The caret Package: A Unified Interface for Predictive Models

Max Kuhn

Pfizer Global R&D
Nonclinical Statistics
Groton, CT
max.kuhn@pfizer.com

February 26, 2014

Motivation

Theorem (No Free Lunch)

In the absence of any knowledge about the prediction problem, no model can be said to be uniformly better than any other

Given this, it makes sense to use a variety of different models to find one that best fits the data

R has many packages for predictive modeling (aka machine learning)(aka pattern recognition) ...

Model Function Consistency

Since there are many modeling packages written by different people, there are some inconsistencies in how models are specified and predictions are made.

For example, many models have only one method of specifying the model (e.g. formula method only)

The table below shows the syntax to get probability estimates from several classification models:

obj Class	**Package**	`predict` **Function Syntax**
`lda`	`MASS`	`predict(obj)` (no options needed)
`glm`	`stats`	`predict(obj, type = "response")`
`gbm`	`gbm`	`predict(obj, type = "response", n.trees)`
`mda`	`mda`	`predict(obj, type = "posterior")`
`rpart`	`rpart`	`predict(obj, type = "prob")`
`Weka`	`RWeka`	`predict(obj, type = "probability")`
`LogitBoost`	`caTools`	`predict(obj, type = "raw", nIter)`

The caret Package

The caret package was developed to:

- create a unified interface for modeling and prediction (currently 147 different models)
- streamline model tuning using resampling
- provide a variety of "helper" functions and classes for day–to–day model building tasks
- increase computational efficiency using parallel processing

First commits within Pfizer: 6/2005

First version on CRAN: 10/2007

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

JSS Paper: www.jstatsoft.org/v28/i05/paper

Book: *Applied Predictive Modeling* (AppliedPredictiveModeling.com)

Example Data

SGI has several data sets on their homepage for `MLC++` software at http://www.sgi.com/tech/mlc/.

One data set can be used to predict telecom customer churn based on information about their account.

Let's read in the data first:

```
> library(C50)
> data(churn)
```

Example Data

```
> str(churnTrain)

'data.frame': 3333 obs. of  20 variables:
 $ state                         : Factor w/ 51 levels "AK","AL","AR",..: 17 36 32 3
 $ account_length                : int  128 107 137 84 75 118 121 147 117 141 ...
 $ area_code                     : Factor w/ 3 levels "area_code_408",..: 2 2 2 1 2 3
 $ international_plan            : Factor w/ 2 levels "no","yes": 1 1 1 2 2 2 1 2 1 1
 $ voice_mail_plan               : Factor w/ 2 levels "no","yes": 2 2 1 1 1 1 2 1 1 1
 $ number_vmail_messages         : int  25 26 0 0 0 0 24 0 0 37 ...
 $ total_day_minutes             : num  265 162 243 299 167 ...
 $ total_day_calls               : int  110 123 114 71 113 98 88 79 97 84 ...
 $ total_day_charge              : num  45.1 27.5 41.4 50.9 28.3 ...
 $ total_eve_minutes             : num  197.4 195.5 121.2 61.9 148.3 ...
 $ total_eve_calls               : int  99 103 110 88 122 101 108 94 80 111 ...
 $ total_eve_charge              : num  16.78 16.62 10.3 5.26 12.61 ...
 $ total_night_minutes           : num  245 254 163 197 187 ...
 $ total_night_calls             : int  91 103 104 89 121 118 118 96 90 97 ...
 $ total_night_charge            : num  11.01 11.45 7.32 8.86 8.41 ...
 $ total_intl_minutes            : num  10 13.7 12.2 6.6 10.1 6.3 7.5 7.1 8.7 11.2 .
 $ total_intl_calls              : int  3 3 5 7 3 6 7 6 4 5 ...
 $ total_intl_charge             : num  2.7 3.7 3.29 1.78 2.73 1.7 2.03 1.92 2.35 3.0
 $ number_customer_service_calls: int  1 1 0 2 3 0 3 0 1 0 ...
 $ churn                         : Factor w/ 2 levels "yes","no": 2 2 2 2 2 2 2 2 2 1
```

Example Data

Here is a vector of predictor names that we might need later.

```
> predictors <- names(churnTrain)[names(churnTrain) != "churn"]
```

Note that the class's leading level is `"yes"`. Many of the functions used here model the probability of the first factor level.

In other words, we want to predict churn, not retention.

Data Splitting

A simple, stratified random split is used here:

```
> allData <- rbind(churnTrain, churnTest)
>
> set.seed(1)
> inTrainingSet <- createDataPartition(allData$churn,
+                                      p = .75, list = FALSE)
> churnTrain <- allData[ inTrainingSet,]
> churnTest  <- allData[-inTrainingSet,]
>
> ## For this presentation, we will stick with the original split
```

Other functions: `createFolds`, `createMultiFolds`, `createResamples`

Data Pre–Processing Methods

`preProcess` calculates values that can be used to apply to any data set (e.g. training, set, unknowns).

Current methods: centering, scaling, spatial sign transformation, PCA or ICA "signal extraction", imputation, Box–Cox transformations and others.

```
> numerics <- c("account_length", "total_day_calls", "total_night_calls")
> ## Determine means and sd's
> procValues <- preProcess(churnTrain[,numerics],
+                          method = c("center", "scale", "YeoJohnson"))
> ## Use the predict methods to do the adjustments
> trainScaled <- predict(procValues, churnTrain[,numerics])
> testScaled  <- predict(procValues, churnTest[,numerics])
```

Data Pre–Processing Methods

```
> procValues


Call:
preProcess.default(x = churnTrain[, numerics], method =
 c("center", "scale", "YeoJohnson"))

Created from 3333 samples and 3 variables
Pre-processing: centered, scaled, Yeo-Johnson transformation

Lambda estimates for Yeo-Johnson transformation:
0.89, 1.17, 0.93
```

`preProcess` can also be called within other functions for each resampling iteration (more in a bit).

Boosted Trees (Original " adaBoost" Algorithm)

A method to "boost" weak learning algorithms (e.g. single trees) into strong learning algorithms.

Boosted trees try to improve the model fit over different trees by considering past fits (not unlike iteratively reweighted least squares)

The basic tree boosting algorithm:

Initialize equal weights per sample;
for $j = 1 \ldots M$ *iterations* **do**
  Fit a classification tree using sample weights (denote the model equation as $f_j(x)$);
  forall the *misclassified samples* **do**
    increase sample weight
  end
  Save a "stage–weight" (β_j) based on the performance of the current model;
end

Boosted Trees (Original " adaBoost" Algorithm)

In this formulation, the categorical response y_i is coded as either $\{-1, 1\}$ and the model $f_j(x)$ produces values of $\{-1, 1\}$.

The final prediction is obtained by first predicting using all M trees, then weighting each prediction

$$f(x) = \frac{1}{M} \sum_{j=1}^{M} \beta_j f_j(x)$$

where f_j is the j^{th} tree fit and β_j is the stage weight for that tree.

The final class is determined by the sign of the model prediction.

In English: the final prediction is a weighted average of each tree's prediction. The weights are based on quality of each tree.

Boosted Trees Parameters

Most implementations of boosting have three tuning parameters:

- number of iterations (i.e. trees)
- complexity of the tree (i.e. number of splits)
- learning rate (aka. "shrinkage"): how quickly the algorithm adapts

Boosting functions for trees in R: `gbm` in the `gbm` package, `ada` in `ada` and `blackboost` in `mboost`.

Using the gbm Package

The gbm function in the gbm package can be used to fit the model, then predict.gbm and other functions are used to predict and evaluate the model.

```
> library(gbm)
> # The gbm function does not accept factor response values so we
> # will make a copy and modify the outcome variable
> forGBM <- churnTrain
> forGBM$churn <- ifelse(forGBM$churn == "yes", 1, 0)
>
> gbmFit <- gbm(formula = churn ~ .,          # Use all predictors
+               distribution = "bernoulli", # For classification
+               data = forGBM,
+               n.trees = 2000,             # 2000 boosting iterations
+               interaction.depth = 7,      # How many splits in each tree
+               shrinkage = 0.01,           # learning rate
+               verbose = FALSE)            # Do not print the details
```

Tuning the gbm Model

This code assumes that we know appropriate values of the three tuning parameters.

As previously discussed, one approach for model tuning is resampling.

We can fit models with different values of the tuning parameters to many resampled versions of the training set and estimate the performance based on the hold–out samples.

From this, a profile of performance can be obtained across different gbm models. An optimal value can be chosen from this profile.

Tuning the gbm Model

foreach *resampled data set* **do**
  Hold–out samples ;
  foreach *combination of tree depth, learning rate and number of trees* **do**
    Fit the model on the resampled data set;
    Predict the hold–outs and save results;
  end
  Calculate the average AUC ROC across the hold–out sets of predictions
end
Determine tuning parameters based on the highest resampled ROC AUC;

Model Tuning using train

In a basic call to train, we specify the predictor set, the outcome data and the modeling technique (eg. boosting via the gbm package):

```
> gbmTune <- train(x = churnTrain[,predictors],
+                  y= churnTrain$churn,
+                  method = "gbm")
>
> # or, using the formula interface
> gbmTune <- train(churn ~ ., data = churnTrain, method = "gbm")
```

Note that train uses the original factor input for classification. For binary outcomes, the function models the probability of the first factor level (churn for these data).

A numeric object would indicate we are doing regression.

One problem is that gbm spits out *a lot* of information during model fitting.

Passing Model Parameters Through `train`

We can pass options through `train` to `gbm` using the three dots.

Let's omit the function's logging using the `gbm` option `verbose = FALSE`.

```
gbmTune <- train(churn ~ ., data = churnTrain,
                 method = "gbm",
                 verbose = FALSE)
```

Changing the Resampling Technique

By default, `train` uses the bootstrap for resampling. We'll switch to 5 repeats of 10–fold cross–validation instead.

We can then change the type of resampling via the control function.

```
ctrl <- trainControl(method = "repeatedcv",
                     repeats = 5)

gbmTune <- train(churn ~ ., data = churnTrain,
                 method = "gbm",
                 verbose = FALSE,
                 trControl = ctrl)
```

Using Different Performance Metrics

At this point `train` will

1. fit a sequence of different `gbm` models,
2. estimate performance using resampling
3. pick the `gbm` model with the best performance

The default metrics for classification problems are accuracy and Cohen's Kappa.

Suppose we wanted to estimate sensitivity, specificity and the area under the ROC curve (and pick the model with the largest AUC).

We need to tell `train` to produce class probabilities, estimate these statistics and to rank models by the ROC AUC.

Using Different Performance Metrics

The `twoClassSummary` function is defined in `caret` and calculates the sensitivity, specificity and ROC AUC. Other custom functions can be used (see `?train`).

```
ctrl <- trainControl(method = "repeatedcv", repeats = 5,
                     classProbs = TRUE,
                     summaryFunction = twoClassSummary)

gbmTune <- train(churn ~ ., data = churnTrain,
                 method = "gbm",
                 metric = "ROC",
                 verbose = FALSE,
                 trControl = ctrl)
```

Expanding the Search Grid

By default, `train` uses a minimal search grid: 3 values for each tuning parameter.

We might also want to expand the scope of the possible `gbm` models to test.

Let's look at tree depths from 1 to 7, boosting iterations from 100 to 1,000 and two different learning rates.

We can create a data frame with these parameters to pass to `train`.

The column names should be the parameter name proceeded by a dot.

See `?train for the tunable parameters for each model type.`

Expanding the Search Grid

```
ctrl <- trainControl(method = "repeatedcv", repeats = 5,
                     classProbs = TRUE,
                     summaryFunction = twoClassSummary)

grid <- expand.grid(interaction.depth = seq(1, 7, by = 2),
                    n.trees = seq(100, 1000, by = 50),
                    shrinkage = c(0.01, 0.1))

gbmTune <- train(churn ~ ., data = churnTrain,
                 method = "gbm",
                 metric = "ROC",
                 tuneGrid = grid,
                 verbose = FALSE,
                 trControl = ctrl)
```

Runnig the Model

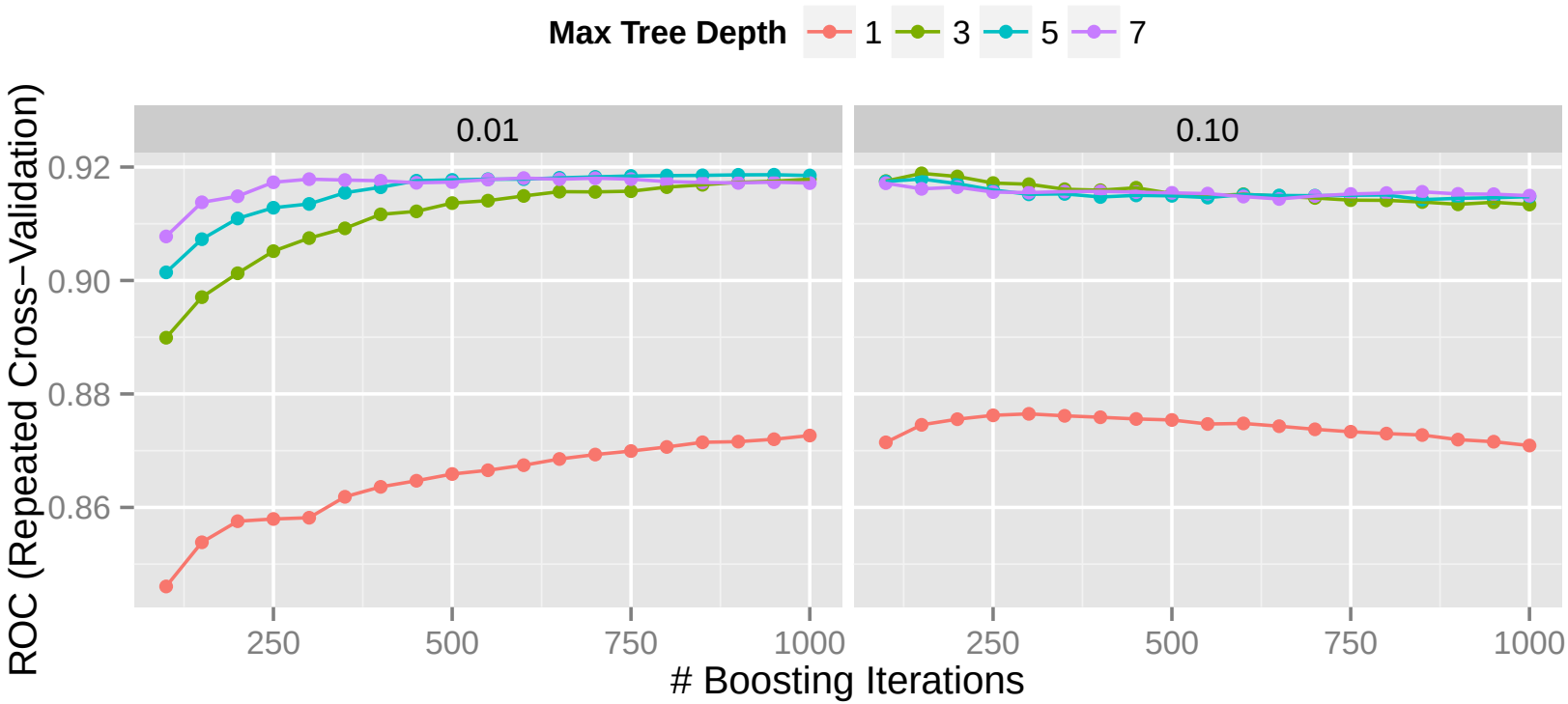
```
> grid <- expand.grid(interaction.depth = seq(1, 7, by = 2),
+                     n.trees = seq(100, 1000, by = 50),
+                     shrinkage = c(0.01, 0.1))
>
> ctrl <- trainControl(method = "repeatedcv", repeats = 5,
+                      summaryFunction = twoClassSummary,
+                      classProbs = TRUE)
>
> set.seed(1)
> gbmTune <- train(churn ~ ., data = churnTrain,
+                  method = "gbm",
+                  metric = "ROC",
+                  tuneGrid = grid,
+                  verbose = FALSE,
+                  trControl = ctrl)
```

Model Tuning

- `train` uses as many "tricks" as possible to reduce the number of models fits (e.g. using sub–models). Here, it uses the `kernlab` function `sigest` to analytically estimate the RBF scale parameter.
- Currently, there are options for 147 models (see `?train` for a list)
- Allows user–defined search grid, performance metrics and selection rules
- Easily integrates with any parallel processing framework that can emulate `lapply`
- Formula and non–formula interfaces
- Methods: `predict`, `print`, `plot`, `ggplot`, `varImp`, `resamples`, `xyplot`, `densityplot`, `histogram`, `stripplot`, ...

Plotting the Results

```
> ggplot(gbmTune) + theme(legend.position = "top")
```



Prediction and Performance Assessment

The `predict` method can be used to get results for other data sets:

```
> gbmPred <- predict(gbmTune, churnTest)
> str(gbmPred)

 Factor w/ 2 levels "yes","no": 2 2 2 2 2 2 2 2 2 2 ...
```

```
> gbmProbs <- predict(gbmTune, churnTest, type = "prob")
> str(gbmProbs)

'data.frame': 1667 obs. of  2 variables:
 $ yes: num  0.0316 0.092 0.3086 0.0307 0.022 ...
 $ no : num  0.968 0.908 0.691 0.969 0.978 ...
```

Predction and Performance Assessment

```
> confusionMatrix(gbmPred, churnTest$churn)

Confusion Matrix and Statistics

          Reference
Prediction  yes   no
       yes  151    8
       no    73 1435

               Accuracy : 0.951
                 95% CI : (0.94, 0.961)
    No Information Rate : 0.866
    P-Value [Acc > NIR] : < 2e-16

                  Kappa : 0.762
 Mcnemar's Test P-Value : 1.15e-12

            Sensitivity : 0.6741
            Specificity : 0.9945
         Pos Pred Value : 0.9497
         Neg Pred Value : 0.9516
             Prevalence : 0.1344
         Detection Rate : 0.0906
   Detection Prevalence : 0.0954
```

Test Set ROC curve via the pROC Package

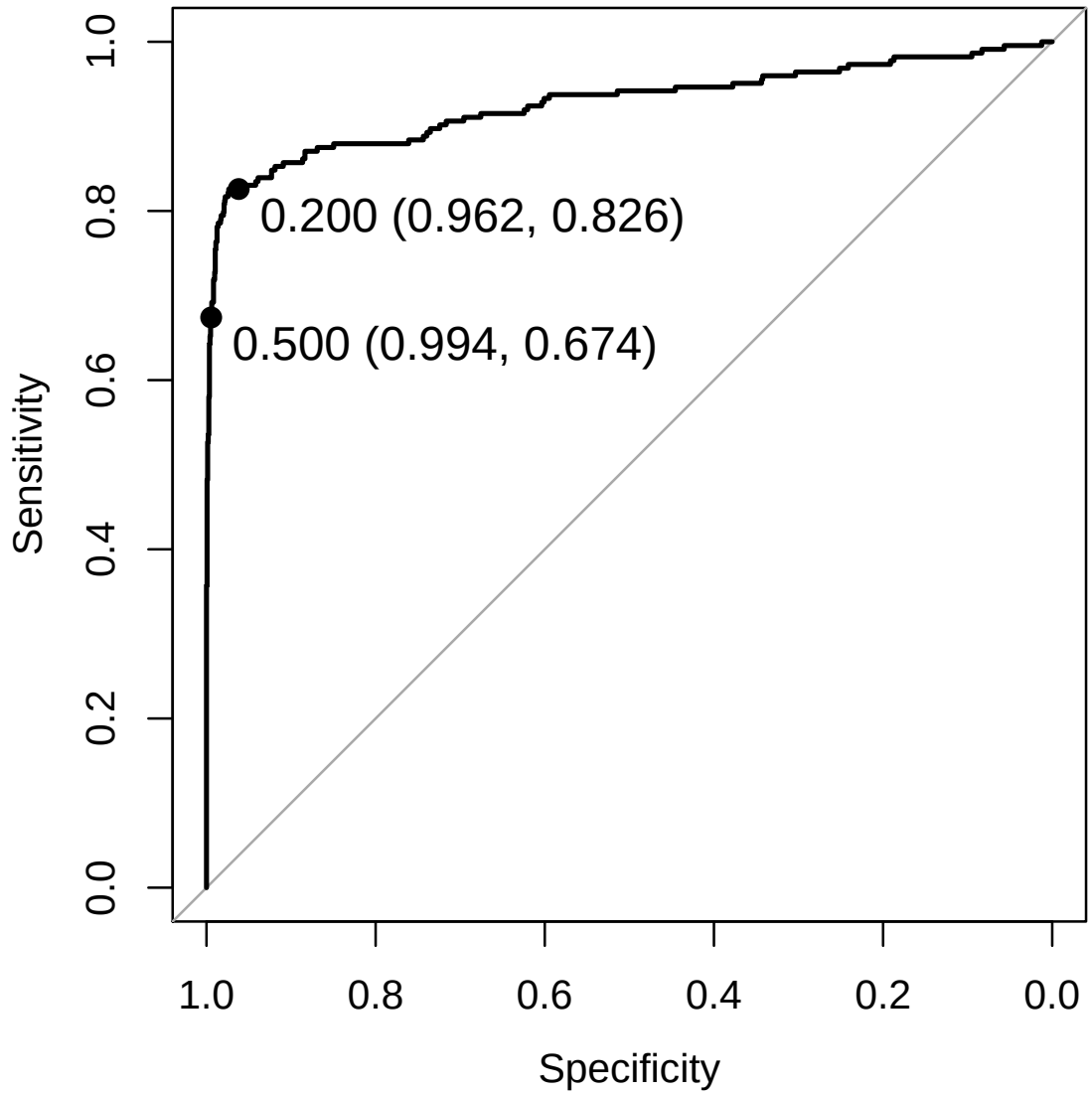
```
> rocCurve <- roc(response = churnTest$churn,
+                 predictor = gbmProbs[, "yes"],
+                 levels = rev(levels(churnTest$churn)))
> rocCurve


Call:
roc.default(response = churnTest$churn, predictor = gbmProbs[,      '

Data: gbmProbs[, "yes"] in 1443 controls (churnTest$churn no) < 224
Area under the curve: 0.927

> # plot(rocCurve)
```

Test Set ROC curve via the pROC Package



Switching To Other Models

It is simple to go between models using `train`.

A different `method` value is needed, any optional changes to `preProc` and potential options to pass through using `...`.

IF we set the seed to the same value prior to calling `train`, the exact same resamples are used to evaluate the model.

Switching To Other Models

Minimal changes are needed to fit different models to the same data:

```
> ## Using the same seed prior to train() will ensure that
> ## the same resamples are used (even in parallel)
> set.seed(1)
> svmTune <- train(churn ~ . , data = churnTrain,
+                  ## Tell it to fit a SVM model and tune
+                  ## over Cost and the RBF parameter
+                  method = "svmRadial",
+                  ## This pre-processing will be applied to
+                  ## these data and new samples too.
+                  preProc = c("center", "scale"),
+                  ## Tune over different values of cost
+                  tuneLength = 10,
+                  trControl = ctrl,
+                  metric = "ROC")
```

Switching To Other Models

How about:

```
> set.seed(1)
> fdaTune <- train(churn ~ . , data = churnTrain,
+                  ## Now try a flexible discriminant model
+                  ## using MARS basis functions
+                  method = "fda",
+                  tuneLength = 10,
+                  trControl = ctrl,
+                  metric = "ROC")
```

Other Functions and Classes

- `nearZeroVar`: a function to remove predictors that are sparse and highly unbalanced
- `findCorrelation`: a function to remove the optimal set of predictors to achieve low pair–wise correlations
- `predictors`: class for determining which predictors are included in the prediction equations (e.g. `rpart`, `earth`, `lars` models) (currently 7 methods)
- `confusionMatrix`, `sensitivity`, `specificity`, `posPredValue`, `negPredValue`: classes for assessing classifier performance
- `varImp`: classes for assessing the aggregate effect of a predictor on the model equations (currently 28 methods)

Other Functions and Classes

- `knnreg`: nearest–neighbor regression
- `plsda`, `splsda`: PLS discriminant analysis
- `icr`: independent component regression
- `pcaNNet`: `nnet:::nnet` with automatic PCA pre–processing step
- `bagEarth`, `bagFDA`: bagging with MARS and FDA models
- `normalize2Reference`: RMA–like processing of Affy arrays using a training set
- `spatialSign`: class for transforming numeric data ($x' = x/||x||$)
- `maxDissim`: a function for maximum dissimilarity sampling
- `rfe`: a class/framework for recursive feature selection (RFE) with external resampling step
- `sbf`: a class/framework for applying univariate filters prior to predictive modeling with external resampling
- `featurePlot`: a wrapper for several `lattice` functions

Thanks

Ray DiGiacomo

R Core

Pfizer's Statistics leadership for providing the time and support to create R packages

`caret` contributors: Jed Wing, Steve Weston, Andre Williams, Chris Keefer, Allan Engelhardt, Tony Cooper, Zachary Mayer and the R Core Team.

Session Info

- R Under development (unstable) (2014-02-14 r65008), `x86_64-apple-darwin10.8.0`
- Base packages: base, datasets, graphics, grDevices, methods, parallel, splines, stats, utils
- Other packages: C50 0.1.0-15, caret 6.0-24, class 7.3-9, doMC 1.3.2, e1071 1.6-1, earth 3.2-6, foreach 1.4.1, gbm 2.1, ggplot2 0.9.3.1, iterators 1.0.6, kernlab 0.9-19, knitr 1.5, lattice 0.20-24, mda 0.4-4, plotmo 1.3-2, plotrix 3.5-2, pls 2.4-3, plyr 1.8, pROC 1.6, randomForest 4.6-7, survival 2.37-7
- Loaded via a namespace (and not attached): car 2.0-19, codetools 0.2-8, colorspace 1.2-4, compiler 3.1.0, dichromat 2.0-0, digest 0.6.4, evaluate 0.5.1, formatR 0.10, grid 3.1.0, gtable 0.1.2, highr 0.3, labeling 0.2, MASS 7.3-29, munsell 0.4.2, nnet 7.3-7, proto 0.3-10, RColorBrewer 1.0-5, Rcpp 0.11.0, reshape2 1.2.2, scales 0.2.3, stringr 0.6.2, tools 3.1.0

This presentation was created with the `knitr` function at 23:23 on Wednesday, Feb 26, 2014.