Contents

1 Preliminaries: Other Packages

1.1 R Code for Installing Packages

```
install.packages("caret")
install.packages("mlbench")
install.packages("AppliedPredictiveModeling")
install.packages("kernlab")
install.packages("e1071")
install.packages("pgmm")
```

1.2 pgmm

1.3 e1071

e1071: Misc Functions of the Department of Statistics (e1071), TU Wien

Functions for latent class analysis, short time Fourier transform, fuzzy clustering, support vector machines, shortest path computation, bagged clustering, naive Bayes classifier, ...

1.4 mlbench: Machine Learning Benchmark Problems

A collection of artificial and real-world machine learning benchmark problems, including, e.g., several data sets from the UCI repository.

1.4.1 Sonar Data Set

Put Description Here

1.5 AppliedPredictiveModeling

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

1.5.1 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.

1.5.2 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.

1.6 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

2 Predictive Modeling

- 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.

3 Key Concepts in Predictive Models

3.1 Steps in building a prediction

1. Find the right data
2. Define your error rate
3. Split data into:
    - Training Set
    - Testing Set
    - Validation Set(optional)
4. On the training set pick features
5. On the training set pick prediction function
6. On the training set cross-validate
7. If no validation - apply 1x to test set
8. If validation - apply to test set and refine
9. If validation - apply 1x to validation

3.2 Type III Errors

- Type III error is related to hypotheses suggested by the data, if tested using the data set that suggested them, are likely to be accepted even when they are not true.
- This is because circular reasoning would be involved: something seems true in the limited data set, therefore we hypothesize that it is true in general, therefore we (wrongly) test it on the same limited data set, which seems to confirm that it is true.
- Generating hypotheses based on data already observed, in the absence of testing them on new data, is referred to as post hoc theorizing.
- The correct procedure is to test any hypothesis on a data set that was not used to generate the hypothesis.

3.3 Binary Classification

Defining true/false positives

In general, Positive = identified and negative = rejected. Therefore:

- True positive = correctly identified
- False positive = incorrectly identified
- True negative = correctly rejected
- False negative = incorrectly rejected

Medical testing example:

- True positive = Sick people correctly diagnosed as sick
- False positive= Healthy people incorrectly identified as sick
- True negative = Healthy people correctly identified as healthy
- False negative = Sick people incorrectly identified as healthy.

3.4 Definitions

Accuracy Rate

The accuracy rate calculates the proportion ofobservations being allocated to the **correct** group by the predictive model. It is calculated as follows:

$$\frac{\text{Number of Correct Classifications}}{\text{Total Number of Classifications}}$$

In the case of Binary Outcomes:

$$= \frac{TP + TN}{TP + FP + TN + FN}$$

Misclassification Rate

The misclassification rate calculates the proportion ofobservations being allocated to the **incorrect** group by the predictive model. It is calculated as follows:

$$\frac{\text{Number of Incorrect Classifications}}{\text{Total Number of Classifications}}$$

In the case of Binary Outcomes:

$$= \frac{FP + FN}{TP + FP + TN + FN}$$

3.5 Olive Oil Example

Load the olive oil data using the commands:

```
install.packages("pgmm")
library(pgmm)
data(olive)
olive = olive[,-1]
```

These data contain information on 572 different Italian olive oils from multiple regions in Italy. *(Areas: (1) North Apulia, (2) Calabria, (3) South Apulia, (4) Sicily, (5) Inland Sardinia, (6) Coastal Sardinia, (7) East Liguria, (8) West Liguria, and (9) Umbria)*

```
> table(olive$Area)

  1   2   3   4   5   6   7   8   9
 25  56 206  36  65  33  50  50  51
```

Fit a classification tree where **Area** is the outcome variable. Then predict the value of area for the following data frame using the tree command with all defaults.

```
library(tree)
head(olive)
```

```
> head(olive)
  Area Palmitic Palmitoleic Stearic Oleic Linoleic Linolenic
1    1     1075          75     226  7823      672        36
2    1     1088          73     224  7709      781        31
3    1      911          54     246  8113      549        31
4    1      966          57     240  7952      619        50
5    1     1051          67     259  7771      672        50
6    1      911          49     268  7924      678        51
  Arachidic Eicosenoic
1        60         29
2        61         29
3        63         29
4        78         35
5        80         46
6        70         44
```

The following code shows how to fit a regression tree using the `tree()` command. Area is the outcome variable, using all the other variables as predictor variables.

```
olive.tree <- tree(Area ~ Palmitic +
     Palmitoleic + Stearic + Oleic + Linoleic +
     Linolenic + Arachidic + Eicosenoic,
     data=olive)
```

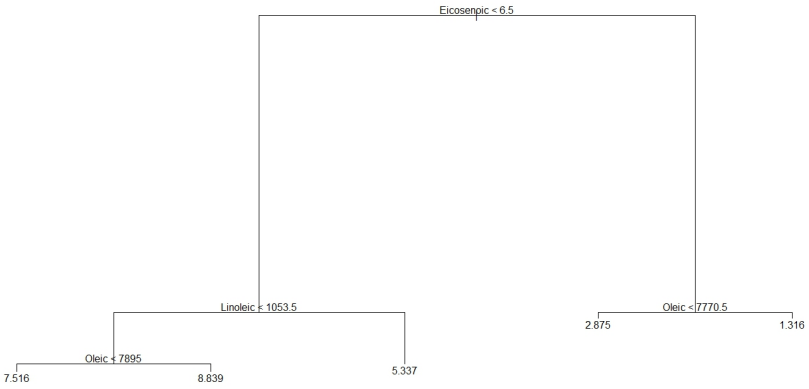

Figure 3.1:

```
plot(olive.tree)
text(olive.tree)

newdata = as.data.frame(t(colMeans(olive)))

predict(olive.tree, newdata)
```

Answer

2.875. It is strange because Region should be a qualitative variable - but tree is reporting the average value of Region as a numeric variable in the leaf predicted for newdata.

Question 5

Suppose that I fit and prune a tree to get the following diagram. What area would I predict for a new value of:

```
olive.tree <- tree(as.factor(Area) ~ Palmitic +
         Palmitoleic + Stearic + Oleic + Linoleic +
         Linolenic + Arachidic + Eicosenoic, data=olive)

plot(olive.tree); text(olive.tree)
```

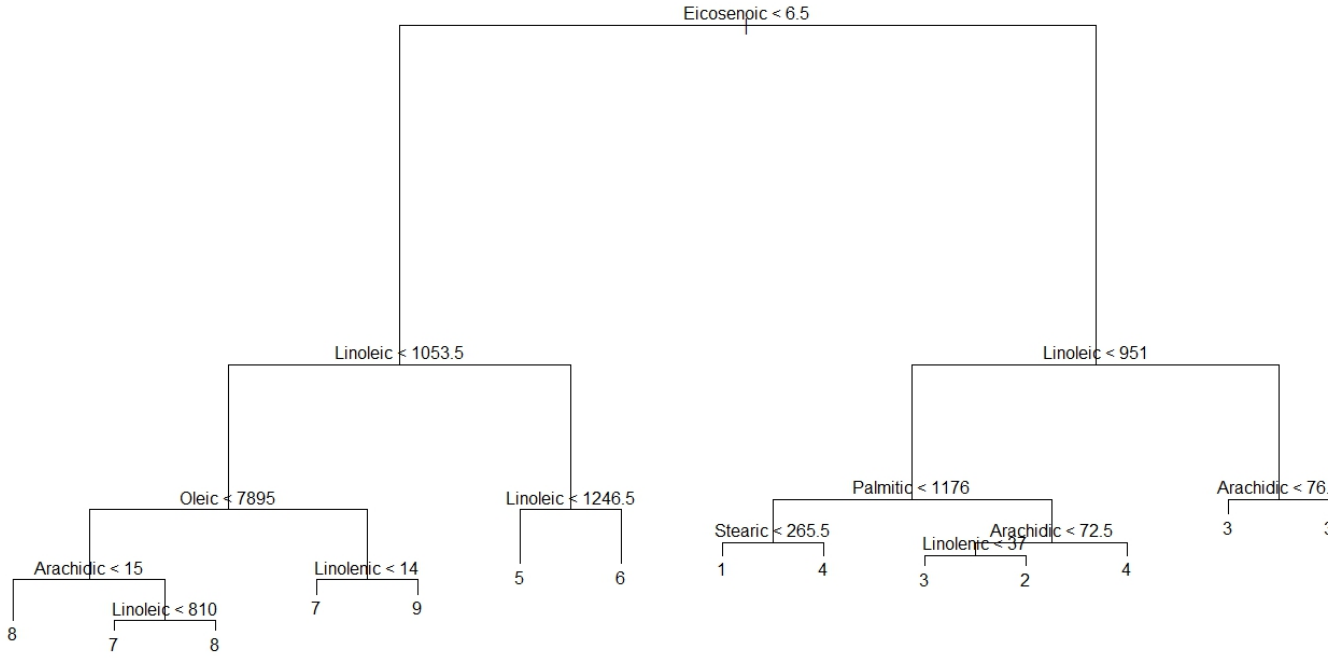

Figure 3.2:

3.5.1 Tree Pruning

The `prune.tree()` command determines a nested sequence of subtrees of the supplied tree by recursively snipping off the least important splits in the regression tree.

```
olive.pruned <- prune.tree(olive.tree,best=6)

plot(olive.pruned); text(olive.pruned)
```

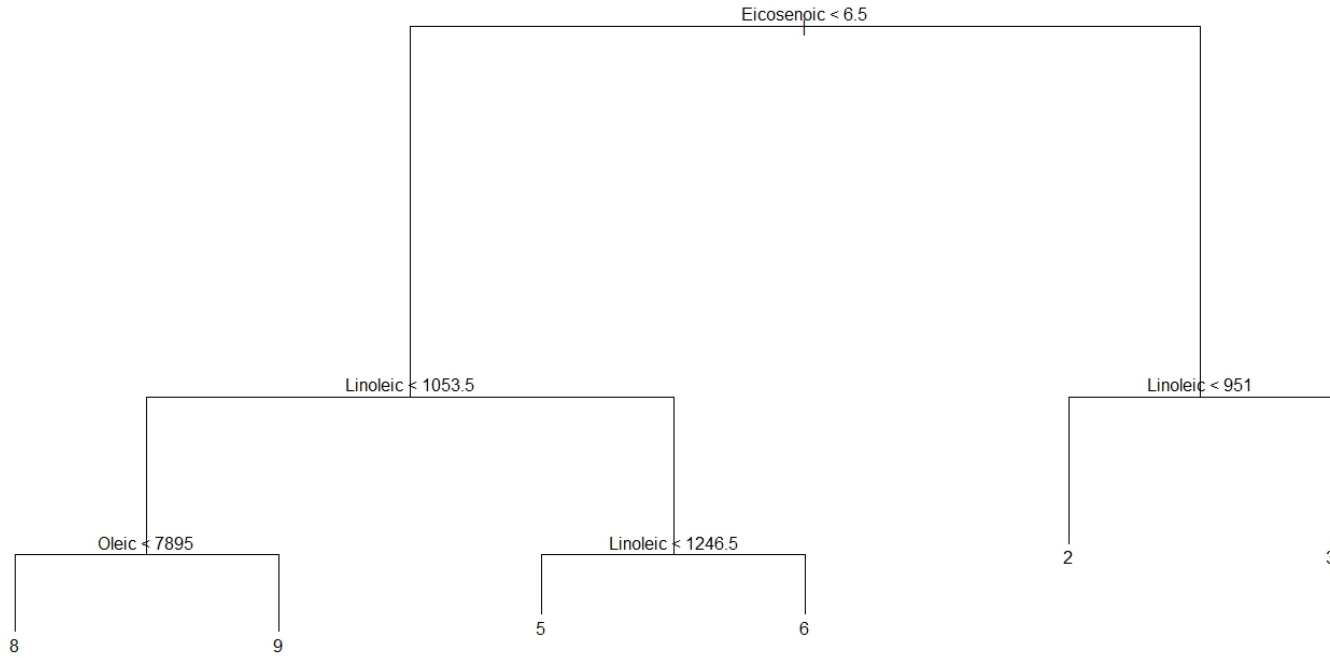

Figure 3.3:

```
newData = data.frame(Palmitic = 1200, Palmitoleic = 120,
          Stearic=200, Oleic=7000, Linoleic = 900,
          Linolenic = 32, Arachidic=60, Eicosenoic=6)

predict(olive.pruned, newData)
```

```
predict(olive.pruned, newData)
  1 2 3 4 5 6         7         8 9
1 0 0 0 0 0 0 0.4842105 0.5157895 0
```

4 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).

5 Model Metrics

5.1 Simple Coin Toss Experiment

Consider a simple (Fair) Coin Toss experiment. If you were to make a guess as to what the next result is at each successive throw, you should expect to be right 50% of the time, after a sufficient number of trials have taken place.

5.2 Kappa Statistic

Computation of the Kappa Statistic bears a resemblence to the computation of the χ^2 test for independence.

- 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% is a lot less impressive with an Expected Accuracy of 75% versus an Expected Accuracy of 50%).

- Computation of Observed Accuracy and Expected Accuracy is integral to comprehension of the kappa statistic, and is most easily illustrated through use of a confusion matrix.
- The function `confusionMatrix` can be used to compute various summaries for classification mode

5.2.1 Computation

Lets begin with a simple confusion matrix from a simple binary classification of Cats and Dogs:

	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%, as will always be the case when either "rater" classifies each class with the same frequency in a binary classification (both

Cats and Dogs contained 15 instances according to ground truth labels in our confusion matrix).

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

$$Kappa = \frac{observedaccuracy - expectedaccuracy}{1 - expectedaccuracy}$$

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.

5.2.2 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% accuracy. If expected accuracy was 80%, that means that the classifier performed 40% (because kappa is 0.4) of 20% (because this is the distance between 80% and 100%) above 80% (because this is a kappa of 0, or random chance), or 88%. So, in that case, each increase in kappa of 0.10 indicates a 2% increase in classification accuracy. If accuracy was instead 50%, a kappa of 0.4 would mean that the classifier performed with an accuracy that is 40% (kappa of 0.4) of 50% (distance between 50% and 100%) greater than 50% (because this is a kappa of 0, or random chance), or 70%. Again, in this case that means that an increase in kappa of 0.1 indicates a 5% increase in classification accuracy.

- 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% is a lot

more impressive with an expected accuracy of 50% versus an expected accuracy of 75%. Expected accuracy as detailed above is susceptible to skewed class distributions, so by controlling for the expected accuracy through the kappa statistic, we allow models of different class distributions to be more easily compared.

- 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.

5.3 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.

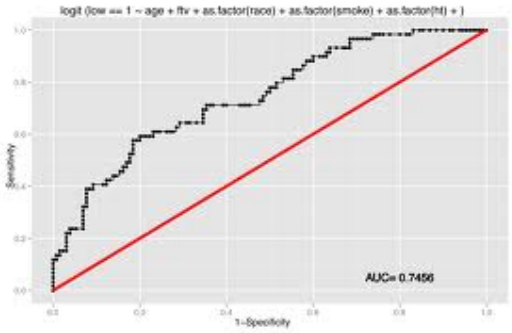

Figure 5.4:

5.4 The ROC Curve

The ***aSAH*** dataset summarizes several clinical and one laboratory variable of 113 patients with an aneurysmal subarachnoid hemorrhage.

```
Xavier Robin, Natacha Turck, Alexandre Hainard, et al. (2011) pROC: an open-source packag
```

```
> tail(aSAH)
    gos6 outcome gender age wfns s100b  ndka
136    5    Good Female  68    4  0.47 10.33
137    4    Good   Male  53    4  0.17 13.87
138    1    Poor   Male  58    5  0.44 15.89
139    5    Good Female  32    1  0.15 22.43
140    5    Good Female  39    1  0.50  6.79
141    5    Good   Male  34    1  0.48 13.45
```

-
-

```
library(pROC)
data(aSAH)
```

```
# Basic example with 2 roc objects
roc1 <- roc(aSAH$outcome, aSAH$s100b)
```

```
> roc1

Call:
roc.default(response = aSAH$outcome, predictor = aSAH$s100b)

Data: aSAH$s100b in 72 controls (aSAH$outcome Good) < 41 cases (aSAH$outcome Poor).
Area under the curve: 0.7314
```

6 Resampling Methods

The resampling methods used by **caret** are:

- bootstrapping,
- k-fold crossvalidation,
- leave-one-out cross-validation,
- leave-group-out cross-validation (i.e., repeated splits without replacement).

6.1 Avoiding Over-Fitting

A major issue that arises when applying regression or classification trees to "real" data with much random error noise concerns the decision when to stop splitting. For example, if you had a data set with 10 cases, and performed 9 splits (determined 9 if-then conditions), you could perfectly predict every single case. In general, if you only split a sufficient number of times, eventually you will be able to "predict" ("reproduce" would be the more appropriate term here)

your original data (from which you determined the splits). Of course, it is far from clear whether such complex results (with many splits) will replicate in a sample of new observations; most likely they will not.

This general issue is also discussed in the literature on tree classification and regression methods, as well as neural networks, under the topic of **overlearning** or **overfitting**. If not stopped, the tree algorithm will ultimately "extract" all information from the data, including information that is not and cannot be predicted in the population with the current set of predictors, i.e., random or noise variation.

The general approach to addressing this issue is first to stop generating new split nodes when subsequent splits only result in very little overall improvement of the prediction. For example, if you can predict 90% of all cases correctly from 10 splits, and 90.1% of all cases from 11 splits, then it obviously makes little sense to add that 11th split to the tree. There are many such criteria for automatically stopping the splitting (tree-building) process.

6.1.1 Pruning

Once the tree building algorithm has stopped, it is always useful to further evaluate the quality of the prediction of the current tree in samples of observations that did not participate in the original computations. These methods are used to "prune back" the tree, i.e., to eventually (and ideally) select a simpler tree than the one obtained when the tree building algorithm stopped, but one that is equally as accurate for predicting or classifying "new" observations.

6.1.2 Crossvalidation

One approach is to apply the tree computed from one set of observations (learning sample) to another completely independent set of observations (testing sample). If most or all of the splits determined by the analysis of the learning sample are essentially based on "random noise," then the prediction for the testing sample will be very poor. Hence one can infer that the selected tree is not very good (useful), and not of the "right size."

6.1.3 V-fold crossvalidation

Continuing further along this line of reasoning (described in the context of crossvalidation above), why not repeat the analysis many times over with different randomly drawn samples from the data, for every tree size starting at the root of the tree, and applying it to the prediction of observations from randomly selected testing samples. Then use (interpret, or accept as your final result) the tree that shows the best average accuracy for cross-validated predicted classifications or predicted values.

In most cases, this tree will not be the one with the most terminal nodes, i.e., the most complex tree. This method for pruning a tree, and for selecting a smaller tree from a sequence of trees, can be very powerful, and is particularly useful for smaller data sets. It is an essential step for generating useful (for prediction) tree models, and because it can be computationally difficult to do, this method is often not found in tree classification or regression software.

7 Cross Validation

Bias Variance Trade-off *http://scott.fortmann-roe.com/docs/BiasVariance.html*

- In a prediction problem, a model is usually given a dataset of known data on which training is run (*training dataset*), and a dataset of unknown data (or *first seen data/ testing dataset*) against which testing the model is performed.
- Cross-validation is mainly used in settings where the goal is prediction, and one wants to estimate how accurately a predictive model will perform in practice.
- The goal of cross validation is to define a dataset to "test" the model in the training phase (i.e., the validation dataset), in order to limit problems like overfitting, give an insight on how the model will generalize to an independent data set (i.e., an unknown dataset, for instance from a real problem), etc.
- Cross-validation is important in guarding against testing hypotheses suggested by the data (called "Type III errors"), especially where further samples are hazardous, costly or impossible to collect

7.0.4 K-fold cross validation

- In k-fold cross-validation, the original data set is randomly partitioned into k equal size subsamples.
- Of the k subsamples, a single subsample is retained as the validation data for testing the model, and the remaining k - 1 subsamples are used as training data.
- The cross-validation process is then repeated k times (the folds), with each of the k subsamples used exactly once as the validation data.
- The k results from the folds can then be averaged (or otherwise combined) to produce a single estimation.
- The advantage of this method over repeated random sub-sampling is that all observations are used for both training and validation, and each observation is used for validation exactly once.

7.0.5 Choosing K - Bias and Variance

In general, when using k-fold cross validation, it seems to be the case that:

- A larger k will produce an estimate with smaller bias but potentially higher variance (on top of being computationally expensive)
- A smaller k will lead to a smaller variance but may lead to a a biased estimate.

7.0.6 Leave-One-Out Cross-Validation

- As the name suggests, leave-one-out cross-validation (LOOCV) involves using a single observation from the original sample as the validation data, and the remaining observations as the training data.
- This is repeated such that each observation in the sample is used once as the validation data.

- This is the same as a K-fold cross-validation with K being equal to the number of observations in the original sampling, i.e. **K=n**.

8 Decision tree learning

Decision tree learning uses a decision tree as a predictive model which maps observations about an item to conclusions about the item's target value. It is one of the predictive modelling approaches used in statistics, data mining and machine learning. More descriptive names for such tree models are classification trees or regression trees. In these tree structures, leaves represent class labels and branches represent conjunctions of features that lead to those class labels.

Decision trees used in data mining are of two main types:

Classification tree analysis is when the predicted outcome is the class to which the data belongs. Regression tree analysis is when the predicted outcome can be considered a real number (e.g. the price of a house, or a patients length of stay in a hospital).

The term Classification And Regression Tree (CART) analysis is an umbrella term used to refer to both of the above procedures, first introduced by Breiman et al.[3] Trees used for regression and trees used for classification have some similarities - but also some differences, such as the procedure used to determine where to split.[3]

Some techniques, often called **ensemble methods**, construct more than one decision tree:

Bagging decision trees, an early ensemble method, builds multiple decision trees by repeatedly resampling training data with replacement, and voting the trees for a consensus prediction.[4] A Random Forest classifier uses a number of decision trees, in order to improve the classification rate. Boosted Trees can be used for regression-type and classification-type problems.[5][6] Rotation forest - in which every decision tree is trained by first applying principal component analysis (PCA) on a random subset of the input features.[7]

8.1 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
```

8.2 Classification and regression trees (CART)

Classification and regression trees (CART) is a non-parametric decision tree learning technique that produces either classification or regression trees, depending on whether the dependent variable is categorical or numeric, respectively.

Decision trees are formed by a collection of rules based on variables in the modeling data set:

Rules based on variables' values are selected to get the best split to differentiate observations based on the dependent variable Once a rule is selected and splits a node into two, the same process is applied to each "child" node (i.e. it is a recursive procedure) Splitting stops when CART detects no further gain can be made, or some pre-set stopping rules are met. (Alternatively, the data are split as much as possible and then the tree is later pruned.) Each branch of the tree ends in a terminal node. Each observation falls into one and exactly one terminal node, and each terminal node is uniquely defined by a set of rules.

A very popular method for predictive analytics is Leo Breiman's Random forests or derived versions of this technique like Random multinomial logit.

The purpose of the analysis is to learn how we can discriminate between the three types of flowers, based on the four measures of width and length of petals and sepals. Discriminant function analysis will estimate several linear combinations of predictor variables for computing classification scores (or probabilities) that allow the user to determine the predicted classification for each observation. A classification tree will determine a set of logical if-then conditions (instead of linear equations) for predicting or classifying cases instead:

The interpretation of this tree is straightforward: If the petal width is less than or equal to 0.8, the respective flower would be classified as Setosa; if the petal width is greater than 0.8 and less than or equal to 1.75, then the respective flower would be classified as Virginic; else, it belongs to class Versicol.

```
boxplot(iris$Petal.Width~iris$Species,horizontal=T,col=c("lightblue","pink","yellow"),for
abline(v=0.80,col="red",lty=2)
```

```
boxplot(iris$Petal.Width~iris$Species,horizontal=T,col=c("lightblue","pink","yellow"),for
```

```
abline(v=0.80,col="red",lty=2)
abline(v=1.75,col="red",lty=2)

Class=numeric(150)
for (i in 1:150)
  {
  if (iris$Petal.Width[i]<0.80) {
     Class[i]=1
   } else if (iris$Petal.Width[i]>1.75) {
     Class[i]=3
   }else Class[i]=2
  }
```

8.3 Regression Trees

The general approach to derive predictions from few simple if-then conditions can be applied to regression problems as well. This example is based on the data file Poverty, which contains 1960 and 1970 Census figures for a random selection of 30 counties.

The research question (for that example) was to determine the correlates of poverty, that is, the variables that best predict the percent of families below the poverty line in a county. A reanalysis of those data, using the regression tree analysis [and v-fold cross-validation, yields the following results:

Again, the interpretation of these results is rather straightforward: Counties where the percent of households with a phone is greater than 72% have generally a lower poverty rate. The greatest poverty rate is evident in those counties that show less than (or equal to) 72% of households with a phone, and where the population change (from the 1960 census to the 170 census) is less than -8.3 (minus 8.3).

These results are straightforward, easily presented, and intuitively clear as well: There are some affluent counties (where most households have a telephone), and those generally have little

poverty. Then there are counties that are generally less affluent, and among those the ones that shrunk most showed the greatest poverty rate.

A quick review of the scatterplot of observed vs. predicted values shows how the discrimination between the latter two groups is particularly well "explained" by the tree model.

9 Random Forests

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.

Random forests are an ensemble learning method for classification (and regression) that operate by constructing a multitude of decision trees at training time and outputting the class that is the mode of the classes output by individual trees.

The algorithm for inducing a random forest was developed by Leo Breiman[1] and Adele Cutler,[2] and "Random Forests" is their trademark. The term came from random decision forests that was first proposed by Tin Kam Ho of Bell Labs in 1995.

The method combines Breiman's "bagging" idea and the random selection of features, introduced independently by Ho[3][4] and Amit and Geman[5] in order to construct a collection of decision trees with controlled variance.

10 RandomForest with R

```
library(randomForest)

# download Titanic Survivors data
data <- read.table("http://math.ucdenver.edu/RTutorial/titanic.txt", h=T, sep="\t")
# make survived into a yes/no
data$Survived <- as.factor(ifelse(data$Survived==1, "yes", "no"))

# split into a training and test set
idx <- runif(nrow(data)) <= .75
data.train <- data[idx,]
data.test <- data[-idx,]
```

Train a random forest

```
rf <- randomForest(Survived ~ PClass + Age + Sex,
             data=data.train, importance=TRUE, na.action=na.omit)
```

How important is each variable in the model?

```
imp <- importance(rf)
o <- order(imp[,3], decreasing=T)
imp[o,]
#              no      yes MeanDecreaseAccuracy MeanDecreaseGini
#Sex    51.49855 53.30255             55.13458         63.46861
#PClass 25.48715 24.12522             28.43298         22.31789
#Age    20.08571 14.07954             24.64607         19.57423
```

Display the confusion matrix

```
# confusion matrix [[True Neg, False Pos], [False Neg, True Pos]]
table(data.test$Survived, predict(rf, data.test),
  dnn=list("actual", "predicted"))
#      predicted
#actual  no yes
#   no  427  16
#   yes 117 195
```

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.

11 The `train` Function

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

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.

11.1 Syntax

The `train` function has the following arguments:

`x:` a matrix or data frame of predictors. Currently, the function only accepts numeric values (i.e., no factors or character variables).

In some cases, the ***`model.matrix`*** *function may be needed to generate a data frame or matrix of purely numeric data.*

`y:` a numeric or factor vector of outcomes. The function determines the type of problem (classification or regression) from the type of the response given in this argument.

`method:` a character string specifying the type of model to be used.

`metric:` a character string with values of "`Accuracy`", "`Kappa`", "`RMSE`" or "`Rsquared`".

The metric value determines the objective function used to select the final model. For example, selecting "Kappa" makes the function select the tuning parameters with the largest value of the mean Kappa statistic computed from the held-out samples.

12 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

13 `trainControl`

```
library(caret)
ctrl <- trainControl(method = "cv", savePred=T, classProb=T)
mod <- train(Species~., data=iris, method = "svmLinear", trControl = ctrl)
head(mod$pred)
```

The "C" is one of tuning parameters for your SVM. Check out the help for the ksvm function in the kernlab package for more details.

Trivial regression example

```
library(caret)
ctrl <- trainControl(method = "cv", savePred=T)
mod <- train(Sepal.Length~., data=iris, method = "svmLinear", trControl = ctrl)
head(mod$pred)
```

14 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

14.1 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.

14.2 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.

14.3 The bag function

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.

```
library(mlbench)
data(BostonHousing)

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

library(rpart)
rpartFit <- train(medv ~ .,
```

```
                  data = BostonHousing,
                  "rpart",
                  tuneLength = 9)
```

14.4 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)
```

15 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.

15.1 createDataPartition

A series of test/training partitions are created using createDataPartition while createResample creates one or more bootstrap samples. createFolds splits the data into k groups while createTimeSlices creates cross-validation sample information to be used with time series data.

```
data(oil)
createDataPartition(oilType, 2)
```

```
$Resample1
 [1]  4  8  9 10 11 12 13 15 18 36 37 41 64 65 70 71 72 73 75
[20] 19 21 22 33 34 35 76 78 79 80 81 86 87 29 62 42 52 55 56
[39] 25 26 44 48 49 51 28 59 91 93 94 92

$Resample2
 [1]  4  6  7  9 10 12 13 14 16 18 37 38 40 64 68 69 72 73 75
[20] 19 20 21 23 24 32 33 35 81 83 84 87 88 29 62 42 53 55 56
[39] 25 26 47 49 50 96 58 59 60 93 95 27

> createResample(oilType, 2)
$Resample1
 [1]  2  3  5  5  6  6  6  6  7  8  9  9  9  9 10 10 10 11 12
[20] 13 14 16 16 19 20 21 21 22 22 23 27 27 27 28 28 29 30 30
[39] 31 34 34 35 35 35 35 36 37 38 39 41 41 41 43 45 46 48 51
[58] 53 54 56 57 58 58 59 65 65 65 66 69 71 73 73 73 75 79 81
[77] 81 82 83 83 84 84 84 84 85 85 85 86 88 88 89 90 90 94 96
[96] 96
```

```
$Resample2
 [1]  4  4  6  6  8 10 11 11 11 12 14 16 16 17 17 17 19 20 20
[20] 20 21 22 22 22 24 26 27 28 30 31 33 34 34 35 37 38 41 41
[39] 43 44 45 45 47 47 54 55 56 57 57 59 59 60 60 61 63 63 65
[58] 65 65 65 66 66 67 68 68 69 69 70 74 77 77 77 77 78 78 79
[77] 80 81 82 82 82 83 83 84 85 85 86 87 87 88 89 90 91 93 94
[96] 94
```

16 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:

    1. 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.

    2. Bayes rule can be applied to the model predictions to form posterior probabilities. 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.

17 Scheduling Data

These data consist of information on 4331 jobs in a high performance computing environment. Seven attributes were recorded for each job along with a discrete class describing the execution time.

The predictors are:

Protocol (the type of computation),

Compounds (the number of data points for each jobs),

InputFields (the number of characteristic being estimated),

Iterations (maximum number of iterations for the computations),

NumPending (the number of other jobs pending at the time of launch),

Hour (decimal hour of day for launch time),

Day (of launch time).

The classes are: VF (very fast), F (fast), M (moderate) and L (long).

```
library(AppliedPredictiveModeling)
data(schedulingData)

library(caret)
set.seed(733)
inTrain <- createDataPartition(schedulingData$Class, p = .75,
   list = FALSE)

training <- schedulingData[ inTrain,]
testing <- schedulingData[-inTrain,]
```

```
> dim(schedulingData)
[1] 4331    8
>
> dim(training)
[1] 3251    8
>
> dim(testing)
[1] 1080    8
>
```

```
library(C50)
oneTree <- C5.0(Class ~ ., data = training)
```

```
>  oneTree

Call:
C5.0.formula(formula = Class ~ ., data = training)

Classification Tree
Number of samples: 3251
Number of predictors: 7

Tree size: 199

Non-standard options: attempt to group attributes
```

```
oneTreePred <- predict(oneTree, testing)
oneTreeProbs <- predict(oneTree, testing, type ="prob")
postResample(oneTreePred, testing$Class)
```

```
> table(testing$Class,oneTreePred)
    oneTreePred
      VF   F   M   L
  VF 512  38   2   0
  F   50 256  26   4
  M    6  46  67   9
  L    0   7  10  47
```

18 Sonar Data Set

The Sonar data consist of 208 data points collected on 60 predictors. The goal is to predict the two classes The Sonar data consist of 208 data points collected on 60 predictors. The goal is to predict the two classes (M for metal cylinder or R for rock).

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

M Metal cylinder

R Rock

```
library(caret)
library(mlbench)
```

```
data(Sonar)
set.seed(107)
```

```
> head(Sonar)
      V1     V2     V3     V4     V5     V6     V7     V8
1 0.0200 0.0371 0.0428 0.0207 0.0954 0.0986 0.1539 0.1601
2 0.0453 0.0523 0.0843 0.0689 0.1183 0.2583 0.2156 0.3481
3 0.0262 0.0582 0.1099 0.1083 0.0974 0.2280 0.2431 0.3771
4 0.0100 0.0171 0.0623 0.0205 0.0205 0.0368 0.1098 0.1276
5 0.0762 0.0666 0.0481 0.0394 0.0590 0.0649 0.1209 0.2467
6 0.0286 0.0453 0.0277 0.0174 0.0384 0.0990 0.1201 0.1833
```

19 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.

The C5.0 classification model was used in this 4-class problem data with Ntrain=165, P=11, using caret R-package by running the code below. The winnowing option was tuned over in the model, which is a kind of feature selection approach. This excerpt I quote regarding winnowing from the companion book of caret, a must-have book in my opinion to realize hidden gems coded in the package: Kuhn M, Johnson K. Applied predictive modeling. 1st edition. New York: Springer. 2013.

C5.0 also has an option to winnow or remove predictors: an initial algorithm uncovers which

predictors have a relationship with the outcome, and the nal model is created from only the important predictors. To do this, the training set is randomly split in half and a tree is created for the purpose of evaluating the utility of the predictors (call this the winnowing tree). Two procedures characterize the importance of each predictor to the model: 1. Predictors are considered unimportant if they are not in any split in the winnowing tree. 2. The half of the training set samples not included to create the winnowing tree are used to estimate the error rate of the tree. The error rate is also estimated without each predictor and compared to the error rate when all the predictors are used. If the error rate improves without the predictor, it is deemed to be irrelevant and is provisionally removed.

```
c50Grid <- expand.grid(.trials = c(1:9, (1:10)*10),
                       .model = c("tree", "rules"),
                       .winnow = c(TRUE, FALSE))
c50Grid
set.seed(1) # important to have reproducible results
c5Fitvac <- train(Class ~ .,
                  data = training,
                  method = "C5.0",
                  tuneGrid = c50Grid,
                  trControl = ctrl,
                  metric = "Accuracy", # not needed it is so by default
                  importance=TRUE, # not needed
                  preProc = c("center", "scale"))
> c5Fitvac$finalModel$tuneValue
   trials model winnow
16     70  tree  FALSE
CV tuning output:
enter image description here
```

```
Excerpt from the C5.0 tree output:


> c5Fitvac$finalModel$tree
[1] "id=\"See5/C5.0 2.07 GPL Edition 2014-01-22\"\nentries=\"70\"\ntype=\"2\" class=\"Q\'
Now importance of predictors:


> predictors(c5Fitvac )
 [1] "IL23R"   "IL12RB2" "IL8"     "IL23A"   "IL6ST"   "IL12A"   "IL12RB1"
 [8] "IL27RA"  "IL12B"   "IL17A"   "EBI3"
```

Why is it in the plot, the accuracy levels of No-winnowing about two times that of the winnowing? Can you please help interpreting this output when it says winnow = FALSE? How to visualize the tree output, instead of the computed junk text that appeared in my case? is there any way to behold a tree instead of crowded symbols?

20 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.

21 Pruning the tree

Pruning is the process of removing leaves and branches to improve the performance of the decision tree when it moves from the training data (where the classification is known) to real-world applications (where the classification is unknown – it is what you are trying to predict).

The tree-building algorithm makes the best split at the root node where there are the largest number of records and, hence, a lot of information. Each subsequent split has a smaller and less representative population with which to work. Towards the end, idiosyncrasies of training records at a particular node display patterns that are peculiar only to those records. These patterns can become meaningless and sometimes harmful for prediction if you try to extend rules based on them to larger populations.

For example, say the classification tree is trying to predict height and it comes to a node containing one tall person named X and several other shorter people. It can decrease diversity at that node by a new rule saying "people named X are tall" and thus classify the training data. In a wider universe this rule can become less than useless. (Note that, in practice, we do not include irrelevant fields like "name", this is just an illustration.)

Pruning methods solve this problem – they let the tree grow to maximum size, then remove smaller branches that fail to generalize.

Since the tree is grown from the training data set, when it has reached full structure it usually suffers from over-fitting (i.e. it is "explaining" random elements of the training data that are not likely to be features of the larger population of data). This results in poor performance on real life data. Therefore, it has to be pruned using the validation data set .

22 Variable importance

Random forests can be used to rank the importance of variables in a regression or classification problem in a natural way. The following technique was described in Breiman's original paper[1] and is implemented in the R package randomForest.