Building Predictive Models in R Using the caret Package

Max Kuhn
Pfizer Global R&D

Abstract

The **caret** package, short for classification and regression training, contains numerous tools for developing predictive models using the rich set of models available in R. The package focuses on simplifying model training and tuning across a wide variety of modeling techniques. It also includes methods for pre-processing training data, calculating variable importance, and model visualizations. An example from computational chemistry is used to illustrate the functionality on a real data set and to benchmark the benefits of parallel processing with several types of models.

Keywords: model building, tuning parameters, parallel processing, R, **NetWorkSpaces**.

1. Introduction

The use of complex classification and regression models is becoming more and more commonplace in science, finance and a myriad of other domains (Ayres 2007). The R language (R Development Core Team 2008) has a rich set of modeling functions for both classification and regression, so many in fact, that it is becoming increasingly more difficult to keep track of the syntactical nuances of each function. The **caret** package, short for classification and regression training, was built with several goals in mind:

- to eliminate syntactical differences between many of the functions for building and predicting models,

- to develop a set of semi-automated, reasonable approaches for optimizing the values of the tuning parameters for many of these models and

- create a package that can easily be extended to parallel processing systems.

The package contains functionality useful in the beginning stages of a project (e.g., data splitting and pre-processing), as well as unsupervised feature selection routines and methods to tune models using resampling that helps diagnose over-fitting.

The package is available at the Comprehensive R Archive Network at http://CRAN.R-project.org/package=caret. **caret** depends on over 25 other packages, although many of these are listed as "suggested" packages which are not automatically loaded when **caret** is started. Packages are loaded individually when a model is trained or predicted. After the package is installed and loaded, help pages can be found using `help(package = "caret")`. There are three package vignettes that can be found using the `vignette` function (e.g., `vignette(package = "caret")`).

For the remainder of this paper, the capabilities of the package are discussed for: data splitting and pre-processing; tuning and building models; characterizing performance and variable importance; and parallel processing tools for decreasing the model build time. Rather than discussing the capabilities of the package in a vacuum, an analysis of an illustrative example is used to demonstrate the functionality of the package. It is assumed that the readers are familiar with the various tools that are mentioned. Hastie *et al.* (2001) is a good technical introduction to these tools.

2. An illustrative example

In computational chemistry, chemists often attempt to build predictive relationships between the structure of a chemical and some observed endpoint, such as activity against a biological target. Using the structural formula of a compound, chemical descriptors can be generated that attempt to capture specific characteristics of the chemical, such as its size, complexity, "greasiness" etc. Models can be built that use these descriptors to predict the outcome of interest. See Leach and Gillet (2003) for examples of descriptors and how they are used.

Kazius *et al.* (2005) investigated using chemical structure to predict mutagenicity (the increase of mutations due to the damage to genetic material). An Ames test (Ames *et al.* 1972) was used to evaluate the mutagenicity potential of various chemicals. There were 4,337 compounds included in the data set with a mutagenicity rate of 55.3%. Using these compounds, the **dragonX** software (Talete SRL 2007) was used to generate a baseline set of 1,579 predictors, including constitutional, topological and connectivity descriptors, among others. These variables consist of basic numeric variables (such as molecular weight) and counts variables (e.g., number of halogen atoms).

The descriptor data are contained in an R data frame names `descr` and the outcome data are in a factor vector called `mutagen` with levels `"mutagen"` and `"nonmutagen"`. These data are available from the package website http://caret.R-Forge.R-project.org/.

3. Data preparation

Since there is a finite amount of data to use for model training, tuning and evaluation, one of the first tasks is to determine how the samples should be utilized. There are a few schools of thought. Statistically, the most efficient use of the data is to train the model using all of the samples and use resampling (e.g., cross-validation, the bootstrap etc.) to evaluate the efficacy of the model. Although it is possible to use resampling incorrectly (Ambroise and

McLachlan 2002), this is generally true. However, there are some non-technical reasons why resampling alone may be insufficient. Depending on how the model will be used, an external test/validation sample set may be needed so that the model performance can be characterized on data that were not used in the model training. As an example, if the model predictions are to be used in a highly regulated environment (e.g., clinical diagnostics), the user may be constrained to "hold-back" samples in a validation set.

For illustrative purposes, we will do an initial split of the data into training and test sets. The test set will be used only to evaluate performance (such as to compare models) and the training set will be used for all other activities.

The function `createDataPartition` can be used to create stratified random splits of a data set. In this case, 75% of the data will be used for model training and the remainder will be used for evaluating model performance. The function creates the random splits within each class so that the overall class distribution is preserved as well as possible.

```
R> library("caret")
R> set.seed(1)
R> inTrain <- createDataPartition(mutagen, p = 3/4, list = FALSE)
R>
R> trainDescr <- descr[inTrain,]
R> testDescr  <- descr[-inTrain,]
R> trainClass <- mutagen[inTrain]
R> testClass  <- mutagen[-inTrain]
R>
R> prop.table(table(mutagen))
mutagen
   mutagen nonmutagen
 0.5536332  0.4463668
R> prop.table(table(trainClass))
trainClass
   mutagen nonmutagen
 0.5535055  0.4464945
```

In cases where the outcome is numeric, the samples are split into quartiles and the sampling is done within each quartile. Although not discussed in this paper, the package also contains method for selecting samples using maximum dissimilarity sampling (Willett 1999). This approach to sampling can be used to partition the samples into training and test sets on the basis of their predictor values.

There are many models where predictors with a single unique value (also known as "zero-variance predictors") will cause the model to fail. Since we will be tuning models using resampling methods, a random sample of the training set may result in some predictors with more than one unique value to become a zero-variance predictor (in our data, the simple split of the data into a test and training set caused three descriptors to have a single unique value in the training set). These so-called "near zero-variance predictors" can cause numerical problems during resampling for some models, such as linear regression.

As an example of such a predictor, the variable `nR04` is the number of number of 4-membered rings in a compound. For the training set, almost all of the samples ($n = 3,233$) have no

4-member rings while 18 compounds have one and a single compound has 2 such rings. If these data are resampled, this predictor might become a problem for some models.

To identify this kind of predictors, two properties can be examined:

- First, the percentage of unique values in the training set can be calculated for each predictor. Variables with low percentages have a higher probability of becoming a zero variance predictor during resampling. For `nR04`, the percentage of unique values in the training set is low (9.2%). However, this in itself is not a problem. Binary predictors, such as dummy variables, are likely to have low percentages and should not be discarded for this simple reason.

- The other important criterion to examine is the skewness of the frequency distribution of the variable. If the ratio of most frequent value of a predictor to the second most frequent value is large, the distribution of the predictor may be highly skewed. For `nR04`, the frequency ratio is large ($179 = 3233/18$), indicating a significant imbalance in the frequency of values.

If *both* these criteria are flagged, the predictor may be a near zero-variance predictor. It is suggested that if:

1. the percentage of unique values is less than 20% *and*

2. the ratio of the most frequent to the second most frequent value is greater than 20,

the predictor may cause problem for some models. The function `nearZeroVar` can be used to identify near zero-variance predictors in a dataset. It returns an index of the column numbers that violate the two conditions above.

Also, some models are susceptible to multicollinearity (i.e., high correlations between predictors). Linear models, neural networks and other models can have poor performance in these situations or may generate unstable solutions. Other models, such as classification or regression trees, might be resistant to highly correlated predictors, but multicollinearity may negatively affect interpretability of the model. For example, a classification tree may have good performance with highly correlated predictors, but the determination of which predictors are in the model is random.

If there is a need to minimize the effect of multicollinearity, there are a few options. First, models that are resistant to large between-predictor correlations, such as partial least squares, can be used. Also, principal component analysis can be used to reduce the number of dimensions in a way that removes correlations (see below). Alternatively, we can identify and remove predictors that contribute the most to the correlations.

In linear models, the traditional method for reducing multicollinearity is to identify the offending predictors using the variable inflation factor (VIF). For each variable, this statistic measures the increase in the variation of the model parameter estimate in comparison to the optimal situation (i.e., an orthogonal design). This is an acceptable technique when linear models are used and there are more samples than predictors. In other cases, it may not be as appropriate.

As an alternative, we can compute the correlation matrix of the predictors and use an algorithm to remove the a subset of the problematic predictors such that all of the pairwise

correlations are below a threshold:

repeat

Find the pair of predictors with the largest absolute correlation;

For both predictors, compute the average correlation between each predictor and all of the other variables;

Flag the variable with the largest mean correlation for removal;

Remove this row and column from the correlation matrix;

until *no correlations are above a threshold* ;

This algorithm can be used to find the minimal set of predictors that can be removed so that the pairwise correlations are below a specific threshold. Note that, if two variables have a high correlation, the algorithm determines which one is involved with the most pairwise correlations and is removed.

For illustration, predictors that result in absolute pairwise correlations greater than 0.90 can be removed using the `findCorrelation` function. This function returns an index of column numbers for removal.

```
R> ncol(trainDescr)
[1] 1576
R> descrCorr <- cor(trainDescr)
R> highCorr <- findCorrelation(descrCorr, 0.90)
R> trainDescr <- trainDescr[, -highCorr]
R> testDescr  <-  testDescr[, -highCorr]
R> ncol(trainDescr)
[1] 650
```

For chemical descriptors, it is not uncommon to have many very large correlations between the predictors. In this case, using a threshold of 0.90, we eliminated 926 descriptors from the data.

Once the final set of predictors is determined, the values may require transformations before being used in a model. Some models, such as partial least squares, neural networks and support vector machines, need the predictor variables to be centered and/or scaled. The `preProcess` function can be used to determine values for predictor transformations using the training set and can be applied to the test set or future samples. The function has an argument, `method`, that can have possible values of `"center"`, `"scale"`, `"pca"` and `"spatialSign"`. The first two options provide simple location and scale transformations of each predictor (and are the default values of `method`). The `predict` method for this class is then used to apply the processing to new samples

```
R> xTrans <- preProcess(trainDescr)
R> trainDescr <- predict(xTrans, trainDescr)
R> testDescr  <- predict(xTrans,  testDescr)
```

The `"pca"` option computes loadings for principal component analysis that can be applied to any other data set. In order to determine how many components should be retained, the `preProcess` function has an argument called `thresh` that is a threshold for the cumulative percentage of variance captured by the principal components. The function will add components until the cumulative percentage of variance is above the threshold. Note that the data

are automatically scaled when `method = "pca"`, even if the `method` value did not indicate that scaling was needed. For PCA transformations, the predict method generates values with column names `"PC1"`, `"PC2"`, etc.

Specifying `method = "spatialSign"` applies the spatial sign transformation (Serneels *et al.* 2006) where the predictor values for each sample are projected onto a unit circle using $x^* = x/||x||$. This transformation may help when there are outliers in the x space of the training set.

4. Building and tuning models

The `train` function can be used to select values of model tuning parameters (if any) and/or estimate model performance using resampling. As an example, a radial basis function support vector machine (SVM) can be used to classify the samples in our computational chemistry data. This model has two tuning parameters. The first is the scale function σ in the radial basis function

$$K(a, b) = \exp(-\sigma||a - b||^2)$$

and the other is the cost value C used to control the complexity of the decision boundary. We can create a grid of candidate tuning values to evaluate. Using resampling methods, such as the bootstrap or cross-validation, a set of modified data sets are created from the training samples. Each data set has a corresponding set of hold-out samples. For each candidate tuning parameter combination, a model is fit to each resampled data set and is used to predict the corresponding held out samples. The resampling performance is estimated by aggregating the results of each hold-out sample set. These performance estimates are used to evaluate which combination(s) of the tuning parameters are appropriate. Once the final tuning values are assigned, the final model is refit using the entire training set.

For the `train` function, the possible resampling methods are: bootstrapping, k-fold cross-validation, leave-one-out cross-validation, and leave-group-out cross-validation (i.e., repeated splits without replacement). By default, 25 iterations of the bootstrap are used as the resampling scheme. In this case, the number of iterations was increased to 200 due to the large number of samples in the training set.

For this particular model, it turns out that there is an analytical method for directly estimating a suitable value of σ from the training data (Caputo *et al.* 2002). By default, the `train` function uses the `sigest` function in the **kernlab** package (Karatzoglou *et al.* 2004) to initialize this parameter. In doing this, the value of the cost parameter C is the only tuning parameter.

The `train` function has the following arguments:

> `x:` a matrix or data frame of predictors. Currently, the function only accepts numeric values (i.e., no factors or character variables). In some cases, the `model.matrix` function may be needed to generate a data frame or matrix of purely numeric data

> `y:` a numeric or factor vector of outcomes. The function determines the type of problem (classification or regression) from the type of the response given in this argument.

> `method:` a character string specifying the type of model to be used. See Table 1 for the possible values.

metric: a character string with values of `"Accuracy"`, `"Kappa"`, `"RMSE"` or `"Rsquared"`. This value determines the objective function used to select the final model. For example, selecting `"Kappa"` makes the function select the tuning parameters with the largest value of the mean Kappa statistic computed from the held-out samples.

trControl: takes a list of control parameters for the function. The type of resampling as well as the number of resampling iterations can be set using this list. The function `trainControl` can be used to compute default parameters. The default number of resampling iterations is 25, which may be too small to obtain accurate performance estimates in some cases.

tuneLength: controls the size of the default grid of tuning parameters. For each model, `train` will select a grid of complexity parameters as candidate values. For the SVM model, the function will tune over $C = 10^{-1}, 1, 10$. To expand the size of the default list, the `tuneLength` argument can be used. By selecting `tuneLength = 5`, values of C ranging from 0.1 to $1,000$ are evaluated.

tuneGrid: can be used to define a specific grid of tuning parameters. See the example below.

... : the three dots can be used to pass additional arguments to the functions listed in Table 1. For example, we have already centered and scaled the predictors, so the argument `scaled = FALSE` can be passed to the `ksvm` function to avoid duplication of the pre-processing.

We can tune and build the SVM model using the code below.

```
R> bootControl <- trainControl(number = 200)
R> set.seed(2)
R> svmFit <- train(trainDescr, trainClass,
+    method = "svmRadial", tuneLength = 5,
+    trControl = bootControl, scaled = FALSE)
Model 1: sigma=0.0004329517, C=1e-01
Model 2: sigma=0.0004329517, C=1e+00
Model 3: sigma=0.0004329517, C=1e+01
Model 4: sigma=0.0004329517, C=1e+02
Model 5: sigma=0.0004329517, C=1e+03
R> svmFit

Call:
train.default(x = trainDescr, y = trainClass, method = "svmRadial",
    scaled = FALSE, trControl = bootControl, tuneLength = 5)

3252 samples
650 predictors

summary of bootstrap (200 reps) sample sizes:
    3252, 3252, 3252, 3252, 3252, 3252, ...
```

```
boot resampled training results across tuning parameters:

  sigma     C       Accuracy  Kappa   Accuracy SD  Kappa SD  Selected
  0.000433  0.1     0.705     0.395   0.0122       0.0252
  0.000433  1       0.806     0.606   0.0109       0.0218
  0.000433  10      0.818     0.631   0.0104       0.0211     *
  0.000433  100     0.8       0.595   0.0112       0.0227
  0.000433  1000    0.782     0.558   0.0111       0.0223

Accuracy was used to select the optimal model
```

In this output, each row in the table corresponds to a specific combination of tuning parameters. The "Accuracy" column is the average accuracy of the 200 held-out samples and the column labeled as "Accuracy SD" is the standard deviation of the 200 accuracies.

The Kappa statistic is a measure of concordance for categorical data that measures agreement relative to what would be expected by chance. Values of 1 indicate perfect agreement, while a value of zero would indicate a lack of agreement. Negative Kappa values can also occur, but are less common since it would indicate a negative association between the observed and predicted data. Kappa is an excellent performance measure when the classes are highly unbalanced. For example, if the mutagenicity rate in the data had been very small, say 5%, most models could achieve high accuracy by predicting all compounds to be nonmutagenic. In this case, the Kappa statistic would result in a value near zero. The Kappa statistic given here is the unweighted version computed by the `classAgreement` function in the **e1071** package (Dimitriadou *et al.* 2008). The Kappa columns in the output above are also summarized across the 200 resampled Kappa estimates.

As previously mentioned, the "optimal" model is selected to be the candidate model with the largest accuracy. If more than one tuning parameter is "optimal" then the function will try to choose the combination that corresponds to the least complex model. For these data, σ was estimated to be 0.000433 and $C = 10$ appears to be optimal. Based on these values, the model was refit to the original set of 3,252 samples and this object is stored in `svmFit$finalModel`.

```
R> svmFit$finalModel
Support Vector Machine object of class "ksvm"

SV type: C-svc  (classification)
 parameter : cost C = 10

Gaussian Radial Basis kernel function.
 Hyperparameter : sigma =  0.000432951668058316

Number of Support Vectors : 1616

Objective Function Value : -9516.185
Training error : 0.082411
Probability model included.
```

Model	`method` value	Package	Tuning parameters
Recursive partitioning	`rpart`	**rpart***	`maxdepth`
	`ctree`	**party**	`mincriterion`
Boosted trees	`gbm`	**gbm***	`interaction.depth,` `n.trees, shrinkage`
	`blackboost`	**mboost**	`maxdepth, mstop`
	`ada`	**ada**	`maxdepth, iter, nu`
Other boosted models	`glmboost`	**mboost**	`mstop`
	`gamboost`	**mboost**	`mstop`
	`logitboost`	**caTools**	`nIter`
Random forests	`rf`	**randomForest***	`mtry`
	`cforest`	**party**	`mtry`
Bagged trees	`treebag`	**ipred**	None
Neural networks	`nnet`	**nnet**	`decay, size`
Partial least squares	`pls`*	**pls**, **caret**	`ncomp`
Support vector machines (RBF kernel)	`svmRadial`	**kernlab**	`sigma, C`
Support vector machines (polynomial kernel)	`svmPoly`	**kernlab**	`scale, degree, C`
Gaussian processes (RBF kernel)	`gaussprRadial`	**kernlab**	`sigma`
Gaussian processes (polynomial kernel)	`gaussprPoly`	**kernlab**	`scale, degree`
Linear least squares	`lm`*	**stats**	None
Multivariate adaptive regression splines	`earth`*, `mars`	**earth**	`degree, nprune`
Bagged MARS	`bagEarth`*	**caret**, **earth**	`degree, nprune`
Elastic net	`enet`	**elasticnet**	`lambda, fraction`
The lasso	`lasso`	**elasticnet**	`fraction`
Relevance vector machines (RBF kernel)	`rvmRadial`	**kernlab**	`sigma`
Relevance vector machines (polynomial kernel)	`rvmPoly`	**kernlab**	`scale, degree`
Linear discriminant analysis	`lda`	**MASS**	None
Stepwise diagonal discriminant analysis	`sddaLDA,` `sddaQDA`	**SDDA**	None
Logistic/multinomial regression	`multinom`	**nnet**	`decay`
Regularized discriminant analysis	`rda`	**klaR**	`lambda, gamma`
Flexible discriminant analysis (MARS basis)	`fda`*	**mda**, **earth**	`degree, nprune`

Table 1: Models used in `train` (* indicates that a model-specific variable importance method is available, see Section 9.). *(continued on next page)*

Model	`method` value	Package	Tuning parameters
Bagged FDA	`bagFDA`*	**caret**, **earth**	`degree`, `nprune`
Least squares support vector machines (RBF kernel)	`lssvmRadial`	**kernlab**	`sigma`
k nearest neighbors	`knn3`	**caret**	`k`
Nearest shrunken centroids	`pam`*	**pamr**	`threshold`
Naive Bayes	`nb`	**klaR**	`usekernel`
Generalized partial least squares	`gpls`	**gpls**	`K.prov`
Learned vector quantization	`lvq`	**class**	`k`

Table 1: Models used in `train` (* indicates that a model-specific variable importance method is available, see Section 9.).

In many cases, more control over the grid of tuning parameters is needed. For example, for boosted trees using the `gbm` function in the **gbm** package (Ridgeway 2007), we can tune over the number of trees (i.e., boosting iterations), the complexity of the tree (indexed by `interaction.depth`) and the learning rate (also known as `shrinkage`). As an example, a user could specify a grid of values to tune over using a data frame where the rows correspond to tuning parameter combinations and the columns are the names of the tuning variables (preceded by a dot). For our data, we will generate a grid of 50 combinations and use the `tuneGrid` argument to the `train` function to use these values.

```
R> gbmGrid <- expand.grid(.interaction.depth = (1:5) * 2,
+     .n.trees = (1:10)*25, .shrinkage = .1)
R> set.seed(2)
R> gbmFit <- train(trainDescr, trainClass,
+     method = "gbm", trControl = bootControl, verbose = FALSE,
+     bag.fraction = 0.5, tuneGrid = gbmGrid)
Model 1: interaction.depth= 2, shrinkage=0.1, n.trees=250
 collapsing over other values of n.trees
Model 2: interaction.depth= 4, shrinkage=0.1, n.trees=250
 collapsing over other values of n.trees
Model 3: interaction.depth= 6, shrinkage=0.1, n.trees=250
 collapsing over other values of n.trees
Model 4: interaction.depth= 8, shrinkage=0.1, n.trees=250
 collapsing over other values of n.trees
Model 5: interaction.depth=10, shrinkage=0.1, n.trees=250
 collapsing over other values of n.trees
```

In this model, we generated 200 bootstrap replications for each of the 50 candidate models, computed performance and selected the model with the largest accuracy. In this case the model automatically selected an interaction depth of 8 and used 250 boosting iterations (although other values may very well be appropriate; see Figure 1). There are a variety
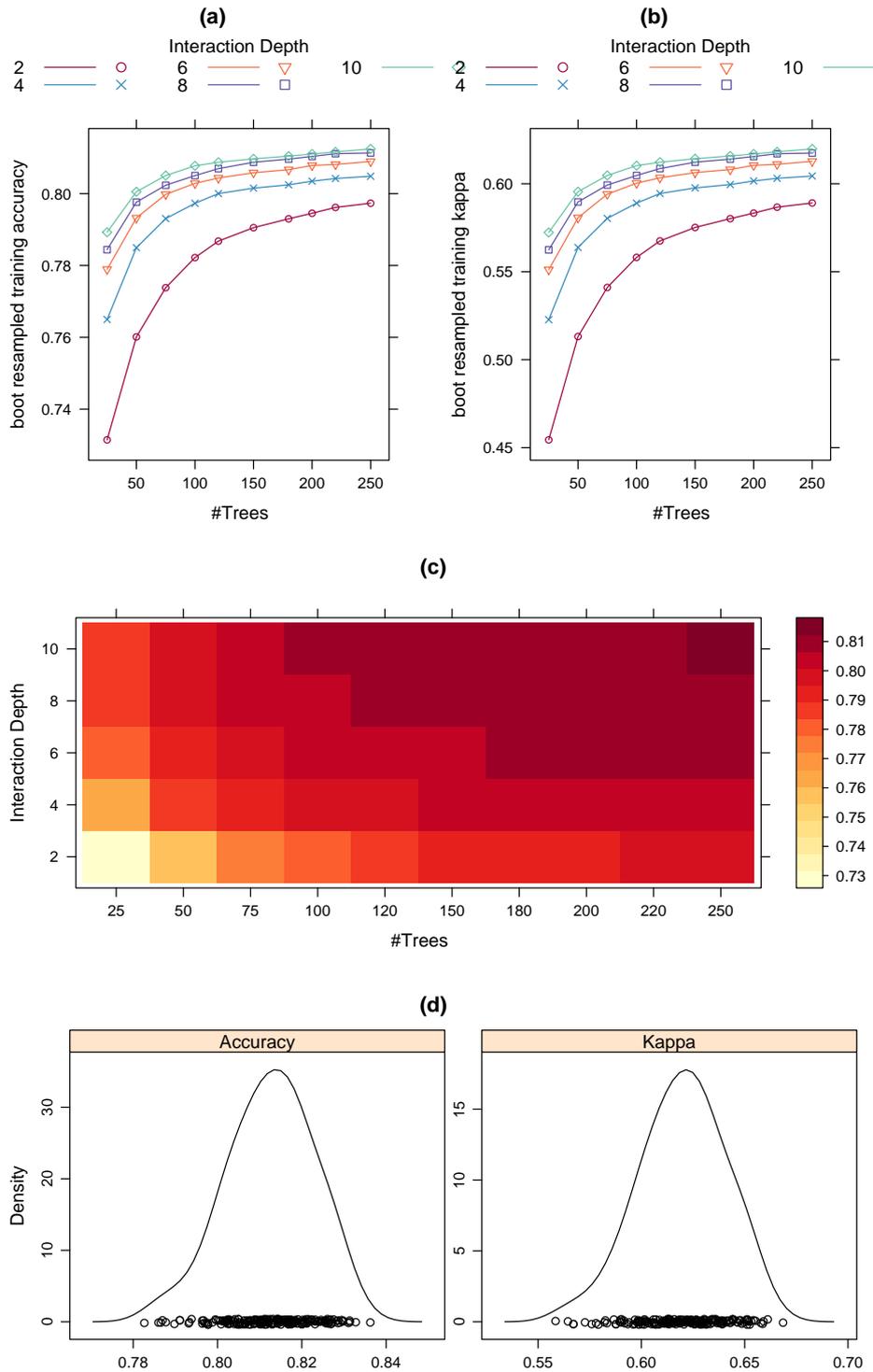
Figure 1: Examples of plot functions for `train` objects. (a) A plot of the classification accuracy versus the tuning factors (using `plot(gbmFit)`). (b) Similarly, a plot of the Kappa statistic profiles (`plot(gbmFit, metric = "Kappa")`). (c) A level plot of the accuracy values (`plot(gbmFit, plotType = "level")`). (d) Density plots of the 200 bootstrap estimates of accuracy and Kappa for the final model (`resampleHist(gbmFit)`).

of different visualizations for `train` objects. Figure 1 shows several examples plots created using `plot.train` and `resampleHist`.

Note that the output states that the procedure was "collapsing over other values of n.trees". For some models (`method` values of `pls`, `plsda`, `earth`, `rpart`, `gbm`, `gamboost`, `glmboost`, `blackboost`, `ctree`, `pam`, `enet` and `lasso`), the `train` function will fit a model that can be used to derive predictions for some sub-models. For example, since boosting models save the model results for each iteration of boosting, `train` can fit the model with the largest number of iterations and derive the other models where the other tuning parameters are the same but fewer number of boosting iterations are requested. In the example above, for a model with `interaction.depth = 2` and `shrinkage = .1`, we only need to fit the model with the largest number of iterations (250 in this example). Holding the interaction depth and shrinkage constant, the computational time to get predictions for models with less than 250 iterations is relatively cheap. For the example above, we fit a total of $200 \times 5 = 1,000$ models instead of $25 \times 5 \times 10 = 10,000$. The `train` function tries to exploit this idea for as many models as possible.

For recursive partitioning models, an initial model is fit to all of the training data to obtain the possible values of the maximum depth of any node (`maxdepth`). The tuning grid is created based on these values. If `tuneLength` is larger than the number of possible `maxdepth` values determined by the initial model, the grid will be truncated to the `maxdepth` list. The same is also true for nearest shrunken centroid models, where an initial model is fit to find the range of possible threshold values, and MARS models (see Section 7).

Also, for the `glmboost` and `gamboost` functions from the **mboost** package (Hothorn and Bühlmann 2007), an additional tuning parameter, `prune`, is used by train. If `prune = "yes"`, the number of trees is reduced based on the AIC statistic. If `"no"`, the number of trees is kept at the value specified by the `mstop` parameter. See Bühlmann and Hothorn (2007) for more details about AIC pruning.

In general, the functions in the **caret** package assume that there are no missing values in the data or that these values have been handled via imputation or other means. For the `train` function, there are some models (such as `rpart`) that can handle missing values. In these cases, the data passed to the `x` argument can contain missing values.

5. Prediction of new samples

As previously noted, an object of class `train` contains an element called `finalModel`, which is the fitted model with the tuning parameter values selected by resampling. This object can be used in the traditional way to generate predictions for new samples, using that model's `predict` function. For the most part, the prediction functions in R follow a consistent syntax, but there are exceptions. For example, boosted tree models produced by the **gbm** function also require the number of trees to be specified. Also, `predict.mvr` from the **pls** package (Mevik and Wehrens 2007) will produce predictions for every candidate value of `ncomp` that was tested. To avoid having to remember these nuances, **caret** offers several functions to deal with these issues.

The function `predict.train` is an interface to the model's `predict` method that handles any extra parameter specifications (such as previously mentioned for `gbm` and PLS models). For example:

```
R> predict(svmFit$finalModel, newdata = testDescr)[1:5]
[1] mutagen    nonmutagen nonmutagen nonmutagen mutagen
Levels: mutagen nonmutagen

R> predict(svmFit, newdata = testDescr)[1:5]
[1] mutagen    nonmutagen nonmutagen nonmutagen mutagen
Levels: mutagen nonmutagen
```

In cases where predictions are needed for multiple models based on the same data set, `predict.list` can be used implicitly:

```
R> models <- list(svm = svmFit, gbm = gbmFit)
R> testPred <- predict(models, newdata = testDescr)
R> lapply(testPred, function(x) x[1:5])
$svm
[1] mutagen    nonmutagen nonmutagen nonmutagen mutagen
Levels: mutagen nonmutagen


$gbm
[1] mutagen    mutagen    mutagen    nonmutagen mutagen
Levels: mutagen nonmutagen
```

`predict.train` produces a vector of predictions for each model. The function `extractPrediction` can be used to obtain predictions for training, test and/or unknown samples at once and will return the data in a data frame. For example:

```
R> predValues <- extractPrediction(models,
+    testX = testDescr, testY = testClass)
R> testValues <- subset(predValues, dataType == "Test")
R> head(testValues)
            obs       pred     model dataType
3253    mutagen    mutagen svmRadial     Test
3254 nonmutagen nonmutagen svmRadial     Test
3255    mutagen nonmutagen svmRadial     Test
3256 nonmutagen nonmutagen svmRadial     Test
3257    mutagen    mutagen svmRadial     Test
3258    mutagen    mutagen svmRadial     Test

R> table(testValues$model)
     gbm svmRadial
    1083      1083
R> nrow(testDescr)
[1] 1083
```

The output has columns for the observed outcome, the model prediction, the model type and the data type (i.e., training, test or unknown).

Many classification models listed in Table 1 can produce class probabilities. The values can be accessed using `predict.train` using the argument `type = "prob"`. In this case, the
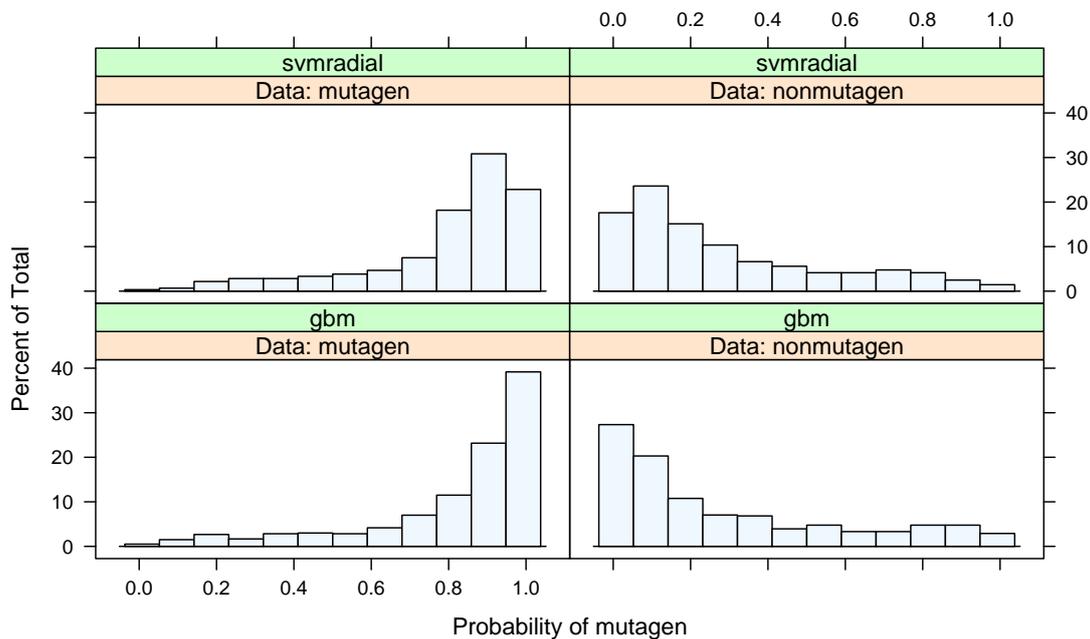
Figure 2: Histograms of the "mutagen" class probability for two models (produced using `plotClassProbs(testProbs)`). The panels correspond to the model used and the observed class (labeled as `Data` in the panels).

function will return a data frame with probability columns for each class. Also, the function `extractProb` is similar to `extractPrediction`, but will produce the class probabilities for each class in the data. An additional column of probabilities is supplied for each level of the factor variable supplied to `train`.

```
R> probValues <- extractProb(models,
+    testX = testDescr, testY = testClass)
R> testProbs <- subset(probValues, dataType == "Test")
R> str(testProbs)
'data.frame':        2166 obs. of  6 variables:
 $ mutagen   : num  0.6274 0.2970 0.1691 0.0177 0.9388 ...
 $ nonmutagen: num  0.3726 0.7030 0.8309 0.9823 0.0612 ...
 $ obs       : Factor w/ 2 levels "mutagen","nonmutagen": 1 2 1 2 1 1 2 2 2 2 ...
 $ pred      : Factor w/ 2 levels "mutagen","nonmutagen": 1 2 2 2 1 1 1 2 2 2 2 ...
 $ model     : chr  "svmRadial" "svmRadial" "svmRadial" "svmRadial" ...
 $ dataType  : chr  "Test" "Test" "Test" "Test" ...
```

For classification models, the function `plotClassProbs` function can create a **lattice** plot of histograms (Sarkar 2008) to visualize the distributions of class probabilities using the output from `plotClassProbs`. Figure 2 was generated using `plotClassProbs(testProbs)` and shows histograms of the probability of the mutagen prediction across different models and the true classes.

6. Characterizing performance

caret also contains several functions that can be used to describe the performance of classification models. For classification models, the functions `sensitivity`, `specificity`, `posPredValue` and `negPredValue` can be used to characterize performance where there are two classes. By default, the first level of the outcome factor is used to define the "positive" result, although this can be changed.

The function `confusionMatrix` can be used to compute various summaries for classification models. For example, we can assess the support vector machine's performance on the test set.

```
R> svmPred <- subset(testValues, model == "svmRadial")
R> confusionMatrix(svmPred$pred, svmPred$obs)
Loading required package: class
Confusion Matrix and Statistics

           Reference
Prediction   mutagen nonmutagen
  mutagen        528        102
  nonmutagen      72        381

           Accuracy : 0.8393
             95% CI : (0.8161, 0.8607)
No Information Rate : 0.554
P-Value [Acc > NIR] : 6.196e-89

              Kappa : 0.6729

        Sensitivity : 0.88
        Specificity : 0.7888
     Pos Pred Value : 0.8381
     Neg Pred Value : 0.8411
```

In addition to the cross-tabulation of the observed and predicted values, various statistics are calculated. The confidence interval for the accuracy rate uses the default binomial confidence interval method used in `binom.test`.

The "no-information rate" shown on the output is the largest proportion of the observed classes (there were more mutagens than nonmutagens in this test set). A one-sided hypothesis test is also computed to evaluate whether the overall accuracy rate is greater than the rate of the largest class. Like Kappa, this is helpful for data sets where there is a large imbalance between the classes.

When there are three or more classes, `confusionMatrix` will show the confusion matrix and a set of "one-versus-all" results. For example, in a three class problem, the sensitivity of the first class is calculated against all the samples in the second and third classes (and so on).

Receiver operating characteristic (ROC) curves can also be computed using the **caret** package. The `roc` function takes as input a numeric score and a factor variable with the true class labels.

Larger values of the numeric data should indicate that the sample is more likely to have come from the first level of the factor. For example, the class probability associated with the "mutagen" class and the observed classes can be used to compute an ROC curve for these data ("mutagen" is the first factor level in `svmProb$obs`). For example, in the code below, `svmProb$mutagen` contains the mutagen class probabilities.

```
R> svmProb <- subset(testProbs, model == "svmRadial")
R> svmROC <- roc(svmProb$mutagen, svmProb$obs)
R> str(svmROC)
 num [1:1083, 1:3]     NA 0.00352 0.00401 0.00460 0.00589 ...
 - attr(*, "dimnames")=List of 2
  ..$ : NULL
  ..$ : chr [1:3] "cutoff" "sensitivity" "specificity"
```

The result is a matrix with columns for the cutoff, the sensitivity and specificity. These can be used to create an ROC curve for the test set. The function `aucRoc` can be used to compute the simple area under the curve statistic (via the trapezoidal rule).

7. Regression models

Using `train` to build regression models is almost exactly the same process as the one shown in the previous sections. For regression models, performance is calculated using the root mean squared error and R^2 instead of accuracy and the Kappa statistic. However, there are many models where there is no notion of model degrees of freedom (such as random forests) or where there are more parameters than training set samples. Given this, `train` ignores degrees of freedom when computing performance values. For example, to compute R^2, the correlation coefficient is computed between the observed and predicted values and squared. When comparing models, the performance metrics will be on the same scale, but these metrics do not penalize model complexity (as adjusted R^2 does) and will tend to favor more complex fits over simpler models.

For multivariate adaptive regression spline (MARS) models, the **earth** package (Milborrow 2007) is used when a model type of `mars` or `earth` is requested. The tuning parameters used by `train` are `degree` and `nprune`. The parameter `nk` is not automatically specified and the default in the **earth** function is used. For example, suppose a training set with 40 predictors is used with a MARS model using `degree = 1` and `nprune = 20`. An initial model with `nk = 41` is fit and is pruned down to 20 terms. This number includes the intercept and may include "singleton" terms instead of pairs. Alternate model training schemes can be used by passing `nk` and/or `pmethod` to the `earth` function.

caret also includes a function, `plotObsVsPred`, that can produce a **lattice** plot of the observed responses versus the predicted values for various models and data sets.

8. Other modeling functions

The package also includes other model functions. The `knn3` function is a clone of `knn` from the **MASS** package (Venables and Ripley 2002) whose `predict` function can return the vote

proportions for each of the classes (instead of just the winning class). Also, there are functions that produce bagged versions of MARS and flexible discriminant analysis (FDA) models. These two functions, `bagEarth` and `bagFDA`, can be used to produce smoother prediction functions/decision boundaries while including integrated feature selection. Another function, `plsda`, builds partial least squares discriminant analysis models (Barker and Rayens 2003). In this case, a matrix of dummy variables is created with a column for each class. This response matrix is used as an input to the `plsr` function of the **pls** package. The predict function can be used to produce either the raw model predictions, the class probabilities (computed using the softmax equation) or the class prediction.

9. Predictor importance

The generic function `varImp` can be used to characterize the general effect of predictors on the model. The `varImp` function works with the following object classes: `lm`, `mars`, `earth`, `randomForest`, `gbm`, `mvr` (in the **pls** package), `rpart`, `RandomForest` (from the **party** package), `pamrtrained`, `bagEarth`, `bagFDA`, `classbagg` and `regbagg`. `varImp` also works with objects produced by `train`, but this is a simple wrapper for the specific models previously listed.

Each model characterizes predictor importance differently:

- **Linear models**: The absolute value of the t statistic for each model parameter is used.

- **Random forest** from Liaw and Wiener (2002): "For each tree, the prediction accuracy on the out-of-bag portion of the data is recorded. Then the same is done after permuting each predictor variable. The difference between the two accuracies are then averaged over all trees, and normalized by the standard error. For regression, the MSE is computed on the out-of-bag data for each tree, and then the same computed after permuting a variable. The differences are averaged and normalized by the standard error. If the standard error is equal to 0 for a variable, the division is not done." `varImp.randomForest` is a simple wrapper around the `importance` function from that package. Similarly, for `RandomForest` objects, `varImp` is a wrapper around `varimp` in the **party** package.

- **Partial least squares**: The variable importance measure here is based on weighted sums of the absolute regression coefficients. The weights are a function of the reduction of the sums of squares across the number of PLS components and are computed separately for each outcome. Therefore, the contribution of the coefficients are weighted proportionally to the reduction in the sums of squares.

- **Recursive partitioning**: The reduction in the loss function (e.g., mean squared error) attributed to each variable at each split is tabulated and the sum is returned. Also, since there may be candidate variables that are important but are not used in a split, the top competing variables are also tabulated at each split. This can be turned off using the `maxcompete` argument in `rpart.control`. This method does not currently provide class-specific measures of importance when the response is a factor.

- **Bagged trees**: The same methodology as a single tree is applied to all bootstrapped trees and the total importance is returned

- **Boosted trees**: This method uses the same approach as a single tree, but sums the importances over each boosting iteration (see Ridgeway 2007).

- **Multivariate adaptive regression splines**: MARS models include a backwards elimination feature selection routine that looks at reductions in the generalized cross-validation (GCV) estimate of error. The `varImp` function tracks the changes in model statistics, such as the GCV, for each predictor and accumulates the reduction in the statistic when each predictor's feature is added to the model. This total reduction is used as the variable importance measure. If a predictor was never used in any MARS basis function, it has an importance value of zero. There are three statistics that can be used to estimate variable importance in MARS models. Using `varImp(object, value = "gcv")` tracks the reduction in the generalized cross-validation statistic as terms are added. Also, the option `varImp(object, value = "grsq")` compares the GCV statistic for each model to the intercept only model. However, there are some cases when terms are retained in the model that result in an increase in GCV. Negative variable importance values for MARS are set to a small, non-zero number. Alternatively, using `varImp(object, value = "rss")` monitors the change in the residual sums of squares (RSS) as terms are added, which will never be negative.

- **Bagged MARS and FDA**: For these objects, importance is calculated using the method for MARS/earth models (as previously described) for each bagged model. The overall importance is aggregated across the bagged results.

- **Nearest shrunken centroid models**: The difference between the class centroids and the overall centroid is used to measure the variable influence (see `pamr.predict`). The larger the difference between the class centroid and the overall center of the data, the larger the separation between the classes. The training set predictions must be supplied when an object of class `pamrtrained` is given to `varImp`.

For the mutagenicity example, we can assess which predictors had the largest impact on the model.

```
R> gbmImp <- varImp(gbmFit, scale = FALSE)
R> gbmImp
gbm variable importance

  only 20 most important variables shown (out of 650 )

        Overall
piPC09   115.80
PCR      102.16
MATS1p    58.24
N.078     49.45
AMW       48.51
nR03      45.54
nN.       40.85
Mor12m    38.89
O.057     37.93
```

```
MAXDN      35.07
R2u.       29.03
R3e        28.06
ClogP      27.53
N.069      26.30
Ms         26.14
D.Dr03     25.96
BELm1      25.32
Mor24m     24.38
RDF040m    22.36
T.N..N.    22.34
```

The top two descriptors, the molecular multiple path count of order 09 and the ratio of multiple path count over path count, have large contributions relative to the other descriptors. These two descriptors have a correlation of 0.76, indicating that they might be measuring similar underlying mechanisms (both descriptors are measures of molecular complexity). For this example, there were 218 predictors with importance values equal to zero, meaning that these were not used in any splits in any of the trees.

A plot method for `varImp` is included that produces a "needle plot" of the importance values where the predictors are sorted from most-important to least. Figure 3 provides an example for the boosted tree model.

The advantages of using a model-based approach are that it is more closely tied to the model performance and that the importance *may* be able to incorporate the correlation structure between the predictors into the importance calculation. Regardless of how the importance is calculated:

- For most classification models, each predictor will have a separate variable importance for each class (the exceptions are FDA, classification trees, `trees` and boosted trees).

- All measures of importance are scaled to have a maximum value of 100, unless the `scale` argument of `varImp.train` is set to `FALSE` (as in the example above).

If there is no model-specific way to estimate importance, the importance of each predictor can be evaluated individually using a "filter" approach. This approach is most accessible using `varImp.train` either for models that are not included in the list above or by using `useModel = FALSE` in `varImp.train`.

For classification models, ROC curve analysis is conducted on each predictor. For two class problems, a series of cutoffs is applied to the predictor data to predict the class. The sensitivity and specificity are computed for each cutoff and the ROC curve is computed. The trapezoidal rule is used to compute the area under the ROC curve. This area is used as the measure of variable importance. For multi-class outcomes, the problem is decomposed into all pairwise problems and the area under the curve is calculated for each class pair (i.e., class 1 vs. class 2, class 2 vs. class 3 etc.). For a specific class, the maximum area under the curve across the relevant pairwise AUC's is used as the variable importance measure.

Figure 4 shows a hypothetical example of this approach for three classes. For this predictor, three ROC curves are computed, along with the area under the ROC curve. To determine the importance of this predictor to class 1, the maximum of the two AUCs related to this class
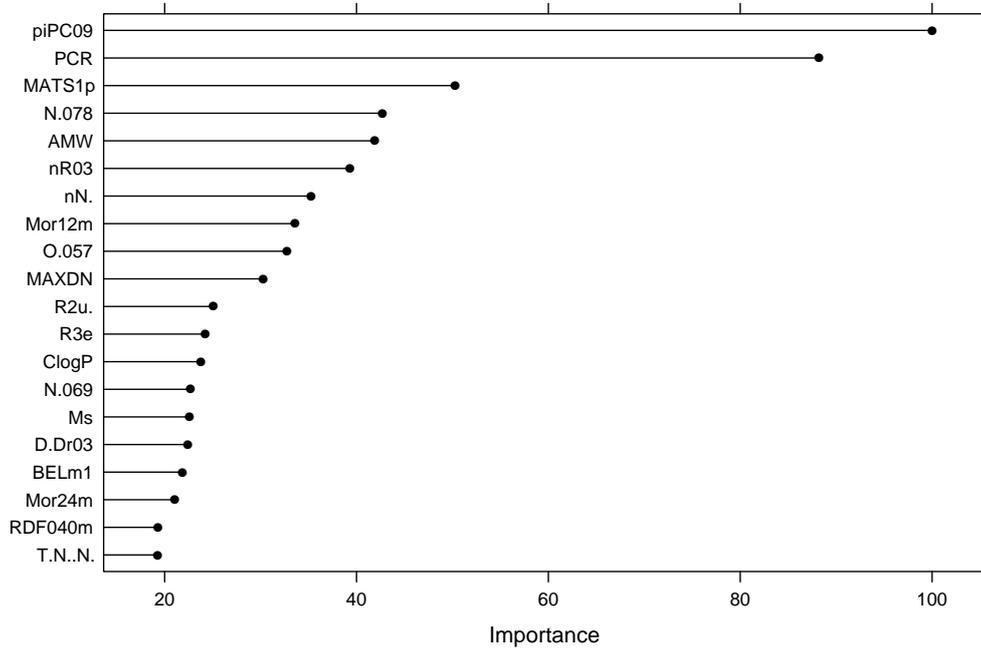
Figure 3: A needle plot of the boosted tree variable importance values (produced using `plot(varImp(gbmFit), top = 20))`.
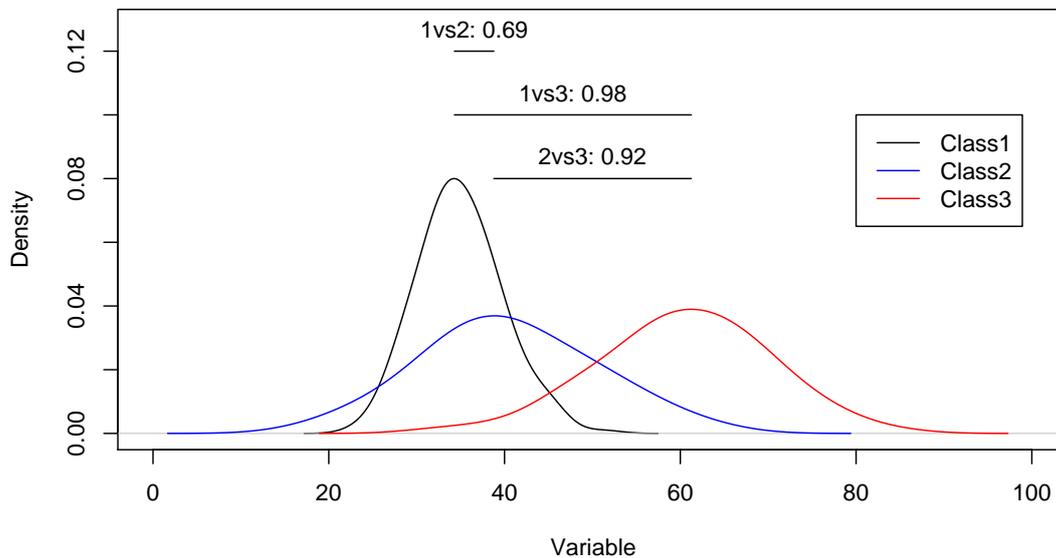

Figure 4: An example of the model-free technique for characterizing predictor importance in classification models using ROC curves. Each pairwise ROC curve is calculated and the maximum AUC for each class is computed.

is computed $(0.98 = \max(0.98, 0.69))$. In other words, this predictor is important to at least one class. The importances of this predictor to classes 2 and 3 are 0.92 and 0.98, respectively.

For regression, the relationship between each predictor and the outcome is evaluated. An argument, `nonpara`, is used to pick the model fitting technique. When `nonpara = FALSE`, a linear model is fit and the absolute value of the t value for the slope of the predictor is used. Otherwise, a loess smoother is fit between the outcome and the predictor. The R^2 statistic is calculated for this model against the intercept only null model. This number is returned as a relative measure of variable importance.

10. Parallel processing

If a model is tuned using resampling, the number of model fits can become large as the number of tuning combinations increases. To reduce the training time, parallel processing can be used. For example, to train the support vector machine model in Section 4, each of the 5 candidate models was fit to 200 separate bootstrap samples. Since each bootstrap sample is independent of the other, these 1,000 models could be computed in parallel (this is also true for the different flavors of cross-validation).

NetWorkSpaces (Scientific Computing Associates, Inc. 2007) is a software system that facilitates parallel processing when multiple processors are available. A sister R package to **caret**, called **caretNWS**, uses **NetWorkSpaces** to build multiple models simultaneously. When a candidate model is resampled during parameter tuning, the resampled datasets are sent in roughly equal sized batches to different "workers," which could be processors within a single machine or across computers. Once their models are built, the results are returned to the original R session. **NetWorkSpaces** is available in R using the **nws** package and can be used across many platforms.

There is a large degree of syntactical similarity between the **caret** and **caretNWS** packages. The former uses the `train` function to build the model and the latter uses `trainNWS`. Almost all of the other syntax is the same. For example, to fit the SVM model from Section 4, we could use:

```
R> set.seed(2)
R> svmFit <- trainNWS(trainDescr, trainClass,
+    method = "svmRadial", tuneLength = 5, scaled = FALSE)
```

Recall that the `tuneLength` parameter sets the size of the search grid. Due to time constraints, the default number of bootstrap samples (25) was used for model training. Each of the 25 sets of bootstrap samples was split across 5 processors.

To characterize the benefit of using parallel processing in this manner, the previous support vector machine and boosted tree models were refit using multiple processors. Additionally, a partial least squares classification model was also fit using **caret**'s `plsda` function, where up to 40 components were evaluated. For each of these three models, the execution times were recorded when utilizing $P = 1, 2, 3, 4, 5, 10, 15$ and 20 processors on a single AMD Opteron system containing eight quad-core chips using Red Hat Linux (version 2.6.9).

One common metric used to assess the efficacy of parallelization is $speedup = T_{seq}/T_{par}$, where T_{seq} and T_{par} denote the execution times to train the model serially and in parallel,
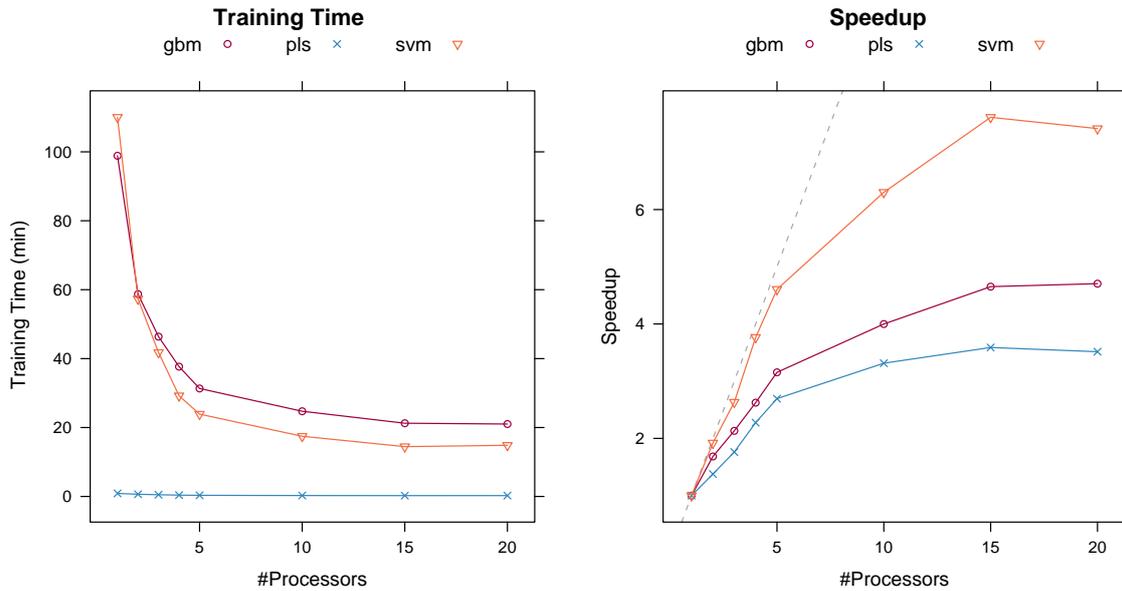
Figure 5: Training time profiles using parallel processing via `trainNWS`. The left panel shows the elapsed time to train various types of models using single or multiple processors. The panel on the right shows the "speedup," defined to be the time for serial execution divided by the parallel execution time. The reference line shows the maximum theortical speedup.

respectively. Excluding systems with sophisticated shared memory capabilities, the maximum possible speedup attained by parallelization with P processors is equal to P (Amdahl 1967). Factors affecting the speedup include the overhead of starting the parallel workers, data transfer, the percentage of the algorithm's computations that can be done in parallel, etc.

Figure 5 shows the results of the benchmarking study. In the left panel, the actual training time for each of the models is shown. Irrespective of the number of processors used, the PLS model is much more efficient than the other models. This is most likely due to PLS solving straight-forward, well optimized linear equations. Unfortunately, partial least squares produces linear boundaries which may not be flexible enough for some problems. For the support vector machine and boosted tree models, the rate of decrease in training time appears to slow after 15 processors.

On the right-hand panel, the speedup is plotted. For each model, there is a decrease in the training time as more nodes are added, although there was little benefit of adding more than 15 workers. The support vector machine comes the closest to the theoretical speedup boundary when five or less workers are used. Between 5 and 15 workers, there is an additional speedup, but at a loss of efficiency. After 15 workers, there is a negligible speedup effect. For boosted trees, the efficiency of adding parallel workers was low, but there was more than a four-fold speedup obtained by using more than 10 workers. Although PLS benefits from parallel processing, it does not show significant gains in training time and efficiency. Recall from Section 4 that boosted trees and partial least squares exploit the use of sub-models to efficiently derive predictions for some of the combinations of tuning parameters. Support vector machines are not able to take advantage of sub-models, which is probably one factor

related to why it benefits more from parallel processing than the other models.

One downside to parallel processing in this manner is that the dataset is held in memory for every node used to train the model. For example, if `trainNWS` is used to compute the results from 50 bootstrap samples using P processors, P data sets are held in memory. For large datasets, this can become a problem if the additional processors are on the same machines where they are competing for the same physical memory. In the future, this might be resolved using specialized software that exploits systems with a shared memory architecture.

More research is needed to determine when it is advantageous to parallel process, given the type of model and the dimensions of the training set.

11. Summary

An R package has been described that can be used to facilitate the predictive modeling process. A wide variety of functions have been described, ranging from data splitting and pre-processing methods to tools for estimating performance and tuning models. Providing a unified interface to different models is a a major theme of the package. Currently, `caret` works with a large number of existing modeling functions and will continue to add new models as they are developed.

Computational details

All computations and graphics in this paper have been obtained using R version 2.6.1 (R Development Core Team 2008) using the packages: **ada** 2.0-1 (Culp *et al.* 2007), **caret** 3.45, **caret-NWS** 0.23, **class** 7.2-42 (Venables and Ripley 1999), **e1071** 1.5-18 (Dimitriadou *et al.* 2008), **earth** 2.0-2 (Milborrow 2007), **elasticnet** 1.02 (Zou and Hastie 2005), **gbm** 1.6-3 (Ridgeway 2003), **gpls** 1.3.1 (Ding 2004), **ipred** 0.8-5 (Peters *et al.* 2002), **kernlab** 0.9-5 (Karatzoglou *et al.* 2004), **klaR** 0.5-6 (Weihs *et al.* 2005), **MASS** 7.2-42 (Venables and Ripley 1999), **mboost** 1.0-2 (Hothorn and Bühlmann 2007), **mda** 0.3-2 (Hastie and Tibshirani 1998), **nnet** 7.2-42 (Venables and Ripley 1999), **nws** 1.7.1.0 (Scientific Computing Associates, Inc. 2007), **pamr** 1.31 (Hastie *et al.* 2003), **party** 0.9-96 (Hothorn *et al.* 2006), **pls** 2.1-0 (Mevik and Wehrens 2007), **randomForest** 4.5-25 (Liaw and Wiener 2002), **rpart** 3.1-39 (Therneau and Atkinson 1997) and **SDDA** 1.0-3 (Stone 2008).

Acknowledgments

References

Ambroise C, McLachlan G (2002). "Selection Bias in Gene Extraction on the Basis of Microarray Gene–Expression Data." *Proceedings of the National Academy of Sciences*, **99**, 6562–6566. URL http://www.pnas.org/content/99/10/6562.abstract.

Amdahl GM (1967). "Validity of the Single-Processor Approach To Achieving Large Scale Computing Capabilities." In "American Federation of Information Processing Societies Conference Proceedings," volume 30, pp. 483–485.

Ames BN, Gurney EG, Miller JA, Bartsch H (1972). "Carcinogens as Frameshift Mutagens: Metabolites and Derivatives of 2–acetylaminofluorene and other Aromatic Amine Carcinogens." *Proceedings of the National Academy of Sciences*, **69**(11), 3128–3132. doi:10.1073/pnas.69.11.3128.

Ayres I (2007). *Super Crunchers*. Bantam. URL http://www.randomhouse.com/bantamdell/supercrunchers/.

Barker M, Rayens W (2003). "Partial Least Squares for Discrimination." *Journal of Chemometrics*, **17**(3), 166–173.

Bühlmann P, Hothorn T (2007). "Boosting Algorithms: Regularization, Prediction and Model Fitting." *Statistical Science*, **22**(4), 477–505. doi:10.1214/07-STS242.

Caputo B, Sim K, Furesjo F, Smola A (2002). "Appearance-Based Object Recognition Using SVMs: Which Kernel Should I Use?" *Proceedings of Neural Information Processing Systems Workshop on Statistical methods for Computational Experiments In Visual Processing and Computer Vision*.

Culp M, Johnson K, Michailides G (2007). "**ada**: An R Package for Stochastic Boosting." *Journal of Statistical Software*, **17**(2), 1–27. URL http://www.jstatsoft.org/v17/i02/.

Dimitriadou E, Hornik K, Leisch F, Meyer D, Weingessel A (2008). *e1071: Misc Functions of the Department of Statistics (e1071), TU Wien*. R package version 1.5-18, URL http://CRAN.R-project.org/package=e1071.

Ding B (2004). *gpls: Classification Using Generalized Partial Least Squares*. R package version 1.3.1, URL http://CRAN.R-project.org/package=gpls.

Hastie T, Tibshirani R (1998). *mda: Mixture and Flexible Discriminant Analysis*. R package version 0.3-2, URL http://CRAN.R-project.org/package=mda.

Hastie T, Tibshirani R, Friedman JH (2001). *The Elements of Statistical Learning*. Springer-Verlag, New York. URL http://www-stat.stanford.edu/~tibs/ElemStatLearn/.

Hastie T, Tibshirani R, Narasimhan B, Chu G (2003). *pamr: Pam – Prediction Analysis for Microarrays*. R package version 1.31, URL http://CRAN.R-project.org/package=pamr.

Hothorn T, Bühlmann P (2007). *mboost: Model-Based Boosting*. R package version 1.0-2, URL http://CRAN.R-project.org/package=mboost.

Hothorn T, Hornik K, Zeileis A (2006). "Unbiased Recursive Partitioning: A Conditional Inference Framework." *Journal of Computational and Graphical Statistics*, **15**(3), 651–674.

Karatzoglou A, Smola A, Hornik K, Zeileis A (2004). "**kernlab** – An S4 Package for Kernel Methods in R." *Journal of Statistical Software*, **11**(9). URL http://www.jstatsoft.org/v11/i09/.

Kazius J, McGuire R, Bursi R (2005). "Derivation and Validation of Toxicophores for Mutagenicity Prediction." *Journal of Medicinal Chemistry*, **48**(1), 312–320. doi:10.1021/jm040835a.

Leach A, Gillet V (2003). *An Introduction to Chemoinformatics*. Kluwer Academic Publishers.

Liaw A, Wiener M (2002). "Classification and Regression by **randomForest**." R *News*, **2**(3), 18–22. URL http://CRAN.R-project.org/doc/Rnews/.

Mevik BH, Wehrens R (2007). "The **pls** Package: Principal Component and Partial Least Squares Regression in R." *Journal of Statistical Software*, **18**(2). URL http://www.jstatsoft.org/v18/i02/.

Milborrow S (2007). ***earth**: Multivariate Adaptive Regression Spline Models*. R package version 2.0-2, URL http://CRAN.R-project.org/package=earth.

Peters A, Hothorn T, Lausen B (2002). "**ipred**: Improved Predictors." R *News*, **2**(2), 33–36. URL http://CRAN.R-project.org/doc/Rnews/.

R Development Core Team (2008). R: *A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL http://www.R-project.org/.

Ridgeway G (2003). ***gbm**: Generalized Boosted Regression Models*. R package version 1.6-3, URL http://CRAN.R-project.org/package=gbm.

Ridgeway G (2007). "Generalized Boosted Models: A Guide to the **gbm** Package." R package vignette, URL http://CRAN.R-project.org/package=gbm.

Sarkar D (2008). ***lattice**: Multivariate Data Visualization with* R. Springer-Verlag, New York.

Scientific Computing Associates, Inc (2007). ***NetWorkSpacs** for* R *User Guide*. R package version 1.7.1.0, URL http://CRAN.R-project.org/package=nws.

Serneels S, DeNolf E, VanEspen P (2006). "Spatial Sign Preprocessing: A Simple Way To Impart Moderate Robustness to Multivariate Estimators." *Journal of Chemical Information and Modeling*, **46**(3), 1402–1409. doi:10.1021/ci050498u.

Stone G (2008). ***SDDA**: Stepwise Diagonal Discriminant Analysis*. R package version 1.0-3, URL http://CRAN.R-project.org/package=SDDA.

Talete SRL (2007). ***DRAGON** for Windows and Linux*. URL http://www.talete.mi.it/.

Therneau TM, Atkinson EJ (1997). "An Introduction to Recursive Partitioning Using the **rpart** Routine." URL http://www.mayo.edu/hsr/techrpt/61.pdf.

Venables B, Ripley B (1999). ***VR**: Bundle of **MASS**, **class**, **nnet**, **spatial***. R package version 7.2-42, URL http://CRAN.R-project.org/package=VR.

Venables WN, Ripley BD (2002). *Modern Applied Statistics with* S. 4th edition. Springer-Verlag, New York.

Weihs C, Ligges U, Luebke K, Raabe N (2005). "**klaR**: Analyzing German Business Cycles." In D Baier, R Decker, L Schmidt-Thieme (eds.), "Data Analysis and Decision Support," pp. 335–343. Springer-Verlag, Berlin.

Willett P (1999). "Dissimilarity-Based Algorithms for Selecting Structurally Diverse Sets of Compounds." *Journal of Computational Biology*, **6**(3-4), 447–457. doi:10.1089/106652799318382.

Zou H, Hastie T (2005). *elasticnet: Elastic-Net for Sparse Estimation and Sparse PCA*. R package version 1.02, URL http://CRAN.R-project.org/package=elasticnet.

Affiliation:

Max Kuhn
Nonclinical Statistics
Pfizer Global R&D
Groton, CT, United States of America
E-mail: Max.Kuhn@Pfizer.com