Data 624 Batch 2 Homework

Homework Chapter 6-8 & Recommender System

Subhalaxmi Rout, Kenan Sooklall,Leo Yi, Christian Thieme, Devin Teran

7/18/2021

[**Chapter 6 Exercises**](#_heading=h.ivzr50qwg3gi) **2**

[**Exercise 6**](#_heading=h.mufzvjfa1cy2) **2**

[**Chapter 7 Exercises**](#_heading=h.bdoqs1oufnot) **19**

[**Exercise 7.2**](#_heading=h.ds73n4bf91kz) **19**

[**Exercise 7.5**](#_heading=h.gggdnlkevjiv) **28**

[**Chapter 8 Exercises**](#_heading=h.srfymdq6tvgt) **36**

[**Exercise 8.1**](#_heading=h.eyq4jp36ym0) **36**

[**Exercise 8.2**](#_heading=h.rxlsuahddzny) **42**

[**Exercise 8.3**](#_heading=h.tqwkxprbvlhy) **43**

[**Recommender System**](#_heading=h.qfwl4fh98zgv) **53**

[**Market Basket Analysis in R**](#_heading=h.evsysb7sofq0) **53**

[**Market Basket Analysis in Python**](#_heading=h.ormyzahv6t6y) **67**

[**Comparison of Market Basket Analysis in Python vs R**](#_heading=h.8yj9wskpux4i) **76**

# Chapter 6 Exercises

# Exercise 6

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

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

library(AppliedPredictiveModeling)  
data(ChemicalManufacturingProcess)  
data <- ChemicalManufacturingProcess

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

### 

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

There are a total of 24 rows having NAs.

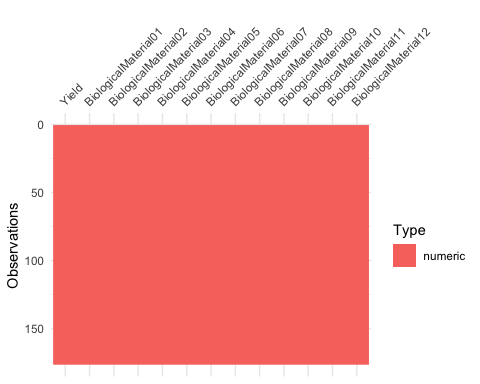
sum(!complete.cases(data))

## [1] 24

**Missing Yield & Biological Predictors** Let’s first review the **yield** and **biological** predictors. The function, **vis\_dat**, is an easy way to visualize missing data. The y-axis represents the observation with row 1 at the top and increasing as you move down the y-axis. Each column of data is represented on the x-axis. If a value is missing, you’ll see it highlighted in grey otherwise it will be pink.

Here there are no values missing from any of the biological or yield columns.

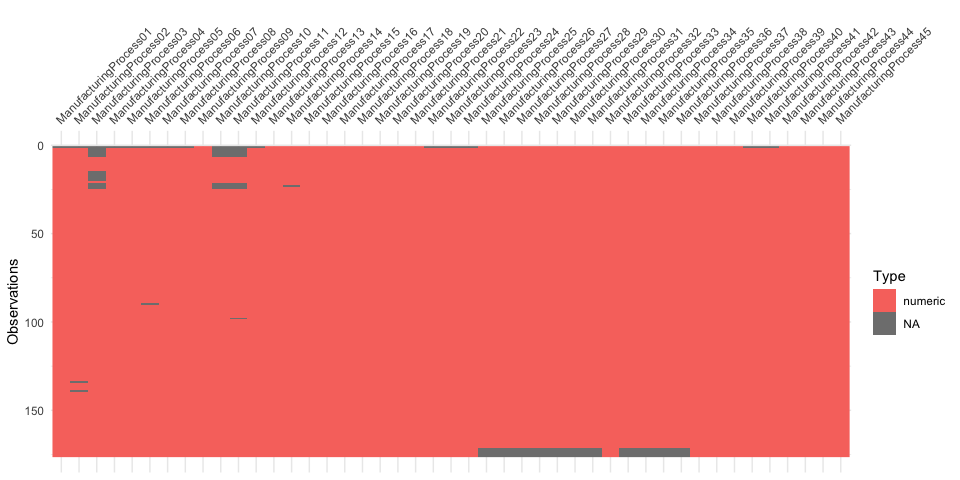
yield\_and\_bio <- data[,c(1:13)]  
  
vis\_dat(yield\_and\_bio)



### Missing Process Predictors

Now let’s review if any values are missing from the **manufacturing process** predictors. We see more grey rectangles which indicates there are more na values.

process <- data[,c(14:58)]  
  
vis\_dat(process)



**The first observation includes many na values:** \* ManufacturingProcess01 to ManufacturingProcess08 \* ManufacturingProcess10 to ManufacturingProcess12 \* ManufacturingProcess22 to ManufacturingProcess24 \* ManufacturingProcess40 to ManufacturingProcess41

**Many of the observations from row 1-25 are missing the many values:** \* ManufacturingProcess03, ManufacturingProcess10, & ManufacturingProcess11

**Observations 170-176 are are missing the most values:** \* ManufacturingProcess25 to ManufacturingProcess36 with the exception of ManufacturingProcess32

### Impute Missing Values

In order to complete our analysis, we are going to impute these missing values using the **preProcess()** function using method **knnImpute** which can be found in the caret package.

If we view the first row which we previously saw was missing a ton of values, we now see they have been filled in.

imp <- preProcess(data, method = "knnImpute")  
imputed <- predict(imp,data)  
imputed[c(1),]

## Yield BiologicalMaterial01 BiologicalMaterial02 BiologicalMaterial03  
## 1 -1.179267 -0.2261036 -1.514098 -2.683036  
## BiologicalMaterial04 BiologicalMaterial05 BiologicalMaterial06  
## 1 0.2201765 0.4941942 -1.382888  
## BiologicalMaterial07 BiologicalMaterial08 BiologicalMaterial09  
## 1 -0.1313107 -1.233131 -3.396289  
## BiologicalMaterial10 BiologicalMaterial11 BiologicalMaterial12  
## 1 1.10053 -1.838655 -1.770922  
## ManufacturingProcess01 ManufacturingProcess02 ManufacturingProcess03  
## 1 0.2154105 0.5662872 0.376581  
## ManufacturingProcess04 ManufacturingProcess05 ManufacturingProcess06  
## 1 0.5655598 -0.4459347 -0.5414997  
## ManufacturingProcess07 ManufacturingProcess08 ManufacturingProcess09  
## 1 -0.15967 -0.3095182 -1.720152  
## ManufacturingProcess10 ManufacturingProcess11 ManufacturingProcess12  
## 1 -0.07700901 -0.09157342 -0.4806937  
## ManufacturingProcess13 ManufacturingProcess14 ManufacturingProcess15  
## 1 0.9771151 0.8093999 1.184644  
## ManufacturingProcess16 ManufacturingProcess17 ManufacturingProcess18  
## 1 0.3303945 0.9263296 0.1505348  
## ManufacturingProcess19 ManufacturingProcess20 ManufacturingProcess21  
## 1 0.4563798 0.3109942 0.2109804  
## ManufacturingProcess22 ManufacturingProcess23 ManufacturingProcess24  
## 1 0.05833309 0.8317688 0.8907291  
## ManufacturingProcess25 ManufacturingProcess26 ManufacturingProcess27  
## 1 0.1200183 0.1256347 0.3460352  
## ManufacturingProcess28 ManufacturingProcess29 ManufacturingProcess30  
## 1 0.7826636 0.5943242 0.7566948  
## ManufacturingProcess31 ManufacturingProcess32 ManufacturingProcess33  
## 1 -0.1952552 -0.4568829 0.9890307  
## ManufacturingProcess34 ManufacturingProcess35 ManufacturingProcess36  
## 1 -1.720272 -0.8869472 -0.6557774  
## ManufacturingProcess37 ManufacturingProcess38 ManufacturingProcess39  
## 1 -1.154024 0.7174727 0.231727  
## ManufacturingProcess40 ManufacturingProcess41 ManufacturingProcess42  
## 1 0.05969714 -0.06900773 0.2027957  
## ManufacturingProcess43 ManufacturingProcess44 ManufacturingProcess45  
## 1 2.405647 -0.01588055 0.6437185

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

Here we are going to start by using 80% of the data set to train our model. The training set contains 144 rows and 58 columns. Similarly, the test set contains 32 rows and 58 columns. The data set has a target variable or dependent variable called Yield.

We will use the dataframe with imputed values, *imputed*. To do this we randomly sample 80% of the observations based on their row index.

*#for reproducibility*  
set.seed(123)  
train\_index <- sample(nrow(imputed),nrow(imputed)\*0.8)  
  
*#get subset of ids*  
train\_pred <- imputed[train\_index,-1]  
train\_output <- imputed[train\_index,1]  
  
test\_pred <- imputed[-train\_index,-1]  
test\_output <- imputed[-train\_index,1]

### Modeling Data with Ridge Regression

For this problem, we will use the **ridge regression** model. Ridge regression is used when the predictors in the data set are highly correlated by adding a penalty error to the sum of squared errors. It uses, *alpha*, as a tuning parameter for the penalty.

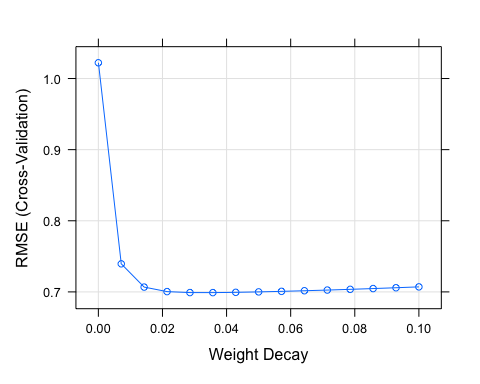
Next we are going to determine which value of lambda returns the best model. We know which lambda is best because it will produce the model with the lowest value of RMSE.

ctrl <- trainControl(method = "cv", number = 10)  
*#creating grid of possible values for lambda*  
ridgeGrid <- data.frame(.lambda = seq(0, .1, length = 15))  
  
set.seed(123)*#for reproducibility*ridgeRegFit <- train(train\_pred,train\_output,  
 method='ridge',  
 *#Our different alpha penalty values*  
 tuneGrid=ridgeGrid,  
 trControl = ctrl,  
 *#center & get pred variables on same scale*  
 preProc=c("center","scale"))  
ridgeRegFit

## Ridge Regression   
##   
## 140 samples  
## 57 predictor  
## Pre-processing: centered (57), scaled (57)   
## Resampling: Cross-Validated (10 fold)   
## Summary of sample sizes: 126, 126, 125, 125, 125, 127, ...   
## Resampling results across tuning parameters:  
##   
## lambda RMSE Rsquared MAE   
## 0.000000000 1.0221525 0.3674877 0.7492669  
## 0.007142857 0.7394789 0.5192863 0.6083392  
## 0.014285714 0.7067733 0.5427204 0.5836053  
## 0.021428571 0.7004064 0.5455998 0.5818266  
## 0.028571429 0.6990414 0.5459039 0.5802543  
## 0.035714286 0.6989843 0.5459852 0.5790184  
## 0.042857143 0.6993876 0.5461214 0.5776078  
## 0.050000000 0.7000090 0.5463097 0.5763720  
## 0.057142857 0.7007663 0.5465197 0.5761176  
## 0.064285714 0.7016265 0.5467293 0.5763379  
## 0.071428571 0.7025735 0.5469253 0.5767611  
## 0.078571429 0.7035971 0.5471011 0.5771880  
## 0.085714286 0.7046902 0.5472540 0.5775532  
## 0.092857143 0.7058465 0.5473833 0.5778765  
## 0.100000000 0.7070609 0.5474898 0.5784133  
##   
## RMSE was used to select the optimal model using the smallest value.  
## The final value used for the model was lambda = 0.03571429.

This plot shows the change in RMSE as alpha increases.

plot(ridgeRegFit)



Our best model uses alpha equal to 0.0357. This model has an R^2 of 55% and RMSE of 0.7.

best\_ridge\_model = ridgeRegFit$results[ridgeRegFit$results$RMSE == min(ridgeRegFit$results$RMSE),]  
best\_ridge\_model

## lambda RMSE Rsquared MAE RMSESD RsquaredSD MAESD  
## 6 0.03571429 0.6989843 0.5459852 0.5790184 0.0756335 0.1023629 0.07059589

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

Here we use our best ridge regression model to see how well it predicts on our test data. We set s = 1 and mode = ‘fraction’ in order to get a ridge regression model.

ridgePred\_Test <-predict(ridge\_model,newx=as.matrix(test\_pred),s=1,mode="fraction",type="fit")  
ridgeCoeff\_Test <-predict(ridge\_model,newx=as.matrix(test\_pred),s=1,mode="fraction",type="coefficients")

Now we can compare how our model performed on our train data versus our test data.

*# test performance*  
model\_values <- data.frame(obs = test\_output, pred = ridgePred\_Test$fit)  
test\_performance <- defaultSummary(model\_values)  
  
*# train matrices*  
train\_performance <- best\_ridge\_model[,c('RMSE','Rsquared','MAE')]  
  
*# create data frame of both train + test performances*  
perf\_df <- rbind(t(data.frame(test\_performance)), data.frame(train\_performance))  
  
*# Rename column*  
names(perf\_df) <- c("RMSE", "Rsquared", "MAE")  
rownames(perf\_df) <- c("Test","Train")  
  
*# Train and Test matrices*  
perf\_df

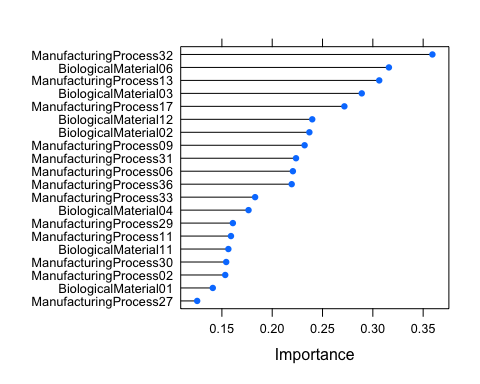
## RMSE Rsquared MAE  
## Test 3.2672513 0.07583988 0.9605606  
## Train 0.6989843 0.54598520 0.5790184

The RMSE value for the test data is 3.3 and the RMSE value for the training set is 0.7. The lower the RMSE value, the better the model. The R^2 value for the test set is lower than the training set, which indicates the model accounts for the variability in the training data better than the test data. These metrics tell us that our model did not perform as well as on the testing data as it did on the training data, which isn’t too surprising.

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

variable\_importance <- varImp(ridgeRegFit,scale=F)

plot(variable\_importance, top = 20)

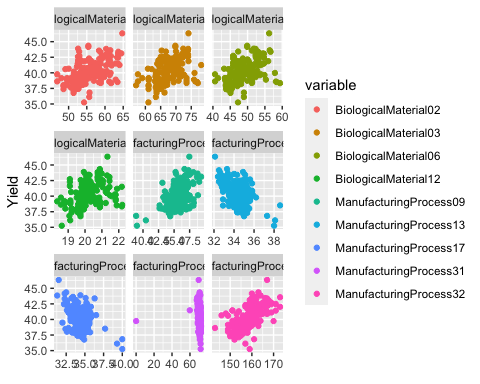


**ManufacturingProcess32** is the most important variable followed by **BiologicalMaterial06**. The manufacturing predictors are appearing more frequently in our list and appear to more important in our model.

## 

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

vi\_df <- variable\_importance$importance %>%  
 data.frame() %>%  
 top\_n(9)  
vi\_df$var <- rownames(vi\_df)  
  
df\_top\_x <- data[,names(imputed) %in% vi\_df$var] %>%  
 bind\_cols(Yield = data$Yield)  
df\_top\_x %>%  
 gather(variable, value, -Yield) %>%  
 ggplot(aes(x = value, y = Yield,color=variable)) +  
 geom\_point() +  
 facet\_wrap(~variable, scales = 'free\_x') +  
 labs(x = element\_blank())



The sub-plots above illustrate positive and negative correlations with the response variable. To improve the yield in the future runs, the manufacturing processes with positive coefficients should be boosted or increased, while the processes with negative coefficients should be reduced. The majority of the important variables have a positive relationship except **ManufacturingProcess17** and **ManufacturingProcess13**. By knowing which variables that affect yield and the relationship of each variable against yield, we can determine how to adjust each material or process to maximize results.

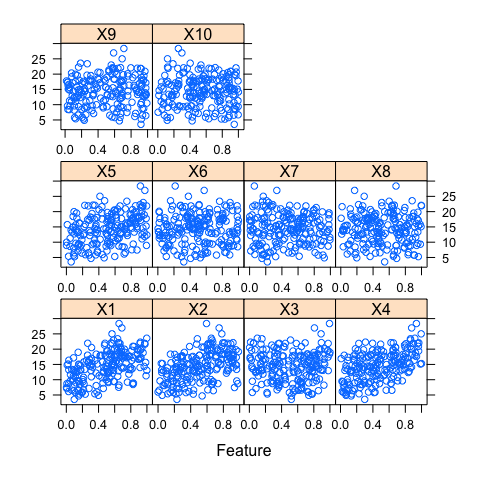
# Chapter 7 Exercises

# Exercise 7.2

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

## 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:

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***  
obj <- featurePlot(trainingData$x, trainingData$y)  
print(obj)



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

Tune several models on these data. For example:

### Knn Model

library(caret)  
set.seed(123)  
knnModel <- train(x = trainingData$x,  
 y = trainingData$y,  
 preProc = c("center", "scale"),  
 tuneLength = 10)

## note: only 9 unique complexity parameters in default grid. Truncating the grid to 9 .

knnModel

## Random Forest   
##   
## 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:  
##   
## mtry RMSE Rsquared MAE   
## 2 2.942963 0.7786600 2.409166  
## 3 2.757348 0.7780577 2.257721  
## 4 2.668106 0.7709207 2.184489  
## 5 2.638548 0.7580862 2.165022  
## 6 2.623661 0.7511115 2.152904  
## 7 2.626736 0.7401869 2.155715  
## 8 2.640345 0.7321122 2.162547  
## 9 2.657499 0.7241281 2.176910  
## 10 2.674946 0.7177772 2.190956  
##   
## RMSE was used to select the optimal model using the smallest value.  
## The final value used for the model was mtry = 6.

knnPred <- predict(knnModel, newdata = testData$x)  
***## The function 'postResample' can be used to get the test set***  
***## performance values***  
kNN\_Results <- postResample(pred = knnPred, obs = testData$y)  
kNN\_Results

## RMSE Rsquared MAE   
## 2.4381466 0.7992296 1.9308194

### MARS Model

Now we are going to run the MARS modeling using the package **earth**. MARS stands for Multivariate Adaptive Regression Splines. Like neural networks and partial least squares, MARS (Friedman 1991) uses surrogate features instead of the original predictors.

When running a MARS model, you are able to set the value for **nprune** which is the number of terms allowed to be included in the final model. Here we’ve chosen 20 since that is more than the number of X columns in order to allow for all the terms to be included and leave room for interactions. You can also set the degree, which is the number of hinge functions that can interact. We’ll pick 3 for the degree.

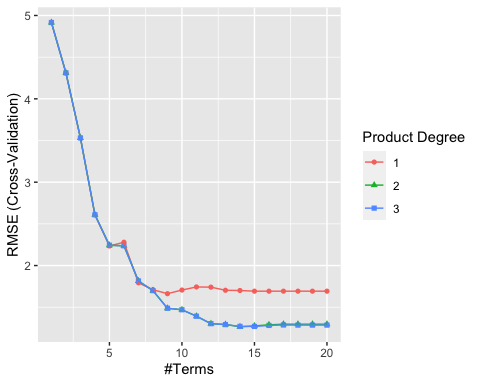
library(earth)  
set.seed(123)  
ctrl <- trainControl(method = "cv", number = 10)  
*#set values for nprune and degree*  
parameter\_Grid <- expand.grid(nprune=seq(1:20),degree=seq(1:3))

*#tune model*  
MARSModel <- train(trainingData$x,  
 trainingData$y,  
 method='earth',  
 metric='RMSE',  
 tuneGrid=parameter\_Grid,  
 trControl = ctrl)  
summary(MARSModel)

## Call: earth(x=data.frame[200,10], y=c(18.46,16.1,17...), keepxy=TRUE, degree=2,  
## nprune=14)  
## coefficients  
## (Intercept) 21.905319  
## h(0.621722-X1) -15.726181  
## h(X1-0.621722) 9.234027  
## h(0.601063-X2) -18.253527  
## h(X2-0.601063) 10.448545  
## h(0.447442-X3) 9.700589  
## h(X3-0.606015) 12.674694  
## h(0.734892-X4) -9.863526  
## h(X4-0.734892) 10.297964  
## h(0.850094-X5) -5.324175  
## h(0.218266-X1) \* h(X2-0.601063) -55.358726  
## h(X1-0.218266) \* h(X2-0.601063) -29.100250  
## h(X1-0.621722) \* h(X2-0.295997) -26.833129  
## h(0.649253-X1) \* h(0.601063-X2) 27.120721  
##   
## Selected 14 of 18 terms, and 5 of 10 predictors (nprune=14)  
## Termination condition: Reached nk 21  
## Importance: X1, X4, X2, X5, X3, X6-unused, X7-unused, X8-unused, X9-unused, ...  
## Number of terms at each degree of interaction: 1 9 4  
## GCV 1.62945 RSS 225.8601 GRSq 0.9338437 RSq 0.953688

Here we are going to visualize each of the models that were created in the previous step. It looks that the model with degree = 3 and 20 terms has the lowest RMSE, which makes it the best model.

ggplot(MARSModel)



Now we will predict using the test data using our best MARS model and review the model results.

mars\_predictions <- predict(MARSModel$finalModel,testData$x)  
  
MARS\_Results <- postResample(pred = mars\_predictions, obs = testData$y)  
MARS\_Results

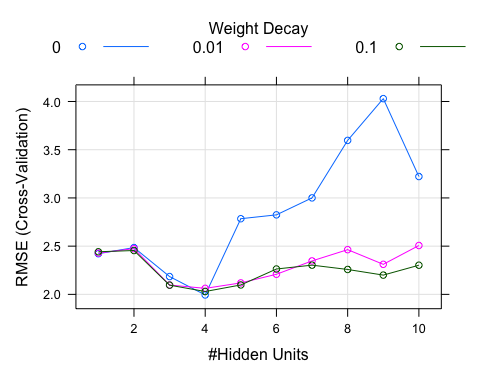
## RMSE Rsquared MAE   
## 1.1722635 0.9448890 0.9324923

### Neural Network Model

Neural networks are powerful nonlinear regression techniques inspired by theories about how the brain works. Like partial least squares, the outcome is modeled by an intermediary set of unobserved variables (called hidden variables or hidden units here). This model has input, hidden layers and output.

Now we will tune the model. The parameters that take on multiple values are **hidden units** and **amount of weight decay**. We will do this using the train function as we have before.

neural\_Grid <- expand.grid(.decay = c(0,0.01,0.1),  
 .size = c(1:10),  
 .bag = FALSE)  
set.seed(123)  
nnetModel <- train(trainingData$x,trainingData$y,  
 method = "avNNet",  
 tuneGrid = neural\_Grid,  
 trControl = ctrl,  
 preProc = c("center","scale"), *#center & scale the data*  
 linout = TRUE,  
 trace = FALSE,  
 *#MaxNWts = 10 \* (ncol(trainingData$x) + 1) + 10 + 1,*  
 maxit = 500)  
plot(nnetModel)



The best model was using the weight decay 0.1 and with 10 hidden units. Now we will predict on the test data using our neural network model and the results.

nn\_predictions <- predict(nnetModel$finalModel,testData$x)  
nn\_Results <- postResample(pred = nn\_predictions, obs = testData$y)  
nn\_Results

## RMSE Rsquared MAE   
## 5.5708330 0.6800083 4.5994262

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

### Model Summary

It looks like the MARS model outperforms kNN since the RMSE & MAE are lower and R^2 is higher.

results <- data.frame(rbind(nn\_Results,kNN\_Results,MARS\_Results))  
results

## RMSE Rsquared MAE  
## nn\_Results 5.570833 0.6800083 4.5994262  
## kNN\_Results 2.438147 0.7992296 1.9308194  
## MARS\_Results 1.172263 0.9448890 0.9324923

### MARS most important predictors

The informative predictors are X1-X5.

varImp(MARSModel$finalModel)

## Overall  
## X1 100.00000  
## X4 85.13408  
## X2 69.22036  
## X5 49.27524  
## X3 39.95037

# Exercise 7.5

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

This code is from week 6 homework problems and imputes missing values and splits the data into training and testing sets.

library(AppliedPredictiveModeling)  
data(ChemicalManufacturingProcess)  
data <- ChemicalManufacturingProcess  
imp <- preProcess(data, method = "knnImpute")  
imputed <- predict(imp,data)  
  
set.seed(123)  
ids <- sample(nrow(imputed),nrow(imputed)\*0.8)  
  
train\_pred <- imputed[ids,-1]  
train\_output <- imputed[ids,1]  
  
test\_pred <- imputed[-ids,-1]  
test\_output <- imputed[-ids,1]

Now we will use several nonlinear regression models to see which performs best on our data.

### 

### Neural Network

As stated previously, neural networks are powerful nonlinear regression techniques inspired by theories about how the brain works. Like partial least squares, the outcome is modeled by an intermediary set of unobserved variables (called hidden variables or hidden units here). This model has input, hidden layers and output.

neural\_Grid <- expand.grid(.decay = c(0,0.01,0.1),  
 .size = c(1:10),  
 .bag = FALSE)  
  
set.seed(123)  
  
nnetTune <- train(train\_pred,train\_output,  
 method = "avNNet",  
 tuneGrid = neural\_Grid,  
 trControl = ctrl,  
 *#center & scale the data*  
 preProc = c("center","scale"),  
 linout = TRUE,  
 trace = FALSE,  
 maxit = 500)

nn\_predictions <- predict(nnetTune$finalModel,test\_pred)  
  
nn\_results <- postResample(pred = nn\_predictions, obs = test\_output)  
nn\_results

## RMSE Rsquared MAE   
## 0.6350121 0.6818911 0.4776292

### 

### SVM

SVMs are a class of powerful, highly flexible modeling techniques. The theory behind SVMs was originally developed in the context of classification models.

library(kernlab)  
svmTune <- train(train\_pred, train\_output,  
 method = "svmRadial",  
 preProc = c("center", "scale"),  
 tuneLength = 14,  
 trControl = trainControl(method = "cv"))

svm\_predictions <- predict(svmTune$finalModel,test\_pred)  
  
SVM\_results <- postResample(pred = svm\_predictions, obs = test\_output)  
SVM\_results

## RMSE Rsquared MAE   
## 0.6522754 0.7663851 0.4791971

### kNearestNeighbors

Now we will run the kNN model and will center and scale the data.

library(caret)  
knnTune <- train(x = train\_pred,  
 y = train\_output,  
 preProc = c("center", "scale"),  
 tuneLength = 10)  
  
knnTune

## Random Forest   
##   
## 140 samples  
## 57 predictor  
##   
## Pre-processing: centered (57), scaled (57)   
## Resampling: Bootstrapped (25 reps)   
## Summary of sample sizes: 140, 140, 140, 140, 140, 140, ...   
## Resampling results across tuning parameters:  
##   
## mtry RMSE Rsquared MAE   
## 2 0.7235649 0.5378986 0.5712462  
## 8 0.6891294 0.5566480 0.5384496  
## 14 0.6857843 0.5507213 0.5341980  
## 20 0.6866935 0.5436037 0.5332588  
## 26 0.6881685 0.5367398 0.5337969  
## 32 0.6899401 0.5308883 0.5350922  
## 38 0.6957190 0.5192489 0.5385073  
## 44 0.7004385 0.5107343 0.5428744  
## 50 0.7048170 0.5027933 0.5469711  
## 57 0.7104203 0.4917456 0.5531747  
##   
## RMSE was used to select the optimal model using the smallest value.  
## The final value used for the model was mtry = 14.

knnPred <- predict(knnTune, newdata = test\_pred)  
  
knn\_Results <- postResample(pred = knnPred, obs = test\_output)  
knn\_Results

## RMSE Rsquared MAE   
## 0.6466251 0.7747683 0.5041527

### MARS

As previously stated, MARS models work like neural networks and partial least squares, MARS (Friedman 1991) uses surrogate features instead of the original predictors.

ctrl <- trainControl(method = "cv", number = 10)  
parameter\_Grid <- expand.grid(nprune=seq(1:20),degree=seq(1:3))  
  
MARS\_Fit <- train(train\_pred,  
 train\_output,  
 method='earth',  
 metric='RMSE',  
 tuneGrid=parameter\_Grid,  
 trControl = ctrl)

mars\_predictions <- predict(MARS\_Fit$finalModel,test\_pred)  
  
MARS\_results <- postResample(pred = mars\_predictions, obs = test\_output)  
MARS\_results

## RMSE Rsquared MAE   
## 0.6441535 0.6703288 0.5029883

## 

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

The kNearestNeighbor model performs the best because it has the highest R^2 value and ties with the lowest RMSE & MAE values.

results <- data.frame(rbind(knn\_Results,MARS\_results,SVM\_results,nn\_results))  
results

## RMSE Rsquared MAE  
## knn\_Results 0.6466251 0.7747683 0.5041527  
## MARS\_results 0.6441535 0.6703288 0.5029883  
## SVM\_results 0.6522754 0.7663851 0.4791971  
## nn\_results 0.6350121 0.6818911 0.4776292

## (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?

In the nonlinear regression model, the manufacturing predictors are seen more but not by much.

nl\_variable\_importance <- varImp(knnTune,scale=F)  
  
topPred\_knn <- as.data.frame(varImp(knnTune$finalModel))  
topPred\_knn <- data.frame(overall = topPred\_knn$Overall,  
 names = rownames(topPred\_knn))  
non\_linear\_top\_predictors <- topPred\_knn[order(topPred\_knn$overall,decreasing = T),]  
head(non\_linear\_top\_predictors,10)

## overall names  
## 44 21.555244 ManufacturingProcess32  
## 3 7.326339 BiologicalMaterial03  
## 6 7.137021 BiologicalMaterial06  
## 25 6.129694 ManufacturingProcess13  
## 12 5.551004 BiologicalMaterial12  
## 18 4.508945 ManufacturingProcess06  
## 29 4.073859 ManufacturingProcess17  
## 21 3.975620 ManufacturingProcess09  
## 43 3.558485 ManufacturingProcess31  
## 4 3.508102 BiologicalMaterial04

In both the linear and nonlinear regression model, 9/10 of the top variables are the same. **BiologicalMaterial04** is in the nonlinear regression model and not in the linear regression model. **BiologicalMaterial02** is in the linear regression model and not in the nonlinear regression model.

topPred\_nonlinear <- head(topPred\_knn[order(topPred\_knn$overall,decreasing = T),],10)$names  
topPred\_linear <- c('ManufacturingProcess32','BiologicalMaterial06','Manufacturing

Process13', 'BiologicalMaterial03','ManufacturingProcess17','BiologicalMaterial12’, 'BiologicalMaterial02','ManufacturingProcess09','ManufacturingProcess31','ManufacturingProcess06')

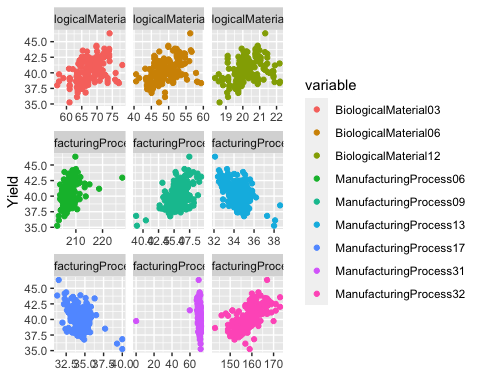
topPreds\_in\_Both <- as.data.frame(cbind(topPred\_nonlinear,topPred\_linear))  
topPreds\_in\_Both

## topPred\_nonlinear topPred\_linear  
## 1 ManufacturingProcess32 ManufacturingProcess32  
## 2 BiologicalMaterial03 BiologicalMaterial06  
## 3 BiologicalMaterial06 ManufacturingProcess13  
## 4 ManufacturingProcess13 BiologicalMaterial03  
## 5 BiologicalMaterial12 ManufacturingProcess17  
## 6 ManufacturingProcess06 BiologicalMaterial12  
## 7 ManufacturingProcess17 BiologicalMaterial02  
## 8 ManufacturingProcess09 ManufacturingProcess09

## 9 ManufacturingProcess31 ManufacturingProcess31  
## 10 BiologicalMaterial04 ManufacturingProcess06

## (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?

x <- 9  
  
vi\_df <- nl\_variable\_importance$importance %>%  
 data.frame() %>%  
 top\_n(x)  
  
vi\_df$var <- rownames(vi\_df)  
  
df\_top\_x <- data[,names(imputed) %in% vi\_df$var] %>%  
 bind\_cols(Yield = data$Yield)  
  
df\_top\_x %>%  
 gather(variable, value, -Yield) %>%  
 ggplot(aes(x = value, y = Yield,color=variable)) +  
 geom\_point() +  
 facet\_wrap(~variable, scales = 'free\_x') +  
 labs(x = element\_blank())



The predictor, ManufacturingProcess32, is the top predictor in both the linear and nonlinear regression models. This predictor contains data which are discrete, whole numbers. There is a strong linear relationship between the two variables with a correlation of 61%. The measurements from ManufacturingProcess32 would be extremely helpful in trying to maximize the Yield.

Looking at the results above, we see few differences than reported in 6.3. We can see that the features are strongly correlated. For BiologicalMaterial 03, 06, and 12 as well as ManufacturingProcess 06, 09, and 32, the more we increase these values, the greater our yield will be. Conversely, for those not mentioned, these variables have negatively correlated relationships, meaning the more we can minimize these values, the higher our yield will be.

# Chapter 8 Exercises

# Exercise 8.1

8.1. Recreate the simulated data from Exercise 7.2:

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

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

library(randomForest)  
library(caret)  
set.seed(123)  
model1 <- randomForest(y ~ ., data = simulated, importance = TRUE,ntree = 1000)  
rfImp1 <- varImp(model1, scale = FALSE)

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

The predictors V6 - V10 have a variable importance score of near 0, which tells me the predictors were not significant in the model.

rfImp1

## Overall  
## V1 8.864171202  
## V2 6.657873300  
## V3 0.841086266  
## V4 7.693792783  
## V5 2.127834542  
## V6 0.151948926  
## V7 0.098320233  
## V8 0.001378668  
## V9 -0.038870929  
## V10 0.014593264

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

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

## [1] 0.9496502

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

After adding the variable that was highly correlated with V1 and rerunning the model, the importance score for V1 did decrease from 8.86 to 5.01. The new variable, duplicateV1, had an importance score of 4.32. This makes sense because we know if you have a variable that has an importance score of X and you add a highly correlated second variable, that each of the correlated variables will have an importance score of roughly X/2.

This is referenced on page 202 of Applied Predictive Modeling book by Max Kuhn and Kjell Johnson.

set.seed(123)  
model2 <- randomForest(y ~ ., data = simulated, importance = TRUE,ntree = 1000)  
rfImp2 <- varImp(model2, scale = FALSE)  
rfImp2

## Overall  
## V1 5.68408955  
## V2 6.14426393  
## V3 0.71319501  
## V4 7.18760270  
## V5 2.29237419  
## V6 0.10729895  
## V7 0.12786886  
## V8 -0.07487799  
## V9 -0.04668380  
## V10 -0.01539013  
## duplicate1 4.00886524

Now we are going to add a second variable that is highly correlated with V1. This means we have 3 total variables that are extremely highly correlated with one another.

simulated$duplicate2 <- simulated$V1 + rnorm(200) \* .1  
cor(simulated$duplicate2, simulated$V1)

## [1] 0.9433552

After creating a third model which included the two new variables that are highly correlated with V1, we see that the importance score for V1 drops even more. V1 importance dropped from 8 to 5.01 to 3.87 in this third model. The duplicate variables have lower importance scores than V1 (duplicate1: 2.76 and duplicate2: 3.94).

Our book tells us that if we have 3 correlated variables with an original importance of X, the model with all 3 correlated variables, will cause the importance score to drop to X/3. Our data follows this statement pretty closely.

set.seed(123)  
model3 <- randomForest(y ~ ., data = simulated, importance = TRUE,ntree = 1000)  
rfImp3 <- varImp(model3, scale = FALSE)  
rfImp3

## Overall  
## V1 5.22223587  
## V2 6.50590767  
## V3 0.54698260  
## V4 7.03347392  
## V5 2.14790635  
## V6 0.14432894  
## V7 0.00327321  
## V8 -0.02730587  
## V9 -0.02370549  
## V10 0.03018101  
## duplicate1 2.87139662  
## duplicate2 2.10851388

## (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 Stroblet al. (2007). Do these importances show the same pattern as the traditional random forest model?

library(party)  
  
bagCtrl <- cforest\_control(mtry = ncol(simulated) - 1)  
baggedTree <- cforest(y ~ ., data = simulated, controls = bagCtrl)

Yes, the conditional inference tress shows the same pattern. V1 important is similar to the duplicate1 and duplicate2, which are about 1/3 of the original important size of V1 which was 8.

Also, variables V6-V10 do not appear to be significant in the model which matches the traditional random forest model.

cfImp <- varImp(baggedTree)  
cfImp

## Overall  
## V1 6.9847360165  
## V2 7.6996953836  
## V3 -0.0155478547  
## V4 10.1204910672  
## V5 2.3475403640  
## V6 0.0498204134  
## V7 0.0003167029  
## V8 -0.0218238989  
## V9 -0.0518217251  
## V10 0.0277142284  
## duplicate1 0.8305148845  
## duplicate2 1.7028880233

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

### Boosted Tree Model

Lets first start off by creating a boosted tree model using the **train** function. When running a boosted tree model, we are able to tune the parameters **interaction depth, number of trees and shrinkage.**

The **interaction depth** is the maximum number of splits the tree can have or the depth of the trees.

The **number of trees** is how many trees will be fit or number of iterations.

The **shrinkage** is the learning rate.

set.seed(123)  
  
gbmTune <- train(simulated[,-11],simulated[,c('y')],  
 method = "gbm",  
 ***## Dont print tons of notes to screen***  
 verbose = FALSE)

Looking at the importance scores of each predictor from the boosted tree model we see some differences. In general V1-V5 are still more important than V6-V10. V4 is still the most important predictor too. The importance scores are much higher and close to 100 for this model while other models were much closer to 0.

*#causing issues have to fix*  
*#gbm\_importance <- caret::varImp(gbmTune)*  
*#gbm\_importance*

### 

### Cubist Model

Now let’s create a cubist model using the function **train** and method = **cubist**.

library(Cubist)  
  
cubistMod <- train(simulated[,-11],  
 simulated[,c('y')],  
 method = "cubist")

Again with the **cubist** model we are seeing variable V1-V5 as the most important predictors. V2 takes the top spot as most important predictor in place of V4 in the other models. Again we are seeing predictors V6-V10 having much lower importance scores that are close to 0.

varImp(cubistMod)

## cubist variable importance  
##   
## Overall  
## V1 100.0000  
## V2 90.0000  
## V4 69.2857  
## V3 58.5714  
## V5 45.7143  
## duplicate1 5.7143  
## V6 3.5714  
## duplicate2 0.7143  
## V9 0.0000  
## V8 0.0000  
## V10 0.0000  
## V7 0.0000

# Exercise 8.2

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

We will do the simulation here with 3 variables having different granularities. The response variable we will choose, would be a function of random selection and some noise.

**Below, a regression tree is fitted using rpart, and the variable importance is calculated using varImp**. –>

library(rpart)  
set.seed(123)  
  
x1 <- rnorm(300, 30, 1)  
x2 <- rnorm(300, 30, 2)  
x3 <- rnorm(300, 30, 3)  
  
z <- (.4 \* x1) + (.1 \* x3) + rnorm(300, 0, sqrt(1- (.16 + .04 + .01)))  
y <- (1.5 \* z) + 10  
  
granular\_simulated <- data.frame(x1 = x1, x2 = x2, x3 = x3, y=y)  
  
rpartfit <- rpart(y ~., data = granular\_simulated)  
  
varimp\_sim <- varImp(rpartfit)  
  
varimp\_sim

## Overall  
## x1 1.2580699  
## x2 0.8976173  
## x3 1.0106568

As we can see, the tree uses x1, and x3 mostly to split, x2 is not related to Y, so the x2 variance score is less. In this simulation, x1 has the smallest standard deviation, so it is the strongest predictor in this case.

# Exercise 8.3

In stochastic gradient boosting the bagging fraction and learning rate will govern the construction of the trees as they are guided by the gradient. Although the optimal values of these parameters should be obtained through the tuning process, it is helpful to understand how the magnitudes of these parameters affect 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?

This question is somewhat hard to answer. This may cause of 2 factor i.e bragging fraction and learning rate. In gradient boosting algorithm, the learner is designed to employ the optimal fitting the gradient in each stage. This greedy strategy might cause over-fitting, and reduce generalization power. Learning rate is used to avoid over-fitting by adding only partial of the predicted value to the previous accumulated predicted values. The higher value will increase the greediness, and focus on fewer variables.

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

Greedy models are less likely to select the optimal global model and are prone to over-fitting. Stochastic models reduce prediction variance. Therefore, the less greedy model on the left (with a 0.1 learning rate) that is also more random (due to only selecting 0.1 of the training set observations to propose the next tree in the expansion) would 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?

Interaction depth specifies the tree depth and node splits. Models with a lower learning rate see greater improvements from interaction depth than models with a higher learning rate. Tree depth and learning rates impact the number of predictors which impacts the model greediness which impacts the predictive ability of the model which impacts the RMSE, all proportionally. Therefore, interaction depth will increase the number of predictors and RMSE. Variable importance will be spread across more predictors.

## Exercise 8.7

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

We decide to work with few tree based models. They are Random Forest (RF), Stochastic Gradient Boosting (SGB), Rule-Based Cubist (CUBE), Classification and Regression Tree (CART), Conditional Inference Tree (CIT).

We are going to the same data for problem 6.3, and 7.5 that went through data imputation, pre-processing and data splitting. Training data - 80%, test data - 20%.

### Random Forest

RF is implemented in the train() function using the rf method which implements Random Forest classification and regression using the randomForest package.

set.seed(123)  
ctrl1 <- trainControl(method = "boot", number = 25)  
tg1 <- expand.grid(mtry=seq(2,38,by=3))  
rfModel <- train(x = train\_pred, y = train\_output, method = "rf",  
 metric = "Rsquared", tuneGrid = tg1, trControl = ctrl1)

### Stochastic Gradient Boosting (SGB)

SGB is implemented in the train() function using the gbm method fits generalized boosted regression models. The tuning parameters declared in the tuning grid to be evaluated are various variable interaction depths (interaction.depth), fitted trees (n.trees), learning rates (shrinkage), and a minimum of 10 observations in the trees terminal nodes (n.minobsinnode).

set.seed(123)  
ctrl2 <- trainControl(method = "boot", number = 25)  
tg2 <- expand.grid(interaction.depth=seq(1,6,by=1), n.trees=c(25,50,100,200),  
 shrinkage=c(0.01,0.05,0.1,0.2), n.minobsinnode=10)  
sgbModel <- train(x = train\_pred, y = train\_output, method = "gbm",  
 metric = "Rsquared", tuneGrid = tg2, trControl = ctrl2, verbose=F)

### 

### Rule-Based Cubist (CUBE)

CUBE is implemented in the train() function using the cubist method which fits a rule-based M5 model with additional corrections based on nearest neighbors in the training set. The parameters committees use for boosting interaction and neighbors use for number of instances.

set.seed(123)  
ctrl3 <- trainControl(method = "boot", number = 25)  
tg3 <- expand.grid(committees = c(1,5,10,20,50,100), neighbors = c(0,1,3,5,7))  
cubeModel <- train(x = train\_pred, y = train\_output, method = "cubist",  
 metric = "Rsquared", tuneGrid = tg3, trControl = ctrl3)

### Classification and Regression Tree

CART is implemented in the train() function using the rpart2 method which tunes the model over the maximum depth (of the tree).

set.seed(123)  
ctrl4 <- trainControl(method = "boot", number = 25)  
tg4 <- expand.grid(maxdepth= seq(1,10,by=1))  
cartModel <- train(x = train\_pred, y = train\_output, method = "rpart2",  
 metric = "Rsquared", tuneGrid = tg4, trControl = ctrl4)

### Conditional Inference Tree

CIT is implemented in the train() function using the ctree2 method where the number of randomly selected predictors to choose from at each split is set by mtry.

set.seed(123)  
ctrl5 <- trainControl(method = "cv")  
tg5 <- expand.grid(maxdepth= seq(1,10,by=1), mincriterion=1-c(0.01, 0.05, 0.1))  
citModel <- train(x = train\_pred, y = train\_output, method = "ctree2",  
 metric = "Rsquared", tuneGrid = tg5, trControl = ctrl5)

## 

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

### Training Set Resampling

metrics <- **function**(models) {  
 RMSE = min(models$results$RMSE)  
 Rsquared = max(models$results$Rsquared)  
 MAE = min(models$results$MAE)  
 return(cbind(RMSE, Rsquared, MAE)) }  
  
data.frame(rbind(metrics(rfModel), metrics(sgbModel),  
 metrics(cubeModel), metrics(cartModel), metrics(citModel)),  
 row.names = c("RF","SGB","CUBE","CART","CIT"))

## RMSE Rsquared MAE  
## RF 0.6604178 0.5515785 0.5042702  
## SGB 0.6569884 0.5369341 0.5075942  
## CUBE 0.5812892 0.6333222 0.4515568  
## CART 0.8347660 0.3057688 0.6549490  
## CIT 0.7669504 0.4020740 0.6064422

### Predict Test Set

fcastRf <- predict(rfModel, newdata = test\_pred)  
fcastSgb <- predict(sgbModel, newdata = test\_pred)  
fcastCube <- predict(cubeModel, newdata = test\_pred)  
fcastCart <- predict(cartModel, newdata = test\_pred)  
fcastCit <- predict(citModel, newdata = test\_pred)  
  
data.frame(rbind(  
 postResample(pred = fcastRf, obs = test\_output),  
 postResample(pred = fcastSgb, obs = test\_output),  
 postResample(pred = fcastCube, obs = test\_output),  
 postResample(pred = fcastCart, obs = test\_output),  
 postResample(pred = fcastCit, obs = test\_output)  
),  
 row.names = c("RF","SGB","CUBE","CART","CIT")  
)

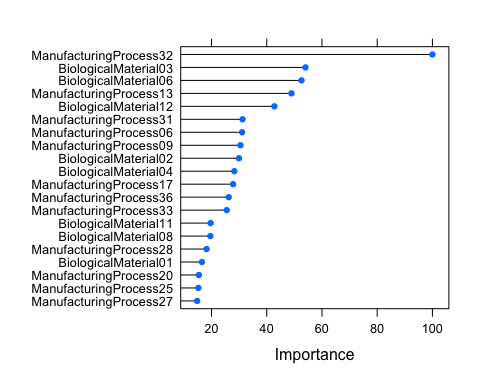
## RMSE Rsquared MAE  
## RF 0.6581917 0.7717416 0.5291642  
## SGB 0.6204107 0.7041805 0.4533152  
## CUBE 0.6711814 0.6763184 0.4197874  
## CART 0.7239807 0.5825921 0.5500855  
## CIT 0.7512187 0.5619932 0.5858097

The randomForest performed the best since it had the highest R^2 value and a lower RMSE & MAE values.

**(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?**

The optimal tree based model is the random Forest model. We can see the most important predictors by using varImp() function.

plot(varImp(rfModel), top=20)



Manufacturing variables dominate the list, out of 20 highest significant predictors 8 are Biological and rest all are manufacture variables.

All models also agree that ManufacturingProcess32 is the most important biological process. BiologicalMaterial03 and 06 were very important in all 3 models.

topPred\_nonlinear <- head(topPred\_knn[order(topPred\_knn$overall,decreasing = T),],10)$names  
topPred\_linear <- c('ManufacturingProcess32','BiologicalMaterial06','ManufacturingProcess13',  
 'BiologicalMaterial03','ManufacturingProcess17','BiologicalMaterial12',  
 'BiologicalMaterial02','ManufacturingProcess09','ManufacturingProcess31','ManufacturingProcess06')  
topTree <- c('ManufacturingProcess32','BiologicalMaterial03','BiologicalMaterial06',  
 'ManufacturingProcess13','BiologicalMaterial12','ManufacturingProcess31',  
 'ManufacturingProcess06','ManufacturingProcess09','BiologicalMaterial02','BiologicalMaterial04')  
topPreds\_in\_Both <- as.data.frame(cbind(topPred\_nonlinear,topPred\_linear,topTree))  
topPreds\_in\_Both

## topPred\_nonlinear topPred\_linear topTree  
## 1 ManufacturingProcess32 ManufacturingProcess32 ManufacturingProcess32  
## 2 BiologicalMaterial03 BiologicalMaterial06 BiologicalMaterial03  
## 3 BiologicalMaterial06 ManufacturingProcess13 BiologicalMaterial06  
## 4 ManufacturingProcess13 BiologicalMaterial03 ManufacturingProcess13  
## 5 BiologicalMaterial12 ManufacturingProcess17 BiologicalMaterial12  
## 6 ManufacturingProcess06 BiologicalMaterial12 ManufacturingProcess31  
## 7 ManufacturingProcess17 BiologicalMaterial02 ManufacturingProcess06  
## 8 ManufacturingProcess09 ManufacturingProcess09 ManufacturingProcess09  
## 9 ManufacturingProcess31 ManufacturingProcess31 BiologicalMaterial02  
## 10 BiologicalMaterial04 ManufacturingProcess06 BiologicalMaterial04

## (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?

getTree(rfModel$finalModel,labelVar=TRUE)

## left daughter right daughter split var split point status  
## 1 2 3 BiologicalMaterial12 -0.342432200 -3  
## 2 4 5 ManufacturingProcess32 0.006316353 -3  
## 3 6 7 ManufacturingProcess16 0.057432521 -3  
## 4 8 9 BiologicalMaterial06 -0.393851959 -3  
## 5 10 11 ManufacturingProcess26 0.129937047 -3  
## 6 12 13 ManufacturingProcess13 -1.042032278 -3  
## 7 14 15 ManufacturingProcess25 -0.010644284 -3  
## 8 16 17 BiologicalMaterial06 -1.440281441 -3  
## 9 0 0 <NA> 0.000000000 -1  
## 10 18 19 BiologicalMaterial02 -0.092636642 -3  
## 11 0 0 <NA> 0.000000000 -1  
## 12 20 21 BiologicalMaterial09 -0.060352325 -3  
## 13 22 23 ManufacturingProcess32 0.469515585 -3  
## 14 24 25 ManufacturingProcess43 -0.301769639 -3  
## 15 26 27 ManufacturingProcess39 -2.225937646 -3  
## 16 0 0 <NA> 0.000000000 -1  
## 17 28 29 ManufacturingProcess42 0.074036525 -3  
## 18 30 31 ManufacturingProcess23 -0.311054032 -3  
## 19 0 0 <NA> 0.000000000 -1  
## 20 0 0 <NA> 0.000000000 -1  
## 21 0 0 <NA> 0.000000000 -1  
## 22 32 33 BiologicalMaterial09 -0.096481608 -3  
## 23 34 35 BiologicalMaterial09 -0.072395419 -3  
## 24 0 0 <NA> 0.000000000 -1  
## 25 36 37 ManufacturingProcess14 -0.144314856 -3  
## 26 38 39 BiologicalMaterial01 1.104572970 -3  
## 27 40 41 ManufacturingProcess42 0.022532855 -3  
## 28 0 0 <NA> 0.000000000 -1  
## 29 42 43 ManufacturingProcess04 0.501809110 -3  
## 30 0 0 <NA> 0.000000000 -1  
## 31 0 0 <NA> 0.000000000 -1  
## 32 44 45 ManufacturingProcess18 0.038963399 -3  
## 33 46 47 ManufacturingProcess27 -0.050874412 -3  
## 34 48 49 BiologicalMaterial12 -0.329504687 -3  
## 35 50 51 ManufacturingProcess16 0.011938856 -3  
## 36 0 0 <NA> 0.000000000 -1  
## 37 52 53 ManufacturingProcess05 0.036258047 -3  
## 38 0 0 <NA> 0.000000000 -1  
## 39 0 0 <NA> 0.000000000 -1  
## 40 54 55 ManufacturingProcess10 0.483844645 -3  
## 41 56 57 ManufacturingProcess31 -0.330249724 -3  
## 42 58 59 ManufacturingProcess09 -0.966809548 -3  
## 43 60 61 BiologicalMaterial06 -1.137297394 -3  
## 44 0 0 <NA> 0.000000000 -1  
## 45 62 63 ManufacturingProcess30 -0.523934774 -3  
## 46 64 65 ManufacturingProcess17 -0.395567765 -3  
## 47 66 67 BiologicalMaterial05 -0.812647477 -3  
## 48 0 0 <NA> 0.000000000 -1  
## 49 0 0 <NA> 0.000000000 -1  
## 50 68 69 BiologicalMaterial04 0.228628851 -3  
## 51 0 0 <NA> 0.000000000 -1  
## 52 0 0 <NA> 0.000000000 -1  
## 53 0 0 <NA> 0.000000000 -1  
## 54 0 0 <NA> 0.000000000 -1  
## 55 0 0 <NA> 0.000000000 -1  
## 56 0 0 <NA> 0.000000000 -1  
## 57 70 71 ManufacturingProcess35 0.175930108 -3  
## 58 0 0 <NA> 0.000000000 -1  
## 59 72 73 ManufacturingProcess37 -0.142976460 -3  
## 60 0 0 <NA> 0.000000000 -1  
## 61 74 75 BiologicalMaterial02 -0.412372477 -3  
## 62 0 0 <NA> 0.000000000 -1  
## 63 0 0 <NA> 0.000000000 -1  
## 64 0 0 <NA> 0.000000000 -1  
## 65 0 0 <NA> 0.000000000 -1  
## 66 76 77 ManufacturingProcess22 0.778918315 -3  
## 67 78 79 ManufacturingProcess35 -0.702098958 -3  
## 68 0 0 <NA> 0.000000000 -1  
## 69 0 0 <NA> 0.000000000 -1  
## 70 80 81 ManufacturingProcess42 0.125540194 -3  
## 71 0 0 <NA> 0.000000000 -1  
## 72 0 0 <NA> 0.000000000 -1  
## 73 82 83 BiologicalMaterial11 -1.107398525 -3  
## 74 84 85 BiologicalMaterial12 -0.381214741 -3  
## 75 0 0 <NA> 0.000000000 -1  
## 76 0 0 <NA> 0.000000000 -1  
## 77 0 0 <NA> 0.000000000 -1  
## 78 0 0 <NA> 0.000000000 -1  
## 79 0 0 <NA> 0.000000000 -1  
## 80 0 0 <NA> 0.000000000 -1  
## 81 86 87 ManufacturingProcess03 0.644458244 -3  
## 82 0 0 <NA> 0.000000000 -1  
## 83 0 0 <NA> 0.000000000 -1  
## 84 88 89 ManufacturingProcess07 0.039917497 -3  
## 85 0 0 <NA> 0.000000000 -1  
## 86 0 0 <NA> 0.000000000 -1  
## 87 0 0 <NA> 0.000000000 -1  
## 88 90 91 ManufacturingProcess20 0.030199145 -3  
## 89 0 0 <NA> 0.000000000 -1  
## 90 0 0 <NA> 0.000000000 -1  
## 91 0 0 <NA> 0.000000000 -1  
## prediction  
## 1 0.08283972  
## 2 -0.44208102  
## 3 0.37446235  
## 4 -0.74192963  
## 5 0.62101861  
## 6 0.06169366  
## 7 0.76542321  
## 8 -0.86262735  
## 9 0.07881484  
## 10 1.04486157  
## 11 -0.50922931  
## 12 0.94540079  
## 13 -0.08216564  
## 14 1.22094973  
## 15 0.46173886  
## 16 -0.28636491  
## 17 -0.96198295  
## 18 1.09710751  
## 19 0.67914001  
## 20 1.54874461  
## 21 0.70406326  
## 22 -0.35049196  
## 23 0.41871014  
## 24 0.58378150  
## 25 1.51057166  
## 26 -0.30424463  
## 27 0.71706669  
## 28 -1.55717961  
## 29 -0.86675148  
## 30 1.34285692  
## 31 0.99880775  
## 32 -0.03527480  
## 33 -0.45556434  
## 34 0.93559661  
## 35 0.07411916  
## 36 1.11258779  
## 37 1.65981561  
## 38 -0.29069939  
## 39 -0.37197084  
## 40 1.15825461  
## 41 0.43631075  
## 42 -0.77635792  
## 43 -0.93777499  
## 44 0.37572657  
## 45 -0.10377503  
## 46 -0.02318084  
## 47 -0.72164650  
## 48 1.06924301  
## 49 0.90886733  
## 50 -0.07087944  
## 51 0.58161426  
## 52 1.75734136  
## 53 1.56228986  
## 54 1.34123149  
## 55 0.70081240  
## 56 1.83861281  
## 57 0.29608054  
## 58 -0.32320797  
## 59 -0.82167291  
## 60 -1.38244597  
## 61 -0.86366316  
## 62 -0.17150125  
## 63 -0.09022979  
## 64 -0.11596575  
## 65 0.06960408  
## 66 -0.61191515  
## 67 -0.84966641  
## 68 0.11023981  
## 69 -0.14332714  
## 70 0.43193933  
## 71 -0.24735461  
## 72 -0.97608868  
## 73 -0.78306897  
## 74 -0.89260073  
## 75 -0.54534995  
## 76 -0.54968443  
## 77 -0.76749194  
## 78 -1.07090538  
## 79 -0.80541862  
## 80 0.01542311  
## 81 0.57077807  
## 82 -0.67538429  
## 83 -0.84767978  
## 84 -0.86555950  
## 85 -1.16301303  
## 86 0.60762113  
## 87 0.38656276  
## 88 -0.84063625  
## 89 -0.96525249  
## 90 -0.82709101  
## 91 -0.85418149

# Recommender System

We worked on the recommender system in Python and R and are going to compare the approach in each program. Each analysis reviewed data from Market Basket.

# Market Basket Analysis in R

The Groceries data set contains a collection of receipts with each line representing 1 receipt and the items purchased. Each line is called a transaction and each column in a row represents an item purchased. The dataset is in GroceryDataSet.csv (comma separated file). Your assignment is to use R to mine the data for association rules. You should report support, confidence and lift and your top 10 rules by lift.

We’ll start with importing the libraries necessary for our analysis.

library(arules)  
library(pander)  
library(arulesViz)  
library(fpp2)  
library(RColorBrewer)

The Groceries Data Set contains a collection of receipts with each line representing 1 receipt and the items purchased. Each line is called a transaction and each column in a row represents an item. Load the csv file to R and lets get in to EDA.

The purpose of market basket analysis is retailers/businesses can analyze the data to discover what items customers are buying together. They use this information for making some profitable decisions.

Load the data from csv using read.transactions() from arules package.

grocery\_df <- read.transactions('https://raw.githubusercontent.com/SubhalaxmiRout002/DATA624/main/Week4/GroceryDataSet.csv', sep = ",", format = "basket")  
  
grocery\_df

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

## Exploratory Data Analysis

Use **summary()** to get the overview of data.

summary(grocery\_df)

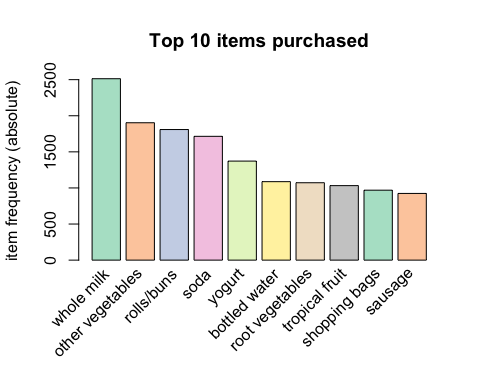
## transactions as itemMatrix in sparse format with  
## 9835 rows (elements/itemsets/transactions) and  
## 169 columns (items) and a density of 0.02609146   
##   
## most frequent items:  
## whole milk other vegetables rolls/buns soda   
## 2513 1903 1809 1715   
## yogurt (Other)   
## 1372 34055   
##   
## 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 gives the number of rows and columns present in the data. It shows the purchase of the most frequent items, in that the whole milk is on the top, 2nd highest is other vegetables, 3rd rank is roll/buns, etc.

Below the most frequently purchased items is the length distribution of the items. There are a total of 32 items, the first item occurs 2159 times, 2nd item occurs 1643 times, and so on. This means 2159 carts have 1st item (whole milk) and 1643 carts have 2nd item(other vegetables). If we add all these items’ frequencies it will sum up to the total number of rows that is 9835. If we look at the distribution, the mean is 4.4, which indicates on average there are 4 items per basket.

In itemFrequencyPlot(grocery\_df, topN=10,type=“absolute”) the first argument is the transaction object to be plotted, with our data that is **grocery\_df**. The parameter topN allows users to choose how many frequency items to plot. topN = 10 would plot the top 10 highest frequency items. The parameter type can be type=“absolute” or type=“relative”. If set equal to absolute, it will plot numeric frequencies of each item independently. If type equals relative, the function will plot how many times these items have appeared as compared to others.

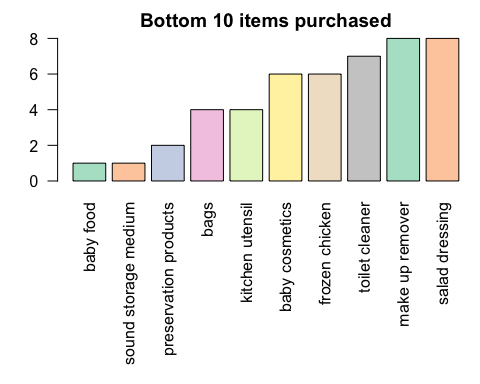
itemFrequencyPlot(grocery\_df, topN = 10, type="absolute", col=brewer.pal(8,'Pastel2'), main = 'Top 10 items purchased')



The above plot shows the same top 5 items as we get from summary():  
1 whole milk  
2 other vegetables  
3 rolls/buns  
4 soda  
5 yogurt

Now let’s view the least purchased items.

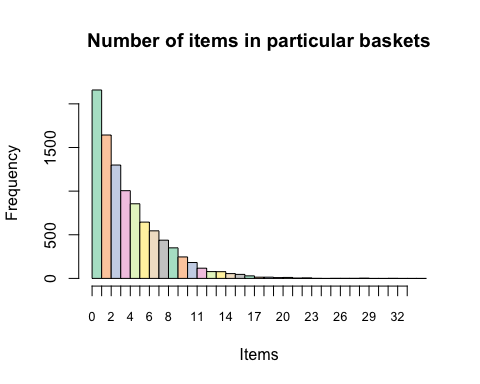
bottom\_10 <- head(sort(itemFrequency(grocery\_df, type="absolute"), decreasing=FALSE), n=10)  
par(mar=c(10.5,3,2, 0.3))  
barplot(bottom\_10, ylab = "Frequency", main = "Bottom 10 items purchased", col=brewer.pal(8,'Pastel2'), las = 2)



The above plot shows the Bottom 10 items purchased.  
1 baby food  
2 sound storage medium  
3 preservation products  
4 bags  
5 kitchen utensil

Items distribution in basket

hist(size(grocery\_df), breaks = 0:35, xaxt="n", ylim=c(0,2200),  
 main = "Number of items in particular baskets", xlab = "Items", col = brewer.pal(8,'Pastel2'))  
axis(1, at=seq(0,33,by=1), cex.axis=0.8)



We can see that the number of baskets decreases with the increase number of items.

## 

## Model Building

In this section, using the APRIORI algorithm, we make some rules and interpret how it works.

### Generating Rules

Next step is to mine the rules using the APRIORI algorithm. The function apriori() is from package arules.

*# Min Support as 0.001, confidence as 0.8.*  
association\_rules <- apriori(grocery\_df, parameter = list(supp=0.001, conf=0.8,maxlen=10), control=list(verbose=F))

The apriori() function will take data as the transaction object on which mining is to be applied. The parameters will allow the user to set min\_sup and min\_confidence. The default values for minimum support (supp) is 0.1, the minimum confidence (conf) is 0.8, and maximum of 10 items (maxlen).

summary(association\_rules)

## set of 410 rules  
##   
## rule length distribution (lhs + rhs):sizes  
## 3 4 5 6   
## 29 229 140 12   
##   
## Min. 1st Qu. Median Mean 3rd Qu. Max.   
## 3.000 4.000 4.000 4.329 5.000 6.000   
##   
## summary of quality measures:  
## support confidence coverage lift   
## Min. :0.001017 Min. :0.8000 Min. :0.001017 Min. : 3.131   
## 1st Qu.:0.001017 1st Qu.:0.8333 1st Qu.:0.001220 1st Qu.: 3.312   
## Median :0.001220 Median :0.8462 Median :0.001322 Median : 3.588   
## Mean :0.001247 Mean :0.8663 Mean :0.001449 Mean : 3.951   
## 3rd Qu.:0.001322 3rd Qu.:0.9091 3rd Qu.:0.001627 3rd Qu.: 4.341   
## Max. :0.003152 Max. :1.0000 Max. :0.003559 Max. :11.235   
## count   
## Min. :10.00   
## 1st Qu.:10.00   
## Median :12.00   
## Mean :12.27   
## 3rd Qu.:13.00   
## Max. :31.00   
##   
## mining info:  
## data ntransactions support confidence  
## grocery\_df 9835 0.001 0.8

The above summary() shows the following:  
\* Parameter Specification: min\_sup=0.001 and min\_confidence=0.8 values with 10 items as max of items in a rule.  
\* Total number of rules: The set of 410 rules  
\* Distribution of rule length: A length of 4 items has the most rules: 229 and length of 6 items have the lowest number of rules:12  
\* Summary of Quality measures: Min and max values for Support, Confidence and Lift.  
\* Information used for creating rules: The data, support, and confidence we provided to the algorithm.

Since there are 410 rules, let’s print only top 10:

inspect(association\_rules[1:10])

## lhs rhs support confidence coverage lift count  
## [1] {liquor,   
## red/blush wine} => {bottled beer} 0.001931876 0.9047619 0.002135231 11.235269 19  
## [2] {cereals,   
## curd} => {whole milk} 0.001016777 0.9090909 0.001118454 3.557863 10  
## [3] {cereals,   
## yogurt} => {whole milk} 0.001728521 0.8095238 0.002135231 3.168192 17  
## [4] {butter,   
## jam} => {whole milk} 0.001016777 0.8333333 0.001220132 3.261374 10  
## [5] {bottled beer,   
## soups} => {whole milk} 0.001118454 0.9166667 0.001220132 3.587512 11  
## [6] {house keeping products,   
## napkins} => {whole milk} 0.001321810 0.8125000 0.001626843 3.179840 13  
## [7] {house keeping products,   
## whipped/sour cream} => {whole milk} 0.001220132 0.9230769 0.001321810 3.612599 12  
## [8] {pastry,   
## sweet spreads} => {whole milk} 0.001016777 0.9090909 0.001118454 3.557863 10  
## [9] {curd,   
## turkey} => {other vegetables} 0.001220132 0.8000000 0.001525165 4.134524 12  
## [10] {rice,   
## sugar} => {whole milk} 0.001220132 1.0000000 0.001220132 3.913649 12

Above rule table shows lsh, rhs, support, confidence, coverage, lift, count. Let's discuss what each term means.  
\* **lhs**: items present in the basket  
\* **rhs**: item more likely bought with lhs  
\* **support**: Fraction of transactions that contain the item-set  
\* **confidence**: For a rule A=>B Confidence shows the percentage in which B is bought with A.  
\* **lift**: Lift gives the correlation between A and B in the rule A=>B. Correlation shows how one item-set A effects the item-set B.  
\* **count**: Frequency of occurrence of an item-set

Using the above output, we can make analysis such as:  
\* 100% of the customers who bought {rice,sugar} also bought {whole milk}.  
\* 92% of the customers who bought {house keeping products,whipped/sour cream} also bought {whole milk}.

### Removing redundant rules

We can remove rules that are subsets of larger rules. Use the code below to remove such rules:

*# get subset rules in vector*  
subset\_rules <- which(colSums(is.subset(association\_rules, association\_rules)) > 1)  
length(subset\_rules)

## [1] 91

### 

### Finding Rules related to single item

Sometimes, we want to work on a specific product. If want to find out what causes influence on the purchase of item X, we can use the appearance option in the apriori command. Appearance gives options to set LHS (IF part) and RHS (THEN part) of the rule.

For example, to find what customers buy before buying ‘whole milk’ run the following line of code:

whmilk\_association\_rules <- apriori(grocery\_df, parameter = list(supp=0.001, conf=0.8), appearance = list(default="lhs",rhs="whole milk"), control=list(verbose=F))  
  
rules\_whmilk\_byconf <- sort(whmilk\_association\_rules, by="confidence", decreasing=TRUE)  
  
inspect(head(rules\_whmilk\_byconf, n = 10))

## lhs rhs support confidence coverage lift count  
## [1] {rice,sugar} => {whole milk} 0.001220132 1 0.001220132 3.913649 12  
## [2] {canned fish,   
## hygiene articles} => {whole milk} 0.001118454 1 0.001118454 3.913649 11  
## [3] {butter,   
## rice,   
## root vegetables} => {whole milk} 0.001016777 1 0.001016777 3.913649 10  
## [4] {flour,   
## root vegetables,   
## whipped/sour cream} => {whole milk} 0.001728521 1 0.001728521 3.913649 17  
## [5] {butter,   
## domestic eggs,   
## soft cheese} => {whole milk} 0.001016777 1 0.001016777 3.913649 10  
## [6] {butter,   
## hygiene articles,   
## pip fruit} => {whole milk} 0.001016777 1 0.001016777 3.913649 10  
## [7] {hygiene articles,   
## root vegetables,   
## whipped/sour cream} => {whole milk} 0.001016777 1 0.001016777 3.913649 10  
## [8] {hygiene articles,   
## pip fruit,   
## root vegetables} => {whole milk} 0.001016777 1 0.001016777 3.913649 10  
## [9] {cream cheese,   
## domestic eggs,   
## sugar} => {whole milk} 0.001118454 1 0.001118454 3.913649 11  
## [10] {curd,   
## domestic eggs,   
## sugar} => {whole milk} 0.001016777 1 0.001016777 3.913649 10

Let's find what customers buy before buying ‘Other vegetables’ run the following line of code:

otherveg\_association\_rules <- apriori(grocery\_df, parameter = list(supp=0.001, conf=0.8),appearance = list(default="lhs",rhs="other vegetables"), control=list(verbose=F))  
  
rules\_otherveg\_byconf <- sort(otherveg\_association\_rules, by="confidence", decreasing=TRUE)  
  
inspect(head(rules\_otherveg\_byconf, n = 10))

## lhs rhs support confidence coverage lift count  
## [1] {citrus fruit,   
## root vegetables,   
## soft cheese} => {other vegetables} 0.001016777 1.0000000 0.001016777 5.168156 10  
## [2] {brown bread,   
## pip fruit,   
## whipped/sour cream} => {other vegetables} 0.001118454 1.0000000 0.001118454 5.168156 11  
## [3] {grapes,   
## tropical fruit,   
## whole milk,   
## yogurt} => {other vegetables} 0.001016777 1.0000000 0.001016777 5.168156 10  
## [4] {ham,   
## pip fruit,   
## tropical fruit,   
## yogurt} => {other vegetables} 0.001016777 1.0000000 0.001016777 5.168156 10  
## [5] {ham,   
## pip fruit,   
## tropical fruit,   
## whole milk} => {other vegetables} 0.001118454 1.0000000 0.001118454 5.168156 11  
## [6] {butter,   
## fruit/vegetable juice,   
## tropical fruit,   
## whipped/sour cream} => {other vegetables} 0.001016777 1.0000000 0.001016777 5.168156 10  
## [7] {newspapers,   
## rolls/buns,   
## soda,   
## whole milk} => {other vegetables} 0.001016777 1.0000000 0.001016777 5.168156 10  
## [8] {citrus fruit,   
## root vegetables,   
## tropical fruit,   
## whipped/sour cream} => {other vegetables} 0.001220132 1.0000000 0.001220132 5.168156 12  
## [9] {oil,   
## root vegetables,   
## whole milk,   
## yogurt} => {other vegetables} 0.001423488 0.9333333 0.001525165 4.823612 14  
## [10] {citrus fruit,   
## root vegetables,   
## tropical fruit,   
## whole milk,   
## yogurt} => {other vegetables} 0.001423488 0.9333333 0.001525165 4.823612 14

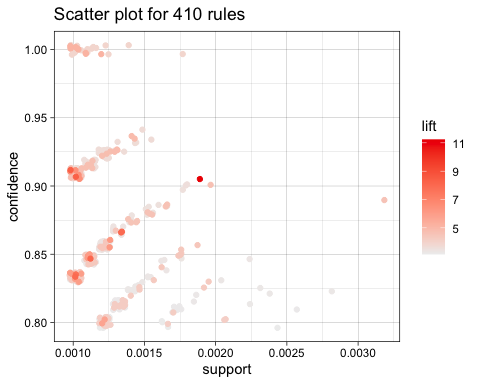
### 

### Visualizing Association Rules

A straight-forward visualization of association rules is to use a scatter plot using plot() of the arulesViz package. It uses Support and Confidence on the axes. In addition, third measure Lift is used by default to color (grey levels) of the points.

*# Filter rules with confidence greater than 0.4 or 40%*  
subRules<-association\_rules[quality(association\_rules)$confidence>0.4]  
*#Plot SubRules*  
plot(subRules)

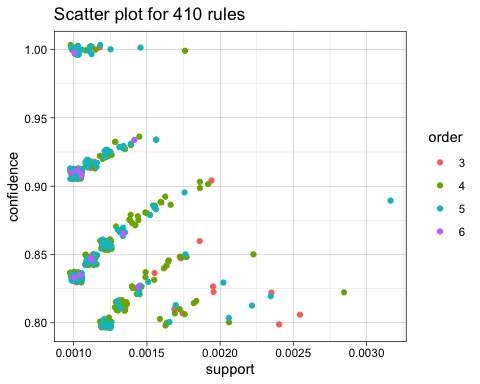
## To reduce overplotting, jitter is added! Use jitter = 0 to prevent jitter.



The above plot shows that rules with high lift have low support. There is one rule in the middle which has moderate support and high lift. And gray points show high support and less lift.

Another way to visualize individual Rule Representation. We need to set the method equal to “two-key plot”. The two-key plot uses support and confidence on x and y-axis respectively. It uses order for coloring. The order is the number of items in the rule.

plot(subRules,method="two-key plot")



As we discussed in EDA section, items 4 has more rules and items 6 has less rules. That we can see in the plot.

### 

### Graph-Based Visualizations

Graph plots are a great way to visualize rules but tend to become congested as the number of rules increases. So it is better to visualize less number of rules with graph-based visualizations. Here we will see top 10 rules from subRules which have the highest confidence.

All the interactive graph has a dropdown, can select respective rules and see the graph.

top10subRules <- head(subRules, n = 10, by = "confidence")

Below, plot an interactive graph: Here we use engine=htmlwidget parameter in plot.

set.seed(123)  
plot(top10subRules, method = "graph", engine = "htmlwidget")

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

The above graph showing more red color high lift, and high confidence. Rule 4 has big size and light red color, this means data suggest if some one buys {flour, root vegetables, whipped/soure cream} that persom may be buy {whole milk}. If we look at Rule 6 it has high confidence and high lift, this means there is high chance a person buys {citrus fruit, root vegetables, soft cheese} will buy {other vegetables}.

Below graph for whole milk in RHS.

set.seed(123)  
plot(head(whmilk\_association\_rules, n= 10), method = "graph", engine = "htmlwidget")

Look at Rule 6, and Rule 8 both rules have high lift and high confidence. Rule 2 and Rule 9 has little low confidence and lift. Below graph for Other vegetables in RHS.

set.seed(123)  
plot(head(otherveg\_association\_rules, n= 10), method = "graph", engine = "htmlwidget")

Above plot, Rule 5 and Rule 7 have higher confidence and higher lift than other rules. Considering Rule 7, if a person buys {hard cheese, oil} more likely to buy {other vegetables}.

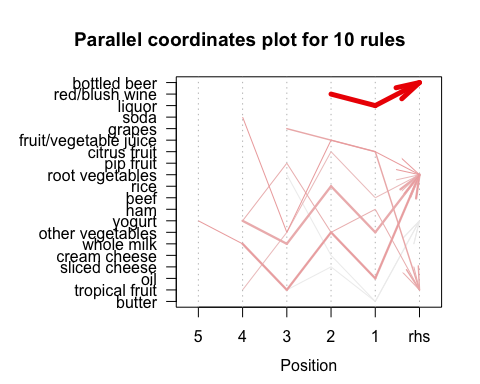
### 

### Individual Rule Representation

This representation is also called the **Parallel Coordinates Plot**. It is useful to visualize which products along with which items cause what kind of sales.

As mentioned above, the RHS is the Consequent or the item we propose the customer will buy; the positions are in the LHS where 2 is the most recent addition to our basket and 1 is the item we previously had.

set.seed(123)  
subRules2<-head(subRules, n=10, by="lift")  
plot(subRules2, method="paracoord")



Above plot, we see the wide dark line of the arrow, which means a person who buys {red/blush wine, liquor} more likely to buy bottled beer. We can see the line next to the dark line, which means who buys {grapes, fruits/vegetables, citrus fruits} less likely to buy {pip fruit}.

# 

# Conclusions

Since association rules are very useful in setting a strategy of the store, we found few conclusions out of the above analysis.

Concerning the least sold products which are baby food, sound storage medium, preservation products, kitchen utensil, bags, frozen chicken, baby cosmetics, toilet cleaner, and salad dressing, the business should consider withdrawing them from sale. Set a sales promotion or promotion by combining these least sold products with more popular ones (e.g. vegetables with whole milk, soda, yogurt, etc.).

Concerning the association rules, the grocery store can place other vegetables near fruits, root vegetables, cheese and place whole milk near rice, cereal, yogurt, and sour cream.

# 

# Market Basket Analysis in Python

In addition to R we wanted to run the market basket analysis in Python and compare how the languages differ in packages, functions and results. R & Python are the 2 most common languages used by data scientists and knowing how to run analyses in both languages is important.

import pandas as pd  
import numpy as np  
import matplotlib.pyplot as plt  
import seaborn as sns**;** sns.set()  
  
from mlxtend.preprocessing import TransactionEncoder  
from mlxtend.frequent\_patterns import apriori, association\_rules, fpgrowth

## 

## 

## Exploratory Data Analysis

The GroceryDataSet is a list of transactions make at a grocery store. The dataset comes from the University of California, Irvine, their machine learning dataset.

pd.set\_option('display.max\_columns', 40)  
  
df **=** pd.read\_csv('<https://raw.githubusercontent.com/SubhalaxmiRout002/>

DATA624/main/Week4/GroceryDataSet.csv', header**=**None).fillna(0)  
df.shape

## (9835, 32)

df.head()

## 0 1 2 \  
## 0 citrus fruit semi-finished bread margarine   
## 1 tropical fruit yogurt coffee   
## 2 whole milk 0 0   
## 3 pip fruit yogurt cream cheese   
## 4 other vegetables whole milk condensed milk   
##   
## 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 \  
## 0 ready soups 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0   
## 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0   
## 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0   
## 3 meat spreads 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0   
## 4 long life bakery product 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0   
##   
## 20 21 22 23 24 25 26 27 28 29 30 31   
## 0 0 0 0 0 0 0 0 0 0 0 0 0   
## 1 0 0 0 0 0 0 0 0 0 0 0 0   
## 2 0 0 0 0 0 0 0 0 0 0 0 0   
## 3 0 0 0 0 0 0 0 0 0 0 0 0   
## 4 0 0 0 0 0 0 0 0 0 0 0 0

df.ne(0).sum().sum() **/** (df.shape[0] **\*** df.shape[1])

## 0.13779550076258262

When loading the data, there were a lot of nan values which were filled with 0 to indicate there are no transactions past the last. The dataset contains 9835 market transactions which are mostly empty with only 13% of it’s cells filled.

## Data Preparation

To get a better understanding of the dataset, the data will be transformed so each row is a transaction and each column is an item where the values are True or False if the item was purchased or not.

all\_items **=** df.apply(**lambda** x: [i **for** i **in** x **if** i **!=** 0], axis**=**1).values.tolist()

mlxtend TransactionEncoder class is used to transform the data

trans\_encoder **=** TransactionEncoder()  
trans\_encoder\_matrix **=** trans\_encoder.fit(all\_items).transform(all\_items)  
tdf **=** pd.DataFrame(trans\_encoder\_matrix, columns**=**trans\_encoder.columns\_).astype(int)  
tdf.head()

## Instant food products UHT-milk abrasive cleaner artif. sweetener \  
## 0 0 0 0 0   
## 1 0 0 0 0   
## 2 0 0 0 0   
## 3 0 0 0 0   
## 4 0 0 0 0   
##   
## baby cosmetics baby food bags baking powder bathroom cleaner beef \  
## 0 0 0 0 0 0 0   
## 1 0 0 0 0 0 0   
## 2 0 0 0 0 0 0   
## 3 0 0 0 0 0 0   
## 4 0 0 0 0 0 0   
##   
## berries beverages bottled beer bottled water brandy brown bread \  
## 0 0 0 0 0 0 0   
## 1 0 0 0 0 0 0   
## 2 0 0 0 0 0 0   
## 3 0 0 0 0 0 0   
## 4 0 0 0 0 0 0   
##   
## butter butter milk cake bar candles ... specialty vegetables spices \  
## 0 0 0 0 0 ... 0 0   
## 1 0 0 0 0 ... 0 0   
## 2 0 0 0 0 ... 0 0   
## 3 0 0 0 0 ... 0 0   
## 4 0 0 0 0 ... 0 0   
##   
## spread cheese sugar sweet spreads syrup tea tidbits toilet cleaner \  
## 0 0 0 0 0 0 0 0   
## 1 0 0 0 0 0 0 0   
## 2 0 0 0 0 0 0 0   
## 3 0 0 0 0 0 0 0   
## 4 0 0 0 0 0 0 0   
##   
## tropical fruit turkey vinegar waffles whipped/sour cream whisky \  
## 0 0 0 0 0 0 0   
## 1 1 0 0 0 0 0   
## 2 0 0 0 0 0 0   
## 3 0 0 0 0 0 0   
## 4 0 0 0 0 0 0   
##   
## white bread white wine whole milk yogurt zwieback   
## 0 0 0 0 0 0   
## 1 0 0 0 1 0   
## 2 0 0 1 0 0   
## 3 0 0 0 1 0   
## 4 0 0 1 0 0   
##   
## [5 rows x 169 columns]

The transformation looks correct but we will verify the first row with the original dataset.

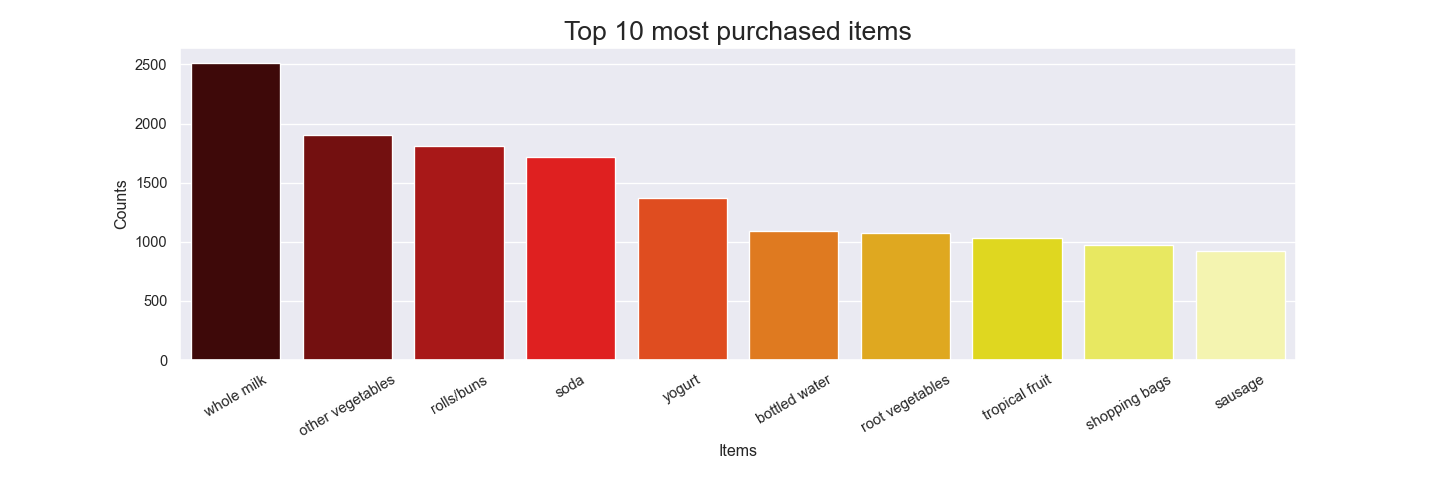
tdf[['citrus fruit', 'semi-finished bread' ,'margarine', 'ready soups']].head()

## citrus fruit semi-finished bread margarine ready soups  
## 0 1 1 1 1  
## 1 0 0 0 0  
## 2 0 0 0 0  
## 3 0 0 0 0  
## 4 0 0 0 0

Those values are correct.

With the data fully transformed we can explore some top 10 most purchased items.

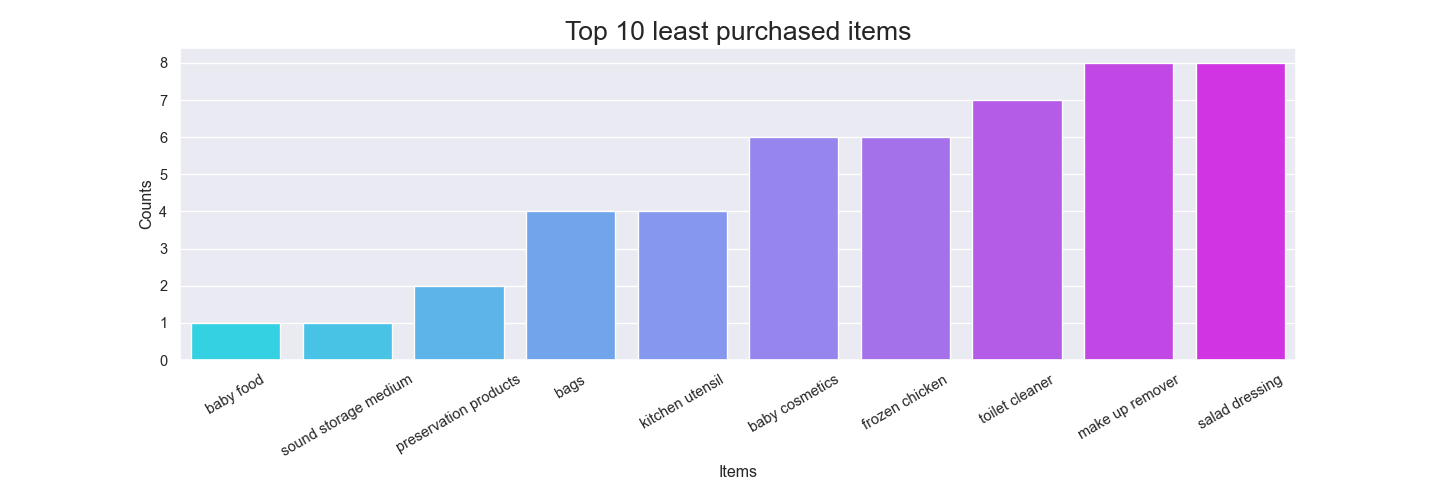
plt.figure(figsize**=**(15,5))**;**  
sns.barplot(x**=**tdf.sum().nlargest(10).index , y**=**tdf.sum().nlargest(10).values, palette **=** 'hot')**;**  
plt.xlabel('Items', size **=** 12)**;**  
plt.xticks(rotation**=**30)**;**  
plt.ylabel('Counts', size **=** 12)**;**  
plt.title('Top 10 most purchased items', size **=** 20)**;**  
plt.subplots\_adjust(bottom**=**0.25, top**=**0.9)**;**  
plt.show()**;**



This looks almost like my grocery transactions, milk and vegetables are definitely on top.

Now let’s review the least 10 purchased items.

plt.figure(figsize**=**(15,5))**;**  
sns.barplot(x**=**tdf.sum().nsmallest(10).index , y**=**tdf.sum().nsmallest(10).values, palette **=** 'cool')**;**  
plt.xlabel('Items', size **=** 12)**;**  
plt.xticks(rotation**=**30)**;**  
plt.ylabel('Counts', size **=** 12)**;**  
plt.title('Top 10 least purchased items', size **=** 20)**;**  
plt.subplots\_adjust(bottom**=**0.25, top**=**0.9)**;**  
plt.show()**;**



This grocery store makes most of its revenue from whole milk and vegetables and the least from baby food and sound storage medium.

tdf.sum(1).describe().iloc[1:]

## mean 4.409456  
## std 3.589385  
## min 1.000000  
## 25% 2.000000  
## 50% 3.000000  
## 75% 6.000000  
## max 32.000000  
## dtype: float64

We can see on average a customer purchases 4.4 items where the longest chain is 32 items long, that was a very active shopper. As expected the minimum purchase is 1 since there are no rows with all 0s.

## 

## Model building

Apriori is a popular algorithm for extracting frequent itemsets with applications in association rule learning. The apriori algorithm has been designed to operate on databases containing transactions, such as the one we are using.

frequent\_itemsets **=** apriori(tdf, min\_support**=**0.001, use\_colnames**=**True, max\_len**=**2).assign(length**=lambda** x: x['itemsets'].map(**lambda** x: len(x)))  
  
frequent\_itemsets.head()

## support itemsets length  
## 0 0.008033 (Instant food products) 1  
## 1 0.033452 (UHT-milk) 1  
## 2 0.003559 (abrasive cleaner) 1  
## 3 0.003254 (artif. sweetener) 1  
## 4 0.017692 (baking powder) 1

rules **=** association\_rules(frequent\_itemsets, metric**=**"lift", min\_threshold**=**1).sort\_values('lift', ascending**=**False)  
rules.head()

## antecedents consequents antecedent support \  
## 4144 (mustard) (mayonnaise) 0.011998   
## 4145 (mayonnaise) (mustard) 0.009151   
## 12 (hamburger meat) (Instant food products) 0.033249   
## 13 (Instant food products) (hamburger meat) 0.008033   
## 2656 (softener) (detergent) 0.005491   
##   
## consequent support support confidence lift leverage \  
## 4144 0.009151 0.001423 0.118644 12.965160 0.001314   
## 4145 0.011998 0.001423 0.155556 12.965160 0.001314   
## 12 0.008033 0.003050 0.091743 11.421438 0.002783   
## 13 0.033249 0.003050 0.379747 11.421438 0.002783   
## 2656 0.019217 0.001118 0.203704 10.600137 0.001013   
##   
## conviction   
## 4144 1.124233   
## 4145 1.170002   
## 12 1.092166   
## 13 1.558640   
## 2656 1.231681

The association\_rules returns 9 columns, but we are only interested in the first two and support, confidence and lift. Antecedents and Consequents are pairs of items that have associations and their strength is given by the lift/confidence.

* Support is the joint probability of the antecedents and consequents [0,1]
* Confidence is the conditional probability P(consequents|antecedents) [0,1]
* Lift is a combination of Support and Confidence, range [0, inf]

### Model exploration

Now that we have the rules we can take a look at different associations.

rules[rules['lift'].gt(1) **&** rules['confidence'].gt(0.6)]

## antecedents consequents antecedent support consequent support \  
## 3780 (honey) (whole milk) 0.001525 0.255516   
## 1596 (cereals) (whole milk) 0.005694 0.255516   
## 4962 (rice) (whole milk) 0.007626 0.255516   
##   
## support confidence lift leverage conviction   
## 3780 0.001118 0.733333 2.870009 0.000729 2.791815   
## 1596 0.003660 0.642857 2.515917 0.002206 2.084555   
## 4962 0.004677 0.613333 2.400371 0.002729 1.925390

Selecting rows with high lift and confidence score we can see a strong association between (honey, cereals and rice) with whole milk.

rules['support'].describe()

## count 5428.000000  
## mean 0.004008  
## std 0.005128  
## min 0.001017  
## 25% 0.001423  
## 50% 0.002339  
## 75% 0.004372  
## max 0.074835  
## Name: support, dtype: float64

We see that the support for all rules in our dataset is very low (i.e., the proportion of transactions that involve items from both baskets), which may be problematic, due to the fact that any results obtained from analysis may not be statistically significant.

rules.loc[rules['lift'].idxmax()]

## antecedents (mustard)  
## consequents (mayonnaise)  
## antecedent support 0.011998  
## consequent support 0.009151  
## support 0.001423  
## confidence 0.118644  
## lift 12.96516  
## leverage 0.001314  
## conviction 1.124233  
## Name: 4144, dtype: object

We see that item sets mustard and mayonnaise have the highest lift. So if we know that a customer has bought mustard they are ~13 times more likely that they will purchase mayonnaise as well. But as we’ve pointed out, due to the fact that the support is very low, we cannot really determine whether this is just a fluke or a real association.

rules.loc[rules['lift'].idxmin()]

## antecedents (specialty chocolate)  
## consequents (rolls/buns)  
## antecedent support 0.030402  
## consequent support 0.183935  
## support 0.005592  
## confidence 0.183946  
## lift 1.000063  
## leverage 0.0  
## conviction 1.000014  
## Name: 5010, dtype: object

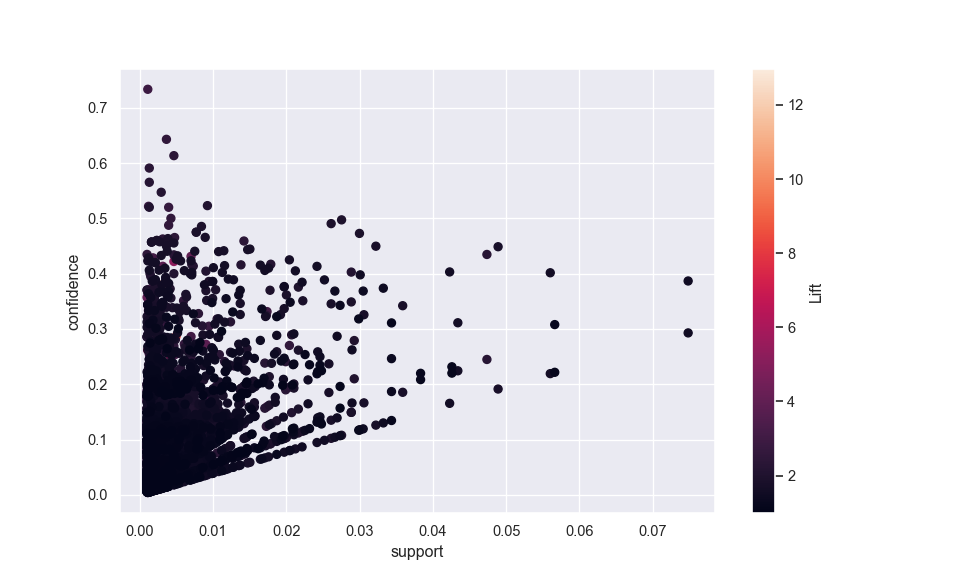
We see that item sets rolls/buns and specialty chocolate have the lowest lift. The low lift makes sense because I can’t think of a reason to have those two items together.

rules[rules['consequents'].map(**lambda** x: list(x)[0]).eq('whole milk')].query('lift > 2')

## antecedents consequents antecedent support \  
## 3780 (honey) (whole milk) 0.001525   
## 1596 (cereals) (whole milk) 0.005694   
## 4962 (rice) (whole milk) 0.007626   
## 2110 (cocoa drinks) (whole milk) 0.002237   
## 4930 (pudding powder) (whole milk) 0.002339   
## 3940 (jam) (whole milk) 0.005389   
## 222 (baking powder) (whole milk) 0.017692   
## 2265 (cooking chocolate) (whole milk) 0.002542   
##   
## consequent support support confidence lift leverage conviction   
## 3780 0.255516 0.001118 0.733333 2.870009 0.000729 2.791815   
## 1596 0.255516 0.003660 0.642857 2.515917 0.002206 2.084555   
## 4962 0.255516 0.004677 0.613333 2.400371 0.002729 1.925390   
## 2110 0.255516 0.001322 0.590909 2.312611 0.000750 1.819850   
## 4930 0.255516 0.001322 0.565217 2.212062 0.000724 1.712313   
## 3940 0.255516 0.002949 0.547170 2.141431 0.001572 1.644069   
## 222 0.255516 0.009253 0.522989 2.046793 0.004732 1.560725   
## 2265 0.255516 0.001322 0.520000 2.035097 0.000672 1.551008

Earlier we saw that whole milk was the most frequently bought item, now we look at it’s association. Items like cereals and honey are at the top probably because we eat cereal with milk and put honey in our tea. A lot of the items involve baking as well, like baking powder, pudding powder, cooking chocolate and cocoa drinks.

sup **=** rules['support'].values**;**  
conf **=** rules['confidence'].values**;**  
lift **=** rules['lift'].values**;**  
  
plt.figure(figsize**=**(10,6))**;**  
sc **=** plt.scatter(sup,conf,c**=**lift)**;**  
plt.colorbar(sc,label**=**'Lift')**;**  
plt.xlabel('support')**;**  
plt.ylabel('confidence')**;**  
plt.show()**;**



rules[rules['lift'].gt(10)]

## antecedents consequents antecedent support \  
## 4144 (mustard) (mayonnaise) 0.011998   
## 4145 (mayonnaise) (mustard) 0.009151   
## 12 (hamburger meat) (Instant food products) 0.033249   
## 13 (Instant food products) (hamburger meat) 0.008033   
## 2656 (softener) (detergent) 0.005491   
## 2657 (detergent) (softener) 0.019217   
## 3976 (liquor) (red/blush wine) 0.011083   
## 3977 (red/blush wine) (liquor) 0.019217   
##   
## consequent support support confidence lift leverage \  
## 4144 0.009151 0.001423 0.118644 12.965160 0.001314   
## 4145 0.011998 0.001423 0.155556 12.965160 0.001314   
## 12 0.008033 0.003050 0.091743 11.421438 0.002783   
## 13 0.033249 0.003050 0.379747 11.421438 0.002783   
## 2656 0.019217 0.001118 0.203704 10.600137 0.001013   
## 2657 0.005491 0.001118 0.058201 10.600137 0.001013   
## 3976 0.019217 0.002135 0.192661 10.025484 0.001922   
## 3977 0.011083 0.002135 0.111111 10.025484 0.001922   
##   
## conviction   
## 4144 1.124233   
## 4145 1.170002   
## 12 1.092166   
## 13 1.558640   
## 2656 1.231681   
## 2657 1.055968   
## 3976 1.214833   
## 3977 1.112532

Finally I wanted to look at the relationship with support and confidence and how they influence lift, we can see a slight regression where support and confidence both increase in a linear fashion. Some of the larger lift values share similar support/confidence with small values so they don’t apper in the plot. The table above shows the largest lift values.

# 

# Conclusion

Exploring the grocery data set has exposed a lot of secrets in the grocery business. By implementing an apriori() algorithm we found that certain items have strong association with other items making the pair a good target for coupons/adds. When it comes to item placement it would be a good idea to place items with high lift scores apart so customers will be exposed to other products along the way. I am certain many grocery stores already implement these approaches to maximize profit.

However the analysis done here shouldn’t be stretched too far since we saw the support values are too low. More data is needed to make strong connections between items. Furthermore the dataset was small making it more likely that what we found isn’t statistically significant.

# 

# Comparison of Market Basket Analysis in Python vs R

Here we are going to review the differences and similarities between each language in how they were able to accomplish the different steps in our analysis.

### Loading the Data

Beginning with loading the data, each language tackled the task differently. While there are many ways to load data in each language, in R we used the function **read.transactions() from the package arules**. This directly transformed the data from a csv into a transaction object.

In Python, we used the **read\_csv()** function along with **TransactionEncoder from mlxtend.preprocessing**. This technique was more manual but achieved the same result. Here is a view of the encoded data.

### Viewing Top & Bottom Sold Products

In order to view the top 10 and bottom 10 products that were sold, in R we used the **function itemFrequencyPlot() from the package arules**. In Python we used the **package seaborn and the function barplot()**. Again, there are likely many ways to do this in each language.

### Mining Data for Rules

Mining the Market Basket data in Python and R were very similar. They both used the **function apriori()**. In R, this function is apart of the **package arules** and in Python this function is used as part of the **mlxtend.frequent\_patterns package**. Python also used the function association\_rules from the same mlxtend.frequent\_patterns package.

### Visualizing Rules

In order to visualize the rules, R uses the **function plot() from within the package arulesViz**. In Python we used matplotlib.pyplot in order to view the relationship between support and confidence. Again, there are likely many ways to accomplish the same objective in both languages.

#### 

#### Resources

<https://www.youtube.com/watch?v=dYM-IaB61CA>  
<https://www.statology.org/ridge-regression-in-r/>  
<https://www.analyticsvidhya.com/blog/2016/01/ridge-lasso-regression-python-complete-tutorial/>  
[Data Camp market basket analysis] <https://www.datacamp.com/community/tutorials/market-basket-analysis-r>  
[R51 Association Rules, Market Basket Analysis in R Recommender Systems]<https://www.youtube.com/watch?v=iASqPvQpJ20>