Contents

1 The caret Package

- The caret package (short for Classification And REgression Training) was created to streamline the process for building and evaluating predictive models. Using the package, a practitioner can quickly evaluate many different types of models to find the more appropriate tool for their data.

- The beauty of R is that it provides a large and diverse set of modeling packages. However, since these packages are created by many different people over time, there are a minimal set of conventions that are common to each model. For example, Table B.1 shows the syntax for calculating class probabilities for several different types of classification models.

- Remembering the syntactical variations can be difficult and this discourages users from evaluating a variety of models. One method to reduce this complexity is to provide a unified interface to functions for model building and prediction. caret provides such an interface for across a wide vary of models (over 140). The package also provides many options for data pre-processing and resampling-based parameter tuning techniques (Chaps. 3 and 4).

- In this text, resampling is the primary approach for optimizing predictive models with tuning parameters. To do this, many alternate versions of the training set are used to train the model and predict a holdout set. This process is repeated many times to get performance estimates that generalize to new data sets.

- Each of the resampled data sets is independent of the others, so there is no formal requirement that the models must be run sequentially. If a computer with multiple processors or cores is available, the computations could be spread across these workers to increase the computational efficiency. caret leverages one of the parallel processing frameworks in R to do just this.

- The foreach package allows R code to be run either sequentially or in parallel

using several different technologies, such as the multicore or Rmpi packages (see *Schmidberger et al. (2009)* for summaries and descriptions of the available options). There are several R packages that work with foreach to implement these techniques, such as **doMC** (for multicore) or doMPI (for Rmpi).

- To tune a predictive model using multiple workers, the syntax in the caret package functions (e.g., train, rfe or sbf) does not change. A separate function is used to register the parallel processing technique and specify the number of workers to use. For example, to use the multicore package (not available on Windows) with five cores on the same machine, the package is loaded and then registered:

```
library(doMC)
registerDoMC(cores = 5)
## All subsequent models are then run in parallel
model <- train(y ~ ., data = training, method = "rf")
```

- The syntax for other packages associated with foreach is very similar. Note that as the number of workers increases, the memory required also increases. For example, using five workers would keep a total of six versions of the data in memory. If the data are large or the computational model is demanding, performance can be affected if the amount of required memory exceeds the physical amount available.

- Does this help reduce the time to fit models? The job scheduling data (Chap. 17) was modeled multiple times with different number of workers for several models. Random forest was used with 2,000 trees and tuned over 10 values of mtry. Variable importance calculations were also conducted during each model fit. Linear discriminant analysis was also run, as was a costsensitive radial basis function support vector machine (tuned over 15 cost values).

- All models were tuned using five repeats of 10-fold cross-validation. The results are shown in Fig.B.1. The y-axis corresponds to the total execution time (encompassing model tuning and the final model fit) versus the number of workers.

- Random forest clearly took the longest to train and the LDA models were very computationally efficient. The total time (in minutes) decreased as the number of workers increase but stabilized around seven workers. The data for this plot were generated in a randomized fashion so that there should be no bias in the run order.

- The bottom right panel shows the speedup which is the sequential time divided by the parallel time. For example, a speedup of three indicates that the parallel version was three times faster than the sequential version. At best, parallelization can achieve linear speedups; that is, for M workers, the parallel time is 1/M. For these models, the speedup is close to linear until four or five workers are used. After this, there is a small improvement in performance. Since LDA is already computationally efficient, the speed-up levels off more rapidly than the other models. While not linear, the decrease in execution time is helpfula nearly 10 h model fit was decreased to about 90 min.

- Note that some models, especially those using the RWeka package, may not be able to be run in parallel due to the underlying code structure. One additional trick that train exploits to increase computational efficiency is to use sub-models; a single model fit can produce predictions formultiple tuning parameters. For example, in most implementations of boosted models, a model trained on B boosting iterations can produce predictions for models for iterations less than B. For the grant data, a gbm model was fit that evaluated 200 distinct combinations of the three tuning parameters (see Fig. 14.10).

- In reality, train only created objects for 40 models and derived the other predictions from these objects. More detail on the caret package can be found in Kuhn

(2008) or the four extended manuals (called vignettes) on the package web site (Kuhn 2010).

2 The caret package

Prediction is a central problem in machine learning that involves inducing a model from a set of training instances that is then applied to future instances to predict a target variable of interest. Several commonly used predictive algorithms, such as logistic regression, neural networks, decision trees, and Bayesian networks, typically induce a single model from a training set of instances, with the intent of applying it to all future instances.

We call such a model a population-wide model because it is intended to be applied to an entire population of future instances. A population-wide model is optimized to predict well on average when applied to expected future instances. Recent research in machine learning has shown that inducing models that are specic to the particular features of a given instance can improve predictive performances (Gottrup et al., 2005). We call such a model an instance-specic model since it is constructed specically for a particular instance (case).

The structure and parameters of an instance-specic model are specialized to the particular features of an instance, so that it is optimized to predict especially well for that instance. The goal of inducing an instance-specic model is to obtain optimal prediction for the instance at hand. This is in contrast to the induction of a population-wide model where the goal is to obtain optimal predictive performance on average on all future instances.

The caret package (short for Classification And REgression Training) is a set of functions that attempt to streamline the process for creating predictive models. The package contains tools for:

- data splitting

- pre-processing
- feature selection
- model tuning using resampling
- variable importance estimation

3 AppliedPredictiveModeling

This package contains several data set and A few functions from Kuhn's and Johnson's Springer book *'Applied Predictive Modeling'*.

3.1 Fuel Economy Data Set

The http://fueleconomy.gov website, run by the U.S. Department of Energys Ofce of Energy Efciency and Renewable Energy and the U.S. Environmental Protection Agency, lists different estimates of fuel economy for passenger cars and trucks. For each vehicle, various characteristics are recorded such as the engine displacement or number of cylinders. Along with these values, laboratory measurements are made for the city and highway miles per gallon (MPG) of the car.

3.2 Hepatic Data Set

This data set was used to develop a model for predicting compounds probability of causing hepatic injury (i.e. liver damage). This data set consisted of 281 unique compounds; 376 predictors were measured or computed for each. The response was categorical (either "None", "Mild" or "Severe"),and was highly unbalanced.

This kind of response often occurs in pharmaceutical data because companies steer away from creating molecules that have undesirable characteristics. Therefore, well-behaved molecules often greatly outnumber undesirable molecules. The predictors

consisted of measurements from 184 biological screens and 192 chemical feature predictors. The biological predictors represent activity for each screen and take values between 0 and 10 with a mode of 4. The chemical feature predictors represent counts of important sub-structures as well as measures of physical properties that are thought to be associated with hepatic injury.

4 Bagging

The `bag` function offers a general platform for bagging classification and regression models. Like rfe and sbf, it is open and models are specified by declaring functions for the model fitting and prediction code (and several built-in sets of functions exist in the package). The function bagControl has options to specify the functions (more details below).

The function also has a few non-standard features:

- The argument `var` can enable random sampling of the predictors at each bagging iteration. This is to de-correlate the bagged models in the same spirit of random forests (although here the sampling is done once for the whole model). The default is to use all the predictors for each model.

- The `bagControl` function has a logical argument called downSample that is useful for classification models with severe class imbalance. The bootstrapped data set is reduced so that the sample sizes for the classes with larger frequencies are the same as the sample size for the minority class.

- If a parallel backend for the foreach package has been loaded and registered, the bagged models can be trained in parallel.

5 Bagging and Boosting

These are different approaches to improve the performance of your model (so-called meta-algorithms):

Bagging (stands for Bootstrap Aggregation) is the way decrease the variance of your prediction by generating additional data for training from your original dataset using combinations with repetitions to produce multisets of the same cardinality/size as your original data. By increasing the size of your training set you can't improve the model predictive force, but just decrease the variance, narrowly tuning the prediction to expected outcome.

Boosting is a an approach to calculate the output using several different models and then average the result using a weighted average approach. By combining the advantages and pitfalls of these approaches by varying your weighting formula you can come up with a good predictive force for a wider range of input data, using different narrowly tuned models.

Stacking is a similar to boosting: you also apply several models to you original data. The difference here is, however, that you don't have just an empirical formula for your weight function, rather you introduce a meta-level and use another model/approach to estimate the input together with outputs of every model to estimate the weights or, in other words, to determine what models perform well and what badly given these input data. As you see, these all are different approaches to combine several models into a better one, and there is no single winner here: everything depends upon your domain and what you're going to do. You can still treat stacking as a sort of more advances boosting, however, the difficulty of finding a good approach for your meta-level makes it difficult to apply this approach in practice.

Short examples of each:

Bagging: Ozone data. Boosting: is used to improve optical character recognition (OCR) accuracy. Stacking: is used in K-fold cross validation algorithms.

Bagging and boosting are meta-algorithms that pool decisions from multiple classiers

5.1 Overview on Bagging

- Invented by Leo Breiman: Bootstrap aggregating.
- L. Breiman, Bagging predictors, Machine Learning, 24(2):123-140, 1996.
- Majority vote from classiers trained on bootstrap samples of the training data.
- Generate B bootstrap samples of the training data: random sampling with replacement.
- Train a classier or a regression function using each bootstrap sample.
- For classication: majority vote on the classication results.
- For regression: average on the predicted values.
- Reduces variation.
- Improves performance for unstable classiers which vary signicantly with small changes in the data set, e.g., CART.
- Found to improve CART a lot, but not the nearest neighbor classifier.

5.2 Overview on Boosting

- Iteratively learning weak classiers
- Final result is the weighted sum of the results of weak classiers.
- Many dierent kinds of boosting algorithms: Adaboost (Adaptive boosting) by Y. Freund and R. Schapire is the first.
- Examples of other boosting algorithms:
- LPBoost: Linear Programming Boosting is a margin-maximizing classication algorithm with boosting.

- BrownBoost: increase robustness against noisy datasets. Discard points that are repeatedly misclassied.
- LogitBoost: J. Friedman, T. Hastie and R. Tibshirani, Additive logistic regression: a statistical view of boosting, Annals of Statistics, 28(2), 337-407, 2000.

```
library(mlbench)
data(BostonHousing)

lmFit <- train(medv ~ . + rm:lstat,
               data = BostonHousing,
               "lm")

library(rpart)
rpartFit <- train(medv ~ .,
                  data = BostonHousing,
                  "rpart",
                  tuneLength = 9)
```

6 Partitioning Data

First, we split the data into two groups: a ***training set*** and a ***test set***. To do this, the `createDataPartition` function is used.

- Simple Splitting Based on the Outcome
- Splitting Based on the Predictors

- Data Splitting for Time Series

6.1 Classification Example

```
data(iris)
TrainData <- iris[,1:4]
TrainClasses <- iris[,5]

knnFit1 <- train(TrainData, TrainClasses,
                 method = "knn",
                 preProcess = c("center", "scale"),
                 tuneLength = 10,
                 trControl = trainControl(method = "cv"))

knnFit2 <- train(TrainData, TrainClasses,
                 method = "knn",
                 preProcess = c("center", "scale"),
                 tuneLength = 10,
                 trControl = trainControl(method = "boot"))



library(MASS)
nnetFit <- train(TrainData, TrainClasses,
                 method = "nnet",
                 preProcess = "range",
                 tuneLength = 2,
                 trace = FALSE,
                 maxit = 100)
```

7 Kappa Statistic

The Kappa statistic (or value) is a metric that compares an Observed Accuracy with an Expected Accuracy (random chance). The kappa statistic is used not only to evaluate a single classifier, but also to evaluate classifiers amongst themselves. In addition, it takes into account random chance (agreement with a random classifier), which generally means it is less misleading than simply using accuracy as a metric (an Observed Accuracy of 80

7.1 Computation

Cats Dogs Cats— 10 — 7 — Dogs— 5 — 8 — Assume that a model was built using supervised machine learning on labeled data. This doesn't always have to be the case; the kappa statistic is often used as a measure of reliability between two human raters. Regardless, columns correspond to one "rater" while rows correspond to another "rater". In supervised machine learning, one "rater" reflects ground truth (the actual values of each instance to be classified), obtained from labeled data, and the other "rater" is the machine learning classifier used to perform the classification. Ultimately it doesn't matter which is which to compute the kappa statistic, but for clarity's sake lets say that the columns reflect ground truth and the rows reflect the machine learning classifier classifications.

From the confusion matrix we can see there are 30 instances total ($10 + 7 + 5 + 8 = 30$). According to the first column 15 were labeled as Cats ($10 + 5 = 15$), and according to the second column 15 were labeled as Dogs ($7 + 8 = 15$). We can also see that the model classified 17 instances as Cats ($10 + 7 = 17$) and 13 instances as Dogs ($5 + 8 = 13$).

Observed Accuracy is simply the number of instances that were classified correctly

throughout the entire confusion matrix, i.e. the number of instances that were labeled as Cats via ground truth and then classified as Cats by the machine learning classifier, or labeled as Dogs via ground truth and then classified as Dogs by the machine learning model. To calculate Observed Accuracy, we simply add the number of instances that the machine learning classifier agreed with the ground truth label, and divide by the total number of instances. For this confusion matrix, this would be 0.6 ((10 + 8) / 30 = 0.6).

Before we get to the equation for the kappa statistic, one more value is needed: the Expected Accuracy. This value is defined as the accuracy that any random classifier would be expected to achieve based on the confusion matrix. The Expected Accuracy is directly related to the number of instances of each class (Cats and Dogs), along with the number of instances that the machine learning classifier agreed with the ground truth label. To calculate Expected Accuracy for our confusion matrix, first multiply the marginal frequency of Cats for one "rater" by the marginal frequency of Cats for the second "rater", and divide by the total number of instances. The marginal frequency for a certain class by a certain "rater" is just the sum of all instances the "rater" indicated were that class. In our case, 15 (10 + 5 = 15) instances were labeled as Cats according to ground truth, and 17 (10 + 7 = 17) instances were classified as Cats by the machine learning classifier. This results in a value of 8.5 (15 * 17 / 30 = 8.5). This is then done for the second class as well (and can be repeated for each additional class if there are more than 2). 15 (10 + 5 = 15) instances were labeled as Dogs according to ground truth, and 13 (10 + 7 = 17) instances were classified as Dogs by the machine learning classifier. This results in a value of 6.5 (15 * 13 / 30 = 6.5). The final step is to add all these values together, and finally divide again by the total number of instances, resulting in an Expected Accuracy of 0.5 ((8.5 + 6.5) / 30 = 0.5). In our example, the Expected Accuracy turned out to be 50

The kappa statistic can then be calculated using both the Observed Accuracy (0.60) and the Expected Accuracy (0.50) and the formula:

Kappa = (observed accuracy - expected accuracy)/(1 - expected accuracy)

So, in our case, the kappa statistic equals: (0.60 - 0.50)/(1 - 0.50) = 0.20.

As another example, here is a less balanced confusion matrix and the corresponding calculations:

Cats Dogs Cats— 22 — 9 — Dogs— 7 — 13 — Ground truth: Cats (29), Dogs (22) Machine Learning Classifier: Cats (31), Dogs (20) Total: (51) Observed Accuracy: ((22 + 13) / 51) = 0.69 Expected Accuracy: ((29 * 31 / 51) + (22 * 20 / 51)) / 51 = 0.51 Kappa: (0.69 - 0.51) / (1 - 0.51) = 0.37

In essence, the kappa statistic is a measure of how closely the instances classified by the machine learning classifier matched the data labeled as ground truth, controlling for the accuracy of a random classifier as measured by the expected accuracy. Not only can this kappa statistic shed light into how the classifier itself performed, the kappa statistic for one model is directly comparable to the kappa statistic for any other model used for the same classification task.

Interpretation

There is not a standardized interpretation of the kappa statistic. According to Wikipedia (citing their paper), Landis and Koch considers 0-0.20 as slight, 0.21-0.40 as fair, 0.41-0.60 as moderate, 0.61-0.80 as substantial, and 0.81-1 as almost perfect. Fleiss considers kappas ¿ 0.75 as excellent, 0.40-0.75 as fair to good, and ¡ 0.40 as poor. It is important to note that both scales are somewhat arbitrary. At least two further considerations should be taken into account when interpreting the kappa statistic. First, the kappa statistic should always be compared with an accompanied confusion matrix if possible to obtain the most accurate interpretation. Consider the following confusion matrix:

Cats Dogs Cats— 60 — 125 — Dogs— 5 — 5000—

The kappa statistic is 0.47, well above the threshold for moderate according to Landis and Koch and fair-good for Fleiss. However, notice the hit rate for classifying Cats. Less than a third of all Cats were actually classified as Cats; the rest were all classified as Dogs. If we care more about classifying Cats correctly (say, we are allergic to Cats but not to Dogs, and all we care about is not succumbing to allergies as opposed

to maximizing the number of animals we take in), then a classifier with a lower kappa but better rate of classifying Cats might be more ideal.

Second, acceptable kappa statistic values vary on the context. For instance, in many inter-rater reliability studies with easily observable behaviors, kappa statistic values below 0.70 might be considered low. However, in studies using machine learning to explore unobservable phenomena like cognitive states such as day dreaming, kappa statistic values above 0.40 might be considered exceptional.

So, in answer to your question about a 0.40 kappa, it depends. If nothing else, it means that the classifier achieved a rate of classification 2/5 of the way between whatever the expected accuracy was and 100

Classifiers built and evaluated on data sets of different class distributions can be compared more reliably through the kappa statistic (as opposed to merely using accuracy) because of this scaling in relation to expected accuracy. It gives a better indicator of how the classifier performed across all instances, because a simple accuracy can be skewed if the class distribution is similarly skewed. As mentioned earlier, an accuracy of 80

That's about all I have. If anyone notices anything left out, anything incorrect, or if anything is still unclear, please let me know so I can improve the answer.

References I found helpful:

Includes a succinct description of kappa: http://standardwisdom.com/softwarejournal/2011/12/c
matrix-another-single-value-metric-kappa-statistic/

8 kernlab: Kernel-based Machine Learning Lab

Description Kernel-based machine learning methods for classification, regression, clustering, novelty detection, quantile regression and dimensionality reduction. Among other methods kernlab includes Support Vector Machines, Spectral Clustering, Kernel PCA, Gaussian Processes and a QP solver.

cran.r-project.org/web/packages/kernlab/vignettes/kernlab.pdf

9 Oil

Brodnjak-Voncina et al. (2005) describe a set of data where seven fatty acid compositions were used to classify commercial oils as either pumpkin (labeled A), sunflower (B), peanut (C), olive (D), soybean (E), rapeseed (F) and corn (G). There were 96 data points contained in their Table 1 with known results. The breakdown of the classes is given in below:

```
data(oil)
dim(fattyAcids)
[1] 96  7
table(oilType)
```

```
oilType
 A  B  C  D  E  F  G
37 26  3  7 11 10  2
```

As a note, the paper states on page 32 that there are 37 unknown samples while the table on pages 33 and 34 shows that there are 34 unknowns.

Using the data from the Examples section of `caret::createFolds`

```
library(caret)
data(oil)
part <- createDataPartition(oilType, 2)
fold <- createFolds(oilType, 2)

length(Reduce(intersect, part))
# [1] 27
```

```
length(Reduce(intersect, fold))
#[1] 0
```

Looks like `createDataPartition` split your data into smaller pieces, but allows for the same example to appear in different splits.

`createFolds` doesn't allow different examples to appear in different splits of the folds.

Basically, createDataPartition is used when you need to make one or more simple two-way splits of your data. For example, if you want to make a training and test set and keep your classes balanced, this is what you could use. It can also make multiple splits of this kind (or leave-group-out CV aka Monte Carlos CV aka repeated training test splits).

createFolds is exclusively for k-fold CV. Their usage is simular when you use the returnTrain = TRUE option in createFolds.

Partial Least Squares Discriminant Analysis

The plsda function is a wrapper for the plsr function in the pls package that does not require a formula interface and can take factor outcomes as arguments. The classes are broken down into dummy variables (one for each class). These 0/1 dummy variables are modeled by partial least squares.

From this model, there are two approaches to computing the class predictions and probabilities:

the softmax technique can be used on a per-sample basis to normalize the scores so that they are more "probability like" (i.e. they sum to one and are between zero and one). For a vector of model predictions for each class X, the softmax class probabilities are computed as. The predicted class is simply the class with the largest model prediction, or equivalently, the largest class probability. This is the default behavior for plsda. Bayes rule can be applied to the model predictions to form posterior probabili-

ties. Here, the model predictions for the training set are used along with the training set outcomes to create conditional distributions for each class. When new samples are predicted, the raw model predictions are run through these conditional distributions to produce a posterior probability for each class (along with the prior). Bayes rule can be used by specifying probModel = "Bayes". An additional parameter, prior, can be used to set prior probabilities for the classes. The advantage to using Bayes rule is that the full training set is used to directly compute the class probabilities (unlike the softmax function which only uses the current sample's scores). This creates more realistic probability estimates but the disadvantage is that a separate Bayesian model must be created for each value of ncomp, which is more time consuming.

For the sonar data set, we can fit two PLS models using each technique and predict the class probabilities for the test set.

```
plsFit <- plsda(training, trainClass, ncomp = 20)
plsFit
Partial least squares classification, fitted with the kernel algorithm.
The softmax function was used to compute class probabilities.

Call:
plsr(formula = y ~ x, ncomp = ncomp, data = tmpData)
plsBayesFit <- plsda(training, trainClass, ncomp = 20,
                     probMethod = "Bayes")
plsBayesFit
Partial least squares classification, fitted with the kernel algorithm.
Bayes rule was used to compute class probabilities.

Call:
plsr(formula = y ~ x, ncomp = ncomp, data = tmpData)
```

```
predict(plsFit, head(testing), type = "prob")
, , 20 comps


        M      R
4  0.6228 0.3772
6  0.5241 0.4759
12 0.3884 0.6116
16 0.1925 0.8075
17 0.1801 0.8199
19 0.1337 0.8663
predict(plsBayesFit, head(testing), type = "prob")
, , ncomp20


          M       R
4  0.950775 0.04922
6  0.585469 0.41453
12 0.076099 0.92390
16 0.002769 0.99723
17 0.003715 0.99628
19 0.023820 0.97618
```

Similar to plsda, caret also contains a function splsda that allows for classification using sparse PLS. A dummy matrix is created for each class and used with the spls function in the spls package. The same approach to estimating class probabilities is used for plsda and splsda.

9.1 Random Forests, an Ensemble Method

The random forest (Breiman, 2001) is an ensemble approach that can also be thought of as a form of nearest neighbor predictor.

Ensembles are a divide-and-conquer approach used to improve performance. The main principle behind ensemble methods is that a group of weak learners can come together to form a strong learner. The figure below (taken from here) provides an example. Each classifier, individually, is a weak learner, while all the classifiers taken together are a strong learner.

The data to be modeled are the blue circles. We assume that they represent some underlying function plus noise. Each individual learner is shown as a gray curve. Each gray curve (a weak learner) is a fair approximation to the underlying data. The red curve (the ensemble strong learner) can be seen to be a much better approximation to the underlying data.

I'm running a random forest model using R's caret package, and running varImp on the returned object gives me the averaged variable importance across the number of bootstrap iterations. However, I would rather assess variable importance for each iteration. Is this possible using the caret package?

Reproducible example:

```
library(caret)
mod <- train(Species ~ ., data = iris,
         method = "cforest",
         controls = cforest_unbiased(ntree = 10))
varImp(mod)
```

returns:

```
cforest variable importance
Overall
Petal.Width  100.0000
Petal.Length  86.6279
Sepal.Length   0.5814
Sepal.Width    0.0000
```

what I'm interested in is rather a list of length=number of bootstrap resamples with variable importance for each iteration. This might be possible using some combination of returnResamp = "all" and a custom summaryFunction but I'm not wise enough to know.

10 ROC Curves

This type of graph is called a Receiver Operating Characteristic curve (or ROC curve.) It is a plot of the true positive rate against the false positive rate for the different possible cutpoints of a diagnostic test.

An ROC curve demonstrates several things:

It shows the tradeoff between sensitivity and specificity (any increase in sensitivity will be accompanied by a decrease in specificity). The closer the curve follows the left-hand border and then the top border of the ROC space, the more accurate the test. The closer the curve comes to the 45-degree diagonal of the ROC space, the less accurate the test. The slope of the tangent line at a cutpoint gives the likelihood ratio (LR) for that value of the test. You can check this out on the graph above. Recall that the LR for T4 ¡ 5 is 52. This corresponds to the far left, steep portion of the curve. The LR for T4 ¿ 9 is 0.2. This corresponds to the far right, nearly horizontal portion of the curve. The area under the curve is a measure of text accuracy.

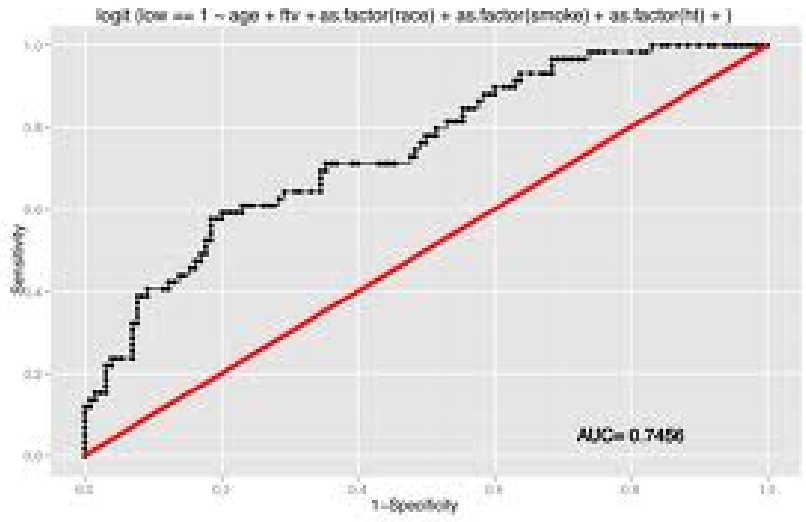

Figure 10.1:

11 RuleBased Models

if-then statements generated by a tree dene a unique route to one terminal node for any sample. A rule is a set of if-then conditions that have been collapsed into independent conditions. For the example:

```
if X1 >= 1.7 and X2 >= 202.1 then Class = 1
if X1 >= 1.7 and X2 < 202.1 then Class = 1
if X1 < 1.7 then Class = 2
```

12 Summary Outputs

The `summaryFunction` argument is used to pas in a function that takes the observed and predicted values and estimate some measure of performance. Two such functions are already included in the package: defaultSummary and twoClassSummary.

The latter will compute measures specic to twoclass problems, such as the area under the ROC curve, the sensitivity and specicity. Since the ROC curve is based on the predicted class probabilities (which are not computed automatically), another

option is required. The classProbs = TRUE option is used to include these calculations.

13 The `train` Function

This function sets up a grid of tuning parameters for a number of classification and regression routines, fits each model and calculates a resampling based performance measure.

`train` can be used to tune models by picking the complexity parameters that are associated with the optimal resampling statistics. For particular model, a grid of parameters (if any) is created and the model is trained on slightly different data for each candidate combination of tuning parameters. Across each data set, the performance of held-out samples is calculated and the mean and standard deviation is summarized for each combination. The combination with the optimal resampling statistic is chosen as the final model and the entire training set is used to fit a final model.

One of the primary tools in the package is the train function which can be used to

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