A Short Introduction to the caret Package

Max Kuhn
max.kuhn@pfizer.com

January 2, 2015

The caret package (short for **c**lassification **a**nd **re**gression **t**raining) contains functions to streamline the model training process for complex regression and classification problems. The package utilizes a number of R packages but tries not to load them all at package start-up[1]. The package "suggests" field includes 26 packages. caret loads packages as needed and assumes that they are installed. Install caret using

```
> install.packages("caret", dependencies = c("Depends", "Suggests"))
```

to ensure that all the needed packages are installed.

The **main help pages** for the package are at:

http://caret.r-forge.r-project.org/

Here, there are extended examples and a large amount of information that previously found in the package vignettes.

caret has several functions that attempt to streamline the model building and evaluation process, as well as feature selection and other techniques.

One of the primary tools in the package is the `train` function which can be used to

- evaluate, using resampling, the effect of model tuning parameters on performance
- choose the "optimal" model across these parameters
- estimate model performance from a training set

[1] By adding formal package dependencies, the package startup time can be greatly decreased

More formally:

```
Define sets of model parameter values to evaluate
for each parameter set do
    for each resampling iteration do
        Hold–out specific samples
        [Optional] Pre–process the data
        Fit the model on the remainder
        Predict the hold–out samples
    end
    Calculate the average performance across hold–out predictions
end
Determine the optimal parameter set
Fit the final model to all the training data using the optimal parameter set
```

There are options for customizing almost every step of this process (e.g. resampling technique, choosing the optimal parameters etc). To demonstrate this function, the Sonar data from the mlbench package will be used.

The Sonar data consist of 208 data points collected on 60 predictors. The goal is to predict the two classes (`M` for metal cylinder or `R` for rock).

First, we split the data into two groups: a training set and a test set. To do this, the `createDataPartition` function is used:

```
> library(caret)
> library(mlbench)
> data(Sonar)
> set.seed(107)
> inTrain <- createDataPartition(y = Sonar$Class,
+                                ## the outcome data are needed
+                                p = .75,
+                                ## The percentage of data in the
+                                ## training set
+                                list = FALSE)
>                               ## The format of the results
>
> ## The output is a set of integers for the rows of Sonar
> ## that belong in the training set.
> str(inTrain)
```

```
 int [1:157, 1] 98 99 100 101 102 104 107 108 111 112 ...
 - attr(*, "dimnames")=List of 2
  ..$ : NULL
  ..$ : chr "Resample1"
```

Bt default, `createDataPartition` does a stratified random split of the data. To partition the data:

```
> training <- Sonar[ inTrain,]
> testing  <- Sonar[-inTrain,]
> nrow(training)

[1] 157

> nrow(testing)

[1] 51
```

To tune a model using Algorithm **??**, the `train` function can be used. More details on this function can be found at:

http://caret.r-forge.r-project.org/training.html

Here, a partial least squares discriminant analysis (PLSDA) model will be tuned over the number of PLS components that should be retained. The most basic syntax to do this is:

```
> plsFit <- train(Class ~ .,
+                 data = training,
+                 method = "pls",
+                 ## Center and scale the predictors for the training
+                 ## set and all future samples.
+                 preProc = c("center", "scale"))
```

However, we would probably like to customize it in a few ways:

- expand the set of PLS models that the function evaluates. By default, the function will tune over three values of each tuning parameter.
- the type of resampling used. The simple bootstrap is used by default. We will have the function use three repeats of 10–fold cross–validation.
- the methods for measuring performance. If unspecified, overall accuracy and the Kappa statistic are computed. For regression models, root mean squared error and R^2 are computed. Here, the function will be altered to estimate the area under the ROC curve, the sensitivity and specificity

To change the candidate values of the tuning parameter, either of the `tuneLength` or `tuneGrid` arguments can be used. The `train` function can generate a candidate set of parameter values and the `tuneLength` argument controls how many are evaluated. In the case of PLS, the function uses a sequence of integers from 1 to `tuneLength`. If we want to evaluate all integers between 1 and 15, setting `tuneLength = 15` would achieve this. The `tuneGrid` argument is used when specific values are desired. A data frame is used where each row is a tuning parameter setting and each column is a tuning parameter. An example is used below to illustrate this.

The syntax for the model would then be:

```
> plsFit <- train(Class ~ .,
+                 data = training,
+                 method = "pls",
+                 tuneLength = 15,
+                 preProc = c("center", "scale"))
```

To modify the resampling method, a `trainControl` function is used. The option `method` controls the type of resampling and defaults to `"boot"`. Another method, `"repeatedcv"`, is used to specify repeated K–fold cross–validation (and the argument `repeats` controls the number of repetitions). K is controlled by the `number` argument and defaults to 10. The new syntax is then:

```
> ctrl <- trainControl(method = "repeatedcv",
+                      repeats = 3)
> plsFit <- train(Class ~ .,
+                 data = training,
+                 method = "pls",
+                 tuneLength = 15,
+                 trControl = ctrl,
+                 preProc = c("center", "scale"))
```

Finally, to choose different measures of performance, additional arguments are given to `trainControl`. The `summaryFunction` argument is used to pas in a function that takes the observed and predicted values and estimate some measure of performance. Two such functions are already included in the package: `defaultSummary` and `twoClassSummary`. The latter will compute measures specific to two–class problems, such as the area under the ROC curve, the sensitivity and specificity. Since the ROC curve is based on the predicted class probabilities (which are not computed automatically), another option is required. The `classProbs = TRUE` option is used to include these calculations.

Lastly, the function will pick the tuning parameters associated with the best results. Since we are using custom performance measures, the criterion that should be optimized must also be specified. In the call to `train`, we can use `metric = "ROC"` to do this.

The final model fit would then be:

```
> ctrl <- trainControl(method = "repeatedcv",
+                      repeats = 3,
+                      classProbs = TRUE,
+                      summaryFunction = twoClassSummary)
> plsFit <- train(Class ~ .,
+                 data = training,
+                 method = "pls",
+                 tuneLength = 15,
+                 trControl = ctrl,
+                 metric = "ROC",
+                 preProc = c("center", "scale"))


> plsFit

Partial Least Squares
```

```
157 samples
 60 predictor
  2 classes: 'M', 'R'

Pre-processing: centered, scaled
Resampling: Cross-Validated (10 fold, repeated 3 times)

Summary of sample sizes: 142, 141, 141, 141, 142, 142, ...

Resampling results across tuning parameters:

  ncomp  ROC        Sens       Spec       ROC SD     Sens SD    Spec SD
   1     0.8111772  0.7120370  0.7172619  0.1177583  0.1573858  0.1850478
   2     0.8713211  0.7782407  0.8130952  0.1066032  0.1630141  0.1539514
   3     0.8660962  0.7699074  0.8339286  0.1069895  0.1548614  0.1184648
   4     0.8660136  0.7740741  0.7714286  0.1028392  0.1704134  0.1448749
   5     0.8504216  0.7532407  0.7845238  0.1049264  0.1685590  0.1747078
   6     0.8352679  0.7574074  0.8035714  0.1090738  0.1520460  0.1671863
   7     0.8093419  0.7296296  0.7791667  0.1270386  0.1510365  0.1687237
   8     0.8080688  0.7259259  0.7744048  0.1403985  0.1740438  0.1766201
   9     0.8122933  0.7291667  0.7517857  0.1433122  0.1410158  0.1753622
  10     0.8182870  0.7296296  0.7654762  0.1378789  0.1474260  0.1781851
  11     0.8303241  0.7416667  0.7702381  0.1161916  0.1440423  0.1825742
  12     0.8203869  0.7458333  0.7738095  0.1339745  0.1532842  0.1802467
  13     0.8223958  0.7337963  0.7744048  0.1325278  0.1566296  0.1756838
  14     0.8165840  0.7375000  0.7833333  0.1398943  0.1630494  0.1695202
  15     0.8090939  0.7337963  0.7744048  0.1435415  0.1633646  0.1756838

ROC was used to select the optimal model using  the largest value.
The final value used for the model was ncomp = 2.
```

In this output the grid of results are the average resampled estimates of performance. The note at the bottom tells the user that PLS components were found to be optimal. Based on this value, a final PLS model is fit to the whole data set using this specification and this is the model that is used to predict future samples.

The package has several functions for visualizing the results. One method for doing this is the `plot` function for `train` objects. The command `plot(plsFit)` produced the results seen in Figure 1 and shows the relationship between the resampled performance values and the number of PLS components.

To predict new samples, `predict.train` can be used. For classification models, the default behavior is to calculated the predicted class. Using the option `type = "prob"` can be used to compute class probabilities from the model. For example:

```
> plsClasses <- predict(plsFit, newdata = testing)
> str(plsClasses)

 Factor w/ 2 levels "M","R": 2 1 1 2 1 2 2 2 2 2 ...
```

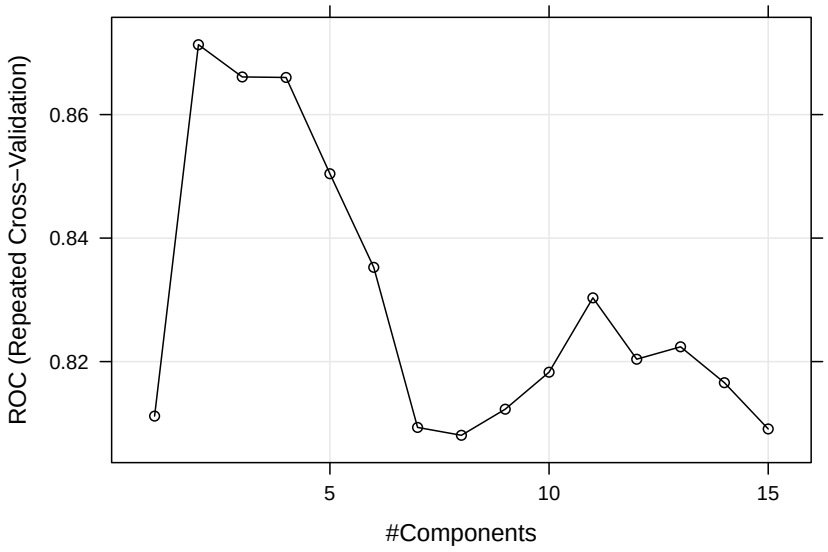

Figure 1: `plot(plsFit)` shows the relationship between the number of PLS components and the resampled estimate of the area under the ROC curve.

```
> plsProbs <- predict(plsFit, newdata = testing, type = "prob")
> head(plsProbs)
```

```
           M         R
4  0.3762529 0.6237471
5  0.5229047 0.4770953
8  0.5839468 0.4160532
16 0.3660142 0.6339858
20 0.7351013 0.2648987
25 0.2135788 0.7864212
```

caret contains a function to compute the confusion matrix and associated statistics for the model fit:

```
> confusionMatrix(data = plsClasses, testing$Class)
```

```
Confusion Matrix and Statistics

          Reference
Prediction  M  R
         M 20  7
         R  7 17

               Accuracy : 0.7255
```

```
               95% CI : (0.5826, 0.8411)
    No Information Rate : 0.5294
    P-Value [Acc > NIR] : 0.003347

                  Kappa : 0.4491
 Mcnemar's Test P-Value : 1.000000

            Sensitivity : 0.7407
            Specificity : 0.7083
         Pos Pred Value : 0.7407
         Neg Pred Value : 0.7083
             Prevalence : 0.5294
         Detection Rate : 0.3922
   Detection Prevalence : 0.5294
      Balanced Accuracy : 0.7245

       'Positive' Class : M
```

To fit an another model to the data, `train` can be invoked with minimal changes. Lists of models available can be found at:

http://caret.r-forge.r-project.org/modelList.html

http://caret.r-forge.r-project.org/bytag.html

For example, to fit a regularized discriminant model to these data, the following syntax can be used:

```
> ## To illustrate, a custom grid is used
> rdaGrid = data.frame(gamma = (0:4)/4, lambda = 3/4)
> set.seed(123)
> rdaFit <- train(Class ~ .,
+                 data = training,
+                 method = "rda",
+                 tuneGrid = rdaGrid,
+                 trControl = ctrl,
+                 metric = "ROC")
> rdaFit

Regularized Discriminant Analysis

157 samples
 60 predictor
  2 classes: 'M', 'R'

No pre-processing
Resampling: Cross-Validated (10 fold, repeated 3 times)

Summary of sample sizes: 142, 141, 141, 141, 142, 142, ...
```

```
Resampling results across tuning parameters:

  gamma  ROC        Sens       Spec       ROC SD      Sens SD    Spec SD
  0.00   0.8276868  0.7685185  0.7446429  0.11387360  0.1796630  0.1564605
  0.25   0.8775050  0.8013889  0.7803571  0.09803615  0.1434662  0.1631352
  0.50   0.8706928  0.8023148  0.7559524  0.10382645  0.1614643  0.1590523
  0.75   0.8598049  0.7972222  0.7422619  0.13100966  0.1490801  0.1651138
  1.00   0.7664269  0.6611111  0.6755952  0.15827666  0.1752499  0.2487249

Tuning parameter 'lambda' was held constant at a value of 0.75
ROC was used to select the optimal model using  the largest value.
The final values used for the model were gamma = 0.25 and lambda = 0.75.
```

```
> rdaClasses <- predict(rdaFit, newdata = testing)
> confusionMatrix(rdaClasses, testing$Class)

Confusion Matrix and Statistics

          Reference
Prediction  M  R
         M 22  5
         R  5 19

               Accuracy : 0.8039
                 95% CI : (0.6688, 0.9018)
    No Information Rate : 0.5294
    P-Value [Acc > NIR] : 4.341e-05

                  Kappa : 0.6065
 Mcnemar's Test P-Value : 1

            Sensitivity : 0.8148
            Specificity : 0.7917
         Pos Pred Value : 0.8148
         Neg Pred Value : 0.7917
             Prevalence : 0.5294
         Detection Rate : 0.4314
   Detection Prevalence : 0.5294
      Balanced Accuracy : 0.8032

       'Positive' Class : M
```

How do these models compare in terms of their resampling results? The `resamples` function can be used to collect, summarize and contrast the resampling results. Since the random number seeds were initialized to the same value prior to calling `train`, the same folds were used for each model. To assemble them:

```
> resamps <- resamples(list(pls = plsFit, rda = rdaFit))
> summary(resamps)
```

```
Call:
summary.resamples(object = resamps)

Models: pls, rda
Number of resamples: 30

ROC
      Min. 1st Qu. Median   Mean 3rd Qu. Max. NA's
pls 0.5397  0.8333 0.8672 0.8713  0.9509    1    0
rda 0.6964  0.8080 0.8967 0.8775  0.9524    1    0

Sens
      Min. 1st Qu. Median   Mean 3rd Qu. Max. NA's
pls 0.3333    0.75 0.7778 0.7782  0.8750    1    0
rda 0.4444    0.75 0.8750 0.8014  0.8854    1    0

Spec
      Min. 1st Qu. Median   Mean 3rd Qu. Max. NA's
pls 0.5000  0.7143 0.8571 0.8131  0.9688    1    0
rda 0.1429  0.7143 0.7500 0.7804  0.8571    1    0
```

There are several functions to visualize these results. For example, a Bland–Altman type plot can be created using `xyplot(resamps, what = "BlandAltman")` (see Figure 2). The results look similar. Since, for each resample, there are paired results a paired t–test can be used to assess whether there is a difference in the average resampled area under the ROC curve. The `diff.resamples` function can be used to compute this:

```
> diffs <- diff(resamps)
> summary(diffs)

Call:
summary.diff.resamples(object = diffs)

p-value adjustment: bonferroni
Upper diagonal: estimates of the difference
Lower diagonal: p-value for H0: difference = 0

ROC
    pls    rda
pls        -0.006184
rda 0.6785

Sens
    pls   rda
pls       -0.02315
rda 0.194

Spec
    pls    rda
```

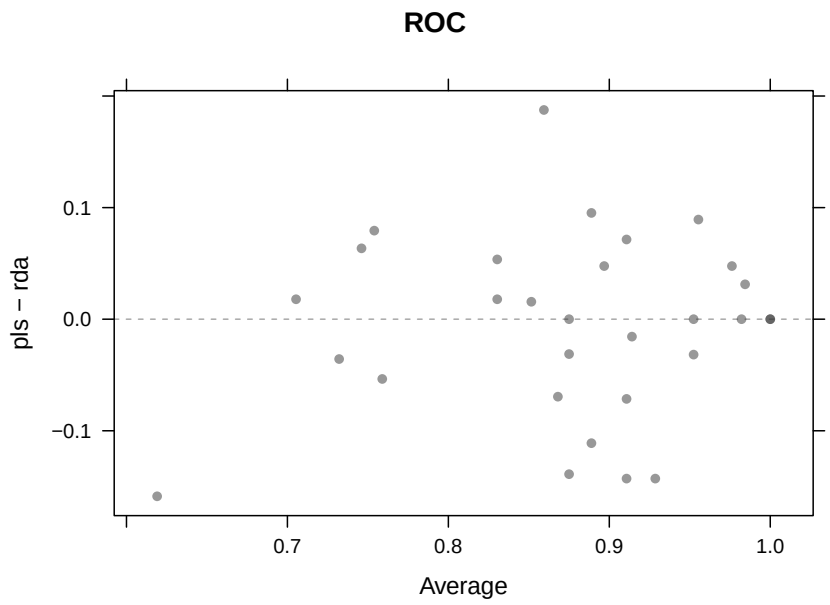

Figure 2: A Bland–Altman plot of the resampled ROC values produced using `xyplot(resamps, what = "BlandAltman")`.

```
pls        0.03274
rda 0.2299
```

Based on this analysis, the difference between the models is -0.006 ROC units (the RDA model is slightly higher) and the two–sided p–value for this difference is 0.67846.