The `caret` Package

Max Kuhn
max.kuhn@pfizer.com

February 17, 2012

Contents

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: ada, affy, Boruta, bst, caTools, class, Cubist, e1071, earth ($\geq$ 2.2-3), elasticnet, ellipse, evtree, fastICA, foba, gam, GAMens ($\geq$ 1.1.1), gbm, glmnet ($\geq$ 1.7.1), gpls, grid, hda, HDclassif, Hmisc, ipred, kernlab, klaR, kohonen, lars, leaps, LogicForest, logicFS, LogicReg, MASS, mboost, mda, mgcv, mlbench, neuralnet, nnet, nodeHarvest, obliqueRF, pamr, partDSA, party ($\geq$ 0.9-99992), penalized, penalizedLDA, pls, pROC, proxy, qrnn, quantregForest, randomForest, RANN, rda, relaxo, rFerns, rocc, rpart, rrcov, rrlda, RSNNS, RWeka ($\geq$ 0.4-1), sda, SDDA, sparseLDA ($\geq$ 0.1-1), spls, stepPlr, superpc, vbmp. 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.

1 Model Training and Parameter Tuning

caret has several functions that attempt to streamline the model building and evaluation process.

The caret function can be used to

- evaluate, using resampling, the effect of model tuning parameters on performance
- choose the "optimal" model across these parameters

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

- estimate model performance from a training set

More formally:

```
1  Define sets of model parameter values to evaluate
2  for each parameter set do
3  |  for each resampling iteration do
4  |  |  Hold–out specific samples
5  |  |  [Optional] Pre–process the data
6  |  |  Fit the model on the remainder
7  |  |  Predict the hold–out samples
8  |  end
9  |  Calculate the average performance across hold–out predictions
10 end
11 Determine the optimal parameter set
12 Fit the final model to all the training data using the optimal parameter set
```

First, a specific model must be chosen. Currently, 136 are available using caret; see Tables 1, 2 and 3 for details.

In Tables 1, 2 and 3, there are lists of tuning parameters that can potentially be optimized. The first step in tuning the model (line 1 in Algorithm 1) is to choose a set of parameters to evaluate. For example, if fitting a Partial Least Squares (PLS) model, the number of PLS components to evaluate must be specified.

Once the model and tuning parameter values have been defined, the type of resampling should be also be specified. Currently, k–fold cross–validation (once or repeated), leave–one–out cross–validation and bootstrap (simple estimation or the 632 rule) resampling methods can be used by caret. After resampling, the process produces a profile of performance measures is available to guide the user as to which tuning parameter values should be chosen. By default, the function automatically chooses the tuning parameters associated with the best value, although different algorithms can be used (see Section 2.5).

1.1 An Example

As an example, the multidrug resistance reversal (MDRR) agent data is used to determine a predictive model for the "ability of a compound to reverse a leukemia cell's resistance to adriamycin" (Svetnik et al, 2003). For each sample (i.e. compound), predictors are calculated that reflect characteristics of the molecular structure. These molecular descriptors are then used to predict assay results that reflect resistance.

The data are accessed using `data(mdrr)`. This creates a data frame of predictors called `mdrrDescr` and a factor vector with the observed class called `mdrrClass`.

To start, we will:

- use unsupervised filters to remove predictors with unattractive characteristics (e.g. spare distributions or high inter–predictor correlations)
- split the entire data set into a training and test set

See the package vignette "caret Manual – Data and Functions" for more details about these operations.

```
> print(ncol(mdrrDescr))
```

```
[1] 342
```

```
> nzv <- nearZeroVar(mdrrDescr)
> filteredDescr <- mdrrDescr[, -nzv]
> print(ncol(filteredDescr))
```

```
[1] 297
```

```
> descrCor <- cor(filteredDescr)
> highlyCorDescr <- findCorrelation(descrCor, cutoff = .75)
> filteredDescr <- filteredDescr[,-highlyCorDescr]
> print(ncol(filteredDescr))
```

```
[1] 50
```

```
> set.seed(1)
> inTrain <- sample(seq(along = mdrrClass), length(mdrrClass)/2)
> trainDescr <- filteredDescr[inTrain,]
> testDescr <- filteredDescr[-inTrain,]
> trainMDRR <- mdrrClass[inTrain]
> testMDRR <- mdrrClass[-inTrain]
> print(length(trainMDRR))
```

```
[1] 264
```

```
> print(length(testMDRR))
```

```
[1] 264
```

1.2 Basic Parameter Tuning

By default, simple bootstrap resampling is used for line 4 in Algorithm 1. Others are availible, such as repeated K–fold cross–validation. The function `trainControl` can be used to specifiy the type of resampling:

```
> fitControl <- trainControl(## 10-fold CV
+                            method = "repeatedcv",
+                            number = 10,
+                            ## repeated three times
+                            repeats = 3,
+                            ## Save all the resampling results
+                            returnResamp = "all")
```

More information about `trainControl` is given in Section 2.3.

The first two arguments to caret are the predictor and outcome data objects, respectively. The third argument, `method`, specifies the type of model (see Tables 1, 2 and 3). We will fit a boosted tree model via the gbm package. The basic syntax for fitting this model using repeated cross–vlaidation is shown below:

```
> gbmFit1 <- train(trainDescr, trainMDRR,
+                  method = "gbm",
+                  trControl = fitControl,
+                  ## This last option is actually one
+                  ## for gbm() that passes through
+                  verbose = FALSE)
> gbmFit1

264 samples
 50 predictors
  2 classes: 'Active', 'Inactive'

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

Summary of sample sizes: 238, 237, 238, 238, 237, 237, ...

Resampling results across tuning parameters:

  interaction.depth  n.trees  Accuracy  Kappa  Accuracy SD  Kappa SD
  1                  50       0.813     0.614  0.0753       0.157
  1                  100      0.817     0.624  0.0606       0.124
  1                  150      0.814     0.619  0.0691       0.141
  2                  50       0.813     0.617  0.066        0.135
  2                  100      0.818     0.627  0.0717       0.146
  2                  150      0.808     0.607  0.0679       0.138
  3                  50       0.815     0.618  0.0606       0.128
  3                  100      0.812     0.614  0.0622       0.127
  3                  150      0.814     0.619  0.0582       0.118

Tuning parameter 'shrinkage' was held constant at a value of 0.1
Accuracy was used to select the optimal model using  the largest value.
The final values used for the model were interaction.depth = 2, n.trees =
 100 and shrinkage = 0.1.
```

For a gradient boosting machine (GBM) model, there are three main tuning parameters:

- number of iterations, *i.e. trees*, (called `n.trees` in the `gbm` function)
- complexity of the tree, called `interaction.depth`
- learning rate: how quickly the algorithm adapts, called `shrinkage`

The default values tested for this model are shown in the first two columns (`shrinkage` is not shown beause the grid set of candidate models all use a value of 0.1 for this tuning parameter). The column labeled "`Accuracy`" is the overall agreement rate averaged over cross–validation iterations. The agreement standard deviation is also calculated from the cross-validation results. The column "`Kappa`" is Cohen's (unweighted) Kappa statistic averaged across the resampling results. caret works with specific models (see Tables 1, 2 and 3). For these models, caret can automatically create a grid of tuning parameters. By default, if p is the number of tuning parameters, the grid size is 3^p. For example, regularized discriminant analysis (RDA) models have two parameters (`gamma` and `lambda`), both of which lie on $[0, 1]$. The default training grid would produce nine combinations in this two–dimensional space.

1.3 Notes

- There is a formula interface (e.g. `train(y~., data = someData)` that can be used. One of the issues with a large number of predictors is that the objects related to the formula which are saved can get very large. In these cases, it is best to stick with the non–formula interface described above.
- The function determines the type of problem (classification or regression) from the type of the response given in the `y` argument.
- The ... option can be used to pass parameters to the fitting function. For example, in random forest models, you can specify the number of trees to be used in the call to caret.
- For regression models (i.e. a numeric outcome), a similar table would be produced showing the average root mean squared error and average R^2 value statistic across tuning parameters, otherwise known as Q^2 (see the note below related to this calculation). For regression models, the classical R^2 statistic cannot be compared between models that contain an intercept and models that do not. Also, some models do not have an intercept only null model.

  To approximate this statistic across different types of models, the square of the correlation between the observed and predicted outcomes is used. This means that the R^2 values produced by caret will not match the results of `lm` and other functions.

Also, the correlation estimate does not take into account the degrees of freedom in a model and thus does not penalize models with more parameters. For some models (e.g random forests or on–linear support vector machines) there is no clear sense of the degrees of freedom, so this information cannot be used in R^2 if we would like to compare different models.

- The nearest shrunken centroid model of Tibshirani et al (2003) is specified using `method = "pam"`. For this model, there must be at least two samples in each class. caret will ignore classes where there are less than two samples per class from every model fit during bootstrapping or cross–validation (this model only).

- 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 the details below).

- For multivariate adaptive regression splines (MARS), the `earth` package is used with a model type of `mars` or `earth` is requested. The tuning parameters used by caret are `degree` and `nprune`. The parameter `nk` is not automatically specified and, if not specified, the default in the `earth` function is used.

  For example, suppose a training set with 40 predictors is used with `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. Also, using `method = 'gcvEarth'` will use the basic GCV pruning procedure and only tune the degree.

  Also, there may be cases where the message such as "specified 'nprune' 29 is greater than the number of available model terms 24, forcing 'nprune' to 24" show up after the model fit. This can occur since the `earth` function may not actually use the number of terms in the initial model as specified by `nk`. This may be because the `earth` function removes terms with linear dependencies and the forward pass counts as if terms were added in pairs (although singleton terms may be used). By default, the caret function fits and initial MARS model is used to determine the number of possible terms in the training set to create the tuning grid. Resampled data sets may produce slightly different models that do not have as many possible values of `nprune`.

- For the `glmboost` and `gamboost` functions from the `mboost` package, 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 the `mboost` package vignette for more details about AIC pruning.

- The partitioning model of Molinaro *et al.* (2010) has a tuning parameter that is the number of partitions in the model. The R function `partDSA` has the argument `cut.off.growth` which is described as "the maximum number of terminal partitions to be considered when building the model." Since this is the maximum, the user might ask for a model with X partitions but the model can only predict $Y < X$. In these cases, the model predictions will be based on the largest model available (Y).

- For generalized additive models, a formula is generated from the data. First, predictors with degenerate distributions are excluded (via the `nearZeroVar` function). Then, the number of distinct values for each predictor is calculated. If this value is greater than 10, the predictor is entered into the formula via a smoothed term (otherwise a linear term is used). For models in the `gam` package, the smooth terms have the same amount of smoothing applied to them (i.e. equal `df` or `span` across all the smoothed predictors).

- For some models (blackboost, bstTree, bstLs, bstSm, cubist, earth, enet, foba, gamboost, gbm, glmboost, glmnet, kernelpls, lars, lars2, lasso, leapForward, leapBackward, leapSeq, logitBoost, pam, partDSA, pcr, PenalizedLDA, pls, relaxo, rpart2, rpart, scrda, simpls, superpc, widekernelpls), the caret function will fit a model that can be used to derive predictions for some sub-models. For example, for MARS (via the `earth` function), for a fixed degree, a model with a maximum number of terms will be fit and the predictions of all of the requested models with the same degree and smaller number of terms will be computed using `update.earth` instead of fitting a new model. When the `verboseIter` option of the `trainControl` function is used, a line is printed for the "top–level" model (instead of each model in the tuning grid).

- There are `print` and `plot` methods for the `train` class. The `plot` method visualizes the profile of average resampled performance values across the different tuning parameters using scatter plots or level plots. See Figures 1 and 2 for examples. Functions that visualize the individual resampling results for caret objects are discussed in Section 5.1.

- Using the first set of tuning parameters that are optimal (in the sense of accuracy or mean squared error), caret automatically fits a model with these parameters to the entire training data set. That model object is accessible in the `finalModel` object within caret. For example, `gbmFit$finalModel` is the same object that would have been produced using a direct call to the `gbm` function with the final tuning parameters.

There is additional functionality in caret that is described in the next section.

Table 1: Dual use models (regression and classification) available in train.

Type	method	Package	Parameters
Generalized linear model			
	glm	stats	None
	glmStepAIC	MASS	None
Generalized additive model			
	gam	mgcv	select, method
	gamLoess	gam	degree, span
	gamSpline	gam	df
Recursive Partitioning			
	rpart	rpart	cp
	rpart2	rpart	maxdepth
	ctree	party	mincriterion
	ctree2	party	maxdepth
Boosted Trees			
	gbm	gbm	interaction.depth, shrinkage, n.trees
	blackboost	mboost	maxdepth, mstop
	ada	ada	iter, maxdepth, nu
	bstTree	bst	maxdepth, nu, mstop
Other Boosted Models			
	glmboost	mboost	mstop, prune
	gamboost	mboost	mstop, prune
	bstLs	bst	mstop, nu
	bstSm	bst	nu, mstop
Random Forests			
	rf	randomForest	mtry
	parRF	randomForest	mtry
	cforest	party	mtry
	Boruta	Boruta	mtry
Bagging			
	treebag	ipred	None
	bag	caret	vars
	logicBag	logicFS	ntrees, nleaves
	bagEarth	caret	degree, nprune

Table 1: *(continued)*

Type	method	Package	Parameters
Other Trees			
	nodeHarvest	nodeHarvest	maxinter, mode
	partDSA	partDSA	cut.off.growth, MPD
Multivariate Adaptive Regression Spline			
	earth	earth	nprune, degree
	gcvEarth	earth	degree
Logic Regression			
	logreg	LogicReg	treesize, ntrees

Table 1: *(continued)*

Type	method	Package	Parameters
Elastic Net			
	glmnet	glmnet	lambda, alpha
Neural Networks			
	nnet	nnet	decay, size
	mlp	RSNNS	size
	mlpWeightDecay	RSNNS	size, decay
	pcaNNet	caret	decay, size
	avNNet	caret	decay, size, bag
Radial Basis Function Networks			
	rbf	RSNNS	size
Partial Least Squares			
	pls	pls	ncomp
	kernelpls	pls	ncomp
	simpls	pls	ncomp
	widekernelpls	pls	ncomp
Sparse Partial Least Squares			
	spls	spls	eta, K, kappa
Support Vector Machines			
	svmLinear	kernlab	C
	svmRadial	kernlab	C, sigma
	svmRadialCost	kernlab	C
	svmPoly	kernlab	degree, scale, C
Gaussian Processes			
	gausssprLinear	kernlab	None
	gausssprRadial	kernlab	sigma
	gausssprPoly	kernlab	scale, degree
K Nearest Neighbor			
	knn	caret	k
Self-Organizing Maps			
	xyf	kohonen	xdim, ydim, topo, xweight
	bdk	kohonen	ydim, xweight, xdim, topo

Table 2: Regression only models available in train.

Type	method	Package	Parameters
Linear Least Squares			
	lm	stats	None
	lmStepAIC	MASS	None
	leapForward	leaps	nvmax
	leapBackward	leaps	nvmax
	leapSeq	leaps	nvmax
Principal Component Regression			
	pcr	pls	ncomp
Independent Component Regression			
	icr	caret	n.comp
Robust Linear Regression			
	rlm	MASS	None
Neural Networks			
	neuralnet	neuralnet	layer2, layer3, layer1
Quantile Regression Forests			
	qrf	quantregForest	mtry
Quantile Regression Neural Networks			
	qrnn	qrnn	n.hidden, penalty, bag
Rule–Based Models			
	M5Rules	RWeka	smoothed, pruned
	M5	RWeka	smoothed, rules, pruned
	cubist	Cubist	neighbors, committees
Projection Pursuit Regression			
	ppr	stats	nterms
Penalized Linear Models			
	penalized	penalized	lambda2, lambda1
	ridge	elasticnet	lambda
	lars	lars	fraction
	lars2	lars	step
	enet	elasticnet	lambda, fraction
	lasso	elasticnet	fraction
	foba	foba	lambda, k

Table 2: *(continued)*

Type	method	Package	Parameters
Relevance Vector Machines			
	`rvmLinear`	`kernlab`	None
	`rvmRadial`	`kernlab`	`sigma`
	`rvmPoly`	`kernlab`	`scale`, `degree`
Supervised Principal Components			
	`superpc`	`superpc`	`threshold`, `n.components`

Table 3: Classification only models available in `train`.

Type	method	Package	Parameters
Linear Discriminant Analysis			
	`lda`	`MASS`	None
	`Linda`	`rrcov`	None
Quadratic Discriminant Analysis			
	`qda`	`MASS`	None
	`QdaCov`	`rrcov`	None
Stabilized Linear Discriminant Analysis			
	`slda`	`ipred`	None
Heteroscedastic Discriminant Analysis			
	`hda`	`hda`	`newdim`, `lambda`, `gamma`
Shrinkage Linear Discriminant Analysis			
	`sda`	`sda`	`diagonal`
Sparse Linear Discriminant Analysis			
	`sparseLDA`	`sparseLDA`	`lambda`, `NumVars`
	`PenalizedLDA`	`penalizedLDA`	`K`, `lambda`
Stepwise Discriminant			
	`stepLDA`	`klaR`	`direction`, `maxvar`
Stepwise Diagonal Discriminant Analysis			
	`sddaLDA`	`SDDA`	None
	`sddaQDA`	`SDDA`	None
Regularized Discriminant Analysis			
	`rda`	`klaR`	`gamma`, `lambda`
Mixture Discriminant Analysis			
	`mda`	`mda`	`subclasses`
Sparse Mixture Discriminant Analysis			
	`smda`	`sparseLDA`	`R`, `NumVars`, `lambda`
Penalized Discriminant Analysis			
	`pda`	`mda`	`lambda`
	`pda2`	`mda`	`df`
High Dimensional Discriminant Analysis			
	`hdda`	`HDclassif`	`model`, `threshold`

Table 3: *(continued)*

Type	method	Package	Parameters
Flexible Discriminant Analysis (MARS basis)			
	`fda`	`mda`	`degree`, `nprune`
Robust Regularized Linear Discriminant Analysis			
	`rrlda`	`rrlda`	`alpha`, `lambda`
Bagging			
	`bagFDA`	`caret`	`degree`, `nprune`
Logistic/Multinomial Regression			
	`multinom`	`nnet`	`decay`
	`plr`	`stepPlr`	`lambda`, `cp`

Table 3: *(continued)*

Type	method	Package	Parameters
LogitBoost			
	logitBoost	caTools	nIter
Logistic Model Trees			
	LMT	RWeka	iter
Radial Basis Function Networks			
	rbfDDA	RSNNS	negativeThreshold
Rule-Based Models			
	J48	RWeka	C
	OneR	RWeka	None
	PART	RWeka	threshold, pruned
	JRip	RWeka	NumOpt
Random Ferns			
	rFerns	rFerns	depth
Logic Forests			
	logforest	LogForest	None
Bayesian Multinomial Probit Model			
	vbmpRadial	vbmp	estimateTheta
Least Squares Support Vector Machines			
	lssvmRadial	kernlab	sigma
Nearest Shrunken Centroids			
	pam	pamr	threshold
	scrda	rda	alpha, delta
Naive Bayes			
	nb	klaR	fL, usekernel
Generalized Partial Least Squares			
	gpls	gpls	K.prov
Learned Vector Quantization			
	lvq	class	k, size
ROC Curves			
	rocc	rocc	xgenes

2 Customizing the Tuning Process

There are a few ways to customize the process of selecting tuning/complexity parameters and building the final model.

2.1 Pre–Processing Options

As previously mentioned, caret can pre–process the data in various ways prior to model fitting. The caret function `preProcess` is automatically used. This function can be used for centering and scaling, imputation (see details below), applying the spatial sign transformation and feature extraction via principal component analysis or independent component analysis. Options to the `preProcess` function can be passed via the `trainControl` function.

These processing steps would be applied during any predictions generated using `predict.train`, `extractPrediction` or `extractProbs` (see Section 3 later in this document). The pre–processing would **not** be applied to predictions that directly use the `object$finalModel` object.

For imputation, there are two methods currently implemented:

- k–nearest neighbors takes a sample with missing values and finds the k closest samples in the training set. The average of the k training set values for that predictor are used as a substitute for the original data. When calculating the distances to the training set samples, the predictors used in the calculation are the ones with no missing values for that sample and no missing values in the training set.
- another approach is to fit a bagged tree model for each predictor using the training set samples. This is usually a fairly accurate model and can handle missing values. When a predictor for a sample requires imputation, the values for the other predictors are fed through the bagged tree and the prediction is used as the new value. This model can have significant computational cost.

If there are missing values in the training set, PCA and ICA models only use complete samples.

2.2 Alternate Tuning Grids

The tuning parameter grid can be specified by the user. The argument `tuneGrid` can take a data frame with columns for each tuning parameter (see Tables 1, 2 and 3 for specific details). The column names should be the same as the fitting function's arguments with a period preceding the name. For the previously mentioned RDA example, the names would be `.gamma` and `.lambda`. caret will tune the model over each combination of values in the rows.

We can fix the learning rate and evaluate more than three values of `n.trees`:

```
> gbmGrid <-  expand.grid(.interaction.depth = c(1, 3),
+                         .n.trees = c(10, 50, 100, 150, 200, 250, 300),
+                         .shrinkage = 0.1)



> set.seed(3)
> gbmFit2 <- train(trainDescr, trainMDRR,
+                  method = "gbm",
+                  trControl = fitControl,
+                  verbose = FALSE,
+                  ## Now specify the exact models
+                  ## to evaludate:
+                  tuneGrid = gbmGrid)
> gbmFit2

264 samples
 50 predictors
  2 classes: 'Active', 'Inactive'

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

Summary of sample sizes: 237, 237, 237, 237, 238, 237, ...

Resampling results across tuning parameters:

  interaction.depth  n.trees  Accuracy  Kappa  Accuracy SD  Kappa SD
  1                  10       0.762     0.494  0.0614       0.131
  1                  50       0.807     0.601  0.0701       0.146
  1                  100      0.819     0.629  0.0694       0.14
  1                  150      0.815     0.621  0.0729       0.148
  1                  200      0.82      0.633  0.072        0.144
  1                  250      0.81      0.613  0.0728       0.146
  1                  300      0.807     0.608  0.0698       0.138
  3                  10       0.795     0.572  0.0699       0.147
  3                  50       0.824     0.639  0.0635       0.13
  3                  100      0.812     0.615  0.0633       0.129
  3                  150      0.812     0.615  0.0648       0.132
  3                  200      0.818     0.626  0.0681       0.137
  3                  250      0.814     0.618  0.0706       0.143
  3                  300      0.815     0.622  0.0689       0.137

Tuning parameter 'shrinkage' was held constant at a value of 0.1
Accuracy was used to select the optimal model using  the largest value.
The final values used for the model were interaction.depth = 3, n.trees =
 50 and shrinkage = 0.1.
```

2.3 The trainControl Function

The function trainControl generates parameters that further control how models are created, with possible values:

- method: The resampling method: boot, boot632, cv, LOOCV, LGOCV, repeatedcv and oob. The last value, out–of–bag estimates, can only be used by random forest, bagged trees, bagged earth, bagged flexible discriminant analysis, or conditional tree forest models. GBM models are not included (the gbm package maintainer has indicated that it would not be a good idea to choose tuning parameter values based on the model OOB error estimates with boosted trees). Also, for leave–one–out cross–validation, no uncertainty estimates are given for the resampled performance measures.
- number and repeats: number controls with the number of folds in K–fold cross–validation or number of resampling iterations for bootstrapping and leave–group–out cross–validation. repeats applied only to repeated K–fold cross–validation. Suppose that method = "repeatedcv", number = 10 and repeats = 3, then three separate 10–fold cross–validations are used as the resampling scheme.
- verboseIter: A logical for printing a training log.
- returnData: A logical for saving the data into a slot called trainingData.
- p: For leave-group out cross-validation: the training percentage
- classProbs: a logical value determining whether class probabilities should be computed for held–out samples during resample. Examples of using this argument are given in Section 2.4.
- index: a list with elements for each resampling iteration. Each list element is the sample rows used for training at that iteration. When these values are not specified, caret will generate them.
- summaryFunction: a function to compute alternate performance summaries. See Section 2.4 for more details.
- selectionFunction: a function to choose the optimal tuning parameters. See Section 2.5 for more details and examples.
- PCAthresh, ICAcomp and k: these are all options to pass to the preProcess function (when used).
- returnResamp: a character string containing one of the following values: "all", "final" or "none". This specifies how much of the resampled performance measures to save.

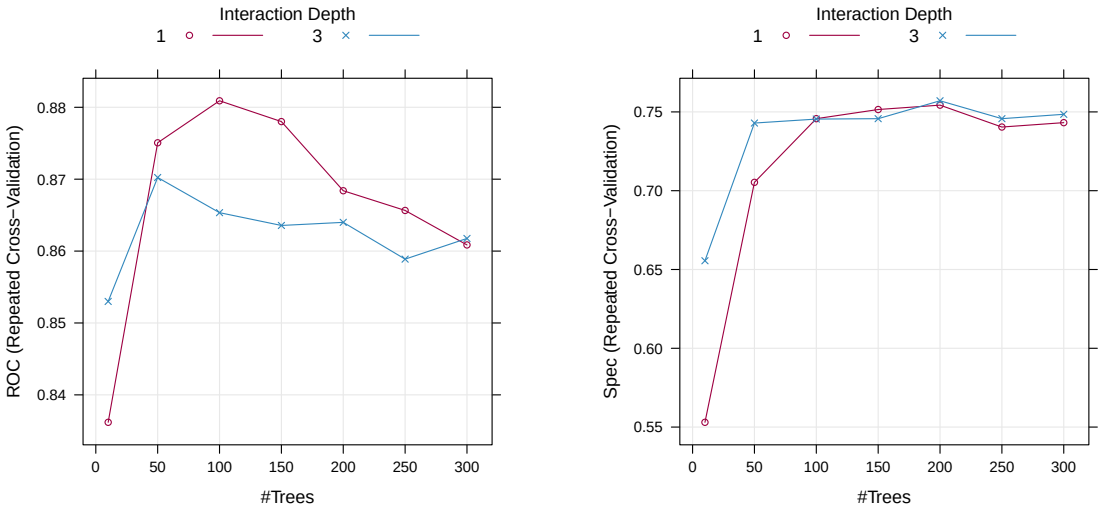

Figure 1: Examples of output from `plot.tain`. **left** a plot produced using `plot(gbmFit3)` showing the relationship between the number of boosting iterations, the interaction depth and the resampled classification accuracy **right** the same plot, but the Kappa statistic is plotted using `plot(gbmFit3, metric = "Kappa")`

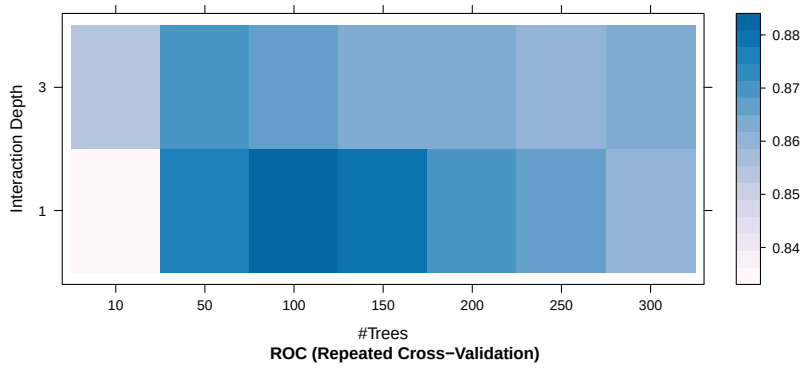

Figure 2: For the boosted tree example in Section 2.2, using `plot(gbmFit metric = "Kappa", plotType = "level")` shows the relationship (using a `levelplot`) between the number of boosting iterations, the interaction depth and the resampled estimate of the Kappa statistic.

2.4 Alternate Performance Metrics

The user can change the metric used to determine the best settings. By default, RMSE and R^2 are computed for regression while accuracy and Kappa are computed for classification. Also by default, the parameter values are chosen using RMSE and accuracy, respectively for regression and classification. The `metric` argument of the caret function allows the user to control which the optimality criterion is used. For example, in problems where there are a low percentage of samples in one class, using `metric = "Kappa"` can improve quality of the final model.

If none of these parameters are satisfactory, the user can also compute custom performance metrics. The `trainControl` function has a argument called `summaryFunction` that specifies a function for computing performance. The function should have these arguments:

- `data` is a reference for a data frame or matrix with columns called `obs` and `pred` for the observed and predicted outcome values (either numeric data for regression or character values for classification). Currently, class probabilities are not passed to the function. The values in data are the held–out predictions (and their associated reference values) for a single combination of tuning parameters. If the `classProbs` argument of the `trainControl` object is set to `TRUE`, additional columns in `data` will be present that contains the class probabilities. The names of these columns are the same as the class levels.
- `lev` is a character string that has the outcome factor levels taken from the training data. For regression, a value of `NULL` is passed into the function.
- `model` is a character string for the model being used (i.e. the value passed to the `method` value of caret).

The output to the function should be a vector of numeric summary metrics with non–null names. By default, caret evaluate classification models in terms of the predicted classes. Optionally, class probabilities can also be used to measure performance. To obtain predicted class probabilities within the resampling process, the argument `classProbs` in `trainControl` must be set to `TRUE`. This merges columns of probabilities into the predictions generated from each resample (there is a column per class and the column names are the class names).

As shown in the last section, custom functions can be used to calculate performance scores that are averaged over the resamples. Another built–in function, `twoClassSummary`, will compute the sensitivity, specificity and area under the ROC curve.

```
> twoClassSummary

function (data, lev = NULL, model = NULL)
{
    require(pROC)
    if (!all(levels(data[, "pred"]) == levels(data[, "obs"])))
        stop("levels of observed and predicted data do not match")
    rocObject <- pROC:::roc(data$obs, data[, lev[1]])
    out <- c(rocObject$auc, sensitivity(data[, "pred"], data[,
        "obs"], lev[1]), specificity(data[, "pred"], data[, "obs"],
        lev[2]))
    names(out) <- c("ROC", "Sens", "Spec")
    out
}
<environment: namespace:caret>
```

To rebuild the boosted tree model using this criterion, we can see the relationship between the tuning parameters and the area under the ROC curve using the following code:

```
> fitControl <- trainControl(method = "repeatedcv",
+                            number = 10,
+                            repeats = 3,
+                            returnResamp = "all",
+                            ## Estimate class probabilities
+                            classProbs = TRUE,
+                            ## Evaluate performance using
+                            ## the following function
+                            summaryFunction = twoClassSummary)



> set.seed(3)
> gbmFit3 <- train(trainDescr, trainMDRR,
+                  method = "gbm",
+                  trControl = fitControl,
+                  verbose = FALSE,
+                  tuneGrid = gbmGrid,
+                  ## Specify which metric to optimize
+                  metric = "ROC")
> gbmFit3

264 samples
 50 predictors
  2 classes: 'Active', 'Inactive'

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

Summary of sample sizes: 237, 237, 237, 237, 238, 237, ...
```

```
Resampling results across tuning parameters:

  interaction.depth  n.trees  ROC    Sens   Spec   ROC SD  Sens SD  Spec SD
  1                  10       0.836  0.943  0.553  0.0802  0.0717   0.097
  1                  50       0.875  0.88   0.705  0.0533  0.09     0.107
  1                  100      0.881  0.869  0.746  0.0535  0.0965   0.111
  1                  150      0.878  0.869  0.752  0.0635  0.0999   0.108
  1                  200      0.868  0.859  0.754  0.0593  0.105    0.109
  1                  250      0.866  0.839  0.74   0.0649  0.113    0.12
  1                  300      0.861  0.83   0.743  0.0651  0.109    0.115
  3                  10       0.853  0.901  0.656  0.0673  0.0731   0.132
  3                  50       0.87   0.862  0.743  0.0636  0.111    0.101
  3                  100      0.865  0.859  0.745  0.066   0.101    0.113
  3                  150      0.864  0.853  0.746  0.0646  0.109    0.105
  3                  200      0.864  0.848  0.757  0.0649  0.116    0.1
  3                  250      0.859  0.842  0.746  0.0666  0.107    0.106
  3                  300      0.862  0.846  0.748  0.0658  0.116    0.11

Tuning parameter 'shrinkage' was held constant at a value of 0.1
ROC was used to select the optimal model using  the largest value.
The final values used for the model were interaction.depth = 1, n.trees =
 100 and shrinkage = 0.1.
```

In this case, the average area under the ROC curve associated with the optimal tuning parameters was 0.881 across the 30 resamples.

2.5 Choosing the Final Model

Another method for customizing the tuning process is to modify the algorithm that is used to select the "best" parameter values, given the performance numbers. By default, the caret function chooses the model with the largest performance value (or smallest, for mean squared error in regression models). Other schemes for selecting model can be used. Breiman et al (1984) suggested the "one standard error rule" for simple tree–based models. In this case, the model with the best performance value is identified and, using resampling, we can estimate the standard error of performance. The final model used was the simplest model within one standard error of the (empirically) best model. With simple trees this makes sense, since these models will start to over-fit as they become more and more specific to the training data.

caret allows the user to specify alternate rules for selecting the final model. The argument `selectionFunction` can be used to supply a function to algorithmically determine the final model. There are three existing functions in the package: `best` is chooses the largest/smallest value, `oneSE` attempts to capture the spirit of Breiman et al (1984) and `tolerance` selects the least complex model within some percent tolerance of the best value. See `?best` for more details.

User–defined functions can be used, as long as they have the following arguments:

- x is a data frame containing the tune parameters and their associated performance metrics. Each row corresponds to a different tuning parameter combination
- metric a character string indicating which performance metric should be optimized (this is passed in directly from the metric argument of caret.
- maximize is a single logical value indicating whether larger values of the performance metric are better (this is also directly passed from the call to caret).

The function should output a single integer indicating which row in x is chosen.

As an example, if we chose the previous boosted tree model on the basis of overall accuracy (Figure 1), we would choose: interaction depth = 1, n trees = 100, shrinkage = 0.1. However, the scale in this plots is fairly tight, with accuracy values ranging from 0.836 to 0.881. A less complex model (e.g. fewer, more shallow trees) might also yield acceptable accuracy.

The tolerance function could be used to find a less complex model based on $(x - x_{best})/x_{best} \times 100$, which is the percent difference. For example, to select parameter values based on a 2% loss of performance:

```
> whichTwoPct <- tolerance(gbmFit3$results, "ROC", 2, TRUE)
> cat("best model within 2 pct of best:\n")

best model within 2 pct of best:

> gbmFit3$results[whichTwoPct,]

  interaction.depth n.trees shrinkage       ROC Sens     Spec      ROCSD
2                 1      50       0.1 0.8750637 0.88 0.705303 0.05331201
      SensSD    SpecSD
2 0.08998136 0.1071993
```

This indicates that we can get a less complex model with and accuracy of 0.875 (compared to the "pick the best" value of 0.881).

The main issue with these functions is related to ordering the models from simplest to complex. In some cases, this is easy (e.g. simple trees, partial least squares), but in cases such as this model, the ordering of models is subjective. For example, is a boosted tree model using 100 iterations and a tree depth of 2 more complex than one with 50 iterations and a depth of 8? The package makes some choices regarding the orderings. In the case of boosted trees, the package assumes that increasing the number of iterations adds complexity at a faster rate than increasing the tree depth, so models are ordered on the number of iterations then ordered with depth. See ?best for more examples for specific models.

2.6 Parallel Processing

If a model is tuned using resampling, the number of model fits can become large as the number of tuning combinations increases (see the two loops in Algorithm 1). To reduce the training time, parallel processing can be used. For example, to train the gradient boosting machine model in Section 1.2, each of the 9 candidate models was fit to 30 separate resamples. Since each resample is independent of the other, these 270 models could be computed in parallel.

R has several packages that facilitates parallel processing when multiple processors are available (see Schmidberger et al., 2009). caret can be used to build multiple models simultaneously. As of caret version 4.99, a new parallel processing framework is used to increase the computational efficiency. The foreach package allows parallel computations using several different technologies. Although the execution times using foreach is similar to the framework used prior to version 4.99, there are a few advantages:

- the call to `train` (or `rfe` or `sbf`) does not change. Parallel "backends" are registered with foreach prior to the call to `train`
- compared to the kludgy techniques in caret prior to version 4.99, foreach does a much better job of managing memory.
- foreach code can be added in several places in the package and nest parallelism can be used.

For example, to use the multicore package to parallelize the computations, invoking these commands prior to `train` would split the computations into two workers:

```
> library(doMC)
> registerDoMC(2)
```

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, respectively. Excluding systems with sophisticated shared memory capabilities, the maximum possible speedup attained by parallelization with P processors is equal to P. 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 3 shows the results of a benchmarking study run on a 16 core machine. In the left panel, the actual training time for a radial basis function SVM model for a data set with 5,000 samples and 400 predictors. The model was tuned over 10 values of the cost parameter using 50 bootstrap samples. The "new" curve corresponds to the foreach infrastructure.

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 parallelism 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

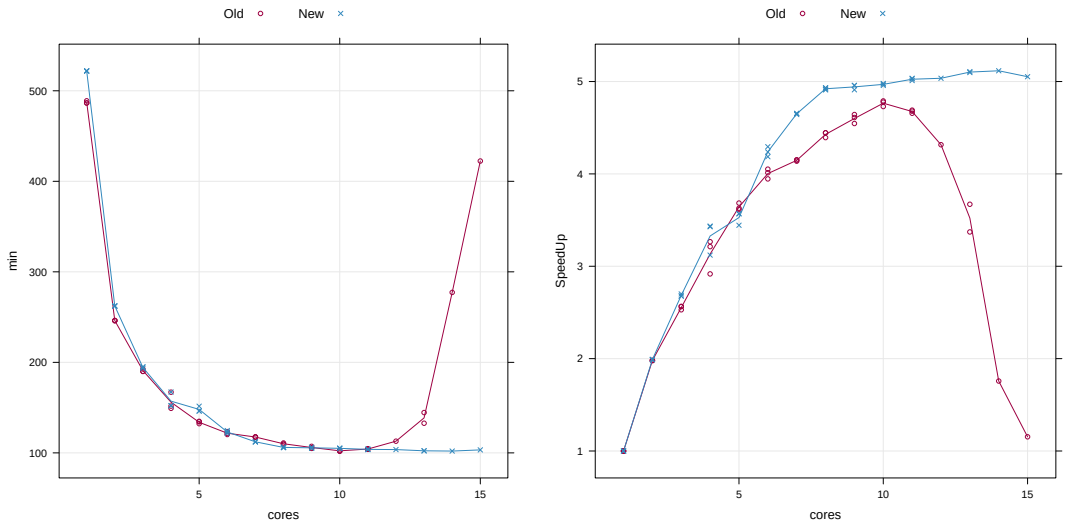

Figure 3: Training time profiles using parallel processing via caret for a benchmarking data set run on a 16 core machine. The left panel shows the elapsed time to train a model using single or multiple processors. The panel on the right shows the "speedup," defined to be the time for serial execution divided by the parallel execution time. The "old" line corresponds to version 4.98 and below while the "new" curve is using foreach.

become a problem if the additional processors are on the same machines where they are competing for the same physical memory. The old codebase starts to slow down around 10 workers due to exhausting the physical memory on the machine. The new codebase does a better job at managing memory with no additional slowdown.

3 Extracting Predictions and Class Probabilities

As previously mentioned, objects produced by the caret function contain the "optimized" model in the `finalModel` sub–object. Predictions can be made from these objects as usual. In some cases, such as `pls` or `gbm` objects, additional parameters from the optimized fit may need to be specified. In these cases, the caret objects uses the results of the parameter optimization to predict new samples.

For example, we can load the Boston Housing data:

```
> library(mlbench)
> data(BostonHousing)
> # we could use the formula interface too
> bhDesignMatrix <-  model.matrix(medv ~. - 1, BostonHousing)
```

split the data into random training/test groups:

```
> set.seed(4)
> inTrain <- createDataPartition(BostonHousing$medv, p = .8, list = FALSE, times = 1)
> trainBH <- bhDesignMatrix[inTrain,]
> testBH <- bhDesignMatrix[-inTrain,]
> trainMedv <- BostonHousing$medv[inTrain]
> testMedv <- BostonHousing$medv[-inTrain]
```

fit partial least squares and multivariate adaptive regression spline models:

```
> set.seed(5)
> plsFit <- train(trainBH, trainMedv,
+                 "pls",
+                 preProcess = c("center", "scale"),
+                 tuneLength = 10,
+                 trControl = trainControl(verboseIter = FALSE,
+                                          returnResamp = "all"))
> set.seed(5)
> marsFit <- train(trainBH, trainMedv,
+                  "earth",
+                  tuneLength = 10,
+                  trControl = trainControl(verboseIter = FALSE,
+                                           returnResamp = "all"))
>
```

To obtain predictions for the MARS model, `predict.earth` can be used.

```
> marsPred1 <- predict(marsFit$finalModel, newdata = testBH)
> head(marsPred1)
```

```
            y
[1,] 34.18241
[2,] 20.90113
[3,] 18.83659
[4,] 14.56850
[5,] 16.44564
[6,] 22.12989
```

Alternatively, `predict.train` can be used to get a vector of predictions for the optimal model only:

```
> marsPred2 <- predict(marsFit, newdata = testBH)
> head(marsPred2)
```

```
[1] 34.18241 20.90113 18.83659 14.56850 16.44564 22.12989
```

Note that the `plsFit` object used pre–processing. In this case, we cannot directly call `predict.mvr` and expect to get the same answers as `predict.train`. The latter function knows that centering

and scaling is required and execute these calculations on the new samples, whereas predict.mvr does not. For the pls function, there is an argument called scale that can be used instead of the pre–processing options in the caret function.

For multiple models, the objects can be grouped using a list and predicted simultaneously:

```
> bhModels <- list(pls = plsFit, mars = marsFit)
> bhPred1 <- predict(bhModels, newdata = testBH)
> str(bhPred1)

List of 2
 $ pls : num [1:99] 30.2 21.9 16.1 16 15.8 ...
 $ mars: num [1:99] 34.2 20.9 18.8 14.6 16.4 ...
```

In some cases,observed outcomes and their associated predictions may be needed for a set of models. In this case, extractPrediction can be used. This function takes a list of models and test and/or unknown samples as inputs and returns a data frame of predictions:

```
> allPred <- extractPrediction(bhModels,
+                              testX = testBH,
+                              testY = testMedv)
> testPred <- subset(allPred, dataType == "Test")
> head(testPred)

     obs     pred model dataType object
408 34.7 30.15640   pls     Test    pls
409 21.7 21.87263   pls     Test    pls
410 20.2 16.06634   pls     Test    pls
411 15.2 16.01122   pls     Test    pls
412 15.6 15.80842   pls     Test    pls
413 14.5 17.94325   pls     Test    pls

> ddply(testPred, .(model), defaultSummary)

  model     RMSE  Rsquared
1 earth 4.605244 0.8016275
2   pls 5.501675 0.7286127

>
```

The output of extractPrediction is a data frame with columns:

- obs, the observed data
- pred, the predicted values from each model
- model, a character string ("rpart", "pls" etc.)

- dataType, a character string for the type of data:
  - "Training" data are the predictions on the training data from the optimal model,
  - "Test" denote the predictions on the test set (if one is specified),
  - "Unknown" data are the predictions on the unknown samples (if specified). Only the predictions are produced for these data. Also, if the quick prediction of the unknowns is the primary goal, the argument unkOnly can be used to only process the unknowns.

Some classification models can produce probabilities for each class. The functions predict.train and predict.list can be used with the type = "probs" argument to produce data frames of class probabilities (with one column per class). Also, the function extractProbs can be used to get these probabilities from one or more models. The results are very similar to what is produced by extractPrediction but with columns for each class. The column pred is still the predicted class from the model.

4 Evaluating Test Sets

A function, postResample, can be used obtain the same performance measures as generated by caret for regression or classification.

4.1 Confusion Matrices

caret also contains several functions that can be used to describe the performance of 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 (i.e. the event of interest), although this can be changed.

The function confusionMatrix can also be used to summarize the results of a classification model:

```
> mdrrPredictions <- extractPrediction(list(gbmFit3), testX = testDescr, testY = testMDRR)
> mdrrPredictions <- mdrrPredictions[mdrrPredictions$dataType == "Test",]
> sensitivity(mdrrPredictions$pred, mdrrPredictions$obs)

[1] 0.7866667

> confusionMatrix(mdrrPredictions$pred, mdrrPredictions$obs)

Confusion Matrix and Statistics

          Reference
Prediction Active Inactive
  Active      118       27
```

```
  Inactive     32       87

               Accuracy : 0.7765
                 95% CI : (0.7214, 0.8253)
    No Information Rate : 0.5682
    P-Value [Acc > NIR] : 1.092e-12

                  Kappa : 0.5469
 Mcnemar's Test P-Value : 0.6025

            Sensitivity : 0.7867
            Specificity : 0.7632
         Pos Pred Value : 0.8138
         Neg Pred Value : 0.7311
             Prevalence : 0.5682
         Detection Rate : 0.4470
   Detection Prevalence : 0.5492

       'Positive' Class : Active
```

The "no–information rate" is the largest proportion of the observed classes (there were more actives than inactives in this test set). A hypothesis test is also computed to evaluate whether the overall accuracy rate is greater than the rate of the largest class. Also, the prevalence of the "positive event" is computed from the data (unless passed in as an argument), the detection rate (the rate of true events also predicted to be events) and the detection prevalence (the prevalence of predicted events).

Suppose a 2×2 table with notation

	Reference	
Predicted	Event	No Event
Event	A	B
No Event	C	D

The formulas used here are:

$$
\begin{aligned}
Sensitivity &= \frac{A}{A+C} \\
Specificity &= \frac{D}{B+D} \\
Prevalence &= \frac{A+C}{A+B+C+D} \\
PPV &= \frac{sensitivity \times prevalence}{((sensitivity \times prevalence) + ((1-specificity) \times (1-prevalence))} \\
NPV &= \frac{specificity \times (1-prevalence)}{((1-sensitivity) \times prevalence) + ((specificity) \times (1-prevalence))} \\
Detection\ Rate &= \frac{A}{A+B+C+D} \\
Detection\ Prevalence &= \frac{A+B}{A+B+C+D}
\end{aligned}
$$

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).

Also, a resampled estimate of the training set can also be obtained using `confusionMatrix.train`. For each resampling iteration, a confusion matrix is created from the hold–out samples and these values can be aggregated to diagnose issues with the model fit.

For example, in the two–class SVM model used in Section 1.2, we could use:

```
> confusionMatrix(gbmFit3)

Cross-Validated (10 fold, repeated 3 times) Confusion Matrix

(entries are percentages of table totals)

          Reference
Prediction Active Inactive
  Active     48.5     12.1
  Inactive    7.6     31.8
```

These values are the percentages that hold–out samples landed in the confusion matrix during resampling. There are several methods for normalizing these values. See `?confusionMatrix.train` for details.

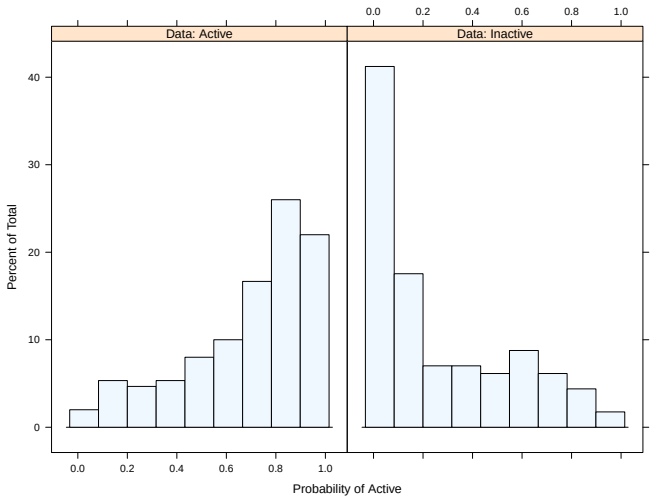

Figure 4: The predicted class probabilities from a gradient boosting machine fit for the MDRR test set. This plot was created using `plotClassProbs(mdrrProbs)`.

Plotting Predictions and Probabilities

Two functions, `plotObsVsPred` and `plotClassProbs`, are interfaces to lattice to plot model results. For regression, `plotObsVsPred` plots the observed versus predicted values by model type and data (e.g. test). See Figure 5 for example. For classification data, `plotObsVsPred` plots the accuracy rates for models/data in a dotplot.

To plot class probabilities, `plotClassProbs` will display the results by model, data and true class (for example, Figure 4).

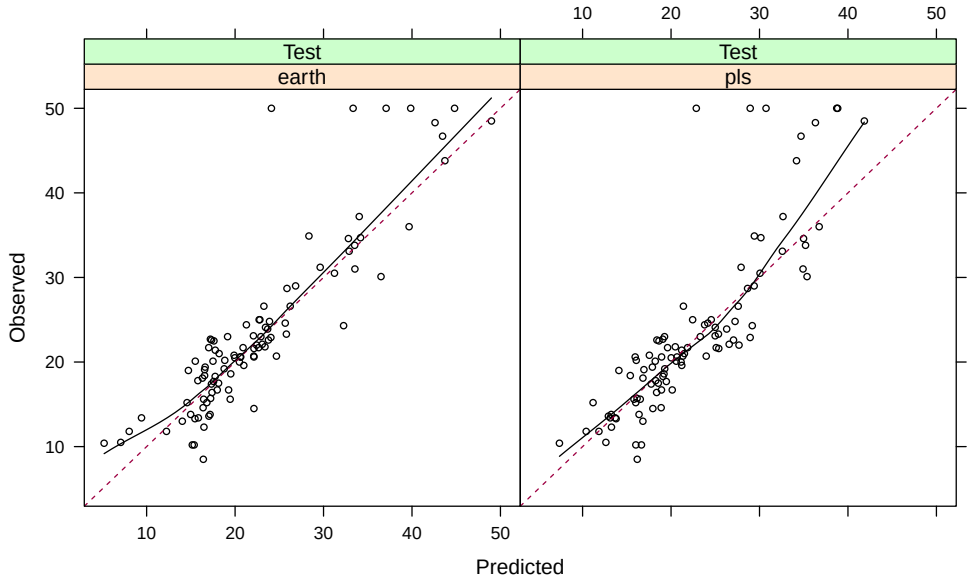

Figure 5: The results of using `plotObsVsPred` to show plots of the observed median home price against the predictions from two models. The plot shows the training and test sets in the same Lattice plot

5 Exploring and Comparing Resampling Distributions

5.1 Within–Model

There are several Lattice functions than can be used to explore relationships between tuning parameters and the resampling results for a specific model:

- `xyplot` and `stripplot` can be used to plot resampling statistics against (numeric) tuning parameters.
- `histogram` and `densityplot` can also be used to look at distributions of the tuning parameters across tuning parameters.

For example, the following statements produces the images in Figure 6.

```
> xyplot(marsFit, type= c("g", "p", "smooth"), degree = 2)
> densityplot(marsFit, as.table = TRUE, subset = nprune < 10)
```

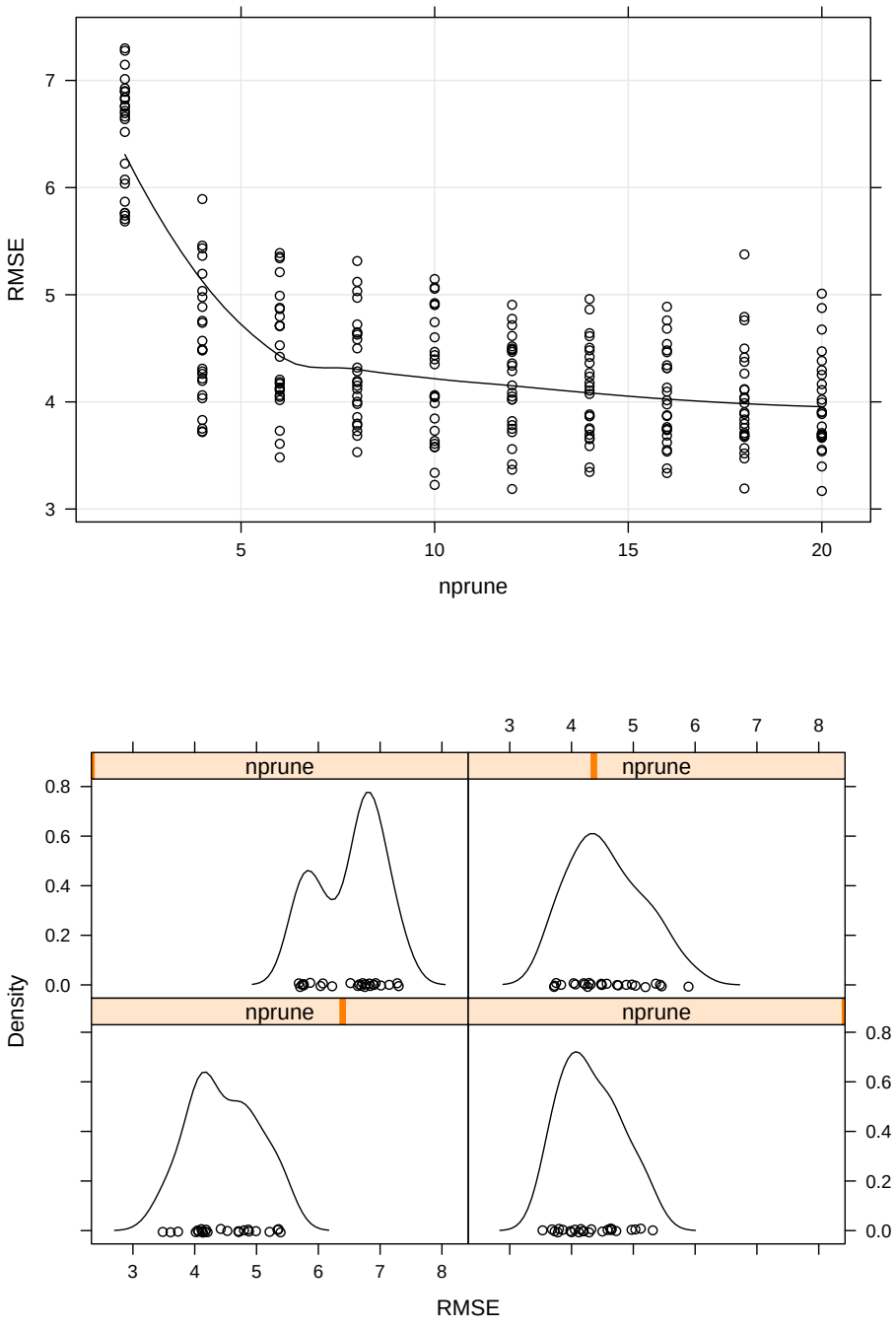

Figure 6: Scatter plots and density plots of the resampled RMSE by the number of retained terms for the MARS model fit to the Boston Housing data

5.2 Between–Models

caret also includes functions to characterize the differences between models (generated using caret, sbf or rfe) via their resampling distributions. These functions are based on the work of Hothorn et al. (2005) and Eugster et al (2008).

Using the blood-brain barrier data (see ?BloodBrain), three regression models were created: an rpart tree, a conditional inference tree using ctree, M5 rules using M5Rules and a MARS model using earth. We ensure that the models use the same resampling data sets. In this case, 100 leave–group–out cross–validation was employed.

```
> data(BloodBrain)
> set.seed(1)
> tmp <- createDataPartition(logBBB, p = 0.8, times = 100)
> bbbControl <- trainControl(method = "LGOCV", index = tmp, timingSamps = 50)
> rpartFit <- train(bbbDescr, logBBB,
+                   "rpart",
+                   tuneLength = 16,
+                   trControl = bbbControl)
> ctreeFit <- train(bbbDescr, logBBB,
+                  "ctree2",
+                   tuneLength = 10,
+                   trControl = bbbControl)
> earthFit <- train(bbbDescr, logBBB,
+                   "earth",
+                   tuneLength = 20,
+                   trControl = bbbControl)
> m5Fit <- train(bbbDescr, logBBB,
+                "M5Rules",
+                trControl = bbbControl)
```

Given these models, can we make statistical statements about their performance differences? To do this, we first collect the resampling results using resamples.

```
> resamps <- resamples(list(CART = rpartFit,
+                           CondInfTree = ctreeFit,
+                           MARS = earthFit,
+                           M5 = m5Fit))


> resamps

Call:
resamples.default(x = list(CART = rpartFit, CondInfTree = ctreeFit,     MARS = earthFit, M5 = m5Fit))

Models: CART, CondInfTree, MARS, M5
Number of resamples: 100
Performance metrics: RMSE, Rsquared
Time estimates for: everything, final model fit, prediction
```

```
> summary(resamps)

Call:
summary.resamples(object = resamps)

Models: CART, CondInfTree, MARS, M5
Number of resamples: 100

RMSE
              Min. 1st Qu. Median   Mean 3rd Qu.   Max.
CART        0.4905  0.5870 0.6480 0.6400  0.6869 0.8094
CondInfTree 0.4528  0.5934 0.6375 0.6427  0.6873 0.8685
MARS        0.4387  0.5709 0.6073 0.6128  0.6601 0.8327
M5          0.4607  0.5689 0.6219 0.6308  0.6763 0.8341

Rsquared
               Min. 1st Qu. Median   Mean 3rd Qu.   Max.
CART        0.11790  0.2751 0.3293 0.3398  0.3982 0.6351
CondInfTree 0.07711  0.2852 0.3517 0.3449  0.4099 0.6164
MARS        0.18800  0.3381 0.4146 0.4141  0.4939 0.6515
M5          0.12260  0.3080 0.3910 0.3950  0.4744 0.6286
```

There are several Lattice plot methods that can be used to visualize the resampling distributions: density plots, box–whisker plots, scatterplot matrices and scatterplots of summary statistics. In the latter case, the plot consists of a scatterplot between the two models. (See Figure 7). In Figure 8, density plots of the data are shown. In this figure, the R^2 distributions indicate that M5 rules and MARS appear to be similar to one another but different from the two tree–based models. However, this pattern is inconsistent with the root mean squared error distributions.

```
> bwplot(resamps, metric = "RMSE")
> densityplot(resamps, metric = "RMSE")
> xyplot(resamps,
+        models = c("CART", "MARS"),
+        metric = "RMSE")
> splom(resamps, metric = "RMSE")
```

Since models are fit on the same versions of the training data, it makes sense to make inferences on the differences between models. In this way we reduce the within–resample correlation that may exist. We can compute the differences, then use a simple t–test to evaluate the null hypothesis that there is no difference between models.

```
> difValues <- diff(resamps)
> difValues

Call:
diff.resamples(x = resamps)
```

```
Models: CART, CondInfTree, MARS, M5
Metrics: RMSE, Rsquared
Number of differences: 6
p-value adjustment: bonferroni
```

```
> summary(difValues)
```

```
Call:
summary.diff.resamples(object = difValues)

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

RMSE
            CART      CondInfTree MARS      M5
CART                  -0.002721    0.027124  0.009166
CondInfTree 1.0000000              0.029845  0.011887
MARS        0.0019641 0.0006745             -0.017958
M5          1.0000000 1.0000000   0.2662612

Rsquared
            CART      CondInfTree MARS      M5
CART                  -0.005135   -0.074297 -0.055242
CondInfTree 1.0000000             -0.069162 -0.050106
MARS        3.646e-09 6.541e-08              0.019055
M5          0.0002888 0.0010240   0.6796093
```

Note that these results are consistent with the patterns shown in Figure 8; there are more differences in the R^2 distributions than in the error distributions.

Several Lattices methods also exist to plot the differences (density and box–whisker plots) or the inferential results (level and dot plots). Figures 9 and 10 show examples of level and dot plots.

```
> dotplot(difValues)
> densityplot(difValues,
+             metric = "RMSE",
+             auto.key = TRUE,
+             pch = "|")
> bwplot(difValues,
+        metric = "RMSE")
> levelplot(difValues, what = "differences")
```

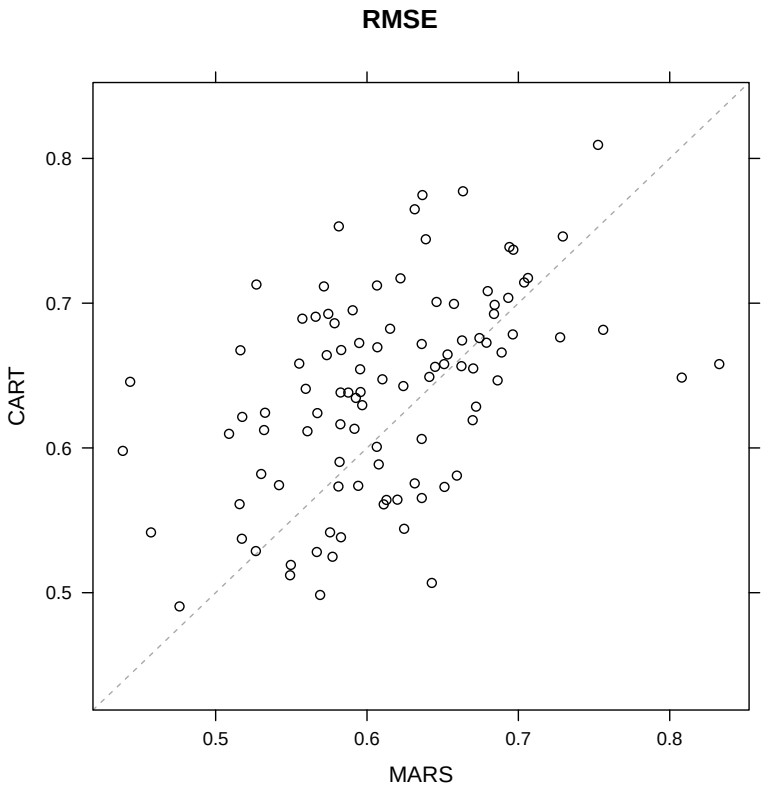

Figure 7: Examples of output from `xyplot(resamps, models = c("CART", "MARS"))`.

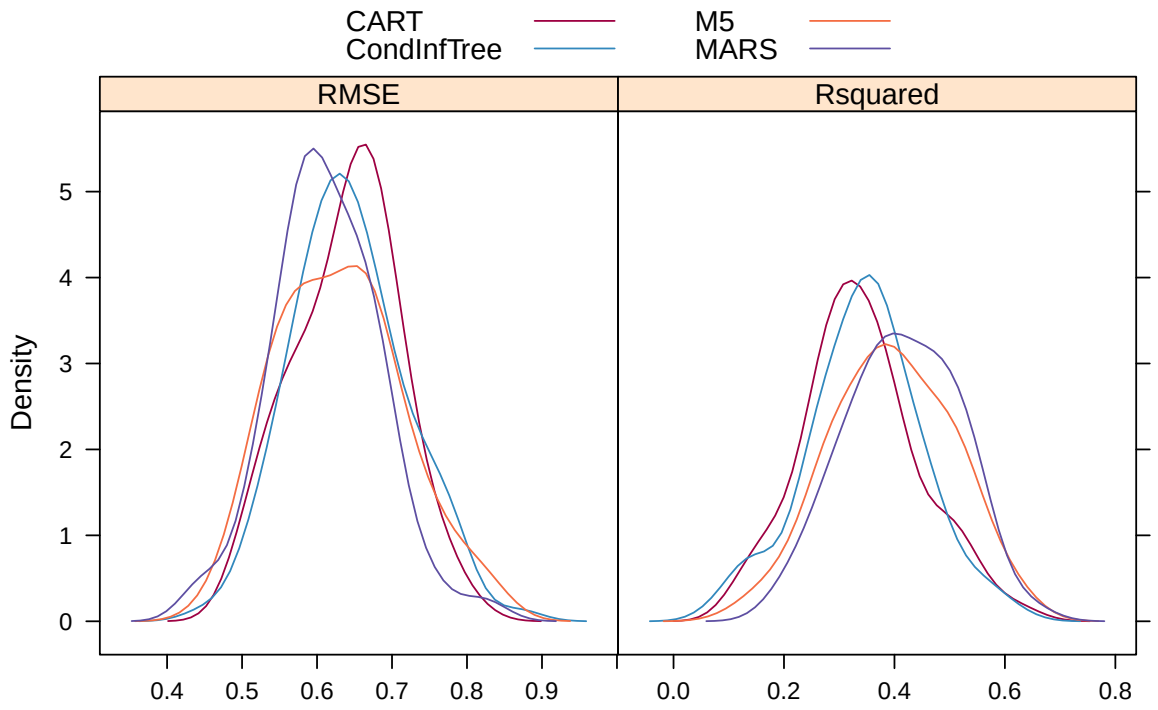

Figure 8: Examples of output from `densityplot(resamps)`. Looking at R^2, M5 rules and MARS appear to be similar to one another but different from the two tree–based models. However, this pattern is inconsistent with the root mean squared error distributions.

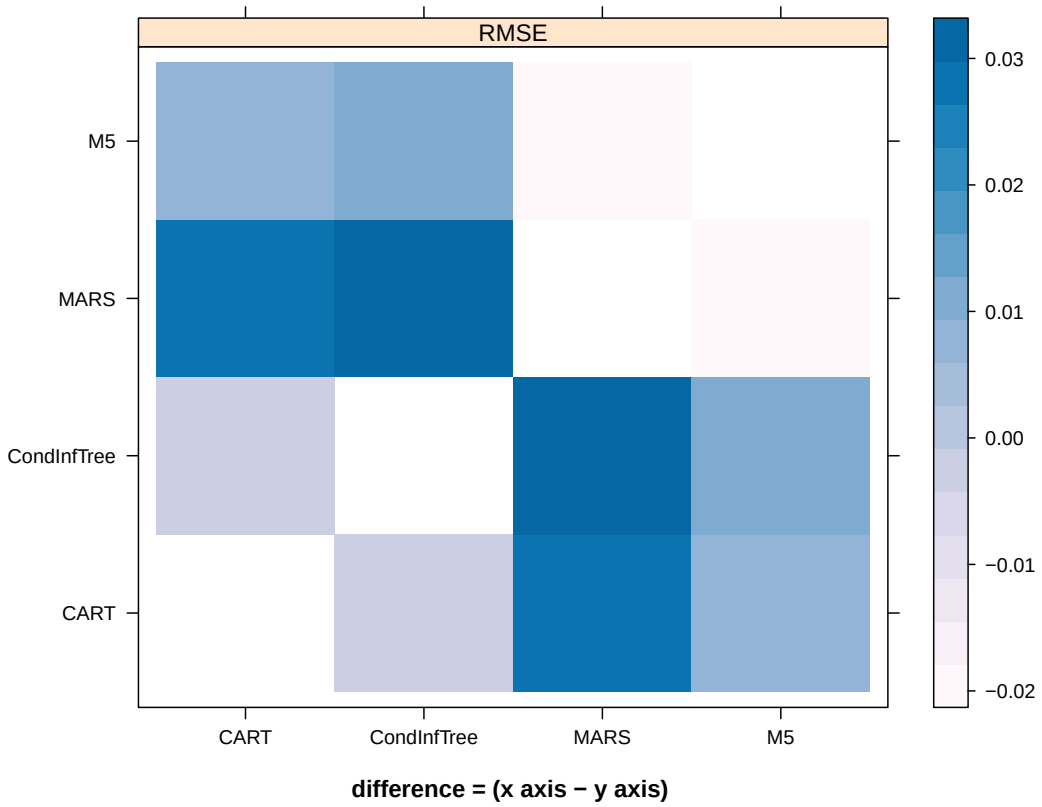

Figure 9: Examples of output from `levelplot(difValues, what = "differences")`. The pair–wise differences in RMSE are shown

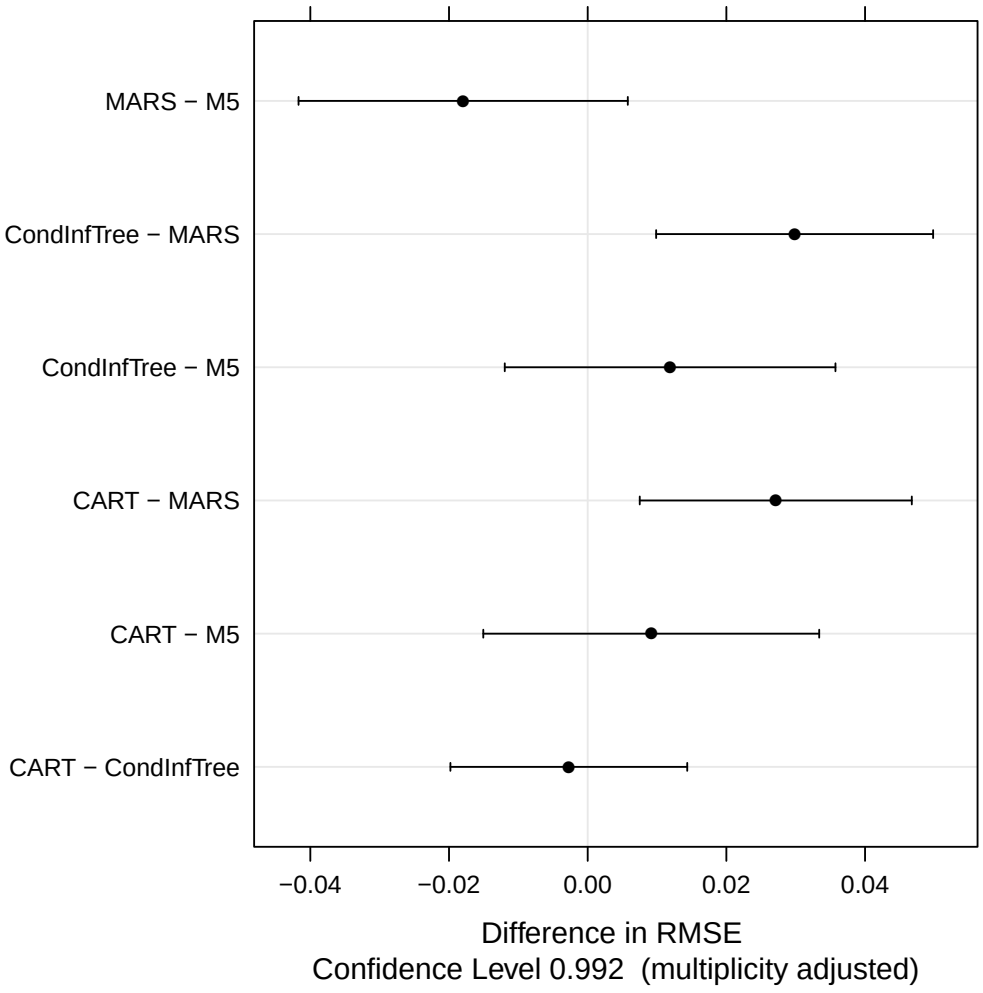

Figure 10: Examples of output from `dotplot(difValues)`. The differences in RMSE and their associated confidence intervals are shown.

6 Custom Methods for train

Although there are currently more than 130 methods available to train, there may be the need to create custom model functions (e.g. testing a new model etc). One application of custom models would be to create diverse ensembles of models. For example, a set of different classification models may be fit to the same data and a "pick–the–winner" approach can be taken (or the average of the class probabilities could be used, see Kuncheva (2004) or Seni and Elder (2010)). train already has a framework for resampling and tuning models and predict.train can be used to encapsulate the ensemble of models into one call for prediction.

Why Not Re–Write train Altogether?

One could make a strong argument that back–fitting the package to work with custom models is a bit kludgy. This is probably true, but there are several munches of the current code base that, if implemented in a more formal manner, would make the new code base overly complicated.

For example, the current code base exploits models that can produce predictions for many tuning parameters using a single model object. For example PLS can fit a K component model and make predictions from models with $1 \ldots K$ components. This potentially saves us $K-1$ model fits. PLS is fairly fast, but pother models with this same feature (gbm, cubist, etc) are computationally taxing and this saves a lot of time.

Also, the package currently does many types of resampling and some, such as the bootstrap 632 estimator, are not as simple as others. In this case, an additional model must be fit for all the data to get the apparent error rate. Formalizing this (or starting over) with a new code base would be unnecessarily complex.

6.1 How To Write Custom Methods

The user will need to deviate from the standard call in two ways:

- use `method = "custom"` in the call to train, and
- add the required functions for the model using trainControl

The custom argument of trainControl requires a list of named functions with the following elements: parameters, model, prediction, prob and sort.

6.1.1 The parameters Argument

This element is used to specify or generate the models tuning parameters. This can be done either as a function to generate them or a data frame of the actual parameters.

Inputs:

- data: a data frame of the training set data. The outcome will be in a column labeled .outcome. If the formula method for train was invoked, the data passed into this function will have been processed (i.e. dummy variables have been created etc).
- len an optional parameter passed in form the tuneLength argument to train

Outputs: a data frame where

- all columns start with a dot
- there is at least one row

Instead of a function, the final data frame can be passed in

6.1.2 The model Function

This element fits the model and any other functions (e.g. pre–processing of the data)

Inputs:

- data: a data frame of the training set data. The outcome will be in a column labeled .outcome. If case weights were specified in the train call, these are in the column .modelWeights. If the formula method for train was invoked, the data passed into this function will have been processed (i.e. dummy variables have been created etc).
- weights case weights
- parameter a single row data frame with the current tuning parameter
- levels: either NULL or a character vector or factor labels
- last a logical vector for the final model fit with the selected tuning parameters and the full training set
- ... arguments passed form train to this function

Outputs: a list with at least one element:

- fit: the object corresponding to the trained model

Anything else can be attached to this object. If custom pre–processing is required, this can be estimated in the model function and attached to the output list. Subsequent calls to the prediction and probability functions will have the entire list available, so the processing can be applied to the new data.

6.1.3 The prediction Function

This should be a function that produces either a number vector (for regression) or a factor (or character) vector for classification.

Inputs:

- object: a list with two elements resulting from the model function
- newdata: a matrix or data frame of predictors to be processed through the model (and possibly pre-processing routine)

The output should be either a numeric, character or factor vector. For classification, factors are converted to character elsewhere to ensure the proper levels are in the output.

6.1.4 The probability Argument

For classification models, this function should generate a data frame of class probabilities. For regression, a value of NULL can be used.

Inputs:

- object: a list with two elements resulting from the model function
- newdata: a matrix or data frame of predictors to be processed through the model (and possibly pre-processing routine)

The output should be a data frame with these characteristics:

- as many columns are factor levels
- column names are the same as the factor levels and in the same order

6.1.5 The sort Function

There are cases where multiple tuning parameters yield the same level of performance. In these situations, train will choose the parameters associated with the most simplistic model. This function should take the grid of tuning parameters and order them from least complex to most complex.

The input is a data frame of tuning parameters (without the preceding dot in the name).

The output is the same data frame sorted appropriately.

6.2 An Example

As an example, suppose we want to test out rpart models where we tune over the complexity parameter and the minimum number of samples in a node to do further splitting (a.k.a minsplit).

We'll use the Blood–brain barrier data in caret to illustrate.

First, we would need to create a training grid with the candidate values of cp and minsplit. When using the nominal rpart method in train, an initial rpart model is created and the unique values of the complexity parameter are obtained from the sub–object cptable. We will test two values of minsplit: 10 and 30. First, we get the unique C_p values for minsplit = 10

```
> ## rpart requires a formula method
> tmpData <- bbbDescr
> tmpData$logBBB <- logBBB
> cpValues10 <- rpart(logBBB ~ ., data = tmpData,
+                     control = rpart.control(minsplit = 10))$cptable[,"CP"]
> cpValues30 <- rpart(logBBB ~ ., data = tmpData,
+                     control = rpart.control(minsplit = 30))$cptable[,"CP"]
> head(cpValues10)
```

```
         1          2          3          4          5          6
0.36985527 0.09302467 0.05786716 0.03435632 0.03422223 0.02945773
```

From these, we will create the tuning grid of canidiate models (3 values of minsplit for each of the possible C_p values:

```
> rpartGrid <- data.frame(.cp = c(cpValues10, cpValues30),
+                         .minsplit =
+                         c(rep(10, length(cpValues10)),
+                           rep(30, length(cpValues30))))
>
```

We can now write a model function:

```
> modelFunc <- function(data, parameter, levels, last, ...)
+   {
+     library(rpart)
+     ctrl <- rpart.control(cp = parameter$.cp,
+                           minsplit = parameter$.minsplit)
+
+     list(fit = rpart(.outcome ~ ., data = data, control = ctrl))
+   }
```

It is a good idea to load the rpart package and anything else needed within the function.

The prediciton function is simple:

```
> predFunc <- function(object, newdata)
+   {
+     library(rpart)
+     predict(object$fit, newdata)
+   }
```

Sorting by complexity is somewhat subjective. Both parameters govern how deep the tree can be. We will sort by `cp` then `minsplit`:

```
> sortFunc <- function(x)  x[order(x$cp, x$minsplit),]
```

Now we can create a control object for `train`:

```
> ctrl <- trainControl(custom = list(
+                        parameters = rpartGrid,
+                        model = modelFunc,
+                        prediction = predFunc,
+                        probability = NULL,
+                        sort = sortFunc),
+                      method = "cv")
> set.seed(581)
> customRpart <- train(bbbDescr, logBBB, "custom", trControl = ctrl)
```

The `predict`, `print`, `plot` and `resamples` methods work with custom models. In the case of `plot.train`, the axis and key labels will be the parameter names. However, `update` can be used to make the labels more descriptive:

```
> rpartPlot <- plot(customRpart, xTrans = log10)
> rpartPlot <- update(rpartPlot, xlab = "Complexity Parameter")
```

7 Session Information

- R version 2.14.0 (2011-10-31), `x86_64-apple-darwin9.8.0`
- Locale: `C/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8`
- Base packages: base, datasets, grDevices, graphics, grid, methods, splines, stats, utils
- Other packages: Hmisc 3.9-1, MASS 7.3-16, caret 5.13-037, class 7.3-3, cluster 1.14.1, codetools 0.2-8, e1071 1.6, earth 3.2-1, ellipse 0.3-5, foreach 1.3.2, gam 1.06.2, gbm 1.6-3.1, ipred 0.8-11, iterators 1.0.5, kernlab 0.9-14, klaR 0.6-6, lattice 0.20-0, leaps 2.9, mlbench 2.1-0, nnet 7.3-1, pROC 1.5, plotmo 1.3-1, plotrix 3.3-3, pls 2.3-0, plyr 1.7.1, proxy 0.4-7, randomForest 4.6-6, reshape 0.8.4, rpart 3.1-51, survival 2.36-10
- Loaded via a namespace (and not attached): compiler 2.14.0, tools 2.14.0

8 References

Breiman, Friedman, Olshen, and Stone. (1984) *Classification and Regression Trees*. Wadsworth.

Eugster et al. (2008), "Exploratory and inferential analysis of benchmark experiments, " *Ludwigs-Maximilians-Universitat Munchen, Department of Statistics, Tech. Rep* vol. 30

Hothorn et al. (2005), "The design and analysis of benchmark experiments, " *Journal of Computational and Graphical Statistics*, 14, 675–699

Kuncheva (2004), *Combining Pattern Classifiers: Methods and Algorithms*. Wiley.

Molinaro et al. (2010), "partDSA: deletion/substitution/addition algorithm for partitioning the covariate space in prediction," *Bioinformatics*, 26, 1357–1363

Rand (1971), "Objective criteria for the evaluation of clustering methods," *Journal of the American Statistical Association* 66, 846–850.

Schmidberger et al. (2009), " State-of-the-art in Parallel Computing with R," *Journal of Statistical Software*, 31

Seni and Elder (2010) *Ensemble Methods in Data Mining: Improving Accuracy Through Combining Predictions*. Morgan and Claypool Publishers

Svetnik, V., Wang, T., Tong, C., Liaw, A., Sheridan, R. P. and Song, Q. (2005), "Boosting: An ensemble learning tool for compound classification and QSAR modeling," *Journal of Chemical Information and Modeling*, 45, 786 –799.

Tibshirani, R., Hastie, T., Narasimhan, B., Chu, G. (2003), "Class prediction by nearest shrunken centroids, with applications to DNA microarrays," *Statistical Science*, 18, 104–117.