 130, 130, 130, ...
## Resampling results across tuning parameters:
##
##   cp           RMSE      Rsquared   MAE
##   0.006981015  1.632161  0.3699470  1.259034
##   0.011992974  1.626144  0.3696324  1.266241
##   0.023373738  1.627136  0.3618059  1.292127
##   0.024563839  1.620649  0.3674781  1.284892
##   0.028390214  1.572989  0.3833939  1.240389
##   0.038586639  1.518084  0.4134392  1.189689
##   0.039139242  1.524097  0.4104859  1.201891
##   0.063569237  1.503427  0.4261186  1.160598
##   0.073785212  1.528985  0.3968948  1.183857
##   0.397829889  1.762144  0.3155174  1.393891
##
## RMSE was used to select the optimal model using the smallest value.
## The final value used for the model was cp = 0.06356924.
```

```
singletree.model$bestTune
```

```
##           cp
## 8 0.06356924
```

```
# plot RMSE
plot(singletree.model)
```



```
data.frame(Rsquared=singletree.model[["results"]][["Rsquared"]][as.numeric(rownames(singletree.model$bes
           RMSE=singletree.model[["results"]][["RMSE"]][as.numeric(rownames(singletree.model$bestTune))]
```

```
##    Rsquared     RMSE
## 1 0.4261186 1.503427
```

RMSE was used to select the optimal model using the smallest value. The final value used for the model was cp = 0.0636 which results the R^2 as 0.43 and RMSE as 1.50.

Random Forest

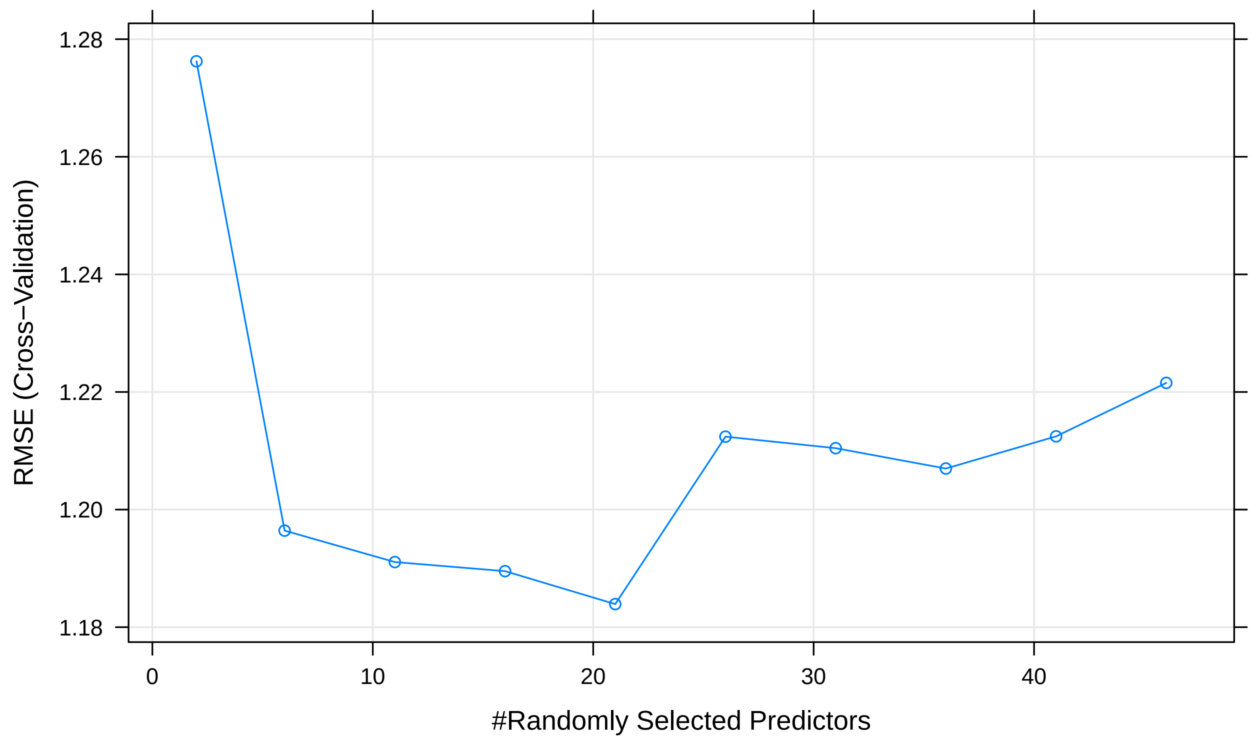
```r
set.seed(317)

randfrst.model <- train(x=X.train,
                        y=y.train,
                        method = "rf",
                        tuneLength = 10,
                        trControl = trainControl(method = "cv"))

randfrst.model
```

```
## Random Forest
##
## 144 samples
##  46 predictor
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 130, 129, 130, 130, 130, 130, ...
## Resampling results across tuning parameters:
##
##   mtry  RMSE      Rsquared   MAE
##    2    1.276239  0.6476209  1.0233583
##    6    1.196421  0.6602487  0.9339669
##   11    1.191080  0.6497139  0.9129452
##   16    1.189529  0.6467315  0.9056901
##   21    1.183924  0.6485636  0.8873649
##   26    1.212404  0.6209487  0.9079482
##   31    1.210441  0.6204198  0.9031250
##   36    1.206981  0.6206485  0.9049780
##   41    1.212455  0.6210801  0.9049894
##   46    1.221547  0.6108959  0.9150768
##
## RMSE was used to select the optimal model using the smallest value.
## The final value used for the model was mtry = 21.
```

```r
randfrst.model$bestTune
```

```
##   mtry
## 5   21
```

```r
# plot RMSE
plot(randfrst.model)
```



```
data.frame(Rsquared=randfrst.model[["results"]][["Rsquared"]][as.numeric(rownames(randfrst.model$bestTun
           RMSE=randfrst.model[["results"]][["RMSE"]][as.numeric(rownames(randfrst.model$bestTune))])
```

```
##    Rsquared     RMSE
## 1 0.6485636 1.183924
```

RMSE was used to select the optimal model using the smallest value. The final value used for the model was mtry = 21. The best tuned model produces the R^2 as 0.65 and RMSE as 1.18.

Boosted tree

```
set.seed(317)

# boosting regression trees via stochastic gradient boosting machines

gbmGrid <- expand.grid(interaction.depth = seq(1, 7, by = 2),
                       n.trees = seq(100, 1000, by = 50),
                       shrinkage = 0.1,
                       n.minobsinnode = 5)

gbm.model <- train(x=X.train,
                   y=y.train,
                   method = "gbm",
                   tuneGrid = gbmGrid,
                   trControl = trainControl(method = "cv"),
                   verbose = FALSE)

gbm.model
```

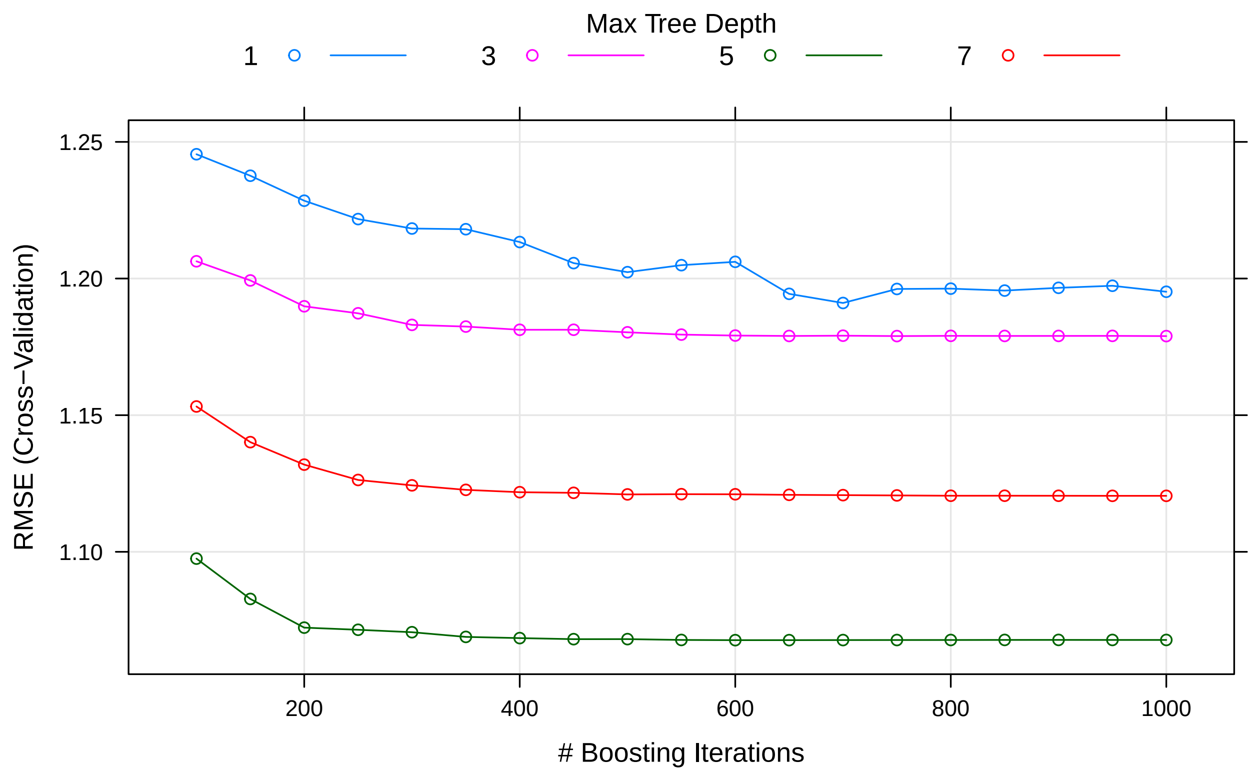
```
## Stochastic Gradient Boosting
##
```

```
## 144 samples
##  46 predictor
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 130, 129, 130, 130, 130, 130, ...
## Resampling results across tuning parameters:
##
##   interaction.depth  n.trees  RMSE      Rsquared   MAE
##   1                   100     1.245481  0.5880556  0.9705447
##   1                   150     1.237622  0.6008957  0.9759223
##   1                   200     1.228476  0.6052772  0.9641847
##   1                   250     1.221752  0.6166068  0.9540344
##   1                   300     1.218302  0.6229132  0.9498020
##   1                   350     1.218046  0.6237690  0.9526492
##   1                   400     1.213357  0.6256700  0.9447584
##   1                   450     1.205642  0.6308861  0.9281967
##   1                   500     1.202328  0.6348055  0.9276323
##   1                   550     1.204889  0.6355892  0.9319917
##   1                   600     1.206117  0.6355583  0.9276326
##   1                   650     1.194399  0.6396886  0.9148295
##   1                   700     1.191048  0.6422960  0.9168934
##   1                   750     1.196183  0.6398491  0.9198499
##   1                   800     1.196306  0.6407975  0.9191642
##   1                   850     1.195592  0.6410272  0.9173856
##   1                   900     1.196600  0.6421816  0.9183799
##   1                   950     1.197354  0.6409147  0.9201573
##   1                  1000     1.195162  0.6424359  0.9163325
##   3                   100     1.206314  0.6124249  0.9472317
##   3                   150     1.199297  0.6190014  0.9407936
##   3                   200     1.189840  0.6250466  0.9296289
##   3                   250     1.187276  0.6263765  0.9227402
##   3                   300     1.183037  0.6288432  0.9183676
##   3                   350     1.182423  0.6284030  0.9162822
##   3                   400     1.181248  0.6285164  0.9138591
##   3                   450     1.181256  0.6284298  0.9133442
##   3                   500     1.180325  0.6290453  0.9128685
##   3                   550     1.179481  0.6293308  0.9121291
##   3                   600     1.179134  0.6295049  0.9117780
##   3                   650     1.178986  0.6296207  0.9114500
##   3                   700     1.179089  0.6295915  0.9115069
##   3                   750     1.178941  0.6297018  0.9114119
##   3                   800     1.179029  0.6296324  0.9115384
##   3                   850     1.178997  0.6296409  0.9114904
##   3                   900     1.179007  0.6296601  0.9115480
##   3                   950     1.179023  0.6296633  0.9115578
##   3                  1000     1.178937  0.6297117  0.9115178
##   5                   100     1.097512  0.6843965  0.8515388
##   5                   150     1.082756  0.6898860  0.8395575
##   5                   200     1.072261  0.6941257  0.8326550
##   5                   250     1.071477  0.6950774  0.8317633
##   5                   300     1.070587  0.6959038  0.8304226
##   5                   350     1.068868  0.6969003  0.8284091
##   5                   400     1.068432  0.6969812  0.8280019
##   5                   450      1.068044  0.6971940  0.8279726
##   5                   500      1.068057  0.6972513  0.8277665
##   5                   550      1.067750  0.6974572  0.8274557
##   5                   600      1.067675  0.6974953  0.8273943
##   5                   650      1.067687  0.6975075  0.8274587
##   5                   700      1.067714  0.6974893  0.8274819
##   5                   750      1.067728  0.6974924  0.8275003
##   5                   800      1.067734  0.6974882  0.8275076
##   5                   850      1.067764  0.6974703  0.8275328
##   5                   900      1.067771  0.6974656  0.8275466
##   5                   950      1.067755  0.6974762  0.8275388
##   5                  1000      1.067753  0.6974788  0.8275391
##   7                   100      1.153181  0.6516039  0.8865860
##   7                   150      1.140137  0.6586639  0.8839726
##   7                   200      1.131911  0.6639755  0.8787452
##   7                   250      1.126289  0.6672349  0.8749078
##   7                   300      1.124328  0.6682094  0.8746135
##   7                   350      1.122680  0.6692330  0.8738965
##   7                   400      1.121815  0.6696657  0.8737773
##   7                   450      1.121583  0.6698267  0.8736295
##   7                   500      1.121007  0.6701581  0.8734428
##   7                   550      1.121088  0.6701483  0.8738288
##   7                   600      1.121060  0.6701446  0.8738345
##   7                   650      1.120853  0.6702578  0.8737752
##   7                   700      1.120740  0.6703230  0.8737045
##   7                   750      1.120638  0.6703815  0.8736995
##   7                   800      1.120524  0.6704295  0.8736275
##   7                   850      1.120531  0.6704285  0.8736628
##   7                   900      1.120512  0.6704485  0.8736670
##   7                   950      1.120490  0.6704581  0.8736549
##   7                  1000      1.120489  0.6704580  0.8736659
##
## Tuning parameter 'shrinkage' was held constant at a value of 0.1
##
## Tuning parameter 'n.minobsinnode' was held constant at a value of 5
## RMSE was used to select the optimal model using the smallest value.
## The final values used for the model were n.trees = 600, interaction.depth =
##  5, shrinkage = 0.1 and n.minobsinnode = 5.
```

```
gbm.model$bestTune
```

```
##    n.trees interaction.depth shrinkage n.minobsinnode
## 49     600                 5       0.1              5
```

```
# plot RMSE
plot(gbm.model)
```



```
data.frame(Rsquared=gbm.model[["results"]][["Rsquared"]][as.numeric(rownames(gbm.model$bestTune))],
           RMSE=gbm.model[["results"]][["RMSE"]][as.numeric(rownames(gbm.model$bestTune))])
```

```
##   Rsquared     RMSE
## 1 0.642296 1.191048
```

RMSE was used to select the optimal model using the smallest value. The final values used for the model were n.trees = 600, interaction.depth = 5, shrinkage = 0.1 and n.minobsinnode = 5. Tuning parameters 'shrinkage' and 'n.minobsinnode' were held constant at 0.1 and 5 respectively. The best tuned model produces the R^2 as 0.64 and RMSE as 1.19.

Cubist

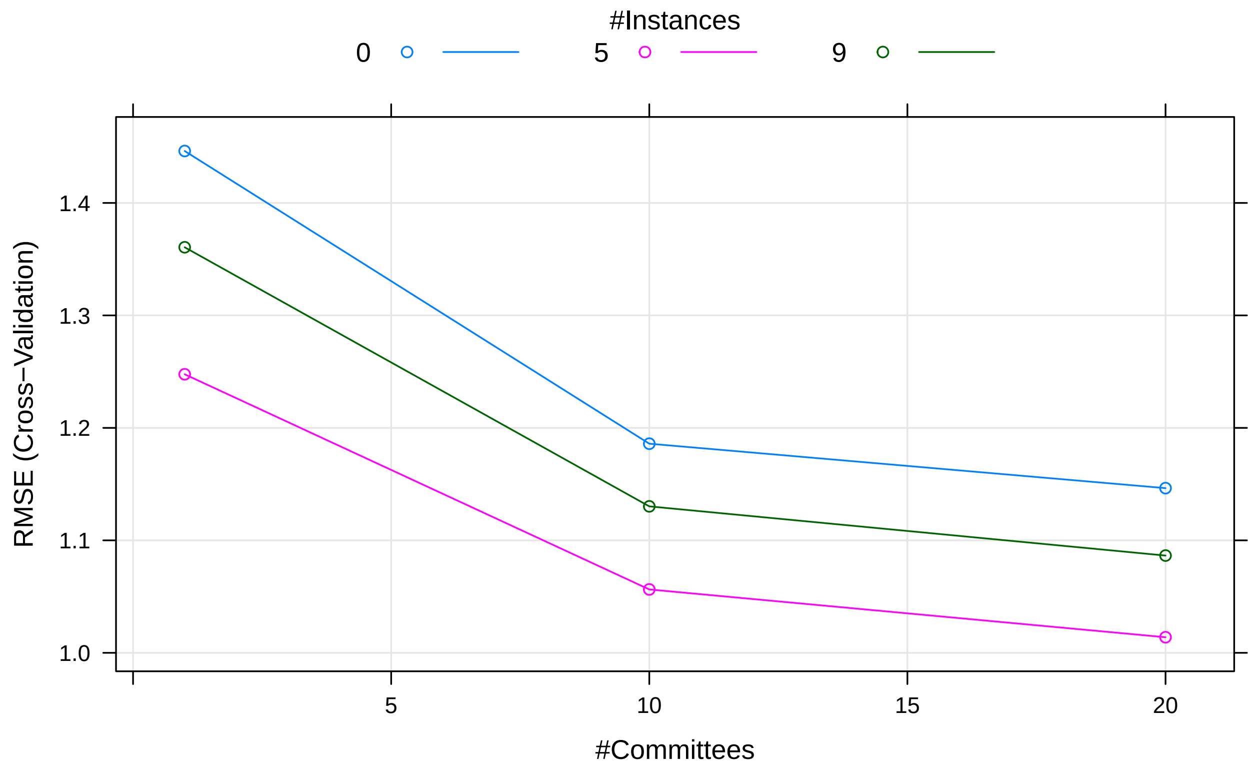
```
set.seed(317)

cubist.model <- train(x=X.train,
                          y=y.train,
                          method = "cubist",
                          tuneLength = 10,
                          trControl = trainControl(method = "cv"))

cubist.model
```

```
## Cubist
##
## 144 samples
##  46 predictor
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 130, 129, 130, 130, 130, 130, ...
## Resampling results across tuning parameters:
##
##   committees  neighbors  RMSE      Rsquared   MAE
##    1          0          1.446175  0.5208671  1.0971163
##    1          5          1.247655  0.6369449  0.9430697
##    1          9          1.360571  0.5766159  1.0101461
##   10          0          1.185945  0.6244915  0.9039320
##   10          5          1.056348  0.7040391  0.8110469
##   10          9          1.130220  0.6573072  0.8566637
##   20          0          1.146430  0.6357934  0.9033064
##   20          5          1.013859  0.7134196  0.7958342
##   20          9          1.086474  0.6681592  0.8417684
##
## RMSE was used to select the optimal model using the smallest value.
## The final values used for the model were committees = 20 and neighbors = 5.
```

```
cubist.model$bestTune
```

```
##   committees neighbors
## 8         20         5
```

```
# plot RMSE
plot(cubist.model)
```



```
data.frame(Rsquared=cubist.model[["results"]][["Rsquared"]][as.numeric(rownames(cubist.model$bestTune))]
           RMSE=cubist.model[["results"]][["RMSE"]][as.numeric(rownames(cubist.model$bestTune))])
```

```
##    Rsquared     RMSE
## 1 0.7134196 1.013859
```

RMSE was used to select the optimal model using the smallest value. The final values used for the model were committees = 20 and neighbors = 5. The best tuned model produces the R^2 as 0.71 and RMSE as 1.01. Among all we see Cubist model has the best R^2 value on training data.

(a)

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

Optimal resampling

```
set.seed(317)
summary(resamples(list(SingTree=singletree.model, RandFrst=randfrst.model, Boosting=gbm.model, Cubist=cu
```

```
##
## Call:
## summary.resamples(object = resamples(list(SingTree = singletree.model,
##  RandFrst = randfrst.model, Boosting = gbm.model, Cubist = cubist.model)))
##
## Models: SingTree, RandFrst, Boosting, Cubist
## Number of resamples: 10
##
## MAE
##               Min.   1st Qu.    Median      Mean   3rd Qu.     Max. NA's
## SingTree 0.8849405 0.9998667 1.2269072 1.1605976 1.3006355 1.345884    0
## RandFrst 0.6318366 0.7986381 0.9357105 0.8873649 0.9912005 1.078099    0
## Boosting 0.5408440 0.7622352 0.8426787 0.8273943 0.9030208 1.195998    0
## Cubist   0.5316364 0.6814313 0.8311277 0.7958342 0.9238664 1.016086    0
##
## RMSE
##               Min.   1st Qu.   Median     Mean  3rd Qu.     Max. NA's
## SingTree 1.1013993 1.3380246 1.563225 1.503427 1.646477 1.786671    0
## RandFrst 0.7636308 1.0817320 1.238111 1.183924 1.341517 1.418591    0
## Boosting 0.6738099 0.9737539 1.059665 1.067675 1.279173 1.352110    0
## Cubist   0.6170292 0.8172101 1.017437 1.013859 1.216264 1.416342    0
##
## Rsquared
##               Min.   1st Qu.    Median      Mean   3rd Qu.      Max. NA's
## SingTree 0.1798730 0.2862052 0.4971034 0.4261186 0.5190161 0.7061185    0
## RandFrst 0.4759452 0.5567358 0.6571640 0.6485636 0.7422394 0.8380936    0
## Boosting 0.5353858 0.5929949 0.6727539 0.6974953 0.8103443 0.8738694    0
## Cubist   0.5544881 0.6303457 0.7180886 0.7134196 0.8048236 0.8769379    0
```

Test set performance

```
set.seed(317)
sngpred <- predict(singletree.model, newdata = X.test)
rfpred <- predict(randfrst.model, newdata = X.test)
gbmpred <- predict(gbm.model, newdata = X.test)
cubpred <- predict(cubist.model, newdata = X.test)


data.frame(rbind(SingTree=postResample(pred=sngpred,obs = y.test),
                 RandFrst=postResample(pred=rfpred,obs = y.test),
                 Boosting=postResample(pred=gbmpred,obs = y.test),
                 Cubist=postResample(pred=cubpred,obs = y.test)))
```

```
##                RMSE  Rsquared       MAE
## SingTree 1.2053977 0.4710258 1.0119301
## RandFrst 0.8686833 0.7146764 0.7112018
## Boosting 0.9331870 0.7080137 0.7611015
## Cubist   0.8071695 0.7599164 0.6023495
```

Seeing the results above, it is evident that the Cubist model predicted the test responses with best accuracy R^2=0.76 and RMSE=0.81

(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?

```
varImp(cubist.model)
```

```
## cubist variable importance
##
##   only 20 most important variables shown (out of 46)
##
##                        Overall
## ManufacturingProcess32  100.00
## ManufacturingProcess17   63.74
## BiologicalMaterial06     63.74
## ManufacturingProcess13   59.34
## ManufacturingProcess09   46.15
## ManufacturingProcess04   29.67
## ManufacturingProcess02   28.57
## BiologicalMaterial03     26.37
## ManufacturingProcess33   26.37
## BiologicalMaterial11     21.98
## ManufacturingProcess39   20.88
## ManufacturingProcess28   18.68
## BiologicalMaterial08     15.38
## ManufacturingProcess15   13.19
## ManufacturingProcess45   13.19
## BiologicalMaterial05     13.19
## ManufacturingProcess20   12.09
## ManufacturingProcess37   10.99
## BiologicalMaterial09      9.89
## BiologicalMaterial01      9.89
```

We can see here too that ManufacturingProcess variables dominates the list but ranks seem different between linear and non linear models.

It was stated earlier that svm model performed best among nonlinear regression models.

```
set.seed(317)
# tune svm among non linear models
svmmodel <- train(X.train,
                y.train,
                method = "svmRadial",
                preProcess = c("center","scale"),
                tuneLength = 10,
                trControl = trainControl(method = "cv"))

varImp(svmmodel)
```

```
## loess r-squared variable importance
##
##   only 20 most important variables shown (out of 46)
##
##                         Overall
## ManufacturingProcess32  100.00
## ManufacturingProcess13   98.56
## BiologicalMaterial06     88.33
## ManufacturingProcess17   84.64
## ManufacturingProcess36   80.34
## ManufacturingProcess09   80.21
## BiologicalMaterial03     76.25
## ManufacturingProcess06   66.88
## ManufacturingProcess11   53.71
## BiologicalMaterial11     50.67
## ManufacturingProcess33   50.38
## ManufacturingProcess02   46.10
## ManufacturingProcess30   45.04
## BiologicalMaterial08     40.42
## BiologicalMaterial09     39.87
## ManufacturingProcess12   36.23
## BiologicalMaterial01     36.10
## ManufacturingProcess15   31.14
## ManufacturingProcess01   26.59
## ManufacturingProcess26   25.66
```

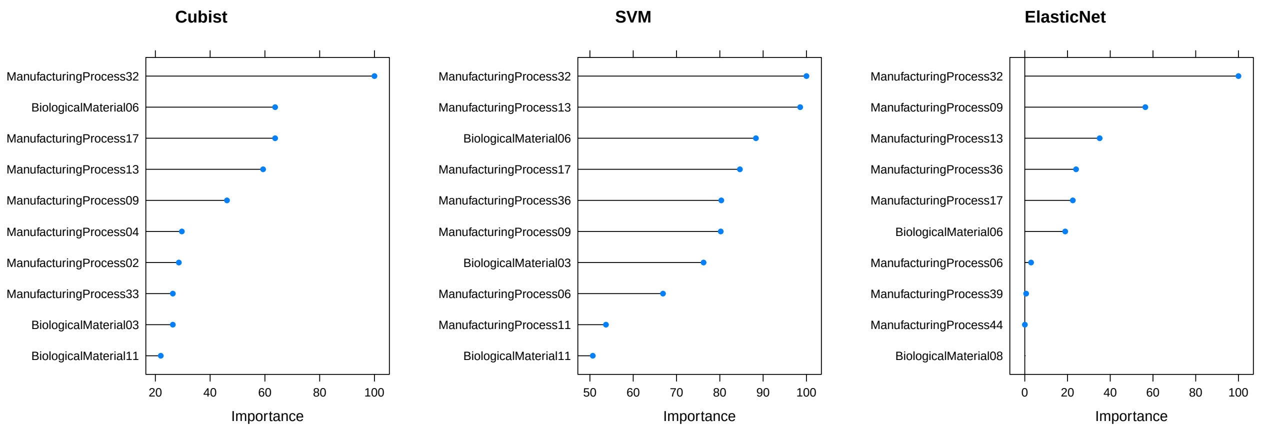
It was stated earlier that elasticnet model that best fitted the data among linear models.

```
set.seed(317)
# tune elastic net model among linear models
chem.enet.fit <- train(x=X.train,
                       y=y.train,
                       method="glmnet",
                       metric="Rsquared",
                       trControl=trainControl(method = "cv",number=10),
                       tuneLength = 5
                 )


varImp(chem.enet.fit)
```

```
## glmnet variable importance
##
##   only 20 most important variables shown (out of 46)
##
##                           Overall
## ManufacturingProcess32 100.00000
## ManufacturingProcess09  56.42573
## ManufacturingProcess13  35.05073
## ManufacturingProcess36  24.00339
## ManufacturingProcess17  22.47464
## BiologicalMaterial06    18.92913
## ManufacturingProcess06   2.94998
## ManufacturingProcess39   0.64278
## ManufacturingProcess44   0.04296
## ManufacturingProcess01   0.00000
## ManufacturingProcess24   0.00000
## ManufacturingProcess11   0.00000
## ManufacturingProcess16   0.00000
## ManufacturingProcess22   0.00000
## ManufacturingProcess21   0.00000
## BiologicalMaterial01     0.00000
## ManufacturingProcess30   0.00000
## ManufacturingProcess38   0.00000
## ManufacturingProcess34   0.00000
## ManufacturingProcess10   0.00000
```

From the importance predictors above from Treebased: Cubist, linear: elasticnet and nonlinear: svm, we can see that rank ManufacturingProcess32 remains on top. Among all, it is evident that ManufacturingProcess predictors dominate the list. Below are the plots of top 10 Imp predictors from these 3 models.

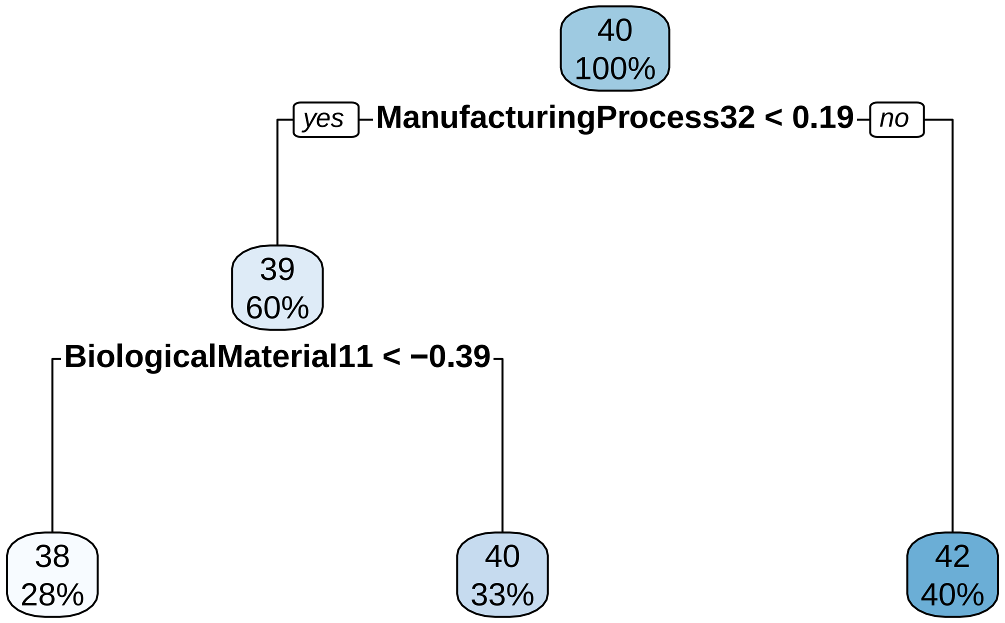
```
plt1 <- plot(varImp(cubist.model), top=10, main="Cubist")
plt2 <- plot(varImp(svmmodel), top=10, main="SVM")
plt3 <- plot(varImp(chem.enet.fit), top=10, main="ElasticNet")
gridExtra::grid.arrange(plt1, plt2, plt3, ncol=3)
```



(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?

```
rpart.plot(singletree.model$finalModel)
```



The plot above is for optimal single tree model and it depicts that split begins with `ManufacturingProcess32` and if it is less than 0.19, the yield will be 39 else the yield would be 42. This view of the data does provide additional knowledge about the process predictors and their relationship with yield; the higher value of `ManufacturingProcess32` leads to higher yield and vice versa.