The `caret` Package

Max Kuhn
max.kuhn@pfizer.com

January 25, 2009

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 <- preProcess(trainDescr)
> trainDescr <- predict(preProcValues, trainDescr)
> testDescr <- predict(preProcValues, testDescr)
```

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,
+     returnResamp = "all", 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 (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.

```
> 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

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

LGOCV resampled training results across tuning parameters:

  C    sigma   Accuracy  Kappa   Accuracy SD  Kappa SD  Selected
  0.1  0.0222  0.581     0.0522  0.0212       0.0533
  1    0.0222  0.837     0.665   0.0406       0.0846    *
  10   0.0222  0.807     0.609   0.0562       0.114
  100  0.0222  0.807     0.609   0.0533       0.109
```

[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 using the largest value.

The final values used in 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.

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

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  Selected
  1                  100      0.1        0.81      0.611  0.0479       0.0958    *
  1                  300      0.1        0.798     0.587  0.0416       0.0831
  1                  500      0.1        0.791     0.573  0.0431       0.0862
  3                  100      0.1        0.804     0.598  0.0427       0.0864
  3                  300      0.1        0.799     0.59   0.0417       0.0835
  3                  500      0.1        0.799     0.59   0.043        0.0861

Accuracy was used to select the optimal model using the largest value.

The final values used in the model were interaction.depth = 1, n.trees = 100 and shrinkage = 0.1.
```

Some notes about the use of train:

- 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 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. 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, 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.

- For some models (`pls`, `plsda`, `earth`, `rpart`, `gbm`, `gamboost`, `glmboost`, `blackboost`, `ctree`, `pam`, `superpc`, `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.

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
Recursive Partitioning	rpart	rpart	maxdepth
	ctree	party	mincriterion
	ctree2	party	maxdepth
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
Elastic Net (glm)	glmnet	glmnet	alpha, lambda
Neural Networks	nnet	nnet	decay, size
Partial Least Squares	pls	pls, caret	ncomp
Sparse Partial Least Squares	spls	spls, caret	K, eta, kappa
Support Vector Machines (RBF kernel)	svmRadial	kernlab	sigma, C
Support Vector Machines (polynomial kernel)	svmPoly	kernlab	scale, degree, C
Gaussian Processes (RBF kernel)	gausssprRadial	kernlab	sigma
Gaussian Processes (polynomial kernel)	gausssprPoly	kernlab	scale, degree
	Regression Models		
Linear Least Squares	lm	stats	None
Multivariate Adaptive Regression Splines	earth, mars	earth	degree, nprune
Bagged MARS	bagEarth	caret, earth	degree, nprune
M5 Rules	M5Rules	RWeka	pruned
Elastic Net	enet	elasticnet	lambda, fraction
The Lasso	lasso	elasticnet	fraction
Projection Pursuit	ppr	stats	nterms

(continued on next page)

Table 1: Models used in train

Model	method Value	Package	Tuning Parameters
Regression			
Penalized Linear Models	penalized	penalized	lambda1, lambda2
Regression Splines			
Relevance Vector Machines (RBF kernel)	rvmRadial	kernlab	sigma
Relevance Vector Machines (polynomial kernel)	rvmPoly	kernlab	scale, degree
Supervised Principal Components	superpc	superpc	n.components, threshold
	Classification Models		
Linear Discriminant Analysis	lda	MASS	None
Quadratic Discriminant Analysis	qda	MASS	None
Stabilised Linear Discriminant Analysis	slda	ipred	None
Shrinkage Linear Discriminant Analysis	sda	sda	diagonal
Sparse Linear Discriminant Analysis	sparseLDA	sparseLDA	NumVars, lambda
Stepwise Diagonal Discriminant Analysis	sddaLDA, sddaQDA	SDDA	None
Regularized Discriminant Analysis	rda	klaR	lambda, gamma
Mixture Discriminant Analysis	mda	mda	subclasses
Penalized Discriminant Analysis	pda	mda	lambda
	pda2	mda	df
Flexible Discriminant Analysis (MARS basis)	fda	mda, earth	degree, nprune
Bagged FDA	bagFDA	caret, earth	degree, nprune
Logistic/Multinomial Regression	multinom	nnet	decay
LogitBoost	logitboost	caTools	nIter
Logistic Model Trees	LMT	RWeka	iter
C4.5 decision trees	J48	RWeka	C
Least Squares Support Vector	lssvmRadial	kernlab	sigma

(continued on next page)

Table 1: Models used in train

Model	method Value	Package	Tuning Parameters
Machines (RBF kernel)			
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

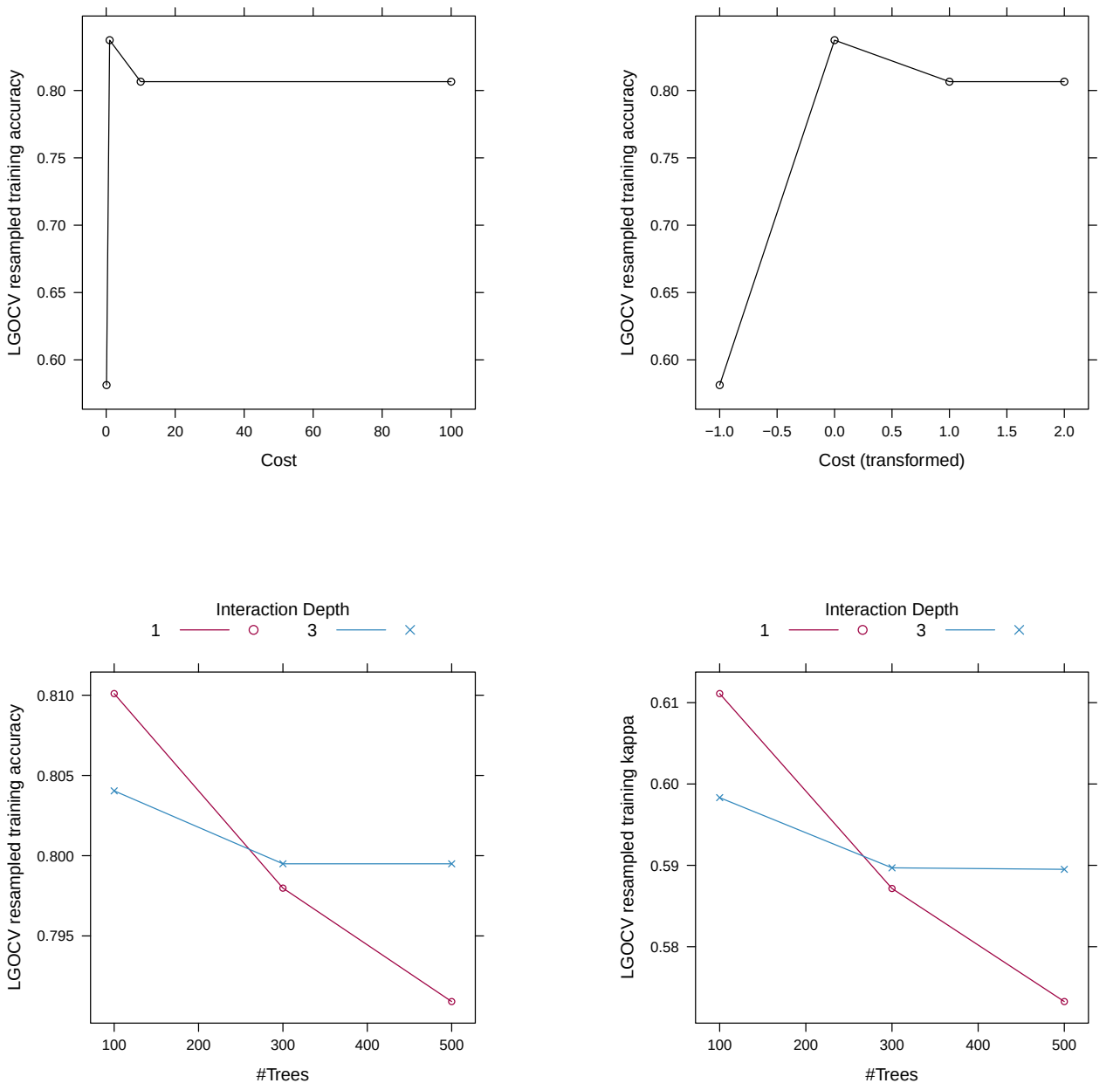

Figure 1: Examples of output from `plot.tain`. **top left** a plot produced using `plot(svmFit)` showing the relationship between SVM cost parameter and the resampled classification accuracy. Although this model has two tuning parameters, a constant value for the parameter `sigma` was used. **top right**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

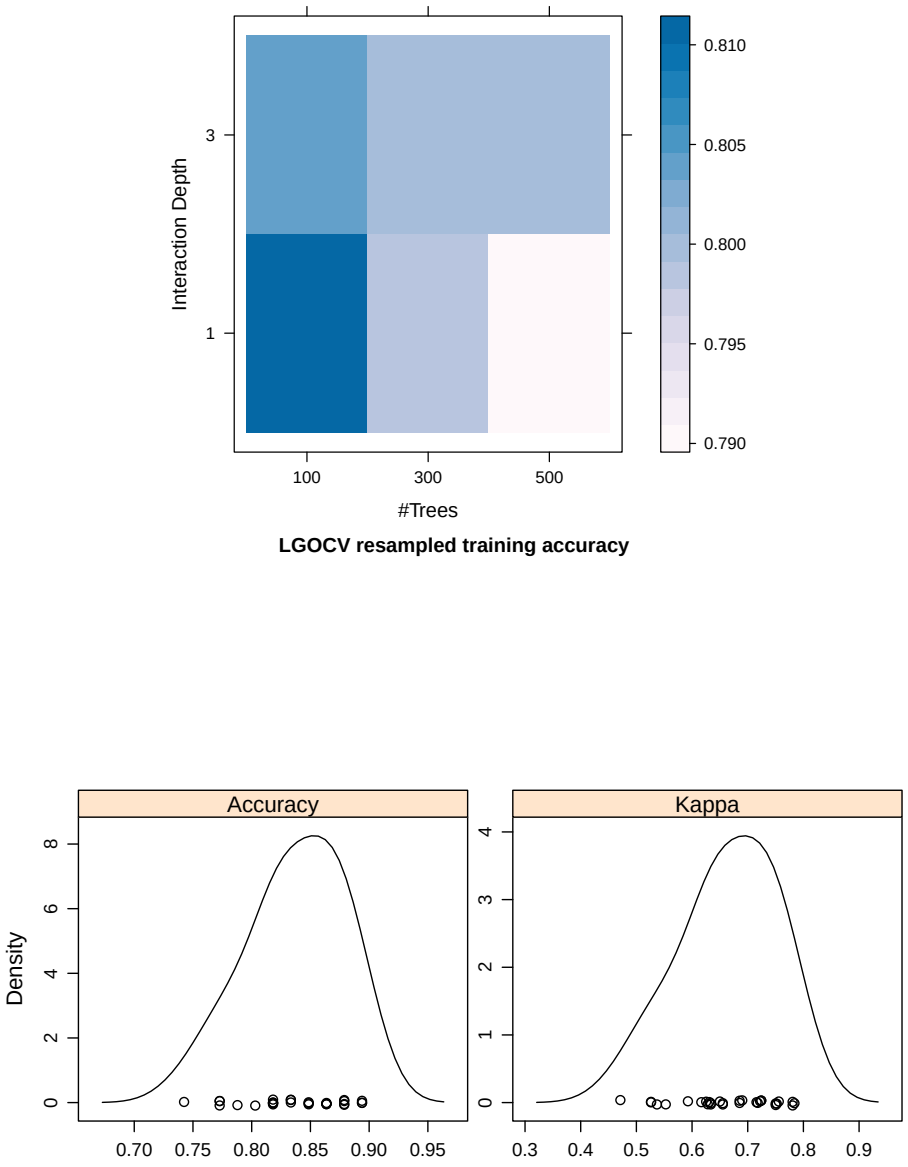

Figure 2: More examples. **top:** A plot produced using `plot(gbmFit metric = "Kappa", plotType = "level")` showing the relationship (using a `levelplot`) between the number of boosting iterations, the interaction depth and the resampled estimate of the Kappa statistic. **bottom:** A plot of the resampling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

2 Customizing the Tuning Process

There are a few ways to customize the process of selecting tuning/complexity parameters. First, as previously shown with the boosted tree code, you can choose specific values of the tuning parameter (instead of the defaults).

Secondly, 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.
- `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.

As an example, classification accuracy in two–class problems can be decomposed into sensitivity and specificity. We can use these values to tune the parameters using the following function:

```
> newSummary <- function(data, lev, model) {
+     out <- c(sensitivity(data[, "pred"], data[, "obs"], lev[1]),
+         specificity(data[, "pred"], data[, "obs"], lev[2]))
+     names(out) <- c("Sens", "Spec")
+     out
+ }
```

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

```
> fitControl$summaryFunction <- newSummary
> set.seed(2)
> svmNew <- train(trainDescr, trainMDRR, method = "svmRadial",
+     metric = "Spec", tuneLength = 4, trControl = fitControl)
> svmNew

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

264 samples
50 predictors

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

LGOCV resampled training results across tuning parameters:

  C    sigma   Sens   Spec    Sens SD  Spec SD  Selected
  0.1  0.0222  1      0.0471  0        0.0483
  1    0.0222  0.904  0.753   0.053    0.0825
  10   0.0222  0.812  0.8     0.0686   0.0884
  100  0.0222  0.812  0.8     0.0613   0.0837   *

Spec was used to select the optimal model using the largest value.

The final values used in the model were C = 100 and sigma = 0.0222.
```

Based on this model and the original SVM model, 60% accuracy can be achieved by being very biased towards sensitivity.

The third 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 overfit 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 SVM model on the basis of specificity, we would choose a cost value of 100, the most complex model. Lower cost values would produce approximately the same performance with less complex models (with the exception of cost = 0.1). 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 cost values based on 2% and 6% losses of performance:

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

best model within 2 pct of best:

> svmNew$results[whichTwoPct, ]

   C      sigma      Sens Spec     SensSD     SpecSD
3 10 0.02222875 0.8117117  0.8 0.06862484 0.08844885

> whichSixPct <- tolerance(svmNew$results, "Spec", 6, TRUE)
> cat("\n\nbest model within 6 pct of best:\n")

best model within 6 pct of best:

> svmNew$results[whichSixPct, ]

  C      sigma      Sens      Spec     SensSD   SpecSD
2 1 0.02222875 0.9036036 0.7528736 0.05296353 0.082542
```

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 most cases, 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.

Finally, the function `trainControl`, generates parameters that further control how models are resampled 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). Also, for leave–one–out cross–validation, no uncertainty estimates are given for the resampled performance measures.
- `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.
- `summaryFunction`: previously mentioned
- `selectionFunction`: previously mentioned
- `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.

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

split the data into random training/test groups:

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

fit partial least squares and multivariate adaptive regression spline models:

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

To obtain predictions for the PLS model, predict.mvr can be used. In this case, the number of components must be manually specified or all of the sub-models are predicted:

```
> plsPred1 <- predict(plsFit$finalModel, newdata = as.matrix(testBH))
> dim(plsPred1)

[1] 99  1  9
```

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

```
> plsPred2 <- predict(plsFit, newdata = testBH)
> length(plsPred2)
```

```
[1] 99
```

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] 34.4 20.2 18.2 14.4 15.7 ...
```

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
408 34.7 30.15640   pls     Test
409 21.7 21.87263   pls     Test
410 20.2 16.06634   pls     Test
411 15.2 16.01122   pls     Test
412 15.6 15.80842   pls     Test
413 14.5 17.94325   pls     Test
```

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

```
model: earth
    RMSE Rsquared
4.678437 0.793299
------------------------------------------------------------
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 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 (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

            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 : 0.553

       'Postive' 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 accuarcy rate is greater than the rate of the largest class. Also, the prevalence of the "postivie 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:

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

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).

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] 0.8724269
```

See Figure 4 for an example.

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

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).

5 References

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

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.

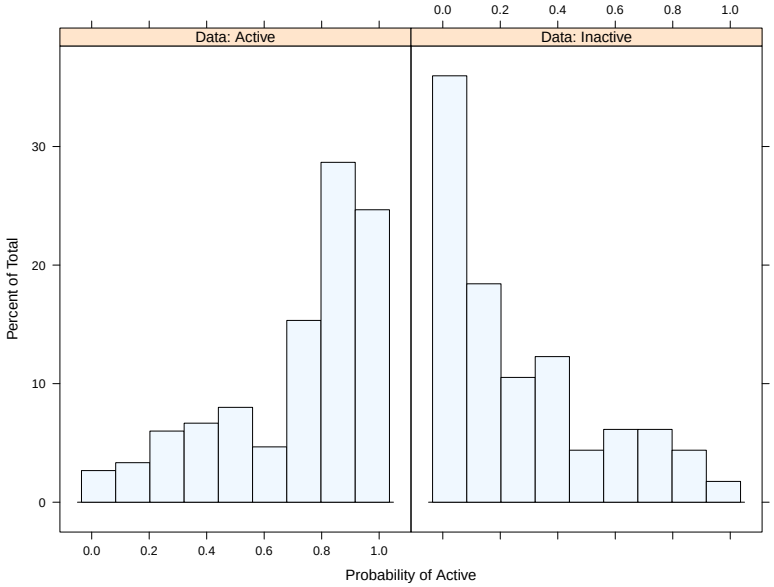

Figure 3: The predicted class probabilities from a support vector machine fit for the MBRR test set. This plot was created using `plotClassProbs(mbrrProbs)`.

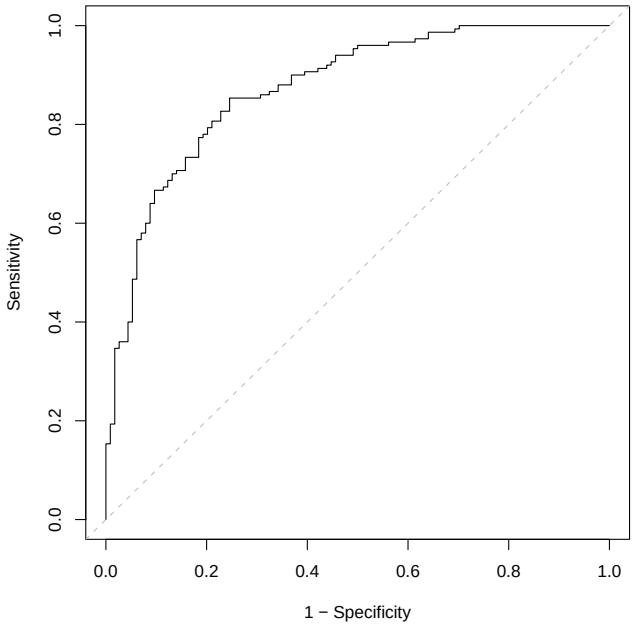

Figure 4: An ROC curve from the predicted class probabilities from a support vector machine fit for the MBRR test set.

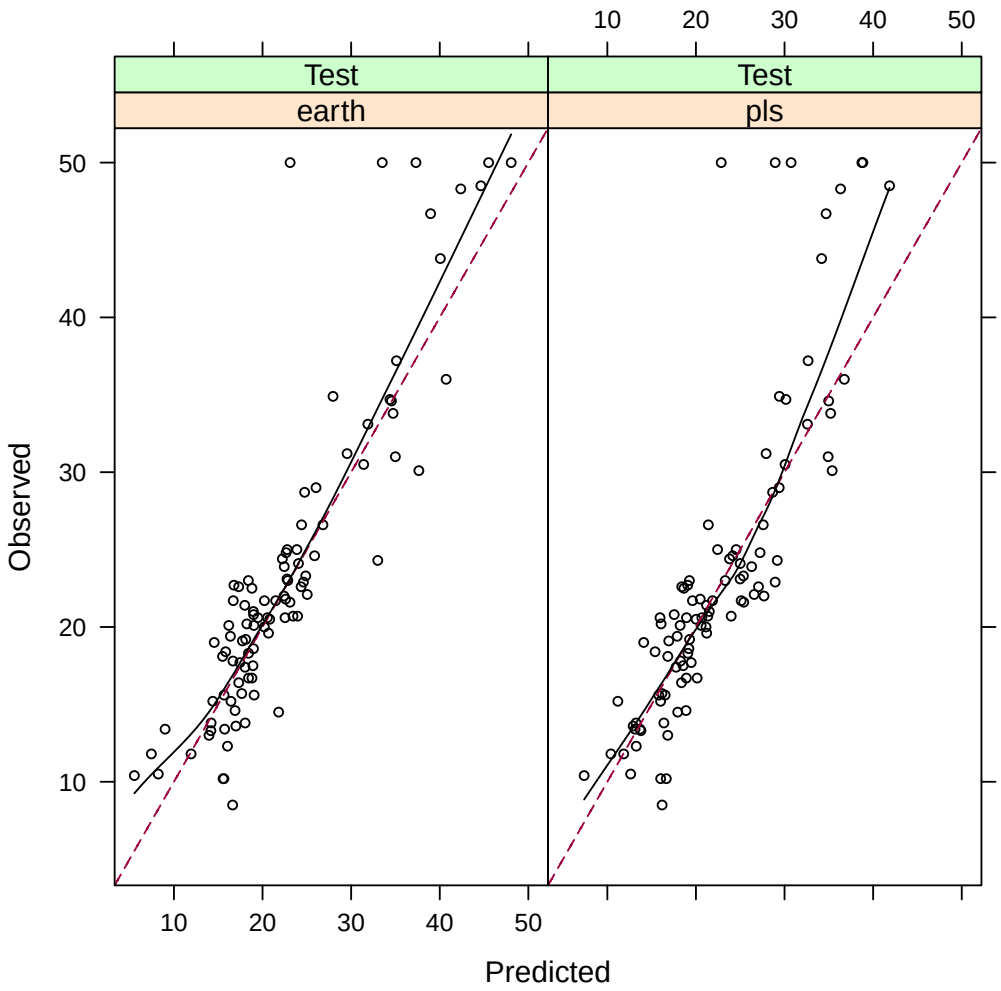

Figure 5: The results of using `plotObsVsPred` to show plots of the observed median home price against the predictions from two models. The plot shows the training and test sets in the same Lattice plot