Data 624: Predictive Analytics HW:2

Group 2: Alice Friedman, Kayleah Griffen, Michael Ippolito, Josh Iden

2023-06-19

Table of Contents

## Introduction

This homework assignment includes problems from:

Kuhn & Johnson. “Applied Predictive Modeling”

This accompanies readings from KJ Chapters 6, 7, and 8.

Additionally readings on recommender systems along with a recommender system problem was assigned by the professor.

## Week 4: KJ 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 the 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:

> library(AppliedPredictiveModeling)  
> data(chemicalManufacturing)

**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.**

# Load data  
data(ChemicalManufacturingProcess)

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

We’ll first generate a summary to get a first look at the data and to show missing values, then we’ll get a count of missing values per variable.

# Show missing value counts  
dfchem <- ChemicalManufacturingProcess # To avoid typing this many letters  
df\_na <- data.frame(Predictor = colnames(dfchem), Missing=colSums(is.na(dfchem))) %>%  
 filter(Missing > 0) %>%  
 arrange(desc(Missing))   
  
flextable(df\_na) |>  
 width(width = 2) |>  
 fontsize(size = 10) |>  
 line\_spacing(space = 0.5) |>  
 hline(part = "all")

| Predictor | Missing |
| --- | --- |
| ManufacturingProcess03 | 15 |
| ManufacturingProcess11 | 10 |
| ManufacturingProcess10 | 9 |
| ManufacturingProcess25 | 5 |
| ManufacturingProcess26 | 5 |
| ManufacturingProcess27 | 5 |
| ManufacturingProcess28 | 5 |
| ManufacturingProcess29 | 5 |
| ManufacturingProcess30 | 5 |
| ManufacturingProcess31 | 5 |
| ManufacturingProcess33 | 5 |
| ManufacturingProcess34 | 5 |
| ManufacturingProcess35 | 5 |
| ManufacturingProcess36 | 5 |
| ManufacturingProcess02 | 3 |
| ManufacturingProcess06 | 2 |
| ManufacturingProcess01 | 1 |
| ManufacturingProcess04 | 1 |
| ManufacturingProcess05 | 1 |
| ManufacturingProcess07 | 1 |
| ManufacturingProcess08 | 1 |
| ManufacturingProcess12 | 1 |
| ManufacturingProcess14 | 1 |
| ManufacturingProcess22 | 1 |
| ManufacturingProcess23 | 1 |
| ManufacturingProcess24 | 1 |
| ManufacturingProcess40 | 1 |
| ManufacturingProcess41 | 1 |

Overall, the missing data does not appear to be structurally missing, there does not appear to be a pattern, and the missing data is not informatively missing. One option to impute is called Multiple Imputation by Chained Equation, or MICE, and performs multiple regression over the sample data and takes averages. Another option is called KNN-imputation which finds the “nearest neighbors” to a missing value and uses a weighted average to fill them. KNN is suitable if the dataset is small and has mostly continuous data.

dfimp <- knnImputation(ChemicalManufacturingProcess,k=3)  
  
# Verify no more missing values exist  
print(paste0('Missing values after imputation: ', sum(is.na(dfimp))))

## [1] "Missing values after imputation: 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?**

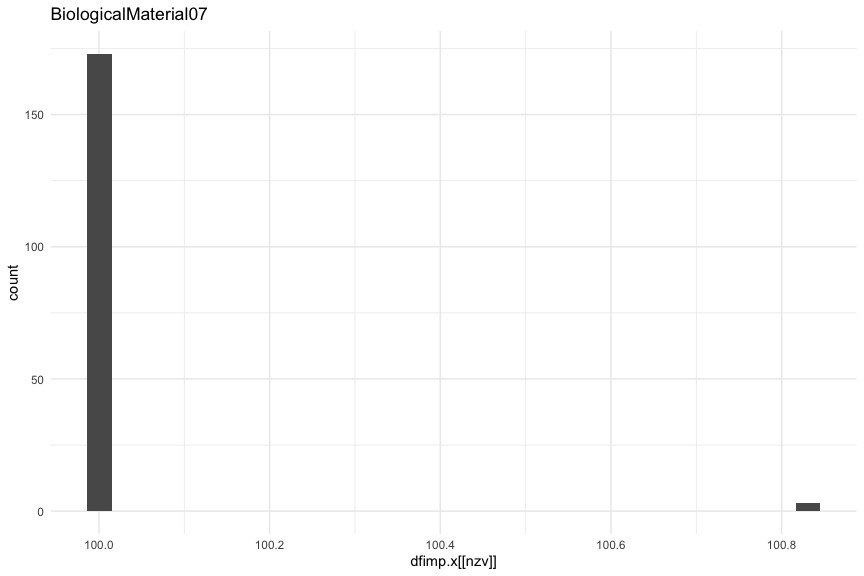
First we will do exploratory data analysis and do any pre-processing necessary.

We can check the distribution of the predictors and see what have near-zero variance and remove these from the predictor set.

dfimp.x <- dfimp |> dplyr::select(-Yield)  
nzv <- nearZeroVar(dfimp.x)  
paste0("Inds near zero variance to be removed: ", colnames(dfimp.x[nzv]))

## [1] "Inds near zero variance to be removed: BiologicalMaterial07"

dfimp.x |> ggplot(aes(x = dfimp.x[[nzv]])) + geom\_histogram(bins = 30) + ggtitle(colnames(dfimp.x[nzv]))



dfimp.x <- dfimp.x[-nzv]

Next we can inspect the outliers.

# check for outliers  
  
df\_outliers <- data.frame(predictor = c(), outliers = c())  
  
for (i in 1:length(dfimp.x)){  
 data <- dfimp.x[[i]]  
 quartiles <- quantile(data, probs=c(.25, .75), na.rm = FALSE)  
 IQR <- IQR(data)  
 Lower <- quartiles[1] - 1.5\*IQR  
 Upper <- quartiles[2] + 1.5\*IQR  
 outliers <- which(data < Lower | data > Upper)  
 data[outliers] <- NA  
 # print(paste0(colnames(dfimp.x[i]), " outliers: ", length(outliers)))  
 df\_temp <- data.frame(predictor = colnames(dfimp.x[i]),  
 outliers = length(outliers))  
 df\_outliers <- rbind(df\_outliers, df\_temp)  
}  
  
df\_outliers |>  
 arrange(desc(outliers)) |>  
 flextable() |>  
 width(width = 2) |>  
 fontsize(size = 10) |>  
 line\_spacing(space = 0.5) |>  
 hline(part = "all")

| predictor | outliers |
| --- | --- |
| ManufacturingProcess34 | 53 |
| ManufacturingProcess02 | 35 |
| ManufacturingProcess12 | 33 |
| ManufacturingProcess41 | 32 |
| ManufacturingProcess40 | 31 |
| ManufacturingProcess15 | 20 |
| ManufacturingProcess21 | 17 |
| ManufacturingProcess45 | 17 |
| ManufacturingProcess03 | 14 |
| ManufacturingProcess16 | 14 |
| ManufacturingProcess27 | 14 |
| BiologicalMaterial10 | 11 |
| ManufacturingProcess29 | 11 |
| ManufacturingProcess42 | 11 |
| ManufacturingProcess05 | 10 |
| ManufacturingProcess26 | 10 |
| ManufacturingProcess35 | 10 |
| ManufacturingProcess43 | 10 |
| ManufacturingProcess14 | 9 |
| ManufacturingProcess39 | 9 |
| ManufacturingProcess44 | 9 |
| ManufacturingProcess10 | 8 |
| ManufacturingProcess18 | 8 |
| ManufacturingProcess20 | 8 |
| ManufacturingProcess25 | 7 |
| BiologicalMaterial11 | 6 |
| ManufacturingProcess06 | 6 |
| ManufacturingProcess31 | 6 |
| ManufacturingProcess09 | 5 |
| ManufacturingProcess11 | 5 |
| ManufacturingProcess19 | 5 |
| ManufacturingProcess38 | 5 |
| BiologicalMaterial01 | 4 |
| ManufacturingProcess01 | 4 |
| ManufacturingProcess13 | 4 |
| ManufacturingProcess17 | 4 |
| ManufacturingProcess33 | 4 |
| BiologicalMaterial04 | 3 |
| BiologicalMaterial09 | 3 |
| ManufacturingProcess32 | 3 |
| ManufacturingProcess36 | 3 |
| ManufacturingProcess04 | 2 |
| ManufacturingProcess30 | 2 |
| ManufacturingProcess37 | 2 |
| BiologicalMaterial05 | 1 |
| BiologicalMaterial06 | 1 |
| BiologicalMaterial08 | 1 |
| BiologicalMaterial02 | 0 |
| BiologicalMaterial03 | 0 |
| BiologicalMaterial12 | 0 |
| ManufacturingProcess07 | 0 |
| ManufacturingProcess08 | 0 |
| ManufacturingProcess22 | 0 |
| ManufacturingProcess23 | 0 |
| ManufacturingProcess24 | 0 |
| ManufacturingProcess28 | 0 |

Some of the predictors contain a significant amount - approximately 30% - of outliers. So our modeling technique must be robost to outliers.

Next we can examine the skewness.

skew.vals <- apply(dfimp.x, MARGIN = 2, skewness) |>  
 data.frame() |>  
 rename("skew" = 1) |>  
 filter(skew > 2 | skew < -2)   
  
sprintf("Number of skewed variables: %d",nrow(skew.vals))

## [1] "Number of skewed variables: 19"

df\_skews <- data.frame(Predictor = rownames(skew.vals), Skew = skew.vals$skew)  
  
df\_skews |>  
 arrange(desc(Skew)) |>  
 flextable() |>  
 width(width = 2) |>  
 fontsize(size = 10) |>  
 line\_spacing(space = 0.5) |>  
 hline(part = "all")

| Predictor | Skew |
| --- | --- |
| ManufacturingProcess43 | 9.054875 |
| ManufacturingProcess06 | 3.053842 |
| ManufacturingProcess05 | 2.597509 |
| BiologicalMaterial10 | 2.402378 |
| ManufacturingProcess41 | 2.178365 |
| ManufacturingProcess01 | -3.933573 |
| ManufacturingProcess45 | -4.077941 |
| ManufacturingProcess39 | -4.269121 |
| ManufacturingProcess30 | -4.820423 |
| ManufacturingProcess44 | -4.970355 |
| ManufacturingProcess42 | -5.450008 |
| ManufacturingProcess29 | -10.210674 |
| ManufacturingProcess31 | -11.972154 |
| ManufacturingProcess16 | -12.420225 |
| ManufacturingProcess20 | -12.638327 |
| ManufacturingProcess27 | -12.703893 |
| ManufacturingProcess18 | -12.736138 |
| ManufacturingProcess25 | -12.817192 |
| ManufacturingProcess26 | -12.856303 |

Generally, a skewness between -0.5 and 0.5 indicates a relatively small amount of skewness (0 = perfect symmetry). Any values lower than -2 or greater than 2 are considered skewed. A positive skew indicates a right-skew, while a negative skew indicates a left-skew. We tried several methods to eliminate the skew - in this case because of the negative numbers the log transformation and BoxCox would not work. For this reason we will use the YeoJohnson. We will also center and scale.

trans <- preProcess(dfimp.x, method = c("YeoJohnson", "center","scale"))  
df.x <- predict(trans, dfimp.x)  
  
## TODO: Grid ## printing plist in loop works fine, but fails with grid.arrange because they are all the same  
# plist <- list()  
#   
# for (i in 1:length(df.x)){  
# plist[[i]] <- df.x |> ggplot(aes(x = df.x[[i]])) + geom\_histogram(bins = 30) + xlab(colnames(df.x)[i])  
# }  
#   
# library(gridExtra)  
# n <- length(plist)  
# nCol <- floor(sqrt(n))  
# do.call("grid.arrange", c(plist, ncol=nCol))

We can take another look now at skewness.

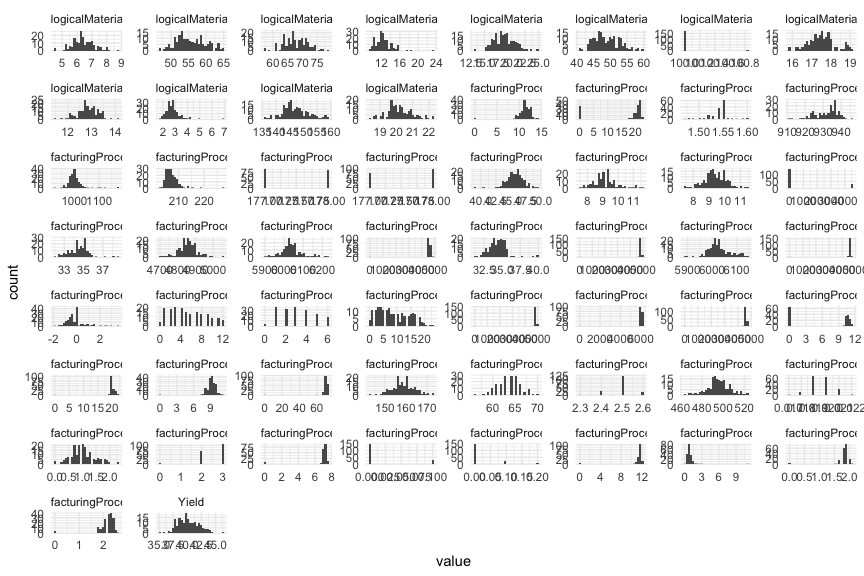
skew.vals <- apply(df.x, MARGIN = 2, skewness) |>  
 data.frame() |>  
 rename("skew" = 1) |>  
 filter(skew > 2 | skew < -2) |>  
 arrange(desc(skew))  
  
sprintf("Number of skewed variables: %d",nrow(skew.vals))

## [1] "Number of skewed variables: 17"

dfchem %>% gather(key, value) %>% ggplot(aes(x=value)) + geom\_histogram() + facet\_wrap(~key, scales = "free")

## `stat\_bin()` using `bins = 30`. Pick better value with `binwidth`.

## Warning: Removed 106 rows containing non-finite values (`stat\_bin()`).

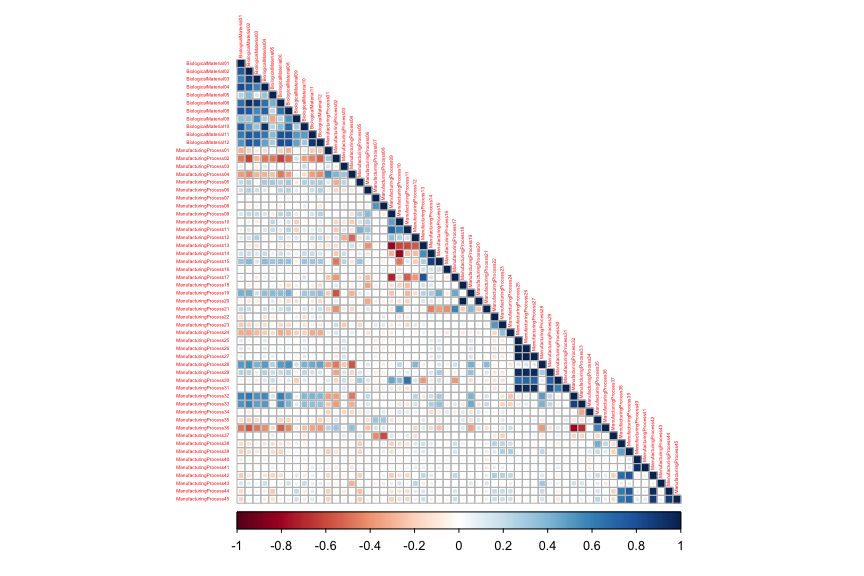


According to the book, when there are a lot of highly correlated predictors manually removing them can become difficult because of complicated relationships between the predictors, therefore models that can handle collinearity should be considered (pg. 111). Additionally, principle component regression which can be used to handle correlated predictors or cases where there are more predictors than observations can lead to principle components that are not related to the response variable (pg.112). As a result of this, partial least squares (pls) is recommended when there are correlated predictors and a linear regression solution is desired (pg. 112). What PLS does is “PLS finds components that maximally summarize the variation of the predictors while simultaneously requiring these components to have maximum correlation with the response” (pg. 114). According to the book, prior to conducting PLS the data should be centered and scaled.

df\_skews <- data.frame(Predictor = rownames(skew.vals), Skew = skew.vals$skew)

df\_skews |> arrange(desc(Skew)) |> flextable() |> width(width = 2) |> fontsize(size = 10) |> line\_spacing(space = 0.5) |> hline(part = “all”)

Interestingly, the "YeoJohnson" transformation did not correct all of the skewnesses.  
  
Now we take a look at a correlation matrix.  
  
  
```r  
dfimp.x |>  
 cor() |>  
 round(2) |>  
 corrplot::corrplot(method = "square",  
 order = "alphabet",  
 tl.cex = 0.3,  
 type = "lower")



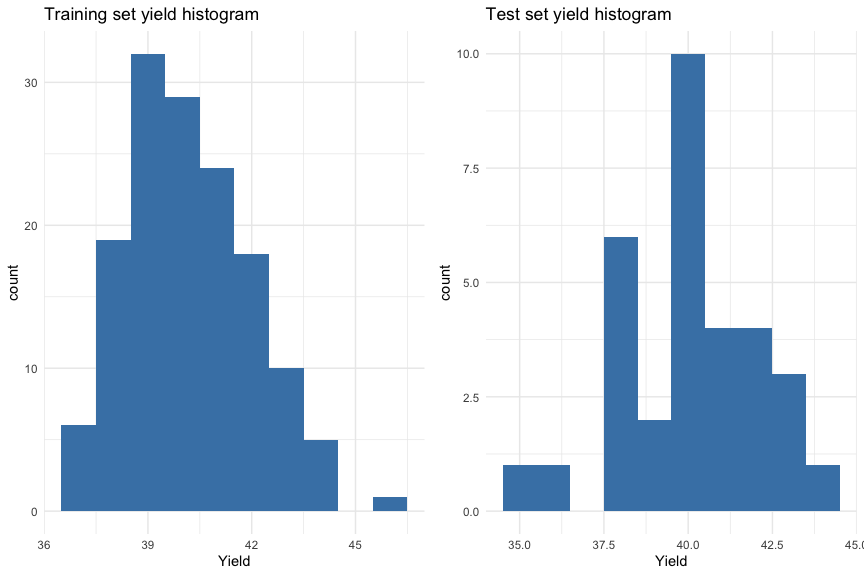
Based on this graphic, we can see that there is a lot of correlation in the predictors. According to the book, when there are a lot of highly correlated predictors manually removing them can become difficult because of complicated relationships between the predictors, therefor models that can handle collinearity should be considered (pg. 111). Additionally, principle component regression which can be used to handle correlated predictors or cases where there are more predictors than observations can lead to principle components that are not related to the response variable (pg.112). As a result of this, partial least squares (pls) is recommended when there are correlated predictors and a linear regression solution is desired (pg. 112). What PLS does is “PLS finds components that maximally summarize the variation of the predictors while simultaneously requiring these components to have maximum correlation with the response” (pg. 114). According to the book, prior to conducting PLS the data should be centered and scaled. Another option that is capable of handling collinearity is the elastic net function which is a combination of the ridge and lasso penalty meaning it is able to regularize and feature select (pg. 127). We will attempt both the PLS and the elastic net and compare them to see what has better results.

Now we will split the data into a training and a test set.

set.seed("09041996")  
  
df <- df.x  
df$Yield <- dfchem$Yield  
  
## Split training and test  
train\_indices <- createDataPartition(df$Yield,  
 p = 0.8,  
 list = FALSE,  
 times = 1)  
  
dftrain <- df[train\_indices,]  
dftest <- df[-train\_indices,]

Next we will examine the distribution of the yield for both the training and the test sets.

# Examine train and test sets  
plot1 <- ggplot(data = dftrain, aes(x=Yield)) + geom\_histogram(fill = "steelblue", binwidth = 1) +   
 ggtitle('Training set yield histogram')   
  
plot2 <- ggplot(data = dftest, aes(x=Yield)) + geom\_histogram(fill = "steelblue", binwidth = 1) +   
 ggtitle('Test set yield histogram')   
  
grid.arrange(plot1, plot2, ncol = 2)



ctrl <- trainControl(method = "cv", number = 10)  
  
data.train.x <- dftrain |> dplyr::select(-Yield)  
data.train.y <- dftrain |> dplyr::select(Yield)  
  
# tune and fit the model with pre processing   
plsTune <- train(data.train.x, data.train.y$Yield,  
 method = "pls",  
 tuneLength = 20,  
 trControl = ctrl)  
  
# plsTune  
  
# elastic net regression  
# enetGrid <- expand.grid(.lambda=c(0, 0.01, 0.1), .fraction=seq(0.05, 1, length=20))  
enetTune <- train(data.train.x, data.train.y$Yield,   
 method='enet',   
 trControl=ctrl)  
# enetTune

optimal.model <- plsTune$bestTune[['ncomp']]  
optimal.rmse <- plsTune$results[['RMSE']][[3]]  
  
cat(sprintf("The optimal number of components to minimize the RMSE is %i\nThe RMSE using %i components is %f.",  
 optimal.model,  
 optimal.model,  
 optimal.rmse))

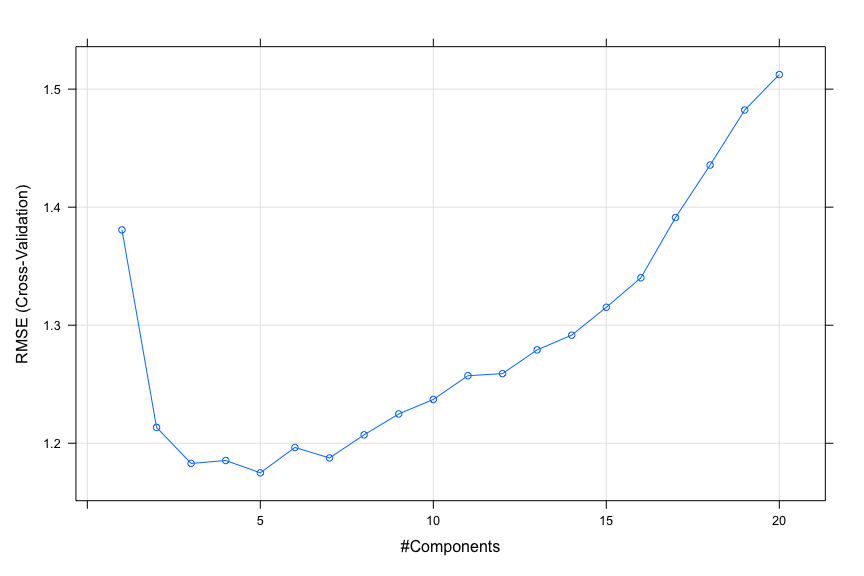
## The optimal number of components to minimize the RMSE is 5  
## The RMSE using 5 components is 1.182888.

optimal.fraction <- enetTune$bestTune[['fraction']]  
optimal.lambda <- enetTune$bestTune[['lambda']]  
optimal.rmse <- enetTune$results[['RMSE']][[1]]  
  
cat(sprintf("The optimal fraction to minimize the RMSE is %f\nThe lambda to minimize RMSE is %i\n The optimal RMSE is %f.",  
 optimal.fraction,  
 optimal.lambda,  
 optimal.rmse))

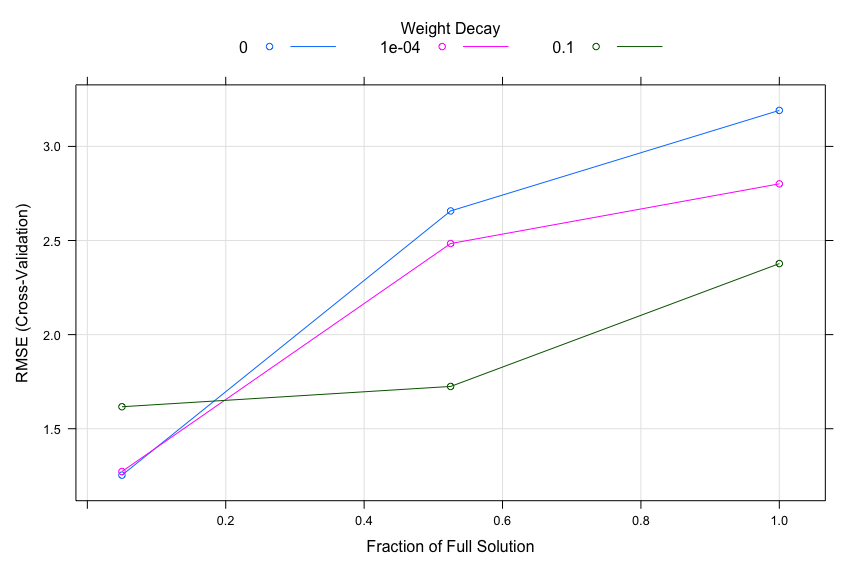
## The optimal fraction to minimize the RMSE is 0.050000  
## The lambda to minimize RMSE is 0  
## The optimal RMSE is 1.253215.

The pls net had the lower RMSE for the training data.

plot(plsTune)



plot(enetTune)



### (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?**

data.test.x <- dftest |> dplyr::select(-Yield)  
data.test.y <- dftest |> dplyr::select(Yield)  
  
preds <- predict(plsTune, data.test.x)  
postResample(preds, data.test.y$Yield)

## RMSE Rsquared MAE   
## 1.1427936 0.6744083 0.8887247

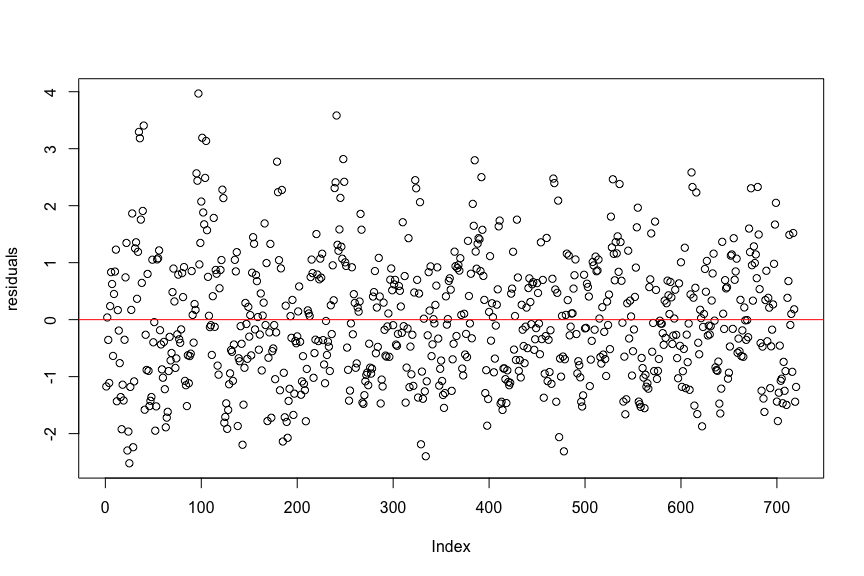
preds <- predict(enetTune, data.test.x)  
postResample(preds, data.test.y$Yield)

## RMSE Rsquared MAE   
## 1.1966731 0.7248341 0.9850749

The RMSE value approximately the same in the test set as on the training set for both, but the RMSE was lower for the PLS than the elastic net on the test data.

We can check out the residuals for the pls.

plot(residuals(plsTune), ylab='residuals')  
abline(h = 0, col = 'red')



The residuals appear to be random and centered around mean zero. The model appears to fit well.

### (e)

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

To understand what predictors are most important to the model the “variable importance in the projection” designed by Wold et. Al can be used (pg. 118). This metric was created because without it, it can be difficult to understand the importance of any one predictor because they are linear combinations of predictors. Using the function caret::varImp we can interpret the results with a scaled list of the importance of each variables for the pls tuned model, where 100 is the most important and 0 is the least. Of the top factors, about half are biological and half are manufacturing, while overall, manufacturing processes dominate.

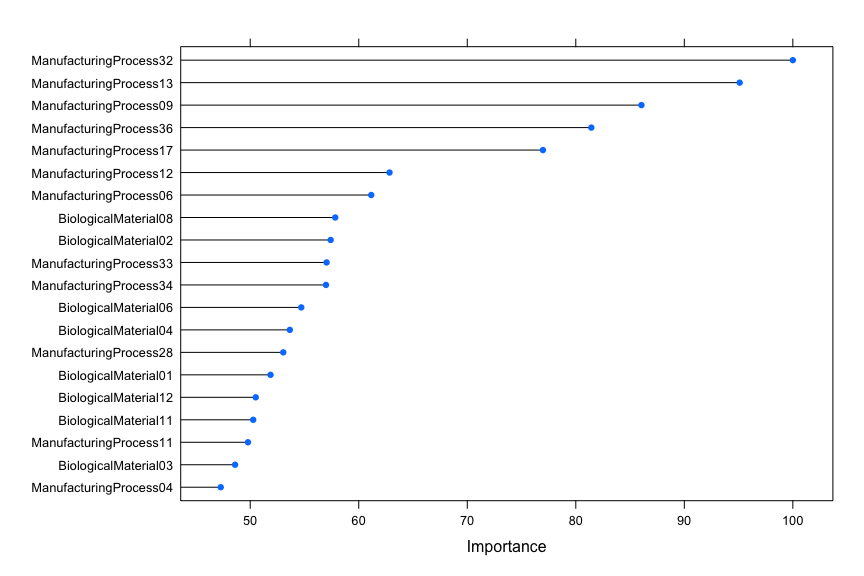
var.imp <- varImp(plsTune)

##   
## Attaching package: 'pls'

## The following object is masked from 'package:caret':  
##   
## R2

## The following object is masked from 'package:stats':  
##   
## loadings

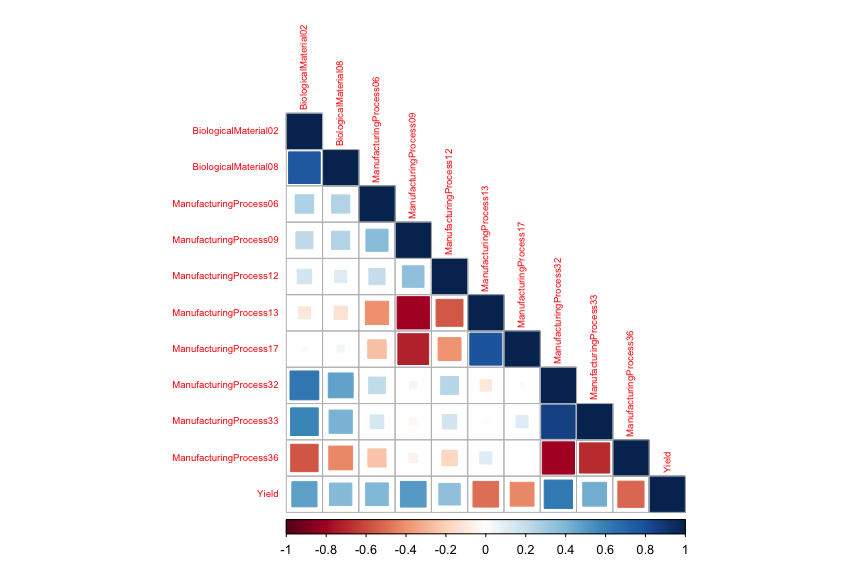
plot(var.imp, top = 20)



### (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?**

# store 10 most important variable names  
m.imp <- var.imp$importance |>  
 data.frame() |>  
 arrange(desc(Overall)) |>  
 head(10) |>  
 row.names()  
  
# correlation matrix of 10 most important variables and response variable  
dfimp |>  
 dplyr::select(all\_of(c("Yield",m.imp))) |>  
 cor() |>  
 round(2) |>  
 corrplot::corrplot(method = "square",  
 order = "alphabet",  
 tl.cex = 0.6,  
 type = "lower")



## Week 5

This correlation matrix helps make clear processes that can be increased or decreased to improve yield. We can observe that we might reduce the processes with strong negative correlation with the yield - Manufacturing Processes 13, 17, and 36 - as reducing those processes will increase the yield. Because PLS was the best-performing model, it isn’t possible to quantify the increase in yield that could be realized if certain processes were increased or decreased. This is due to the fact that PLS mathematically transforms the actual features into new, latent predictors to remove collinearity among variables while selecting the features that have the greatest contribution to variance. While it isn’t possible to directly quantify cost savings, we can say that the greatest savings will be realized by adjusting those with the most relative importance (i.e. manufacturing process #32, 13, 09, 36, etc.).

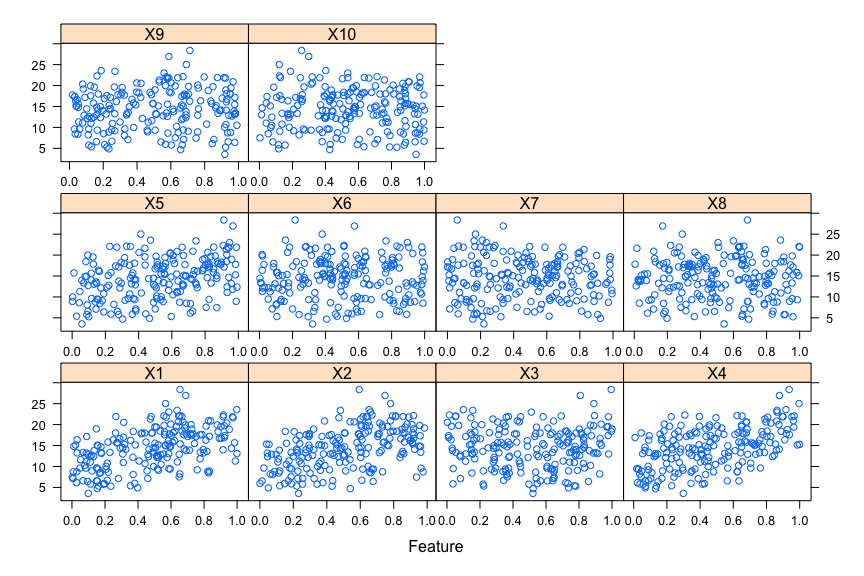
### 7.2

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(πx1x2) + 20(x3 − 0.5)2 + 10x4 + 5x5 + N(0, σ2)

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). The package mlbench contains a function called mlbench.friedman1 that simulates these data:

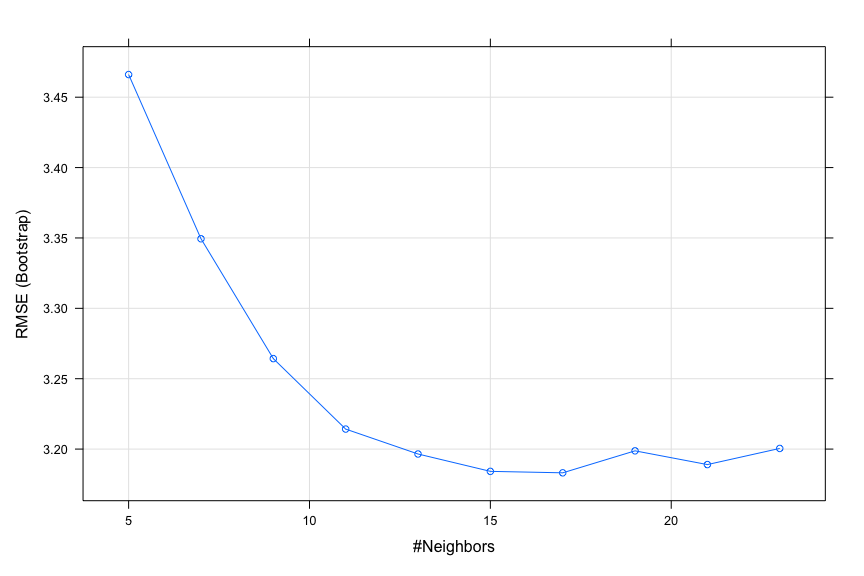
library(mlbench)  
set.seed(200)  
trainingData <- mlbench.friedman1(200, sd = 1)  
## We convert the 'x' data from a matrix to a data frame  
## One reason is that this will give the columns names.  
trainingData$x <- data.frame(trainingData$x)  
## Look at the data using  
featurePlot(trainingData$x,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



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

Tune several models on these data. For example:

# Results data frame  
df\_results <- data.frame(matrix(nrow=4, ncol=3))  
colnames(df\_results) <- c('Model', 'Tuning.Parameters', 'RMSE')  
  
library(caret)  
knnModel <- train(x = trainingData$x,   
 y = trainingData$y,   
 method = "knn",  
 preProc = c("center", "scale"),  
 tuneLength = 10)  
# knnModel  
  
knnPred <- predict(knnModel, newdata = testData$x)  
## The function 'postResample' can be used to get the test set  
## perforamnce values  
knn.results <- postResample(pred = knnPred, obs = testData$y)  
  
df\_results[1,] = data.frame(  
 Model='knnModel',   
 Tuning.Parameters=paste0('k=', knnModel$bestTune[['k']]),   
 RMSE=min(knn.results[['RMSE']])  
)  
  
plot(knnModel)



### NEURAL NETWORK MODEL

First we remove predcictors to ensure that the maximum absolute pairwise correlation between the predictors is less than 0.75,

findCorrelation(cor(trainingData$x), cutoff = .75)

## integer(0)

No predictors exceed a pairwise correlation of .75 or greater. Next, we create a specific candidate set of models to evaluate:

library(doParallel)  
# Find out how many cores are available:   
detectCores()

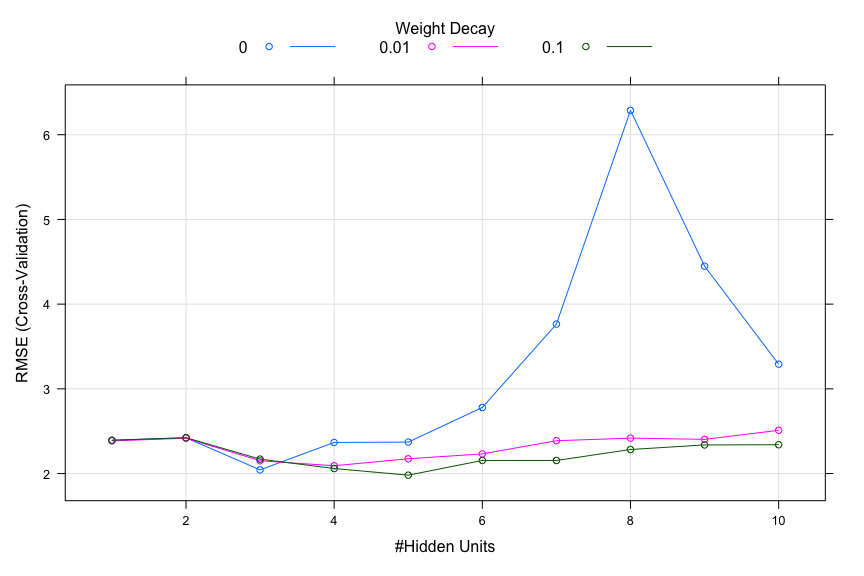
## [1] 4

## [1] 16  
# Create cluster with desired number of cores:   
cl <- makeCluster(16)  
# Register cluster:   
registerDoParallel(cl)  
# Find out how many cores are being used  
getDoParWorkers()

## [1] 16

# specify and store the resampling method  
ctrl <- trainControl(method = "cv",   
 allowParallel = TRUE,  
 number = 10)  
  
nnetGrid <- expand.grid(.decay = c(0, 0.01, .1),  
 .size = c(1:10),  
 .bag = FALSE)  
  
set.seed(100)   
nnetModel <- train(trainingData$x, trainingData$y,  
 method = "avNNet",  
 tuneGrid = nnetGrid,  
 trControl = ctrl,  
 preProc = c("center","scale"),  
 linout = TRUE,  
 trace = FALSE,  
 MaxNwts = 10 \* (ncol(trainingData$x) + 1) + 10 + 1,  
 maxit = 500,  
 allowParallel = TRUE)  
  
  
stopCluster(cl)  
registerDoSEQ()  
  
# nnetModel

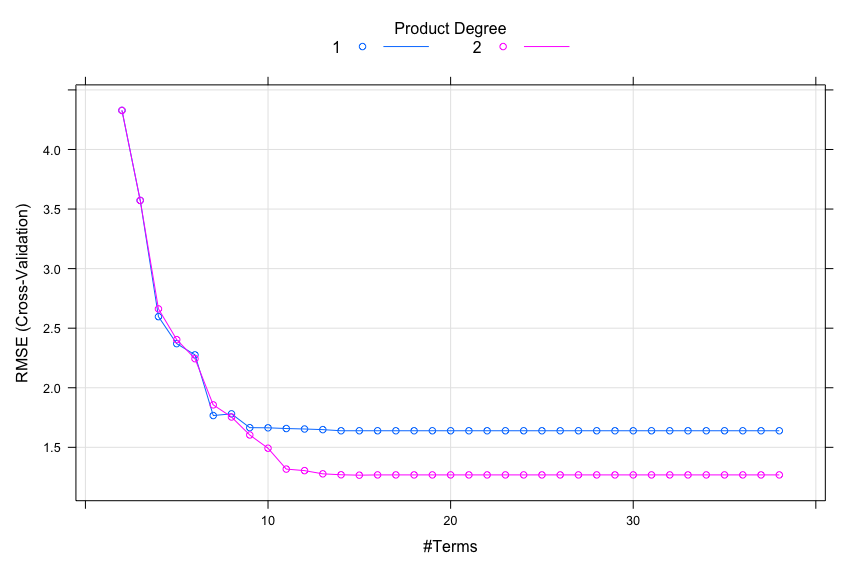
nnetPred <- predict(nnetModel, newdata = testData$x)  
nnet.results <- postResample(pred = nnetPred, obs = testData$y)  
  
df\_results[2,] = data.frame(  
 Model='Neural Net',   
 Tuning.Parameters=paste0('decay=', nnetModel$bestTune[['decay']], ', size=', nnetModel$bestTune[['size']], ', bag=False'),   
 RMSE=min(nnet.results[['RMSE']])  
)  
plot(nnetModel)



### MARS MODEL (Multivariate Adaptive Regression Splines)

marsGrid <- expand.grid(.degree = 1:2, .nprune = 2:38)  
  
set.seed(100)  
  
marsModel <- train(trainingData$x, trainingData$y,  
 method = "earth",  
 # Explicitly declare the candidate models to test,  
 tuneGrid = marsGrid,  
 trControl = ctrl)  
  
# marsModel

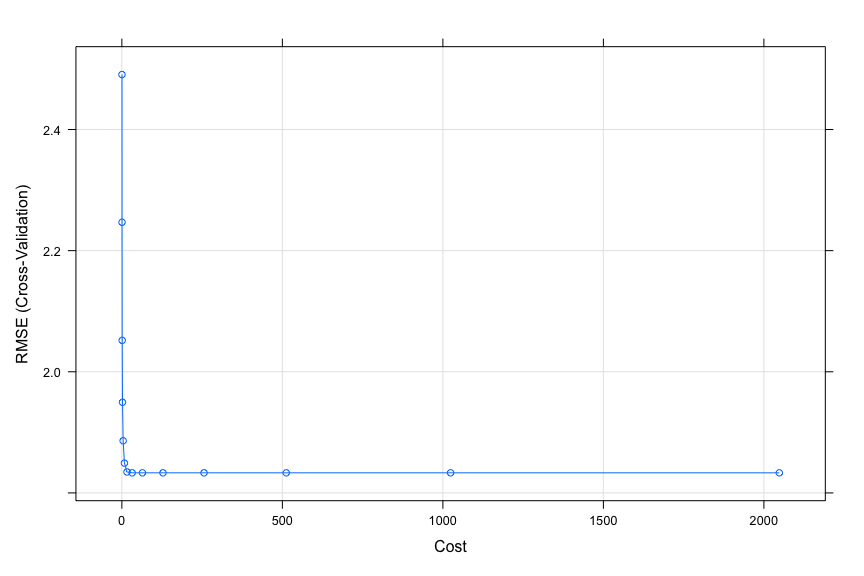
marsPred <- predict(marsModel, newdata = testData$x)  
mars.results <- postResample(pred = marsPred, obs = testData$y)  
  
df\_results[3,] = data.frame(  
 Model='MARS',   
 Tuning.Parameters=paste0('nprune=', marsModel$bestTune[['nprune']], ', degree=', marsModel$bestTune[['degree']]),  
 RMSE=min(mars.results[['RMSE']])  
)  
plot(marsModel)



### SVM (Support Vector Machines)

svmModel <- train(trainingData$x, trainingData$y,  
 method = "svmRadial",  
 preProcess = c("center","scale"),  
 tuneLength = 14,  
 trControl = ctrl)  
  
# svmModel

svmPred <- predict(svmModel, newdata = testData$x)  
svm.results <- postResample(pred = svmPred, obs = testData$y)  
  
df\_results[4,] = data.frame(  
 Model='SVM',   
 Tuning.Parameters=paste0('sigma=', svmModel$bestTune[['sigma']], ', C=', svmModel$bestTune[['C']]),  
 RMSE=min(svm.results[['RMSE']])  
)  
plot(svmModel)



Now we can compare the final results,

df\_results |> flextable() |>  
 width(width = 2) |>  
 fontsize(size = 10) |>  
 line\_spacing(space = 1) |>  
 hline(part = "all")

| Model | Tuning.Parameters | RMSE |
| --- | --- | --- |
| knnModel | k=17 | 3.204059 |
| Neural Net | decay=0.1, size=5, bag=False | 2.132825 |
| MARS | nprune=15, degree=2 | 1.158995 |
| SVM | sigma=0.0631548298520486, C=32 | 2.074163 |

The MARS model performs best according to the RMSE. The RMSE was significantly less than that of its closest competitor, SVM. The model-averaged neural network model performed similarly to SVM. The K-nearest neighbor model performed poorly, with an RMSE well above the others. This is not surprising, as KNN models typically perform poorly when there are predictors that don’t contribute significantly to the response; as stated in the problem description, in this data set there are five such predictors. The other models account for “noisy” predictors by pruning or applying a weight or decay factor.

vimp <- varImp(marsModel)   
df\_vimp <- data.frame(Variable = rownames(vimp$importance), Importance = vimp$importance)  
df\_vimp |> flextable() |>  
 width(width = 2) |>  
 fontsize(size = 10) |>  
 line\_spacing(space = 0.5) |>  
 hline(part = "all")

| Variable | Overall |
| --- | --- |
| X1 | 100.00000 |
| X4 | 75.23592 |
| X2 | 48.72974 |
| X5 | 15.51884 |
| X3 | 0.00000 |

The MARS model does indeed select X1-X5 as the most informative variables! It does set X3 to 0, interestingly.

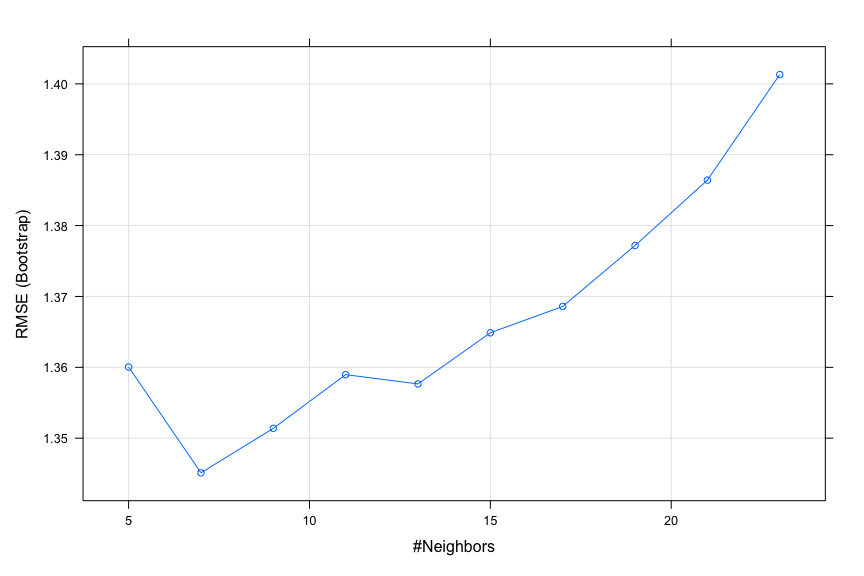
### 7.5

Exercise 6.3 describes data for a chemical manufacturing process. Use the same data imputation, data splitting, and pre-processing steps as before and train several nonlinear regression models.

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

set.seed("09041996")  
data("ChemicalManufacturingProcess")  
cmp\_i <- knnImputation(ChemicalManufacturingProcess,k=3)  
  
cmp\_i2 <- cmp\_i |> select(-Yield)  
  
trans <- preProcess(cmp\_i2, method = c("center", "scale", "YeoJohnson", "nzv"))  
cmp\_i2 <- predict(trans, cmp\_i2)  
  
cmp\_i2$Yield <- cmp\_i$Yield  
  
cmp\_i <- cmp\_i2  
  
i <- createDataPartition(cmp\_i$Yield,  
 p = 0.8,  
 list = FALSE,  
 times = 1)  
  
x <- cmp\_i[,colnames(cmp\_i)[colnames(cmp\_i) != 'Yield']]  
y <- cmp\_i[,colnames(cmp\_i)[colnames(cmp\_i) == 'Yield']]  
  
data.train.x <- x[i,]  
data.train.y <- y[i]  
data.test.x <- x[-i,]  
data.test.y <- y[-i]

library(caret)  
  
# Results data frame  
df\_results <- data.frame(matrix(nrow=4, ncol=3))  
colnames(df\_results) <- c('Model', 'Tuning.Parameters', 'RMSE')  
  
knnModel <- train(x = data.train.x,   
 y = data.train.y,   
 method = "knn",  
 tuneLength = 10)  
# knnModel  
  
knnPred <- predict(knnModel, newdata = data.test.x)  
## The function 'postResample' can be used to get the test set  
## perforamnce values  
knn.results <- postResample(pred = knnPred, obs = data.test.y)  
  
df\_results[1,] = data.frame(  
 Model='knnModel',   
 Tuning.Parameters=paste0('k=', knnModel$bestTune[['k']]),   
 RMSE=min(knn.results[['RMSE']])  
)  
  
plot(knnModel)



Next try neural network.

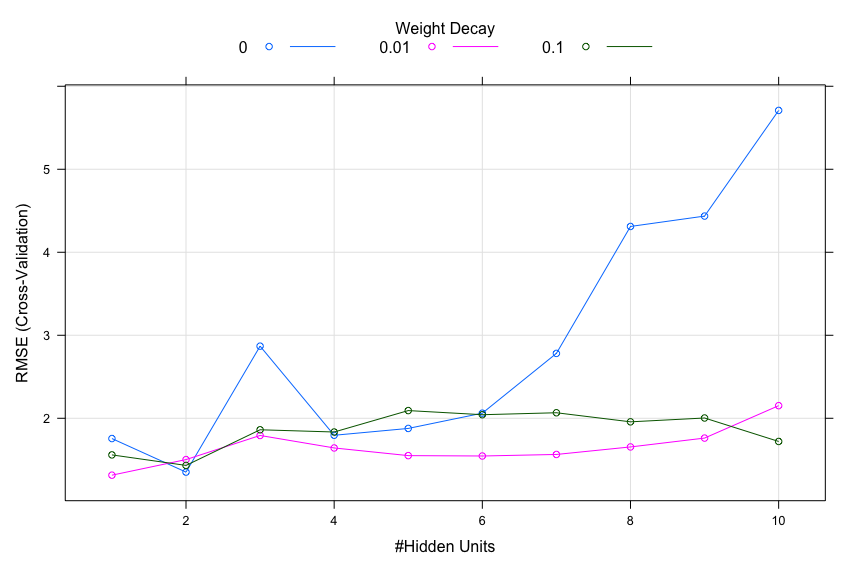
# remove predictors so absolute pairwise correlation is less than 0.75  
  
tooHigh <- findCorrelation(cor(data.train.x), cutoff = .75)  
  
  
trainXnnet <- data.train.x[, -tooHigh]  
testXnnet <- data.test.x[, -tooHigh]  
  
# Find out how many cores are available:   
detectCores()

## [1] 4

## [1] 16  
# Create cluster with desired number of cores:   
cl <- makeCluster(16)  
# Register cluster:   
registerDoParallel(cl)  
# Find out how many cores are being used  
getDoParWorkers()

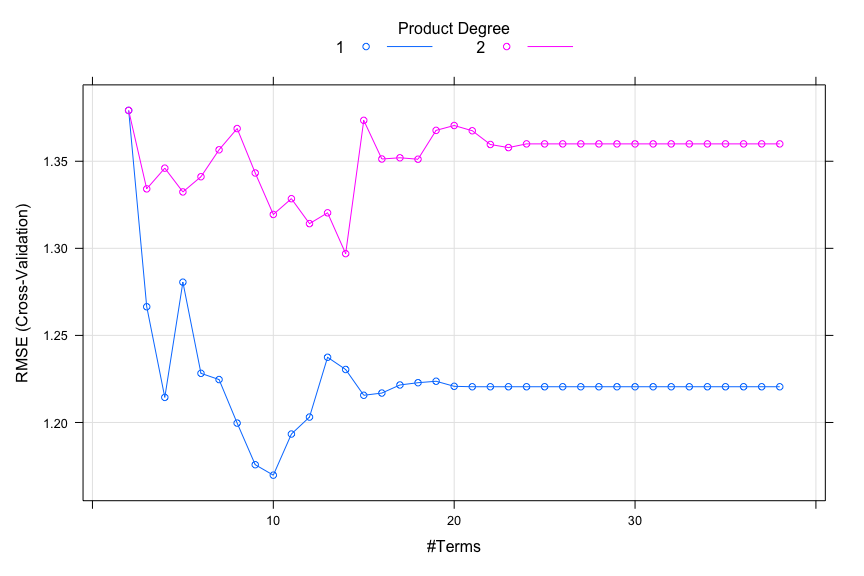
## [1] 16

nnetGrid <- expand.grid(.decay = c(0, 0.01, .1), .size = c(1:10), .bag = FALSE)  
  
nnetModel <- train(x = trainXnnet,   
 y = data.train.y,  
 method = "avNNet",  
 tuneGrid = nnetGrid,  
 trControl = trainControl(method = "cv"),  
 preProc = c("center", "scale"),  
 linout = TRUE,  
 trace = FALSE,  
 MaxNWts = 10 \* (ncol(trainXnnet) + 1) + 10 + 1,  
 maxit = 500,  
 allowParallel = TRUE)  
  
stopCluster(cl)  
registerDoSEQ()  
  
# nnetModel  
  
nnetPredict <- predict(nnetModel, newdata = testXnnet)  
## The function 'postResample' can be used to get the test set  
## perforamnce values  
nnet.results <- postResample(pred = nnetPredict, obs = data.test.y)  
  
df\_results[2,] = data.frame(  
 Model='Neural Net',   
 Tuning.Parameters=paste0('decay=', nnetModel$bestTune[['decay']], ', size=', nnetModel$bestTune[['size']], ', bag=False'),   
 RMSE=min(nnet.results[['RMSE']])  
)  
plot(nnetModel)



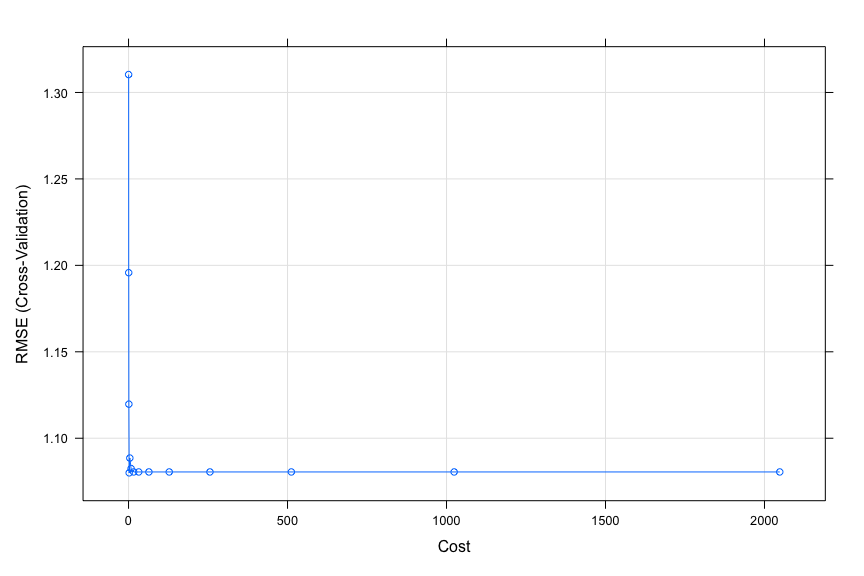
Next try MARS.

marsGrid <- expand.grid(.degree = 1:2, .nprune = 2:38)  
  
  
MARSModel <- train(x = data.train.x,   
 y = data.train.y,   
 method = "earth",  
 tuneGrid = marsGrid,  
 trControl = trainControl(method = "cv"))  
  
  
# MARSModel  
  
MARSPred <- predict(MARSModel, newdata = data.test.x)  
## The function 'postResample' can be used to get the test set  
## perforamnce values  
mars.results <- postResample(pred = MARSPred, obs = data.test.y)  
  
df\_results[3,] = data.frame(  
 Model='MARS',   
 Tuning.Parameters=paste0('nprune=', MARSModel$bestTune[['nprune']], ', degree=', MARSModel$bestTune[['degree']]),  
 RMSE=min(mars.results[['RMSE']])  
)  
plot(MARSModel)



Next try SVM.

svmModel <- train(x = data.train.x,   
 y = data.train.y,   
 method = "svmRadial",  
 tuneLength = 14,  
 trControl = trainControl(method = "cv"))  
# svmModel  
  
svmPred <- predict(svmModel, newdata = data.test.x)  
## The function 'postResample' can be used to get the test set  
## perforamnce values  
svm.results <- postResample(pred = svmPred, obs = data.test.y)  
  
df\_results[4,] = data.frame(  
 Model='SVM',   
 Tuning.Parameters=paste0('sigma=', svmModel$bestTune[['sigma']], ', C=', svmModel$bestTune[['C']]),  
 RMSE=min(svm.results[['RMSE']])  
)  
plot(svmModel)



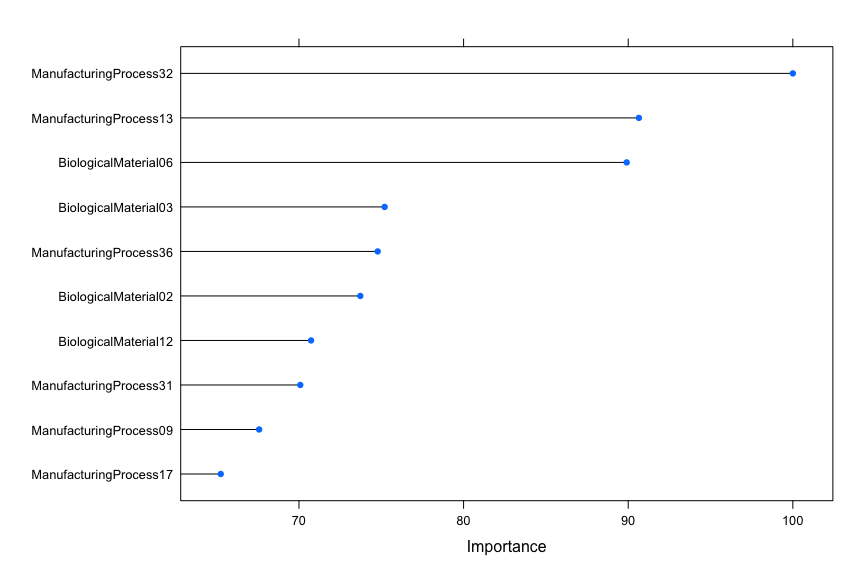
df\_results |> flextable() |>  
 width(width = 2) |>  
 fontsize(size = 10) |>  
 line\_spacing(space = 1) |>  
 hline(part = "all")

| Model | Tuning.Parameters | RMSE |
| --- | --- | --- |
| knnModel | k=7 | 1.643013 |
| Neural Net | decay=0.01, size=1, bag=False | 1.904619 |
| MARS | nprune=10, degree=1 | 1.598742 |
| SVM | sigma=0.0132601357266887, C=2 | 1.288996 |

The optimal resampling and test performance was from the SVM model.

1. 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?

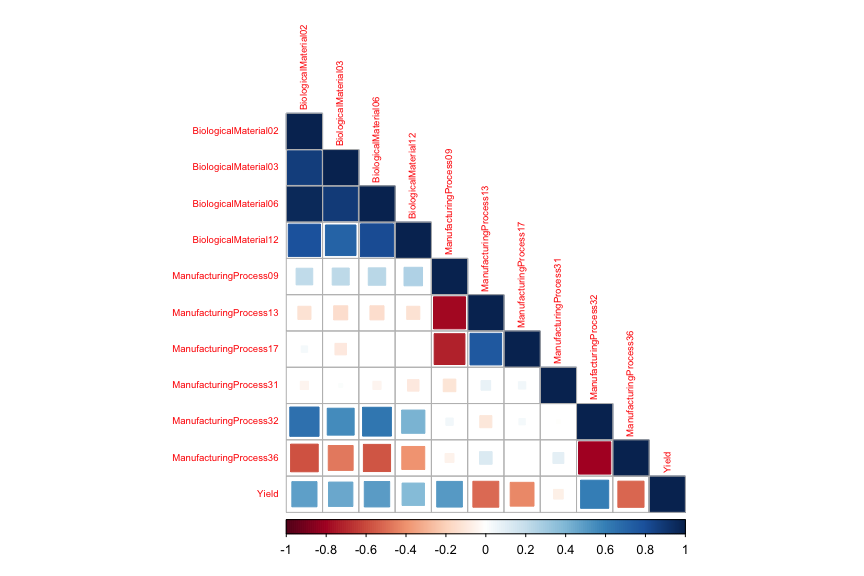
# variable importance  
var.imp <- varImp(svmModel)  
plot(var.imp, top = 10)



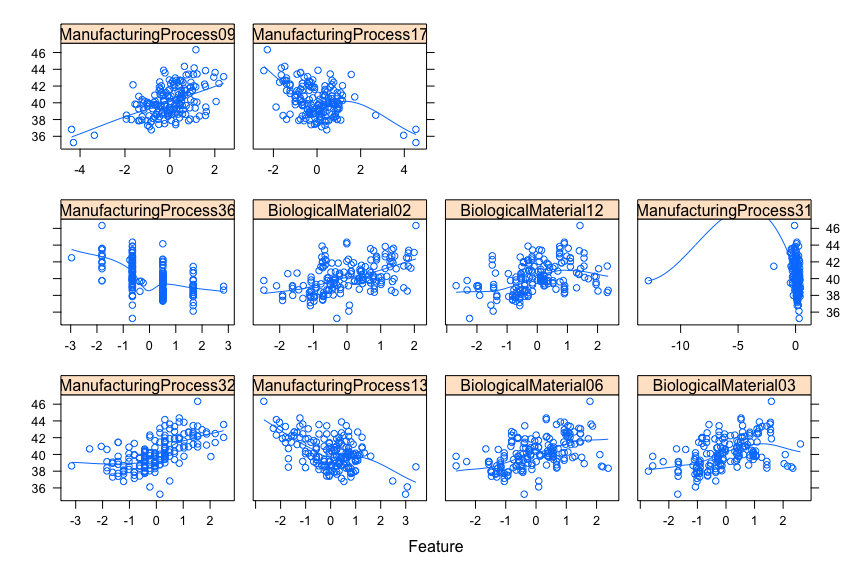
Among the ten most important variables, six are process and four are biological. For the linear model, eight were process and two were biological. The idea that process variables are more important to yield holds. This is convenient because process variables can be controlled.

1. 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?

# store 10 most important variable names  
m.imp <- var.imp$importance |>  
 data.frame() |>  
 arrange(desc(Overall)) |>  
 head(10) |>  
 row.names()  
  
# correlation matrix of 10 most important variables and response variable  
cmp\_i |>  
 select(all\_of(c("Yield",m.imp))) |>  
 cor() |>  
 round(2) |>  
 corrplot::corrplot(method = "square",  
 order = "alphabet",  
 tl.cex = 0.6,  
 type = "lower")



featurePlot(df[,m.imp], cmp\_i$Yield,  
 between = list(x = 1, y = 1),  
 type = c("p","smooth"))



These plots do reveal the intuition about the relationship with yield. We can see the positive and negative linear relationships between the variables and yield in the FeaturePlot, corresponding to the correlations in the matrix plot.

### 8.1

Recreate the simulated data from Exercise 7.2

library(mlbench)  
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"

1. Fit a random forest model to all of the predictors, then estimate the variable importance scores:

library(randomForest)

## randomForest 4.7-1.1

## Type rfNews() to see new features/changes/bug fixes.

##   
## Attaching package: 'randomForest'

## The following object is masked from 'package:gridExtra':  
##   
## combine

## The following object is masked from 'package:dplyr':  
##   
## combine

## The following object is masked from 'package:ggplot2':  
##   
## margin

library(caret)  
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)?

# Display importance values  
  
df\_rfImp1 <- data.frame(Variable = rownames(rfImp1), Importance = rfImp1$Overall)  
  
df\_rfImp1 %>%  
 arrange(desc(Importance)) %>%  
 flextable() |>  
 width(width = 2) |>  
 fontsize(size = 10) |>  
 line\_spacing(space = 0.5) |>  
 hline(part = "all")

| Variable | Importance |
| --- | --- |
| V1 | 8.83890885 |
| V4 | 7.58822553 |
| V2 | 6.49023056 |
| V5 | 2.27426009 |
| V3 | 0.67583163 |
| V6 | 0.17436781 |
| V7 | 0.15136583 |
| V9 | -0.02989832 |
| V8 | -0.03078937 |
| V10 | -0.08529218 |

As shown, the uninformative predictors (V6-V10) have a significantly lower relative importance in the model, which is consistent with our earlier findings.

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

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

## [1] 0.9396216

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)  
  
  
df\_rfImp2 <- data.frame(Variable = rownames(rfImp2), Importance = rfImp2$Overall)  
  
df\_rfImp2 %>%  
 arrange(desc(Importance)) %>%  
 flextable() |>  
 width(width = 2) |>  
 fontsize(size = 10) |>  
 line\_spacing(space = 0.5) |>  
 hline(part = "all")

| Variable | Importance |
| --- | --- |
| V4 | 6.93924427 |
| V1 | 6.29780744 |
| V2 | 6.08038134 |
| duplicate1 | 3.56411581 |
| V5 | 2.03104094 |
| V3 | 0.58410718 |
| V6 | 0.07947642 |
| V10 | -0.00715093 |
| V7 | -0.02566414 |
| V9 | -0.08839463 |
| V8 | -0.11007435 |

The variable importance of V1 went down, and the importance of the duplicate1 variable was approximately the same - so effectively the variable importance is split between the two correlated variables.

What happens when you add another predictor that is also highly correlated with V1?

simulated$duplicate2 <- simulated$V1 + rnorm(200) \* .2  
cor(simulated$duplicate2, simulated$V1)

## [1] 0.7732382

model3 <- randomForest(y ~ ., data = simulated,  
 importance = TRUE,  
 ntree = 1000)  
rfImp3 <- varImp(model3, scale = FALSE) |> arrange(desc(Overall))  
  
df\_rfImp3 <- data.frame(Variable = rownames(rfImp3), Importance = rfImp3$Overall)  
  
df\_rfImp3 %>%  
 arrange(desc(Importance)) %>%  
 flextable() |>  
 width(width = 2) |>  
 fontsize(size = 10) |>  
 line\_spacing(space = 0.5) |>  
 hline(part = "all")

| Variable | Importance |
| --- | --- |
| V4 | 7.51171970 |
| V2 | 6.48305559 |
| V1 | 6.19983255 |
| duplicate1 | 3.24628944 |
| V5 | 2.27902033 |
| duplicate2 | 1.09060225 |
| V3 | 0.42757857 |
| V6 | 0.12585434 |
| V7 | 0.05716457 |
| V10 | -0.01113780 |
| V9 | -0.05339077 |
| V8 | -0.08621813 |

As shown in the table, the relative importances of V1 and duplicate1 decreased with the introduction of variable duplicate2. The sum of these three importances are roughly on par with the importance of V1 in the first model and the sum of v1 + duplicate1 in the second model. This is consistent with the way regression trees are sensitive to highly correlated variables.

1. 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?

library(party)

## Loading required package: grid

## Loading required package: mvtnorm

## Loading required package: modeltools

## Loading required package: stats4

## Loading required package: strucchange

## Loading required package: zoo

##   
## Attaching package: 'zoo'

## The following objects are masked from 'package:base':  
##   
## as.Date, as.Date.numeric

## Loading required package: sandwich

##   
## Attaching package: 'strucchange'

## The following object is masked from 'package:stringr':  
##   
## boundary

##   
## Attaching package: 'party'

## The following object is masked from 'package:dplyr':  
##   
## where

model\_forest <- cforest(y ~ ., data = simulated)  
  
m\_t <- varimp(model\_forest, conditional = TRUE)   
m\_f <- varimp(model\_forest, conditional = FALSE)   
  
l\_t <- cbind(m\_t, m\_f)  
  
df\_t <- data.frame(Variable = rownames(l\_t),   
 Conditional\_True = l\_t[,1],  
 Conditional\_False = l\_t[,2])  
  
df\_t |> flextable() |>  
 width(width = 2) |>  
 fontsize(size = 10) |>  
 line\_spacing(space = 0.5) |>  
 hline(part = "all")

| Variable | Conditional\_True | Conditional\_False |
| --- | --- | --- |
| V1 | 2.487590376 | 6.174055354 |
| V2 | 4.580552877 | 5.966632261 |
| V3 | 0.027953446 | 0.019674248 |
| V4 | 5.615551510 | 7.332605965 |
| V5 | 1.057098660 | 1.708606673 |
| V6 | 0.015134623 | 0.030054426 |
| V7 | 0.005205066 | 0.003222082 |
| V8 | 0.005728994 | -0.008127304 |
| V9 | -0.008507938 | -0.004419864 |
| V10 | -0.010683929 | -0.014711047 |
| duplicate1 | 0.690821538 | 2.743152766 |
| duplicate2 | 0.067750307 | 0.519475346 |

Using conditional inference, with the toggle being True, has the effect of lowering the importance of the two variables that are highly correlated with V1; however V1 is also reduced. Variable V1 itself remained third in the order of relative importance among other variables in both cases. It didn’t increase in importance and, in fact, actually had a lower magnitude, but the overall effect was that the importances of the duplicate variables were reduced for the toggle being True. This can be attributed to the fact that statistical inference tests are exhaustively conducted against all possible predictors and across all possible split points, thereby penalizing models with higher numbers of splits.

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

Boosted tree.

# Init results table  
dfr <- data.frame(matrix(nrow=0, ncol=3))  
colnames(dfr) <- c('Model', 'Variable', 'Rank')  
  
# Boosted tree  
library(gbm)

## Loaded gbm 2.1.8.1

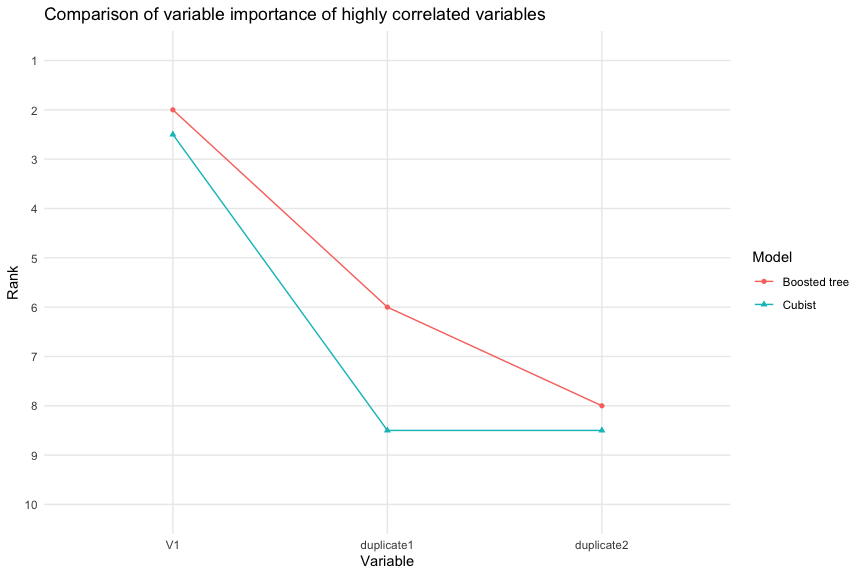
set.seed(77)  
fitgbm1 <- gbm(y ~ ., data=simulated, distribution='gaussian', n.trees=1000)  
tmpdf <- varImp(fitgbm1, numTrees=1000)  
  
# Results table  
dfr <- rbind(dfr, data.frame(  
 Model='Boosted tree',   
 Variable=rownames(tmpdf),  
 Rank=rank(-tmpdf$Overall)))

Cubist.

# Cubist  
library(Cubist)  
set.seed(77)  
fitcub1 <- cubist(x=simulated %>% dplyr::select(-y), y=simulated$y)  
tmpdf <- varImp(fitcub1, numTrees=1000)  
  
# Results table  
dfr <- rbind(dfr, data.frame(  
 Model='Cubist',   
 Variable=rownames(tmpdf),  
 Rank=rank(-tmpdf$Overall)))

Compare variable importance of the models.

# Plot variable importance  
dfr %>%  
 filter(Variable %in% c('V1', 'duplicate1', 'duplicate2')) %>%  
 ggplot(aes(x=factor(Variable, level=c('V1', 'duplicate1', 'duplicate2')), y=11-Rank, group=Model, color=Model, shape=Model)) +  
 geom\_point() +  
 geom\_line() +  
 scale\_y\_discrete(name='Rank', limits=rev(factor(seq(1, 10)))) +  
 xlab('Variable') +  
 ggtitle('Comparison of variable importance of highly correlated variables')



As shown in the graph, the trees tend to split the variable importance among the three correlated predictors, although most downplayed the importance of the first duplicated variable.

### 8.2

Use a simulation to show tree bias with different granularities.

library(rpart)  
# Set seed  
set.seed(77)  
  
# Generate three discrete variables  
s1 <- sample(x=seq(0, 9), size=1000, replace=T)  
s2 <- sample(x=seq(0, 99), size=1000, replace=T)  
s3 <- sample(x=seq(0, 999), size=1000, replace=T)  
  
# Generate three continuous variables  
s4 <- runif(n=1000, min=0, max=9)  
s5 <- runif(n=1000, min=0, max=99)  
s6 <- runif(n=1000, min=0, max=999)  
  
# Generate outcome variable  
y <- s1 + s2 + s3 + s4 + s5 + s6 + runif(n=1000, min=0, max=1)  
  
# Create data frame  
df82 <- data.frame(outcome=y, discrete1=s1, discrete2=s2, discrete3=s3, continuous1=s4, continuous2=s5, continuous3=s6)  
  
# Simple CART  
fitcart <- rpart(outcome ~ ., data=df82)  
  
df\_fit <- varImp(fitcart)  
df\_fit <- data.frame(Variable = rownames(df\_fit), Importance = df\_fit$Overall)  
df\_fit %>%  
 arrange(desc(Importance)) %>%  
 flextable() |>  
 width(width = 2) |>  
 fontsize(size = 10) |>  
 line\_spacing(space = 0.5) |>  
 hline(part = "all")

| Variable | Importance |
| --- | --- |
| discrete3 | 4.9019351 |
| continuous3 | 2.7644374 |
| discrete2 | 0.3735736 |
| continuous2 | 0.3698630 |
| continuous1 | 0.2510469 |
| discrete1 | 0.0000000 |

As expected, the variables with many discrete values had variable importances that far outweighed the others.

### 8.3

In stochastic gradient boosting the bagging fraction and learning rate will govern the construction of the trees as they are guided by the gradient. Although the optimal values of these parameters should be obtained through the tuning process, it is helpful to understand how the magnitudes of these parameters affect magnitudes of variable importance. Figure 8.24 provides the variable importance plots for boosting using two extreme values for the bagging fraction (0.1 and 0.9) and the learning rate (0.1 and 0.9) for the solubility data. The left-hand plot has both parameters set to 0.1, and the right-hand plot has both set to 0.9:

1. 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?

*The bagging fraction represents the fraction of data selected to train each iteration of the trees. A bagging fraction of 0.1 means 10% of the data is being selected for each iteration; subsequently, each iteration may be built using very different data, so there will be many different important predictors, whereas when, say, 0.9 or 90% of the data is being selected for each iteration, each tree should have similar important predictors, as they are seeing more similar data.*

*Boosting can be susceptible to over-fitting, as boosting will select an optimal weak learner at each iteration. To offset this, regularization or shrinkage is applied as a learning rate which represents the fraction of a current prediction to be added to the previous iteration’s predictions. As the learning rate increases, a larger fraction of predictions is added to the model - but this is not a good thing. A learning rate of 1 indicates no shrinkage, high error, focusing on fewer variables, the model is fast and greedy. If the learning rate is set to a low value, there is less of a penalty against larger coefficients; therefore, less influential predictors will remain small relative to more influential predictors. On the other hand, if the learning rate is high, the penalty against larger coefficients will be greater, resulting in reduced influence for these predictors. The idea behind increasing the learning rate is to try reducing overfitting. This results in fewer variables with greater importance–i.e., a more parsimonious model. The model on the left, with a lower learning rate, could be overfit, as there are more variables with greater importance (KJ pp. 206-207).*

*With a smaller learning rate and a smaller bagging fraction, the variable importance may be more evenly distributed because there will be more trees created which gives more opportunity for a variable to be deemed “important”.*

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

*A lower learning rate indicates slower learning and fewer errors, while a lower bagging fraction indicates a smaller proportion of training data used for fitting each iteration, which limits the model’s exposure to the full range of data, possibly leading to over-fitting. However the lower learning rate may provide more accurate generalization over unseen samples, so we believe the model on the left would be more predictive.*

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

*According to the book, “When regression tree are used as the base learner, simple gradient boosting for regression has two tuning parameters: tree depth and number of iterations. Tree depth in this context is also known as interaction depth, since each subsequential split can be thought of as a higher-level interaction term with all of the other previous split predictors” (KJ p.205). Based on this, increasing the interaction depth means increasing the tree depth and doing so has the possibility of leading to overfitting the data. We would think that a smaller tree depth would lead to a variable importance that more rapidly decreases to 0, while a greater tree depth would have a more gradual decrease to 0.*

### 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:

We’ll load, preprocess, and split the data as before.

set.seed("09041996")  
data("ChemicalManufacturingProcess")  
cmp\_i <- knnImputation(ChemicalManufacturingProcess,k=3)  
  
cmp\_i2 <- cmp\_i |> select(-Yield)  
  
trans <- preProcess(cmp\_i2, method = c("center", "scale", "YeoJohnson", "nzv"))  
cmp\_i2 <- predict(trans, cmp\_i2)  
  
cmp\_i2$Yield <- cmp\_i$Yield  
  
cmp\_i <- cmp\_i2  
  
i <- createDataPartition(cmp\_i$Yield,  
 p = 0.8,  
 list = FALSE,  
 times = 1)  
  
x <- cmp\_i[,colnames(cmp\_i)[colnames(cmp\_i) != 'Yield']]  
y <- cmp\_i[,colnames(cmp\_i)[colnames(cmp\_i) == 'Yield']]  
  
data.train.x <- x[i,]  
data.train.y <- y[i]  
data.test.x <- x[-i,]  
data.test.y <- y[-i]  
  
# specify 10x cross-validation  
ctrl <- trainControl(method='cv', number=10)  
  
# Results data frame  
dfr <- data.frame(matrix(nrow=6, ncol=3))  
colnames(dfr) <- c('Model', 'Tuning.Parameters', 'Test.RMSE')

Basic CART model tuned using complexity parameter.

# Basic CART model - tuned using complexity parameter  
cartModel <- train(data.train.x, data.train.y, method='rpart', tuneLength=10, trControl=ctrl)

## Warning in nominalTrainWorkflow(x = x, y = y, wts = weights, info = trainInfo,  
## : There were missing values in resampled performance measures.

# cartModel  
  
cartPredict <- predict(cartModel, newdata = data.test.x)  
  
cart.results <- postResample(pred = cartPredict, obs = data.test.y)  
  
dfr[1,] = data.frame(  
 Model='Basic CART (tuned w/complexity parameter)',   
 Tuning.Parameters=paste0('cp=', cartModel$bestTune[['cp']]),   
 Test.RMSE=(cart.results[['RMSE']])  
)

Basic CART model tuned using node depth.

# Basic CART model - tuned using complexity parameter  
  
cart2Model <- train(data.train.x, data.train.y, method='rpart2', tuneLength=10, trControl=ctrl)

## note: only 8 possible values of the max tree depth from the initial fit.  
## Truncating the grid to 8 .

# cart2Model  
  
cart2Predict <- predict(cart2Model, newdata = data.test.x)  
  
cart2.results <- postResample(pred = cart2Predict, obs = data.test.y)  
  
dfr[2,] = data.frame(  
 Model='Basic CART (tuned w/node depth)',   
 Tuning.Parameters=paste0('maxdepth=', cart2Model$bestTune[['maxdepth']]),   
 Test.RMSE=(cart2.results[['RMSE']])  
)

Bagged CART.

# Bagged CART  
bagModel <- train(data.train.x, data.train.y, method='treebag', trControl=ctrl)  
# bagModel  
  
bagPredict <- predict(bagModel, newdata = data.test.x)  
  
bag.results <- postResample(pred = bagPredict, obs = data.test.y)  
  
dfr[3,] = data.frame(  
 Model='Bagged CART',   
 Tuning.Parameters='',   
 Test.RMSE=(bag.results[['RMSE']])  
)

Random forest.

# Random forest  
rfModel <- train(data.train.x, data.train.y, method='rf', tuneLength=10, trControl=ctrl)  
# rfModel  
  
rfPredict <- predict(rfModel, newdata = data.test.x)  
  
rf.results <- postResample(pred = rfPredict, obs = data.test.y)  
  
dfr[4,] = data.frame(  
 Model='Random forest',   
 Tuning.Parameter=paste0('mtry=', rfModel$bestTune[['mtry']]),   
 Test.RMSE=(rf.results[['RMSE']]))

Stochastic gradient boosting.

# Had to set message=FALSE to prevent thousands of trace messages  
  
# Stochastic gradient boosting  
gbmGrid <- expand.grid(.interaction.depth=seq(1, 7, by=2),  
 .n.trees=seq(100, 1000, by=50),  
 .shrinkage=c(0.01, 0.1),  
 .n.minobsinnode=10)  
gbmModel <- train(data.train.x, data.train.y, method='gbm', tuneGrid=gbmGrid, trControl=ctrl, verbose = FALSE)  
  
# gbmModel  
  
gbmPredict <- predict(gbmModel, newdata = data.test.x)  
  
gbm.results <- postResample(pred = gbmPredict, obs = data.test.y)  
  
dfr[5,] = data.frame(  
 Model='Stochastic gradient boosting',   
 Tuning.Parameters=paste0('interaction.depth=', gbmModel$bestTune[['interaction.depth']],   
 ', n.trees=', gbmModel$bestTune[['n.trees']],   
 ', shrinkage=', gbmModel$bestTune[['shrinkage']],  
 ', n.minobsinnode=10'),   
 Test.RMSE=(gbm.results[['RMSE']]))

Cubist.

# Cubist  
cubGrid <- expand.grid(.committees=c(seq(1, 10), seq(20, 100, by=10)), .neighbors=c(0, 1, 5, 9))  
  
cubModel <- train(data.train.x, data.train.y, method='cubist', tuneGrid=cubGrid, trControl=ctrl)  
# cubModel  
  
cubPredict <- predict(cubModel, newdata = data.test.x)  
  
cub.results <- postResample(pred = cubPredict, obs = data.test.y)  
  
dfr[6,] = data.frame(  
 Model='Cubist',   
 Tuning.Parameters=paste0('committees=', cubModel$bestTune[['committees']],   
 ', neighbors=', cubModel$bestTune[['neighbors']]),  
 Test.RMSE=(cub.results[['RMSE']]))

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

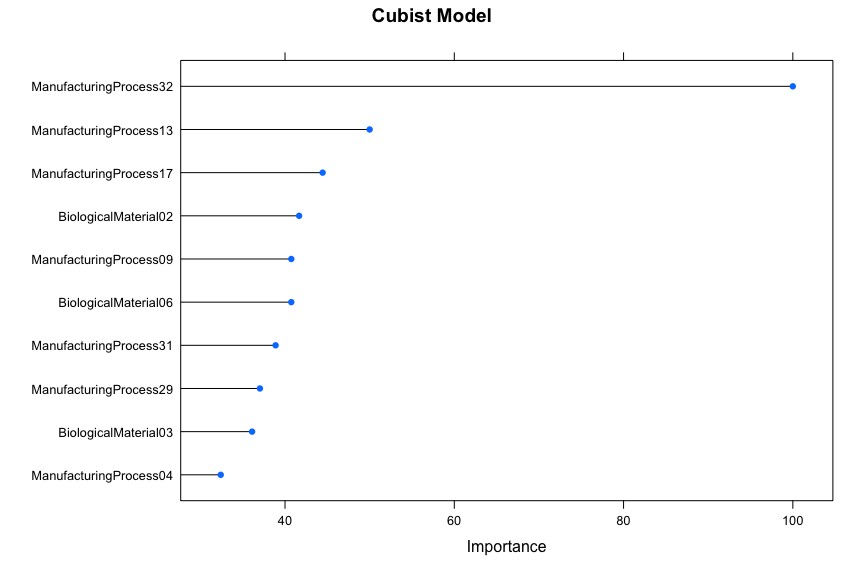
dfr |> flextable() |>  
 width(width = 2) |>  
 fontsize(size = 10) |>  
 line\_spacing(space = 1) |>  
 hline(part = "all")

| Model | Tuning.Parameters | Test.RMSE |
| --- | --- | --- |
| Basic CART (tuned w/complexity parameter) | cp=0.362468783025057 | 1.619029 |
| Basic CART (tuned w/node depth) | maxdepth=1 | 1.619029 |
| Bagged CART |  | 1.412404 |
| Random forest | mtry=20 | 1.354041 |
| Stochastic gradient boosting | interaction.depth=5, n.trees=1000, shrinkage=0.01, n.minobsinnode=10 | 1.203054 |
| Cubist | committees=100, neighbors=1 | 1.055449 |

The cubist model had the best performance.

1. 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(cubModel), top = 10, main = "Cubist Model")



In this model we see process variables comprise seven of the top ten most important variables, which is similar to what we observed in the linear and nonlinear models. Among the ten most important variables, six are process and four are biological for the nonlinear model. For the linear model, eight were process and two were biological. The idea that process variables are more important to yield holds. This is convenient because process variables can be controlled.

1. 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?

library(partykit)

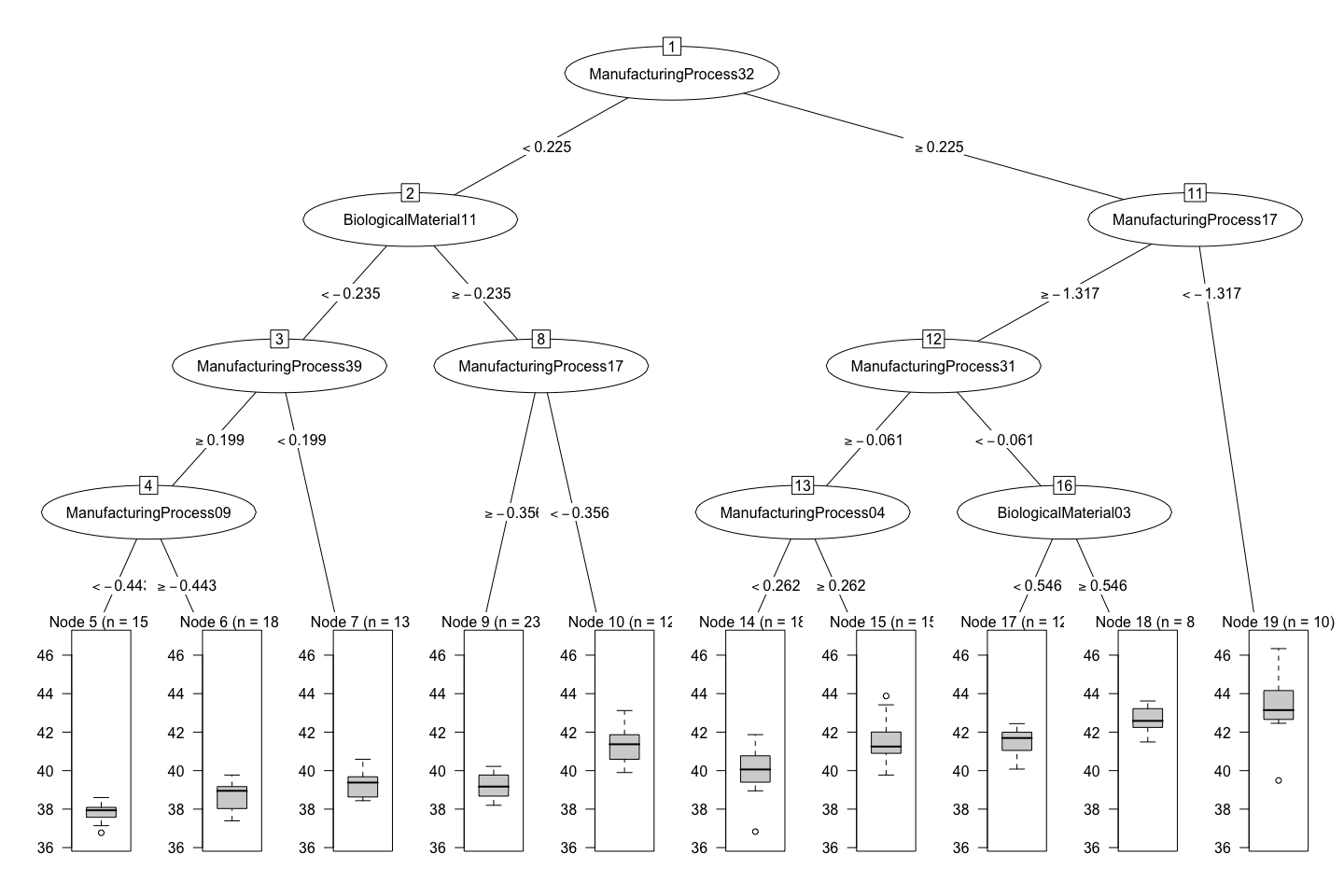
## Loading required package: libcoin

##   
## Attaching package: 'partykit'

## The following objects are masked from 'package:party':  
##   
## cforest, ctree, ctree\_control, edge\_simple, mob, mob\_control,  
## node\_barplot, node\_bivplot, node\_boxplot, node\_inner, node\_surv,  
## node\_terminal, varimp

## The following object is masked from 'package:flextable':  
##   
## width

library(rpart)  
data.train <- cmp\_i[i,]  
rpartTree <- rpart(Yield ~ ., data = data.train)  
rpartTree2 <- as.party(rpartTree)  
plot(rpartTree2)



We can observe the following:

* The root node for this tree is ManufacturingProcess32. We see the split occurs when the variable is < 0.225.
* Each child node represents a subset of the data at the denoted split criteria.
* The tree depth is the number of splits in the tree. A deeper tree with more splits indicates more complex decision rules and potentially more overfitting to the data. In this chart, we have three splits.
* The leaf nodes at the bottom represent the terminal nodes that contain the final predicted values for the response for its branch of the tree, we can see that the predicted yield is generally higher when the ManufacturingProcess32 is greater than 0.225, and we can observe where the exceptions to this occur.
* You can follow the rightmost tree branches down to find how to get the most yield. Similarly you can find weaknesses by going down the left branches to find what creates the lowest yield results.

### Recommender Problem

Imagine 1000 receipts sitting on your table. Each receipt represents a transaction with items that were purchased. The receipt is a representation of stuff that went into a customer’s basket – and therefore ‘Market Basket Analysis’.

That is exactly what the Groceries Data Set contains: a collection of receipts with each line representing 1 receipt and the items purchased. Each line is called a **transaction** and each column in a row represents an **item**.

Here is the dataset = [GroceryDataSet.csv]('https://github.com/klgriffen96/summer23_data624/blob/main/hw_2/GroceryDataSet.csv') (comma separated file).

Your assignment is to use R to mine the data for association rules. You should report support, confidence and lift and your top 10 rules by lift.

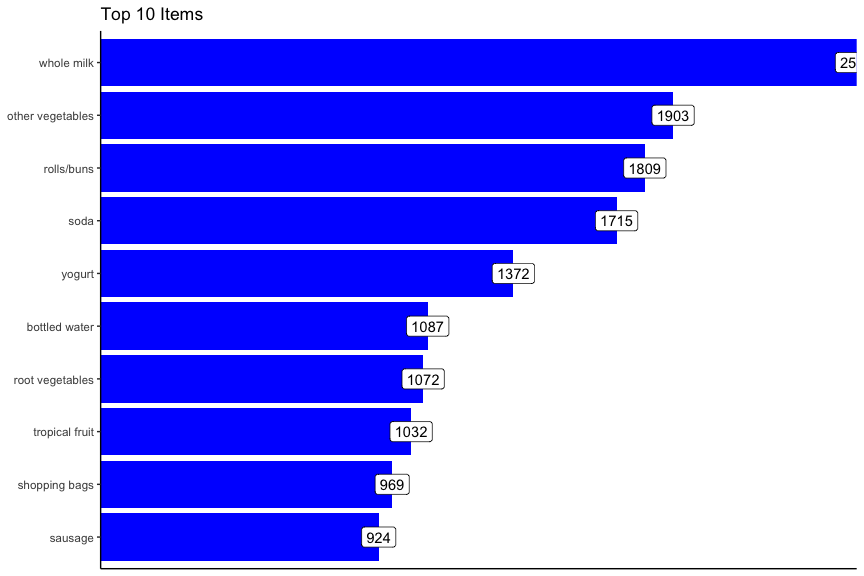
#### Exploratory Data Analysis

First, we will pull in the data (stored on github) into a dataframe. The data contains a row for every transaction. For our EDA we will want a row for every item, along with a transaction ID.

url <- "https://raw.githubusercontent.com/klgriffen96/summer23\_data624/main/hw\_2/GroceryDataSet.csv"  
data <- read.csv(url, header=FALSE)  
data[data == ""] <- NA # replace empty cells with NA  
data$tid <- as.integer(row.names(data)) # create transaction ID column using row indices  
  
# pivot data  
baskets <- data |>   
 pivot\_longer(cols = starts\_with("V"),   
 values\_to = "item",   
 values\_drop\_na = TRUE) |>  
 dplyr::select(tid, item)

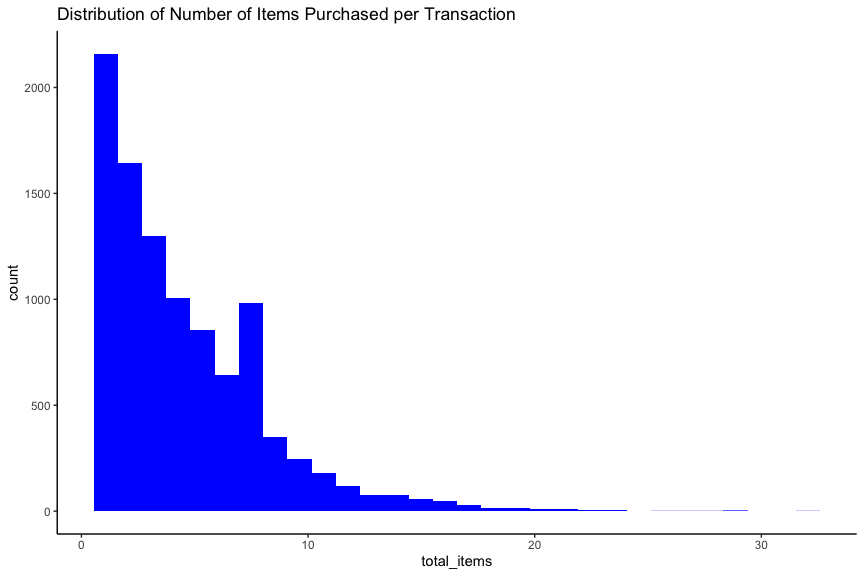
We can visualize the frequency of items in our baskets,

baskets |>  
 group\_by(item) |>  
 summarize(count = n()) |>  
 arrange(desc(count)) |>   
 head(10) |>  
 ggplot(aes(x=reorder(item,count), y=count)) +  
 geom\_col(fill="blue") +  
 geom\_label(aes(label=count)) +  
 labs(x=NULL, y=NULL) + ggtitle('Top 10 Items') +  
 scale\_y\_discrete(breaks=NULL) +  
 coord\_flip() +   
 theme\_classic()



Next we can look at the distribution of how many items were purchased in each transaction.

baskets |>  
 group\_by(tid) |>  
 summarize(total\_items = n()) |>  
 ggplot(aes(x=total\_items)) +  
 geom\_histogram(fill="blue", bins=30) +  
 theme\_classic() +  
 labs(title = "Distribution of Number of Items Purchased per Transaction")



#### Market Basket Analysis

Now we can shift to using the functionality of the arules package. We will read in the file to a transactions object. The data is in “basket” format, meaning every row is a set of items that represent one transaction. After loading we can take a look at the top ten items with the itemFrequencyPlot function.

library(arules)

## Loading required package: Matrix

##   
## Attaching package: 'Matrix'

## The following objects are masked from 'package:tidyr':  
##   
## expand, pack, unpack

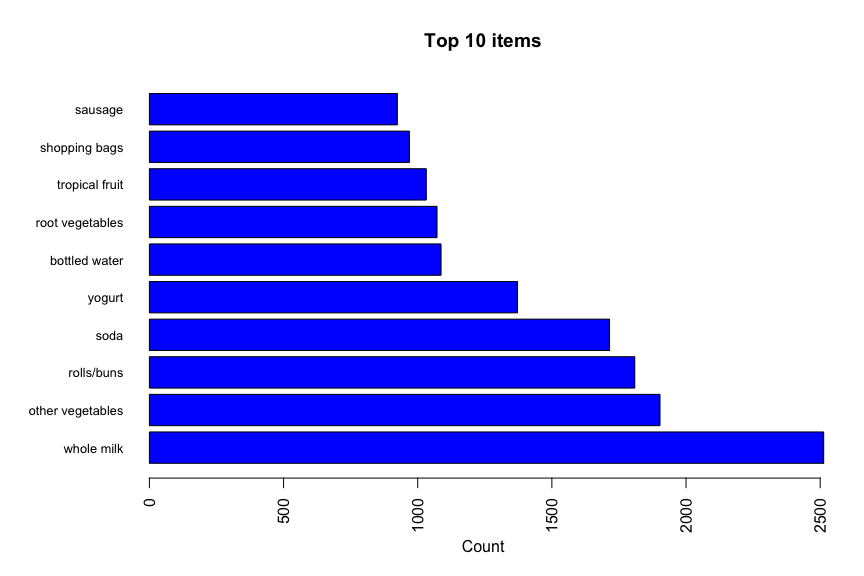
##   
## Attaching package: 'arules'

## The following object is masked from 'package:modeltools':  
##   
## info

## The following object is masked from 'package:dplyr':  
##   
## recode

## The following objects are masked from 'package:base':  
##   
## abbreviate, write

library(arulesViz)  
library(flextable)  
basket\_tr <- read.transactions(file = "https://raw.githubusercontent.com/klgriffen96/summer23\_data624/main/hw\_2/GroceryDataSet.csv",  
 sep = ",", format = "basket",   
 rm.duplicates = FALSE,  
 quote = "", skip = 0,  
 encoding = "unknown")  
  
itemFrequencyPlot(basket\_tr,   
 topN=10,   
 type="absolute",   
 xlab="Count",   
 ylab="",  
 col="blue",   
 main="Top 10 items",  
 cex.names=0.8,  
 horiz=TRUE)



We can confirm that this frequency plot is the same as what we generated before with our EDA step.

Association rule mining consists of two subtasks which provide insights into the relationships between items in transactional data:

* **Frequent itemset generation**
* **Rule generation**

##### Support

First we’ll take a look at the itemsets in the data to identify the itemsets that occur most frequently. An **itemset** is a combination of items that appear together in a set of transactions. The **support** measures the frequency or occurrence of an itemset in the data by dividing the number of transactions that contain a specific itemset by the total number of transactions. High support indicates the itemset appears frequently.

# Frequent itemsets for all items   
# Interest Measures  
support <- 0.01  
  
parameters = list(  
 support = support,  
 minlen = 2, # Minimal number of items per item set  
 maxlen = 10, # Maximal number of items per item set  
 target = "frequent itemsets"  
 )  
  
support\_all <- apriori(basket\_tr,  
 parameter = parameters)

## Apriori  
##   
## Parameter specification:  
## confidence minval smax arem aval originalSupport maxtime support minlen  
## NA 0.1 1 none FALSE TRUE 5 0.01 2  
## maxlen target ext  
## 10 frequent itemsets TRUE  
##   
## Algorithmic control:  
## filter tree heap memopt load sort verbose  
## 0.1 TRUE TRUE FALSE TRUE 2 TRUE  
##   
## Absolute minimum support count: 98   
##   
## set item appearances ...[0 item(s)] done [0.00s].  
## set transactions ...[169 item(s), 9835 transaction(s)] done [0.00s].  
## sorting and recoding items ... [88 item(s)] done [0.00s].  
## creating transaction tree ... done [0.00s].  
## checking subsets of size 1 2 3 4 done [0.00s].  
## sorting transactions ... done [0.00s].  
## writing ... [245 set(s)] done [0.00s].  
## creating S4 object ... done [0.00s].

# 10 most frequent items  
support\_all |>  
 as("data.frame") |>  
 arrange(desc(support)) |>  
 head(5) |>  
 flextable::flextable() |>  
 flextable::width(width = 2) |>  
 flextable::fontsize(size = 10) |>  
 flextable::line\_spacing(space = 1) |>  
 flextable::hline(part = "all")

| items | support | count |
| --- | --- | --- |
| {other vegetables,whole milk} | 0.07483477 | 736 |
| {rolls/buns,whole milk} | 0.05663447 | 557 |
| {whole milk,yogurt} | 0.05602440 | 551 |
| {root vegetables,whole milk} | 0.04890696 | 481 |
| {other vegetables,root vegetables} | 0.04738180 | 466 |

Interpreting these results, the support column indicates the proportion of transactions that contain each itemset: 7.5% of all transactions contain “other vegetables” and “whole milk”, 5.7% of all transactions contain “other vegetables”rolls/buns” and “whole milk”, and so on.

##### Rules

Extracting rules allows us to observe the support, confidence, and lift measures of frequent itemsets.

conf = 0.4  
support <- 0.01  
  
parameters = list(  
 support = support,  
 confidence = conf,  
 minlen = 2, # Minimal number of items per item set  
 maxlen = 10, # Maximal number of items per item set  
 target = "rules"  
 )  
  
rules <- apriori(basket\_tr,  
 parameter= parameters)

## Apriori  
##   
## Parameter specification:  
## confidence minval smax arem aval originalSupport maxtime support minlen  
## 0.4 0.1 1 none FALSE TRUE 5 0.01 2  
## maxlen target ext  
## 10 rules TRUE  
##   
## Algorithmic control:  
## filter tree heap memopt load sort verbose  
## 0.1 TRUE TRUE FALSE TRUE 2 TRUE  
##   
## Absolute minimum support count: 98   
##   
## set item appearances ...[0 item(s)] done [0.00s].  
## set transactions ...[169 item(s), 9835 transaction(s)] done [0.00s].  
## sorting and recoding items ... [88 item(s)] done [0.00s].  
## creating transaction tree ... done [0.00s].  
## checking subsets of size 1 2 3 4 done [0.00s].  
## writing ... [62 rule(s)] done [0.00s].  
## creating S4 object ... done [0.00s].

The arulesViz package contains a number of functions that enhance graphical display of association rules. The inspectDT function outputs an interactive display which can be sorted by column:

inspectDT(rules)

## PhantomJS not found. You can install it with webshot::install\_phantomjs(). If it is installed, please make sure the phantomjs executable can be found via the PATH variable.

Plotting the rules with the engine = "plotly" argument creates an interactive visualization that allows the user to hover over data points in order to view association rule metrics,

plot(rules, jitter=2, engine = "plotly")

##### Confidence

Each itemset consists of antecedents and consequents. **Antecedents** are the items or itemsets on the left-hand side of the association rule that represents a set of items or conditions that act as the premise or condition for the rule. **Consequents** are the items or itemsets on the right-hand side of the association rule that represents the set of items or outcomes that are predicted or observed based on the presence of the antecent. **Confidence** measures the likelihood or probability of finding the consequent item(s) in a rule given the antecedent(s). It is the proportion of transactions containing the antecedent that also contain the consequent. High confidence suggests a strong association between the antecedent and consequent.

rules |>  
 as("data.frame") |>  
 arrange(desc(confidence)) |>  
 head(10) |>  
 flextable::flextable() |>  
 flextable::width(width = 1) |>  
 flextable::fontsize(size = 10) |>  
 flextable::line\_spacing(space = 1) |>  
 flextable::hline(part = "all")

| rules | support | confidence | coverage | lift | count |
| --- | --- | --- | --- | --- | --- |
| {citrus fruit,root vegetables} => {other vegetables} | 0.01037112 | 0.5862069 | 0.01769192 | 3.029608 | 102 |
| {root vegetables,tropical fruit} => {other vegetables} | 0.01230300 | 0.5845411 | 0.02104728 | 3.020999 | 121 |
| {curd,yogurt} => {whole milk} | 0.01006609 | 0.5823529 | 0.01728521 | 2.279125 | 99 |
| {butter,other vegetables} => {whole milk} | 0.01148958 | 0.5736041 | 0.02003050 | 2.244885 | 113 |
| {root vegetables,tropical fruit} => {whole milk} | 0.01199797 | 0.5700483 | 0.02104728 | 2.230969 | 118 |
| {root vegetables,yogurt} => {whole milk} | 0.01453991 | 0.5629921 | 0.02582613 | 2.203354 | 143 |
| {domestic eggs,other vegetables} => {whole milk} | 0.01230300 | 0.5525114 | 0.02226741 | 2.162336 | 121 |
| {whipped/sour cream,yogurt} => {whole milk} | 0.01087951 | 0.5245098 | 0.02074225 | 2.052747 | 107 |
| {rolls/buns,root vegetables} => {whole milk} | 0.01270971 | 0.5230126 | 0.02430097 | 2.046888 | 125 |
| {other vegetables,pip fruit} => {whole milk} | 0.01352313 | 0.5175097 | 0.02613116 | 2.025351 | 133 |

We can interpret the confidence that 58% of customers that bought “citrus fruit” and “root vegetables” also bought “other vegetables”, and so on.

##### Lift

Lift measures the strength of association between the antecedent and consequent items compared to their individual occurrences. It is the ratio of the observed support to the expected support if the antecedent and consequent were independent. Lift greater than 1 indicates a positive association, while lift less than 1 indicates a negative association.

rules |>  
 as("data.frame") |>  
 arrange(desc(lift)) |>  
 head(10) |>  
 flextable::flextable() |>  
 flextable::width(width = 1) |>  
 flextable::fontsize(size = 10) |>  
 flextable::line\_spacing(space = 1) |>  
 flextable::hline(part = "all")

| rules | support | confidence | coverage | lift | count |
| --- | --- | --- | --- | --- | --- |
| {citrus fruit,root vegetables} => {other vegetables} | 0.01037112 | 0.5862069 | 0.01769192 | 3.029608 | 102 |
| {root vegetables,tropical fruit} => {other vegetables} | 0.01230300 | 0.5845411 | 0.02104728 | 3.020999 | 121 |
| {rolls/buns,root vegetables} => {other vegetables} | 0.01220132 | 0.5020921 | 0.02430097 | 2.594890 | 120 |
| {root vegetables,yogurt} => {other vegetables} | 0.01291307 | 0.5000000 | 0.02582613 | 2.584078 | 127 |
| {whipped/sour cream,yogurt} => {other vegetables} | 0.01016777 | 0.4901961 | 0.02074225 | 2.533410 | 100 |
| {root vegetables,whole milk} => {other vegetables} | 0.02318251 | 0.4740125 | 0.04890696 | 2.449770 | 228 |
| {onions} => {other vegetables} | 0.01423488 | 0.4590164 | 0.03101169 | 2.372268 | 140 |
| {pork,whole milk} => {other vegetables} | 0.01016777 | 0.4587156 | 0.02216573 | 2.370714 | 100 |
| {whipped/sour cream,whole milk} => {other vegetables} | 0.01464159 | 0.4542587 | 0.03223183 | 2.347679 | 144 |
| {pip fruit,whole milk} => {other vegetables} | 0.01352313 | 0.4493243 | 0.03009659 | 2.322178 | 133 |

Again, looking at the first line, we can interpret the lift of 3.03 to represent the occurrence of the antecedent is 3.03 times *likelier* when the antecedent items are present, and so on.