Contents

1 Preliminaries: Load Packages **4**

1.1 R Code for Installing Packages 4

2 Predictive Modeling **5**

3 Key Concepts in Predictive Models **6**

3.1 Steps in building a prediction 6

3.2 Type III Errors 7

3.3 Binary Classification 8

3.4 Definitions 8

3.5 Olive Oil Example 10

3.5.1 Tree Pruning 14

4 CaRT **15**

4.1 Overview 15

4.2 Classification and Regression Problems 15

4.3 Regression-type problems 16

4.4 Classification-type problem 17

4.5 Classification and Regression Trees (CaRT) 18

4.5.1 Classification Trees 18

4.6 Iris Tree Example 20

4.6.1 Iris Example - Using a Training Set 21

4.7 Tree Pruning 24

4.8 Random Forests 25

4.9 Regression Trees 27

4.10 Advantages of Classification and Regression Trees (CaRT) Methods 31

4.11 Avoiding Over-Fitting: Pruning, Crossvalidation, and V-fold Crossvalidation . . 32

4.12 Computational Details 34

4.13 Specifying the Criteria for Predictive Accuracy 36

5 Model Metrics **36**
5.1 Simple Coin Toss Experiment . . . 36
5.2 Kappa Statistic . . . 37
5.2.1 Computation . . . 37
5.2.2 Interpretation . . . 40
5.3 ROC Curves . . . 43
5.4 The ROC Curve . . . 44

6 Resampling Methods **45**
6.1 Avoiding Over-Fitting . . . 45
6.1.1 Pruning . . . 46
6.1.2 Crossvalidation . . . 46
6.1.3 V-fold crossvalidation . . . 47

7 Cross Validation **48**
7.0.4 K-fold cross validation . . . 49
7.0.5 Choosing K - Bias and Variance . . . 49
7.0.6 Leave-One-Out Cross-Validation . . . 49

8 Pruning the tree **51**

9 RandomForest with R **52**

10 caret-preprocessing **55**
10.1 Creating Dummy Variables . . . 55
10.2 Zero- and Near Zero-Variance Predictors . . . 56
10.3 Identifying Correlated Predictors . . . 59
10.4 Linear Dependencies . . . 60
10.5 Centering and Scaling . . . 62
10.6 Imputation . . . 63
10.7 Transforming Predictors . . . 63

11 The `train` Function **64**

11.1 Syntax 65

12 Partitioning Data **66**

13 `trainControl` **66**

14 Bagging and Boosting **67**

14.1 Overview on Bagging 67

14.2 Overview on Boosting 68

14.3 The `bag` function 69

14.4 Classification Example 70

15 Sonar Data Set **71**

15.1 Partial least squares 72

16 Scheduling Data **73**

17 Summary Outputs **76**

18 Variable importance **76**

18.1 Model Specific Metrics 79

1 Preliminaries: Load Packages

1.1 `R` Code for Installing Packages

```
install.packages("tree")
install.packages("caret")
install.packages("mlbench")
install.packages("AppliedPredictiveModeling")
install.packages("kernlab")

install.packages("e1071")

install.packages("pgmm")
```

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)
```

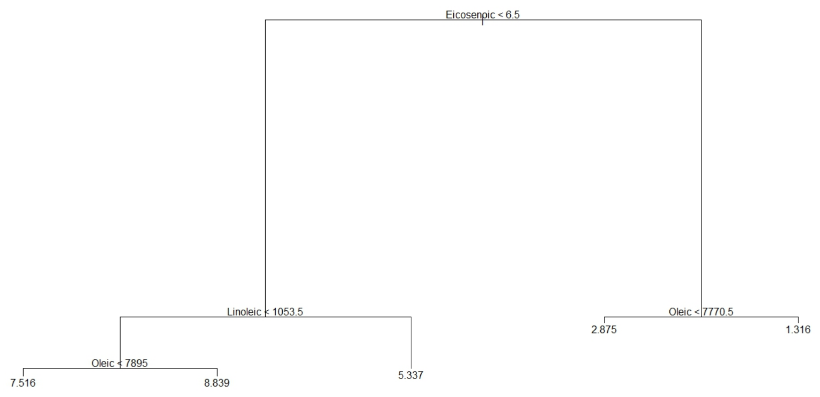

Figure 3.1:

```
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)
```

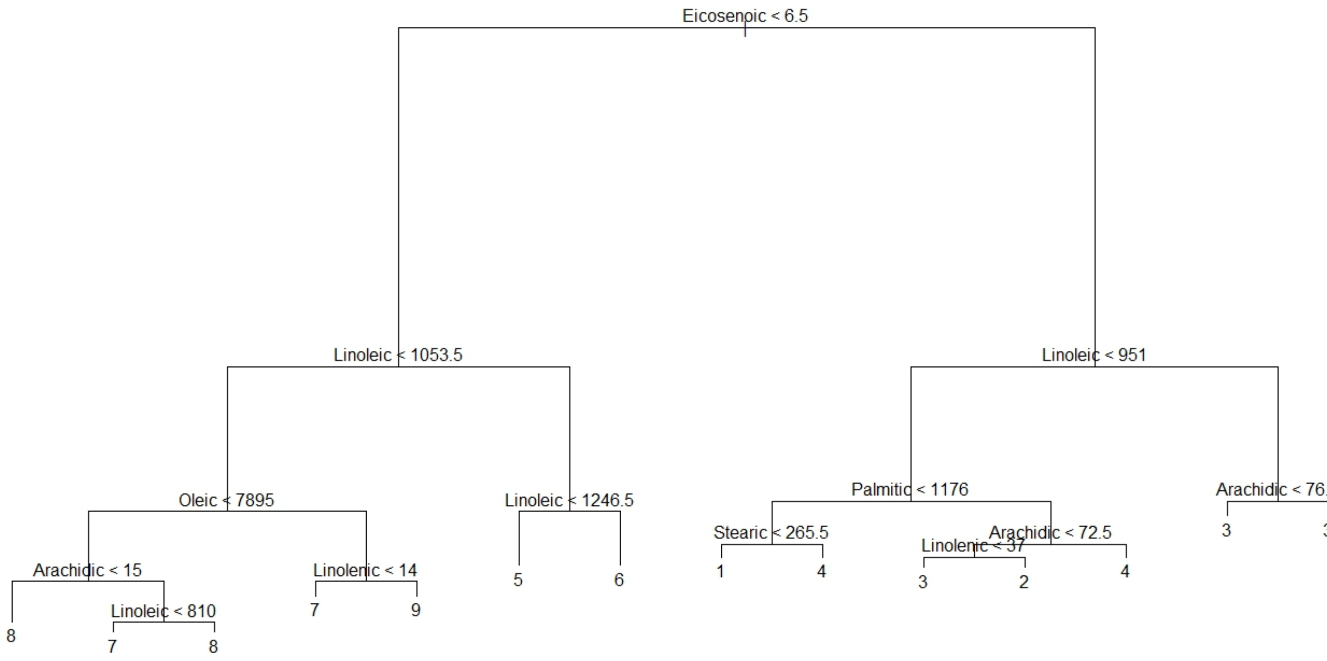

Figure 3.2:

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)
```

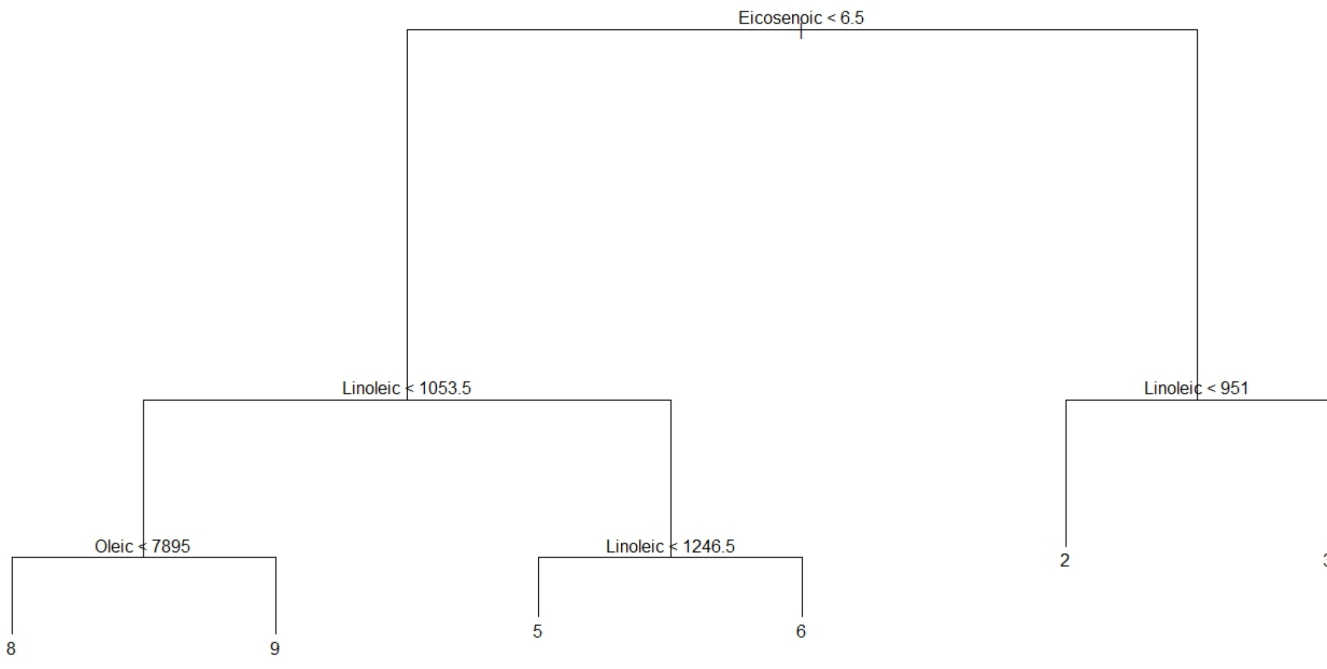

Figure 3.3:

```
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 CaRT

4.1 Overview

CaRT, a recursive partitioning method, builds classification and regression trees for predicting continuous dependent variables (regression) and categorical predictor variables (classification). The classic C&RT algorithm was popularized by Breiman et al.

4.2 Classification and Regression Problems

- There are numerous algorithms for predicting continuous variables or categorical variables from a set of continuous predictors and/or categorical factor effects. For example, in GLM (General Linear Models) and GRM (General Regression Models), we can specify a linear combination (design) of continuous predictors and categorical factor effects (e.g., with two-way and three-way interaction effects) to predict a continuous dependent variable.

- In GDA (General Discriminant Function Analysis), we can specify such designs for predicting categorical variables, i.e., to solve classification problems.

4.3 Regression-type problems

- Regression-type problems are generally those where we attempt to predict the values of a continuous variable from one or more continuous and/or categorical predictor variables. For example, we may want to predict the selling prices of single family homes (a continuous dependent variable) from various other continuous predictors (e.g., square footage) as well as categorical predictors (e.g., style of home, such as ranch, two-story, etc.; zip code or telephone area code where the property is located, etc.; note that this latter variable would be categorical in nature, even though it would contain numeric values or codes).

- If we used simple multiple regression, or some general linear model (GLM) to predict the selling prices of single family homes, we would determine a linear equation for these variables that can be used to compute predicted selling prices.

- There are many different analytic procedures for fitting linear models (GLM, GRM, Regression), various types of nonlinear models (e.g., Generalized Linear/Nonlinear Models (GLZ), Generalized Additive Models (GAM), etc.), or completely custom-defined nonlinear models (see Nonlinear Estimation), where we can type in an arbitrary equation containing parameters to be estimated. CHAID also analyzes regression-type problems, and produces results that are similar (in nature) to those computed by C&RT. Note that various neural network architectures are also applicable to solve regression-type problems.

4.4 Classification-type problem

- Classification-type problems are generally those where we attempt to predict values of a **categorical dependent variable** (class, group membership, etc.) from one or more continuous and/or categorical predictor variables.

- **Examples** - For example, we may be interested in predicting who will or will not graduate from college, or who will or will not renew a subscription.

- **Binary Outcomes** - These would be examples of simple binary classification problems, where the categorical dependent variable can only assume two distinct and mutually exclusive values.

- **Multichotomous Outcomes** - In other cases, we might be interested in predicting which one of multiple different alternative consumer products (e.g., makes of cars) a person decides to purchase, or which type of failure occurs with different types of engines. In those cases there are multiple categories or classes for the categorical dependent variable.

4.5 Classification and Regression Trees (CaRT)

In most general terms, the purpose of the analyses via tree-building algorithms is to determine a set of ***if-then*** logical (split) conditions that permit accurate prediction or classification of cases.

4.5.1 Classification Trees

- For example, consider the widely referenced Iris data classification problem introduced by Fisher.
- The `iris` data file containts the lengths and widths of sepals and petals of three types of irises: Setosa, Versicolor, and Virginica).
- 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 ***Virginica***;
  - else, it belongs to class ***Versicolor***.

```
iris.pw.tree <- tree(Species ~ Petal.Width,data=iris)
plot(iris.pw.tree)
text(iris.pw.tree)
```

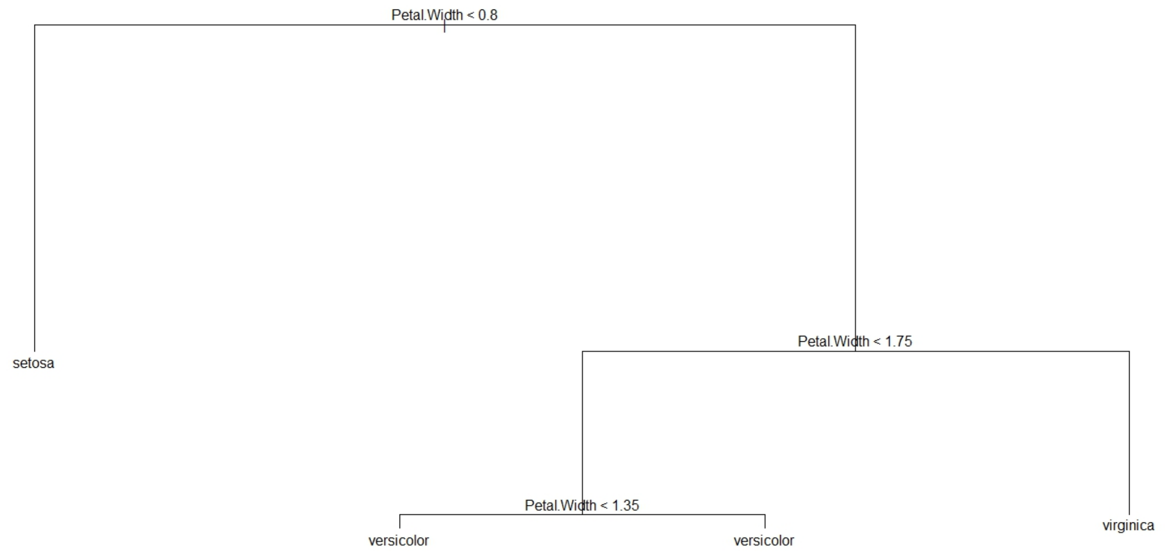

Figure 4.4:

4.6 Iris Tree Example

- Using all four variables and all of the 150 cases

```
iris.tree <- tree(Species ~ Sepal.Length+Sepal.Width+
Petal.Length+Petal.Width,data=iris)
plot(iris.tree)
text(iris.tree)
```

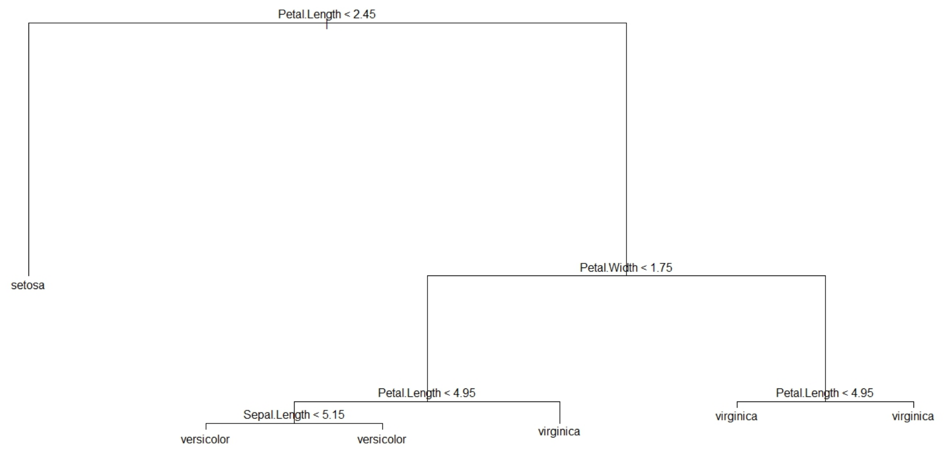

Figure 4.5:

4.6.1 Iris Example - Using a Training Set

- set up a training data set of 100 randomly selected cases
- Compare results with full data set tree.

```
set.seed(1234)
Train1Index = sample(1:150,100)
iristrain1=iris[Train1Index,]

iristrain1.tree <- tree(Species ~ Sepal.Length+Sepal.Width+
Petal.Length+Petal.Width,data=iristrain1)
plot(iristrain1.tree)
text(iristrain1.tree)
```

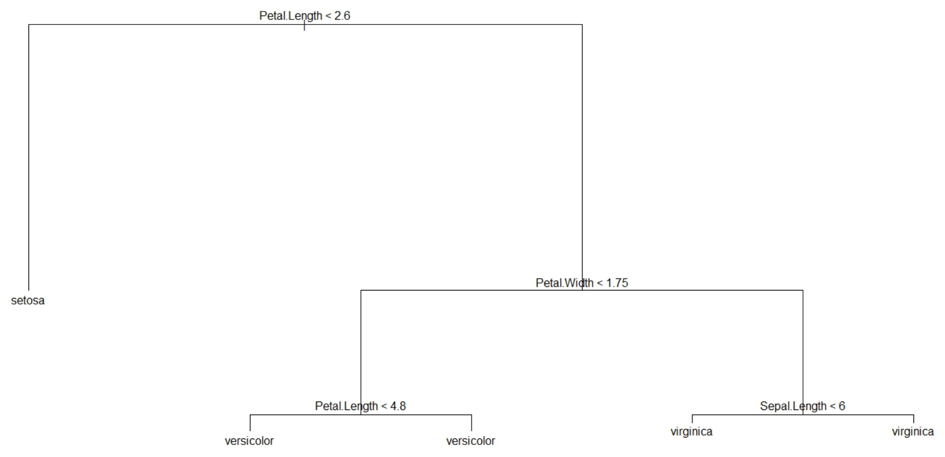

Figure 4.6:

- Another test set (picked using a different seed)

```
set.seed(3456)


Train2Index = sample(1:150,100)
iristrain2=iris[Train2Index,]


iristrain2.tree <- tree(Species ~ Sepal.Length+Sepal.Width+
  Petal.Length+Petal.Width,data=iristtrain2)
plot(iristrain2.tree)
text(iristrain2.tree)
```

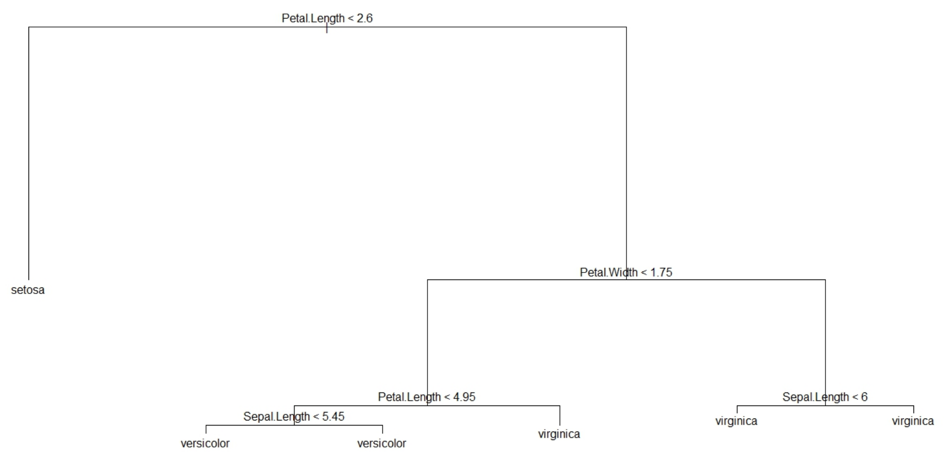

Figure 4.7:

```
> table(iristrain1$Species)

    setosa versicolor  virginica
        38         29         33
> table(iristrain2$Species)

    setosa versicolor  virginica
        34         32         34
```

- Set up a test data set to compare trees
- In this case, probably little difference

```
newdata=iris[131:150,]
predict(iristrain1.tree,newdata)
predict(iristrain2.tree,newdata)
```

4.7 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 classification tree.

```
iris.pruned <- prune.tree(iris.tree,best=3)
plot(iris.pruned); text(iris.pruned)
plot(iris.pruned); text(iris.pruned)
```

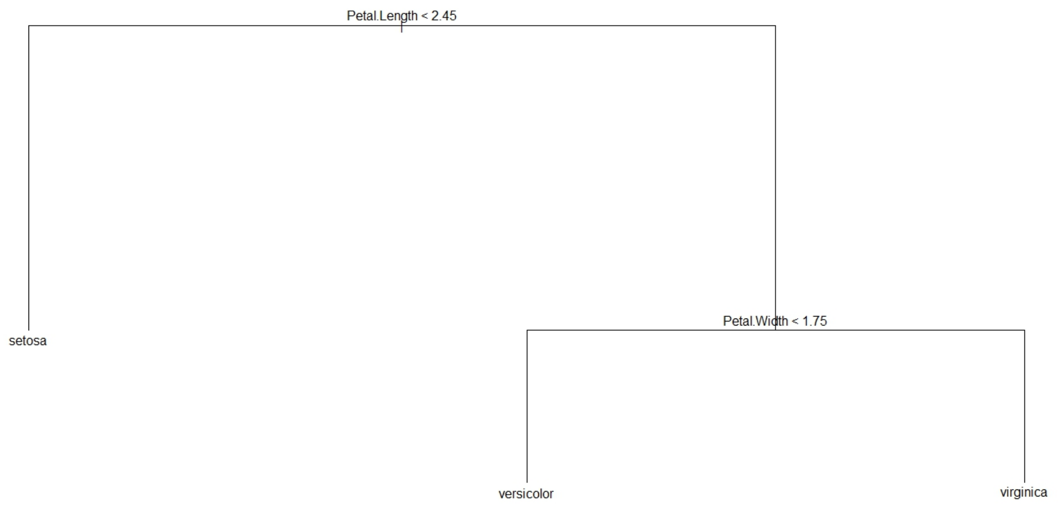

Figure 4.8:

4.8 Random Forests

Random forests improve predictive accuracy by generating a large number of bootstrapped trees (based on random samples of variables), classifying a case using each tree in this new "forest", and deciding a final predicted outcome by combining the results across all of the trees (an average in regression, a majority vote in classification).

```
library(randomForest)
iris.rf <- randomForest(Species ~ Sepal.Length+Sepal.Width+
Petal.Length+Petal.Width,data=iris)
print(iris.rf) # view results
importance(iris.rf) # importance of each predictor
```

```
> iris.rf <- randomForest(Species ~ Sepal.Length+Sepal.Width+
+ Petal.Length+Petal.Width,data=iris)
> print(iris.rf) # view results

Call:
 randomForest(formula = Species ~ Sepal.Length + Sepal.Width +      Petal.Length + Petal.
               Type of random forest: classification
                     Number of trees: 500
No. of variables tried at each split: 2

        OOB estimate of  error rate: 4%
Confusion matrix:
           setosa versicolor virginica class.error
setosa         50          0         0        0.00
versicolor      0         47         3        0.06
virginica       0          3        47        0.06
```

```
> importance(iris.rf) # importance of each predictor
             MeanDecreaseGini
Sepal.Length         9.654747
Sepal.Width          2.591063
Petal.Length        44.131695
Petal.Width         42.850817
```

Making a Prediction

```
library(randomForest)
iris.t1.rf <- randomForest(Species ~ Sepal.Length+Sepal.Width+
Petal.Length+Petal.Width,data=iristest1)



newdata2index=sample(1:150,20)
newdata2=iris[newdata2index,]

predict(iris.t1.rf,newdata2)
```

4.9 Regression Trees

- The general approach to derive predictions from few simple `if-then` conditions can be applied to regression problems as well.
- In this example, we will use the `cu.summary` data set in the ***rpart*** package.

```
> library(rpart)
> tail(cu.summary)
                     Price Country Reliability Mileage Type
Ford Aerostar V6     12267     USA     average      18  Van
Mazda MPV V6         14944   Japan Much better      19  Van
Mitsubishi Wagon 4   14929   Japan        <NA>      20  Van
Nissan Axxess 4      13949   Japan        <NA>      20  Van
Nissan Van 4         14799   Japan        <NA>      19  Van
Volkswagen Vanagon 4 14080 Germany        <NA>      NA  Van
```

```
fit <- rpart(Mileage~Price + Country + Reliability + Type,
   method="anova", data=cu.summary)

printcp(fit) # display the results
```

Display results of fitted tree

```
> printcp(fit) # display the results

Regression tree:
rpart(formula = Mileage ~ Price + Country + Reliability + Type,
    data = cu.summary, method = "anova")

Variables actually used in tree construction:
[1] Price Type

Root node error: 1354.6/60 = 22.576

n=60 (57 observations deleted due to missingness)

        CP nsplit rel error  xerror     xstd
1 0.622885      0   1.00000 1.00671 0.174008
2 0.132061      1   0.37711 0.52578 0.099645
3 0.025441      2   0.24505 0.36496 0.077946
4 0.011604      3   0.21961 0.35238 0.078224
5 0.010000      4   0.20801 0.35522 0.074295
```

```
# plot the regression tree
plot(fit, uniform=TRUE,
   main="Regression Tree for Mileage ")
text(fit, use.n=TRUE, all=TRUE, cex=1.0)
```

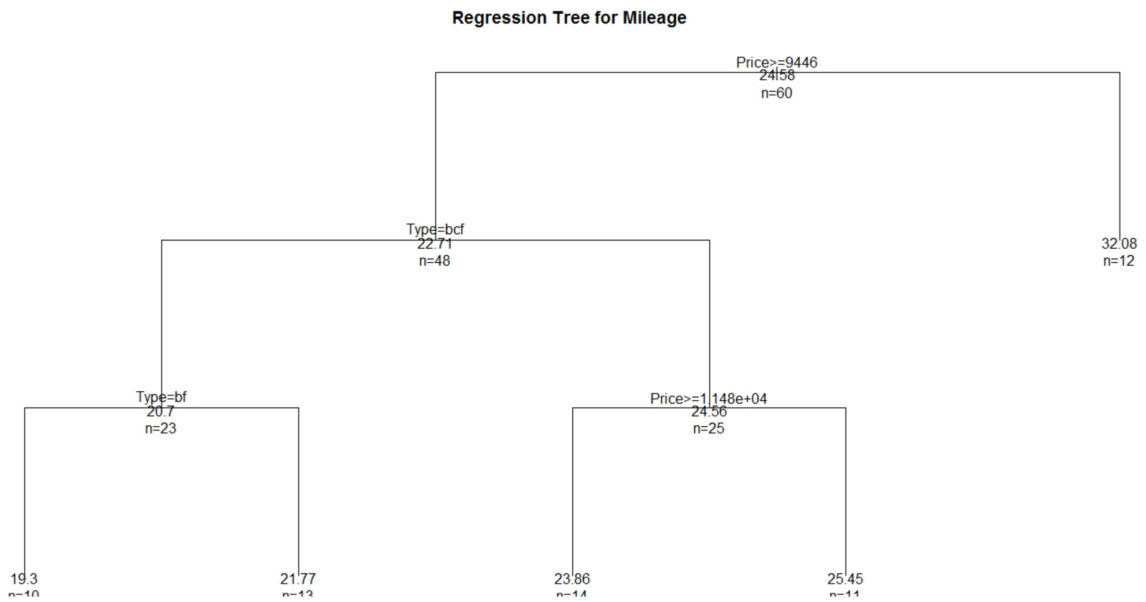

Figure 4.9:

Compare predicted values with observed values.

```
> cbind(predict(fit,cu.summary),cu.summary$Mileage)
                                  [,1] [,2]
Acura Integra 4               23.85714   NA
Dodge Colt 4                  32.08333   NA
Dodge Omni 4                  32.08333   NA
Eagle Summit 4                32.08333   33
Ford Escort   4               32.08333   33
Ford Festiva 4                32.08333   37
GEO Metro  3                  32.08333   NA
GEO Prizm  4                  25.45455   NA
................
```

4.10 Advantages of Classification and Regression Trees (CaRT) Methods

- As mentioned earlier, there are a large number of methods that an analyst can choose from when analyzing classification or regression problems.

- Tree classification techniques, when they ”work” and produce accurate predictions or predicted classifications based on few logical if-then conditions, have a number of advantages over many of those alternative techniques.

- **Simplicity of results** - In most cases, the interpretation of results summarized in a tree is very simple. This simplicity is useful not only for purposes of rapid classification of new observations (it is much easier to evaluate just one or two logical conditions, than to compute classification scores for each possible group, or predicted values, based on all predictors and using possibly some complex nonlinear model equations), but can also often yield a much simpler ”model” for explaining why observations are classified or predicted in a particular manner (e.g., when analyzing business problems, it is much easier to present a few simple *if-then* statements to management, than some elaborate equations).

- Tree methods are nonparametric and nonlinear. The final results of using tree methods for classification or regression can be summarized in a series of (usually few) logical if-then conditions (tree nodes). Therefore, there is no implicit assumption about underlying relationships between between variables.

- Thus, tree methods are particularly well suited for data mining tasks, where there is often little a priori knowledge nor any coherent set of theories or predictions regarding which variables are related and how. In those types of data analyses, tree methods can often reveal simple relationships between just a few variables that could have easily gone unnoticed using other analytic techniques.

4.11 Avoiding Over-Fitting: Pruning, Crossvalidation, and V-fold Crossvalidation

- 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 we had a data set with 20 cases, and performed 19 splits (determined 9 if-then conditions), we could perfectly predict every single case.

- In general, if we only split a sufficient number of times, eventually we will be able to "predict" our original data (from which we 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.

- **Overfitting** - This general issue is relevant to tree classification (as well as neural networks and regression methods)

- 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 we 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.

Criteria

There are many such criteria for automatically stopping the splitting (tree-building) process.

- **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.

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

- **V-fold crossvalidation** Continuing further along this line of reasoning, we 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 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.

4.12 Computational Details

The process of computing classification and regression trees can be characterized as involving four basic steps:

1. Specifying the criteria for predictive accuracy
2. Selecting splits
3. Determining when to stop splitting
4. Selecting the "right-sized" tree.

```
library(caret)
set.seed(3456)

#Split the data in training and testing data

trainIndex <- createDataPartition(iris$Species, p = .8,
                                  list = FALSE,
                                  times = 1)
head(trainIndex)

irisTrain <- iris[ trainIndex,]
irisTest  <- iris[-trainIndex,]

# Fit a model using the training data
iris.caret <- train(Species ~ ., data = irisTrain,
   method = "rpart", maxdepth=4)

iris.caret

# Try it out on the testing data
iris.caret.test <- predict(iris.caret,irisTest)



table(iris.caret.test,irisTest$Species)
```

```
> table(iris.caret.test,irisTest$Species)
```

```
iris.caret.test setosa versicolor virginica
     setosa         10          0          0
     versicolor      0         10          0
     virginica       0          0         10
```

4.13 Specifying the Criteria for Predictive Accuracy

The classification and regression trees (C&RT) algorithms are generally aimed at achieving the best possible predictive accuracy. Operationally, the most accurate prediction is defined as the prediction with the minimum costs. The notion of costs was developed as a way to generalize, to a broader range of prediction situations, the idea that the best prediction has the lowest misclassification rate. In most applications, the cost is measured in terms of proportion of misclassified cases, or variance. In this context, it follows, therefore, that a prediction would be considered best if it has the lowest misclassification rate or the smallest variance. The need for minimizing costs, rather than just the proportion of misclassified cases, arises when some predictions that fail are more catastrophic than others, or when some predictions that fail occur more frequently than others.

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.

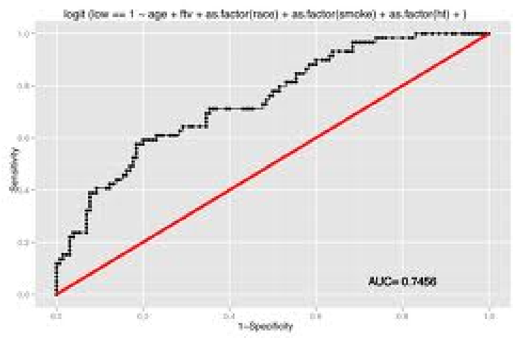

Figure 5.10:

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 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 .

9 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
```

```
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
```

10 caret-preprocessing

10.1 Creating Dummy Variables

The function `dummyVars` can be used to generate a complete (less than full rank parameterized) set of dummy variables from one or more factors. The function takes a formula and a data set and outputs an object that can be used to create the dummy variables using the predict `method.` For example, the `etitanic` data set in the ***earth*** package includes two factors: pclass1 (with levels 1st, 2nd, 3rd) and *sex* (with levels female, male). The base `R` function `model.matrix` would generate the following variables:

```
library(earth)
data(etitanic)
head(model.matrix(survived ~ ., data = etitanic))
```

```
  (Intercept) pclass2nd pclass3rd sexmale     age sibsp parch
1           1         0         0       0 29.0000     0     0
2           1         0         0       1  0.9167     1     2
3           1         0         0       0  2.0000     1     2
4           1         0         0       1 30.0000     1     2
5           1         0         0       0 25.0000     1     2
6           1         0         0       1 48.0000     0     0
```

Using `dummyVars`:

```
dummies <- dummyVars(survived ~ ., data = etitanic)
head(predict(dummies, newdata = etitanic))
```

```
  pclass.1st pclass.2nd pclass.3rd sex.female sex.male     age sibsp parch
1          1          0          0          1        0 29.0000     0     0
2          1          0          0          0        1  0.9167     1     2
3          1          0          0          1        0  2.0000     1     2
4          1          0          0          0        1 30.0000     1     2
5          1          0          0          1        0 25.0000     1     2
6          1          0          0          0        1 48.0000     0     0
```

Note there is no intercept and each factor has a dummy variable for each level, so this parameterization may not be useful for some model functions, such as `lm`*.*

10.2 Zero- and Near Zero-Variance Predictors

In some situations, the data generating mechanism can create predictors that only have a single unique value (i.e. a ”zero-variance predictor”). For many models (excluding tree-based models), this may cause the model to crash or the fit to be unstable.

Similarly, predictors might have only a handful of unique values that occur with very low frequencies. For example, in the drug resistance data, the nR11 descriptor (number of 11-membered rings) data have a few unique numeric values that are highly unbalanced:

```
data(mdrr)
data.frame(table(mdrrDescr$nR11))
```

```
  Var1 Freq
1    0  501
2    1    4
3    2   23
```

The concern here that these predictors may become zero-variance predictors when the data are split into cross-validation/bootstrap sub-samples or that a few samples may have an undue influence on the model. These ”near-zero-variance” predictors may need to be identified and eliminated prior to modeling.

To identify these types of predictors, the following two metrics can be calculated:

1. the frequency of the most prevalent value over the second most frequent value (called the ”frequency ratio”), which would be near one for well-behaved predictors and very large for highly-unbalanced data¿

2. the ”percent of unique values” is the number of unique values divided by the total number of samples (times 100) that approaches zero as the granularity of the data increases

If the frequency ratio is less than a pre-specified threshold and the unique value percentage is less than a threshold, we might consider a predictor to be near zero-variance.

Looking at the MDRR data, the `nearZeroVar` function can be used to identify near zero-variance variables (the saveMetrics argument can be used to show the details and usually defaults to FALSE):

```
nzv <- nearZeroVar(mdrrDescr, saveMetrics = TRUE)
nzv[nzv$nzv, ][1:10, ]
       freqRatio percentUnique zeroVar  nzv
nTB        23.00        0.3788   FALSE TRUE
nBR       131.00        0.3788   FALSE TRUE
nI        527.00        0.3788   FALSE TRUE
nR03      527.00        0.3788   FALSE TRUE
nR08      527.00        0.3788   FALSE TRUE
```

```
nR11       21.78        0.5682   FALSE TRUE
nR12       57.67        0.3788   FALSE TRUE
D.Dr03    527.00        0.3788   FALSE TRUE
D.Dr07    123.50        5.8712   FALSE TRUE
D.Dr08    527.00        0.3788   FALSE TRUE
dim(mdrrDescr)
[1] 528 342

nzv <- nearZeroVar(mdrrDescr)
filteredDescr <- mdrrDescr[, -nzv]
dim(filteredDescr)
[1] 528 297
```

By default, `nearZeroVar` will return the positions of the variables that are flagged to be problematic.

10.3 Identifying Correlated Predictors

While there are some models that thrive on correlated predictors (such as `pls`), other models may benefit from reducing the level of correlation between the predictors.

Given a correlation matrix, the `findCorrelation` function uses the following algorithm to flag predictors for removal:

```
descrCor <- cor(filteredDescr)
highCorr <- sum(abs(descrCor[upper.tri(descrCor)]) > 0.999)
```

For the previous MDRR data, there are 65 descriptors that are almost perfectly correlated, such as the total information index of atomic composition (IAC) and the total information content index (neighborhood symmetry of 0-order) (TIC0) (correlation = 1). The code chunk below shows the effect of removing descriptors with absolute correlations above 0.75.

```
descrCor <- cor(filteredDescr)
summary(descrCor[upper.tri(descrCor)])
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
-0.9960 -0.0537  0.2500  0.2610  0.6550  1.0000
highlyCorDescr <- findCorrelation(descrCor, cutoff = 0.75)
filteredDescr <- filteredDescr[, -highlyCorDescr]
descrCor2 <- cor(filteredDescr)
summary(descrCor2[upper.tri(descrCor2)])
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
-0.7070 -0.0538  0.0442  0.0669  0.1890  0.7450
```

10.4 Linear Dependencies

The function `findLinearCombos` uses the QR decomposition of a matrix to enumerate sets of linear combinations (if they exist). For example, consider the following matrix that is could have been produced by a less-than-full-rank parameterizations of a two-way experimental layout:

```
ltfrDesign <- matrix(0, nrow = 6, ncol = 6)
ltfrDesign[, 1] <- c(1, 1, 1, 1, 1, 1)
ltfrDesign[, 2] <- c(1, 1, 1, 0, 0, 0)
ltfrDesign[, 3] <- c(0, 0, 0, 1, 1, 1)
ltfrDesign[, 4] <- c(1, 0, 0, 1, 0, 0)
ltfrDesign[, 5] <- c(0, 1, 0, 0, 1, 0)
ltfrDesign[, 6] <- c(0, 0, 1, 0, 0, 1)
```

Note that columns two and three add up to the first column. Similarly, columns four, five and six add up the first column. `findLinearCombos` will return a list that enumerates these dependencies.

For each linear combination, it will incrementally remove columns from the matrix and test to see if the dependencies have been resolved. `findLinearCombos` will also return a vector of column positions can be removed to eliminate the linear dependencies:

```
comboInfo <- findLinearCombos(ltfrDesign)
comboInfo
```

```
$linearCombos
$linearCombos[[1]]
[1] 3 1 2


$linearCombos[[2]]
[1] 6 1 4 5
```

```
$remove
[1] 3 6
ltfrDesign[, -comboInfo$remove]
     [,1] [,2] [,3] [,4]
[1,]    1    1    1    0
[2,]    1    1    0    1
[3,]    1    1    0    0
[4,]    1    0    1    0
[5,]    1    0    0    1
[6,]    1    0    0    0
```

These types of dependencies can arise when large numbers of binary chemical fingerprints are used to describe the structure of a molecule.

10.5 Centering and Scaling

The `preProcess` class can be used for many operations on predictors, including ***centering*** and ***scaling***. The function `preProcess` estimates the required parameters for each operation and `predict.preProcess` is used to apply them to specific data sets.

In the example below, the half of the MDRR data are used to estimate the location and scale of the predictors. The function `preProcess` doesn't actually pre-process the data. `predict.preProcess` is used to pre-process this and other data sets.

```
set.seed(96)
inTrain <- sample(seq(along = mdrrClass), length(mdrrClass)/2)
training <- filteredDescr[inTrain, ]
test <- filteredDescr[-inTrain, ]
trainMDRR <- mdrrClass[inTrain]
testMDRR <- mdrrClass[-inTrain]
preProcValues <- preProcess(training, method = c("center", "scale"))
trainTransformed <- predict(preProcValues, training)
testTransformed <- predict(preProcValues, test)
```

The ***preProcess*** option ”ranges” scales the data to the interval [0, 1].

10.6 Imputation

- In statistics, imputation is the process of replacing missing data with substituted values.
- `preProcess` can be used to ***impute*** data sets based only on information in the training set. One method of doing this is with K-nearest neighbors.
- For an arbitrary sample, the K closest neighbors are found in the training set and the value for the predictor is imputed using these values (e.g. using the mean).
- Using this approach will automatically trigger `preProcess` to center and scale the data, regardless of what is in the method argument.
- Alternatively, bagged trees can also be used to impute. For each predictor in the data, a bagged tree is created using all of the other predictors in the training set.
- When a new sample has a missing predictor value, the bagged model is used to predict the value. While, in theory, this is a more powerful method of imputing, the computational costs are much higher than the nearest neighbor technique.

10.7 Transforming Predictors

- In some cases, there is a need to use principal component analysis (PCA) to transform the data to a smaller subspace where the new variable are uncorrelated with one another. The preProcess class can apply this transformation by including "`pca`" in the method argument.
- Doing this will also force scaling of the predictors. Note that when PCA is requested, `predict.preProcess` changes the column names to PC1, PC2 and so on.
- Similarly, independent component analysis (ICA) can also be used to find new variables that are linear combinations of the original set such that the components are independent (as opposed to uncorrelated in PCA). The new variables will be labeled as IC1, IC2 and so on.

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 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).

M Metal cylinder

R Rock

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

```
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
```

15.1 Partial least squares

Partial least squares regression (PLS regression) is a statistical method that bears some relation to principal components regression; instead of finding hyperplanes of minimum variance between the response and independent variables, it finds a linear regression model by projecting the predicted variables and the observable variables to a new space. Because both the X and Y data are projected to new spaces, the PLS family of methods are known as bilinear factor models. Partial least squares Discriminant Analysis (PLS-DA) is a variant used when the Y is categorical.

```
training <- Sonar[ inTrain,]
testing <- Sonar[-inTrain,]
nrow(training)
nrow(testing)

plsFit <- train(Class ~ .,
 data = training,
 method = "pls",
 ## Center and scale the predictors for the training
 ## set and all future samples.
 preProc = c("center", "scale"))
```

```
ctrl <- trainControl(method = "repeatedcv", repeats = 3)
plsFit <- train(Class ~ .,
data = training,
method = "pls",
tuneLength = 15,
trControl = ctrl,
preProc = c("center", "scale"))
```

16 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
```

17 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.

18 Variable importance

Random Forests can output a list of predictor variables that they believe to be important in predicting the outcome.

If nothing else, you can subset the data to only include the most ”important” variables, and use that with another model.

The randomForest package in R has two measures of importance.

- One is ”total decrease in node impurities from splitting on the variable, averaged over all trees.”
- The other is based on a permutation test. The idea is that if the variable is not important (the null hypothesis), then rearranging the values of that variable will not degrade prediction accuracy.

```
# simulate the data
x1=rnorm(1000)
x2=rnorm(1000,x1,1)
y=2*x1+rnorm(1000,0,.5)
df=data.frame(y,x1,x2,x3=rnorm(1000),x4=rnorm(1000),x5=rnorm(1000))

# run the randomForest implementation
library(randomForest)
rf1 <- randomForest(y~., data=df, mtry=2, ntree=50, importance=TRUE)
importance(rf1,type=1)



# run the party implementation
library(party)
cf1 <- cforest(y~.,data=df,control=cforest_unbiased(mtry=2,ntree=50))
varimp(cf1)
varimp(cf1,conditional=TRUE)
```

For the randomForest, the ratio of importance of the the first and second variable is 4.53. For party without accounting for correlation it is 7.35. And accounting for correlation, it is 369.5. The higher ratios are better because it means that the importance of the first variable is more prominent

Variable importance evaluation functions can be separated into two groups: those that use the model information and those that do not. The advantage of using a model-based approach is that is more closely tied to the model performance and that it may be able to incorporate the correlation structure between the predictors into the importance calculation. Regardless of how the importance is calculated:

- For most classification models, each predictor will have a separate variable importance for each class (the exceptions are classification trees, bagged trees and boosted trees).
- All measures of importance are scaled to have a maximum value of 100, unless the scale argument of varImp.train is set to FALSE.

18.1 Model Specific Metrics

The following methods for estimating the contribution of each variable to the model are available:

Linear Models the absolute value of the t-statistic for each model parameter is used.

Random Forest from the R package: "For each tree, the prediction accuracy on the out-of-bag portion of the data is recorded. Then the same is done after permuting each predictor variable. The difference between the two accuracies are then averaged over all trees, and normalized by the standard error. For regression, the MSE is computed on the out-of-bag data for each tree, and then the same computed after permuting a variable. The differences are averaged and normalized by the standard error. If the standard error is equal to 0 for a variable, the division is not done."