The `caret` Package

Max Kuhn
max.kuhn@pfizer.com

May 27, 2011

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, caTools, class, Cubist, e1071,, earth ($\geq$ 2.2-3), elasticnet, ellipse, fastICA, foba, foreach, gam, GAMens ($\geq$ 1.1.1),, gbm, glmnet, gpls, grid, hda, HDclassif, Hmisc, ipred, kernlab, klaR, lars, LogicForest,, logicFS, LogicReg, MASS, mboost, mda, mgcv, mlbench, neuralnet, nnet, nodeHarvest, pamr,, partDSA, party ($\geq$ 0.9-99992), penalized, pls, proxy, qrnn, quantregForest, randomForest, RANN, rda, relaxo, rocc, rpart, rrcov, 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 train function can be used to

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

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

More formally:

```
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, 114 are available using `train`; see Table 1 for details.

In Table 1, there is a list 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 `train`. 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 2.6).

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

To estimate model performance across the tuning parameters "leave group out cross–validation" (`LGOCV`) can be used. This technique is repeated splitting of the data into training and test sets (without replacement). If the resampling method is not specified, simple bootstrapping is used. To train a support vector machine classification model (radial basis function kernel) on these multidrug resistance reversal agent data, we can first setup a control object[2] that specifies the type of resampling used, the number of data splits (30), the proportion of data in the sub–training sets (75%) and whether the iterations should be printed as they occur. In this case, we need to specify the proportion of samples used in each resampled training set. We also set the seed.

```
> fitControl <- trainControl(method = "LGOCV",
+                            p = .75,
+                            number = 30,
+                            returnResamp = "all",
+                            verboseIter = FALSE)
> set.seed(2)
```

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

The first two arguments to `train` are the predictor and outcome data objects, respectively. The third argument, `method`, specifies the type of model (see Table 1). For this model, the tuning parameters are the cost value (the `C` argument in `kernlab`'s `ksvm` function) and the radius of the RBF (the `sigma` argument to the kernel function). The `tuneLength` argument sets the size of the grid used to search the tuning parameter space and `trControl` is the control parameter for the `train` function. The `preProcess` argument is a string of operations that can be used on each of the 30 resampled set of predictors; in this case, we simply center and scale the data prior to modeling and prediction. The held–out samples are normalized using the means and standard deviations from the corresponding data set used to fit the model. See the help file for the `preProcessing` function for a list of possible techniques.

```
> svmFit <- train(trainDescr, trainMDRR,
+                 method = "svmRadial",
+                 preProcess = c("center", "scale"),
+                 tuneLength = 4,
+                 trControl = fitControl)
> svmFit

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

[2]This is optional; to use the default specifications, the control object does not need to be specified

```
Pre-processing: centered, scaled
Resampling: Repeated Train/Test Splits (30 reps, 0.75%)

Summary of sample sizes: 198, 198, 198, 198, 198, 198, ...

Resampling results across tuning parameters:

  C     Accuracy  Kappa  Accuracy SD  Kappa SD
  0.25  0.803     0.583  0.0383       0.0844
  0.5   0.837     0.663  0.0432       0.0904
  1     0.837     0.665  0.0406       0.0846
  2     0.823     0.638  0.0454       0.0931

Tuning parameter 'sigma' was held constant at a value of 0.0222
Accuracy was used to select the optimal model using  the largest value.
The final values used for the model were C = 1 and sigma = 0.0222.
```

There are two tuning parameters for this model: `sigma` is a parameter for the kernel function that can be used to expand/contract the distance function and `C` is the cost parameter that can be used as a regularization term that controls the complexity of the model. For this model, the function `sigest` in the `kernlab` package is used to provide a good estimate of the `sigma` parameter, so that only the cost parameter is tuned. This tuning scheme is the default, but can be modified (details are below).

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.

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).

`caret` works with specific models (see Table 1). For these models, `train` 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 `train`. In Section 2.2, there is an the example using gradient boosting machines (GBM) where the default trace for a `gbm` model was turned off using the `verbose` argument to `gbm`.

- 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 `train` 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. `train` 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 `train` 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 train 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 less 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, cubist, earth, enet, foba, gamboost, gbm, glmboost, glmnet, lars, lars2, lasso, logitBoost, pam, partDSA, pcr, pls, relaxo, rpart, scrda, superpc), the train 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 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. 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 `train` 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), `train` automatically fits a model with these parameters to the entire training data set. That model object is accessible in the `finalModel` object within `train`. For example, `svmFit$finalModel` is the same object that would have been produced using a direct call to the `ksvm` function with the final tuning parameters.

There is additional functionality in `train` that is described in the next section.

Table 1: Models used in `train`

Model	`method` Value	Package	Tuning Parameters
	"Dual–Use Models"		
Generalized linear model	`glm`	`stats`	None
	`glmStepAIC`	`MASS`	None
Generalized additive model	`gam`	`mgcv`	`select`, `method`
	`gamLoess`	`gam`	`span`, `degree`
	`gamSpline`	`gam`	`df`
Recursive Partitioning	`rpart`	`rpart`	`maxdepth`
	`ctree`	`party`	`mincriterion`
	`ctree2`	`party`	`maxdepth`
Boosted Trees	`gbm`	`gbm`	`n.trees`, `shrinkage`, `interaction.depth`
	`blackboost`	`mboost`	`maxdepth`, `mstop`
	`ada`	`ada`	`maxdepth`, `iter`, `nu`
Other Boosted Models	`glmboost`	`mboost`	`mstop`
	`gamboost`	`mboost`	`mstop`
Random Forests	`rf`	`randomForest`	`mtry`
	`parRF`	`randomForest`, `foreach`	`mtry`
	`cforest`	`party`	`mtry`
	`Boruta`	`Boruta`	`mtry`
Bagging	`treebag`	`ipred`	None
	`bag`	`caret`	`vars`
	`logicBag`	`logicFS`	`ntrees`, `nleaves`
Other Trees	`nodeHarvest`	`nodeHarvest`	`maxinter`, `mode`
	`partDSA`	`partDSA`	`cut.off.growth`, `MPD`
Multivariate Adaptive Regression Splines	`earth`, `mars`	`earth`	`degree`, `nprune`
	`gcvEarth`	`earth`	`degree`
Bagged MARS	`bagEarth`	`caret`, `earth`	`degree`, `nprune`
Logic Regression	`logreg`	`LogicReg`	`ntrees`, `treesize`
Elastic Net (glm)	`glmnet`	`glmnet`	`alpha`, `lambda`

(continued on next page)

Table 1: Models used in `train`

Model	`method` Value	Package	Tuning Parameters
Neural Networks	`nnet`	`nnet`	`decay`, `size`
	`pcaNNet`	`caret`, `nnet`	`decay`, `size`
Partial Least Squares	`pls`	`pls` , `caret`	`ncomp`
Sparse Partial Least Squares	`spls`	`spls` , `caret`	`K`, `eta`, `kappa`
Support Vector Machines	`svmLinear`	`kernlab`	none
	`svmRadial`	`kernlab`	`sigma`, `C`
	`svmRadialCost`	`kernlab`	`C`
	`svmPoly`	`kernlab`	`scale`, `degree`, `C`
Gaussian Processes	`gausssprLinear`	`kernlab`	none
	`gausssprRadial`	`kernlab`	`sigma`
	`gausssprPoly`	`kernlab`	`scale`, `degree`
k Nearest Neighbors	`knn`	`caret`	`k`
	Regression Only Models		
Linear Least Squares	`lm`	`stats`	None
	`lmStepAIC`	`MASS`	None
Principal Component Regression	`pcr`	`pls`	`ncomp`
Independent Component Regression	`icr`	`caret`	`n.comp`
Robust Linear Regression	`rlm`	`MASS`	None
Neural Networks	`neuralnet`	`neuralnet`	`layer1`, `layer2`, `layer3`
Quantile Regression Forests	`qrf`	`quantregForest`	`mtry`
Quantile Regression Neural Networks	`qrnn`	`qrnn`	`n.hidden`, `penalty`, `bag`
Rule–Based Models	`M5Rules`	`RWeka`	`pruned`, `smoothed`
	`cubist`	`Cubist`	`committees`, `neighbors`
Projection Pursuit Regression	`ppr`	`stats`	`nterms`
Penalized Linear Models	`penalized`	`penalized`	`lambda1`, `lambda2`
	`lars`	`lars`	`fraction`
	`lars2`	`lars`	`step`
	`enet`	`elasticnet`	`lambda`, `fraction`

(continued on next page)

Table 1: Models used in `train`

Model	`method` Value	Package	Tuning Parameters
	`lasso`	`elasticnet`	`fraction`
	`foba`	`foba`	`lambda`, `k`
Relevance Vector Machines	`rvmLinear`	`kernlab`	none
	`rvmRadial`	`kernlab`	`sigma`
	`rvmPoly`	`kernlab`	`scale`, `degree`
Supervised Principal Components	`superpc`	`superpc`	`n.components`, `threshold`
	Classification Only Models		
Linear Discriminant Analysis	`lda`	`MASS`	None
	`Linda`	`rrcov`	None
Quadratic Discriminant Analysis	`qda`	`MASS`	None
	`QdaCov`	`rrcov`	None
Stabilized Linear Discriminant Analysis	`slda`	`ipred`	`diagonal`
Heteroscedastic Discriminant Analysis	`hda`	`hda`	`newdim`, `lambda`, `gamma`
Shrinkage Linear Discriminant Analysis	`sda`	`sda`	`diagonal`
Sparse Linear Discriminant Analysis	`sparseLDA`	`sparseLDA`	`NumVars`, `lambda`
Stepwise Discriminant	`stepLDA`,	`klaR`	None
Stepwise Diagonal Discriminant Analysis	`sddaLDA`, `sddaQDA`	`SDDA`	None
Regularized Discriminant Analysis	`rda`	`klaR`	`lambda`, `gamma`
Mixture Discriminant Analysis	`mda`	`mda`	`subclasses`
Sparse Mixture Discriminant Analysis	`smda`	`sparseLDA`	`NumVars`, `R`, `lambda`
Penalized Discriminant Analysis	`pda`	`mda`	`lambda`
	`pda2`	`mda`	`df`
High Dimensional Discriminant Analysis	`hdda`	`HDclassif`	`model`, `threshold`
Flexible Discriminant Analysis (MARS basis)	`fda`	`mda`, `earth`	`degree`, `nprune`
Bagged FDA	`bagFDA`	`caret`, `earth`	`degree`, `nprune`
Logistic/Multinomial Regression	`multinom`	`nnet`	`decay`
	`plr`	`stepPlr`	`lambda`, `cp`
LogitBoost	`logitBoost`	`caTools`	`nIter`

(continued on next page)

Table 1: Models used in `train`

Model	`method` **Value**	**Package**	**Tuning Parameters**
Logistic Model Trees	`LMT`	`RWeka`	`iter`
Rule–Based Models	`J48`	`RWeka`	`C`
	`OneR`	`RWeka`	None
	`PART`	`RWeka`	`threshold`, `pruned`
	`JRip`	`RWeka`	`NumOpt`
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`	`usekernel`, `fL`
Generalized Partial Least Squares	`gpls`	`gpls`	`K.prov`
Learned Vector Quantization	`lvq`	`class`	`size`, `k`
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, `train` 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 Table 1 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`. `train` will tune the model over each combination of values in the rows.

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`

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)
> gbmFit <- train(trainDescr, trainMDRR,
+                 method = "gbm",
+                 tuneGrid = gbmGrid,
+                 trControl = fitControl,
+                 ## This next option is directly passed
+                 ## from train() to gbm()
+                 verbose = FALSE)
> gbmFit


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

Pre-processing: None
Resampling: Repeated Train/Test Splits (30 reps, 0.75%)

Summary of sample sizes: 198, 198, 198, 198, 198, 198, ...

Resampling results across tuning parameters:

  interaction.depth  n.trees  Accuracy  Kappa  Accuracy SD  Kappa SD
  1                  10       0.757     0.487  0.0376       0.0822
  1                  50       0.796     0.581  0.0366       0.0742
  1                  100      0.811     0.612  0.0421       0.0847
  1                  150      0.802     0.593  0.0444       0.0898
  1                  200      0.801     0.593  0.0439       0.0884
  1                  250      0.797     0.586  0.0426       0.0861
  1                  300      0.796     0.583  0.0458       0.093
  3                  10       0.789     0.561  0.0365       0.0768
  3                  50       0.811     0.612  0.0366       0.0735
  3                  100      0.807     0.604  0.0423       0.0851
```

```
  3                  150      0.804      0.598  0.0468       0.0942
  3                  200      0.801      0.592  0.0403       0.0808
  3                  250      0.8        0.59   0.047        0.0936
  3                  300      0.802      0.593  0.0444       0.0887

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 resampled 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 2.5.

- `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 2.6 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.
- workers, computeFunction, and computeArgs: these are options for parallel processing and are discussed in Section 2.7

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 train 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 train).

The output to the function should be a vector of numeric summary metrics with non–null names.

caret contains an alternate summary function called twoClassSummary that, for binary classification models, will compute the sensitivity, specificity and the area under the ROC curve. This is discussed more in the next section.

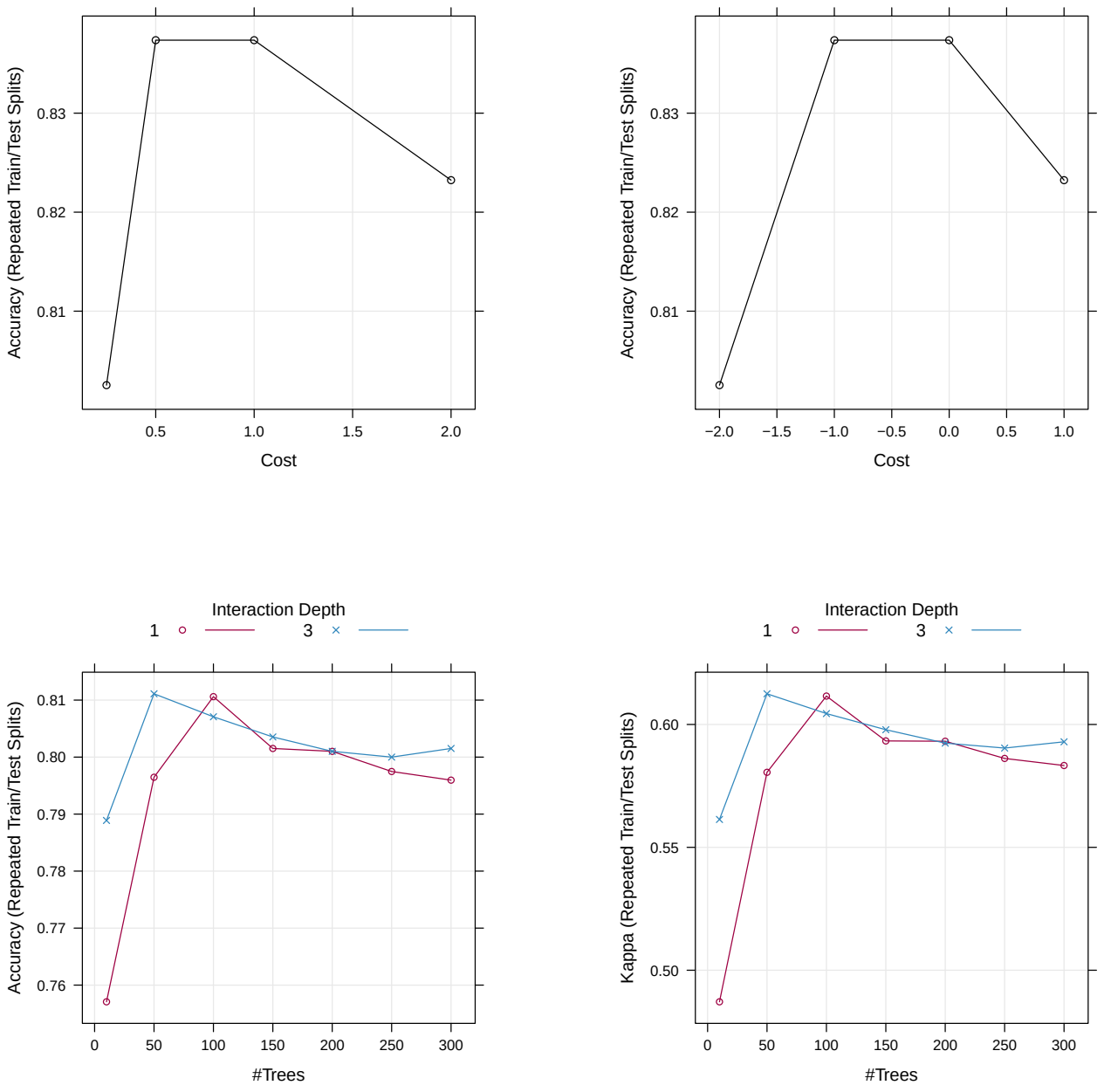

Figure 1: Examples of output from `plot.tain`. **top left** a plot produced using `plot(svmFit)` showing the relationship between SVM cost parameter and the resampled classification accuracy. Although this model has two tuning parameters, a constant value for the parameter `sigma` was used. **top right** the same plot but the `xTrans` argument was used to log–transform the cost parameter. **bottom left** a plot produced using `plot(gbmFit)` showing the relationship between the number of boosting iterations, the interaction depth and the resampled classification accuracy **bottom right** the same plot, but the Kappa statistic is plotted using `plot(gbmFit metric = "Kappa")`

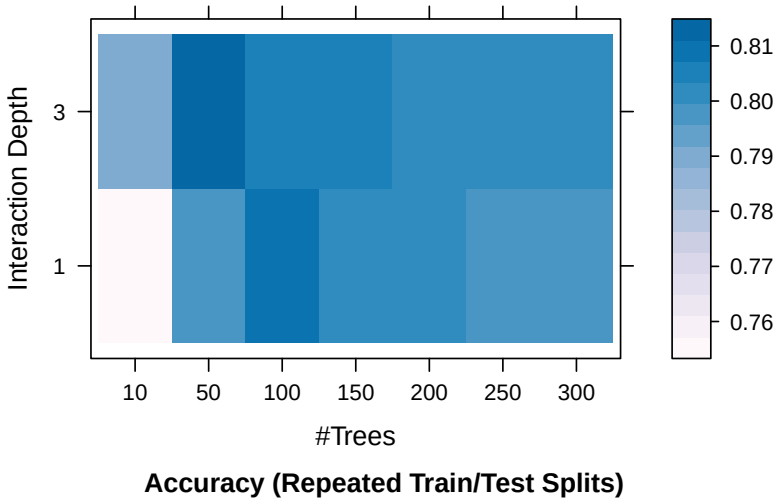

Figure 2: For the boosted tree example in Section 2.2, using `plot(gbmFit metric = "Kappa", plotType = "level")` shows the relationship (using a `levelplot`) between the number of boosting iterations, the interaction depth and the resampled estimate of the Kappa statistic.

As an example of a custom metric, suppose we want to used the Rand Index (Rand, 1971) to measure the similarity of the observed and predicted data. The `e1071` package contains a function called `classAgreement` that computes this value. We can use the following function to estimate the version of the Rand index that is corrected for random agreement:

```
> Rand <- function (data, lev, model)
+ {
+   library(e1071)
+   tab <- table(data[, "pred"], data[, "obs"])
+   out <- classAgreement(tab)$crand
+   names(out) <- "cRand"
+   out
+ }
```

To rebuild the support vector machine model using this criterion, we can see the relationship between the tuning parameters and the Rand index via the following code:

```
> fitControl$summaryFunction <- Rand
> set.seed(2)
> svmNew <- train(trainDescr, trainMDRR,
+                 method = "svmRadial",
+                 preProcess = c("center", "scale"),
+                 metric = "cRand",
+                 tuneLength = 4,
+                 trControl = fitControl)
> svmNew

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

Pre-processing: centered, scaled
Resampling: Repeated Train/Test Splits (30 reps, 0.75%)

Summary of sample sizes: 198, 198, 198, 198, 198, 198, ...

Resampling results across tuning parameters:

  C     cRand  cRand SD
  0.25  0.362  0.0927
  0.5   0.454  0.114
  1     0.453  0.108
  2     0.417  0.11

Tuning parameter 'sigma' was held constant at a value of 0.0222
cRand was used to select the optimal model using  the largest value.
The final values used for the model were C = 0.5 and sigma = 0.0222.
```

2.5 Performance Class Probabilities: ROC Curves

By default, `train` 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 (see Section 4.2 for details).

For example:

```
> fitControl <- trainControl(method = "LGOCV",
+                            p = .75,
+                            number = 30,
+                            classProbs = TRUE,
+                            summaryFunction = twoClassSummary,
+                            returnResamp = "all",
+                            verboseIter = FALSE)
> set.seed(2)
> svmROC <- train(trainDescr, trainMDRR,
+                 method = "svmRadial",
+                 tuneLength = 4,
+                 metric = "ROC",
+                 trControl = fitControl)
> svmROC

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

Pre-processing: None
Resampling: Repeated Train/Test Splits (30 reps, 0.75%)

Summary of sample sizes: 198, 198, 198, 198, 198, 198, ...

Resampling results across tuning parameters:

  C     Sens   Spec   ROC    Sens SD  Spec SD  ROC SD
  0.25  0.959  0.603  0.877  0.0338   0.0918   0.0364
  0.5   0.931  0.718  0.894  0.05     0.0785   0.0335
```

```
  1     0.904  0.753  0.895  0.053    0.0825   0.037
  2     0.869  0.764  0.888  0.0631   0.08     0.0386

Tuning parameter 'sigma' was held constant at a value of 0.0222
ROC was used to select the optimal model using  the largest value.
The final values used for the model were C = 1 and sigma = 0.0222.
```

In this case, the average area under the ROC curve was 0.895 across the 30 resamples.

2.6 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 `train` 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.

`train` 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 `train`.
- `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 `train`).

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 = 3, n trees = 50, shrinkage = 0.1. However, the scale in

this plots is fairly tight, with accuracy values ranging from 0.757 to 0.811. 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(gbmFit$results, "Accuracy", 2, TRUE)
> cat("best model within 2 pct of best:\n")
```

```
best model within 2 pct of best:
```

```
> gbmFit$results[whichTwoPct,]
```

```
  interaction.depth n.trees shrinkage  Accuracy     Kappa AccuracySD   KappaSD
7                 1      50       0.1 0.7964646 0.5805309 0.03661601 0.0741651
```

This indicates that we can get a less complex model with and accuracy of 0.796 (compared to the "pick the best" value of 0.811).

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.

2.7 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 support vector machine model in Section 1.2, each of the 4 candidate models was fit to 30 separate resamples. Since each resample is independent of the other, these 120 models could be computed in parallel.

R has several packages that facilitates parallel processing when multiple processors are available (see Schmidberger et al., 2009). `train` can be used 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. Examples of R packages that can be used for parallel processing with train are nws and Rmpi, among others.

To run in parallel, any function that emulates lapply can be used. For example, the snow package has a parallel lapply function that uses MPI. We can write a wrapper for this function:

```
> mpiCalcs <- function(X, FUN, ...)
+   {
+     theDots <- list(...)
+     parLapply(theDots$cl, X, FUN)
+   }
```

To use this function, a cluster must be started using MPI:

```
> cl <- makeCluster(5, "MPI")
        5 slaves are spawned successfully. 0 failed.
```

and to use this cluster, the trainControl object must be modified:

```
> fitControl <- trainControl(method = "LGOCV",
+                            p = .75,
+                            number = 30,
+                            returnResamp = "all",
+                            verboseIter = FALSE
+                            workers = 5,
+                            computeFunction = mpiCalcs,
+                            computeArgs = list(cl = cl))
```

The last three options specify the number of workers, the function that emulates lapply and the extra arguments to this function (such as the cluster object). The documentation page for train has another example using the nws package.

This control object is passed into train with no other modifications. The command stopCluster(cl) will shut down the workers.

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.

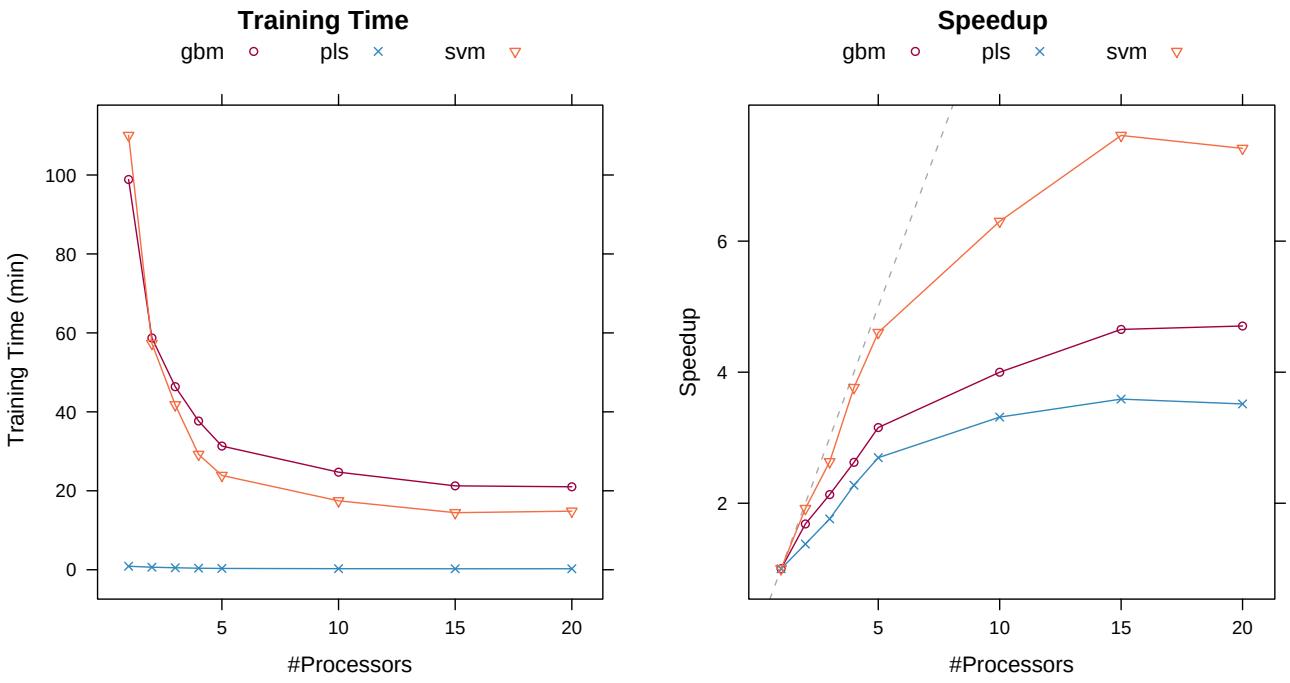

Figure 3: Training time profiles using parallel processing via `train` for a benchmarking data set run on a 32 core machine. The left panel shows the elapsed time to train various types of models using single or multiple processors. The panel on the right shows the "speedup," defined to be the time for serial execution divided by the parallel execution time. The reference line shows the maximum theoretical speedup.

Figure 3 shows the results of a benchmarking study run on a 32 core machine. 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.

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 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.

3 Extracting Predictions and Class Probabilities

As previously mentioned, objects produced by the `train` 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 `train` 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 `train` 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
```

```
> by(
+    testPred,
+    list(model = testPred$model),
+    function(x) postResample(x$pred, x$obs))
```

```
model: earth
     RMSE  Rsquared
4.6052438 0.8016275
------------------------------------------------------------
model: pls
     RMSE  Rsquared
5.5016752 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 train 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:

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

[1] 0.8066667

> confusionMatrix(mbrrPredictions$pred, mbrrPredictions$obs)

Confusion Matrix and Statistics

          Reference
Prediction Active Inactive
```

```
 Active      121       25
 Inactive     29       89

               Accuracy : 0.7955
                 95% CI : (0.7417, 0.8424)
    No Information Rate : 0.5682
    P-Value [Acc > NIR] : 6.255e-15

                  Kappa : 0.5849
 Mcnemar's Test P-Value : 0.6831

            Sensitivity : 0.8067
            Specificity : 0.7807
         Pos Pred Value : 0.8288
         Neg Pred Value : 0.7542
             Prevalence : 0.5682
         Detection Rate : 0.4583
   Detection Prevalence : 0.5530

       '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).

4.2 ROC Curves

The function `roc`[3] can be used to calculate the sensitivity and specificity used in an ROC plot. For example, using the previous support vector machine fit to the MBRR data, the predicted class probabilities on the test set can used to create an ROC curve. The area under the ROC curve, via the trapezoidal rule, is calculated using the `aucRoc` function.

```
> mbrrProbs <- extractProb(list(svmFit), testX = testDescr, testY = testMDRR)
> mbrrProbs <- mbrrProbs[mbrrProbs$dataType == "Test",]
> mbrrROC <- roc(mbrrProbs$Active, mbrrProbs$obs)
> aucRoc(mbrrROC)
```

```
[1] 0.8724269
```

See Figure 5 for an example.

[3] I'm looking into using the `ROCR` package for ROC curves, so don't get too attached to these functions

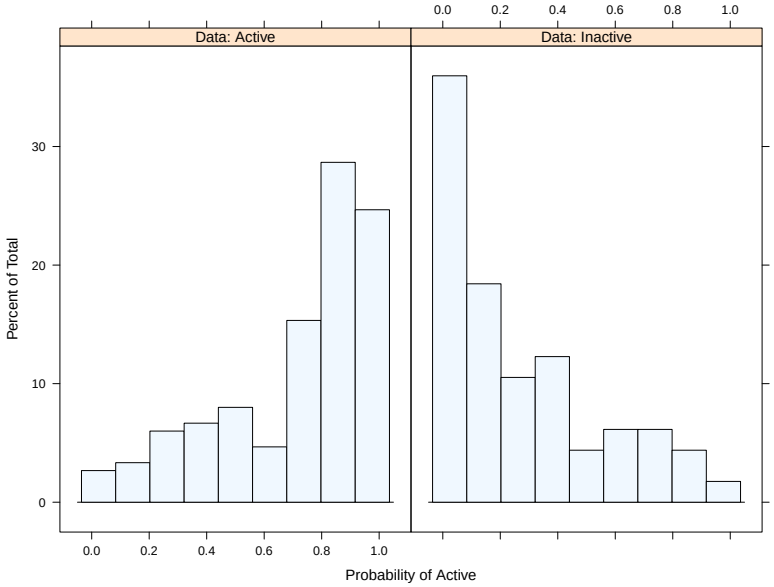

Figure 4: The predicted class probabilities from a support vector machine fit for the MBRR test set. This plot was created using `plotClassProbs(mbrrProbs)`.

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 Figures 6 and 5 for examples. 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).

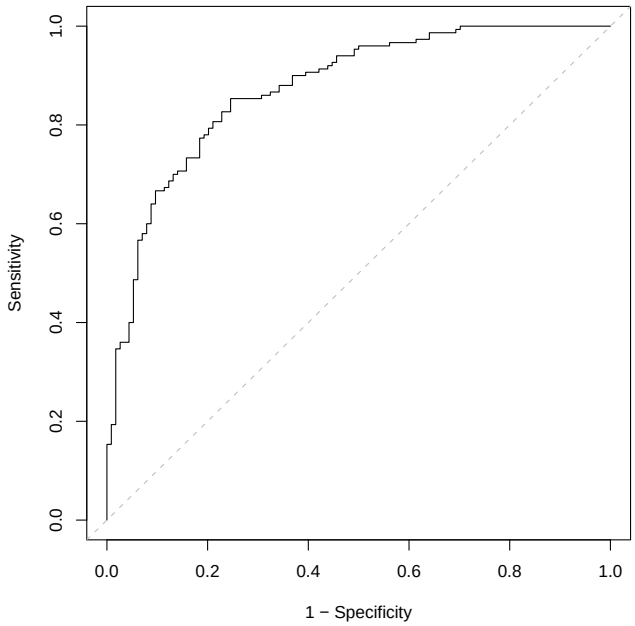

Figure 5: An ROC curve from the predicted class probabilities from a support vector machine fit for the MBRR test set.

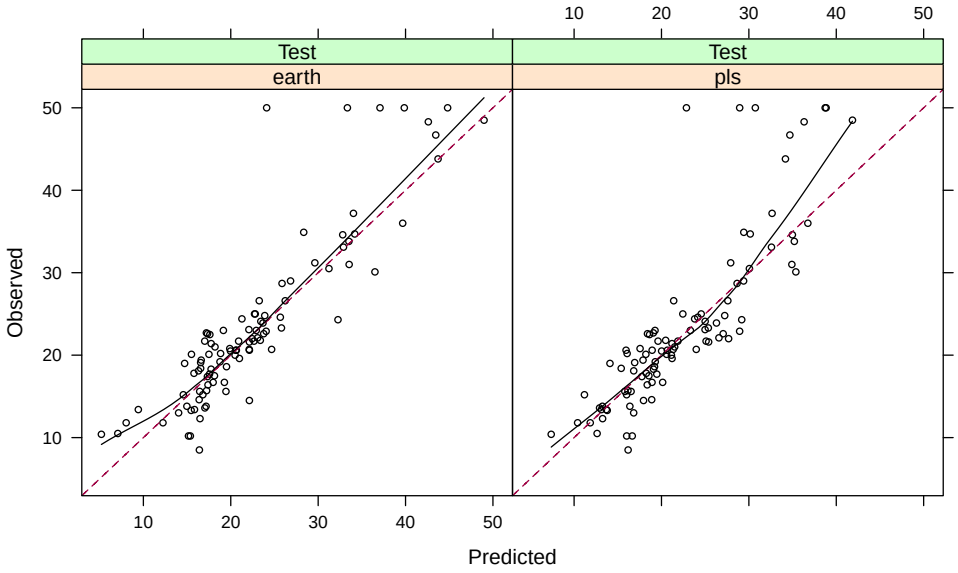

Figure 6: The results of using `plotObsVsPred` to show plots of the observed median home price against the predictions from two models. The plot shows the training and test sets in the same Lattice plot

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 7.

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

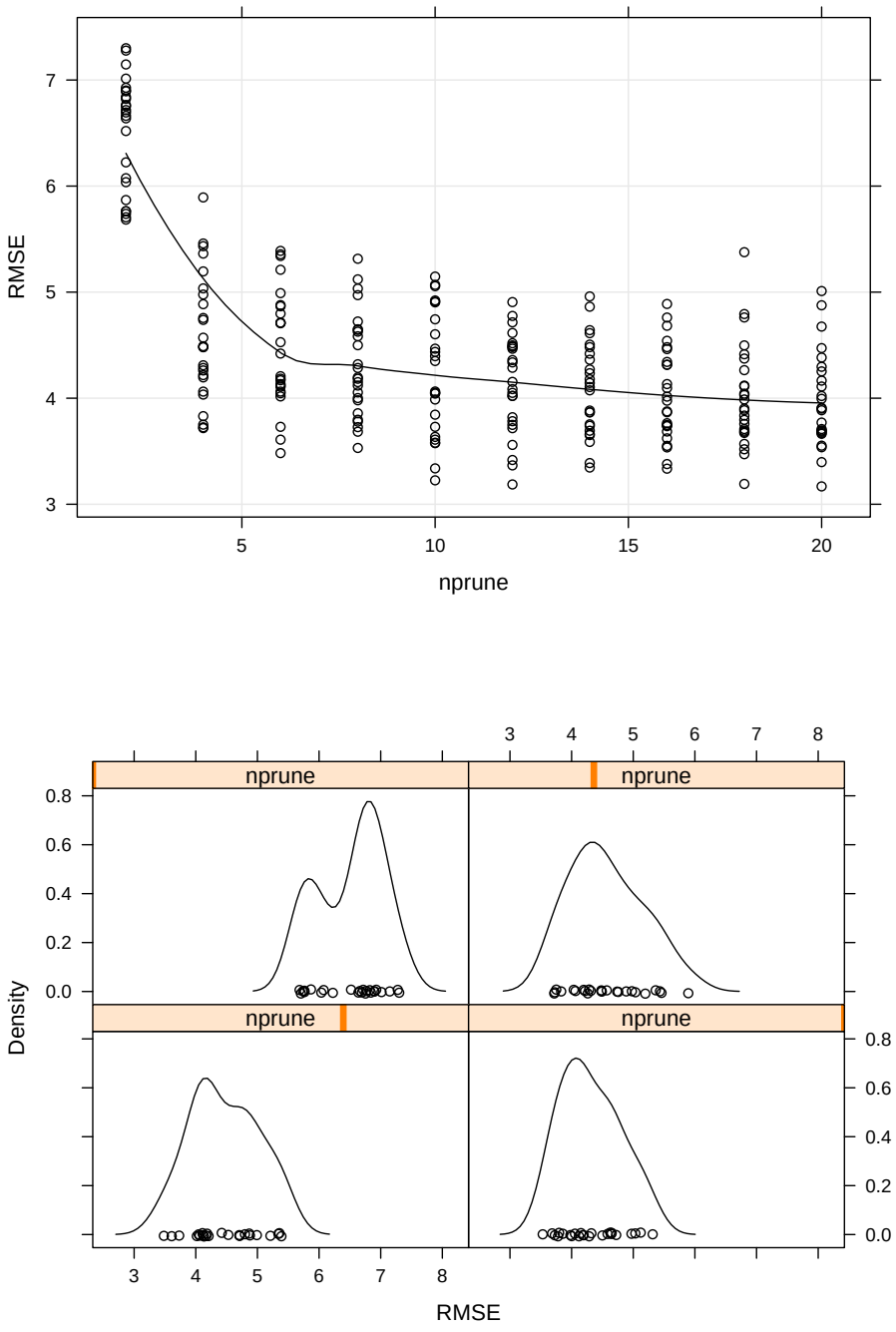

Figure 7: Scatter plots and density plots of the resampled RMSE by the number of retained terms for the MARS model fit to the Boston Housing data

5.2 Between–Models

caret also includes functions to characterize the differences between models (generated using train, 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)
> rpartFit <- train(bbbDescr, logBBB,
+                   "rpart",
+                   tuneLength = 16,
+                   trControl = trainControl(method = "LGOCV", index = tmp))

Fitting: maxdepth=16

> ctreeFit <- train(bbbDescr, logBBB,
+                  "ctree2",
+                   tuneLength = 10,
+                   trControl = trainControl(method = "LGOCV", index = tmp))

Fitting: maxdepth=1
Fitting: maxdepth=2
Fitting: maxdepth=3
Fitting: maxdepth=4
Fitting: maxdepth=5
Fitting: maxdepth=6
Fitting: maxdepth=7
Fitting: maxdepth=8
Fitting: maxdepth=9
Fitting: maxdepth=10

> earthFit <- train(bbbDescr, logBBB,
+                   "earth",
+                   tuneLength = 20,
+                   trControl = trainControl(method = "LGOCV", index = tmp))
```

```
Fitting: degree=1, nprune=76
```

```
> m5Fit <- train(bbbDescr, logBBB,
+                "M5Rules",
+                trControl = trainControl(method = "LGOCV", index = tmp))
```

```
Fitting: pruned=Yes
Fitting: pruned=No
```

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
```

```
Models: CART, CondInfTree, MARS, M5
Number of resamples: 100
Performance metrics: RMSE, Rsquared
```

```
> 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        0.4927  0.5806 0.6417 0.6331  0.6764 0.7819
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        0.12070  0.2749 0.3436 0.3455  0.4049 0.5981
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 the differences between two models on the y–axis and the average on the x–axis (See Figure 8). In Figure 9, 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                  -0.009607    0.020238  0.002280
CondInfTree 0.8305096               0.029845  0.011887
MARS        0.0447524 0.0006745             -0.017958
M5          1.0000000 1.0000000   0.2662612

Rsquared
            CART      CondInfTree MARS       M5
CART                   0.0005807  -0.0685809 -0.0495256
CondInfTree 1.000000               -0.0691617 -0.0501063
MARS        5.753e-08 6.541e-08                0.0190553
M5          0.002156  0.001024    0.679609
```

Note that these results are consistent with the patterns shown in Figure 9; 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 10 and 11 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")
```

6 Session Information

- R version 2.11.1 (2010-05-31), `x86_64-apple-darwin9.8.0`
- Locale: `en_US/en_US/en_US/C/en_US/en_US`

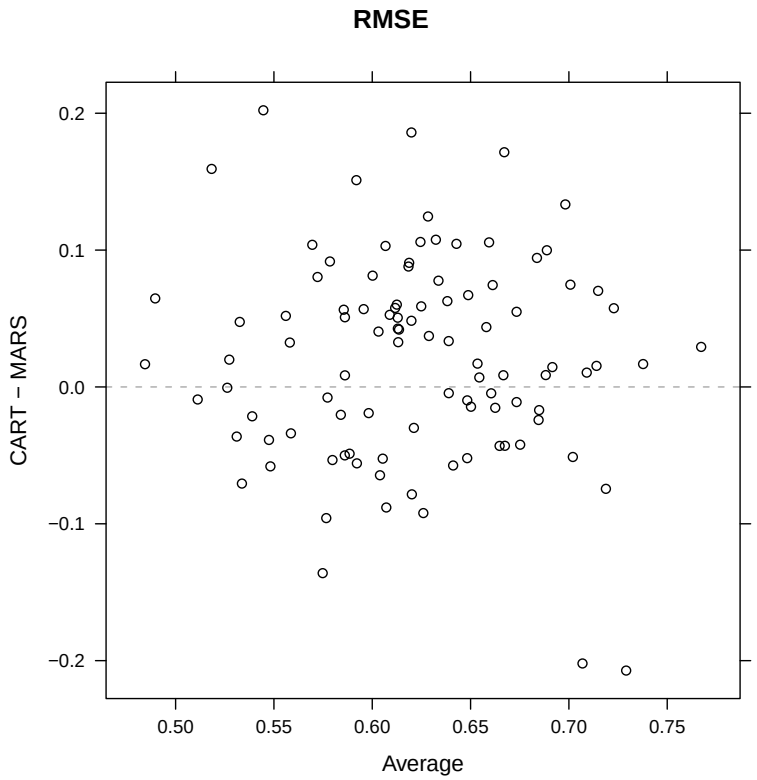

Figure 8: Examples of output from `xyplot(resamps, models = c("CART", "MARS"))`. The averages and differences of the two models is shown for each resampling data set.

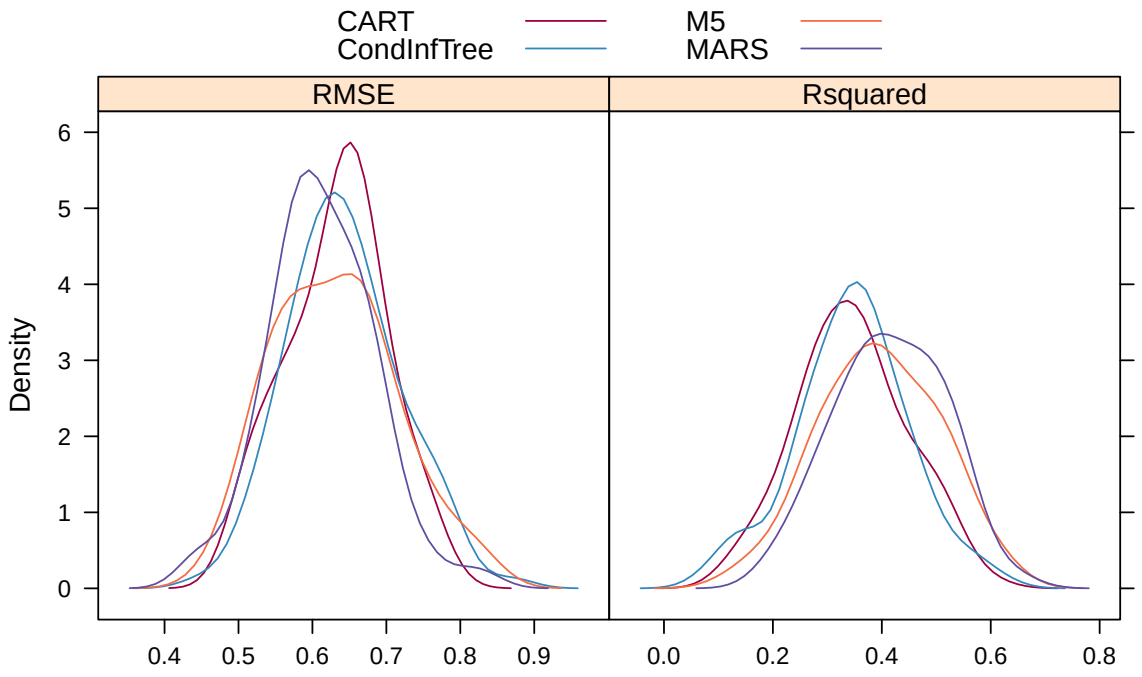

Figure 9: Examples of output from `densityplot(resamps)`. Looking at R^2, M5 rules and MARS appear to be similar to one another but different from the two tree–based models. However, this pattern is inconsistent with the root mean squared error distributions.

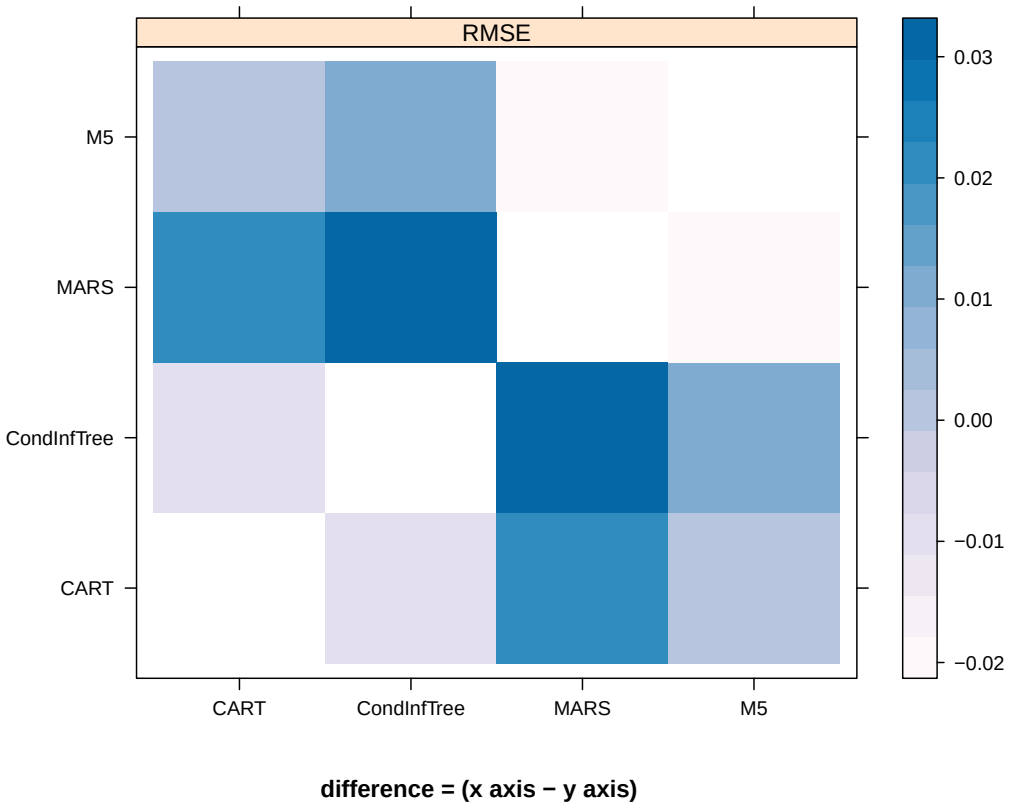

Figure 10: Examples of output from `levelplot(difValues, what = "differences")`. The pair–wise differences in RMSE are shown

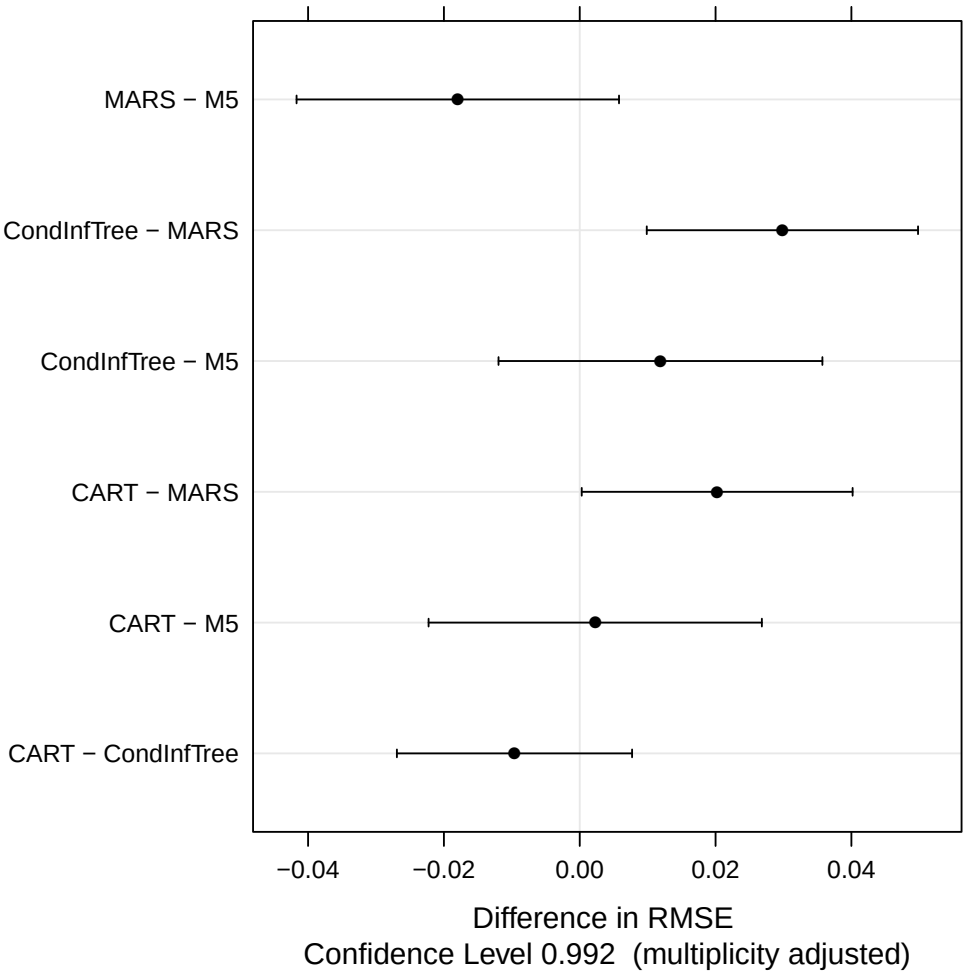

Figure 11: Examples of output from `dotplot(difValues)`. The differences in RMSE and their associated confidence intervals are shown.

- Base packages: base, datasets, graphics, grDevices, grid, methods, splines, stats, stats4, tools, utils
- Other packages: akima 0.5-4, caret 4.89, class 7.3-2, cluster 1.12.3, e1071 1.5-24, earth 2.4-5, ellipse 0.3-5, gam 1.03, gbm 1.6-3.1, Hmisc 3.8-3, ipred 0.8-8, kernlab 0.9-12, klaR 0.6-4, lattice 0.18-8, leaps 2.9, MASS 7.3-6, mlbench 2.1-0, modeltools 0.2-17, mvtnorm 0.9-95, nnet 7.3-1, plotrix 3.0-5, pls 2.1-0, plyr 1.2.1, proxy 0.4-6, randomForest 4.5-36, reshape 0.8.3, rpart 3.1-46, survival 2.35-8
- Loaded via a namespace (and not attached): coin 1.0-17, colorspace 1.0-1, party 0.9-99992, rJava 0.8-7, RWeka 0.4-4, RWekajars 3.7.2-1

7 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

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

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.