The `caret` Package

Max Kuhn
max.kuhn@pfizer.com

November 26, 2007

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

To optimize tuning parameters of models, `train` can be used to fit many predictive models over a grid of parameters and return the "best" model (based on resampling statistics). See Table 1 for the models currently available.

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

- center and scale the training and test set using the predictor means and standard deviations from the training 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 = 0.75)
> 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
```

```
> preProcValues <- apply(trainDescr, 2, processData)
> trainDescr <- applyProcessing(trainDescr, preProcValues)
> testDescr <- applyProcessing(testDescr, preProcValues)
```

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[1] 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 = 0.75, number = 30,
+     verboseIter = FALSE)
> set.seed(2)
```

The first two arguments to `train` are the predictor and outcome data objects, respectively. The third argument, `method`, specifies the type of model. For this model, the tuning parameters are the cost value (`C` and the radius of the RBF (`sigma`)). 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.

```
> svmFit <- train(trainDescr, trainMDRR, method = "svmradial",
+     tuneLength = 4, trControl = fitControl)
> svmFit


Call:
train.default(x = trainDescr, y = trainMDRR, method = "svmradial",
    trControl = fitControl, tuneLength = 4)

264 samples, 50 predictors

largest class: 56.06% (Active)

summary of leave group out cross-validation (30 reps) sample sizes:
    198, 198, 198, 198, 198, 198, ...

LGOCV resampled training results across tuning parameters:

  sigma    C    Accuracy  Kappa   Accuracy SD  Kappa SD  Optimal
  0.00571  0.1  0.574     0.0346  0.0128       0.0323
  0.00571  1    0.829     0.649   0.0438       0.091     *
  0.00571  10   0.814     0.621   0.0467       0.0951
  0.00571  100  0.777     0.553   0.0472       0.0933
```

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

```
Accuracy was used to select the optimal model
```

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.

Alternatively, the 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 our 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, the amount of "shrinkage" in a gradient boosting model is fixed at 0.1 and the other meta–parameters can be manually specified:

```
> gbmGrid <- expand.grid(.interaction.depth = c(1, 3), .n.trees = c(100, 300,
+     500), .shrinkage = 0.1)
> set.seed(3)
> gbmFit <- train(trainDescr, trainMDRR, "gbm", tuneGrid = gbmGrid, trControl = fitControl,
+     verbose = FALSE)
> gbmFit


Call:
train.default(x = trainDescr, y = trainMDRR, method = "gbm",
    verbose = FALSE, trControl = fitControl, tuneGrid = gbmGrid)

264 samples, 50 predictors

largest class: 56.06% (Active)

summary of leave group out cross-validation (30 reps) sample sizes:
    198, 198, 198, 198, 198, 198, ...

LGOCV resampled training results across tuning parameters:

  interaction.depth  n.trees  shrinkage  Accuracy  Kappa  Accuracy SD  Kappa SD  Optimal
  1                  100      0.1        0.809     0.608  0.0414       0.0833
  1                  300      0.1        0.796     0.583  0.049        0.098
  1                  500      0.1        0.784     0.558  0.0448       0.0901
  3                  100      0.1        0.809     0.608  0.0437       0.0876    *
  3                  300      0.1        0.804     0.598  0.0388       0.0791
  3                  500      0.1        0.802     0.594  0.039        0.0802

Accuracy was used to select the optimal model
```

Some notes about the use of train:

- 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 the example above, 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.
- 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, 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 form 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.

- For some models (pls, plsda, earth, rpart, gbm, gamboost, glmboost, blackboost, ctree, pam, enet and lasso), the train function will fit a model that can be used to derive predictions for some sub-models. For example, 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. See Figures 1 and 2 for examples. This is also a function, resampleHist, that will plot a histogram or density plot of the resampled performance estimates for the optimal model. Figure 2 shows and example of this type of plot for the support vector machine example.

- 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, gbmFit$finalModel is the same object that would have been produced using a direct call to the gbm function. 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 the model selection procedure.

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

- method: The resampling method: boot, cv, LOOCV, LGOCV 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).

- number: Either the number of folds or number of resampling iterations

- verboseIter: A logical for printing a training log.

- returnData: A logical for saving the data

- p: For leave-group out cross-validation: the training percentage
- 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.

Note that caret picks the "best" values of the tuning parameters based on resampling measures and does not use the one standard deviation rule typically used by recursive partitioning models. In many cases, the user will need to look at the profiles of performance measures across the tuning parameters to get an idea of the truly best choice.

Also, for leave–one–out cross–validation, no uncertainty estiamtes are given for the resampled performance measures.

Table 1: Models used in train

Model	method Value	Package	Tuning Parameters
Recursive partitioning	rpart	rpart	maxdepth
	ctree	party	mincriterion
Boosted Trees	gbm	gbm	interaction.depth, n.trees, shrinkage
	blackboost	gbm	maxdepth, mstop
	ada	ada	maxdepth, iter, nu
Other Boosted Models	glmboost	mboost	mstop
	gamboost	mboost	mstop
Random forests	rf	randomForest	mtry
	cforest	party	mtry
Bagged Trees	treebag	ipred	None
Neural networks	nnet	nnet	decay, size
Partial least squares	pls	pls, caret	ncomp
Support Vector Machines (RBF kernel)	svmradial	kernlab	sigma, C
Support Vector Machines (polynomial kernel)	svmpoly	kernlab	scale, degree, C
Linear least squares	lm	stats	None
Multivariate adaptive regression splines	earth, mars	earth	degree, nprune
Bagged MARS	bagEarth	caret, earth	degree, nprune
Elastic Net	enet	elasticnet	lambda, fraction
The Lasso	lasso	elasticnet	fraction
Linear discriminant analysis	lda	MASS	None
Logistic/multinomial regression	multinom	nnet	decay
Regularized discriminant analysis	rda	klaR	lambda, gamma
Flexible discriminant analysis (MARS basis)	fda	mda, earth	degree, nprune
Bagged FDA	bagFDA	caret, earth	degree, nprune
k nearest neighbors	knn3	caret	k
Nearest shrunken centroids	pam	pamr	threshold
Naive Bayes	nb	klaR	usekernel
Generalized partial least squares	gpls	gpls	K.prov
Learned vector quantization	lvq	class	k

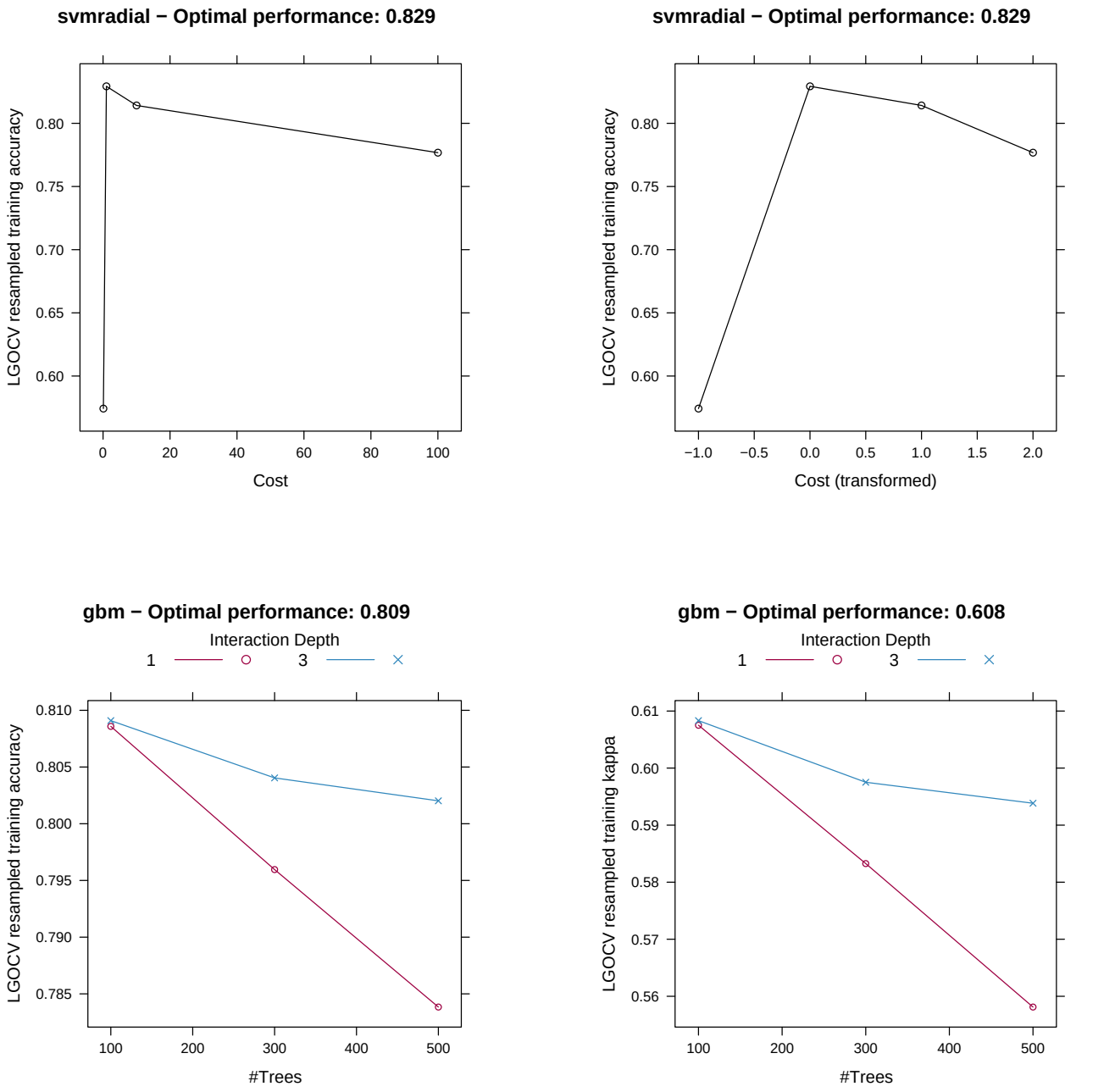

Figure 1: Examples of output from `plot.tain`. **top left** a plot produced using `plot(svmFit)` showing the relationship between SVM cost parameter and the resampled classification accuracy. Although this model has two tuning parameters, a constant value for the parameter `sigma` was used. **top right** 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")`

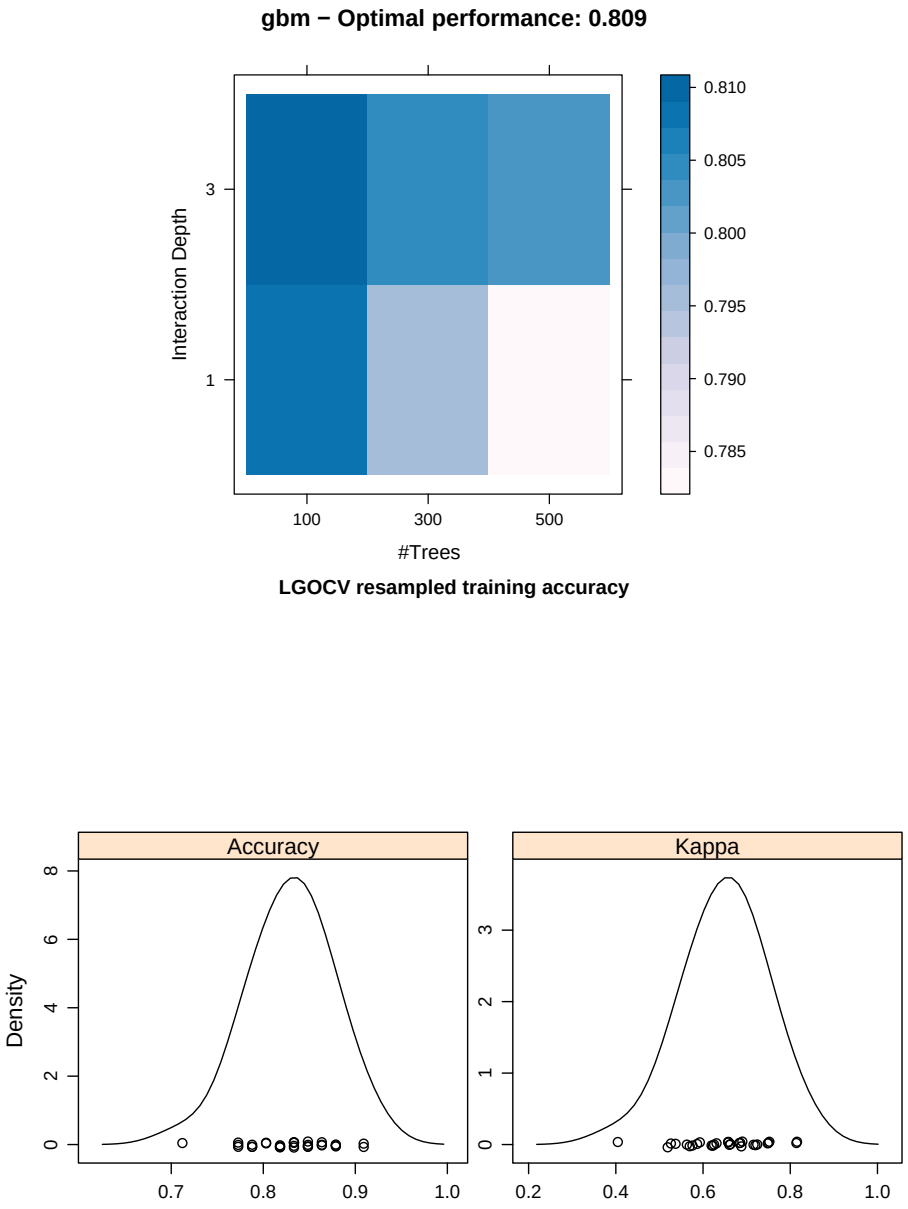

Figure 2: More examples. **top:** A plot produced using `plot(gbmFit metric = "Kappa", plotType = "level")` showing the relationship (using a `levelplot`) between the number of boosting iterations, the interaction depth and the resampled estimate of the Kappa statistic. **bottom:** A plot of the resampling estimates of performance from the optimal support vector machine model produced using `resampleHist(svmFit, type = "density", layout = c(2, 1), adjust = 1.5)`.

2 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. Alternatively, predictions can be extracted from a series of model using the function `extractPrediction`.

For example, we can load the Boston Housing data:

```
> library(mlbench)
> data(BostonHousing)
> bhDesignMatrix <- model.matrix(medv ~ . - 1, BostonHousing)
```

split the data into random training/test groups:

```
> set.seed(4)
> inTrain <- createDataPartition(BostonHousing$medv, p = 0.8, list = FALSE,
+     times = 1)
> trainBH <- bhDesignMatrix[inTrain, ]
> testBH <- bhDesignMatrix[-inTrain, ]
> trainMedv <- BostonHousing$medv[inTrain]
> testMedv <- BostonHousing$medv[-inTrain]
```

fit a regression tree, random forest and multivariate adaptive regression spline model (none of these models require centering and scaling):

```
> rpartFit <- train(trainBH, trainMedv, "rpart", tuneLength = 9,
+     trControl = trainControl(verboseIter = FALSE))
> marsFit <- train(trainBH, trainMedv, "mars", trControl = trainControl(verboseIter = FALSE))
> rfFit <- train(trainBH, trainMedv, "rf", trControl = trainControl(verboseIter = FALSE,
+     method = "oob"))
```

obtain predictions for the test samples for both models:

```
> bhPredictions <- extractPrediction(list(rpartFit, marsFit, rfFit),
+     testX = testBH, testY = testMedv)
> bhTestPred <- bhPredictions[bhPredictions$dataType != "Resampled",
+     ]
> str(bhPredictions)

'data.frame':        1518 obs. of  4 variables:
 $ obs     : num  16.5 15 13.6 14.5 13.9 16.6 14.8 12.7 13.2 13.1 ...
 $ pred    : num  18.1 18.1 18.1 18.1 18.1 ...
 $ model   : Factor w/ 3 levels "mars","rf","rpart": 3 3 3 3 3 3 3 3 3 3 ...
 $ dataType: Factor w/ 2 levels "Test","Training": 2 2 2 2 2 2 2 2 2 2 ...
```

and evaluate the test set:

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

model: mars
    RMSE Rsquared
3.525408 0.853236
------------------------------------------------------------
model: rf
     RMSE  Rsquared
2.0695123 0.9542742
------------------------------------------------------------
model: rpart
     RMSE  Rsquared
4.0730971 0.8046558
```

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 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.

3 Evaluating Models

A function, `postResample`, can be used obtain the same performance measures as generated by `train`.

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, 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] 0.7933333
```

```
> confusionMatrix(mbrrPredictions$pred, mbrrPredictions$obs)
```

```
Confusion Matrix and Statistics

          Reference
Prediction Active Inactive
  Active      119       27
  Inactive     31       87

Statistics:
Accuracy       0.7803
Kappa          0.5542
Sensitivity    0.7933
Specificity    0.7632
Pos Pred Value 0.8151
Neg Pred Value 0.7373

Class = Active was used to define a positive result
```

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).

ROC Curves

The function roc[2] 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] 0.8749415
```

See Figure 4 for an example.

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 5 and 4 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 3).

4 References

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.

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

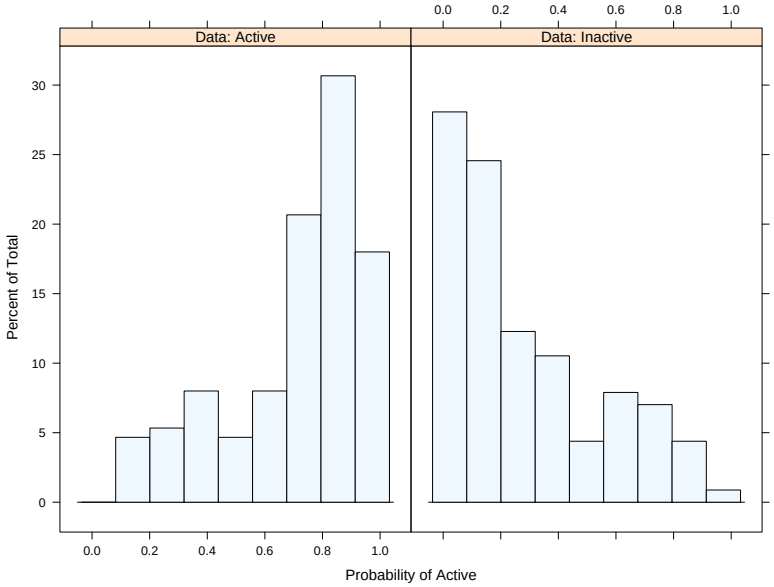

Figure 3: The predicted class probabilities from a support vector machine fit for the MBRR test set. This plot was created using `plotClassProbs(mbrrProbs)`.

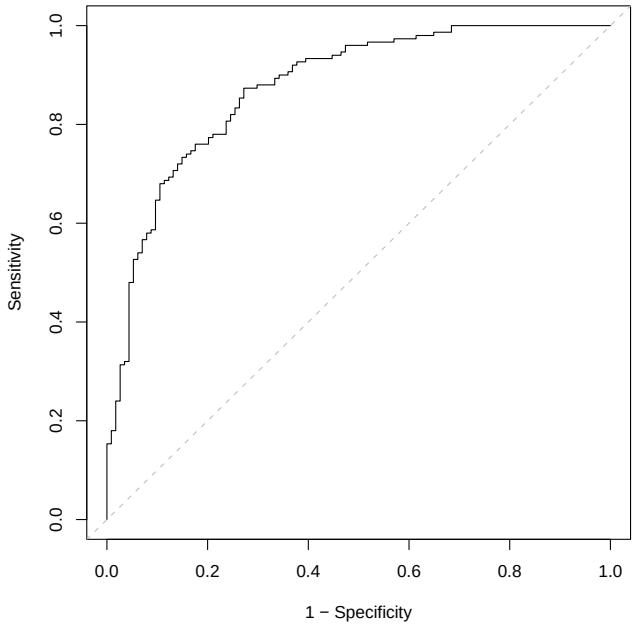

Figure 4: An ROC curve from the predicted class probabilities from a support vector machine fit for the MBRR test set.

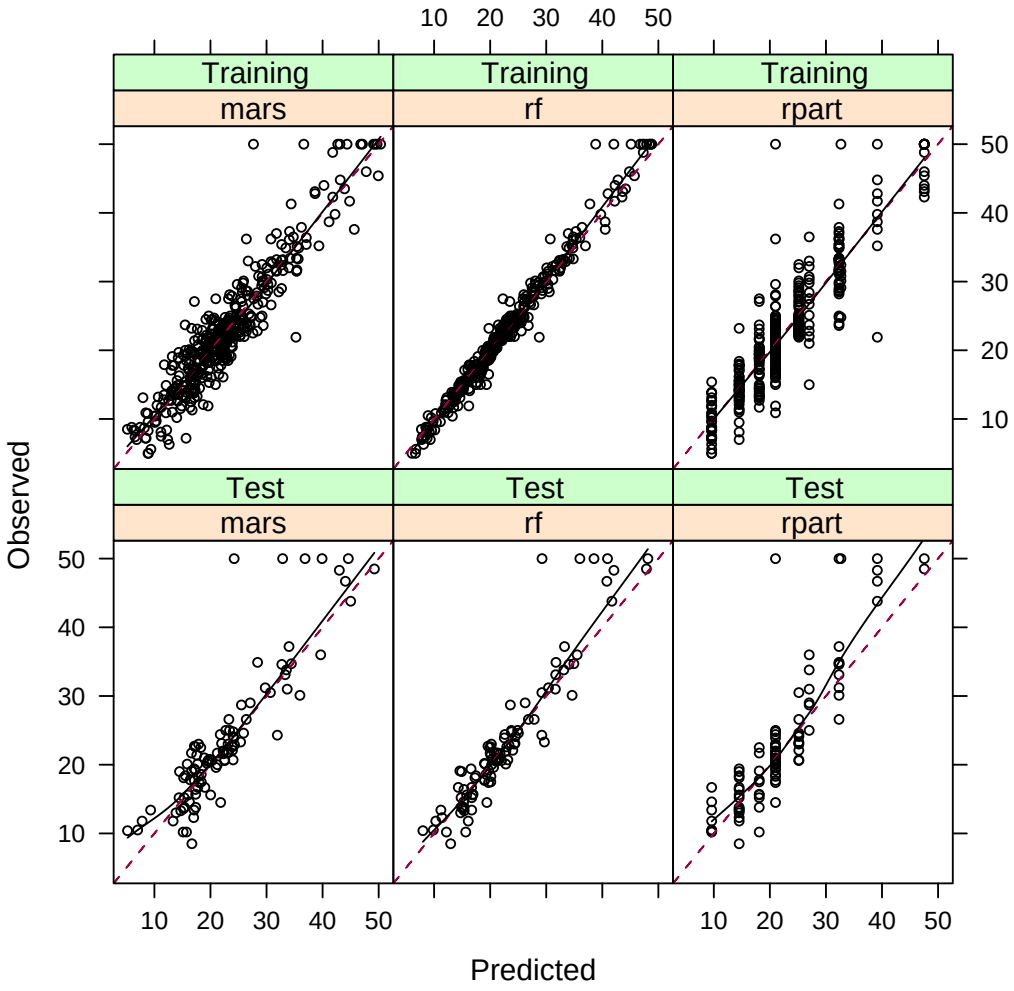

Figure 5: The results of using `plotObsVsPred` to show plots of the observed median home price against the predictions from two models. The plot shows the training and test sets in the same Lattice plot