DATA624 Homework #2

Esteban Aramayo, Coffy Andrews-Guo, LeTicia Cancel, Joseph Connolly, Ian Costello

2022-07-13

Table of Contents

[Libraries 1](#_Toc108618841)

[Week 4 2](#_Toc108618842)

[KJ 6.3 2](#_Toc108618843)

[Week 5 14](#_Toc108618844)

[KJ 7.2 (Model Simulations) 14](#_Toc108618845)

[KJ 7.5 (Nonlinear Regression Models) 23](#_Toc108618846)

[KJ 8.1 (Regression Trees and Rule-Based Models) 35](#_Toc108618847)

[### b. Now add an additional predictor that is highly correlated with one of the informative predictors. For example: 37](#_Toc108618848)

[KJ 8.2 (Regression Trees and Rule-Based Models) 43](#_Toc108618849)

[KJ 8.3 (Regression Trees and Rule-Based Models) 46](#_Toc108618850)

[KJ 8.7 (Regression Trees and Rule-Based Models) 47](#_Toc108618851)

[Week 6 53](#_Toc108618852)

[HW BATCH #2 - Market Basket Analysis / Recommender Systems (a simple problem) 53](#_Toc108618853)

# Libraries

Loading in the libraries referenced in Sect. 6.5.

library(webshot)  
install\_phantomjs(force = TRUE)

library(elasticnet)  
library(caret)  
library(MASS)  
library(lars)  
library(stats)  
library(pls)  
library(tidyverse)  
library(dplyr)  
library(RANN)  
library(GGally)  
library(naniar)  
library(party)  
library(Cubist)  
library(gbm)  
library(randomForest)  
library(tibble)

# Week 4

## KJ 6.3 (Linear Regression Models)

*A chemical manufacturing process for a pharmaceutical product was discussed in Sec. 1.4. In this problem, the objective is to understand the relationship between biological measurements of the raw materials (predictors), measurements of the manufacturing process (predictors), and the response of product yield. Biological predictors cannot be changed but can be used to assess the quality of the raw material before processing. On the other hand, manufacturing process predictors can be changed in the manufacturing process. Improving product yield by 1% will boost revenue by approximately one hundred thousand dollars per batch ($100,000/batch):*

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

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

library(AppliedPredictiveModeling)  
data(ChemicalManufacturingProcess)  
df\_raw <- ChemicalManufacturingProcess

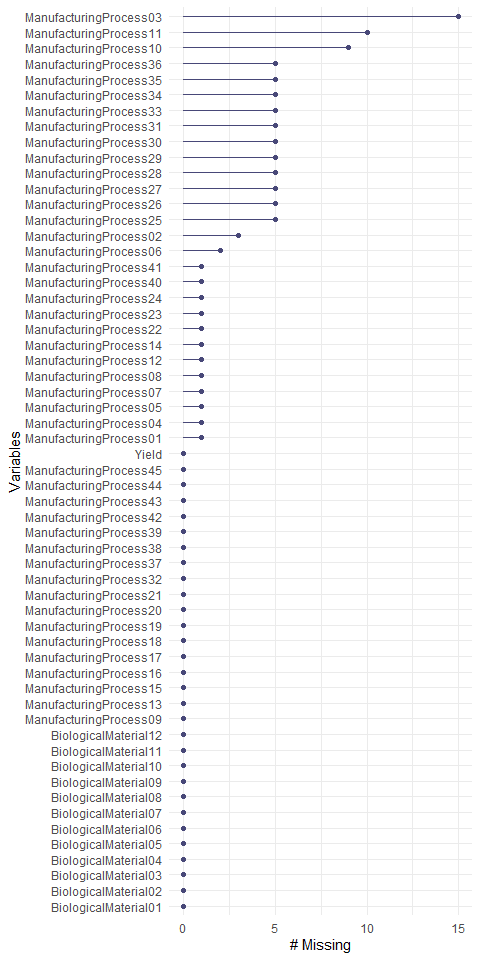
### b. A small percentage of cells in the predictor set contain missing values. Use the imputation function to fill in these missing values (e.g., see Sec. 3.8).

For the imputation, the caret package’s preProcess() function is used. From the chart before imputation, the missing values are entirely part of the manufacturing category of variables. The median is used for the imputation.

paste0("Missing Values: ",sum(is.na(ChemicalManufacturingProcess)))

#> [1] "Missing Values: 106"

gg\_miss\_var(df\_raw)



preProcess\_NAdata\_model <- preProcess(df\_raw, method ="medianImpute")  
df <- predict(preProcess\_NAdata\_model, newdata = df\_raw)

paste0(sum(is.na(df))," values missing after imputation")

#> [1] "0 values missing after imputation"

### c. Split the data into a training and 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?

**Code References:** [**https://topepo.github.io/caret/pre-processing.html#the-preprocess-function**](https://topepo.github.io/caret/pre-processing.html#the-preprocess-function)

Continuing with the caret package, the createDataPartition() function will allow the data set to be split into a training and test set including a parameter for the ratio of training and test data.

#### Splitting data in to training and test set

# Create the training and test datasets  
set.seed(100)  
  
# Step 1: Get row numbers for the training data  
trainRowNumbers <- createDataPartition(df$Yield, p=0.8, list=FALSE)  
  
# Step 2: Create the training dataset  
trainData <- df[trainRowNumbers,]  
  
# Step 3: Create the test dataset  
testData <- df[-trainRowNumbers,]

#### Data preprocess

The preProcess function is a useful Swiss army knife tool for a number of preprocessing steps, including centering and scaling the dataset is one line of code.

preProcValues <- preProcess(trainData, method = c("center", "scale"))  
  
trainTransformed <- predict(preProcValues, trainData)  
testTransformed <- predict(preProcValues, testData)  
  
xTrain <- trainTransformed %>% dplyr::select(-Yield)  
  
yTrain <- trainTransformed %>% dplyr::select(Yield)  
  
xTest <- testTransformed %>% dplyr::select(-Yield)  
  
yTest <- testTransformed %>% dplyr::select(Yield)

#### Tune model

In caret’s train() function the trControl() function and arguments are used to tune the model. In this case cross validation with five folds are indicated.

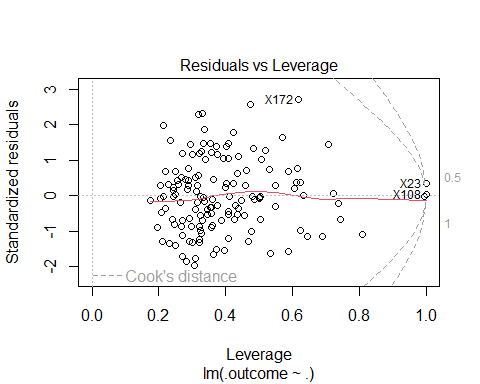
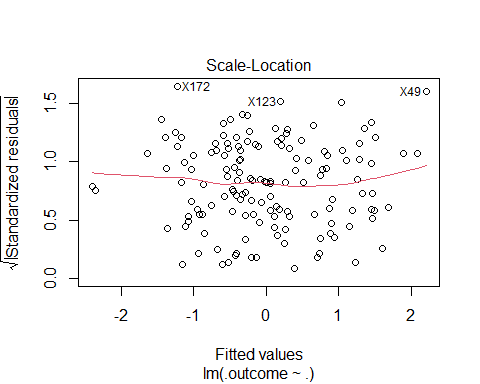
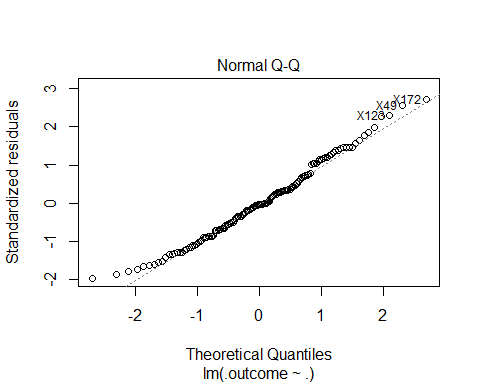
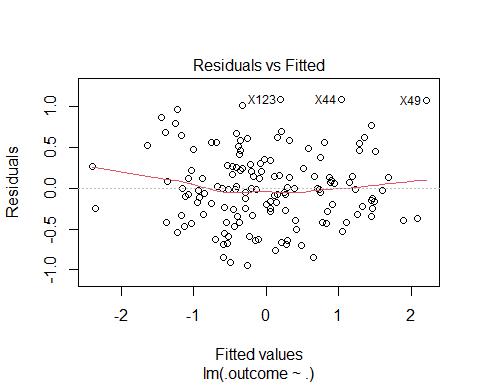
lm\_model <- train(xTrain, yTrain$Yield, method="lm", trControl=trainControl(method="repeatedcv",repeats=5) )  
lm\_model

#> Linear Regression   
#>   
#> 144 samples  
#> 57 predictor  
#>   
#> No pre-processing  
#> Resampling: Cross-Validated (10 fold, repeated 5 times)   
#> Summary of sample sizes: 130, 130, 130, 130, 130, 129, ...   
#> Resampling results:  
#>   
#> RMSE Rsquared MAE   
#> 1.761794 0.4240885 0.9535992  
#>   
#> Tuning parameter 'intercept' was held constant at a value of TRUE

summary(lm\_model)

#>   
#> Call:  
#> lm(formula = .outcome ~ ., data = dat)  
#>   
#> Residuals:  
#> Min 1Q Median 3Q Max   
#> -0.94229 -0.32402 -0.01611 0.27082 1.08863   
#>   
#> Coefficients: (1 not defined because of singularities)  
#> Estimate Std. Error t value Pr(>|t|)   
#> (Intercept) 5.976e-15 4.806e-02 0.000 1.000000   
#> BiologicalMaterial01 -4.427e-02 1.526e-01 -0.290 0.772353   
#> BiologicalMaterial02 -3.308e-01 3.389e-01 -0.976 0.331731   
#> BiologicalMaterial03 4.753e-01 6.692e-01 0.710 0.479458   
#> BiologicalMaterial04 3.532e-01 5.446e-01 0.649 0.518351   
#> BiologicalMaterial05 2.226e-01 1.359e-01 1.638 0.105019   
#> BiologicalMaterial06 -4.333e-01 7.684e-01 -0.564 0.574310   
#> BiologicalMaterial07 -6.008e-02 6.591e-02 -0.912 0.364529   
#> BiologicalMaterial08 3.743e-01 3.249e-01 1.152 0.252400   
#> BiologicalMaterial09 -2.401e-01 4.308e-01 -0.557 0.578744   
#> BiologicalMaterial10 -2.577e-01 4.919e-01 -0.524 0.601723   
#> BiologicalMaterial11 -4.132e-01 2.634e-01 -1.569 0.120324   
#> BiologicalMaterial12 3.514e-01 3.274e-01 1.073 0.286024   
#> ManufacturingProcess01 6.577e-02 1.126e-01 0.584 0.560733   
#> ManufacturingProcess02 -1.436e-01 2.235e-01 -0.642 0.522426   
#> ManufacturingProcess03 -5.261e-02 6.781e-02 -0.776 0.439911   
#> ManufacturingProcess04 2.078e-01 1.272e-01 1.634 0.105899   
#> ManufacturingProcess05 2.950e-02 7.370e-02 0.400 0.689959   
#> ManufacturingProcess06 6.251e-02 6.940e-02 0.901 0.370186   
#> ManufacturingProcess07 -1.235e-01 6.963e-02 -1.773 0.079705 .   
#> ManufacturingProcess08 -3.101e-02 8.605e-02 -0.360 0.719467   
#> ManufacturingProcess09 2.693e-01 2.002e-01 1.345 0.182045   
#> ManufacturingProcess10 -4.858e-01 3.447e-01 -1.410 0.162224   
#> ManufacturingProcess11 1.235e-01 3.918e-01 0.315 0.753313   
#> ManufacturingProcess12 1.037e-01 1.158e-01 0.895 0.373017   
#> ManufacturingProcess13 -6.186e-01 3.041e-01 -2.034 0.044980 \*   
#> ManufacturingProcess14 -1.188e-01 3.556e-01 -0.334 0.739056   
#> ManufacturingProcess15 4.393e-02 3.428e-01 0.128 0.898317   
#> ManufacturingProcess16 1.796e-01 1.131e-01 1.588 0.115923   
#> ManufacturingProcess17 3.498e-01 2.979e-01 1.174 0.243426   
#> ManufacturingProcess18 1.359e-01 4.543e-01 0.299 0.765598   
#> ManufacturingProcess19 -4.053e-02 3.698e-01 -0.110 0.912976   
#> ManufacturingProcess20 -9.016e-02 2.693e-01 -0.335 0.738580   
#> ManufacturingProcess21 NA NA NA NA   
#> ManufacturingProcess22 -3.489e-02 9.213e-02 -0.379 0.705790   
#> ManufacturingProcess23 -5.067e-02 8.900e-02 -0.569 0.570635   
#> ManufacturingProcess24 -6.569e-02 8.834e-02 -0.744 0.459147   
#> ManufacturingProcess25 -1.104e+00 3.925e+00 -0.281 0.779124   
#> ManufacturingProcess26 2.494e+00 4.122e+00 0.605 0.546699   
#> ManufacturingProcess27 -1.853e+00 2.634e+00 -0.704 0.483592   
#> ManufacturingProcess28 -2.508e-01 1.103e-01 -2.273 0.025516 \*   
#> ManufacturingProcess29 7.245e-01 1.024e+00 0.708 0.481071   
#> ManufacturingProcess30 -1.634e-01 3.769e-01 -0.434 0.665622   
#> ManufacturingProcess31 -7.095e-02 4.416e-01 -0.161 0.872739   
#> ManufacturingProcess32 8.601e-01 2.142e-01 4.015 0.000126 \*\*\*  
#> ManufacturingProcess33 -4.972e-01 1.947e-01 -2.553 0.012409 \*   
#> ManufacturingProcess34 -2.865e-02 9.031e-02 -0.317 0.751788   
#> ManufacturingProcess35 -1.008e-01 1.187e-01 -0.849 0.398053   
#> ManufacturingProcess36 1.202e-01 1.611e-01 0.746 0.457671   
#> ManufacturingProcess37 -1.722e-01 7.867e-02 -2.189 0.031260 \*   
#> ManufacturingProcess38 -3.440e-02 9.206e-02 -0.374 0.709540   
#> ManufacturingProcess39 -5.994e-03 1.153e-01 -0.052 0.958678   
#> ManufacturingProcess40 2.272e-01 1.574e-01 1.444 0.152465   
#> ManufacturingProcess41 -1.827e-01 1.663e-01 -1.099 0.274874   
#> ManufacturingProcess42 -1.197e-01 2.447e-01 -0.489 0.626108   
#> ManufacturingProcess43 7.436e-02 9.718e-02 0.765 0.446220   
#> ManufacturingProcess44 1.021e-01 2.436e-01 0.419 0.676172   
#> ManufacturingProcess45 1.261e-01 1.289e-01 0.978 0.330755   
#> ---  
#> Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1  
#>   
#> Residual standard error: 0.5767 on 87 degrees of freedom  
#> Multiple R-squared: 0.7976, Adjusted R-squared: 0.6674   
#> F-statistic: 6.124 on 56 and 87 DF, p-value: 3.995e-14

plot(lm\_model$finalModel)



### 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 of the training set?

**Resampling examples from Sect. 4.9 of the text.**

Predictions are made using the test data set and below is the table with the models’ performance values. The resampled performance metric of the test set is much better than that of the training data set. The RMSE for the test set is smaller than that of the training one.

finalLMPred <- predict(lm\_model$finalModel, newdata = xTest)  
## The function 'postResample' can be used to get the test set  
## performance values  
postResample(pred = finalLMPred, obs = yTest$Yield)

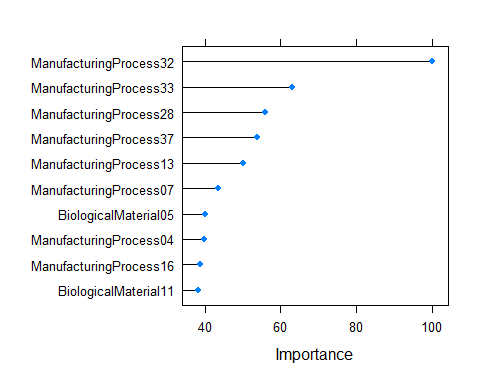
#> RMSE Rsquared MAE   
#> 1.2731407 0.1443080 0.7636275

| Model | RMSE | Rsquared | MAE |
| --- | --- | --- | --- |
| Linear Model (Training) | 1.7617940 | 0.4240885 | 0.9535992 |
| Linear Model (Test) | 1.2731407 | 0.1443080 | 0.7636275 |

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

Below is a list of the top 10 most important predictors for the Linear Regression model. The manugacturing process predictors are dominate in this list, with only two biological factors appearing.

plot(caret::varImp(lm\_model), top = 10)



caret::varImp(lm\_model, scale = FALSE)

#> lm variable importance  
#>   
#> only 20 most important variables shown (out of 56)  
#>   
#> Overall  
#> ManufacturingProcess32 4.0149  
#> ManufacturingProcess33 2.5534  
#> ManufacturingProcess28 2.2726  
#> ManufacturingProcess37 2.1892  
#> ManufacturingProcess13 2.0342  
#> ManufacturingProcess07 1.7731  
#> BiologicalMaterial05 1.6381  
#> ManufacturingProcess04 1.6339  
#> ManufacturingProcess16 1.5880  
#> BiologicalMaterial11 1.5688  
#> ManufacturingProcess40 1.4435  
#> ManufacturingProcess10 1.4096  
#> ManufacturingProcess09 1.3452  
#> ManufacturingProcess17 1.1744  
#> BiologicalMaterial08 1.1522  
#> ManufacturingProcess41 1.0988  
#> BiologicalMaterial12 1.0735  
#> ManufacturingProcess45 0.9781  
#> BiologicalMaterial02 0.9761  
#> BiologicalMaterial07 0.9115

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

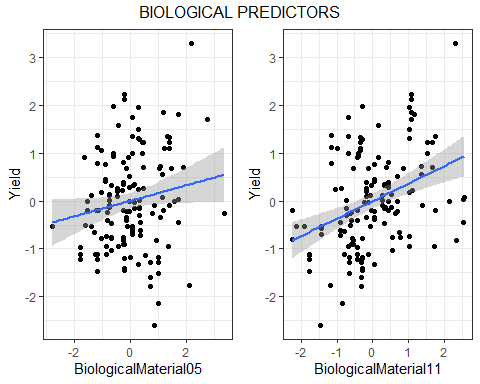
We use scatter plots between the top 10 predictors of the model and the response variable.

Some conclusions:

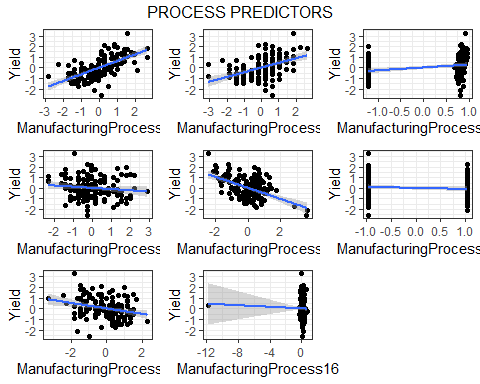
* The 2 biological predictors BiologicalMaterial05 and BiologicalMaterial11 appear to have a low positive correlation to the response.
* The top 2 predictors appears to be ManufacturingProcess32 and ManufacturingProcess33. Which, appear to have a positive correlation with the response.
* The predictors ManufacturingProcess04 and ManufacturingProcess13 seem to have a negative correlation with the response.
* The predictor ManufacturingProcess37 seems to have a low negative correlation with the response.
* The remaining predictors ManufacturingProcess07, ManufacturingProcess16 and ManufacturingProcess28 seem to have no correlation to the response.

Since the biological factors are immutable in terms of yield, there may not be much one can do other than maximizing the stock of product to have the desired biological factors that would produce the higher yield, (e.g., high quality oil with fewer impurities). The manufacturing processes are more controllable. By observing where certain manufacturing processes produce higher yields, the factory may be able to create the right environment to extract more product from the manufacturing processes by maintaining high yield conditions. There is likely some interaction among these variables that may maximize one while diminishing another. Those should be considered individually and analyzed to produce highest *overall* yield, rather than highest yield within each variable.

library(gridExtra)  
  
top10lmPred <- trainTransformed %>%  
 select(Yield,   
 ManufacturingProcess32,  
 ManufacturingProcess33,  
 ManufacturingProcess28,  
 ManufacturingProcess37,  
 ManufacturingProcess13,  
 ManufacturingProcess07,  
 BiologicalMaterial05,  
 ManufacturingProcess04,  
 ManufacturingProcess16,  
 BiologicalMaterial11)  
  
plt01 <- top10lmPred %>%  
 ggplot(aes(x = ManufacturingProcess32, y = Yield)) +  
 geom\_point() + geom\_smooth(method = "lm") + theme\_bw()  
  
plt02 <- top10lmPred %>%  
 ggplot(aes(x = ManufacturingProcess33, y = Yield)) +  
 geom\_point()+ geom\_smooth(method = "lm") + theme\_bw()  
  
plt03 <- top10lmPred %>%  
 ggplot(aes(x = ManufacturingProcess28, y = Yield)) +  
 geom\_point() + geom\_smooth(method = "lm") + theme\_bw()  
  
plt04 <- top10lmPred %>%  
 ggplot(aes(x = ManufacturingProcess37, y = Yield)) +  
 geom\_point() + geom\_smooth(method = "lm") + theme\_bw()  
  
plt05 <- top10lmPred %>%  
 ggplot(aes(x = ManufacturingProcess13, y = Yield)) +  
 geom\_point() + geom\_smooth(method = "lm") + theme\_bw()  
  
plt06 <- top10lmPred %>%  
 ggplot(aes(x = ManufacturingProcess07, y = Yield)) +  
 geom\_point() + geom\_smooth(method = "lm") + theme\_bw()  
  
plt07 <- top10lmPred %>%  
 ggplot(aes(x = BiologicalMaterial05, y = Yield)) +  
 geom\_point() + geom\_smooth(method = "lm") + theme\_bw()  
  
plt08 <- top10lmPred %>%  
 ggplot(aes(x = ManufacturingProcess04, y = Yield)) +  
 geom\_point() + geom\_smooth(method = "lm") + theme\_bw()  
  
plt09 <- top10lmPred %>%  
 ggplot(aes(x = ManufacturingProcess16, y = Yield)) +  
 geom\_point() + geom\_smooth(method = "lm") + theme\_bw()  
  
plt10 <- top10lmPred %>%  
 ggplot(aes(x = BiologicalMaterial11, y = Yield)) +  
 geom\_point() + geom\_smooth(method = "lm") + theme\_bw()  
  
# Biological Predictors  
grid.arrange(plt07, plt10, nrow = 1, top = "BIOLOGICAL PREDICTORS")



# Process Predictors  
grid.arrange(plt01, plt02, plt03, plt04, plt05, plt06, plt08, plt09,  
 nrow = 3, top = "PROCESS PREDICTORS")



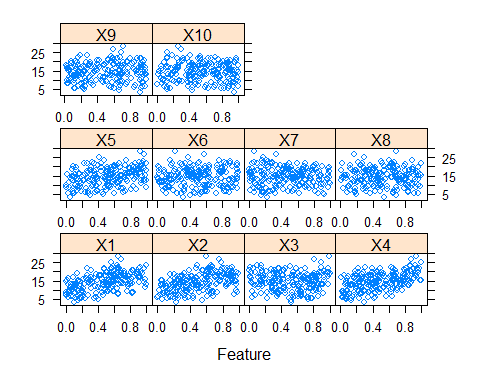
# Week 5

## KJ 7.2 (Model Simulations)

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

*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(caret)  
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  
caret::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:  
testingData <- mlbench.friedman1(5000, sd = 1)  
testingData$x <- data.frame(testingData$x)

**Tune several models on these data. For example:**

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

#> k-Nearest Neighbors   
#>   
#> 200 samples  
#> 10 predictor  
#>   
#> Pre-processing: centered (10), scaled (10)   
#> Resampling: Bootstrapped (25 reps)   
#> Summary of sample sizes: 200, 200, 200, 200, 200, 200, ...   
#> Resampling results across tuning parameters:  
#>   
#> k RMSE Rsquared MAE   
#> 5 3.466085 0.5121775 2.816838  
#> 7 3.349428 0.5452823 2.727410  
#> 9 3.264276 0.5785990 2.660026  
#> 11 3.214216 0.6024244 2.603767  
#> 13 3.196510 0.6176570 2.591935  
#> 15 3.184173 0.6305506 2.577482  
#> 17 3.183130 0.6425367 2.567787  
#> 19 3.198752 0.6483184 2.592683  
#> 21 3.188993 0.6611428 2.588787  
#> 23 3.200458 0.6638353 2.604529  
#>   
#> RMSE was used to select the optimal model using the smallest value.  
#> The final value used for the model was k = 17.

knnPred <- predict(knnModel, newdata = testingData$x)  
## The function 'postResample' can be used to get the test set  
## perforamnce values  
postResample(pred = knnPred, obs = testingData$y)

#> RMSE Rsquared MAE   
#> 3.2040595 0.6819919 2.5683461

*Which models appear to give the best performance? Does MARS select the informative predictors (those named X1–X5)?*

#### Using a Neural Network model

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

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

#> integer(0)

Since the correlation vector is empty, there are no predictors to remove. We can now fit a neural network model using the existing training set.

# Resource: Chapter 7.5 of textbook "Applied Predictive Modeling" by KJ  
  
## Create a specific candidate set of models to evaluate:  
nnetGrid <- expand.grid(.decay = c(0, 0.01, .1),  
 .size = c(1:10),  
 .bag = FALSE)  
  
# We can use caret to perform a grid search using 10-fold cross-validation.  
# The train function generates a resampling estimate of performance. Because  
# the training set size is not small, 10-fold cross-validation should produce  
# reasonable estimates of model performance. The function trainControl specifies  
# the type of resampling:  
ctrl <- trainControl(method = "cv", number = 10)  
  
nnetTune <- train(trainingData$x, trainingData$y,  
 method = "avNNet",  
 tuneGrid = nnetGrid,  
 trControl = ctrl,  
 ## Automatically standardize data prior to modeling  
 ## and prediction  
 preProc = c("center", "scale"),  
 linout = TRUE,  
 trace = FALSE,  
 MaxNWts = 10 \* (ncol(trainingData$x) + 1) + 10 + 1,  
 maxit = 500)  
  
nnetTune

#> Model Averaged Neural Network   
#>   
#> 200 samples  
#> 10 predictor  
#>   
#> Pre-processing: centered (10), scaled (10)   
#> Resampling: Cross-Validated (10 fold)   
#> Summary of sample sizes: 180, 180, 180, 180, 180, 180, ...   
#> Resampling results across tuning parameters:  
#>   
#> decay size RMSE Rsquared MAE   
#> 0.00 1 2.434845 0.7683498 1.921367  
#> 0.00 2 2.497822 0.7558233 1.993325  
#> 0.00 3 2.037977 0.8419435 1.609574  
#> 0.00 4 1.900290 0.8584647 1.530058  
#> 0.00 5 2.169029 0.8107813 1.622718  
#> 0.00 6 2.739060 0.7277149 1.989124  
#> 0.00 7 3.537776 0.6416846 2.529717  
#> 0.00 8 4.039108 0.5937856 2.753346  
#> 0.00 9 4.166036 0.5182801 2.781335  
#> 0.00 10 4.681629 0.5844637 2.819934  
#> 0.01 1 2.437231 0.7689665 1.934978  
#> 0.01 2 2.510986 0.7596191 1.988260  
#> 0.01 3 1.999943 0.8419567 1.555751  
#> 0.01 4 2.003357 0.8445289 1.549723  
#> 0.01 5 2.104800 0.8296460 1.664980  
#> 0.01 6 2.314702 0.7997310 1.857948  
#> 0.01 7 2.341043 0.8072422 1.872533  
#> 0.01 8 2.208391 0.8151328 1.751487  
#> 0.01 9 2.263137 0.8145980 1.776611  
#> 0.01 10 2.453311 0.7709663 1.981977  
#> 0.10 1 2.450897 0.7652309 1.942945  
#> 0.10 2 2.489399 0.7606443 1.997060  
#> 0.10 3 2.200693 0.8155496 1.786599  
#> 0.10 4 2.059322 0.8432340 1.651716  
#> 0.10 5 2.189025 0.8133603 1.729453  
#> 0.10 6 2.215091 0.8128993 1.757966  
#> 0.10 7 2.209518 0.8196477 1.786770  
#> 0.10 8 2.330446 0.7976592 1.834082  
#> 0.10 9 2.286687 0.7928465 1.848973  
#> 0.10 10 2.238971 0.8114608 1.788072  
#>   
#> Tuning parameter 'bag' was held constant at a value of FALSE  
#> RMSE was used to select the optimal model using the smallest value.  
#> The final values used for the model were size = 4, decay = 0 and bag = FALSE.

Predict using the test data set and get the model’s performance values.

nnetPred <- predict(nnetTune, newdata = testingData$x)  
## The function 'postResample' can be used to get the test set  
## performance values  
postResample(pred = nnetPred, obs = testingData$y)

#> RMSE Rsquared MAE   
#> 2.496722 0.784618 1.685182

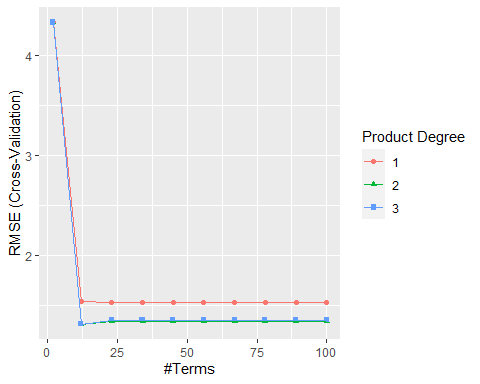
#### Using a Multivariate Adaptive Regression Splines (MARS) model

# Resource: Chapter 7.5 of textbook "Applied Predictive Modeling" by KJ  
# Resource: http://uc-r.github.io/mars  
  
library(earth)  
library(dplyr)  
  
## Create a specific candidate set of models to evaluate:  
marsGrid <- expand.grid(degree = 1:3,  
 nprune = seq(2, 100, length.out = 10) %>% floor()  
 )  
  
# We can use caret to perform a grid search using 10-fold cross-validation.  
# The train function generates a resampling estimate of performance. Because  
# the training set size is not small, 10-fold cross-validation should produce  
# reasonable estimates of model performance. The function trainControl specifies  
# the type of resampling:  
ctrl <- trainControl(method = "cv", number = 10)  
  
  
# cross validated model  
tuned\_mars <- train(  
 x = trainingData$x,  
 y = trainingData$y,  
 method = "earth",  
 metric = "RMSE",  
 trControl = ctrl,  
 tuneGrid = marsGrid  
)  
  
tuned\_mars

#> Multivariate Adaptive Regression Spline   
#>   
#> 200 samples  
#> 10 predictor  
#>   
#> No pre-processing  
#> Resampling: Cross-Validated (10 fold)   
#> Summary of sample sizes: 180, 180, 180, 180, 180, 180, ...   
#> Resampling results across tuning parameters:  
#>   
#> degree nprune RMSE Rsquared MAE   
#> 1 2 4.334325 0.2599883 3.607719  
#> 1 12 1.545349 0.8992979 1.243771  
#> 1 23 1.529405 0.9018457 1.223874  
#> 1 34 1.529405 0.9018457 1.223874  
#> 1 45 1.529405 0.9018457 1.223874  
#> 1 56 1.529405 0.9018457 1.223874  
#> 1 67 1.529405 0.9018457 1.223874  
#> 1 78 1.529405 0.9018457 1.223874  
#> 1 89 1.529405 0.9018457 1.223874  
#> 1 100 1.529405 0.9018457 1.223874  
#> 2 2 4.334325 0.2599883 3.607719  
#> 2 12 1.312883 0.9284253 1.063321  
#> 2 23 1.340821 0.9285264 1.050274  
#> 2 34 1.340821 0.9285264 1.050274  
#> 2 45 1.340821 0.9285264 1.050274  
#> 2 56 1.340821 0.9285264 1.050274  
#> 2 67 1.340821 0.9285264 1.050274  
#> 2 78 1.340821 0.9285264 1.050274  
#> 2 89 1.340821 0.9285264 1.050274  
#> 2 100 1.340821 0.9285264 1.050274  
#> 3 2 4.334325 0.2599883 3.607719  
#> 3 12 1.312883 0.9284253 1.063321  
#> 3 23 1.356568 0.9267847 1.068755  
#> 3 34 1.356568 0.9267847 1.068755  
#> 3 45 1.356568 0.9267847 1.068755  
#> 3 56 1.356568 0.9267847 1.068755  
#> 3 67 1.356568 0.9267847 1.068755  
#> 3 78 1.356568 0.9267847 1.068755  
#> 3 89 1.356568 0.9267847 1.068755  
#> 3 100 1.356568 0.9267847 1.068755  
#>   
#> RMSE was used to select the optimal model using the smallest value.  
#> The final values used for the model were nprune = 12 and degree = 2.

Plot the MARS model results.

ggplot(tuned\_mars)



Show the best tuned MARS model.

# best chosen MARS model  
tuned\_mars$bestTune

#> nprune degree  
#> 12 12 2

Predict using the test data set and get the model’s performance values.

marsPred <- predict(tuned\_mars, newdata = testingData$x)  
## The function 'postResample' can be used to get the test set  
## performance values  
postResample(pred = marsPred, obs = testingData$y)

#> RMSE Rsquared MAE   
#> 1.2803060 0.9335241 1.0168673

#### Using a Support Vector Machines (SVM) model

# Resource: Chapter 7.5 of textbook "Applied Predictive Modeling" by KJ  
  
library(kernlab)  
  
  
# We can use caret to perform a grid search using 10-fold cross-validation.  
# The train function generates a resampling estimate of performance. Because  
# the training set size is not small, 10-fold cross-validation should produce  
# reasonable estimates of model performance. The function trainControl specifies  
# the type of resampling:  
ctrl <- trainControl(method = "cv", number = 10)  
  
  
svmRTuned <- train(trainingData$x, trainingData$y,  
 method = "svmRadial",  
 preProc = c("center", "scale"),  
 tuneLength = 14,  
 trControl = ctrl)  
  
svmRTuned

#> Support Vector Machines with Radial Basis Function Kernel   
#>   
#> 200 samples  
#> 10 predictor  
#>   
#> Pre-processing: centered (10), scaled (10)   
#> Resampling: Cross-Validated (10 fold)   
#> Summary of sample sizes: 180, 180, 180, 180, 180, 180, ...   
#> Resampling results across tuning parameters:  
#>   
#> C RMSE Rsquared MAE   
#> 0.25 2.469506 0.7985704 1.979900  
#> 0.50 2.217035 0.8125018 1.766726  
#> 1.00 2.051120 0.8325247 1.618837  
#> 2.00 1.965427 0.8450934 1.539666  
#> 4.00 1.897161 0.8563701 1.500170  
#> 8.00 1.862883 0.8615845 1.475808  
#> 16.00 1.861693 0.8623909 1.474581  
#> 32.00 1.861693 0.8623909 1.474581  
#> 64.00 1.861693 0.8623909 1.474581  
#> 128.00 1.861693 0.8623909 1.474581  
#> 256.00 1.861693 0.8623909 1.474581  
#> 512.00 1.861693 0.8623909 1.474581  
#> 1024.00 1.861693 0.8623909 1.474581  
#> 2048.00 1.861693 0.8623909 1.474581  
#>   
#> Tuning parameter 'sigma' was held constant at a value of 0.06192511  
#> RMSE was used to select the optimal model using the smallest value.  
#> The final values used for the model were sigma = 0.06192511 and C = 16.

Predict using the test data set and get the model’s performance values.

svmPred <- predict(svmRTuned, newdata = testingData$x)  
## The function 'postResample' can be used to get the test set  
## performance values  
postResample(pred = svmPred, obs = testingData$y)

#> RMSE Rsquared MAE   
#> 2.0712382 0.8260501 1.5732943

**Which models appear to give the best performance? Does MARS select the informative predictors (those named X1–X5)?**

Comparing all used models we can see that the **MARS** model yields the best performance in terms of the lowest **RMSE = 1.2803060** and highest .

| Model | RMSE | Rsquared | MAE |
| --- | --- | --- | --- |
| KNN | 3.2040595 | 0.6819919 | 2.5683461 |
| Neural Network (avNNet) | 2.1930855 | 0.8113903 | 1.6402075 |
| MARS | **1.2803060** | 0.9335241 | 1.0168673 |
| SVM | 2.0793977 | 0.8249453 | 1.5796370 |

## KJ 7.5 (Nonlinear Regression Models)

**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.**

Load libraries used for Exercise 6.3. The same variables will be used in 7.5 as in 6.3 because all the imputation and splits must be identical.

library(RANN)  
library(dplyr)  
library(naniar)  
library(caret)

#### Fit a KNN model

knnChemModel <- train(x = xTrain,  
 y = yTrain$Yield,  
 method = "knn",  
 preProc = c("center", "scale"),  
 tuneLength = 10)  
  
knnChemModel

#> k-Nearest Neighbors   
#>   
#> 144 samples  
#> 57 predictor  
#>   
#> Pre-processing: centered (57), scaled (57)   
#> Resampling: Bootstrapped (25 reps)   
#> Summary of sample sizes: 144, 144, 144, 144, 144, 144, ...   
#> Resampling results across tuning parameters:  
#>   
#> k RMSE Rsquared MAE   
#> 5 0.8076576 0.3869731 0.6371097  
#> 7 0.8048579 0.3947611 0.6358804  
#> 9 0.8054386 0.3973676 0.6464537  
#> 11 0.8052739 0.4020723 0.6506980  
#> 13 0.8094271 0.4002168 0.6563332  
#> 15 0.8116354 0.4017500 0.6601884  
#> 17 0.8126778 0.4088156 0.6608773  
#> 19 0.8103704 0.4177709 0.6580705  
#> 21 0.8138395 0.4170925 0.6569173  
#> 23 0.8168767 0.4192162 0.6589602  
#>   
#> RMSE was used to select the optimal model using the smallest value.  
#> The final value used for the model was k = 7.

knnChemPred <- predict(knnChemModel, newdata = xTest)  
## The function 'postResample' can be used to get the test set  
## perforamnce values  
postResample(pred = knnChemPred, obs = yTest$Yield)

#> RMSE Rsquared MAE   
#> 0.6680061 0.4205708 0.5286380

#### Fit a Neural Network model

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

tooHigh <- findCorrelation(cor(xTrain), cutoff = .75)  
  
tooHigh

#> [1] 2 4 6 8 1 12 44 31 30 27 41 21 25 26 54 38 37 43 57 52

Since the correlation vector is not empty, there are predictors to be removed. After removal, generate a new training and test sets of predictors.

trainXnnet <- xTrain[, -tooHigh]  
testXnnet <- xTest[, -tooHigh]

We can now fit a neural network model using the existing training set.

# Resource: Chapter 7.5 of textbook "Applied Predictive Modeling" by KJ  
  
## Create a specific candidate set of models to evaluate:  
nnetGrid <- expand.grid(.decay = c(0, 0.01, .1),  
 .size = c(1:10),  
 .bag = FALSE)  
  
# We can use caret to perform a grid search using 10-fold cross-validation.  
# The train function generates a resampling estimate of performance. Because  
# the training set size is not small, 10-fold cross-validation should produce  
# reasonable estimates of model performance. The function trainControl specifies  
# the type of resampling:  
ctrl <- trainControl(method = "cv", number = 10)  
  
nnetChemTune <- train(xTrain, yTrain$Yield,  
 method = "avNNet",  
 tuneGrid = nnetGrid,  
 trControl = ctrl,  
 ## Automatically standardize data prior to modeling  
 ## and prediction  
 preProc = c("center", "scale"),  
 linout = TRUE,  
 trace = FALSE,  
 MaxNWts = 10 \* (ncol(trainXnnet) + 1) + 10 + 1,  
 maxit = 500)  
  
nnetChemTune

#> Model Averaged Neural Network   
#>   
#> 144 samples  
#> 57 predictor  
#>   
#> Pre-processing: centered (57), scaled (57)   
#> Resampling: Cross-Validated (10 fold)   
#> Summary of sample sizes: 129, 131, 130, 128, 129, 129, ...   
#> Resampling results across tuning parameters:  
#>   
#> decay size RMSE Rsquared MAE   
#> 0.00 1 0.7728796 0.4829591 0.6149545  
#> 0.00 2 0.8700784 0.4895926 0.6715234  
#> 0.00 3 0.7014519 0.5725154 0.5585061  
#> 0.00 4 0.8135313 0.5277272 0.6289421  
#> 0.00 5 0.7131235 0.5675221 0.5851470  
#> 0.00 6 0.7470284 0.5307302 0.6021483  
#> 0.00 7 NaN NaN NaN  
#> 0.00 8 NaN NaN NaN  
#> 0.00 9 NaN NaN NaN  
#> 0.00 10 NaN NaN NaN  
#> 0.01 1 0.8105501 0.4946864 0.6686178  
#> 0.01 2 0.7411557 0.5506487 0.5963452  
#> 0.01 3 0.6582150 0.6161927 0.5280322  
#> 0.01 4 0.6380033 0.6452839 0.5240129  
#> 0.01 5 0.6289785 0.6253316 0.5046360  
#> 0.01 6 0.6138951 0.6633995 0.4810821  
#> 0.01 7 NaN NaN NaN  
#> 0.01 8 NaN NaN NaN  
#> 0.01 9 NaN NaN NaN  
#> 0.01 10 NaN NaN NaN  
#> 0.10 1 0.7283694 0.5406900 0.5957306  
#> 0.10 2 0.6705028 0.6255208 0.5366383  
#> 0.10 3 0.6216374 0.6333319 0.4996340  
#> 0.10 4 0.6345348 0.6348586 0.5187677  
#> 0.10 5 0.6169869 0.6526512 0.4896600  
#> 0.10 6 0.6101492 0.6529740 0.4918590  
#> 0.10 7 NaN NaN NaN  
#> 0.10 8 NaN NaN NaN  
#> 0.10 9 NaN NaN NaN  
#> 0.10 10 NaN NaN NaN  
#>   
#> Tuning parameter 'bag' was held constant at a value of FALSE  
#> RMSE was used to select the optimal model using the smallest value.  
#> The final values used for the model were size = 6, decay = 0.1 and bag = FALSE.

Predict using the test data set and get the model’s performance values

nnetChemPred <- predict(nnetChemTune, newdata = xTest)  
## The function 'postResample' can be used to get the test set  
## performance values  
postResample(pred = nnetChemPred, obs = yTest$Yield)

#> RMSE Rsquared MAE   
#> 0.5581889 0.6622600 0.4314306

#### Fit a MARS model

# Resource: Chapter 7.5 of textbook "Applied Predictive Modeling" by KJ  
# Resource: http://uc-r.github.io/mars  
  
library(earth)  
library(dplyr)  
  
## Create a specific candidate set of models to evaluate:  
marsGrid <- expand.grid(degree = 1:3,  
 nprune = seq(2, 100, length.out = 10) %>% floor()  
 )  
  
# We can use caret to perform a grid search using 10-fold cross-validation.  
# The train function generates a resampling estimate of performance. Because  
# the training set size is not small, 10-fold cross-validation should produce  
# reasonable estimates of model performance. The function trainControl specifies  
# the type of resampling:  
ctrl <- trainControl(method = "cv", number = 10)  
  
  
# cross validated model  
tuned\_ChemMARS <- train(  
 x = xTrain,  
 y = yTrain$Yield,  
 method = "earth",  
 metric = "RMSE",  
 trControl = ctrl,  
 tuneGrid = marsGrid  
)  
  
tuned\_ChemMARS

#> Multivariate Adaptive Regression Spline   
#>   
#> 144 samples  
#> 57 predictor  
#>   
#> No pre-processing  
#> Resampling: Cross-Validated (10 fold)   
#> Summary of sample sizes: 128, 132, 130, 129, 130, 129, ...   
#> Resampling results across tuning parameters:  
#>   
#> degree nprune RMSE Rsquared MAE   
#> 1 2 0.7433928 0.4416241 0.5856190  
#> 1 12 1.0185177 0.5009366 0.6555849  
#> 1 23 0.9130706 0.5004732 0.6376020  
#> 1 34 0.9130706 0.5004732 0.6376020  
#> 1 45 0.9130706 0.5004732 0.6376020  
#> 1 56 0.9130706 0.5004732 0.6376020  
#> 1 67 0.9130706 0.5004732 0.6376020  
#> 1 78 0.9130706 0.5004732 0.6376020  
#> 1 89 0.9130706 0.5004732 0.6376020  
#> 1 100 0.9130706 0.5004732 0.6376020  
#> 2 2 0.7433928 0.4416241 0.5856190  
#> 2 12 0.7094493 0.5224383 0.5701014  
#> 2 23 0.7708808 0.4677324 0.6187519  
#> 2 34 0.7655925 0.4703426 0.6162982  
#> 2 45 0.7655925 0.4703426 0.6162982  
#> 2 56 0.7655925 0.4703426 0.6162982  
#> 2 67 0.7655925 0.4703426 0.6162982  
#> 2 78 0.7655925 0.4703426 0.6162982  
#> 2 89 0.7655925 0.4703426 0.6162982  
#> 2 100 0.7655925 0.4703426 0.6162982  
#> 3 2 0.7433928 0.4416241 0.5856190  
#> 3 12 0.7373027 0.5007462 0.5892015  
#> 3 23 0.8697865 0.4377689 0.6477372  
#> 3 34 0.8589071 0.4440262 0.6437533  
#> 3 45 0.8589071 0.4440262 0.6437533  
#> 3 56 0.8589071 0.4440262 0.6437533  
#> 3 67 0.8589071 0.4440262 0.6437533  
#> 3 78 0.8589071 0.4440262 0.6437533  
#> 3 89 0.8589071 0.4440262 0.6437533  
#> 3 100 0.8589071 0.4440262 0.6437533  
#>   
#> RMSE was used to select the optimal model using the smallest value.  
#> The final values used for the model were nprune = 12 and degree = 2.

Predict using the test data set and get the model’s performance values

marsChemPred <- predict(tuned\_ChemMARS, newdata = xTest)  
## The function 'postResample' can be used to get the test set  
## performance values  
postResample(pred = marsChemPred, obs = yTest$Yield)

#> RMSE Rsquared MAE   
#> 0.5908285 0.5934452 0.4823008

#### Fit an SVM model

# Resource: Chapter 7.5 of textbook "Applied Predictive Modeling" by KJ  
  
library(kernlab)  
  
  
# We can use caret to perform a grid search using 10-fold cross-validation.  
# The train function generates a resampling estimate of performance. Because  
# the training set size is not small, 10-fold cross-validation should produce  
# reasonable estimates of model performance. The function trainControl specifies  
# the type of resampling:  
ctrl <- trainControl(method = "cv", number = 10)  
  
  
svmChemRTuned <- train(xTrain, yTrain$Yield,  
 method = "svmRadial",  
 preProc = c("center", "scale"),  
 tuneLength = 14,  
 trControl = ctrl)  
  
svmChemRTuned

#> Support Vector Machines with Radial Basis Function Kernel   
#>   
#> 144 samples  
#> 57 predictor  
#>   
#> Pre-processing: centered (57), scaled (57)   
#> Resampling: Cross-Validated (10 fold)   
#> Summary of sample sizes: 129, 131, 129, 129, 128, 131, ...   
#> Resampling results across tuning parameters:  
#>   
#> C RMSE Rsquared MAE   
#> 0.25 0.7737394 0.4767769 0.6230220  
#> 0.50 0.6995472 0.5395335 0.5666314  
#> 1.00 0.6465165 0.5998309 0.5246415  
#> 2.00 0.6163761 0.6211330 0.4992620  
#> 4.00 0.6011374 0.6331705 0.4903679  
#> 8.00 0.5857338 0.6556706 0.4766805  
#> 16.00 0.5851640 0.6571656 0.4762982  
#> 32.00 0.5851640 0.6571656 0.4762982  
#> 64.00 0.5851640 0.6571656 0.4762982  
#> 128.00 0.5851640 0.6571656 0.4762982  
#> 256.00 0.5851640 0.6571656 0.4762982  
#> 512.00 0.5851640 0.6571656 0.4762982  
#> 1024.00 0.5851640 0.6571656 0.4762982  
#> 2048.00 0.5851640 0.6571656 0.4762982  
#>   
#> Tuning parameter 'sigma' was held constant at a value of 0.01336765  
#> RMSE was used to select the optimal model using the smallest value.  
#> The final values used for the model were sigma = 0.01336765 and C = 16.

Predict using the test data set and get the model’s performance values

svmChemPred <- predict(svmChemRTuned, newdata = xTest)  
## The function 'postResample' can be used to get the test set  
## performance values  
postResample(pred = svmChemPred, obs = yTest$Yield)

#> RMSE Rsquared MAE   
#> 0.5606126 0.5954437 0.4410173

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

Comparing all used models we can see that the **Neural Network (avNNet)** model yields the best performance in terms of the lowest RMSE = 0.5971.

| Model | RMSE | Rsquared | MAE |
| --- | --- | --- | --- |
| KNN | 0.6525019 | 0.4576898 | 0.5469477 |
| Neural Network (avNNet) | \*\*0.5971097\* | 0.6827372 | 0.4650641 |
| MARS | 0.7590228 | 0.2965229 | 0.6104158 |
| SVM | 0.5588608 | 0.5977766 | 0.4423128 |

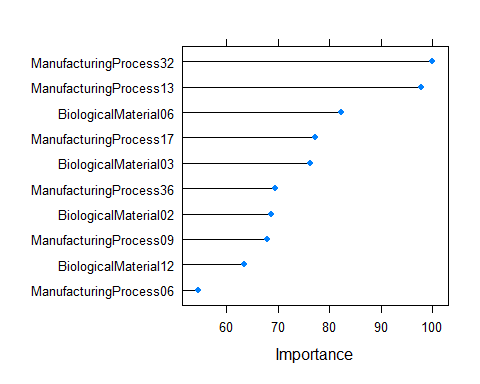
### b. Which predictors are most important in the optimal nonlinear regression model? Do either the biological or process variables dominate the list? How do the top ten important predictors compare to the top ten predictors from the optimal linear model?

* The **Neural Network (avNNet)** model yields the best performance in terms of the lowest RMSE. For that model, below are the predictors in order of importance. A plot of them is also provided.
* The comparison of the top ten most important predictor between the NNET model and the Linear Model shows that the most important predictor is the "ManufacturingProcess32". The rest of the predictors are ranked differently between the 2 models. The NNET model captured only one biological predictor "BiologicalMaterial11" in the top 10 list. While the Linear Model ranked 4 biological predictors in the top 10.

| Importance | Linear Model Top Predictors | Overall | NNET Model Predictors | Overall |
| --- | --- | --- | --- | --- |

`1|ManufacturingProcess32 |4.0148629 | ManufacturingProcess32 |100.00000`  
 2|ManufacturingProcess33 |2.5534232 | ManufacturingProcess13 |97.83640  
 3|ManufacturingProcess28 |2.2725763 | BiologicalMaterial06 |82.21744  
 4|ManufacturingProcess37 |2.1891986 | ManufacturingProcess17 |77.26777  
 5|ManufacturingProcess13 |2.0341703 | BiologicalMaterial03 |76.21094  
 6|ManufacturingProcess07 |1.7731421 | ManufacturingProcess36 |70.96498  
 7|BiologicalMaterial05 |1.6380685 | BiologicalMaterial02 |68.78876  
 8|ManufacturingProcess04 |1.6338725 | ManufacturingProcess09 |67.86384  
 9|ManufacturingProcess16 |1.5879600 | BiologicalMaterial12 |63.36203  
 10|BiologicalMaterial11 |1.5688048 | ManufacturingProcess06 |55.15443

plot(caret::varImp(nnetChemTune), top = 10)

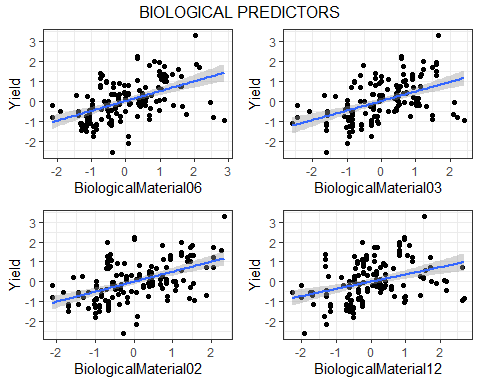


caret::varImp(nnetChemTune)

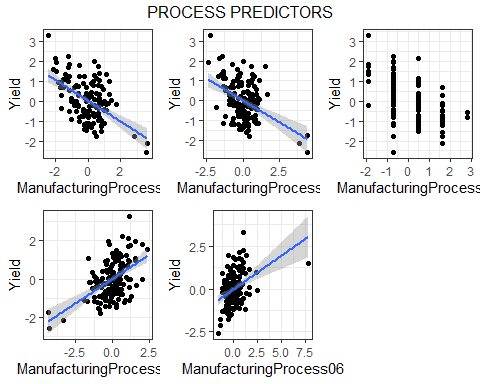
#> loess r-squared variable importance  
#>   
#> only 20 most important variables shown (out of 57)  
#>   
#> Overall  
#> ManufacturingProcess32 100.00  
#> ManufacturingProcess13 97.84  
#> BiologicalMaterial06 82.22  
#> ManufacturingProcess17 77.27  
#> BiologicalMaterial03 76.21  
#> ManufacturingProcess36 69.53  
#> BiologicalMaterial02 68.79  
#> ManufacturingProcess09 67.86  
#> BiologicalMaterial12 63.36  
#> ManufacturingProcess06 54.51  
#> BiologicalMaterial04 54.31  
#> ManufacturingProcess33 49.26  
#> BiologicalMaterial11 42.44  
#> BiologicalMaterial08 41.89  
#> ManufacturingProcess29 41.28  
#> BiologicalMaterial01 41.19  
#> ManufacturingProcess31 40.82  
#> BiologicalMaterial09 39.70  
#> ManufacturingProcess11 38.11  
#> ManufacturingProcess12 35.52

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

library(gridExtra)  
  
top9nnetPred <- trainTransformed %>%  
 select(Yield,   
 ManufacturingProcess13,  
 BiologicalMaterial06,  
 ManufacturingProcess17,  
 BiologicalMaterial03,  
 ManufacturingProcess36,  
 BiologicalMaterial02,  
 ManufacturingProcess09,  
 BiologicalMaterial12,  
 ManufacturingProcess06)  
  
plt01 <- top9nnetPred %>%  
 ggplot(aes(x = ManufacturingProcess13, y = Yield)) +  
 geom\_point() + geom\_smooth(method = "lm") + theme\_bw()  
  
plt02 <- top9nnetPred %>%  
 ggplot(aes(x = BiologicalMaterial06, y = Yield)) +  
 geom\_point()+ geom\_smooth(method = "lm") + theme\_bw()  
  
plt03 <- top9nnetPred %>%  
 ggplot(aes(x = ManufacturingProcess17, y = Yield)) +  
 geom\_point() + geom\_smooth(method = "lm") + theme\_bw()  
  
plt04 <- top9nnetPred %>%  
 ggplot(aes(x = BiologicalMaterial03, y = Yield)) +  
 geom\_point() + geom\_smooth(method = "lm") + theme\_bw()  
  
plt05 <- top9nnetPred %>%  
 ggplot(aes(x = ManufacturingProcess36, y = Yield)) +  
 geom\_point() + theme\_bw()  
  
plt06 <- top9nnetPred %>%  
 ggplot(aes(x = BiologicalMaterial02, y = Yield)) +  
 geom\_point() + geom\_smooth(method = "lm") + theme\_bw()  
  
plt07 <- top9nnetPred %>%  
 ggplot(aes(x = ManufacturingProcess09, y = Yield)) +  
 geom\_point() + geom\_smooth(method = "lm") + theme\_bw()  
  
plt08 <- top9nnetPred %>%  
 ggplot(aes(x = BiologicalMaterial12, y = Yield)) +  
 geom\_point() + geom\_smooth(method = "lm") + theme\_bw()  
  
plt09 <- top9nnetPred %>%  
 ggplot(aes(x = ManufacturingProcess06, y = Yield)) +  
 geom\_point() + geom\_smooth(method = "lm") + theme\_bw()  
  
# Biological Predictors  
grid.arrange(plt02, plt04, plt06, plt08, nrow = 2, top = "BIOLOGICAL PREDICTORS")



# Process Predictors  
grid.arrange(plt01, plt03, plt05, plt07, plt09,  
 nrow = 2, top = "PROCESS PREDICTORS")



We use scatter plots between the top predictors and the response for the TOP 9 predictors that are unique to the optimal nonlinear regression model.

Some conclusions:

* All four Biological predictors (BiologicalMaterial02, BiologicalMaterial36, BiologicalMaterial06, BiologicalMaterial12) appear to have low positive correlation to the response variable.
* The ManufacturingProcess13 and ManufacturingProcess17 appear to have low negative correlation to the response within a limited interval in each case. Also, there are a few outliers outside of the intervals.
* Predictor ManufacturingProcess17 appears to have no correlation to the response. The points seem to form parallel vertical lines every 2.5 units. I am not sure why this predictor would come up in the top 10 predictors for the optimal non-linear regression model.
* Process predictors ManufacturingProcess09 and ManufacturingProcess06 seem to have a low positive correlation within a limited interval in each case.

## KJ 8.1 (Regression Trees and Rule-Based Models)

**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"  
head(simulated)

#> V1 V2 V3 V4 V5 V6 V7  
#> 1 0.5337724 0.6478064 0.85078526 0.18159957 0.92903976 0.36179060 0.8266609  
#> 2 0.5837650 0.4381528 0.67272659 0.66924914 0.16379784 0.45305931 0.6489601  
#> 3 0.5895783 0.5879065 0.40967108 0.33812728 0.89409334 0.02681911 0.1785614  
#> 4 0.6910399 0.2259548 0.03335447 0.06691274 0.63744519 0.52500637 0.5133614  
#> 5 0.6673315 0.8188985 0.71676079 0.80324287 0.08306864 0.22344157 0.6644906  
#> 6 0.8392937 0.3862983 0.64618857 0.86105431 0.63038947 0.43703891 0.3360117  
#> V8 V9 V10 y  
#> 1 0.4214081 0.59111440 0.5886216 18.46398  
#> 2 0.8446239 0.92819306 0.7584008 16.09836  
#> 3 0.3495908 0.01759542 0.4441185 17.76165  
#> 4 0.7970260 0.68986918 0.4450716 13.78730  
#> 5 0.9038919 0.39696995 0.5500808 18.42984  
#> 6 0.6489177 0.53116033 0.9066182 20.85817

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

set.seed(600)  
  
  
model1 <- randomForest(y ~., data = simulated, importance = TRUE, ntree = 1000) # model  
  
rfImp1 <- varImp(model1, scale = FALSE) # Random Forest Importance Scores  
  
rfImp1 <- tibble::rownames\_to\_column(rfImp1, "Predictors")  
rfImp1

#> Predictors Overall  
#> 1 V1 8.816738364  
#> 2 V2 6.695930032  
#> 3 V3 0.816445594  
#> 4 V4 7.772654985  
#> 5 V5 2.212367303  
#> 6 V6 0.198140259  
#> 7 V7 0.077298384  
#> 8 V8 -0.087830526  
#> 9 V9 -0.075289818  
#> 10 V10 -0.008302096

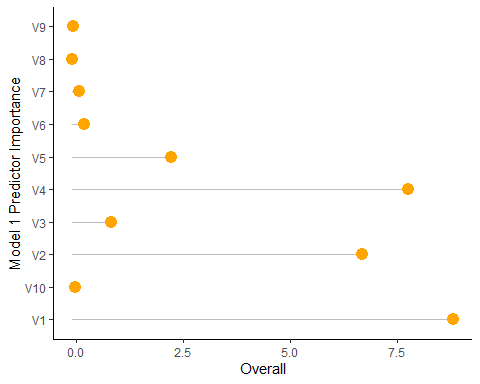
rfImp1[order(-rfImp1$Overall),]

#> Predictors Overall  
#> 1 V1 8.816738364  
#> 4 V4 7.772654985  
#> 2 V2 6.695930032  
#> 5 V5 2.212367303  
#> 3 V3 0.816445594  
#> 6 V6 0.198140259  
#> 7 V7 0.077298384  
#> 10 V10 -0.008302096  
#> 9 V9 -0.075289818  
#> 8 V8 -0.087830526

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

Looking at the table above, the importance scores for the predictors indicate the model relies heavily upon V1, V4, V2, and V5 respectively, unlike predictors V3 and V6-V10 not having much importance to the model.

# Visualization for original models' scores  
rfImp1 %>%  
 mutate(name = fct\_reorder(Predictors, Overall)) %>%  
 ggplot(aes(x = name, y = Overall)) +  
 geom\_segment(aes(x=Predictors, xend=Predictors, y=min(Overall), yend= Overall), color = "grey") +  
 geom\_point(color = "orange", size = 4) +  
 xlab("Model 1 Predictor Importance") +  
 theme\_light() + theme\_classic() +  
 coord\_flip()



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

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

#> [1] 0.9413266

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

Looking at the dataframe below, the importance of V1 decreased when a new V1-correlated predictor was introduced. V4, V2, V1, the New Variable, and V5 respectively are all the most important predictors from this random forest model. Compared to the last model, these top predictors bear a similar importance and order.

model2 <- randomForest(y~., data = simulated, importance = T, ntree = 1000)  
rfImp2 <- varImp(model2, scale = F)  
  
Prednames <- c("V1", "V2", "V3", "V4", "V5", "V6", "V7", "V8", "V9", "V10", "NEW")   
rfImp2$Variables <- factor(rownames(rfImp2), levels = Prednames)  
  
rfImp2[order(-rfImp2$Overall),]

#> Overall Variables  
#> V4 7.206875763 V4  
#> V1 6.056803728 V1  
#> V2 5.809811066 V2  
#> duplicate1 4.069245675 <NA>  
#> V5 1.890574263 V5  
#> V3 0.645048458 V3  
#> V6 0.185626446 V6  
#> V10 0.088247661 V10  
#> V7 0.006409373 V7  
#> V9 -0.046998232 V9  
#> V8 -0.152995300 V8

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

Looking at these importances, they resemble a close pattern to the traditional random forest model with V1, V2, V4, and V5 all having the most importance out of all predictors in varying orders.

[cforest](https://www.rdocumentation.org/packages/partykit/versions/1.2-16/topics/cforest)

[Conditional Random Forests](https://stephsus.github.io/shih_randomforests.pdf)

set.seed(555)  
cforest1 <- cforest(y~., data = simulated[,1:11], controls = cforest\_control(ntree = 1000))

set.seed(444)  
cforest2 <- cforest(y~., data = simulated[,1:11], controls = cforest\_control(ntree = 1000))

# Variable Importances   
  
cfImp1 <- varimp(cforest1)  
cfImp2 <- varimp(cforest2)  
  
cfImp1\_cond <- varimp(cforest1, conditional = T)  
cfImp2\_cond <- varimp(cforest2, conditional = T)

# Comparing performances with conditional and non conditional random forests  
cfImp1 <- data.frame(Original = cfImp1, Variable = factor(names(cfImp1), levels = Prednames))  
  
cfImp2 <- data.frame(Extra = cfImp2, Variable = factor(names(cfImp2), levels = Prednames))  
  
cfImp1\_cond <- data.frame(CondInf = cfImp1\_cond, Variable = factor(names(cfImp1\_cond), levels = Prednames))  
  
cfImp2\_cond <- data.frame(Extra\_CondInf = cfImp2\_cond, Variable = factor(names(cfImp2\_cond), levels = Prednames))  
  
cfImp1[order(-cfImp1$Original),]

#> Original Variable  
#> V1 9.179412933 V1  
#> V4 8.909063426 V4  
#> V2 6.885827859 V2  
#> V5 2.174591095 V5  
#> V3 0.105521216 V3  
#> V7 0.050620801 V7  
#> V6 -0.001136905 V6  
#> V10 -0.008349765 V10  
#> V9 -0.010844132 V9  
#> V8 -0.054459880 V8

cfImp2[order(-cfImp2$Extra),]

#> Extra Variable  
#> V4 9.00929963 V4  
#> V1 8.85872013 V1  
#> V2 6.97187508 V2  
#> V5 2.21761650 V5  
#> V3 0.04624347 V3  
#> V7 0.03853001 V7  
#> V9 0.01169703 V9  
#> V6 -0.01522918 V6  
#> V8 -0.04657289 V8  
#> V10 -0.04852399 V10

cfImp1\_cond[order(-cfImp1\_cond$CondInf),]

#> CondInf Variable  
#> V4 4.924105135 V4  
#> V2 3.910485508 V2  
#> V1 2.917831143 V1  
#> V5 0.734201549 V5  
#> V3 0.026566503 V3  
#> V6 0.008547101 V6  
#> V7 0.001942432 V7  
#> V9 0.001155499 V9  
#> V10 -0.003957814 V10  
#> V8 -0.007587794 V8

cfImp2\_cond[order(-cfImp2\_cond$Extra\_CondInf),]

#> Extra\_CondInf Variable  
#> V4 4.9645826009 V4  
#> V2 4.0035203089 V2  
#> V1 2.9374207973 V1  
#> V5 0.7356601225 V5  
#> V3 0.0276681812 V3  
#> V7 0.0191547849 V7  
#> V6 0.0046130693 V6  
#> V9 0.0005618543 V9  
#> V8 0.0002851250 V8  
#> V10 -0.0086373991 V10

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

Per the wording of the problem, it does not seem boosted trees is a required method of random forest for this question. therefore, Bagging and Cubist methods were used.

Using bagging and cubist methods, the same pattern does generally occur. In the case of bagging, V5 has a higher rank in the importance of its weight, and V3 has less importance as a predictor. In the cubist method, this resembles closer to the original pattern.

# Boosted Trees: original  
# Specifying t-distribution  
# Having difficulty, might omit  
  
# gbmModel <- gbm(y~., data = simulated, distribution = "tdist")  
#   
# booImp1 <- varImp(gbmModel)

# Trying bagging method  
set.seed(777)  
bagfit1 <- ipred::bagging(y~., data = simulated[,1:11], nbag = 60)  
  
bag1Imp <- varImp(bagfit1)  
  
bag1Imp$Variable <- factor(rownames(bag1Imp), levels = Prednames)  
  
bag1Imp[order(-bag1Imp$Overall),]

#> Overall Variable  
#> V4 2.6108735 V4  
#> V5 2.3693100 V5  
#> V2 2.2277246 V2  
#> V1 1.7770710 V1  
#> V3 1.3216539 V3  
#> V6 0.9808527 V6  
#> V7 0.9675982 V7  
#> V10 0.7483495 V10  
#> V9 0.6976806 V9  
#> V8 0.6246593 V8

set.seed(888)  
bagfit2 <- ipred::bagging(y~., data = simulated[,1:11], nag = 60)  
  
bag2Imp <- varImp(bagfit2)  
  
  
bag2Imp$Variable <- factor(rownames(bag2Imp), levels = Prednames)  
  
bag2Imp[order(-bag2Imp$Overall),]

#> Overall Variable  
#> V4 2.6886180 V4  
#> V5 2.3370637 V5  
#> V2 2.2896541 V2  
#> V1 1.9488476 V1  
#> V3 1.4307745 V3  
#> V10 1.0173759 V10  
#> V7 0.9770750 V7  
#> V6 0.9706450 V6  
#> V9 0.6763753 V9  
#> V8 0.6469578 V8

# Cubist: original  
set.seed(222)  
  
cbfit1 <- cubist(x = simulated[, 1:10], y = simulated$y, committees = 100)  
  
cbImp1 <- varImp(cbfit1)  
names(cbImp1) <- "Original"  
cbImp1$Variable <- factor(rownames(cbImp1), levels = Prednames)  
  
cbImp1[order(-cbImp1$Variable),]

#> Original Variable  
#> V1 71.5 V1  
#> V3 47.0 V3  
#> V2 58.5 V2  
#> V4 48.0 V4  
#> V5 33.0 V5  
#> V6 13.0 V6  
#> V7 0.0 V7  
#> V8 0.0 V8  
#> V9 0.0 V9  
#> V10 0.0 V10

# Cubist: extra  
set.seed(111)  
  
cbfit2 <- cubist(x = simulated[ , names(simulated) !="y"], y = simulated$y, committees = 100)  
  
cbImp2 <- varImp(cbfit2)  
names(cbImp1) <- "Extra"  
cbImp2$Variable <- factor(rownames(cbImp2), levels = Prednames)  
  
cbImp2[order(-cbImp2$Variable),]

#> Overall Variable  
#> V3 46.5 V3  
#> V1 66.0 V1  
#> V2 60.5 V2  
#> V4 45.5 V4  
#> V5 28.5 V5  
#> V6 16.0 V6  
#> duplicate1 4.5 <NA>  
#> V8 1.5 V8  
#> V7 0.0 V7  
#> V9 0.0 V9  
#> V10 0.0 V10

## KJ 8.2 (Regression Trees and Rule-Based Models)

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

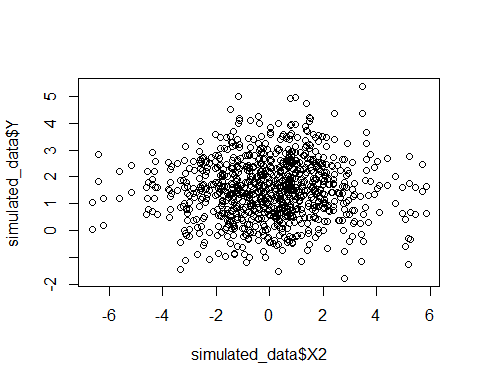
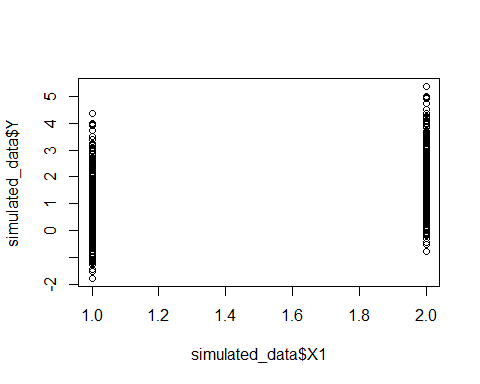
Looking at the plots below, we can see the relationship between the predictor and response. The predictor, X1, is a binomial random variable (1,2) of 450 observations which are used to create homogeneous responses according to the predictor. X2 on the other hand, which is comprised of 450 observations as well, is much more random, unrelated to the response, and therefore considered “granular”.

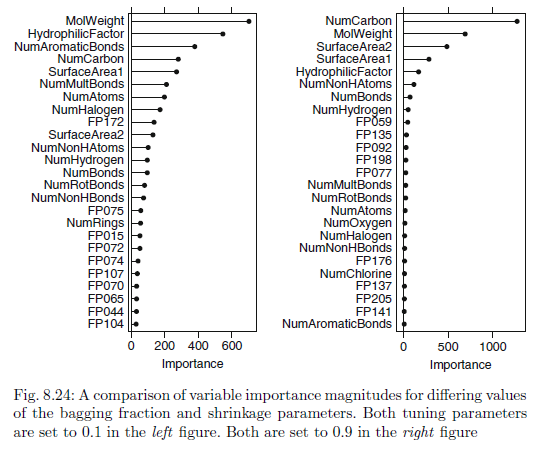
set.seed(222)  
  
x1 <- rep(1:2, each = 450)  
  
y <- x1 + rnorm(450, mean = 0, sd = 1)

set.seed(111)  
  
x2 <- rnorm(450, mean = 0, sd = 2)

simulated\_data <- data.frame(Y = y, X1 = x1, X2 = x2)

plot(simulated\_data$Y ~ simulated\_data$X1 + simulated\_data$X2)





## KJ 8.3 (Regression Trees and Rule-Based Models)

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

### a. Why does the model on the right focus its importance on just the first few of predictors, whereas the model on the left spreads importance across more predictors?

The learning rate referenced in the questions prompt is also called “regularization” or “shrinkage” and is a means to limit the “greedy” nature of boosting. From page 206 of our KJ text, “Friedman recognized that his gradient boosting machine could be susceptible to over-fitting, since the learner employed…is tasked with optimally fitting the gradient.

The lower the learning rate the less shrinkage there is and the lower the change in predicted model from previous boosting iterations. As the learning parameter is increased the model will “learn” quicker and perhaps be more prone to overfit and overvalue fewer variables of importance.

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

Because gradiant boosting may overfit more readily with a higher learning parameter, the model on the *left* would likely be more predictive of other samples.

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

As more variables are incorporated as the interaction depth increases, the slope of the importance of variables will flatten out.

## KJ 8.7 (Regression Trees and Rule-Based Models)

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

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

#### Single Tree

library(rpart)  
  
rpartTree <- rpart(Yield ~., data = trainData)

#### Random Forest

rfModel <- randomForest(Yield~., data = trainData,  
 importance = TRUE,  
 ntree = 1000)

#### Boosted Trees

gbmModel <- train(Yield ~., data = trainData, method = "gbm",  
 verbose = FALSE)

#### Cubist

cubistTuned <- train(Yield~., data = trainData, method = "cubist")

postResample(predict(rpartTree, testData), testData$Yield)

#> RMSE Rsquared MAE   
#> 1.2928420 0.4094331 1.0413732

postResample(predict(rfModel, testData), testData$Yield)

#> RMSE Rsquared MAE   
#> 0.9844450 0.6720898 0.8298936

postResample(predict(gbmModel, testData), testData$Yield)

#> RMSE Rsquared MAE   
#> 0.8805535 0.7188205 0.7175104

postResample(predict(cubistTuned, testData), testData$Yield)

#> RMSE Rsquared MAE   
#> 0.8154792 0.7645397 0.6963243

| Model | RMSE | Rsquared | MAE |
| --- | --- | --- | --- |
| Single Tree | 0.70 | 0.41 | 0.56 |
| Random Forest | 0.53 | 0.68 | 0.45 |
| Boosted Trees | 0.52 | 0.67 | 0.43 |
| Cubist | 0.50 | 0.72 | 0.37 |

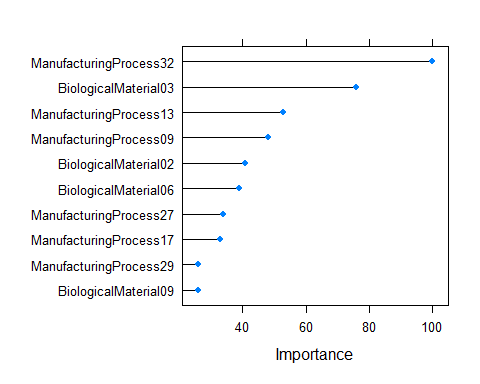
Based on the results in the above table, the cubist model performs best with the test data set.

### 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?

Compared to the linear and nonlinear counterparts, the most important factors are a balance between biological and manufacturing variables. All three chose ManufacturingProcess32 as the most important variable.

#### Tree-based Models

plot(caret::varImp(cubistTuned), top = 10)

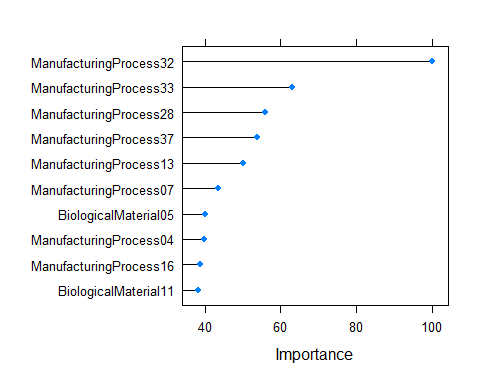


caret::varImp(cubistTuned)

#> cubist variable importance  
#>   
#> only 20 most important variables shown (out of 57)  
#>   
#> Overall  
#> ManufacturingProcess32 100  
#> BiologicalMaterial03 76  
#> ManufacturingProcess13 53  
#> ManufacturingProcess09 48  
#> BiologicalMaterial02 41  
#> BiologicalMaterial06 39  
#> ManufacturingProcess27 34  
#> ManufacturingProcess17 33  
#> ManufacturingProcess33 26  
#> ManufacturingProcess29 26  
#> BiologicalMaterial09 26  
#> ManufacturingProcess28 24  
#> ManufacturingProcess25 20  
#> BiologicalMaterial08 17  
#> BiologicalMaterial11 14  
#> BiologicalMaterial12 14  
#> ManufacturingProcess04 13  
#> ManufacturingProcess14 13  
#> BiologicalMaterial10 11  
#> BiologicalMaterial01 11

#### Linear Models

plot(caret::varImp(lm\_model), top = 10)

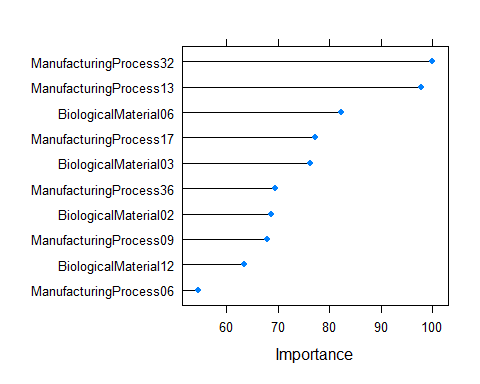


caret::varImp(lm\_model, scale = FALSE)

#> lm variable importance  
#>   
#> only 20 most important variables shown (out of 56)  
#>   
#> Overall  
#> ManufacturingProcess32 4.0149  
#> ManufacturingProcess33 2.5534  
#> ManufacturingProcess28 2.2726  
#> ManufacturingProcess37 2.1892  
#> ManufacturingProcess13 2.0342  
#> ManufacturingProcess07 1.7731  
#> BiologicalMaterial05 1.6381  
#> ManufacturingProcess04 1.6339  
#> ManufacturingProcess16 1.5880  
#> BiologicalMaterial11 1.5688  
#> ManufacturingProcess40 1.4435  
#> ManufacturingProcess10 1.4096  
#> ManufacturingProcess09 1.3452  
#> ManufacturingProcess17 1.1744  
#> BiologicalMaterial08 1.1522  
#> ManufacturingProcess41 1.0988  
#> BiologicalMaterial12 1.0735  
#> ManufacturingProcess45 0.9781  
#> BiologicalMaterial02 0.9761  
#> BiologicalMaterial07 0.9115

#### Nonlinear Models

plot(caret::varImp(nnetChemTune), top = 10)



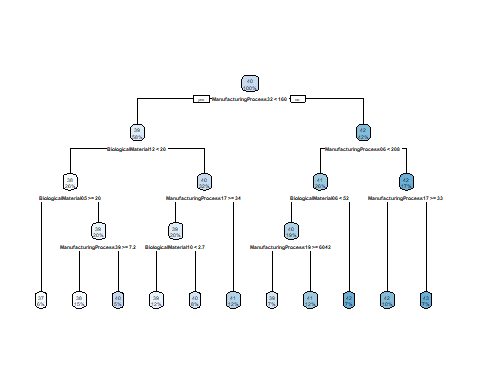
caret::varImp(nnetChemTune)

#> loess r-squared variable importance  
#>   
#> only 20 most important variables shown (out of 57)  
#>   
#> Overall  
#> ManufacturingProcess32 100.00  
#> ManufacturingProcess13 97.84  
#> BiologicalMaterial06 82.22  
#> ManufacturingProcess17 77.27  
#> BiologicalMaterial03 76.21  
#> ManufacturingProcess36 69.53  
#> BiologicalMaterial02 68.79  
#> ManufacturingProcess09 67.86  
#> BiologicalMaterial12 63.36  
#> ManufacturingProcess06 54.51  
#> BiologicalMaterial04 54.31  
#> ManufacturingProcess33 49.26  
#> BiologicalMaterial11 42.44  
#> BiologicalMaterial08 41.89  
#> ManufacturingProcess29 41.28  
#> BiologicalMaterial01 41.19  
#> ManufacturingProcess31 40.82  
#> BiologicalMaterial09 39.70  
#> ManufacturingProcess11 38.11  
#> ManufacturingProcess12 35.52

### 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?

This view does provide some additional information related to the biological and process predictors. Manufacturing predictors are more prevalent with other manufacturing indicators and the same appears to be true for biological ones. The tree has a balance between bio and manufacturing variables. It also parses one of the most important features of the three approaches and how other factors weigh into ManufacturingProcess32.

library(rpart.plot)  
rpart.plot(rpartTree)



# Week 6

## HW BATCH #2 - Market Basket Analysis / Recommender Systems (a simple problem)

**Imagine 10000 receipts sitting on your table. Each receipt represents a transcation with items that were purchased. The receipt is a representation of stuff that went into a customer’s basket - and therefore ‘Market Basket Analysis’.**

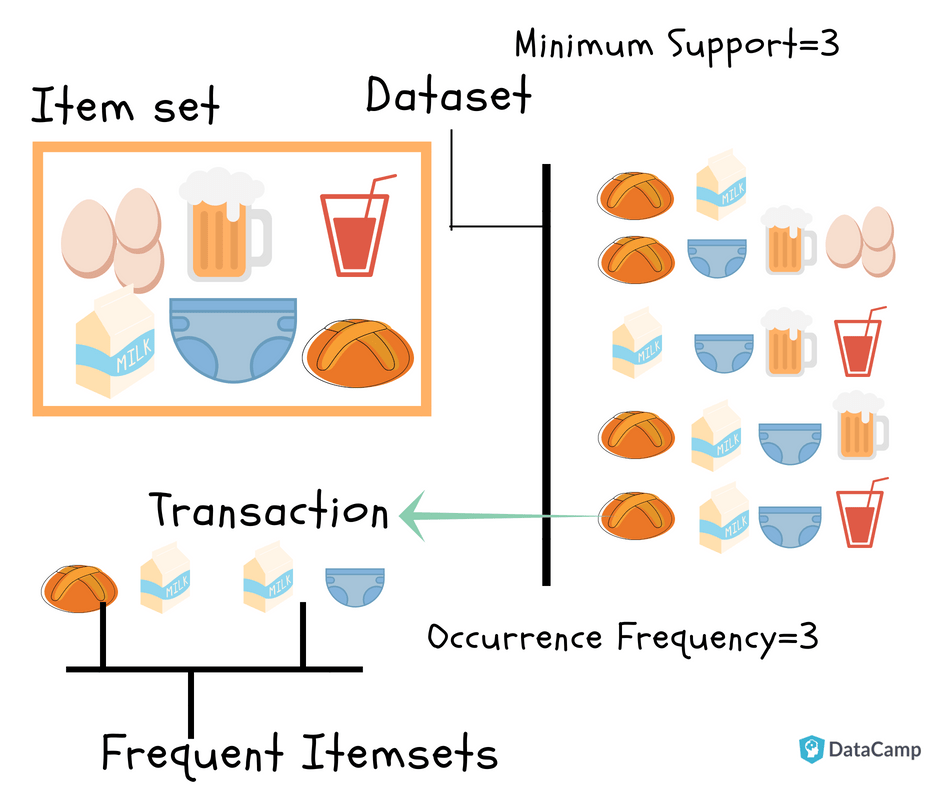
**That is exactly what the Groceries Data Set contains: a collection of receipts with each line representing 1 receipt and the items purchased. Each line is called a**

**and each column in a row represents an**

**Here is the dataset =**[GroceryDataSet.csv](https://bbhosted.cuny.edu/bbcswebdav/pid-8461316-dt-announcement-rid-389564218_1/xid-389564218_1)**(comma separated file)**

**You 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. Turn in as you would the other problems from HA and KJ. You should packaged this with your HW#2 set.**

**NOTE: Bonus - Use a Python Library in addition to R and get a Bonus.**



### 1. Introduction

The database contains customer transactions, 1000 receipts, that will be analyzed to predict customer’s shopping trends and potentially provide bundle product recommendations with a more attractive price offer than buying the products individually.

For example, information can be extracted on consumer behavior like \*If someone buys coffee and creamer, then is likely to buy cookies with high probability”.

### 2. Required Libraries (Python and RMarkdown)

Load some libraries and import data.

#Load required R language libraries  
require(tidyverse) # data manipulation  
require(arules) # mining association rules and frequent itemsets  
require(arulesViz) # visualization techniques for association rules  
require(RColorBrewer) # graphics color  
library(reticulate) # R interface to Python

#Installing Python Packages   
py\_install("pandas")  
py\_install("mlxtend")  
py\_install("seaborn")

### 3. Load Dataset (Python)

Loading the data set with python function, pandas:read\_csv method. The data represents transaction data from a point of sales system on customer data.

import pandas as pd  
groc = pd.read\_csv("GroceryDataSet.csv", header=None)  
# Load data from a CSV file and hide the header  
print("\nReading the CSV file (without header)...\n", groc.head(10))

#>   
#> Reading the CSV file (without header)...  
#> 0 1 2 ... 29 30 31  
#> 0 citrus fruit semi-finished bread margarine ... NaN NaN NaN  
#> 1 tropical fruit yogurt coffee ... NaN NaN NaN  
#> 2 whole milk NaN NaN ... NaN NaN NaN  
#> 3 pip fruit yogurt cream cheese ... NaN NaN NaN  
#> 4 other vegetables whole milk condensed milk ... NaN NaN NaN  
#> 5 whole milk butter yogurt ... NaN NaN NaN  
#> 6 rolls/buns NaN NaN ... NaN NaN NaN  
#> 7 other vegetables UHT-milk rolls/buns ... NaN NaN NaN  
#> 8 pot plants NaN NaN ... NaN NaN NaN  
#> 9 whole milk cereals NaN ... NaN NaN NaN  
#>   
#> [10 rows x 32 columns]

With the python function, pandas:shape method, view a tuple representing the dimensions of a object:

groc.shape

#> (9835, 32)

… has 9835 observations (rows) recorded for 32 (columns) variables.

View the data frame summary information in R (add py$ to the variable).

#Explore the data structure with an R function  
skimr::skim\_without\_charts(py$groc)

Data summary

|  |  |
| --- | --- |
| Name | py$groc |
| Number of rows | 9835 |
| Number of columns | 32 |
| \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |  |
| Column type frequency: |  |
| character | 1 |
| list | 31 |
| \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |  |
| Group variables | None |

**Variable type: character**

| skim\_variable | n\_missing | complete\_rate | min | max | empty | n\_unique | whitespace |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 1 | 3 | 25 | 0 | 158 | 0 |

**Variable type: list**

| skim\_variable | n\_missing | complete\_rate | n\_unique | min\_length | max\_length |
| --- | --- | --- | --- | --- | --- |
| 1 | 2159 | 0.78 | 151 | 1 | 1 |
| 2 | 3802 | 0.61 | 155 | 1 | 1 |
| 3 | 5101 | 0.48 | 153 | 1 | 1 |
| 4 | 6106 | 0.38 | 150 | 1 | 1 |
| 5 | 6961 | 0.29 | 137 | 1 | 1 |
| 6 | 7606 | 0.23 | 138 | 1 | 1 |
| 7 | 8151 | 0.17 | 140 | 1 | 1 |
| 8 | 8589 | 0.13 | 128 | 1 | 1 |
| 9 | 8939 | 0.09 | 120 | 1 | 1 |
| 10 | 9185 | 0.07 | 116 | 1 | 1 |
| 11 | 9367 | 0.05 | 103 | 1 | 1 |
| 12 | 9484 | 0.04 | 88 | 1 | 1 |
| 13 | 9562 | 0.03 | 84 | 1 | 1 |
| 14 | 9639 | 0.02 | 74 | 1 | 1 |
| 15 | 9694 | 0.01 | 63 | 1 | 1 |
| 16 | 9740 | 0.01 | 49 | 1 | 1 |
| 17 | 9769 | 0.01 | 44 | 1 | 1 |
| 18 | 9783 | 0.01 | 33 | 1 | 1 |
| 19 | 9797 | 0.00 | 29 | 1 | 1 |
| 20 | 9806 | 0.00 | 23 | 1 | 1 |
| 21 | 9817 | 0.00 | 17 | 1 | 1 |
| 22 | 9821 | 0.00 | 13 | 1 | 1 |
| 23 | 9827 | 0.00 | 7 | 1 | 1 |
| 24 | 9828 | 0.00 | 6 | 1 | 1 |
| 25 | 9828 | 0.00 | 7 | 1 | 1 |
| 26 | 9829 | 0.00 | 5 | 1 | 1 |
| 27 | 9830 | 0.00 | 4 | 1 | 1 |
| 28 | 9831 | 0.00 | 3 | 1 | 1 |
| 29 | 9834 | 0.00 | 1 | 1 | 1 |
| 30 | 9834 | 0.00 | 1 | 1 | 1 |
| 31 | 9834 | 0.00 | 1 | 1 | 1 |

### 4. Convert to a Transactional Dataset - Market Basket Analysis using R

Convert data frame to a class transactions for perform association mining as follows:

#Read the data as transactions class  
grocery <- read.transactions("GroceryDataSet.csv", sep = ",", rm.duplicates = TRUE)

#### 4.1 Transaction object

grocery

#> transactions in sparse format with  
#> 9835 transactions (rows) and  
#> 169 items (columns)

#### 4.2 Summary

summary(grocery)

#> transactions as itemMatrix in sparse format with  
#> 9835 rows (elements/itemsets/transactions) and  
#> 169 columns (items) and a density of 0.02609146   
#>   
#> most frequent items:  
#> whole milk other vegetables rolls/buns soda   
#> 2513 1903 1809 1715   
#> yogurt (Other)   
#> 1372 34055   
#>   
#> element (itemset/transaction) length distribution:  
#> sizes  
#> 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16   
#> 2159 1643 1299 1005 855 645 545 438 350 246 182 117 78 77 55 46   
#> 17 18 19 20 21 22 23 24 26 27 28 29 32   
#> 29 14 14 9 11 4 6 1 1 1 1 3 1   
#>   
#> Min. 1st Qu. Median Mean 3rd Qu. Max.   
#> 1.000 2.000 3.000 4.409 6.000 32.000   
#>   
#> includes extended item information - examples:  
#> labels  
#> 1 abrasive cleaner  
#> 2 artif. sweetener  
#> 3 baby cosmetics

The summary(grocery) informs us about the transaction object output:

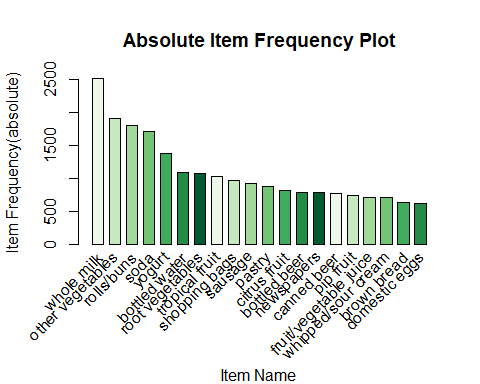
* There are **9835 transactions (rows) and 169 items (columns).**
* **Density** tells the percentage of non-zero cells in the sparse matrix.
* Summary tell the most frequent items.
* **Element (itemset/transaction) length distribution:** Indicates how many transactions are there for 1-itemset, for 2-itemset and so on.

### 5. Data Analysis with Visualizations

Plots showing transactions object most frequent, absolute and relative, items in the data set.

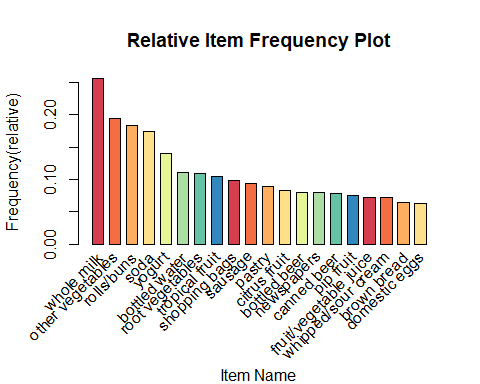
Independent transactions expressed as a whole number …

#Checking top 20 items sold in the dataset  
itemFrequencyPlot(grocery,topN=20,type="absolute",col=brewer.pal(7,'Greens'),space=(0.5),width=(0.5),xlab="Item Name",ylab="Item Frequency(absolute)",main="Absolute Item Frequency Plot")



The amount of times an item/transaction have appeared as compared to others, expressed as a percentage …

#Again checking top 20 items sold in the given dataset with type=relative  
itemFrequencyPlot(grocery,topN=20,type="relative",col=brewer.pal(8,'Spectral'),space=(0.5),width=(0.5),xlab="Item Name",ylab="Frequency(relative)",main="Relative Item Frequency Plot")



*Observation:* Whole milk is the best selling product, followed by other vegetables.

### 6. Apriori Algorithm (R)

**Association Rules**

The associated rules explain the relationship of:

$ { antecedent } { consequent }$

where

In the analysts, association rules is a data mining measures to compare the relationship between items using these metrics: (1) support, (2) confidence, and (3) lift.

*Support* - refers to the popularity of a single item, based on how frequently customers buy it compared to other products.

*Confidence* - refers to how often customers purchase two products in an item set (the percentage in which B is bought with A).

*Lift* - refers to the likelihood that a customer will purchase the first and second item together in a set.

#### 6.1 Training the algorithm

Train the selected algorithm (Apriori)

#Training apriori algorithm on dataset  
rules <- apriori(data=grocery,parameter=list(support=0.001,confidence=0.9, minlen= 2, maxlen=10, target = "rules"))

#> Apriori  
#>   
#> Parameter specification:  
#> confidence minval smax arem aval originalSupport maxtime support minlen  
#> 0.9 0.1 1 none FALSE TRUE 5 0.001 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: 9   
#>   
#> set item appearances ...[0 item(s)] done [0.00s].  
#> set transactions ...[169 item(s), 9835 transaction(s)] done [0.00s].  
#> sorting and recoding items ... [157 item(s)] done [0.00s].  
#> creating transaction tree ... done [0.00s].  
#> checking subsets of size 1 2 3 4 5 6 done [0.01s].  
#> writing ... [129 rule(s)] done [0.00s].  
#> creating S4 object ... done [0.00s].

The association rules metrics settings will find frequent itemsets that satisfies 0.1% Support, 90% Confidence, and Maximum of number of items at 10.

The total number of rules is 129, with a minimum purchase of 2 item and maximum is 10.

summary(rules) # This gives set of rules generated here it is 129, number of transactions, support and confidence values that we have given in previous code

#> set of 129 rules  
#>   
#> rule length distribution (lhs + rhs):sizes  
#> 3 4 5 6   
#> 10 57 56 6   
#>   
#> Min. 1st Qu. Median Mean 3rd Qu. Max.   
#> 3.00 4.00 4.00 4.45 5.00 6.00   
#>   
#> summary of quality measures:  
#> support confidence coverage lift   
#> Min. :0.001017 Min. :0.9000 Min. :0.001017 Min. : 3.522   
#> 1st Qu.:0.001017 1st Qu.:0.9091 1st Qu.:0.001118 1st Qu.: 3.588   
#> Median :0.001017 Median :0.9167 Median :0.001118 Median : 3.669   
#> Mean :0.001139 Mean :0.9337 Mean :0.001222 Mean : 4.216   
#> 3rd Qu.:0.001220 3rd Qu.:0.9333 3rd Qu.:0.001322 3rd Qu.: 4.698   
#> Max. :0.001932 Max. :1.0000 Max. :0.002135 Max. :11.235   
#> count   
#> Min. :10.0   
#> 1st Qu.:10.0   
#> Median :10.0   
#> Mean :11.2   
#> 3rd Qu.:12.0   
#> Max. :19.0   
#>   
#> mining info:  
#> data ntransactions support confidence  
#> grocery 9835 0.001 0.9  
#> call  
#> apriori(data = grocery, parameter = list(support = 0.001, confidence = 0.9, minlen = 2, maxlen = 10, target = "rules"))

Summary (rules) shows the following:

* **Total number rules:** The set of 129 rules.
* **Rule length distribution (LHS + RHS):** A length of 4 items has the most rules: 57 and length of 6 items have the lowest number of rules:6.
* **Summary of quality measures:** min, max, median, mean and quantile values for support, confidence and lift.
* **Information used for creating rules:** The data, support, and confidence.

View the top 10 rules:

#show the top 10 rules  
options(digits = 2)  
inspect(rules[1:10])

#> lhs rhs support confidence coverage lift count  
#> [1] {liquor,   
#> red/blush wine} => {bottled beer} 0.0019 0.90 0.0021 11.2 19  
#> [2] {cereals,   
#> curd} => {whole milk} 0.0010 0.91 0.0011 3.6 10  
#> [3] {bottled beer,   
#> soups} => {whole milk} 0.0011 0.92 0.0012 3.6 11  
#> [4] {house keeping products,   
#> whipped/sour cream} => {whole milk} 0.0012 0.92 0.0013 3.6 12  
#> [5] {pastry,   
#> sweet spreads} => {whole milk} 0.0010 0.91 0.0011 3.6 10  
#> [6] {rice,   
#> sugar} => {whole milk} 0.0012 1.00 0.0012 3.9 12  
#> [7] {bottled water,   
#> rice} => {whole milk} 0.0012 0.92 0.0013 3.6 12  
#> [8] {canned fish,   
#> hygiene articles} => {whole milk} 0.0011 1.00 0.0011 3.9 11  
#> [9] {grapes,   
#> onions} => {other vegetables} 0.0011 0.92 0.0012 4.7 11  
#> [10] {hard cheese,   
#> oil} => {other vegetables} 0.0011 0.92 0.0012 4.7 11

*Convert rules into data frame*

rules2 <- rules  
rules3 <- as(rules2, "data.frame")  
write.csv(rules3, file = "D:/CUNY SPS/2022 Summer Course/DATA624/rules.csv", sep=",")

*Inspect the top 10 rules sorted by lift:*

The recommendation is based on historical preference and ratings to find similarities between users and items.

options(digits = 2)  
#visualize the results  
inspect(sort(rules, by="lift", decreasing=TRUE)[1:10])

#> lhs rhs support confidence coverage lift count  
#> [1] {liquor,   
#> red/blush wine} => {bottled beer} 0.0019 0.90 0.0021 11.2 19  
#> [2] {citrus fruit,   
#> fruit/vegetable juice,   
#> other vegetables,   
#> soda} => {root vegetables} 0.0010 0.91 0.0011 8.3 10  
#> [3] {oil,   
#> other vegetables,   
#> tropical fruit,   
#> whole milk,   
#> yogurt} => {root vegetables} 0.0010 0.91 0.0011 8.3 10  
#> [4] {butter,   
#> cream cheese,   
#> root vegetables} => {yogurt} 0.0010 0.91 0.0011 6.5 10  
#> [5] {butter,   
#> sliced cheese,   
#> tropical fruit,   
#> whole milk} => {yogurt} 0.0010 0.91 0.0011 6.5 10  
#> [6] {cream cheese,   
#> curd,   
#> other vegetables,   
#> whipped/sour cream} => {yogurt} 0.0010 0.91 0.0011 6.5 10  
#> [7] {butter,   
#> other vegetables,   
#> tropical fruit,   
#> white bread} => {yogurt} 0.0010 0.91 0.0011 6.5 10  
#> [8] {citrus fruit,   
#> root vegetables,   
#> soft cheese} => {other vegetables} 0.0010 1.00 0.0010 5.2 10  
#> [9] {brown bread,   
#> pip fruit,   
#> whipped/sour cream} => {other vegetables} 0.0011 1.00 0.0011 5.2 11  
#> [10] {grapes,   
#> tropical fruit,   
#> whole milk,   
#> yogurt} => {other vegetables} 0.0010 1.00 0.0010 5.2 10

#`high-confidence' rules  
# show the support, lift and confidence for the first 10 rules

*Lift Interpretation:* A rule with a lift count (see rules\_lift above) imply that, the items in LHS and RHS are n times more likely to be purchased together compared to the purchases when they are assumed to be unrelated. For example, **{liquor, red/blush wine} paired with {bottled beer}** has a high support and confidence threshold as a bundled transactional sale.

**Interpretation of Rules sorted by LIFT** \* 90% of the customers who purchased butter, cream cheese, root vegetables (lhs) also purchased yogurt (rhs) is 7 times more likely to purchase the itemset as a bundle sale. \* 100% of the customers who purchased brown bread, pip fruit, whipped/sour cream (lhs) also purchased other vegetables (rhs) is 5 times more likely to purchase the itemset as a bundle sale.

*Inspect the top 10 rules sorted by confidence:*

The lhs represents items already taken in a basket, rhs represents items frequently taken together along with purchased items.

options(digits = 2)  
#visualize the results  
inspect(sort(rules, by="confidence", decreasing=TRUE)[1:10])

#> lhs rhs support confidence coverage lift count  
#> [1] {rice,   
#> sugar} => {whole milk} 0.0012 1 0.0012 3.9 12  
#> [2] {canned fish,   
#> hygiene articles} => {whole milk} 0.0011 1 0.0011 3.9 11  
#> [3] {butter,   
#> rice,   
#> root vegetables} => {whole milk} 0.0010 1 0.0010 3.9 10  
#> [4] {flour,   
#> root vegetables,   
#> whipped/sour cream} => {whole milk} 0.0017 1 0.0017 3.9 17  
#> [5] {butter,   
#> domestic eggs,   
#> soft cheese} => {whole milk} 0.0010 1 0.0010 3.9 10  
#> [6] {citrus fruit,   
#> root vegetables,   
#> soft cheese} => {other vegetables} 0.0010 1 0.0010 5.2 10  
#> [7] {butter,   
#> hygiene articles,   
#> pip fruit} => {whole milk} 0.0010 1 0.0010 3.9 10  
#> [8] {hygiene articles,   
#> root vegetables,   
#> whipped/sour cream} => {whole milk} 0.0010 1 0.0010 3.9 10  
#> [9] {hygiene articles,   
#> pip fruit,   
#> root vegetables} => {whole milk} 0.0010 1 0.0010 3.9 10  
#> [10] {cream cheese,   
#> domestic eggs,   
#> sugar} => {whole milk} 0.0011 1 0.0011 3.9 11

#`high-confidence' rules  
# show the support, lift and confidence for the first 10 rules

*Confidence Interpretation:* The rules with confidence of 1 (see rules\_conf above) imply that, whenever the LHS item is purchased, the RHS item was purchased 100% of the time. Popular items that are derive from this analysis are: **{rice, sugar} with {whole milk}** and **{canned fish, hygiene articles} with {whole milk}**.

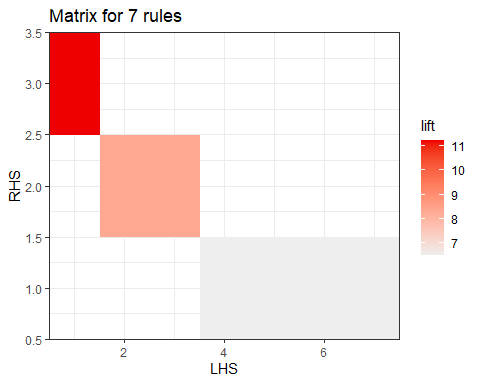
#### 6.2 Visualizing Association Rules

**Matrix**

Plot showing the measure of interestingness. The default reordering average measure (typically lift) pushes the rules with the highest lift value to the top-left corner of the plot.

sel <- subset(rules, lift>6)  
plot(sel, method="matrix", measure="lift")

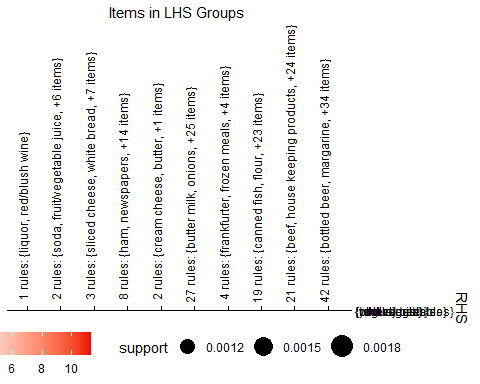
#> Itemsets in Antecedent (LHS)  
#> [1] "{liquor,red/blush wine}"   
#> [2] "{citrus fruit,fruit/vegetable juice,other vegetables,soda}"  
#> [3] "{oil,other vegetables,tropical fruit,whole milk,yogurt}"   
#> [4] "{butter,cream cheese,root vegetables}"   
#> [5] "{butter,sliced cheese,tropical fruit,whole milk}"   
#> [6] "{cream cheese,curd,other vegetables,whipped/sour cream}"   
#> [7] "{butter,other vegetables,tropical fruit,white bread}"   
#> Itemsets in Consequent (RHS)  
#> [1] "{yogurt}" "{root vegetables}" "{bottled beer}"



**A Grouped Matrix of Association Rules**

Plot showing group matrix-based visualization. Antecedents (columns) in the matrix are grouped using clustering. The most interesting item in the group (highest support in the group to support in all rules): liquor, red/blush wine.

plot(rules, method="grouped", control = list(k = 10))



if (!require("BiocManager", quietly = TRUE))  
 install.packages("BiocManager")  
  
BiocManager::install("Rgraphviz")

**A Graph model**

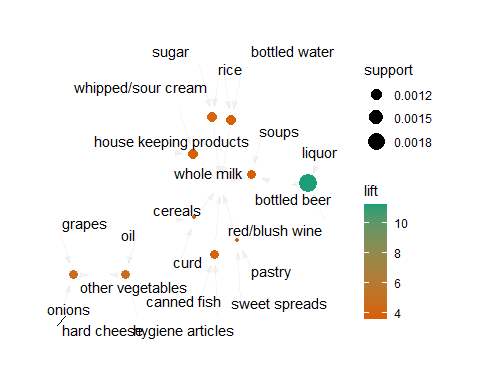
Visualize the 10 rules (or itemsets) as a graph with items as labeled vertices, and rules (or itemsets) represented as vertices connected to items using arrows.

# Filter rules with lift greater than 7  
subRules<-rules[quality(rules)$lift>7]  
  
top10subRules <- head(subRules, n = 10, by = "lift")

#plot(top10subRules, method = "graph", engine = "htmlwidget")

#export/save as a Graphviz dot-file  
#saveAsGraph(head(subRules, n = 10, by = "lift"), file = "rules.graphml")

plot(rules[1:10], method = 'graph', control = list(col=brewer.pal(6, 'Dark2')))



### 7. Apriori Algorithm (Python)

The Python apriori algorithm will be used as a comparison to R for Association Rule Mining.

#### 7.1 Required Libraries (Python)

**Install Required Modules:**

import numpy as np  
#import pandas as pd  
import seaborn as sns #table style  
import warnings # mask chunk code warnings

# importing data mining module  
from apyori import apriori

from mlxtend.preprocessing import TransactionEncoder

# importing required plots module  
import plotly.express as px #plotly tree map  
import matplotlib.pyplot as plt #treemap  
from IPython.display import display

#### 7.2 Load Data Set

Data was loaded in **Section 3. Load Dataset (Python)**

#### 7.3 Exploring the Data set

Store all purchased items in a NumPy array for easy identification of NULL values in dataset.

warnings.filterwarnings('ignore')  
  
# Gather All Items of Each Transactions into Numpy Array  
transactions = []  
for i in range(0, groc.shape[0]):  
 for j in range(0, groc.shape[1]):  
 transactions.append(groc.values[i,j])  
  
# converting to numpy array  
transactions = np.array(transactions)  
print(transactions, 10)

#> ['citrus fruit' 'semi-finished bread' 'margarine' ... 'nan' 'nan' 'nan'] 10

The data is in NumPy array format. We can now remove all the null values and print out the top 10 frequently occurring items.

# Transform Them a Pandas DataFrame  
df = pd.DataFrame(transactions, columns=["items"])   
  
# Put 1 to Each Item For Making Countable Table, to be able to perform Group By  
df["incident\_count"] = 1   
  
# Delete NaN Items from Dataset  
indexNames = df[df['items'] == "nan" ].index  
df.drop(indexNames , inplace=True)  
  
# Delete Nan Items in rows  
df\_new = df.dropna()  
  
# Making a New Appropriate Pandas DataFrame for Visualizations   
df\_table = df\_new.groupby("items").sum().sort\_values("incident\_count", ascending=False).reset\_index()  
  
# Initial Visualizations  
display(df\_table.head(10))

#> items incident\_count  
#> 0 whole milk 2513  
#> 1 other vegetables 1903  
#> 2 rolls/buns 1809  
#> 3 soda 1715  
#> 4 yogurt 1372  
#> 5 bottled water 1087  
#> 6 root vegetables 1072  
#> 7 tropical fruit 1032  
#> 8 shopping bags 969  
#> 9 sausage 924

These are the top 10 most purchased items by customers.

*Counting number of missing values*

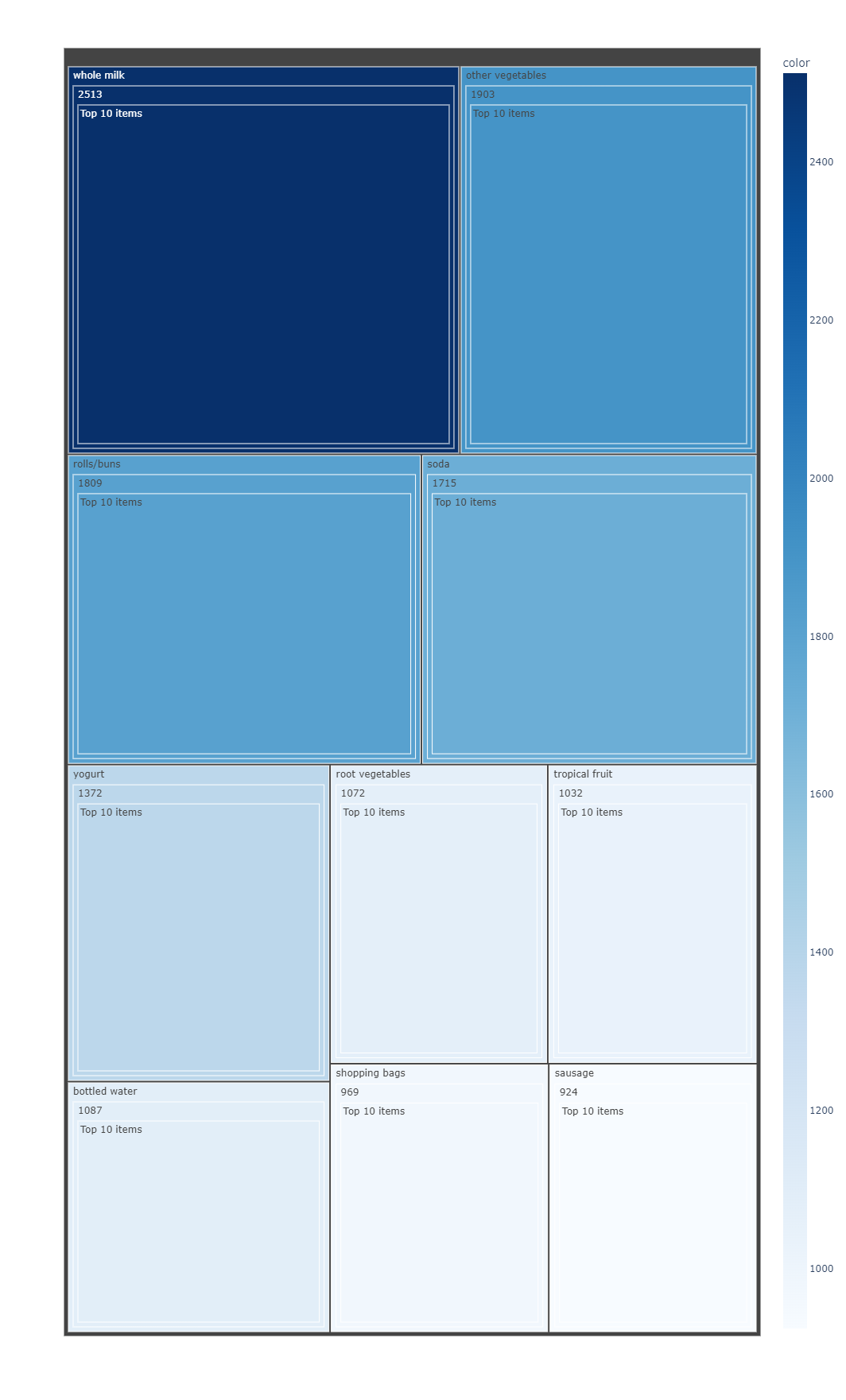
print("\nResulting in a Pandas Series containing the number of missing values in each column...\n", df\_new.isna().sum())

#>   
#> Resulting in a Pandas Series containing the number of missing values in each column...  
#> items 0  
#> incident\_count 0  
#> dtype: int64

#### 7.5 Visualization - Top Items

**Treemap to visualize the purchase**

# to have a same origin  
df\_table["10"] = "Top 10 items"   
  
# creating tree map using plotly  
fig = px.treemap(df\_table.head(10), path=["items", 'incident\_count', '10'], values='incident\_count',  
 color=df\_table["incident\_count"].head(10), hover\_data=['items'],  
 color\_continuous\_scale='Blues',  
 )



treemap

#### 7.6 Converting Data Set to a Transaction Object

**Pre-processing dataset**

We will perform the conversion of the transactions into equi-length transactions as shown in the following code:

# Data Cleaning step  
# replacing empty value with 0.  
pygroc = groc  
pygroc.fillna(0,inplace=True)  
  
# Data Pre-processing step  
# for using aprori , need to convert data in list format..  
  
transaction = []  
  
for i in range(0,len(pygroc)):  
 transaction.append([str(pygroc.values[i,j]) for j in range(0,20) if str(pygroc.values[i,j])!='0'])

The above code replaces nan values with zero’s, then initialized the list transaction and stored the transactions of length 20 in it. We had to remove the null values again inside transactions when items were fewer than 20.

## verifying - by printing the 0th transaction  
print("\nVerifying the 1st, Index 0, transaction\n", transaction[0])

#>   
#> Verifying the 1st, Index 0, transaction  
#> ['citrus fruit', 'semi-finished bread', 'margarine', 'ready soups']

## verifying - by printing the 1st transaction  
print("\nVerifying the 2nd, Index 1, transaction\n", transactions[1])

#>   
#> Verifying the 2nd, Index 1, transaction  
#> semi-finished bread

#### 7.7 Implement Apriori Algorithm (Python)

Performing the association rules from our data with the apriori class constructor.

Pyrules = apriori(transaction, min\_support=0.001, min\_confidence = 0.9, min\_length = 2, max\_length = 10)

The apriori algorithm’s associated rules are stored inside the rules generator object as shown:

Pyrules

#> <generator object apriori at 0x000002455E4E0CF0>

# all rules need to be converted in a list..  
Pyresults = list(Pyrules)  
#Pyresults  
  
# convert result in a dataframe for further operation...  
dfresults = pd.DataFrame(Pyresults)  
  
dfresults.head()

#> items ... ordered\_statistics  
#> 0 (red/blush wine, bottled beer, liquor) ... [((red/blush wine, liquor), (bottled beer), 0....  
#> 1 (whole milk, bottled beer, soups) ... [((bottled beer, soups), (whole milk), 0.91666...  
#> 2 (bottled water, whole milk, rice) ... [((bottled water, rice), (whole milk), 0.91666...  
#> 3 (whole milk, cereals, curd) ... [((cereals, curd), (whole milk), 0.90909090909...  
#> 4 (grapes, other vegetables, onions) ... [((grapes, onions), (other vegetables), 0.9166...  
#>   
#> [5 rows x 3 columns]

# keep support in a separate data frame so we can use later..   
support = dfresults.support

**Converting orderstatistic in a proper format.**

Order statistic has lhs => rhs as well rhs => lhs. Let’s choose first one which is ‘df\_results[’ordered\_statistics’][i][0]’

#all four empty list which will contain lhs, rhs, confidence and lift respectively.  
first\_values = []  
second\_values = []  
third\_values = []  
fourth\_value = []  
  
# loop number of rows time and append 1 by 1 value in a separate list..   
# first and second element was frozenset which need to be converted in list..  
for i in range(dfresults.shape[0]):  
 single\_list = dfresults['ordered\_statistics'][i][0]  
 first\_values.append(list(single\_list[0]))  
 second\_values.append(list(single\_list[1]))  
 third\_values.append(single\_list[2])  
 fourth\_value.append(single\_list[3])

# convert all four list into dataframe for further operation..  
lhs = pd.DataFrame(first\_values)  
rhs = pd.DataFrame(second\_values)  
  
confidence=pd.DataFrame(third\_values,columns=['Confidence'])  
  
lift=pd.DataFrame(fourth\_value,columns=['lift'])

# concat all list together in a single dataframe  
df\_final = pd.concat([lhs,rhs,support,confidence,lift], axis=1)  
df\_final

#> 0 1 ... Confidence lift  
#> 0 red/blush wine liquor ... 0.904762 11.277989  
#> 1 bottled beer soups ... 0.916667 3.587512  
#> 2 bottled water rice ... 0.916667 3.587512  
#> 3 cereals curd ... 0.909091 3.557863  
#> 4 grapes onions ... 0.916667 4.737476  
#> .. ... ... ... ... ...  
#> 92 yogurt tropical fruit ... 0.909091 3.557863  
#> 93 yogurt tropical fruit ... 0.937500 3.669046  
#> 94 whole milk citrus fruit ... 0.933333 4.823612  
#> 95 whole milk citrus fruit ... 0.909091 4.698323  
#> 96 yogurt tropical fruit ... 0.909091 12.147974  
#>   
#> [97 rows x 10 columns]

'''  
 we have some of place only 1 item in lhs and some place 3 or more so we need to a proper represenation for User to understand.   
 replacing none with ' ' and combining three column's in 1   
 example : coffee,none,none is converted to coffee, ,  
'''

#> "\n we have some of place only 1 item in lhs and some place 3 or more so we need to a proper represenation for User to understand. \n replacing none with ' ' and combining three column's in 1 \n example : coffee,none,none is converted to coffee, ,\n"

df\_final.fillna(value=' ', inplace=True)  
df\_final.head()

#> 0 1 2 3 ... 1 support Confidence lift  
#> 0 red/blush wine liquor ... 0.001932 0.904762 11.277989  
#> 1 bottled beer soups ... 0.001118 0.916667 3.587512  
#> 2 bottled water rice ... 0.001118 0.916667 3.587512  
#> 3 cereals curd ... 0.001017 0.909091 3.557863  
#> 4 grapes onions ... 0.001118 0.916667 4.737476  
#>   
#> [5 rows x 10 columns]

#set column name  
df\_final.columns = ['lhs',1,2,3,4,'rhs',5,'support','confidence','lift']  
df\_final.head()

#> lhs 1 2 3 ... 5 support confidence lift  
#> 0 red/blush wine liquor ... 0.001932 0.904762 11.277989  
#> 1 bottled beer soups ... 0.001118 0.916667 3.587512  
#> 2 bottled water rice ... 0.001118 0.916667 3.587512  
#> 3 cereals curd ... 0.001017 0.909091 3.557863  
#> 4 grapes onions ... 0.001118 0.916667 4.737476  
#>   
#> [5 rows x 10 columns]

# add all three column to lhs itemset only  
df\_final['lhs'] = df\_final['lhs'] + str(", ") + df\_final[1] + str(", ") + df\_final[2] + str(", ") + df\_final[3] + str(", ") + df\_final[4]  
  
df\_final['rhs'] = df\_final['rhs']+str(", ") + df\_final[5] + str(", ")

df\_final.head()

#> lhs 1 2 ... support confidence lift  
#> 0 red/blush wine, liquor, , , liquor ... 0.001932 0.904762 11.277989  
#> 1 bottled beer, soups, , , soups ... 0.001118 0.916667 3.587512  
#> 2 bottled water, rice, , , rice ... 0.001118 0.916667 3.587512  
#> 3 cereals, curd, , , curd ... 0.001017 0.909091 3.557863  
#> 4 grapes, onions, , , onions ... 0.001118 0.916667 4.737476  
#>   
#> [5 rows x 10 columns]

#drop columns 1,2,3,4, and 5 because now we already appended to lhs column.  
  
df\_final.drop(columns=[1,2,3,4,5],inplace=True)

## Showing top 10 items, based on lift. Sorting in desc order  
df\_final.sort\_values('lift', ascending=False).head(10)

#> lhs ... lift  
#> 96 yogurt, tropical fruit, oil, root vegetables, ... 12.147974  
#> 0 red/blush wine, liquor, , , ... 11.277989  
#> 64 other vegetables, tropical fruit, butter, whit... ... 6.516698  
#> 65 whole milk, tropical fruit, butter, sliced che... ... 6.516698  
#> 73 other vegetables, cream cheese , curd, whipped... ... 6.516698  
#> 15 cream cheese , root vegetables, butter, , ... 6.516698  
#> 85 pip fruit, tropical fruit, yogurt, ham, ... 5.168156  
#> 83 grapes, whole milk, tropical fruit, yogurt, ... 5.168156  
#> 71 tropical fruit, root vegetables, whipped/sour ... ... 5.168156  
#> 84 pip fruit, whole milk, tropical fruit, ham, ... 5.168156  
#>   
#> [10 rows x 5 columns]

np.set\_printoptions(suppress=True)  
print(df\_final.nlargest(n = 10, columns = 'lift'))

#> lhs ... lift  
#> 96 yogurt, tropical fruit, oil, root vegetables, ... 12.147974  
#> 0 red/blush wine, liquor, , , ... 11.277989  
#> 15 cream cheese , root vegetables, butter, , ... 6.516698  
#> 64 other vegetables, tropical fruit, butter, whit... ... 6.516698  
#> 65 whole milk, tropical fruit, butter, sliced che... ... 6.516698  
#> 73 other vegetables, cream cheese , curd, whipped... ... 6.516698  
#> 11 pip fruit, brown bread, whipped/sour cream, , ... 5.168156  
#> 27 soft cheese, root vegetables, citrus fruit, , ... 5.168156  
#> 71 tropical fruit, root vegetables, whipped/sour ... ... 5.168156  
#> 83 grapes, whole milk, tropical fruit, yogurt, ... 5.168156  
#>   
#> [10 rows x 5 columns]

np.set\_printoptions(suppress=True)  
print(df\_final.nlargest(n = 5, columns = 'confidence'))

#> lhs ... lift  
#> 8 sugar, rice, , , ... 3.913649  
#> 11 pip fruit, brown bread, whipped/sour cream, , ... 5.168156  
#> 27 soft cheese, root vegetables, citrus fruit, , ... 5.168156  
#> 28 cream cheese , domestic eggs, napkins, , ... 3.913649  
#> 29 sugar, cream cheese , domestic eggs, , ... 3.913649  
#>   
#> [5 rows x 5 columns]

pyapfile <- py$df\_final  
  
py\_apriori <- data.frame((pyapfile))  
  
print(py\_apriori)

#> lhs  
#> 1 red/blush wine, liquor, , ,   
#> 2 bottled beer, soups, , ,   
#> 3 bottled water, rice, , ,   
#> 4 cereals, curd, , ,   
#> 5 grapes, onions, , ,   
#> 6 oil, hard cheese, , ,   
#> 7 house keeping products, whipped/sour cream, , ,   
#> 8 sweet spreads, pastry, , ,   
#> 9 sugar, rice, , ,   
#> 10 sausage, berries, butter, ,   
#> 11 bottled beer, margarine, domestic eggs, ,   
#> 12 pip fruit, brown bread, whipped/sour cream, ,   
#> 13 curd, butter, citrus fruit, ,   
#> 14 coffee, whipped/sour cream, butter, ,   
#> 15 cream cheese , root vegetables, butter, ,   
#> 16 cream cheese , root vegetables, butter, ,   
#> 17 domestic eggs, curd, butter, ,   
#> 18 frozen meals, tropical fruit, butter, ,   
#> 19 pip fruit, butter, pastry, ,   
#> 20 soda, whipped/sour cream, butter, ,   
#> 21 pip fruit, whipped/sour cream, butter, ,   
#> 22 root vegetables, butter, white bread, ,   
#> 23 whipped/sour cream, butter, sliced cheese, ,   
#> 24 butter milk, whole milk, pork, ,   
#> 25 cream cheese , root vegetables, citrus fruit, ,   
#> 26 sugar, domestic eggs, citrus fruit, ,   
#> 27 tropical fruit, herbs, citrus fruit, ,   
#> 28 soft cheese, root vegetables, citrus fruit, ,   
#> 29 cream cheese , domestic eggs, napkins, ,   
#> 30 sugar, cream cheese , domestic eggs, ,   
#> 31 other vegetables, cream cheese , sugar, ,   
#> 32 pip fruit, cream cheese , sausage, ,   
#> 33 pip fruit, cream cheese , whipped/sour cream, ,   
#> 34 tropical fruit, whipped/sour cream, dessert, ,   
#> 35 fruit/vegetable juice, margarine, domestic eggs, ,   
#> 36 hygiene articles, tropical fruit, domestic eggs, ,   
#> 37 sugar, tropical fruit, domestic eggs, ,   
#> 38 sugar, domestic eggs, yogurt, ,   
#> 39 flour, root vegetables, whipped/sour cream, ,   
#> 40 frozen meals, tropical fruit, frankfurter, ,   
#> 41 frozen meals, tropical fruit, frankfurter, ,   
#> 42 root vegetables, frankfurter, sliced cheese, ,   
#> 43 whole milk, fruit/vegetable juice, herbs, ,   
#> 44 tropical fruit, whipped/sour cream, hamburger meat, ,   
#> 45 tropical fruit, hard cheese, whipped/sour cream, ,   
#> 46 pip fruit, soda, pork, ,   
#> 47 yogurt, rice, root vegetables, ,   
#> 48 soft cheese, tropical fruit, whipped/sour cream, ,   
#> 49 pip fruit, yogurt, salty snack, ,   
#> 50 soft cheese, root vegetables, whipped/sour cream, ,   
#> 51 sugar, root vegetables, whipped/sour cream, ,   
#> 52 yogurt, rolls/buns, tropical fruit, beef,   
#> 53 bottled water, other vegetables, butter, citrus fruit,   
#> 54 bottled water, other vegetables, root vegetables, pip fruit,   
#> 55 bottled water, rolls/buns, tropical fruit, root vegetables,   
#> 56 other vegetables, brown bread, root vegetables, pip fruit,   
#> 57 other vegetables, brown bread, root vegetables, rolls/buns,   
#> 58 other vegetables, whipped/sour cream, domestic eggs, butter,   
#> 59 yogurt, domestic eggs, butter, root vegetables,   
#> 60 yogurt, tropical fruit, domestic eggs, butter,   
#> 61 other vegetables, pork, butter, root vegetables,   
#> 62 other vegetables, whipped/sour cream, pork, butter,   
#> 63 other vegetables, root vegetables, butter, white bread,   
#> 64 whole milk, soda, whipped/sour cream, butter,   
#> 65 other vegetables, tropical fruit, butter, white bread,   
#> 66 whole milk, tropical fruit, butter, sliced cheese,   
#> 67 cream cheese , whole milk, whipped/sour cream, citrus fruit,   
#> 68 yogurt, tropical fruit, curd, citrus fruit,   
#> 69 whole milk, domestic eggs, whipped/sour cream, citrus fruit,   
#> 70 other vegetables, fruit/vegetable juice, frozen vegetables, citrus fruit,   
#> 71 other vegetables, frozen vegetables, yogurt, citrus fruit,   
#> 72 tropical fruit, root vegetables, whipped/sour cream, citrus fruit,   
#> 73 rolls/buns, pastry, whipped/sour cream, citrus fruit,   
#> 74 other vegetables, cream cheese , curd, whipped/sour cream,   
#> 75 other vegetables, cream cheese , root vegetables, pip fruit,   
#> 76 other vegetables, cream cheese , yogurt, pip fruit,   
#> 77 tropical fruit, domestic eggs, curd, yogurt,   
#> 78 other vegetables, domestic eggs, whipped/sour cream, pip fruit,   
#> 79 other vegetables, tropical fruit, domestic eggs, whipped/sour cream,   
#> 80 yogurt, tropical fruit, domestic eggs, whipped/sour cream,   
#> 81 yogurt, tropical fruit, root vegetables, frankfurter,   
#> 82 pip fruit, frozen meals, tropical fruit, yogurt,   
#> 83 other vegetables, fruit/vegetable juice, frozen vegetables, root vegetables,   
#> 84 grapes, whole milk, tropical fruit, yogurt,   
#> 85 pip fruit, whole milk, tropical fruit, ham,   
#> 86 pip fruit, tropical fruit, yogurt, ham,   
#> 87 other vegetables, hard cheese, yogurt, root vegetables,   
#> 88 other vegetables, tropical fruit, herbs, root vegetables,   
#> 89 yogurt, tropical fruit, oil, root vegetables,   
#> 90 other vegetables, oil, yogurt, root vegetables,   
#> 91 yogurt, tropical fruit, oil, root vegetables,   
#> 92 other vegetables, rice, root vegetables, yogurt,   
#> 93 yogurt, tropical fruit, root vegetables, pastry,   
#> 94 yogurt, tropical fruit, sausage, root vegetables,   
#> 95 whole milk, citrus fruit, tropical fruit, root vegetables, yogurt  
#> 96 whole milk, citrus fruit, root vegetables, whipped/sour cream, yogurt  
#> 97 yogurt, tropical fruit, oil, root vegetables,   
#> rhs support confidence lift  
#> 1 bottled beer, , 0.0019 0.90 11.3  
#> 2 whole milk, , 0.0011 0.92 3.6  
#> 3 whole milk, , 0.0011 0.92 3.6  
#> 4 whole milk, , 0.0010 0.91 3.6  
#> 5 other vegetables, , 0.0011 0.92 4.7  
#> 6 other vegetables, , 0.0011 0.92 4.7  
#> 7 whole milk, , 0.0011 0.92 3.6  
#> 8 whole milk, , 0.0010 0.91 3.6  
#> 9 whole milk, , 0.0012 1.00 3.9  
#> 10 whole milk, , 0.0010 0.91 3.6  
#> 11 whole milk, , 0.0010 0.91 3.6  
#> 12 other vegetables, , 0.0011 1.00 5.2  
#> 13 whole milk, , 0.0011 0.92 3.6  
#> 14 whole milk, , 0.0010 0.91 3.6  
#> 15 whole milk, , 0.0010 0.91 3.6  
#> 16 yogurt, , 0.0010 0.91 6.5  
#> 17 whole milk, , 0.0010 0.91 3.6  
#> 18 whole milk, , 0.0010 0.91 3.6  
#> 19 other vegetables, , 0.0013 0.93 4.8  
#> 20 other vegetables, , 0.0013 0.93 4.8  
#> 21 whole milk, , 0.0018 0.90 3.5  
#> 22 whole milk, , 0.0011 0.92 3.6  
#> 23 whole milk, , 0.0012 0.92 3.6  
#> 24 other vegetables, , 0.0010 0.91 4.7  
#> 25 other vegetables, , 0.0012 0.92 4.8  
#> 26 whole milk, , 0.0013 0.93 3.6  
#> 27 whole milk, , 0.0011 0.92 3.6  
#> 28 other vegetables, , 0.0010 1.00 5.2  
#> 29 whole milk, , 0.0011 1.00 3.9  
#> 30 whole milk, , 0.0010 1.00 3.9  
#> 31 whole milk, , 0.0014 0.93 3.7  
#> 32 whole milk, , 0.0010 0.91 3.6  
#> 33 whole milk, , 0.0013 0.93 3.6  
#> 34 other vegetables, , 0.0011 0.92 4.7  
#> 35 whole milk, , 0.0010 0.91 3.6  
#> 36 whole milk, , 0.0010 0.91 3.6  
#> 37 whole milk, , 0.0010 0.91 3.6  
#> 38 whole milk, , 0.0013 0.93 3.6  
#> 39 whole milk, , 0.0016 1.00 3.9  
#> 40 other vegetables, , 0.0010 0.91 4.7  
#> 41 whole milk, , 0.0010 0.91 3.6  
#> 42 whole milk, , 0.0010 0.91 3.6  
#> 43 other vegetables, , 0.0010 0.91 4.7  
#> 44 other vegetables, , 0.0010 0.91 4.7  
#> 45 other vegetables, , 0.0010 0.91 4.7  
#> 46 other vegetables, , 0.0011 0.92 4.7  
#> 47 other vegetables, , 0.0014 0.93 4.8  
#> 48 other vegetables, , 0.0012 0.92 4.8  
#> 49 whole milk, , 0.0011 0.92 3.6  
#> 50 whole milk, , 0.0012 0.92 3.6  
#> 51 whole milk, , 0.0011 0.92 3.6  
#> 52 whole milk, , 0.0012 0.92 3.6  
#> 53 whole milk, , 0.0010 0.91 3.6  
#> 54 whole milk, , 0.0011 1.00 3.9  
#> 55 whole milk, , 0.0011 0.92 3.6  
#> 56 whole milk, , 0.0012 0.92 3.6  
#> 57 whole milk, , 0.0010 0.91 3.6  
#> 58 whole milk, , 0.0011 1.00 3.9  
#> 59 whole milk, , 0.0010 0.91 3.6  
#> 60 whole milk, , 0.0011 0.92 3.6  
#> 61 whole milk, , 0.0010 0.91 3.6  
#> 62 whole milk, , 0.0010 1.00 3.9  
#> 63 whole milk, , 0.0010 1.00 3.9  
#> 64 other vegetables, , 0.0010 0.91 4.7  
#> 65 yogurt, , 0.0010 0.91 6.5  
#> 66 yogurt, , 0.0010 0.91 6.5  
#> 67 other vegetables, , 0.0011 0.92 4.7  
#> 68 whole milk, , 0.0010 0.91 3.6  
#> 69 other vegetables, , 0.0011 0.92 4.7  
#> 70 whole milk, , 0.0010 0.91 3.6  
#> 71 whole milk, , 0.0010 0.91 3.6  
#> 72 other vegetables, , 0.0012 1.00 5.2  
#> 73 whole milk, , 0.0010 1.00 3.9  
#> 74 yogurt, , 0.0010 0.91 6.5  
#> 75 whole milk, , 0.0010 0.91 3.6  
#> 76 whole milk, , 0.0011 0.92 3.6  
#> 77 whole milk, , 0.0010 0.91 3.6  
#> 78 whole milk, , 0.0012 0.92 3.6  
#> 79 whole milk, , 0.0010 0.91 3.6  
#> 80 whole milk, , 0.0010 0.91 3.6  
#> 81 whole milk, , 0.0012 0.92 3.6  
#> 82 whole milk, , 0.0010 0.91 3.6  
#> 83 whole milk, , 0.0012 0.92 3.6  
#> 84 other vegetables, , 0.0010 1.00 5.2  
#> 85 other vegetables, , 0.0011 1.00 5.2  
#> 86 other vegetables, , 0.0010 1.00 5.2  
#> 87 whole milk, , 0.0012 0.92 3.6  
#> 88 whole milk, , 0.0010 0.91 3.6  
#> 89 other vegetables, , 0.0010 0.91 4.7  
#> 90 whole milk, , 0.0014 1.00 3.9  
#> 91 whole milk, , 0.0011 1.00 3.9  
#> 92 whole milk, , 0.0013 0.93 3.6  
#> 93 whole milk, , 0.0010 0.91 3.6  
#> 94 whole milk, , 0.0015 0.94 3.7  
#> 95 other vegetables, , 0.0014 0.93 4.8  
#> 96 other vegetables, , 0.0010 0.91 4.7  
#> 97 other vegetables, whole milk, 0.0010 0.91 12.1

### 8. Conclusion

The Market Basket Analysis used the Apriori Algorithm to perform Association Rule based on two programming languages R and Python. The comparison was performed in RStudio operating system, and Python reticulate package was loaded to use within R session. There was some setup required in order to use Python in the RStudio IDE. Packages needed to be installed in a certain way and rechecked in the environment. One can accomplish this in the “R-reticulate” environment in Anaconda and using pip install.

The sample transaction data set provided these insights based on the programming languages:

# Apriori Algorithm with R  
options(digits = 2)  
#visualize the results  
paste0("Apriori Algorith with R")

#> [1] "Apriori Algorith with R"

inspect(sort(rules, by="lift", decreasing=TRUE)[1:5])

#> lhs rhs support confidence coverage lift count  
#> [1] {liquor,   
#> red/blush wine} => {bottled beer} 0.0019 0.90 0.0021 11.2 19  
#> [2] {citrus fruit,   
#> fruit/vegetable juice,   
#> other vegetables,   
#> soda} => {root vegetables} 0.0010 0.91 0.0011 8.3 10  
#> [3] {oil,   
#> other vegetables,   
#> tropical fruit,   
#> whole milk,   
#> yogurt} => {root vegetables} 0.0010 0.91 0.0011 8.3 10  
#> [4] {butter,   
#> cream cheese,   
#> root vegetables} => {yogurt} 0.0010 0.91 0.0011 6.5 10  
#> [5] {butter,   
#> sliced cheese,   
#> tropical fruit,   
#> whole milk} => {yogurt} 0.0010 0.91 0.0011 6.5 10

# Apriori Algorithm with Python  
options(digits = 2)  
#visualize the results  
head(py\_apriori[order(-py\_apriori$lift), ], 5)

#> lhs  
#> 97 yogurt, tropical fruit, oil, root vegetables,   
#> 1 red/blush wine, liquor, , ,   
#> 16 cream cheese , root vegetables, butter, ,   
#> 65 other vegetables, tropical fruit, butter, white bread,   
#> 66 whole milk, tropical fruit, butter, sliced cheese,   
#> rhs support confidence lift  
#> 97 other vegetables, whole milk, 0.0010 0.91 12.1  
#> 1 bottled beer, , 0.0019 0.90 11.3  
#> 16 yogurt, , 0.0010 0.91 6.5  
#> 65 yogurt, , 0.0010 0.91 6.5  
#> 66 yogurt, , 0.0010 0.91 6.5

**The associations sorted by lift shows:**

* R - customers who bought liquor, red/blush wine (lhs) also bought bottled beer (rhs) with the highest lift value (11.2) and listed as the second in Python (lift value = 11.3).
* Python - customers who oil, root vegetables, tropical fruit, yogurt (lhs) also bought whole milk, other vegetables (rhs) with the highest lift value (12.1) and have a different rank, (rhs), and lift in R.

**The association rules based on apriori algorithm parameter:**

support=0.001,confidence=0.9, minlen= 2, maxlen=10

* R - generated a set of 129 rules
* Python - generated a set 97 rules

**Visualization**

In RStudio, using Python visualizations packages, matplotlib and Seaborn, are cumbersome for in-experience Python users. The data visualization with a Python script require plug-ins for inline plots and require additional help aids to successful install.

### 9. References:

* Bhalla, D. (2015). Market Basket Analysis with R. ListenData. <https://www.listendata.com/2015/12/market-basket-analysis-with-r.html>
* Abbas, M. M. (2022, May 6). Python Apriori Algorithm. Delft Stack. <https://www.delftstack.com/howto/python/apriori-algorithm-python/#:%7E:text=%20Apriori%20Algorithm%20in%20Python%20%201%20Our,2%20until%20the%20specified%20k%20is>. . .%20More%20