1 caret-preprocessing

1.1 Creating Dummy Variables

The function `dummyVars` can be used to generate a complete (less than full rank parameterized) set of dummy variables from one or more factors. The function takes a formula and a data set and outputs an object that can be used to create the dummy variables using the predict `method.` For example, the `etitanic` data set in the ***earth*** package includes two factors: pclass1 (with levels 1st, 2nd, 3rd) and *sex* (with levels female, male). The base `R` function `model.matrix` would generate the following variables:

```
library(earth)
data(etitanic)
head(model.matrix(survived ~ ., data = etitanic))
```

```
  (Intercept) pclass2nd pclass3rd sexmale     age sibsp parch
1           1         0         0       0 29.0000     0     0
2           1         0         0       1  0.9167     1     2
3           1         0         0       0  2.0000     1     2
4           1         0         0       1 30.0000     1     2
5           1         0         0       0 25.0000     1     2
6           1         0         0       1 48.0000     0     0
```

Using `dummyVars`:

```
dummies <- dummyVars(survived ˜ ., data = etitanic)
head(predict(dummies, newdata = etitanic))
```

```
  pclass.1st pclass.2nd pclass.3rd sex.female sex.male     age sibsp parch
1          1          0          0          1        0 29.0000     0     0
2          1          0          0          0        1  0.9167     1     2
3          1          0          0          1        0  2.0000     1     2
4          1          0          0          0        1 30.0000     1     2
5          1          0          0          1        0 25.0000     1     2
6          1          0          0          0        1 48.0000     0     0
```

Note there is no intercept and each factor has a dummy variable for each level, so this parameterization may not be useful for some model functions, such as `lm`.

1.2 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.

Similarly, 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:

1. 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¿

2. 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.

Looking at the MDRR data, the `nearZeroVar` function can be used to identify near zero-variance variables (the saveMetrics argument 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        23.00        0.3788   FALSE TRUE
nBR       131.00        0.3788   FALSE TRUE
nI        527.00        0.3788   FALSE TRUE
nR03      527.00        0.3788   FALSE TRUE
nR08      527.00        0.3788   FALSE TRUE
```

```
nR11       21.78        0.5682   FALSE TRUE
nR12       57.67        0.3788   FALSE TRUE
D.Dr03    527.00        0.3788   FALSE TRUE
D.Dr07    123.50        5.8712   FALSE TRUE
D.Dr08    527.00        0.3788   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.

1.3 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.

Given a correlation matrix, the `findCorrelation` function uses the following algorithm to flag predictors for removal:

```
descrCor <- cor(filteredDescr)
highCorr <- sum(abs(descrCor[upper.tri(descrCor)]) > 0.999)
```

For the previous MDRR data, there are 65 descriptors that are almost perfectly correlated, 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.
-0.9960 -0.0537  0.2500  0.2610  0.6550  1.0000
highlyCorDescr <- findCorrelation(descrCor, cutoff = 0.75)
filteredDescr <- filteredDescr[, -highlyCorDescr]
descrCor2 <- cor(filteredDescr)
summary(descrCor2[upper.tri(descrCor2)])
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
-0.7070 -0.0538  0.0442  0.0669  0.1890  0.7450
```

1.4 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.

1.5 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].

1.6 Imputation

- In statistics, imputation is the process of replacing missing data with substituted values.
- `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.

1.7 Transforming Predictors

- In some cases, there is a need to use principal component analysis (PCA) to transform the data to a smaller subspace where the new variable are uncorrelated with one another. The preProcess class can apply this transformation by including ”`pca`” in the method argument.
- Doing this will also force scaling of the predictors. Note that when PCA is requested, `predict.preProcess` changes the column names to PC1, PC2 and so on.
- Similarly, independent component analysis (ICA) can also be used to find new variables that are linear combinations of the original set such that the components are independent (as opposed to uncorrelated in PCA). The new variables will be labeled as IC1, IC2 and so on.