# enable multi-core processing  
library(doParallel)  
cl <- makeCluster(detectCores())  
registerDoParallel(cl)

Creating Dummy Variables

?

# Zero- and Near Zero-Variance Predictors

In some situations, the data generating mechanism can create predictors that only have a single unique value (i.e. a "zero-variance predictor"). For many models (excluding tree-based models), this may cause the model to crash or the fit to be unstable.

imilarly, predictors might have only a handful of unique values that occur with very low frequencies. For example, in the drug resistance data, the nR11 descriptor (number of 11-membered rings) data have a few unique numeric values that are highly unbalanced:

**data**(mdrr)

**data.frame**(**table**(mdrrDescr$nR11))

Var1 Freq

1 0 501

2 1 4

3 2 23

The concern here that these predictors may become zero-variance predictors when the data are split into cross-validation/bootstrap sub-samples or that a few samples may have an undue influence on the model. These "near-zero-variance" predictors may need to be identified and eliminated prior to modeling.

To identify these types of predictors, the following two metrics can be calculated:

* the frequency of the most prevalent value over the second most frequent value (called the "frequency ratio''), which would be near one for well-behaved predictors and very large for highly-unbalanced data>
* the "percent of unique values'' is the number of unique values divided by the total number of samples (times 100) that approaches zero as the granularity of the data increases>

If the frequency ratio is less than a pre-specified threshold and the unique value percentage is less than a threshold, we might consider a predictor to be near zero-variance.

We would not want to falsely identify data that have low granularity but are evenly distributed, such as data from a discrete uniform distribution. Using both criteria should not falsely detect such predictors.

Looking at the MDRR data, the **nearZeroVar** function can be used to identify near zero-variance variables (the saveMetricsargument can be used to show the details and usually defaults to FALSE):

nzv <- **nearZeroVar**(mdrrDescr, saveMetrics= TRUE)

nzv[nzv$nzv,][1:10,]

freqRatio percentUnique zeroVar nzv

nTB 23.00000 0.3787879 FALSE TRUE

nBR 131.00000 0.3787879 FALSE TRUE

nI 527.00000 0.3787879 FALSE TRUE

nR03 527.00000 0.3787879 FALSE TRUE

nR08 527.00000 0.3787879 FALSE TRUE

nR11 21.78261 0.5681818 FALSE TRUE

nR12 57.66667 0.3787879 FALSE TRUE

D.Dr03 527.00000 0.3787879 FALSE TRUE

D.Dr07 123.50000 5.8712121 FALSE TRUE

D.Dr08 527.00000 0.3787879 FALSE TRUE

**dim**(mdrrDescr)

[1] 528 342

nzv <- **nearZeroVar**(mdrrDescr)

filteredDescr <- mdrrDescr[, -nzv]

**dim**(filteredDescr)

[1] 528 297

By default, **nearZeroVar** will return the positions of the variables that are flagged to be problematic.

# Identifying Correlated Predictors

While there are some models that thrive on correlated predictors (such as **pls**), other models may benefit from reducing the level of correlation between the predictors.

descrCor <- **cor**(filteredDescr)

highCorr <- **sum**(**abs**(descrCor[**upper.tri**(descrCor)]) > .999)

For the previous MDRR data, there are 65 descriptors that are almost perfectly correlated (|correlation| > 0.999), such as the total information index of atomic composition (IAC) and the total information content index (neighborhood symmetry of 0-order) (TIC0) (correlation = 1). The code chunk below shows the effect of removing descriptors with absolute correlations above 0.75.

descrCor <- **cor**(filteredDescr)

**summary**(descrCor[**upper.tri**(descrCor)])

Min. 1st Qu. Median Mean 3rd Qu. Max.

-0.99610 -0.05373 0.25010 0.26080 0.65530 1.00000

highlyCorDescr <- **findCorrelation**(descrCor, cutoff = .75)

filteredDescr <- filteredDescr[,-highlyCorDescr]

descrCor2 <- **cor**(filteredDescr)

**summary**(descrCor2[**upper.tri**(descrCor2)])

Min. 1st Qu. Median Mean 3rd Qu. Max.

-0.70730 -0.05378 0.04418 0.06692 0.18860 0.74460

# Linear Dependencies

The function **findLinearCombos** uses the QR decomposition of a matrix to enumerate sets of linear combinations (if they exist). For example, consider the following matrix that is could have been produced by a less-than-full-rank parameterizations of a two-way experimental layout:

ltfrDesign <- **matrix**(0, nrow=6, ncol=6)

ltfrDesign[,1] <- **c**(1, 1, 1, 1, 1, 1)

ltfrDesign[,2] <- **c**(1, 1, 1, 0, 0, 0)

ltfrDesign[,3] <- **c**(0, 0, 0, 1, 1, 1)

ltfrDesign[,4] <- **c**(1, 0, 0, 1, 0, 0)

ltfrDesign[,5] <- **c**(0, 1, 0, 0, 1, 0)

ltfrDesign[,6] <- **c**(0, 0, 1, 0, 0, 1)

Note that columns two and three add up to the first column. Similarly, columns four, five and six add up the first column.**findLinearCombos** will return a list that enumerates these dependencies. For each linear combination, it will incrementally remove columns from the matrix and test to see if the dependencies have been resolved. **findLinearCombos** will also return a vector of column positions can be removed to eliminate the linear dependencies:

comboInfo <- **findLinearCombos**(ltfrDesign)

comboInfo

$linearCombos

$linearCombos[[1]]

[1] 3 1 2

$linearCombos[[2]]

[1] 6 1 4 5

$remove

[1] 3 6

ltfrDesign[, -comboInfo$remove]

[,1] [,2] [,3] [,4]

[1,] 1 1 1 0

[2,] 1 1 0 1

[3,] 1 1 0 0

[4,] 1 0 1 0

[5,] 1 0 0 1

[6,] 1 0 0 0

These types of dependencies can arise when large numbers of binary chemical fingerprints are used to describe the structure of a molecule.

# Centering and Scaling

The **preProcess** class can be used for many operations on predictors, including centering and scaling. The function**preProcess** estimates the required parameters for each operation and **predict.preProcess** is used to apply them to specific data sets.

In the example below, the half of the MDRR data are used to estimate the location and scale of the predictors. The function**preProcess** doesn't actually pre-process the data. **predict.preProcess** is used to pre-process this and other data sets.

**set.seed**(96)

inTrain <- **sample**(**seq**(along = mdrrClass), **length**(mdrrClass)/2)

training <- filteredDescr[inTrain,]

test <- filteredDescr[-inTrain,]

trainMDRR <- mdrrClass[inTrain]

testMDRR <- mdrrClass[-inTrain]

preProcValues <- **preProcess**(training, method = **c**("center", "scale"))

trainTransformed <- **predict**(preProcValues, training)

testTransformed <- **predict**(preProcValues, test)

The **preProcess** option "ranges" scales the data to the interval [0, 1].

# Imputation

**preProcess** can be used to impute data sets based only on information in the training set. One method of doing this is with K-nearest neighbors. For an arbitrary sample, the K closest neighbors are found in the training set and the value for the predictor is imputed using these values (e.g. using the mean). Using this approach will automatically trigger **preProcess** to center and scale the data, regardless of what is in the method argument. Alternatively, bagged trees can also be used to impute. For each predictor in the data, a bagged tree is created using all of the other predictors in the training set. When a new sample has a missing predictor value, the bagged model is used to predict the value. While, in theory, this is a more powerful method of imputing, the computational costs are much higher than the nearest neighbor technique.