

Ego autem et domus



mea serviemus Domino.

92

APPLIED PREDICTIVE MODELING

Techniques in R

*Over ninety of the most important models used by
successful Data Scientists; With step by step
instructions on how to build them FAST!*

Dr. N.D. Lewis

Copyright © 2015 by N.D. Lewis

All rights reserved. No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the author, except in the case of brief quotations embodied in critical reviews and certain other noncommercial uses permitted by copyright law. For permission requests, contact the author at: www.AusCov.com.

Disclaimer: Although the author and publisher have made every effort to ensure that the information in this book was correct at press time, the author and publisher do not assume and hereby disclaim any liability to any party for any loss, damage, or disruption caused by errors or omissions, whether such errors or omissions result from negligence, accident, or any other cause.

Ordering Information: Quantity sales. Special discounts are available on quantity purchases by corporations, associations, and others. For details, email: info@NigelDLewis.com

Image photography by Deanna Lewis

ISBN-13: 978-1517516796

ISBN-10: 151751679X

Dedicated to Angela, wife, friend and mother extraordinaire.

Acknowledgments

A special thank you to:

My wife Angela, for her patience and constant encouragement.

My daughter Deanna, for taking hundreds of photographs for this book and my website.

And the readers of my earlier books who contacted me with questions and suggestions.

About This Book

This jam-packed book takes you under the hood with step by step instructions using the popular and free R predictive analytics package. It provides numerous examples, illustrations and exclusive use of real data to help you leverage the power of predictive analytics. A book for every data analyst, student and applied researcher. Here is what it can do for you:

- **BOOST PRODUCTIVITY:** Bestselling author and data scientist Dr. N.D. Lewis will show you how to build predictive analytic models in less time than you ever imagined possible! Even if you're a busy professional or a student with little time. By spending as little as 10 minutes a day working through the dozens of real world examples, illustrations, practitioner tips and notes, you'll be able to make giant leaps forward in your knowledge, strengthen your business performance, broaden your skill-set and improve your understanding.
- **SIMPLIFY ANALYSIS:** You will discover over 90 easy to follow applied predictive analytic techniques that can instantly expand your modeling capability. Plus you'll discover simple routines that serve as a check list you repeat next time you need a specific model. Even better, you'll discover practitioner tips, work with real data and receive suggestions that will speed up your progress. So even if you're completely stressed out by data, you'll still find in this book tips, suggestions and helpful advice that will ease your journey through the data science maze.
- **SAVE TIME:** Imagine having at your fingertips easy access to the very best of predictive analytics. In this book, you'll learn fast effective ways to build powerful models using R. It contains over 90 of the most successful models used for learning from data; With step by step instructions on how to build them easily and quickly.
- **LEARN FASTER:** **92 Applied Predictive Modeling Techniques in R** offers a practical results orientated approach that will boost your productivity, expand your knowledge and create new and exciting opportunities for you to get the very best from your data. The book works because you eliminate the anxiety of trying to master every single mathematical detail. Instead your goal at each step is to simply focus on a single routine using real data that only takes about 5 to 15 minutes to complete. Within this routine is a series of actions by which the predictive analytic model is constructed. All you have to do is follow the steps. They are your checklist for use and reuse.

- IMPROVE RESULTS: Want to improve your predictive analytic results, but don't have enough time? Right now there are a dozen ways to instantly improve your predictive models performance. Odds are, these techniques will only take a few minutes apiece to complete. The problem? You might feel like there's not enough time to learn how to do them all. The solution is in your hands. It uses R, which is free, open-source, and extremely powerful software.

In this rich, fascinating—surprisingly accessible—guide, data scientist Dr. N.D. Lewis reveals how predictive analytics works, and how to deploy its power using the free and widely available R predictive analytics package. The book serves practitioners and experts alike by covering real life case studies and the latest state-of-the-art techniques. Everything you need to get started is contained within this book. Here is some of what is included:

- Support Vector Machines
- Relevance Vector Machines
- Neural networks
- Random forests
- Random ferns
- Classical Boosting
- Model based boosting
- Decision trees
- Cluster Analysis

For people interested in statistics, machine learning, data analysis, data mining, and future hands-on practitioners seeking a career in the field, it sets a strong foundation, delivers the prerequisite knowledge, and whets your appetite for more. Buy the book today. Your next big breakthrough using predictive analytics is only a page away!

OTHER BOOKS YOU WILL ALSO ENJOY

Over 100 Statistical Tests at Your Fingertips!

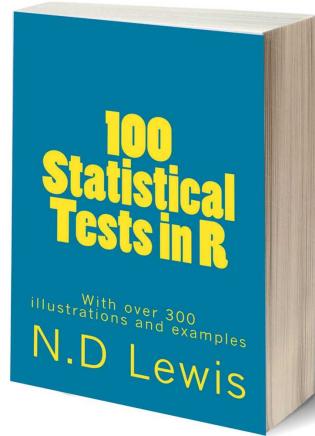
100 Statistical Tests in R is designed to give you rapid access to one hundred of the most popular statistical tests.

It shows you, step by step, how to carry out these tests in the free and popular R statistical package.

The book was created for the applied researcher whose primary focus is on their subject matter rather than mathematical lemmas or statistical theory.

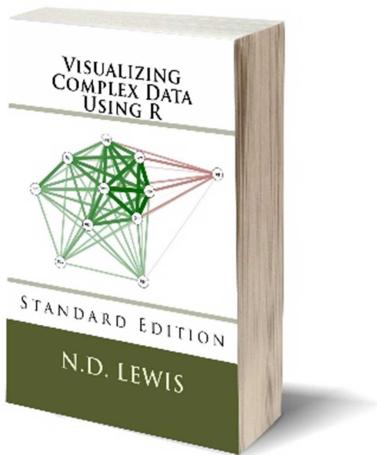
Step by step examples of each test are clearly described, and can be typed directly into R as printed on the page.

To accelerate your research ideas, over three hundred applications of statistical tests across engineering, science, and the social sciences are discussed.



100 Statistical Tests in R - [ORDER YOUR COPY TODAY!](#)

"They laughed as they gave me the data to analyze...But then they saw my charts!"



Wish you had fresh ways to present data, explore relationships, visualize your data and break free from mundane charts and diagrams?

Visualizing complex relationships with ease using R begins here.

In this book you will find innovative ideas to unlock the relationships in your own data and create killer visuals to help you transform your next presentation from good to great.

Visualizing Complex
Data Using R - [ORDER](#)
[YOUR COPY TODAY!](#)

Preface

In writing this text my intention was to collect together in a single place practical predictive modeling techniques, ideas and strategies that have been proven to work but which are rarely taught in business schools, data science courses or contained in any other single text.

On numerous occasions, researchers in a wide variety of subject areas, have asked “how can I quickly understand and build a particular predictive model?” The answer used to involve reading complex mathematical texts and then programming complicated formulas in languages such as C, C++ and Java. With the rise of R, predictive analytics is now easier than ever. **92 Applied Predictive Modeling Techniques in R** is designed to give you rapid access to over ninety of the most popular predictive analytic techniques. It shows you, step by step, how to build each model in the free and popular R statistical package.

The material you are about to read is based on my personal experience, articles I’ve written, hundreds of scholarly articles I’ve read over the years, experimentation some successful some failed, conversations I’ve had with data scientists in various fields and feedback I’ve received from numerous presentations to people just like you.

This book came out of the desire to put predictive analytic tools in the hands of the practitioner. The material is therefore designed to be used by the applied data scientist whose primary focus is on delivering results rather than mathematical lemmas or statistical theory. Examples of each technique are clearly described and can be typed directly into R as printed on the page.

This book in your hands is an enlarged, revised, and updated collection of my previous works on the subject. I’ve condensed into this volume the best practical ideas available.

Data science is all about extracting meaningful structure from data. It is always a good idea for the data scientist to study how other users and researchers have used a technique in actual practice. This is primarily because practice often differs substantially from the classroom or theoretical text books. To this end and to accelerate your progress, actual real world applications of the techniques are given at the start of each section.

These illustrative applications cover a vast range of disciplines incorporating numerous diverse topics such as intelligent shoes, forecasting the stock market, signature authentication, oil sand pump prognostics, detecting deception in speech, electric fish localization, tropical forest carbon mapping, vehicle logo recognition, understanding rat talk, and many more! I have also provided detailed references to these application for further study at the end of each section.

In keeping with the zeitgeist of R, copies of the vast majority of applied articles referenced in this text are available for free.

New users to R can use this book easily and without any prior knowledge. This is best achieved by typing in the examples as they are given and reading the comments which follow. Copies of R and free tutorial guides for beginners can be downloaded at <https://www.r-project.org/>

I have found, over and over, that a data scientist who has exposure to a broad range of modeling tools and applications will run circles around the narrowly focused genius who has only been exposed to the tools of their particular discipline.

Greek philosopher Epicurus once said “I write this not for the many, but for you; each of us is enough of an audience for the other.” Although the ideas in this book reach out to thousands of individuals, I’ve tried to keep Epicurus’s principle in mind—to have each page you read give meaning to just one person - YOU.

I invite you to put what you read in these pages into action. To help you do that, I’ve created “**12 Resources to Supercharge Your Productivity in R**”, it is yours for free. Simply go to <http://www.auscov.com/tools.html> and download it now. It’s my gift to you. It shares with you 12 of the very best resources you can use to boost your productivity in R.

I’ve spoken to thousands of people over the past few years. I’d love to hear your experiences using the ideas in this book. Contact me with your stories, questions and suggestions at Info@NigelDLewis.com.

Now, it’s your turn!

A handwritten signature in black ink that reads "Dr. Nigel D. Lewis". The signature is fluid and cursive, with "Dr." being a small prefix before "Nigel D. Lewis".

P.S. Don’t forget to sign-up for your free copy of 12 Resources to Supercharge Your Productivity in R at <http://www.auscov.com/tools.html>

How to Get the Most from this Book

There are at least three ways to use this book. First, you can dip into it as an efficient reference tool. Flip to the technique you need and quickly see how to calculate it in R. For best results type in the example given in the text, examine the results, and then adjust the example to your own data. Second, browse through the real world examples, illustrations, practitioner tips and notes to stimulate your own research ideas. Third, by working through the numerous examples, you will strengthen your knowledge and understanding of both applied predictive modeling and R.

Each section begins with a brief description of the underlying modeling methodology followed by a diverse array of real world applications. This is followed by a step by step guide using real data for each predictive analytic technique.

☛ PRACTITIONER TIP ☛

If you are using Windows you can easily upgrade to the latest version of R using the `installr` package. Enter the following:

```
> install.packages("installr")
> installr::updateR()
```

If a package mentioned in the text is not installed on your machine you can download it by typing `install.packages("package_name")`. For example to download the `ada` package you would type in the R console:

```
> install.packages("ada")
```

Once a package is installed, you must call it. You do this by typing in the R console:

```
> require(ada)
```

The `ada` package is now ready for use. You only need to type this once, at the start of your R session.

☛ PRACTITIONER TIP ☛

You should only download packages from CRAN using encrypted HTTPS connections. This provides much higher assurance that the code you are downloading is from a legitimate CRAN mirror rather than from another server posing as one. Whilst downloading a package from a HTTPS connection you may run into a error message something like:

```
"unable to access index for repository  
https://cran.rstudio.com/..."
```

This is particularly common on Windows. The `internet2` dll has to be activated on versions before R-3.2.2. If you are using an older version of R before downloading a new package enter the following:

```
> setInternet2(TRUE)
```

Functions in R often have multiple parameters. In the examples in this text I focus primarily on the key parameters required for rapid model development. For information on additional parameters available in a function type in the R console `?function_name`. For example, to find out about additional parameters in the `ada` function, you would type:

```
?ada
```

Details of the function and additional parameters will appear in your default web browser. After fitting your model of interest you are strongly encouraged to experiment with additional parameters. I have also included the `set.seed` method in the R code samples throughout this text to assist you in reproducing the results exactly as they appear on the page. R is available for all the major operating systems. Due to the popularity of windows, examples in this book use the windows version of R.

◆ PRACTITIONER TIP ◆

Can't remember what you typed two hours ago! Don't worry, neither can I! Provided you are logged into the same R session you simply need to type:

```
> history(Inf)
```

It will return your entire history of entered commands for your current session.

You don't have to wait until you have read the entire book to incorporate the ideas into your own analysis. You can experience their marvelous potency for yourself almost immediately. You can go straight to the technique of interest and immediately test, create and exploit it in your own research and analysis.

◆ PRACTITIONER TIP ◆

On 32-bit Windows machines, R can only use up to 3Gb of RAM, regardless of how much you have installed. Use the following to check memory availability:

```
> memory.limit()
```

To remove all objects from memory:

```
rm(list=ls())
```

Applying the ideas in this book will transform your data science practice. If you utilize even one tip or idea from each chapter, you will be far better prepared not just to survive but to excel when faced by the challenges and opportunities of the ever expanding deluge of exploitable data.

Now let's get started!

Part I

Decision Trees

The Basic Idea

We begin with decision trees because they are one of the most popular techniques in data mining¹. They can be applied to both regression and classification problems. Part of the reason for their popularity lies in the ability to present results a simple easy to understand tree format. True to its name, the decision tree selects an outcome by descending a tree of possible decisions.

NOTE... ↗

Decision trees are, in general, a non-parametric inductive learning technique, able to produce classifiers for a given problem which can assess new, unseen situations and/or reveal the mechanisms driving a problem.

An Illustrative Example

It will be helpful to build intuition by first looking at a simple example of what this technique does with data. Imagine you are required to build an automatic rules based system to buy cars. The goal is to make decisions as new vehicles are presented to you. Let us say you have access to data on the attributes (also known as features) - Road tested miles driven, Price of vehicle, Likability of the current owner (measured on a continuous scale from 0 to 100, 100 = “love them!”), Odometer miles, Age of the vehicle in years.

A total of 100 measurements are obtained on each variable and also on the decision (yes or no to purchase the vehicle). You run this data through a decision tree algorithm and it produces the tree shown in Figure 1. Several things are worth pointing out about this decision tree. First, the number of observations falling in “yes” and “no” is reported. For example, in “Road tested miles <100” we see there are 71 observations. Second, the tree is immediately able to model new data using the rules developed. Third, it did not use all of the variables to develop a decision rule (Likability of the current

owner and Price were excluded).

Let us suppose that this tree classified 80% of the observations correctly. It is still worth investigating whether a more parsimonious and more accurate tree can be obtained. One way to achieve this is to transform the variables. Let us suppose we create the additional variable $\frac{Odometer}{Age}$, and then rebuild the tree. The result is shown in Figure 2.

It turns out that this decision tree, which chooses only the transformed data, ignores all the other attributes. Let us assume this tree has a prediction accuracy of 90%. Two important points become evident. First, a more parsimonious and accurate tree was possible and second, to obtain this tree it was necessary to include the variable transformations in the second run of the decision tree algorithm.

The example illustrates that the decision tree must have the variables supplied in the appropriate form to obtain the most parsimonious tree. In practice domain experts will often advise on the appropriate transformation of attributes.

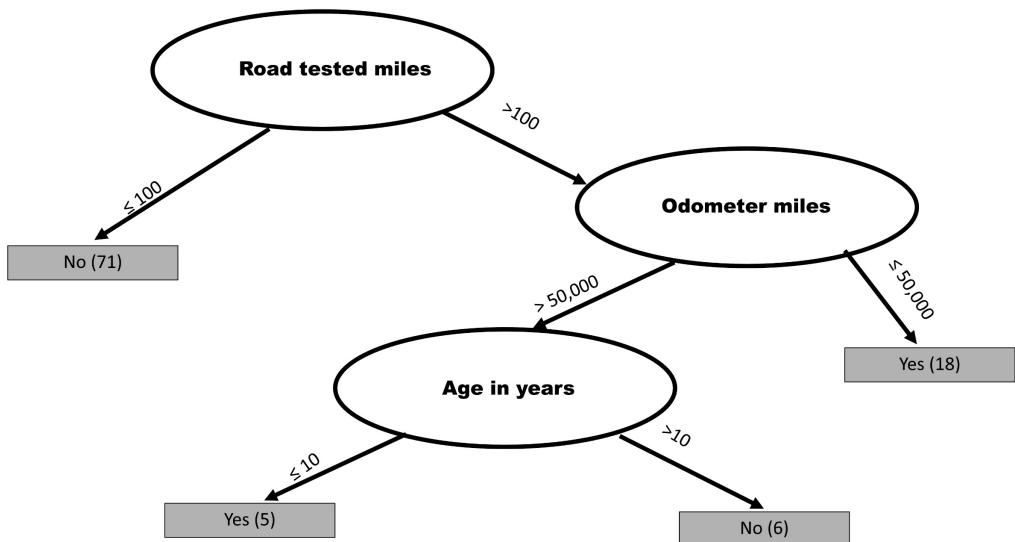


Figure 1: Car buying decision tree

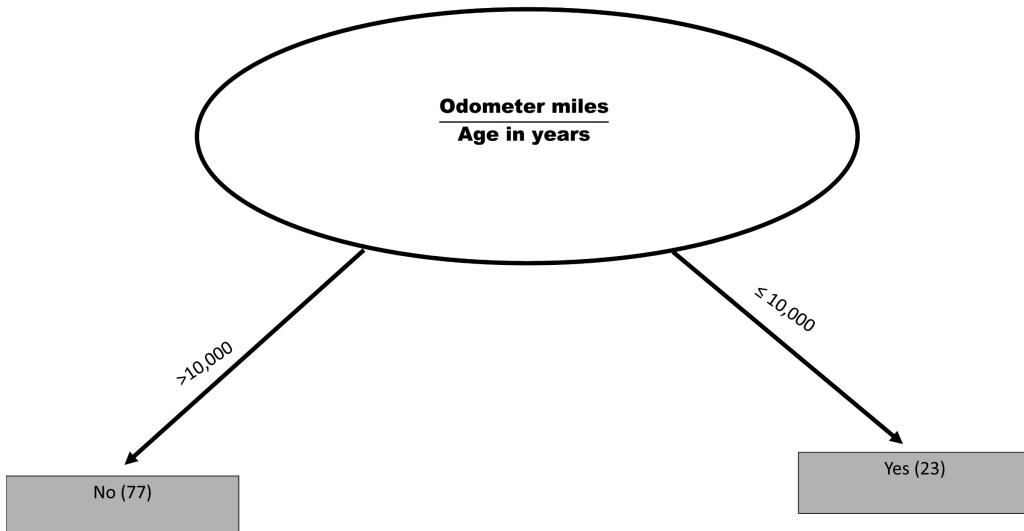


Figure 2: Decision tree obtained by transforming variables

☛ PRACTITIONER TIP ☛

I once developed what I thought was a great statistical model for an area I knew little about. Guess what happened? It was a total flop! Why? Because I did not include domain experts in my design and analysis phase. If you are building a decision trees for knowledge discovery, it is important to include domain experts alongside you and throughout the entire analysis process. Their input will be required to assess the final decision tree and opine on “reasonability”.

Another advantage of using domain experts is that the complexity of the final decision tree, (in terms of the number of nodes or the number of rules that can be extracted from a tree) may be reduced. Inclusion of domain experts almost always helps the data scientist create a more efficient set of rules.

Decision Trees in Practice

The basic idea behind a decision tree is to construct a tree whose leaves are labeled with a particular value for the class attribute and whose inner nodes

represent descriptive attributes. At each internal node in the tree, a single attribute value is compared with a single threshold value. In other words, each node corresponds to a “measurement” of a particular attribute—that is, a question, often of the “yes” or “no” variety, which can be asked about that attribute’s value (e.g. “is age less than 4.8 years?”). One of the two child nodes is then selected based on the result of the comparison; and hence another measurement—or to a leaf.

When a leaf node is reached, the single class associated with that node is the final prediction. In other words, the terminal nodes carry the information required to classify the data.

A real world example of decision trees is shown in Figure 3; they were developed by Koch et al² for understanding biological cellular signaling networks. Notice that four trees of increasing complexity are developed by the researchers.

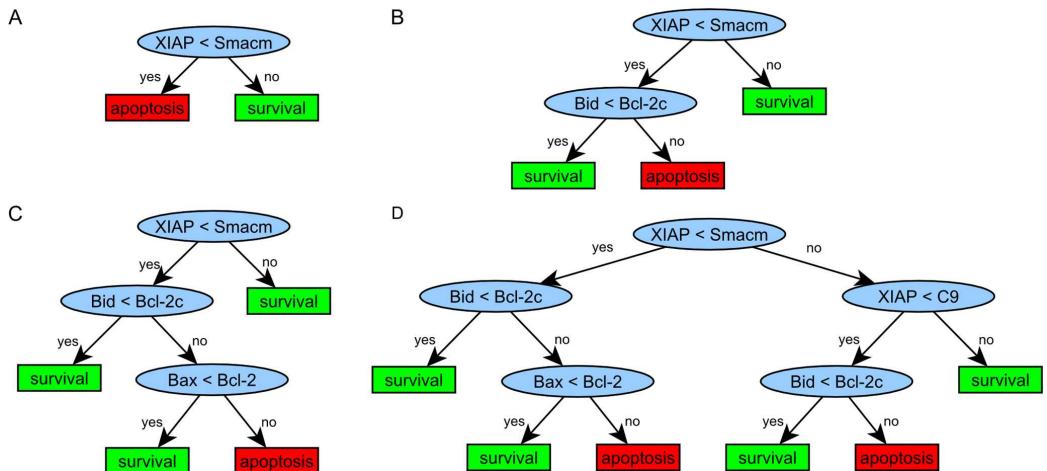


Figure 3: Four decision trees developed by Koch et al. Note that decision tree (A) (B) (C) and (D) have a misclassification error of 30.85% , 24.68%, 21.17% , and 18.13% respectively. However, tree (A) is the easiest to interpret. Source Koch et al.

The Six Advantages of Decision Trees

1. Decision trees can be easy-to-understand with intuitively clear rules understandable to domain experts.
2. Decision trees offer the ability to track and evaluate every step in the decision-making process. This is because each path through a tree consists of a combination of attributes which work together to distinguish

between classes. This simplicity gives useful insights into the inner workings of the method.

3. Decision trees can handle both nominal and numeric input attributes and are capable of handling data sets that contain misclassified values.
4. Decision trees can easily be programmed for use in real time systems. A great illustration of this is the research of Hailemariam et al³ who use a decision tree to determine real time building occupancy.
5. They are relatively inexpensive computationally and work well on both large and small data sets. Figure 4 illustrates an example of a very large decision tree used in Bioinformatics⁴; the smaller tree represents the tuned version for greater readability.
6. Decision trees are considered to be a non-parametric method. This means that decision trees have no assumptions about the space distribution and on the classifier structure.

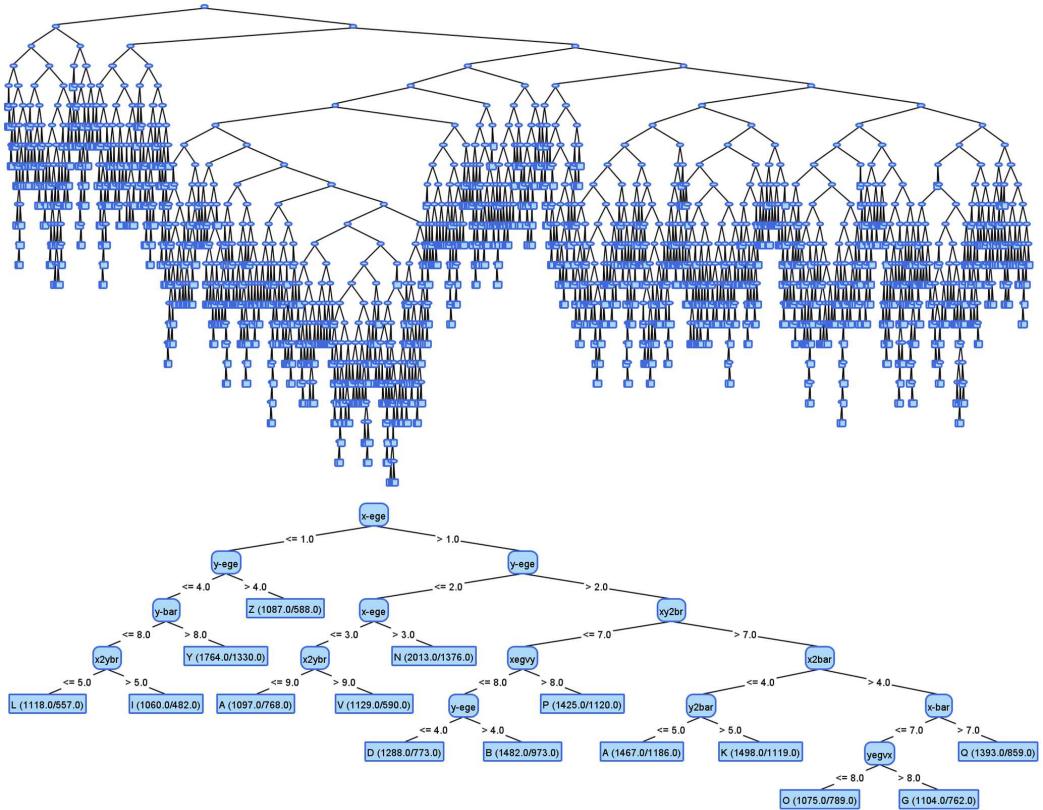


Figure 4: An example of a large Bioinformatics decision tree and the visually tuned version from the research of Stiglic et al.

NOTE... ↗

One weakness of decision trees is the risk of over fitting. This occurs when statistically insignificant patterns end up influencing classification results. An over fitted tree will perform poorly on new data. Bohanec and Bratko⁵ studied the role of pruning a decision tree for better decision making. They found that pruning can reduce the risk of over fitting because it results in smaller decision trees that exploit fewer attributes.

How Decision Trees Work

Exact implementation details differ somewhat depending on the algorithm used, but the general principles are very similar across methods and contain

the following steps.

1. Provide a set (S), of examples with known classified states. This is called the learning set.
2. Select a set of test attributes attributes, a_1, a_2, \dots, a_N . These can be viewed as the parameters of the system and are selected because they contain essential information about the problem of concern. In the car example on page 6 the attributes were - a_1 = Road tested miles driven, a_2 = Price of vehicle, a_3 = Likability of the current owner, a_4 = Odometer miles, a_5 = Age of the vehicle in years.
3. Starting at the top node of the tree (often called the root node) with the entire set of examples S , split S using a test on one or more attributes. The goal is to split S into subsets of increasing classification purity.
4. Check the results of the split. If every partition is pure in the sense that all examples in the partition belong to the same class then stop. Label each leaf node with the name of the class.
5. Recursively split any partitions that are not “pure”.
6. The procedure is stopped when all the newly created nodes are ‘terminal’ ones, containing “pure enough” learning subsets.

Decision tree algorithms vary primarily in how they choose to “split” the data, when to stop splitting and how they prune the trees they produce.

NOTE... ↗

Many of the decision tree algorithms you will encounter are based on a greedy top-down recursive partitioning strategy for tree growth. They use different variants of impurity measures such as - information gain⁶, gain ratio⁷, gini-index⁸ and distance-based measures⁹.

Practical Applications

Intelligent Shoes for Stroke Victims

Zhang et al¹⁰ develop a wearable shoe (SmartShoe) to monitor physical activity in stroke patients. The data set consisted of 12 patients who had experienced a stroke.

Supervised by a physical therapist, SmartShoe collected data from the patients using eight posture and activity groups: sitting, standing, walking, ascending stairs, descending stairs, cycling on a stationary bike, being pushed in a wheelchair, and propelling a wheelchair.

Patients performed each activity from between 1- 3 minutes. Data was collected from the SmartShoe every 2 seconds from which feature vectors were computed.

Half the feature features vectors were selected at random for the training set. The remainder were used for validation. The C5.0 algorithm was used to build the decision tree.

The researchers constructed both subject specific and group decision trees. The group models were developed using Leave-One-Out cross-validation.

The performance results for five of the patients are shown in Table 1. As might be expected the individual models fit better than the group models. For example, for patient 5 the accuracy of the patient specific tree was 98.4%, however using the group tree the accuracy declined to 75.5%. This difference in performance might be in part due to over fitting of the individual specific tree. The group models were trained using data from multiple subjects and therefore can be expected to have lower overall performance scores.

Patient	1	2	3	4	5	Average
Individual	96.5%.	97.4%	99.8%	97.2%	98.4%	97.9%
Group	87.5%	91.1%	64.7%	82.2%	75.5%	80.2%

Table 1: Zhang et al's decision tree performance metrics

• PRACTITIONER TIP •

Decision trees are often validated by calculating sensitivity and specificity. Sensitivity is the ability of the classifier to identify positive results, while specificity is the ability to distinguish negative results.

$$\text{Sensitivity} = \frac{NTP}{NTP + NTN} \times 100 \quad (1)$$

$$\text{Specificity} = \frac{NTN}{NTP + NTN} \times 100 \quad (2)$$

NTP is the number of true positives and NTN is the number of true negatives.

Micro Ribonucleic Acid

MicroRNAs (miRNAs) are non-protein coding Ribonucleic acids (RNAs) that attenuate protein production in P bodies¹¹. Williams et al¹² develop a MicroRNA decision tree. For the training set the researchers used known miRNAs associated from various plant species for positive controls and non-miRNA sequences as negative controls.

The typical size of their training set consisted of 5294 cases using 29 attributes. The model was validated by calculating sensitivity and specificity based on leave-one-out cross-validation.

After training, the researchers focus on attribute usage information. Table 2 shows the top ten attribute usage for a typical training run. The researchers report that other training runs show similar usage. The values represent the percentage of sequences that required that attribute for classification. Several attributes, such as DuplexEnergy, minMatchPercent, and C content, are required for all sequences to be classified. Note that G% and C% are directly related to the stability of the duplex. Sensitivity and specificity was as high as 84.08% and 98.53% respectively.

An interesting question is if all miRNAs in each taxonomic category studied by the researchers are systematically excluded from training while including all others, how well does the predictor do when tested on the excluded category? Table 2 provides the answer. The ability to correctly identify known miRNAs ranged from 78% for the Salicaceae to 100% for seven of the groups shown in Table 2. The researchers conclude by stating: “*We have*

Usage	Attribute
100%	G%
100%	C%
100%	T%
100%	DuplexEnergy
100%	minMatchPercent
100%	DeltaGnorm
100%	G + T
100%	G + C
98%	duplexEnergyNorm
86%	NormEnergyRatio

Table 2: Top ten attribute usage for one training runs of the classifier reported by Williams et al.

shown that a highly accurate universal plant miRNA predictor can be produced by machine learning using C5.0.”

Taxonomic Group	Correctly classified	% of full set excluded
Embryophyta	94%	9.16%
Lycopodiophyta	100%	2.65%
Brassicaceae	100%	20.22%
Caricaceae	100%	0.05%
Euphorbiaceae	100%	0.34%
Fabaceae	100%	27.00%
Salicaceae	78%	3.52%
Solanaceae	93%	0.68%
Vitaceae	94%	4.29%
Rutaceae	100%	0.43%
Panicoideae	95%	8.00%
Poaceae	100%	19.48%
Pooideae	80%	3.18%

Table 3: Results from exclusion of each of the 13 taxonomic groups by Williams et al.

NOTE... ↗

A decision tree produced by an algorithm is usually not optimal in the sense of statistical performance measures such as the log-likelihood, squared errors and so on. It turns out that finding the “optimal tree”, if one exists, is computationally intractable (or NP-hard, technically speaking).

Acute Liver Failure

Nakayama et al¹³ use decision trees for the prognosis of acute liver failure (ALF) patients. The data set consisted of a of 1,022 ALF patients seen between 1998 and 2007 (698 patients seen between 1998 and 2003, and 324 patients seen between 2004 and 2007).

Measurements on 73 medical attributes at the onset of hepatic encephalopathy¹⁴ and 5 days later, were collected from 371 of the 698 patients seen between 1998 and 2003.

Two decision trees were built. The first was used to predict (using 5 attributes) the outcome of the patient at the onset of hepatic encephalopathy. The second decision tree was used to predict (using 7 attributes) the outcome at 5 days after the onset of grade II or more severe hepatic encephalopathy. The decision trees were validated using data from 160 of the 324 patients seen between 2004 and 2007. Decision tree performance is shown in Table 4.

	Decision Tree I Outcome at the onset	Decision Tree II Outcome at 5 days
Accuracy (patients 1998-2003)	79.0%	83.6%
Accuracy (patients 2004-2007)	77.6%	82.6%

Table 4: Nakayama et al’s decision tree performance metrics

NOTE... ↗

The performance of a decision tree is often measured in terms of three characteristics:

- *Accuracy* - The percentage of cases correctly classified.
- *Sensitivity* - The percentage of cases correctly classified as belonging to class A among all observations known to belong to class A.
- *Specificity* - The percentage of cases correctly classified as belonging to class B among all observations known to belong to class B.

Traffic Accidents

de Oña et al¹⁵ investigate the use of decision trees for analyzing road accidents on rural highways in the province of Granada, Spain. Regression-type generalized linear models, Logit models and Probit models have been the techniques most commonly used to conduct such analyses¹⁶.

Three decision tree models are developed (CART, ID3 and C4.5) using data collected from 2003 to 2009. Nineteen independent variables, reported in Table 5, are used to build the decision trees.

Accident type	Age	Atmospheric factors
Safety barriers	Cause	Day of week
Lane width	Lighting	Month
Number of injuries	Number of Occupants	Paved shoulder
Pavement width	Pavement markings	Gender
Shoulder type	Sight distance	Time
Vehicle type		

Table 5: Variables used from police accident reports by de Oña et al

The accuracy results of their analysis are shown in Table 6. Overall, the decision trees showed modest improvement over chance.

CART	C4.5	ID3
55.87	54.16	52.72

Table 6: Accuracy results (percentage) reported by de Oña et al

☛ PRACTITIONER TIP ☛

Even though de Oña et al tested three different decision tree algorithms (CART, ID3 and C4.5), their results led to a very modest improvement over chance. This will also happen to you on very many occasions. Rather than clinging on to a technique (because it is the latest technique or the one you happen to be most familiar with), the professional data scientist seeks out and tests alternative methods. In this text you have over ninety of the best applied modeling techniques at your fingertips. If decision trees don't "cut it" try something else!

Electrical Power Losses

A non-technical loss (NTL) is defined by electrical power companies as any consumed electricity which is not billed. This could be because of measurement equipment failure or fraud. Traditionally power utilities have monitored NTL by making in situ inspections of equipment, especially for those customers that have very high or close to zero levels of energy consumption. Monedero et al¹⁷ develop a CART decision tree to enhance the detection rate of NTL

A sample on 38,575 customer accounts was collected over two years in Catalonia, Spain. For each customer various indicators were measured¹⁸.

The best decision tree had a depth of 5 with the terminal node identifying customers with the highest likelihood of NTL. A total of 176 customers were identified by the model. This number was greater than the expected total of 85, and too many for the utility company to inspect in situ. The researchers therefore merge their results with a Bayesian network model and thereby reduce the estimated number to 64.

This example illustrates the important point that the predictive model is just one aspect that goes into real world decisions. Oftentimes a model will be developed but deemed impracticable by the end user.

NOTE... ↗

Cross-validation refers to a technique used to allow for the training and testing of inductive models. Williams et al used a leave-one-out cross-validation. Leave-one-out cross-validation involves taking out one observation from your sample and training the model with the rest. The predictor just trained is applied to the excluded observation. One of two possibilities will occur: the predictor is correct on the previously unseen control, or not. The removed observation is then returned, and the next observation is removed, and again training and testing are done. This process is repeated for all observations. When this is completed, the results are used to calculate the accuracy of the model, often measured in terms of sensitivity and specificity.

Tracking Tetrahymena Pyriformis Cells

Tracking and matching individual biological cells in real time is a challenging task. Since decision trees provide an excellent tool for real time and rapid decision making they have potential in this area. Wang et al¹⁹ consider the issue of real time tracking and classification of Tetrahymena Pyriformis Cells. The issue is whether a cell in the current video frame is the same cell in the previous video frame.

A 23-dimensional feature vector is developed and whether two regions in different frames represent the same cell is manually determined. The training set consisted of 1000 frames from 2 videos. The videos were captured at 8 frames per second with each frame an 8-bit gray level image. The researchers develop two decision trees, the first (T1) trained using the feature vector and manual classification and the second trained (T2) using a truncated set of features. The error rates for each tree by tree depth are reported in Table 7. Notice that in this case T1 substantially outperforms T2, indicating the importance of using the full feature set.

Tree Depth	T1	T2
5	1.48%	14.02%
8	1.37%	12.49%
10	1.55%	13.61%

Table 7: Error rates by tree depth for T1 & T2 reported by Wang et al.

• PRACTITIONER TIP •

Cross-validation is often used to prevent over-fitting a model to the data. In n-fold cross-validation, we first divide the training set into n subsets of equal size. Sequentially one subset is tested using the classifier trained on the remaining (n-1) subsets. Thus, each instance of the whole training set is predicted once. The advantage of this method over a random selection of training samples is that all observations are used for either training (n times) or evaluation (once). Cross-validation accuracy is measured as the percentage of data that are correctly classified.

Classification Trees

Technique 1

Classification Tree

A classification tree can be built using the package `tree` with the `tree` function:

```
tree(z ~., data, split)
```

Key parameters include `split` which controls whether deviance or gini are used as the splitting criteria, `z` the data-frame of classes; `data` the data set of attributes with which you wish to build the tree.

☞ PRACTITIONER TIP ☞

To obtain information on which version of R you are running, loaded packages and other information use:

```
> sessionInfo()
```

Step 1: Load Required Packages

We build our classification tree using the `Vehicle` data frame contained in the `mlbench` package:

```
> library(tree)
> library(mlbench)
> data(Vehicle)
```

NOTE... 📈

The `Vehicle` data set²⁰ was collected to classify a "Corgie" vehicle silhouette as one of four types (double decker bus, Chevrolet van, Saab 9000 and Opel Manta 400). The data frame contains 846 observations on 18 numerical features extracted from the silhouettes and one nominal variable defining the class of the objects (see Table 8).

You can access the features directly using Table 8 as a reference. For example, a summary of the `Comp` and `Circ` features is obtained by typing:

```
> summary(Vehicle[1])
  Comp
Min.    : 73.00
1st Qu.: 87.00
Median  : 93.00
Mean    : 93.68
3rd Qu.:100.00
Max.    :119.00

> summary(Vehicle[2])
  Circ
Min.    :33.00
1st Qu.:40.00
Median  :44.00
Mean    :44.86
3rd Qu.:49.00
Max.    :59.00
```

Step 2: Prepare Data & Tweak Parameters

We use 500 of the 846 observations to create a randomly selected training sample. We use the observations associated with `train` to build the classification tree.

```
> set.seed(107)
> N=nrow(Vehicle)
> train <- sample(1:N, 500, FALSE)
```

Data-frame Index	R name	Description
1	Comp	Compactness
2	Circ	Circularity
3	D.Circ	Distance Circularity
4	Rad.Ra	Radius ratio
5	Pr.Axis.Ra	pr.axis aspect ratio
6	Max.L.Ra	max.length aspect ratio
7	Scat.Ra	scatter ratio
8	Elong	elongatedness
9	Pr.Axis.Rect	pr.axis rectangularity
10	Max.L.Rect	max.length rectangularity
11	Sc.Var.Maxis	scaled variance along major axis
12	Sc.Var.maxis	scaled variance along minor axis
13	Ra.Gyr	scaled radius of gyration
14	Skew.Maxis	skewness about major axis
15	Skew.maxis	skewness about minor axis
16	Kurt.maxis	kurtosis about minor axis
17	Kurt.Maxis	kurtosis about major axis
18	Holl.Ra	hollows ratio
19	Class type	bus, opel, saab, van

Table 8: Attributes and Class labels for vehicle silhouettes data set

Step 3: Estimate the Decision Tree

Now we are ready to build the decision tree using the training sample. This is achieved by entering:

```
> fit<- tree(Class ~., data = Vehicle[train,], split  
= "deviance")
```

We use deviance as the splitting criteria, a common alternative is to use `split="gini"`. You may be surprised by just how quickly R builds the tree!

☛ PRACTITIONER TIP ☛

It is important to remember that the response variable is a factor. This actually trips up quite a few users who have numeric categories and forget to convert their response variable using `factor()`. To see the levels of the response variable type:

```
> attributes(Vehicle$Class)  
$levels  
[1] "bus"   "opel"  "saab"  "van"  
  
$class  
[1] "factor"
```

For `Vehicle` each level is associated with a different vehicle type ("bus" "opel" "saab" "van").

To see details of the fitted tree type.

```
> fit  
  
node), split, n, deviance, yval, (yprob)  
* denotes terminal node  
  
1) root 500 1386.000 saab ( 0.248000 0.254000  
    0.258000 0.240000 )  
2) Elong < 41.5 229 489.100 opel ( 0.222707  
    0.410480 0.366812 0.000000 )
```

```
14) Skew.Maxis < 64.5 7      9.561 van (
    0.000000 0.000000 0.428571 0.571429 ) *
15) Skew.Maxis > 64.5 64     10.300 van (
    0.015625 0.000000 0.000000 0.984375 ) *
```

At each branch of the tree (after `root`) we see in order:

1. The branch number (e.g. in this case 1,2,14 and 15);
2. the split (e.g. `Elong < 41.5`);
3. the number of samples going along that split (e.g. 229);
4. the deviance associated with that split (e.g. 489.1);
5. the predicted class (e.g. `opel`);
6. the associated probabilities (e.g. (0.222707 0.410480 0.366812 0.000000));
7. and for a terminal node (or leaf), the symbol "*".

☛ PRACTITIONER TIP ☛

If the minimum deviance occurs with a tree with 1 node, then your model is at best no better than random. It is even possible that it may be worse!

A summary of the tree can also be obtained.

```
> summary(fit)

Classification tree:
tree(formula = Class ~ ., data = Vehicle[train, ],
     split = "deviance")
Variables actually used in tree construction:
[1] "Elong"          "Max.L.Ra"        "Comp"           "
     Pr.Axis.Ra"      "Sc.Var.maxis"
[6] "Max.L.Rect"     "D.Circ"         "Skew.maxis"      "
     Circ"            "Kurt.Maxis"
```

```
[11] "Skew.Maxis"  
Number of terminal nodes: 15  
Residual mean deviance: 0.9381 = 455 / 485  
Misclassification error rate: 0.232 = 116 / 500
```

Notice that `summary(fit)` shows:

1. The type of tree, in this case a Classification tree;
2. the formula used to fit the tree;
3. the variables used to fit the tree;
4. the number of terminal nodes in this case 15;
5. the residual mean deviance - 0.9381;
6. the misclassification error rate 0.232 or 23.2%.

We plot the tree, see Figure 1.1.

```
> plot(fit); text(fit)
```

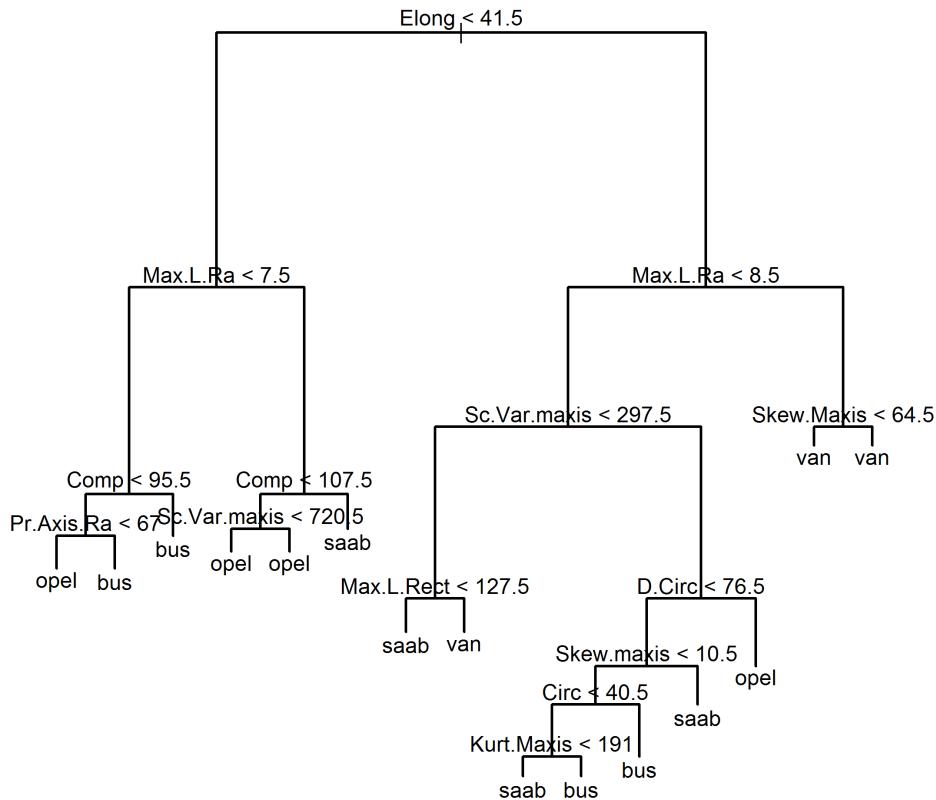


Figure 1.1: Fitted Decision Tree

☞ PRACTITIONER TIP ☚

The height of the vertical lines in Figure 1.1 is proportional to the reduction in deviance. The longer the line the larger the reduction. This allows you to identify the important sections immediately. If you wish to plot the model using uniform lengths use `plot(fit,type="uniform")`.

Step 4: Assess Model

Unfortunately, classification trees have a tendency to over-fit the data. One approach to reduce this risk is to use cross-validation. For each hold out sample we fit the model and note at what level the tree gives the best results (using deviance or the misclassification rate). Then we hold out a different sample and repeat. This can be carried out using the `cv.tree()`function. We use a leave-one-out cross-validation using the misclassification rate and deviance (`FUN=prune.misclass`, followed by `FUN=prune.tree`).

NOTE... ↗

Textbooks and academics used to spend a inordinate amount of time on the subject of when to stop splitting a tree and also pruning techniques. This is indeed an important aspect to consider when building a single tree because if the tree is too large it will tend to over fit the data. If the tree is too small, it might misclassification important characteristics of the relationship between the covariates and the outcome. In actual practice, I do not spend a great deal of time on deciding when to stop splitting a tree or even pruning. This is partly becuase:

1. A single tree is generally only of interest to gain insight about the data if it can be easily interpreted. The default settings in R decision tree functions often are sufficient to create such trees.
2. For use in “pure” prediction activites random forests (see page272) have largely replaced individual decision trees becuase they often produce more acurate predictive models.

The results are plotted out side by side in Figure 1.2. The jagged lines shows where the minimum deviance / misclassification occurred with the cross-validated tree. Since the cross validated misclassification and deviance both reach their minimum close to the number of branches in the original fitted tree there is little to be gained from pruning this tree.

```
> fitM.cv <- cv.tree(fit,K=346,FUN=prune.misclass)
```

```
> fitP.cv <- cv.tree(fit,K=346,FUN=prune.tree)
```

```
> par(mfrow = c(1, 2))
> plot(fitM.cv)
> plot(fitP.cv)
```

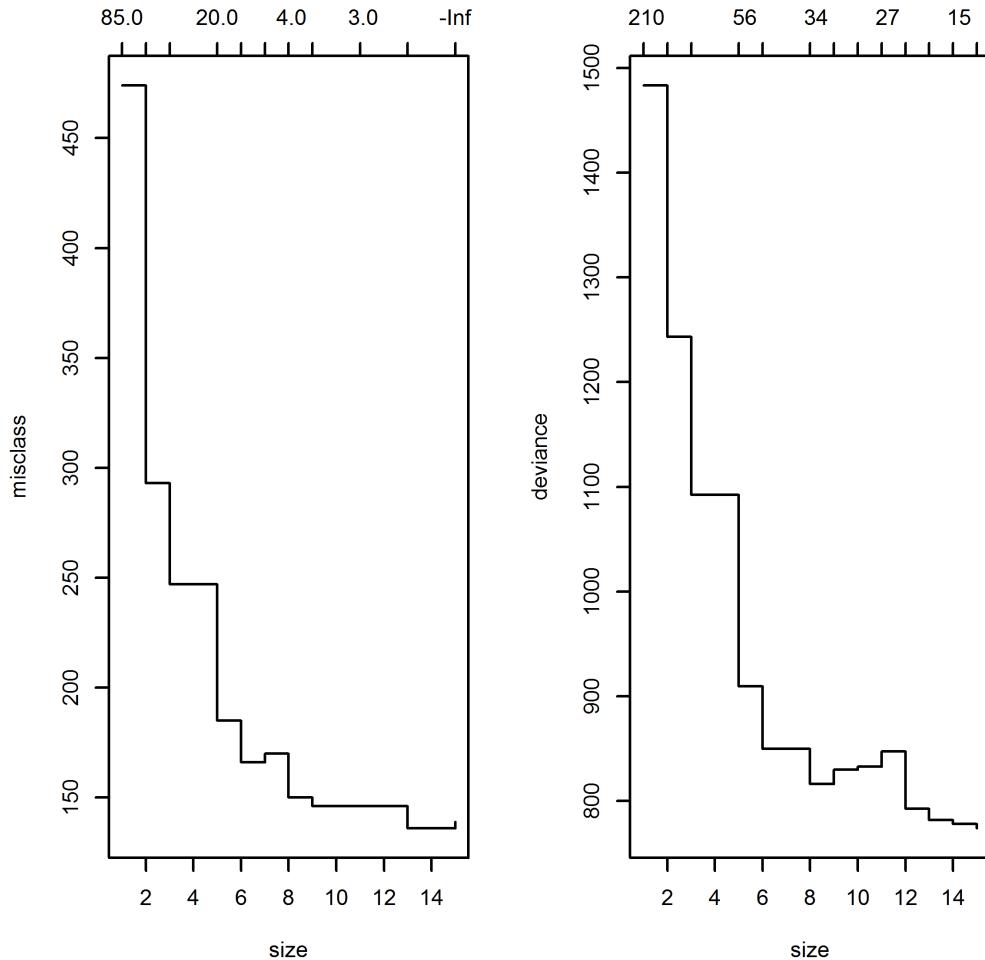


Figure 1.2: Cross Validation results on Vehicle using misclassification and Deviance

Step 5: Make Predictions

We use the validation data set and the fitted decision tree to predict vehicle classes; then we display the confusion matrix and calculate the error rate of the fitted tree. Overall, the model has an error rate of 32%.

TECHNIQUE 1. CLASSIFICATION TREE

```
> pred<-predict(fit,newdata=Vehicle[-train,])  
  
> pred.class <- colnames(pred)[max.col(pred, ties.method = c("random"))]  
  
> table(Vehicle$Class[-train],pred.class,dnn=c( "Observed Class","Predicted Class" ))  
          Predicted Class  
Observed Class bus opel saab van  
    bus      86     1     3     4  
    opel      1    55    20     9  
    saab      4    55    23     6  
    van       2     2     5    70  
  
> error_rate = (1-sum(pred.class== Vehicle$Class [-train]))/346  
> round(error_rate ,3)  
[1] 0.324
```

Technique 2

C5.0 Classification Tree

The C5.0 algorithm²¹ is based on the concepts of entropy, the measure of disorder in a sample, and the information gain of each attribute. Information gain is a measure of the effectiveness of an attribute in reducing the amount of entropy in the sample.

It begins by calculating the entropy of a data sample. The next step is to calculate the information gain for each attribute. This is the expected reduction in entropy by partitioning the data set on the given attribute. From the set of information gain values the best attributes for partitioning the data set are chosen and the decision tree is built.

A C5.0 Classification Tree can be built using the package **C50** with the **C5.0** function:

```
C5.0(z ~., data)
```

Key parameters include **z** the data-frame of classes; **data** the data set of attributes with which you wish to build the tree.

Step 1: Load Required Packages

We build our classification tree using the **Vehicle** data frame contained in the **mlbench** package:

```
> library(C50)
> library(mlbench)
> data(Vehicle)
```

Step 2: Prepare Data & Tweak Parameters

We use 500 of the 846 observations to create a randomly selected training sample. We will use the observations associated with `train` to build the classification tree.

```
> set.seed(107)
> N=nrow(Vehicle)
> train <- sample(1:N, 500, FALSE)
```

Step 3: Estimate and Assess the Decision Tree

Now we are ready to build the decision tree using the training sample. This is achieved by entering:

```
> fit<- C5.0(Class ~., data = Vehicle[train,] )
```

Next we assess variable importance using the `C5imp` function.

```
> C5imp(fit)
```

	Overall
Max.L.Ra	100.0
Elong	100.0
Comp	50.2
Circ	46.2
Skew.maxis	45.2
Scat.Ra	40.0
Max.L.Rect	29.2
Ra.Gyr	23.4
D.Circ	23.2
Skew.Maxis	20.2
Pr.Axis.Rect	17.0
Kurt.maxis	12.2
Pr.Axis.Ra	9.0
Rad.Ra	3.0
Holl.Ra	3.0
Sc.Var.maxis	0.0
Kurt.Maxis	0.0

We observe that `Max.L.Ra` and `Elong` are the two most influential attributes. The attributes `Sc.Var.maxis` and `Kurt.Maxis` are the least influential variables with an influence score of zero.

☛ PRACTITIONER TIP ☛

To assess the importance of attributes by split use:

```
> C5imp(fit, metric = "splits")
```

Step 4: Make Predictions

We use the validation data set and the fitted decision tree to predict vehicle classes; then we display the confusion matrix and calculate the error rate of the fitted tree. Overall, the model has an error rate of 27.5%.

```
> pred<-predict(fit,newdata=Vehicle[-train,],type =
  "class")

> table(Vehicle$Class[-train],pred,dnn=c( "Observed
  Class","Predicted Class" ))
      Predicted Class
Observed Class bus opel saab van
  bus     84    1    5    4
  opel     0   48   31    6
  saab     3   38   47    0
  van      2    1    4   72

> error_rate = (1-sum(pred== Vehicle$Class[-train])/
  346)
> round(error_rate,3)
[1] 0.275
```

◆ PRACTITIONER TIP ◆

To view the estimated probabilities of each class use:

```
> pred<-predict(fit,newdata=Vehicle[-train  
 ,],type = "prob")  
  
> head(round(pred,3))  
      bus    opel    saab    van  
5  0.018  0.004  0.004  0.974  
7  0.050  0.051  0.852  0.048  
10 0.011  0.228  0.750  0.010  
12 0.031  0.032  0.782  0.155  
15 0.916  0.028  0.029  0.027  
19 0.014  0.070  0.903  0.013
```

Technique 3

Conditional Inference Classification Tree

A conditional inference classification tree is a non-parametric regression tree embedding tree-structured regression models. It is essentially a decision tree but with extra information about the distribution of classes in the terminal nodes²². It can be built using the package `party` with the `ctree` function:

```
ctree(z ~ ., data, ...)
```

Key parameters include `z` the data-frame of classes; `data` the data set of attributes with which you wish to build the tree.

Step 1: Load Required Packages

We build our classification tree using the `Vehicle` data frame contained in the `mlbench` package:

```
> library ("party")
> library(mlbench)
> data(Vehicle)
```

Step 2 is outlined on page 23.

Step 3: Estimate and Assess the Decision Tree

Now we are ready to build the decision tree using the training sample. This is achieved by entering:

```
> fit<-ctree(Class ~ ., data = Vehicle[train,],
  controls =ctree_control(maxdepth = 2))
```

Notice we use `controls` with the `maxdepth` parameter to limit the depth of the tree to at most 2. Note if `maxdepth = 0`, no tree is fitted.

Next we plot the tree:

```
> plot(fit)
```

The resultant tree is shown in Figure 3.1. At each internal node a p-value is reported for the split. In this case they are all highly significant (less than 1%). The primary split takes place at `Elong ≤ 41`, `Elong > 41`. The four root nodes are labeled `Node 3`, `Node 4`, `Node 6` and `Node 7` with 62, 167, 88 and 183 observations respectively. Each of these leaf nodes also has a bar chart illustrating the proportion of the four vehicle types that fall into each class at that node.

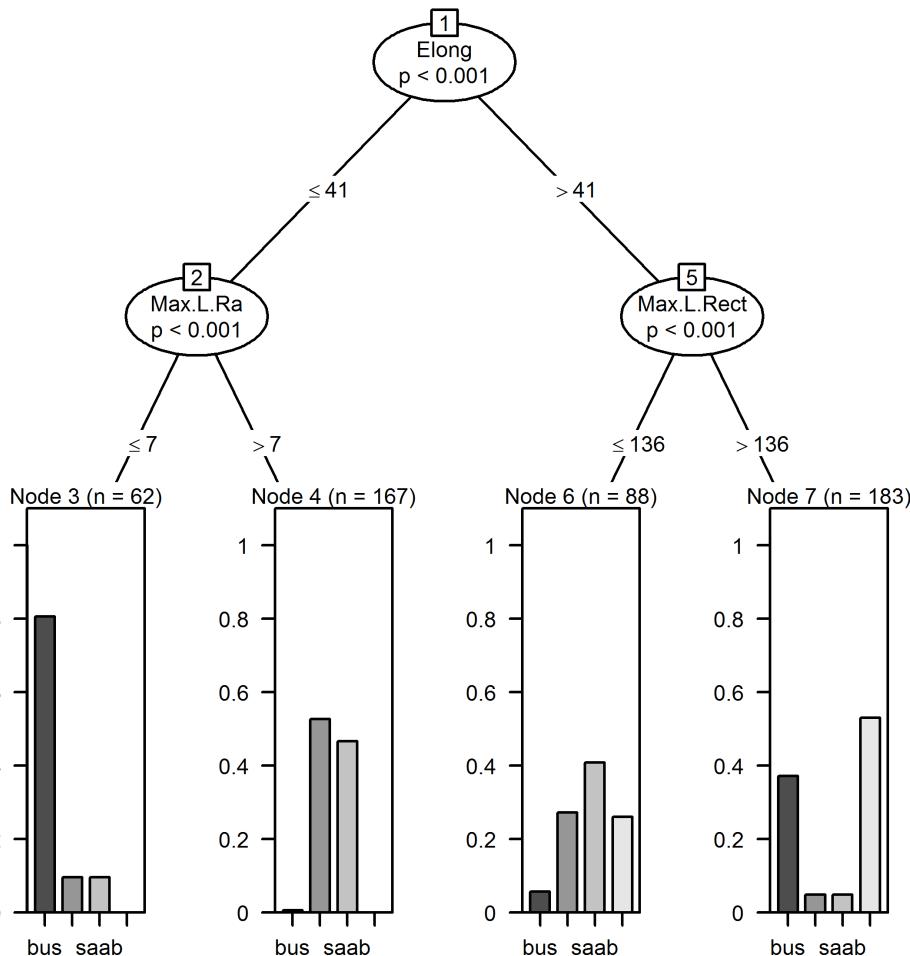


Figure 3.1: Conditional Inference Classification Tree for Vehicle with `maxdepth = 2`

Step 4: Make Predictions

We use the validation data set and the fitted decision tree to predict vehicle classes; we show the confusion matrix and calculate the error rate of `fit`. The misclassification rate is approximately 53% for this tree.

```
> pred<-predict(fit,newdata=Vehicle[-train,],type =
  "response")
```

```
> table(Vehicle$Class[-train], pred, dnn=c( "Observed
  Class", "Predicted Class" ))
      Predicted Class
Observed Class bus opel saab van
  bus       36    0   11   47
  opel      3    50   24    8
  saab      6    58   19    5
  van       0    0   21   58

> error_rate = (1 - sum(pred == Vehicle$Class[-train])) /
  346)

> round(error_rate, 3)
[1] 0.529
```

 **PRACTITIONER TIP** 

Set the parameter `type = "node"` to see which nodes the observations end up in; and `type = "prob"` to view the probabilities. For example to see the distribution for the validation sample type:

```
> pred<-predict(fit,newdata=Vehicle[-train
  ,],type = "node")
> table(pred)
pred
  3   4   6   7
  45 108 75 118
```

We see that 45 observations ended up in node 3 and 118 in node 7.

To assess the predictive power of `fit` we compare it against two other conditional inference classification trees - `fit3` which limits the maximum tree depth to 3 and `fitu` which estimates an unrestricted tree.

```
> fit3<-ctree(Class ~., data = Vehicle[train,],
  controls = ctree_control(maxdepth = 3))

> fitu<-ctree(Class ~., data = Vehicle[train,])
```

We use the validation data set and the fitted decision tree to predict vehicle classes.

```
> pred3<-predict(fit3,newdata=Vehicle[-train,],  
type = "response")
```

```
> predu<-predict(fitu,newdata=Vehicle[-train,],  
type = "response")
```

Next we calculate the error rate of each fitted tree.

```
> error_rate3 = (1-sum(pred3==  
Vehicle$Class[-train])/346)
```

```
> error_rateu = (1-sum(predu==  
Vehicle$Class[-train])/346)
```

```
> tree_1<-round(error_rate,3)  
> tree_2<-round(error_rate3,3)  
> tree_3<-round(error_rateu,3)
```

Finally, we calculate the misclassification error rate for each fitted tree.

```
> err<-cbind(tree_1,tree_2,tree_3)*100  
> row.names(err) <- "error(%)"
```

```
> err  
          tree_1  tree_2  tree_3  
error(%)    52.9    41.6     37
```

The unrestricted tree has the lowest misclassification error rate of 37%.

Technique 4

Evolutionary Classification Tree

NOTE... ↗

The recursive partitioning methods discussed in previous sections build the decision tree using a forward stepwise search where splits are chosen to maximize homogeneity at the next step only. Although this approach is known to be an efficient heuristic the results are only locally optimal. Evolutionary algorithm based trees search over the parameter space of trees using a global optimization method.

A evolutionary classification tree model can be estimated using the package `evtree` with the `evtree` function:

```
evtree(z ~., data, ...)
```

Key parameters include the response variable `Z` which contains the classes; and `data` the data set of attributes with which you wish to build the tree.

Step 1: Load Required Packages

We build our classification tree using the `Vehicle` data frame contained in the `mlbench` package:

```
> library ("evtree")
> library(mlbench)
> data(Vehicle)
```

Step 2: Prepare Data & Tweak Parameters

We use 500 of the 846 observations to create a randomly selected training sample. We will use the observations associated with `train` to build the classification tree.

```
> set.seed(107)
> N=nrow(Vehicle)
> train <- sample(1:N, 500, FALSE)
```

Step 3: Estimate and Assess the Decision Tree

Now we are ready to build the decision tree using the training sample. We use `evtree` to build the tree and `plot` to display the tree. We restrict the tree depth using `maxdepth=2`.

```
> fit<- evtree(Class ~., data = Vehicle[train,],
  control =evtree.control (maxdepth=2) )
> plot(fit)
```

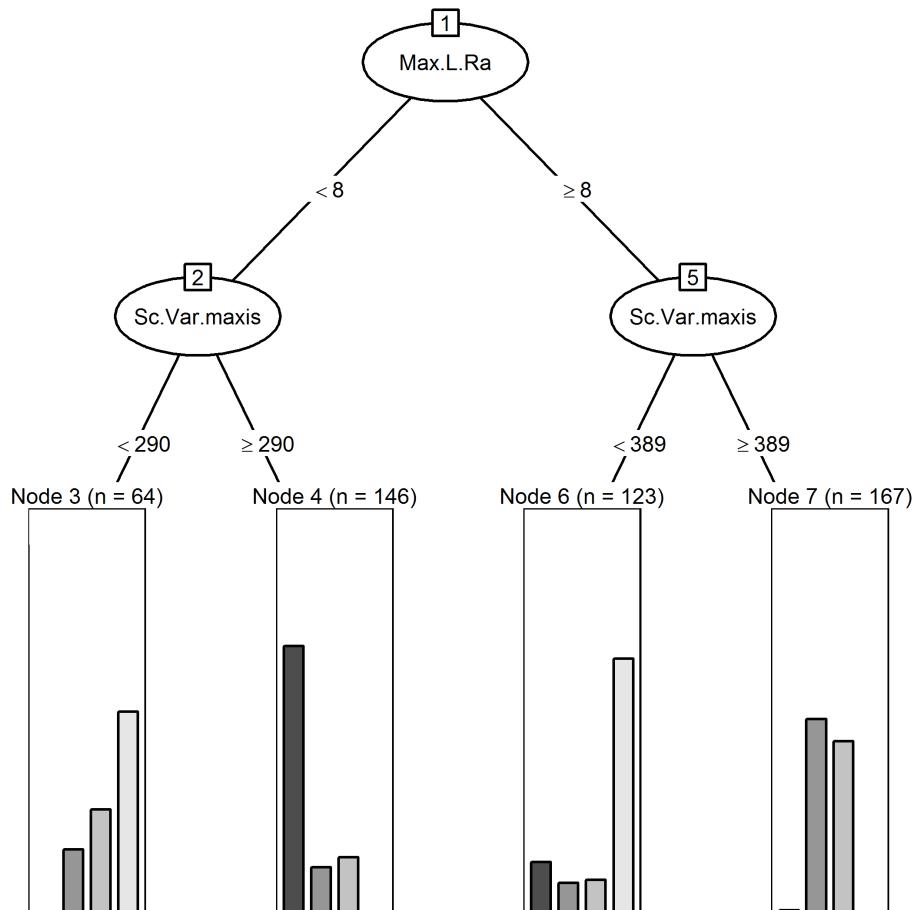


Figure 4.1: Fitted Evolutionary Classification Tree using Vehicle

The tree shown in Figure 4.1 visualizes the decision rules. It also shows, for the terminal nodes, the number of observations and their distribution amongst the classes. Let us take node 3 as an illustration. This node is reached by the rule `max.L.Ra <8` and `Sc.Var.maxis<290`. The node contains 64 observations. Similar details can be obtained by typing `fit`.

```
> fit
```

```
Model formula:  
Class ~ Comp + Circ + D.Circ + Rad.Ra + Pr.Axis.Ra +  
Max.L.Ra +
```

```
Scat.Ra + Elong + Pr.Axis.Rect + Max.L.Rect + Sc
.Var.Maxis +
Sc.Var.maxis + Ra.Gyr + Skew.Maxis + Skew.maxis
+ Kurt.maxis +
Kurt.Maxis + Holl.Ra

Fitted party:
[1] root
| [2] Max.L.Ra < 8
| | [3] Sc.Var.maxis < 290: van (n = 64, err =
45.3%)
| | [4] Sc.Var.maxis >= 290: bus (n = 146, err =
27.4%)
| [5] Max.L.Ra >= 8
| | [6] Sc.Var.maxis < 389: van (n = 123, err =
30.9%)
| | [7] Sc.Var.maxis >= 389: opel (n = 167, err
= 47.3%)

Number of inner nodes:      3
Number of terminal nodes: 4
```

Step 4: Make Predictions

We use the validation data set and the fitted decision tree to predict vehicle classes; then we display the confusion matrix and calculate the error rate of the fitted tree. Overall, the model has an error rate of 39%.

```
> pred<-predict(fit,newdata=Vehicle[-train,],type =
"response")

> table(Vehicle$Class[-train],pred,dnn=c( "Observed
   Class","Predicted Class" ))
           Predicted Class
Observed Class bus opel saab van
          bus    85     0     0    9
          opel   14    48     0   23
          saab   18    57     0   13
          van     0     1     0   78
```

```
> error_rate = (1-sum(pred== Vehicle$Class[-train]))/  
346)  
  
> round(error_rate,3)  
[1] 0.39
```

☛ PRACTITIONER TIP ☛

When using `predict` you can specify any of the following via `type = "response"`, `"prob"`, `"quantile"`, `"density"` or `"node"`. For example to view the estimated probabilities of each class use:

```
> pred<-predict(fit,newdata=Vehicle[-train  
 ,],type = "prob")
```

To see the distribution across nodes you would enter:

```
> pred<-predict(fit,newdata=Vehicle[-train  
 ,],type = "node")
```

Finally, we estimate a unrestricted model, use the validation data set to predict vehicle classes, display the confusion matrix and calculate the error rate of the unrestricted fitted tree. In this case the misclassification error rate is lower at 34.7%.

```
> fit<- evtree(Class ~., data = Vehicle[train,], )  
  
> pred<-predict(fit,newdata=Vehicle[-train,],type =  
"response")  
  
> table(Vehicle$Class[-train],pred,dnn=c( "Observed  
Class","Predicted Class" ))  
          Predicted Class  
Observed Class bus opel saab van  
    bus     72      3     15     4  
    opel     4     30     41    10  
    saab     6     23     55     4  
    van      3      1      6    69  
> error_rate = (1-sum(pred== Vehicle$Class[-train]))/  
346)
```

```
> round(error_rate,3)
[1] 0.347
```

Technique 5

Oblique Classification Tree

NOTE... ↗

A common question is what is different about oblique trees? Here is the answer in a nutshell. For a set of attributes $\{X_1, \dots, X_k\}$ the standard classification tree produces binary partitioned trees by considering axis-parallel splits over continuous attributes (i.e. grown using tests of the form $X_i < C$ versus $X_i \geq C$). This is the most widely used approach to tree-growth. Oblique trees are grown using oblique splits (i.e. grown using tests of the form $\sum_{i=1}^k \alpha_i X_i < C$ versus $\sum_{i=1}^k \alpha_i X_i \geq C$). So for axis-parallel splits a single attribute is used; for oblique splits a weighted combination of attributes is used.

An oblique classification tree model can be estimated using the package `oblique.tree` with the `oblique.tree` function:

```
oblique.tree(z ~., data, oblique.splits = "only",
control = tree.control(...), split.impurity,...)
```

Key parameters include the response variable Z which contains the classes; and `data` the data set of attributes with which you wish to build the tree, `control` with takes arguments from using `tree.control` from the `tree` package, and `split.impurity` which controls the splitting criterion used and takes values "deviance" or "gini".

Step 1: Load Required Packages

We build our classification tree using the `Vehicle` data frame contained in the `mlbench` package:

```
> library ("oblique.tree")
> library(mlbench)
> data(Vehicle)
```

Step 2: Prepare Data & Tweak Parameters

We use 500 of the 846 observations to create a randomly selected training sample. Three attributes are used to build the tree (`Max.L.Ra` , `Sc.Var.maxis` and `Elong`).

```
> set.seed(107)
> N=nrow(Vehicle)
> train <- sample(1:N, 500, FALSE)
> f<-Class ~Max.L.Ra +Sc.Var.maxis +Elong
```

Step 3: Estimate and Assess the Decision Tree

Before estimating the tree we tweak the following:

1. Only allow oblique splits (`oblique.splits = "only"`) ;
2. Use `tree.control` to indicate the number of observations (`nobs=500`) and set the minimum number of observations in a node to 60 (`mincut=60`).

The tree is estimated as follows:

```
>fit<-oblique.tree(f,data=Vehicle[train,], oblique.
  splits = "only", control = tree.control(nobs=500,
  mincut=60) ,split.impurity="deviance")
```

◆ PRACTITIONER TIP ◆

The type of split is indicated by the `oblique.splits` argument. For our analysis we use `oblique.splits = "only"` to grow trees that only use oblique splits. Use `oblique.splits = "on"` to grow trees that use both oblique and axis parallel splits; and `oblique.splits = "off"` to grow traditional classification trees (only use axis-parallel splits).

Details of the tree can be visualized using a combination of `plot` and `text`, see Figure 5.1.

```
> plot(fit);text(fit)
```

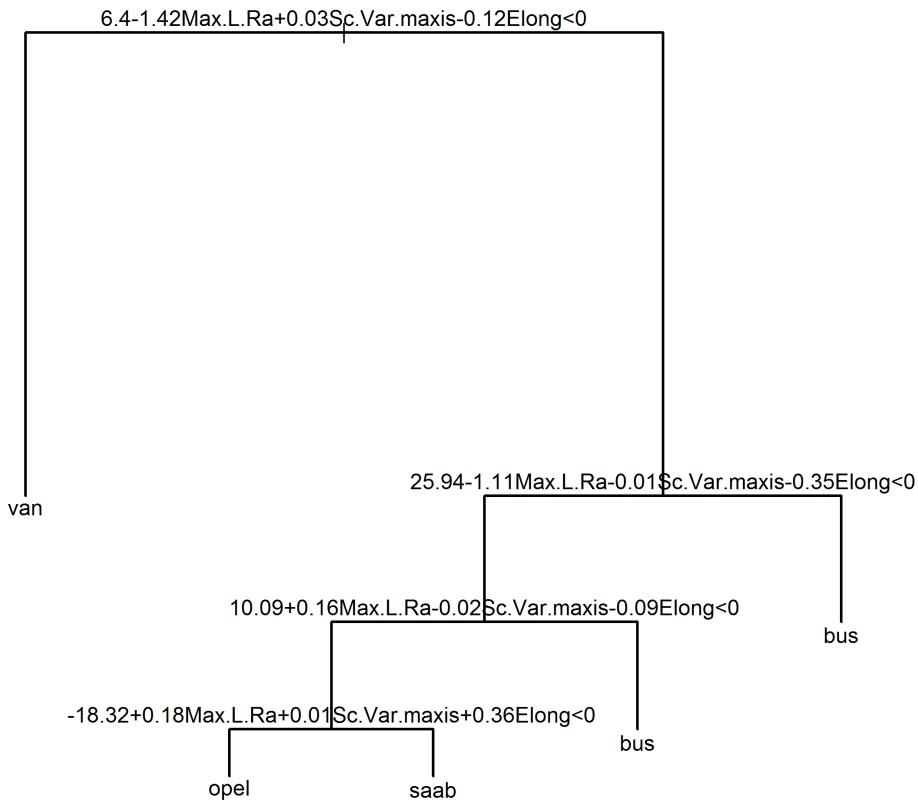


Figure 5.1: Fitted Oblique tree using a subset of `Vehicle` attributes

The tree visualizes the decision rules. The first split occurs where:

$$6.4 - 1.42\text{Max.L.Ra} + 0.03\text{Sc.Var.maxis} - 0.12\text{Elong} < 0$$

If the above holds it leads directly to the leaf node indicating class type = van.

Full details of the tree can be obtained by typing `fit` whilst `summary` gives an overview of the fitted tree:

```
> summary(fit)
```

```
Classification tree:
```

```
oblique.tree(formula = f, data = Vehicle[train, ],
  control = tree.control(nobs = 500,
    mincut = 60), split.impurity = "deviance",
    oblique.splits = "only")
Variables actually used in tree construction:
[1] ""
Number of terminal nodes: 5
Residual mean deviance: 1.766 = 874.2 / 495
Misclassification error rate: 0 = 0 / 500
```

Step 4: Make Predictions

We use the validation data set and the fitted decision tree to predict vehicle classes; then we display the confusion matrix and calculate the error rate of the fitted tree. Overall, the model has an error rate of 39.9%.

```
> pred<-predict(fit,newdata=Vehicle[-train,],type =
  "class")
>
> table(Vehicle$Class[-train],pred,dnn=c( "Observed
  Class","Predicted Class" ))
            Predicted Class
Observed Class bus opel saab van
  bus      77     0    13    4
  opel     17    37    20   11
  saab     23    34    27    4
  van       8     0     4   67
> error_rate = (1-sum(pred== Vehicle$Class[-train]))/
  346)
>
> round(error_rate,3)
[1] 0.399
```

Technique 6

Logistic Model Based Recursive Partitioning

A model based recursive partitioning logistic regression tree can be estimated using the package `party` with the `mob` function:

```
ctree(z ~x+y+z|a+b+c, data, model = glinearModel,  
      family = binomial()...)
```

Key parameters include the binary response variable `Z`; the conditioning covariates `x,y` and `z` and the tree partitioning covariates `a, b` and `c`.

Step 1: Load Required Packages

We build the decision tree using the data frame `PimaIndiansDiabetes2` contained in the `mlbench` package.

```
> library ("party")  
> data("PimaIndiansDiabetes2", package="mlbench")
```

NOTE... ↗

The `PimaIndiansDiabetes2` data set was collected by the National Institute of Diabetes and Digestive and Kidney Diseases²³. It contains 768 observations on 9 variables measured on females at least 21 years old of Pima Indian heritage. Table 9 contains a description of each of the variables.

Name	Description
pregnant	Number of times pregnant
glucose	Plasma glucose concentration (glucose tolerance test)
pressure	Diastolic blood pressure (mm Hg)
triceps	Triceps skin fold thickness (mm)
insulin	2-Hour serum insulin (mu U/ml)
mass	Body mass index
pedigree	Diabetes pedigree function
age	Age (years)
diabetes	test for diabetes - Class variable (neg / pos)

Table 9: Response and independent variables in `PimaIndiansDiabetes2` data frame

Step 2: Prepare Data & Tweak Parameters

For our analysis we use 600 of the 768 observations to train the model. The response variable is `diabetes` and we use `mass` and `pedigree` as logistic regression conditioning variables with the remaining six variables (`glucose`, `pregnant`, `pressure`, `triceps`, `insulin` and `age`) being used as the partitioning variables. The model is stored in `f`.

```
> set.seed(898)
> n=nrow(PimaIndiansDiabetes2)
> train <- sample(1:n, 600, FALSE)
> f<-diabetes ~ mass+ pedigree| glucose +pregnant +
  pressure + triceps + insulin + age
```

NOTE... ↗

The `PimaIndiansDiabetes2` data-set has a large number of misclassified values (recorded as `NA`) particularly for the attributes of `insulin` and `triceps`. In traditional statistical analysis these values would have to be removed or their values interpolated. However, ignoring misclassified values or treating them as another category is often inefficient. A more efficient of the available information used by many decision tree algorithms is to ignore the misclassified data point in the evaluation of a split but distribute them to child nodes using a given rule. For example, the rule might:

1. Distribute misclassified values to the node which has the largest number of instances.
2. Distribute to all child nodes with diminished weights, proportional with the number of instances from each child node.
3. Randomly distribute to one single child node.
4. Create surrogates attributes which closely resemble the test attributes and use them to send misclassified values to child nodes.

Step 3: Estimate & Interpret Decision Tree

We estimate the model using the function `mob` and then use the `plot` function to visualize the tree as shown in Figure 6.1. Since the response variable `diabetes` is binary and `mass` and `pedigree` are numeric a spinogram is used for visualization. The plots in the leaves give spinograms for diabetes versus mass (upper panel) and pedigree (lower panel).

```
> fit <- mob(f, data = PimaIndiansDiabetes2[train,],  
  model = glinearModel, family = binomial())  
> plot(fit)
```

• PRACTITIONER TIP •

As an alternative to using spinograms you can also plot the cumulative density function using the argument `tp_args = list(cdplot = TRUE)` in the `plot` function. In the example in this section you would type:

```
> plot(fit, tp_args = list(cdplot = TRUE))
```

You can also specify the smoothing bandwidth using the `bw` argument. For example:

```
> plot(fmPID, tp_args = list(cdplot = TRUE  
, bw = 15))
```

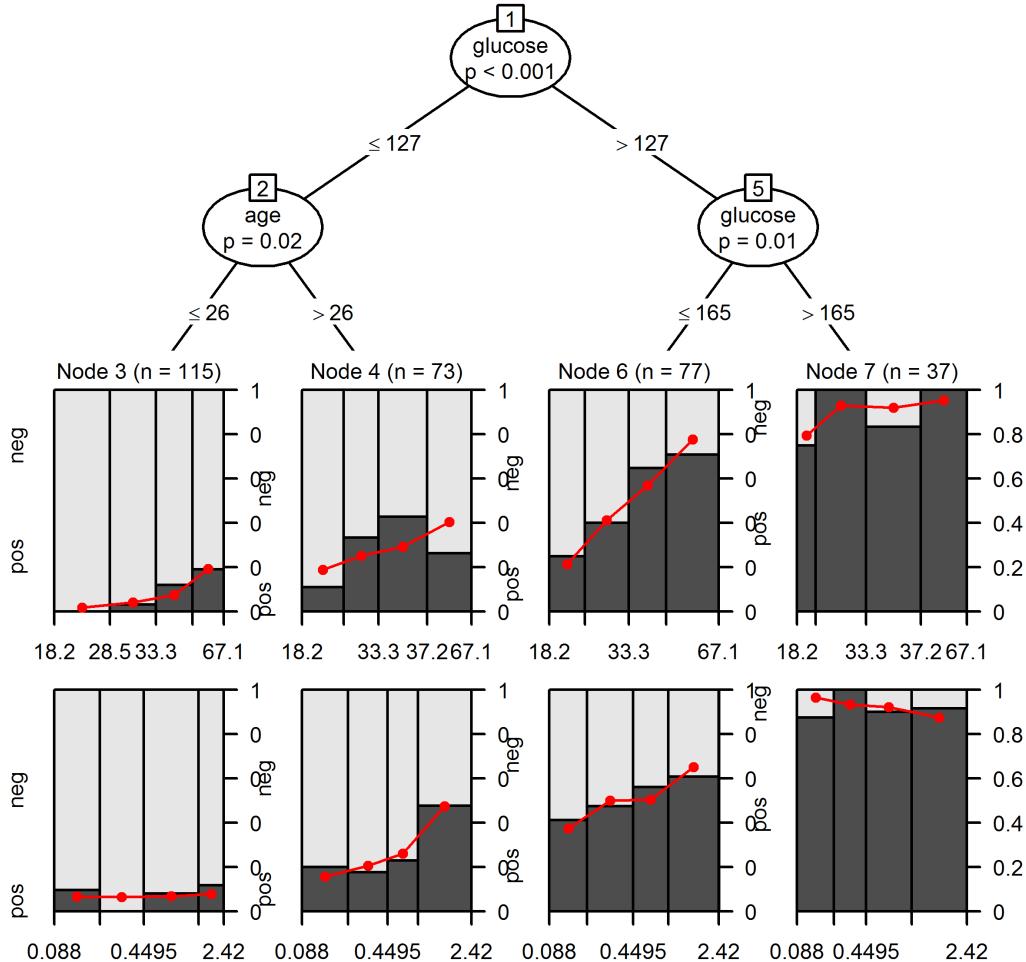


Figure 6.1: Logistic-regression-based tree for the Pima Indians diabetes data.

The fitted lines are the mean predicted probabilities in each group. The decision tree distinguishes four different groups of women:

- Node 3: Women with low glucose and 26 years or younger have on average a low risk of diabetes, however this increases with `mass`, but decreases slightly with `pedigree`.
- Node 4: Women with low glucose and older than 26 years have on average a moderate risk of diabetes which increases with `mass` and `pedigree`.
- Node 5: Women with glucose in the range 127 to 165 have on average

a moderate to high risk of diabetes which increases with `mass` and `pedigree`.

- Node 7: Women with glucose greater than 165 have on average a high risk of diabetes which increases with `mass` and decreases with `pedigree`.

The same interpretation can also be drawn from the coefficient estimates obtained using the `coef` function:

```
> round(coef(fit), 3)

(Intercept) mass pedigree
3      -7.263 0.143   -0.680
4      -4.887 0.076    2.495
6      -5.711 0.149    1.365
7      -3.216 0.225   -2.193
```

When comparing models it can be useful to have the value of the log likelihood function or the Akaike information criterion. These can be obtained by:

```
> logLik(fit)
'log Lik.' -115.4345 (df=15)

> AIC(fit)
[1] 260.8691
```

Step 5: Make Predictions

We use the function `predict` to fit calculated the fitted values using the validation sample and show the confusion table. Then we calculate the misclassification error which returns a value of 26.2%.

```
> pred<-predict(fit, newdata=PimaIndiansDiabetes2[-train,])

> thresh <- 0.5
> predFac <- cut(pred, breaks=c(-Inf, thresh, Inf),
  labels=c("neg", "pos"))

> tb<-table(PimaIndiansDiabetes2$diabetes[-train],
  predFac, dnn=c("actual", "predicted"))

> tb
```

```
predicted
actual  neg  pos
      neg    92   17
      pos    26   29

> error <- 1-(sum(diag(tb))/sum(tb))
> round(error ,3)*100
[1] 26.2
```

☛ PRACTITIONER TIP ☛

Re-run the analysis in this section omitting the attributes of `insulin` and `triceps` and using the `na.omit` method to remove the any remaining misclassified values. Here is some sample code to get you started:

```
temp<-(PimaIndiansDiabetes2)
temp$insulin  <- NULL
temp$triceps <- NULL
temp<-na.omit(temp)
```

What do you notice about the resultant decision tree?

Technique 7

Probit Model Based Recursive Partitioning

A model based recursive partitioning probit regression tree can be estimated using the package `party` with the `mob` function:

```
ctree(z ~x+y+z|a+b+c, data, model = glinearModel,  
      family = binomial(link = "probit"), ...)
```

Key parameters include the binary response variable Z; the conditioning covariates x, y and z and the tree partitioning covariates a, b and c.

Step 1 and step 2 are discussed beginning on page 52.

Step 3: Estimate & Interpret Decision Tree

We estimate the model using the function `mob` and display the coefficients at the leaf nodes using `coef`.

```
> fit <- mob(f, data = PimaIndiansDiabetes2[train,],  
    model = glinearModel, family = binomial(link = "  
    probit"))  
  
> round(coef(fit),3)  
(Intercept) mass pedigree  
3 -4.070 0.078 -0.222  
4 -2.932 0.046 1.474  
6 -3.416 0.089 0.814  
7 -1.174 0.100 -1.003
```

The estimated decision tree is similar to that shown in Figure 6.1 and the interpretation of the coefficients is given on page 56.

Step 5: Make Predictions

For comparison with the logistic regression discussed on page 52 we report the value of the log likelihood function or the Akaike information criterion. Since these values are very close to those obtained by the logistic regression we should expect similar predictive performance.

```
> logLik(fit)
'log Lik.' -115.4277 (df=15)

> AIC(fit)
[1] 260.8555
```

We use the function `predict` with the validation sample and show the confusion table. Then we calculate the misclassification error which returns a value of 26.2%. This is exactly the same error rate observed by the logistic regression model.

```
> pred<-predict(fit, newdata=PimaIndiansDiabetes2[-train,])

> thresh  <- 0.5
> predFac <- cut(pred, breaks=c(-Inf, thresh, Inf),
+                   labels=c("neg", "pos"))

> tb<-table(PimaIndiansDiabetes2$diabetes[-train],
+             predFac, dnn=c("actual", "predicted"))

> tb
      predicted
actual   neg  pos
  neg    92  17
  pos    26  29

> error <- 1-(sum(diag(tb))/sum(tb))
> round(error,3)*100
[1] 26.2
```

Regression Trees for Continuous Response Variables

Technique 8

Regression Tree

A regression tree can be built using the package **tree** with the **tree** function:

```
tree(z ~., data, split)
```

Key parameters include **split** which controls whether deviance or gini are used as the splitting criteria, **z** the data-frame containing the continuous response variable; **data** the data set of attributes with which you wish to build the tree.

Step 1: Load Required Packages

We build our regression tree using the **bodyfat** data frame contained in the **TH.data** package:

```
> library(tree)
> data("bodyfat", package = "TH.data")
```

NOTE... ↗

The **bodyfat** data set was collected by Garcia et al²⁴to develop improved predictive regression equations for body fat content derived from common anthropometric measurements. The original study collected data from 117 healthy German subjects, 46 men and 71 women. The **bodyfat** data frame contains the data collected on 10 variables for the 71 women, see Table 10.

Name	Description
DEXfat	body fat measured by DXA(response variable).
age	age in years.
waistcirc	waist circumference.
hipcirc	hip circumference.
elbowbreadth	breadth of the elbow.
kneebreadth	breadth of the knee.
anthro3a	sum of logarithm of three anthropometric measurements.
anthro3b	sum of logarithm of three anthropometric measurements.
anthro3c	sum of logarithm of three anthropometric measurements.
anthro4	sum of logarithm of three anthropometric measurements.

Table 10: Response and independent variables in `bodyfat` data frame

Step 2: Prepare Data & Tweak Parameters

Following the approach taken by Garcia et al, we use 45 of the 71 observations to build the regression tree. The remainder will be used for prediction. The 45 training observations were selected at random without replacement.

```
> set.seed(465)
> train <- sample(1:71, 45 , FALSE)
```

Step 3: Estimate the Decision Tree

Now we are ready to fit the decision tree using the training sample. We take the log of DEXfat as the response variable.

```
> fit<- tree(log(DXFAT) ~ ., data = bodyfat[train
,], split="deviance")
```

To see details of the fitted tree enter.

```
> fit

node), split, n, deviance, yval
 * denotes terminal node

1) root 45 6.72400 3.364
  2) anthro4 < 5.33 19 1.33200 3.004
```

```
4) anthro4 < 4.545 5 0.25490 2.667 *
```

```
14) waistcirc < 104.25 10 0.07201 3.707 *
15) waistcirc > 104.25 5 0.08854 3.874 *
```

The terminal value at each node is the estimated percent cover of $\log(\text{DEXfat})$. For example, the root node indicates the overall mean of $\log(\text{DEXfat})$ is 3.364 percent. This is approximately the same value we would get from:

```
> mean(log(bodyfat$DEXfat))
[1] 3.359635
```

Following the splits, the first terminal node is 4, where the estimated abundance of $\log(\text{DEXfat})$ that has $\text{anthro4} < 5.33$ and $\text{anthro4} < 4.545$ is 2.667 percent.

A summary of the tree can also be obtained.

```
> summary(fit)

Regression tree:
tree(formula = log(DEXfat) ~ ., data = bodyfat[train
  , , split = "deviance")
Variables actually used in tree construction:
[1] "anthro4"    "hipcirc"     "waistcirc"
Number of terminal nodes: 7
Residual mean deviance: 0.01687 = 0.6411 / 38
Distribution of residuals:
      Min.    1st Qu.    Median    Mean    3rd Qu.
-0.268300 -0.080280   0.009712   0.000000   0.073400
                                         Max.
                                         0.332900
```

Notice that `summary(fit)` shows:

1. The type of tree in this case a **Regression tree**;
2. the formula used to fit the tree;
3. the variables used to fit the tree;

4. the number of terminal nodes in this case 7;
5. the residual mean deviance of 0.01687;
6. the distribution of the residuals, in this case they have a mean of 0,

We plot the tree, see Figure 8.1.

```
> plot(fit); text(fit)
```

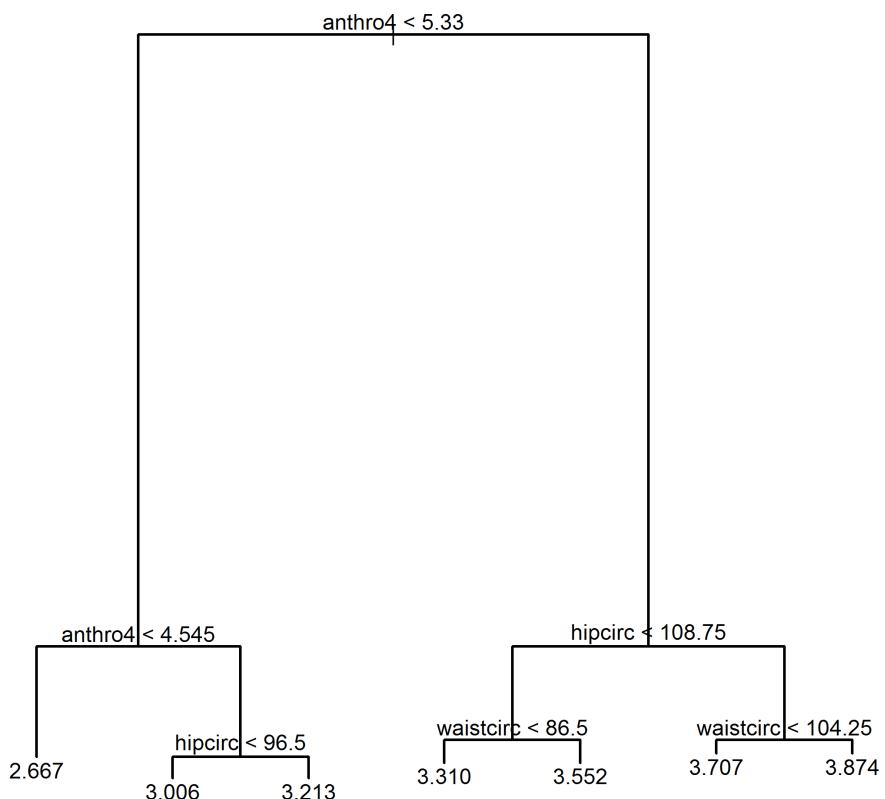


Figure 8.1: Fitted Regression Tree for `bodyfat`

Step 4: Assess Model

We use a leave-one-out cross-validation with the results plotted in Figure 8.2. Since the jagged line reaches a minimum close to the number of branches in the original fitted tree there is little to be gained from pruning this tree.

```
> fit.cv <- cv.tree(fit, K=45)
> plot(fit.cv)
```

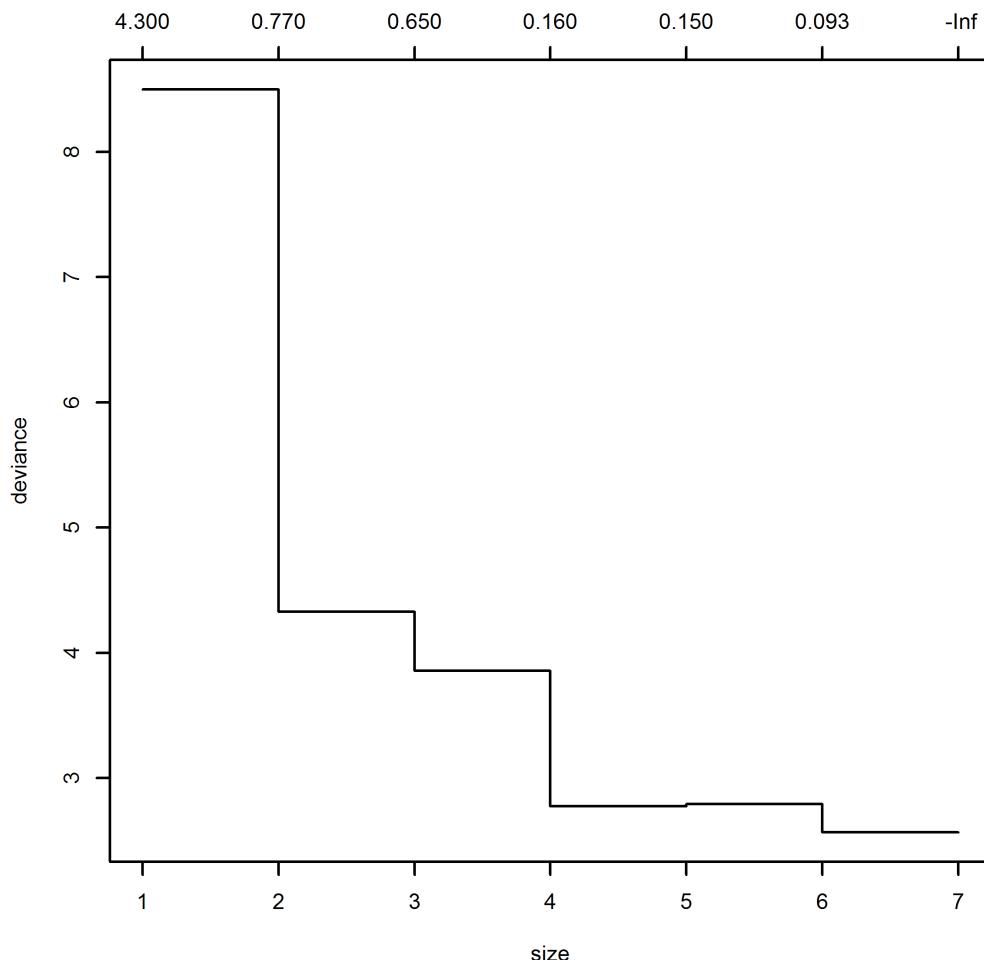


Figure 8.2: Regression tree cross validation results using `bodyfat`

Step 5: Make Predictions

We use the test observations and the fitted decision tree to predict $\log(\text{DEXfat})$. The scatter plot between predicted and observed values is shown in Figure 8.3. The squared correlation coefficient between predicted and observed values is 0.795.

```
> pred<-predict(fit,newdata=bodyfat[-train,])  
  
> plot(bodyfat$DEXfat[-train],pred,xlab="DEXfat",  
+       ylab="Predicted Values",, main="Training Sample  
+       Model Fit")  
  
> round(cor(pred,bodyfat$DEXfat[-train])^2,3)  
[1] 0.795
```

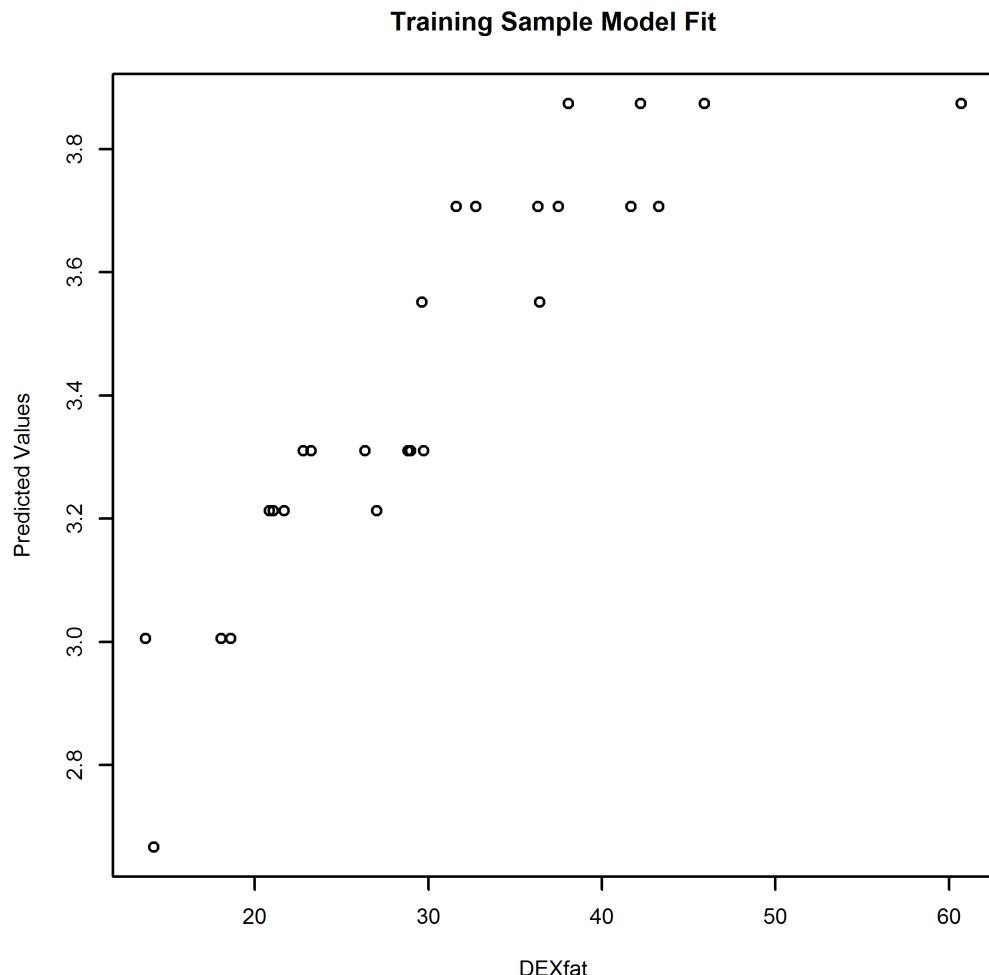


Figure 8.3: Scatterplot of predicted versus observed observations for the regression tree of `bodyfat`

Technique 9

Conditional Inference Regression Tree

A conditional inference regression tree is a non-parametric regression tree embedding tree-structured regression models. It is similar to the regression tree of page 62 but with extra information about the distribution of subjects in the leaf nodes. It can be estimated using the package `party` with the `ctree` function:

```
ctree(z ~ ., data, ...)
```

Key parameters include the continuous response variable `Z`, the data-frame of classes; `data` the data set of attributes with which you wish to build the tree.

Step 1: Load Required Packages

We build the decision tree using the `bodyfat` (see page 62) data frame contained in the `TH.data` package.

```
> library ("party")
> data("bodyfat", package="TH.data")
```

Step 2 is outlined on page 63.

Step 3: Estimate and Assess the Decision Tree

We estimate the model using the training data followed by a plot of the fitted tree, shown in Figure 9.1.

```
> fit<- ctree(log(DEXfat) ~ ., data = bodyfat[train,])
> plot(fit)
```

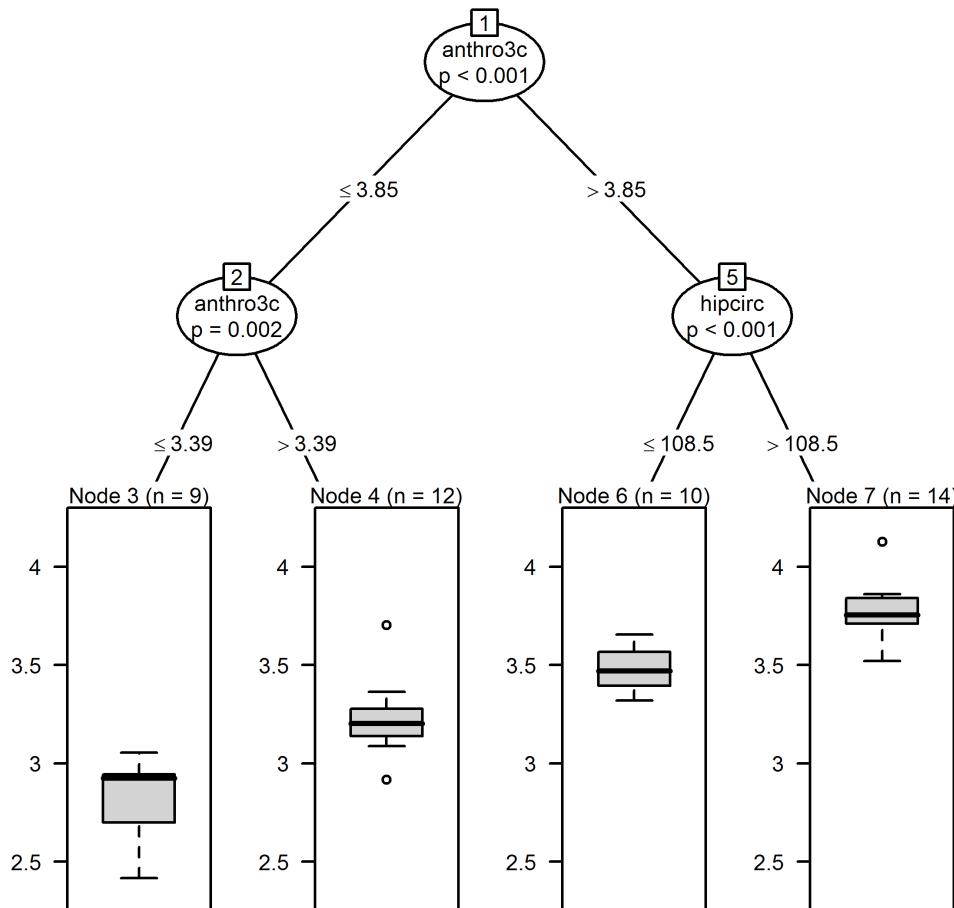


Figure 9.1: Fitted Conditional Inference Regression Tree using `bodyfat`

Further details of the fitted tree can be obtained using the `print` function:

```
> print(fit)
```

```
Conditional inference tree with 4 terminal
nodes
```

```
Response: log(DEXfat)
Inputs: age, waistcirc, hipcirc, elbowbreadth,
        kneebreadth, anthro3a, anthro3b, anthro3c, anthro4
Number of observations: 45

1) anthro3c <= 3.85; criterion = 1, statistic =
   35.215
2) anthro3c <= 3.39; criterion = 0.998, statistic
   = 14.061
   3)* weights = 9
2) anthro3c > 3.39
   4)* weights = 12
1) anthro3c > 3.85
5) hipcirc <= 108.5; criterion = 0.999, statistic
   = 15.862
   6)* weights = 10
5) hipcirc > 108.5
   7)* weights = 14
```

At each branch of the tree (after `root`) we see in order: the branch number, the split rule (e.g.`anthro3c <= 3.85`); the `criterion` reflects the reported p-value and is derived from `statistic`. Terminal nodes (or leaf) are indicated with "*" and `weights` are the number of subjects/ observations at that node.

Step 5: Make Predictions

We use the validation observations and the fitted decision tree to predict `log(DEXfat)`. The scatter plot between predicted and observed values is shown in Figure 9.2. The squared correlation coefficient between predicted and observed values is 0.68.

```
> pred<-predict(fit,newdata=bodyfat[-train,])

> plot(bodyfat$DEXfat[-train],pred,xlab="DEXfat",
      ylab="Predicted Values",, main="Training Sample
      Model Fit")

> round(cor(pred,bodyfat$DEXfat[-train])^2,3)
      [,1]
log(DEXfat) 0.68
```

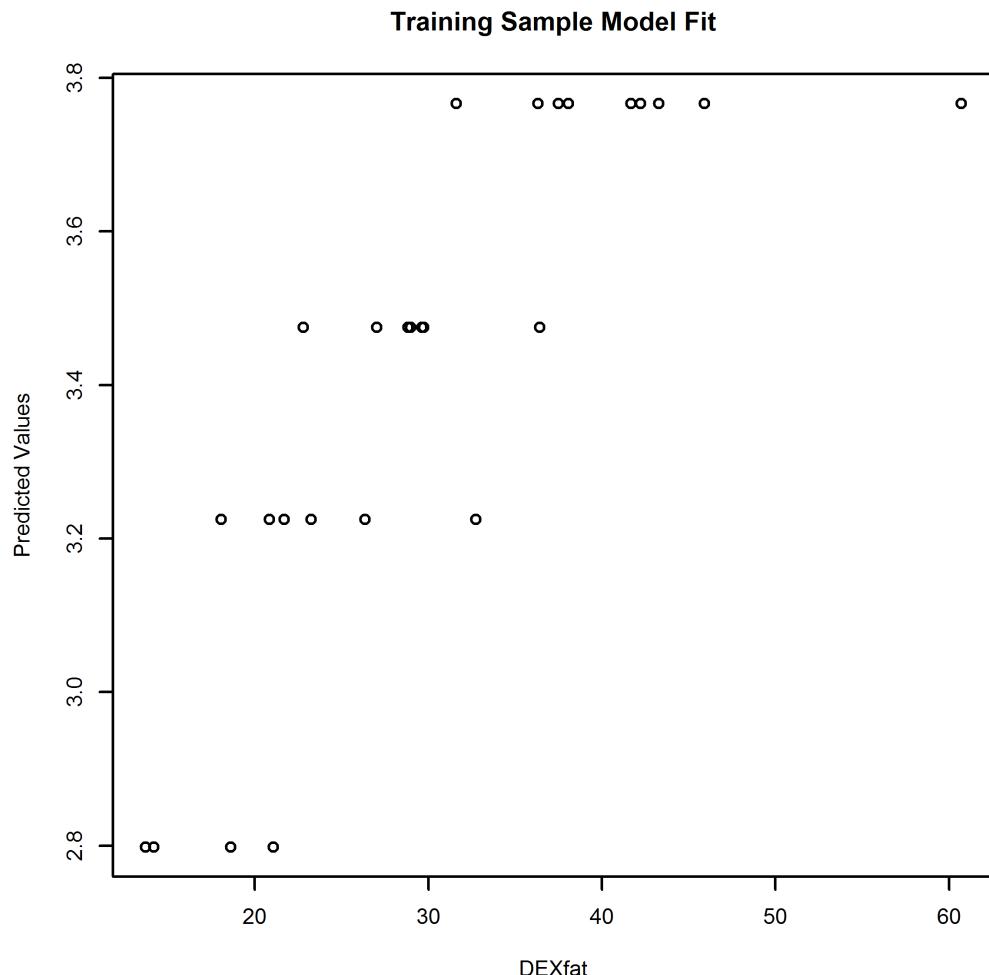


Figure 9.2: Conditional Inference Regression Tree scatter plot for bodyfat

Technique 10

Linear Model Based Recursive Partitioning

A linear model based recursive partitioning regression tree can be estimated using the package `party` with the `mob` function:

```
ctree(z ~x+y+z|a+b+c, data, model = linearModel...)
```

Key parameters include the continuous response variable `Z`; the linear regression covariates `x,y` and `z` and the covariates `a, b` and `c` with which you wish to partition the tree.

Step 1: Load Required Packages

We build the decision tree using the `bodyfat` (see page 62) data frame contained in the `TH.data` package.

```
> library ("party")
> data("bodyfat", package="TH.data")
```

Step 2: Prepare Data & Tweak Parameters

For our analysis we will use the entire `bodyfat` sample. We begin by taking the log of the response variable (`DEXfat`) and the two conditioning variables (`waistcirc,hipcirc`), the remaining covariates form the partitioning set.

```
> bodyfat$DEXfat<-log(bodyfat$DEXfat)
> bodyfat$waistcirc <-log(bodyfat$waistcirc)
```

```
> bodyfat$hipcirc <- log(bodyfat$hipcirc)  
  
> f<- DEXfat~waistcirc + hipcirc|age+ elbowbreadth +  
  kneebreadth+ anthro3a + anthro3b + anthro3c +  
  anthro4
```

Step 3: Estimate & Evaluate Decision Tree

We estimate the model using the function `mob`. Since looking at the printed output can be rather tedious, a visualization is shown in Figure 10.1. By default, this produces partial scatter plots of the response variable against each of the regressors (`waistcirc,hipcirc`) in the terminal nodes. Each scatter plot also shows the fitted values. From this visualization, it can be seen that in the nodes 3, 4 and 5 bodyfat increases with waist and hip circumference. The increase of value appears steepest in node 3 and flattens out somewhat in node 5.

```
> fit <- mob(f, data = bodyfat, model =  
  linearModel, control = mob_control(objfun = logLik)  
)  
  
> plot(fit)
```

☛ PRACTITIONER TIP ☛

Model based recursive partitioning searches for the locally optimal split in the response variable by minimizing the objective function of the model. Typically, this will be something like deviance or the negative log likelihood function. It can be specified using the `mob_control` control function. For example to use deviance you would set `control = mob_control(objfun = deviance)`. In our analysis we use the log likelihood with `control = mob_control(objfun = logLik)`.

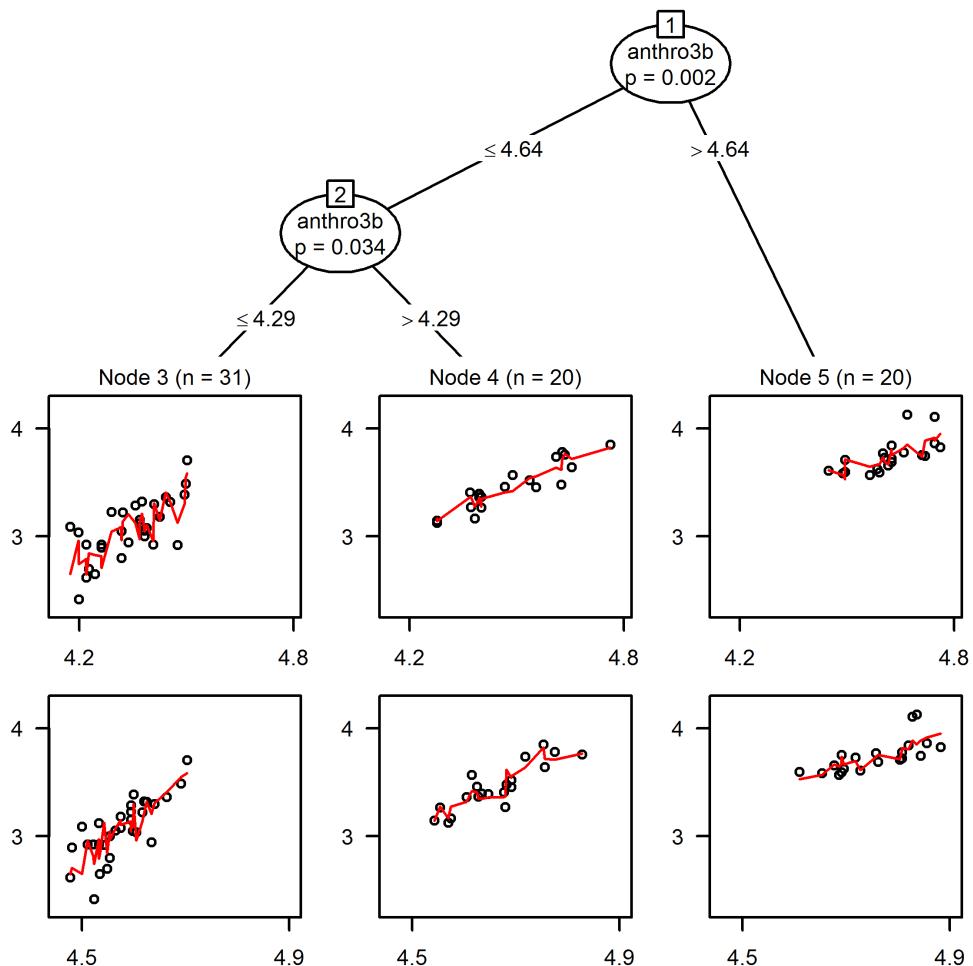


Figure 10.1: Linear model based recursive partitioning tree using `bodyfat`

Further details of the fitted tree can be obtained by typing the fitted model's name.

```
> fit
1) anthro3b <= 4.64; criterion = 0.998, statistic =
   24.549
2) anthro3b <= 4.29; criterion = 0.966, statistic =
   16.962
3)* weights = 31
Terminal node model
Linear model with coefficients:
(Intercept)      waistcirc       hipcirc
```

```
-14.309          1.196          2.660

2) anthro3b > 4.29
  4)* weights = 20
Terminal node model
Linear model with coefficients:
(Intercept)    waistcirc      hipcirc
-5.3867        0.9887        0.9466

1) anthro3b > 4.64
  5)* weights = 20
Terminal node model
Linear model with coefficients:
(Intercept)    waistcirc      hipcirc
-3.5815        0.6027        0.9546
```

The output informs us that the tree consists of five nodes. At each branch of the tree (after `root`) we see in order: the branch number, the split rule (e.g.`anthro3b <= 4.64`). Note `criterion` reflects the reported p-value²⁵ and is derived from `statistic`. Terminal nodes are indicated with "*" and `weights` are the number of subjects/ observations at that node. The output also presents the estimated regression coefficients at the terminal nodes. We can also use `coef` function to obtain a summary of the estimated coefficients and their associated node:

```
> round(coef(fit),3)
(Intercept)  waistcirc  hipcirc
3       -14.309     1.196    2.660
4       -5.387     0.989    0.947
5       -3.582     0.603    0.955
```

The `summary` function also provides detailed statistical information on the fitted coefficients by node. For example `summary(fit)` produces the following (we only show details of node 3).

```
> summary(fit)

$ '3'

Call:
NULL

Weighted Residuals:
```

Min	1Q	Median	3Q	Max
-0.3272	0.0000	0.0000	0.0000	0.4376

Coefficients:

	Estimate	Std. Error	t value	Pr(> t)	
(Intercept)	-14.3093	2.4745	-5.783	3.29e-06	***
waistcirc	1.1958	0.4033	2.965	0.006119	**
hipcirc	2.6597	0.6969	3.817	0.000685	***

Signif. codes:	0	***	0.001	**	0.01
	*		0.05	*	0.1
	.				1

Residual standard error: 0.1694 on 28 degrees of freedom

Multiple R-squared: 0.6941, Adjusted R-squared: 0.6723

F-statistic: 31.77 on 2 and 28 DF, p-value: 6.278e-08

When comparing models it can be useful to have the value of the log likelihood function or the Akaike information criterion. These can be obtained by:

```
> logLik(fit)
'log Lik.' 54.42474 (df=14)

> AIC(fit)
[1] -80.84949
```

PRACTITIONER TIP

The test statistics and p-values computed in each node can be extracted using the function `sctest()`. For example to see the statistics for node 2 you would type `sctest(fit, node = 2)`.

Step 5: Make Predictions

We use the function `predict` and then display the scatter plot between predicted and observed values in Figure 10.2. The squared correlation coefficient between predicted and observed values is 0.89.

```
> pred<-predict(fit,newdata=bodyfat)
```

```
> plot(bodyfat$DEXfat,pred,xlab="DEXfat", ylab="Predicted Values", main="Full Sample Model Fit")  
> round(cor(pred,bodyfat$DEXfat)^2,3)  
[1] 0.89
```

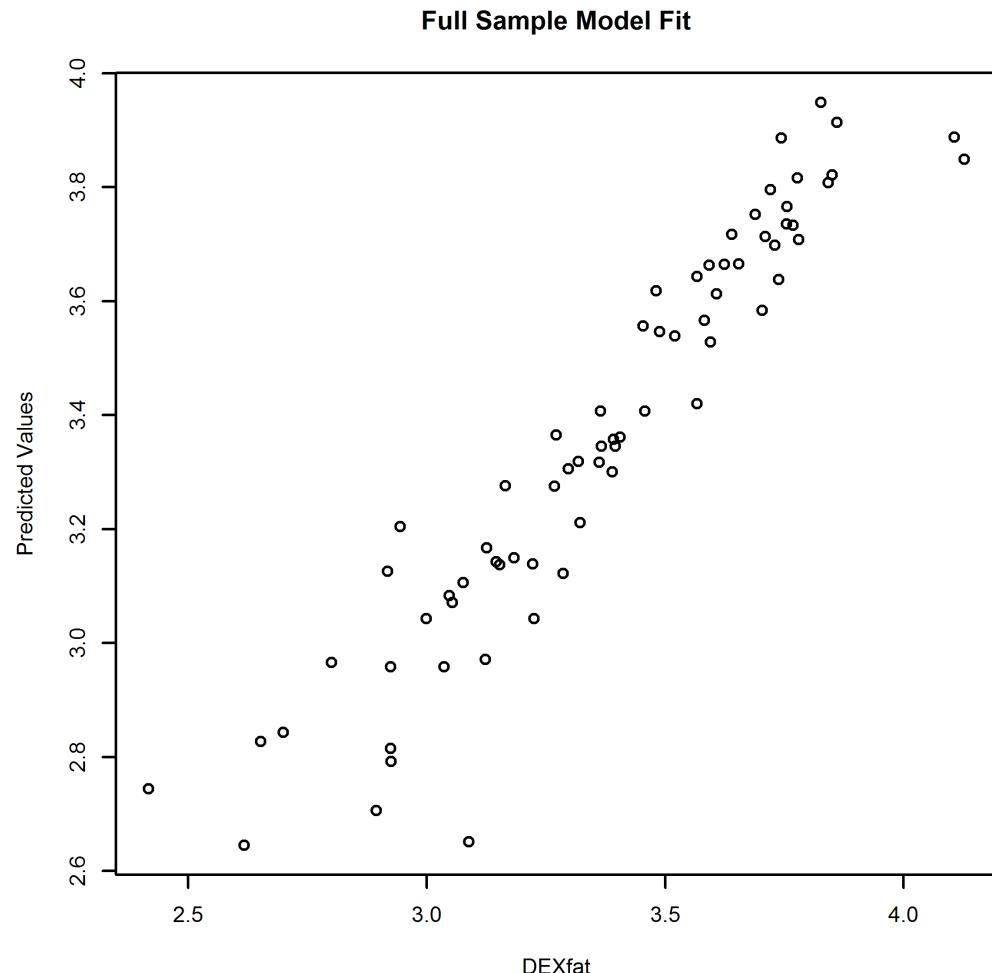


Figure 10.2: Linear model based recursive partitioning tree predicted and observed values using `bodyfat`

Technique 11

Evolutionary Regression Tree

A evolutionary regression tree model can be estimated using the package `evtree` with the `evtree` function:

```
evtree(z ~ ., data, ...)
```

Key parameters include the continuous response variable Z; and the covariates contained in `data`.

Step 1: Load Required Packages

We build the decision tree using the `bodyfat` (see page 62) data frame contained in the `TH.data` package.

```
> library ("evtree")
> data("bodyfat", package="TH.data")
```

Step 2: Prepare Data & Tweak Parameters

For our analysis we will use 45 observations for the training sample. We take the log of the response variable (`DEXfat`) and two of the covariates (`waistcirc`,`hipcirc`). The remaining covariates are used in their original form.

```
> set.seed(465)
> train <- sample(1:71, 45 , FALSE)

> bodyfat$DEXfat<-log(bodyfat$DEXfat)
> bodyfat$waistcirc <-log(bodyfat$waistcirc)
> bodyfat$hipcirc <-log(bodyfat$hipcirc)
```

```
> f<- DEXfat~waistcirc + hipcirc+age+ elbowbreadth +
  kneebreadth+ anthro3a + anthro3b + anthro3c +
  anthro4
```

Step 3: Estimate & Evaluate Decision Tree

We estimate the model using the function `evtree`. A visualization is obtained using `plot` and shown in Figure 11.1. This produces box and whisper plots of the response variable in each leaf. From this visualization, it can be seen that body fat increases as we move from node 3 to node 5.

```
> fit <- evtree(f,   data = bodyfat[train,])
> plot(fit)
```

☛ PRACTITIONER TIP ☛

Notice that `evtree.control` is used to control important aspects of a tree. You can change the number of evolutionary iterations using `niterations`; this is useful if your tree does not converge by the default number of iterations. You can also specify the number of trees in the population using `ntrees` and the tree depth with `maxdepth`. For example, to limit the maximum tree depth to three and the number of iterations to ten thousand, you would enter something along the lines of:

```
fit <- evtree(f,   data = bodyfat[train,],
control =evtree.control (maxdepth=2),
niterations=10000)
```

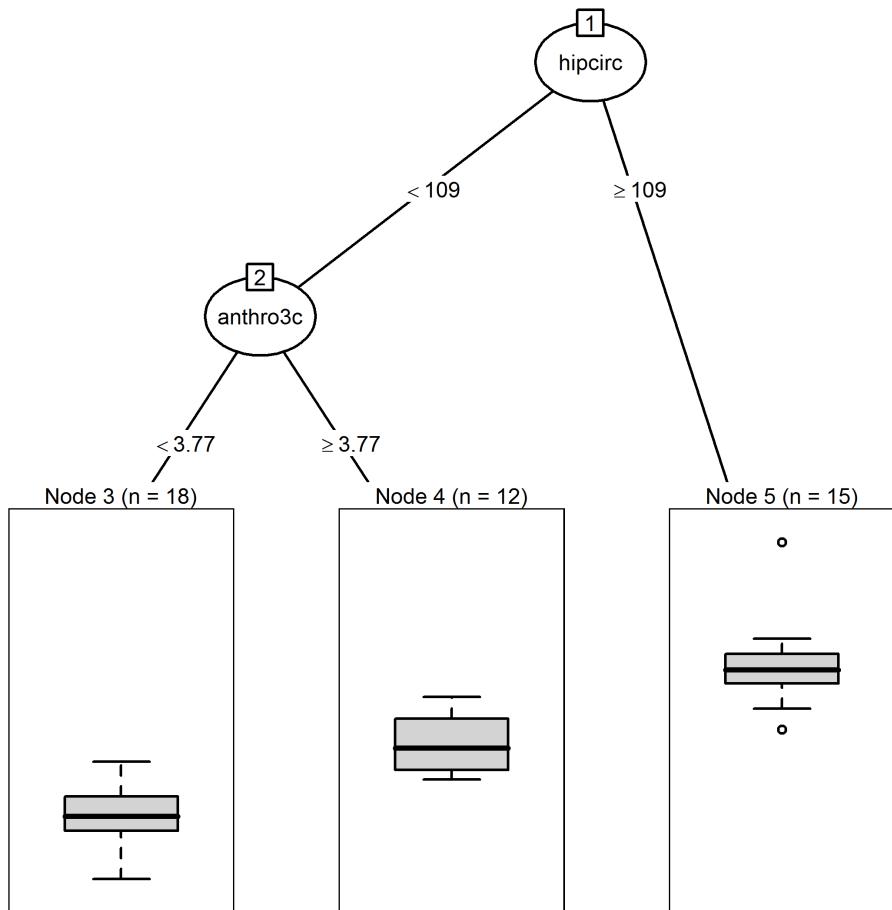


Figure 11.1: Fitted Evolutionary Regression Tree for `bodyfat`

Further details of the fitted tree can be obtained by typing the fitted model's name.

```
> fit
```

```
Model formula:  
DEXfat ~ waistcirc + hipcirc + age + elbowbreadth +  
kneebreadth +  
anthro3a + anthro3b + anthro3c + anthro4
```

```
Fitted party:  
[1] root
```

```
| [2] hipcirc < 109
|   | [3] anthro3c < 3.77: 20.271 (n = 18, err =
385.9)
|   | [4] anthro3c >= 3.77: 31.496 (n = 12, err =
186.8)
| [5] hipcirc >= 109: 43.432 (n = 15, err = 554.8)

Number of inner nodes:      2
Number of terminal nodes: 3
```

☛ PRACTITIONER TIP ☛

Decision tree models can outperform other techniques in the situation when relationships are irregular (e.g., non-monotonic), but they are also known to be inefficient when relationships can be well approximated by simpler models.

Step 5: Make Predictions

We use the function `predict` and then display the scatter plot between predicted and observed values in Figure 11.2. The squared correlation coefficient between predicted and observed values is 0.73.

```
> pred<-predict(fit,newdata=bodyfat[-train,])
>
> plot(bodyfat$DEXfat[-train],pred,xlab="DEXfat",
+       ylab="Predicted Values",, main="Model Fit")
>
> round(cor(pred,bodyfat$DEXfat[-train]))^2,3)
[1] 0.727
```

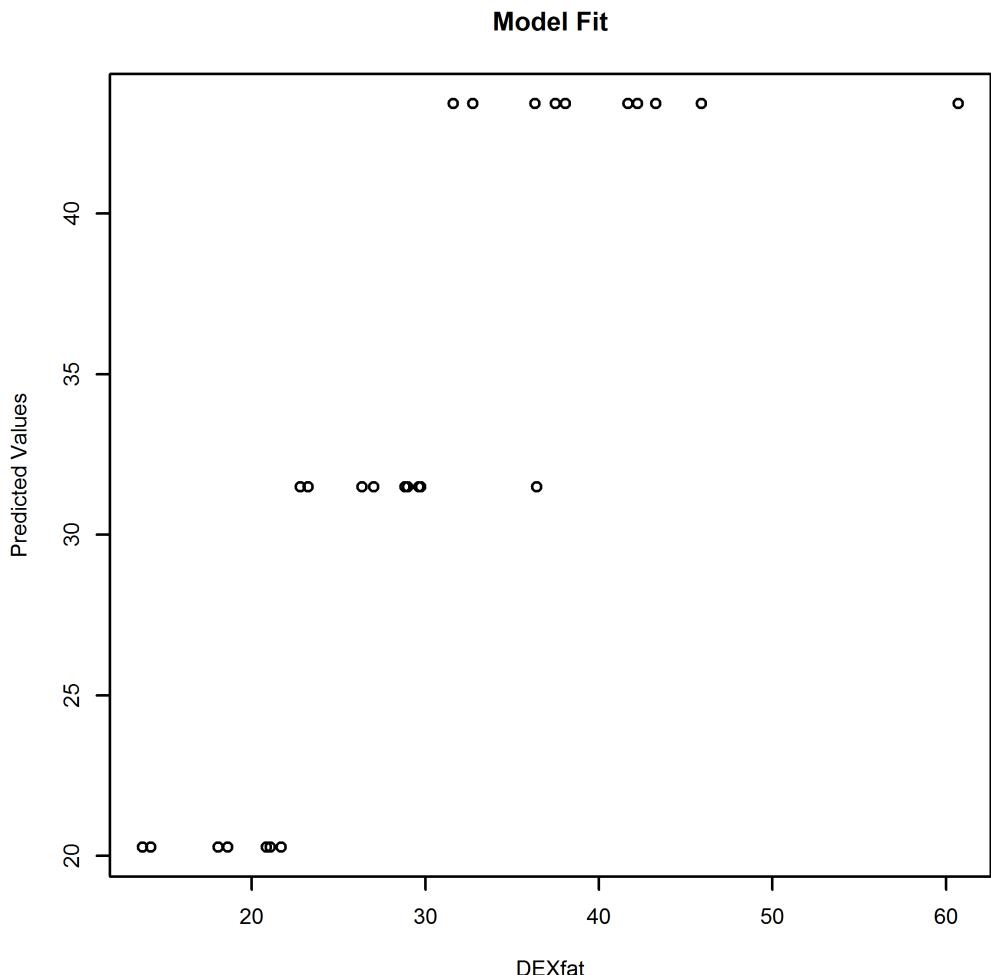


Figure 11.2: Scatter plot of fitted and observed values for the Evolutionary Regression Tree using `bodyfat`

Decision Trees for Count & Ordinal Response Data

☛ PRACTITIONER TIP ☛

An oft believed maxim is the more data the better, whilst this may sometimes be true, it is a good idea to try to rationally reduce the number of attributes you include in your decision tree to the minimum set of highest value attributes. Oates and Jensen²⁶ studied the influence of database size on decision tree complexity. They found tree size and complexity strongly depends on the size of the training set.

It is always worth thinking about and removing uninformative attributes prior to decision tree construction. For practical ideas and additional tips on how to do this see the excellent papers of John²⁷, Brodley and Friedl²⁸, and Cano and Herrera²⁹.

Technique 12

Poisson Decision Tree

A decision tree for a count response variable (y_i) following a Poisson distribution with a mean that depends on the covariates $\{x_1, \dots, x_k\}$ can be built using the package `rpart` with the `rpart` function:

```
rpart(z ~ ., data, method = "poisson")
```

Key parameters include `method = "poisson"` which is used to indicate the type of tree to be built, `z` the data-frame containing the Poisson distributed response variable; `data` the data set of attributes with which you wish to build the tree.

Step 1: Load Required Packages

We build a Poisson decision tree using the `DebTrivedi` data frame contained in the `MixAll` package:

```
> library (rpart)
> library(MixAll)
> data(DebTrivedi)
```

NOTE... ↗

Deb and Trivedi³⁰ model counts of medical care utilization by the elderly in the United States using data from the National Medical Expenditure Survey. They analyze data on 4406 individuals, aged 66 and over, who are covered by Medicare, a public insurance program. The objective is to model the demand for medical care using as the response variable the number of physician/non-physician office and hospital outpatient visits. The data is contained in the `DebTrivedi` data frame available in the `MixAll` package.

Step 2: Prepare Data & Tweak Parameters

The number of physician office visits (`ofp`) is the response variable. The covariates are - `hosp` (number of hospital stays), `health` (self-perceived health status), `numchron` (number of chronic conditions), as well as the socioeconomic variables `gender`, `school` (number of years of education), and `privins` (private insurance indicator).

```
> f<-ofp~hosp+health+numchron+gender+school+privins
```

Step 3: Estimate the Decision Tree & Assess fit

Now we are ready to fit the decision tree.

```
> fit<-rpart(f,data=DebTrivedi, method = "poisson")
```

To see a plot of the tree use the `plot` and `text` methods.

```
> plot(fit); text(fit, use.n=TRUE, cex=.8))
```

Figure 12.1 shows a visualization of the fitted tree. Each of the five terminal nodes reports the event rate, the total number of events and number of observations for that node. For example for the rule chain: `numchron <1.5` → `hosp<0.5` → `numchron <0.5`, the event rate is 3.121 with 923 events at that node.

• PRACTITIONER TIP •

To see the number of events and observations at every node in a decision tree plot add `all = TRUE` to the `text` function.

```
plot(fit); text(fit, use.n=TRUE, all=TRUE,  
cex=.6)
```

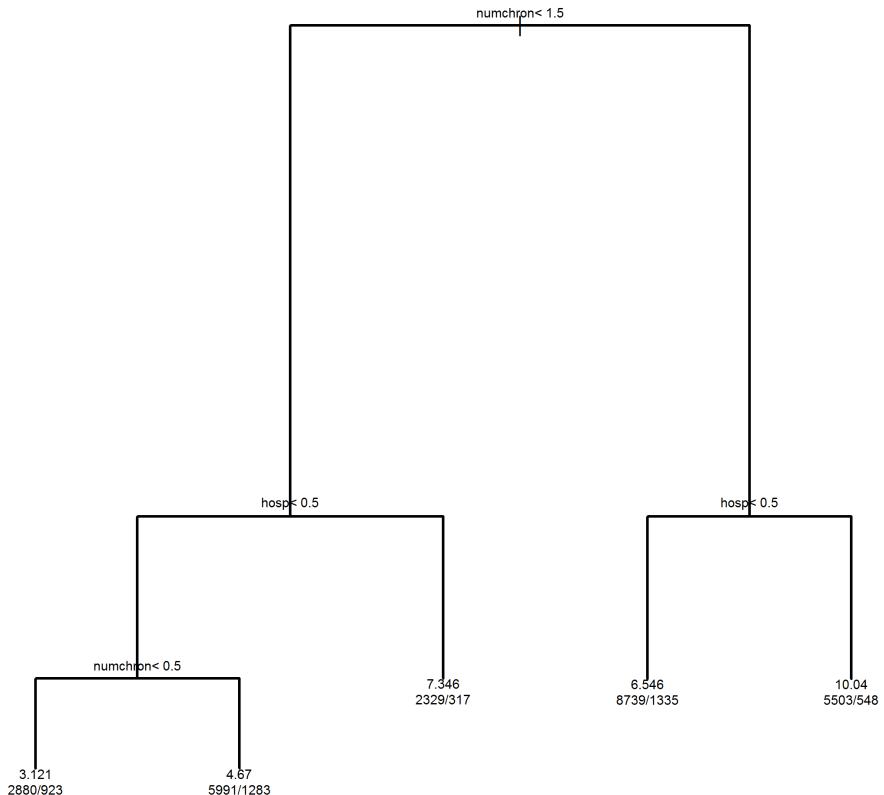


Figure 12.1: Poisson Decision Tree using the `DebTrivedi` data frame

To help validate the decision tree we use the `printcp` function. The ‘`cp`’

part of the function stands for the “complexity parameter” of the tree. The function indicates the optimal tree size based on the cp value.

```
> printcp(fit,digits=3)

Rates regression tree:
rpart(formula = f, data = DebTrivedi, method = "
  poisson")

Variables actually used in tree construction:
[1] hosp      numchron

Root node error: 26943/4406 = 6.12

n= 4406

      CP nsplit rel_error xerror   xstd
1 0.0667      0     1.000 1.000 0.0332
2 0.0221      1     0.933 0.942 0.0325
3 0.0220      2     0.911 0.918 0.0326
4 0.0122      3     0.889 0.896 0.0315
5 0.0100      4     0.877 0.887 0.0315
```

The `printcp` function returns the formula used to fit the tree, the variables used to build the tree (in this case: `hosp`, `numchron`), the root node error (6.12), the number of events at the root node (4406) and the relative error, the cross-validation error(`xerror`) , `xstd` and `CP` at each node split. Each row represents a different height of the tree. In general, more levels in the tree often imply a lower classification error. However, you run the risk of over fitting.

Figure 12.2 plots the `rel_error` and `cp` parameter.

```
> plotcp(fit)
```

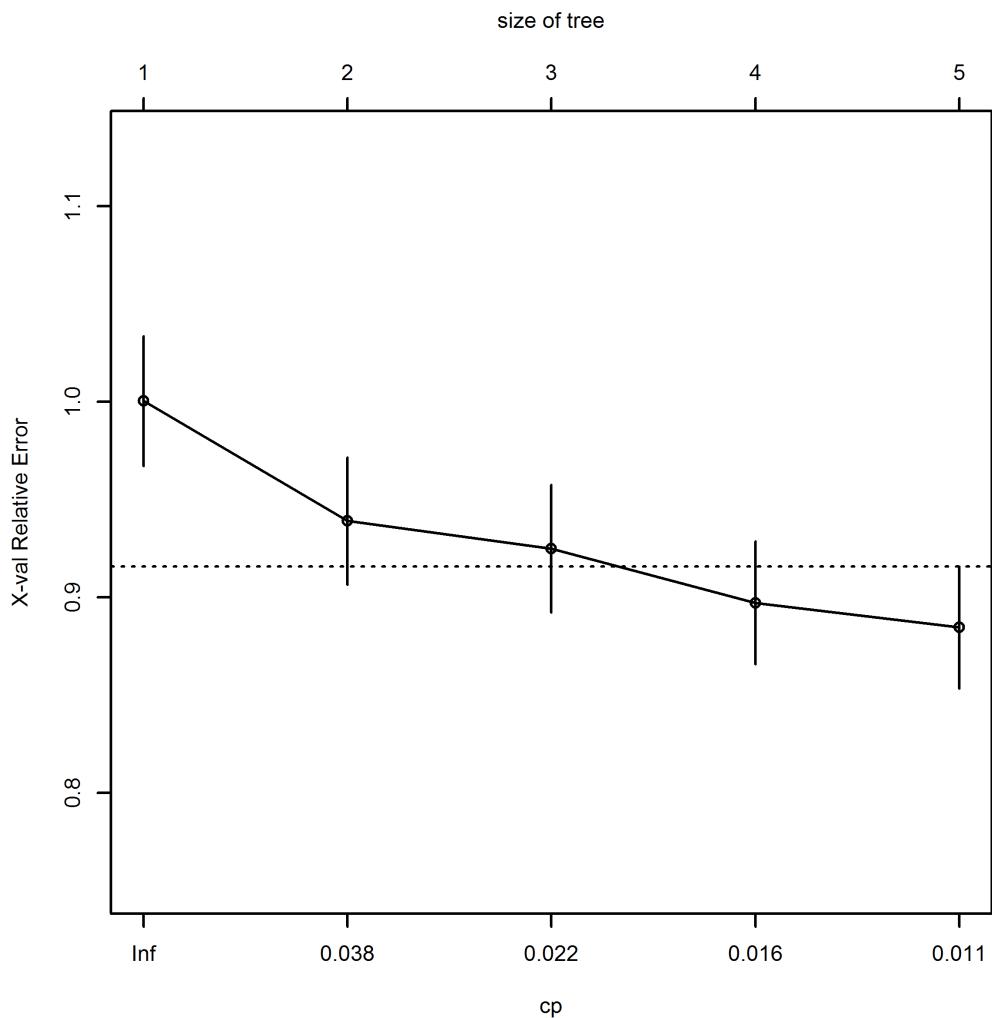


Figure 12.2: Complexity parameter for the Poisson decision tree using the DebTrivedi data frame

☛ PRACTITIONER TIP ☛

A simple rule of thumb is to choose the lowest level where the `rel_error + xstd < xerror`. Another rule of thumb is to prune the tree so that it has the minimum `xerror`. You can do this automatically using the following:

```
pfit<- prune(fit, cp= fit$cptable[which.min(fit$cptable[, "xerror"]),"CP"])
```

`pfit` contains the pruned tree.

Since the tree is relatively parsimonious we retain the original tree. A summary of the tree can also be obtained by typing::

```
> summary(fit)
```

The first part of the output displays similar data to that obtained by using the `printcp` function. This is followed by variable importance details:

Variable importance		
numchron	hosp	health
55	37	8

We see that `numchron` is the most important variable followed by `hosp` and then `health`.

The second part of the summary function gives details of the tree, with the last few lines giving details of the terminal nodes. For example, for node 5 we observe:

```
Node number 5: 317 observations
events=2329, estimated rate=7.346145 , mean
deviance=5.810627
```

Technique 13

Poisson Model Based Recursive Partitioning

A poisson model based recursive partitioning regression tree can be estimated using the package `party` with the `mob` function:

```
ctree(z ~x+y+z|a+b+c, data, model = linearModel,  
      family=poisson(link = "log"),...)
```

Key parameters include the poisson distributed response variable of counts Z; the regression covariates x,y and z and the covariates a, b and c with which you wish to partition the tree.

Step 1: Load Required Packages

We build a Poisson decision tree using the `DebTrivedi` data frame contained in the `MixAll` package: Details of this data frame are given on page 86.

```
> library ("party")  
> library(MixAll)  
> data(DebTrivedi)
```

Step 2: Prepare Data & Tweak Parameters

For our analysis we use all the observations in `DebTrivedi` to estimate a model with the response variable `ofp` (number of physician office visits) and `numchron` (number of chronic conditions) as poisson regression conditioning variables. The remaining variables (`hosp`, `health`, `gender`, `school`, `privins`) are used as the partitioning set. The model is stored in `f`.

```
> f<-ofp~numchron | hosp+health+gender+school+privins
```

Step 3: Estimate the Decision Tree & Assess fit

Now we are ready to fit and plot the decision tree, see Figure 13.1.

```
> fit <- mob(f, data=DebTrivedi, model =
  linearModel, family=poisson(link = "log"))
> plot(fit)
```

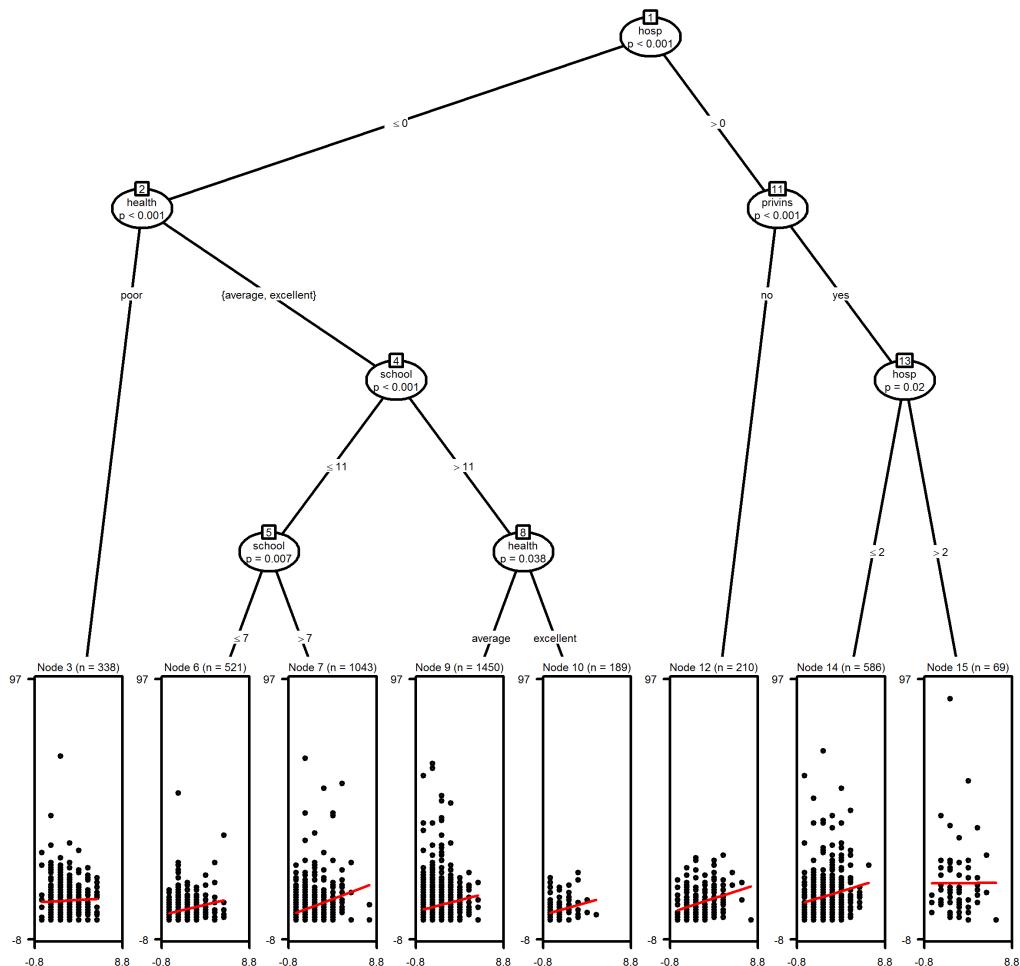


Figure 13.1: Poisson Model Based Recursive Partitioning Tree for DebTrivedi

Coefficient estimates at the leaf nodes are given by:

```
> round(coef(fit),3)
  (Intercept) numchron
3      7.042    0.231
6      2.555    0.892
7      2.672    1.400
9      4.119    0.942
10     2.788    1.043
12     3.769    1.212
14     6.982    1.123
15    14.579    0.032
```

When comparing models it can be useful to have the value of the log likelihood function or the Akaike information criterion. These can be obtained by:

```
> logLik(fit)
'log Lik.' -14109.87 (df=31)

> AIC(fit)
[1] 28281.75
```

The results of the parameter stability tests for any given node can be retrieved using `sctest`. For example, to retrieve the statistics for node 2 enter:

```
> round(sctest(fit, node = 2),2)
            hosp health gender school privins
statistic 13.94   36.25    7.93   24.22   16.87
p.value    0.11    0.00    0.09    0.00    0.00
```

Technique 14

Conditional Inference Ordinal Response Tree

The Conditional Inference Ordinal Response Tree is used when the response variable is measured on an ordinal scale. In marketing, for instance, we often see consumer satisfaction measured on an ordinal scale - “very satisfied”, “satisfied”, “dissatisfied” and “very dissatisfied”. In medical research constructs such as self-perceived health are often measured on an ordinal scale - “very unhealthy,” “unhealthy”, “healthy”, “very healthy”. A conditional inference ordinal response tree can be built using the package `party` with the `ctree` function:

```
ctree(z ~ ., data, ...)
```

Key parameters include the response variable `Z` an ordered factor, the data-frame of classes; `data` the data set of attributes with which you wish to build the tree.

Step 1: Load Required Packages

We build our tree using the `wine` data frame contained in the `ordinal` package:

```
> library ("party")
> library("ordinal")
> data("wine")
```

NOTE... ↗

The `wine` data frame was analyzed by Randall³¹ in an experiment on factors determining the bitterness of wine. The bitterness (`rating`) was measured as 1 = “least bitter” and 5 = “most bitter”. Temperature and contact between juice and skins can be controlled during wine production. Two treatment factors were collected - temperature (`temp`) and contact (`contact`) with each having two levels. Nine judges assessed wine from two bottles from each of the four treatment conditions, resulting in a total of 72 observations in all.

Step 2: Estimate and Assess the Decision Tree

We estimate the model using all of the data with `rating` as the response variable. This is followed by a plot, shown in Figure 14.1, of the fitted tree.

```
> fit<-ctree(rating ~ temp + contact,data= wine)
> plot(fit)
```

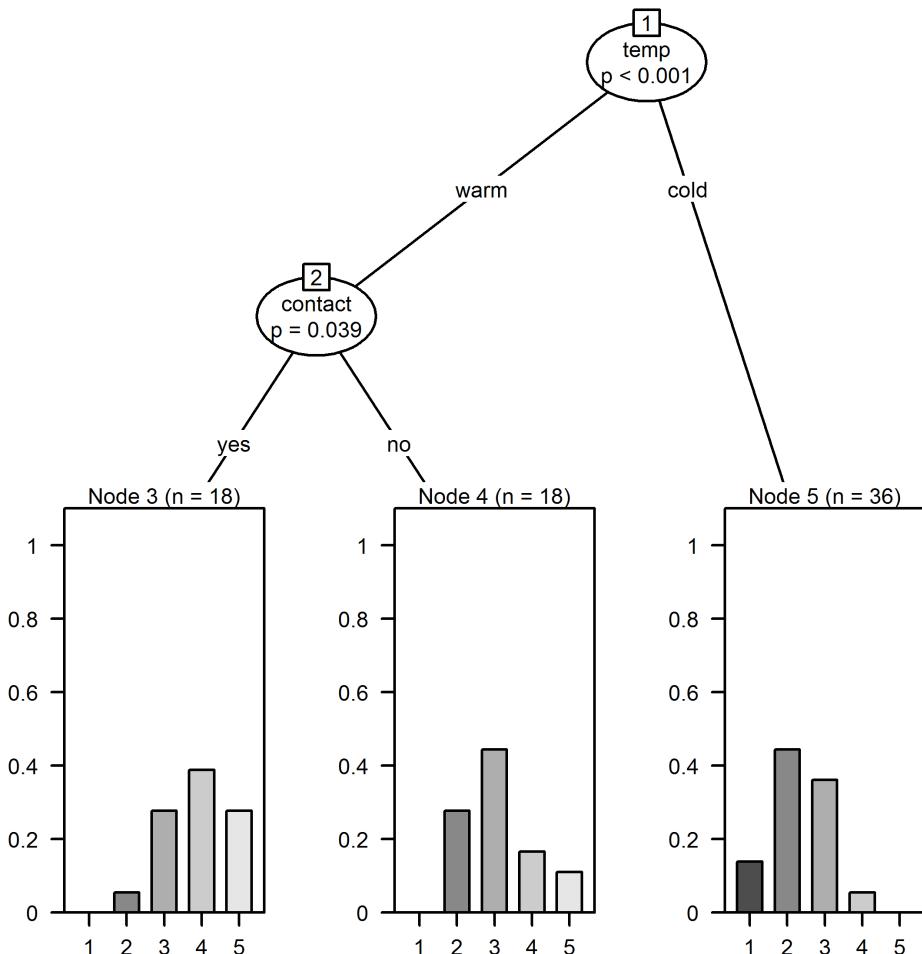


Figure 14.1: Fitted Conditional Inference Ordinal Response Tree using `wine`

The decision tree has three leafs - Node 3 with 18 observations, Node 4 with 18 observations and Node 5 with 36 observations. The covariate temp is highly significant ($p < 0.001$) and contact is significant at the 5% level. Notice the terminal nodes contain the distribution of bitterness scores.

We compare the fitted values with the observed values and print out the confusion matrix and error rate. The overall misclassification rate is 56.9% for the fitted tree.

```
> tb<-table(wine$rating ,pred , dnn=c("actual" , "predicted"))
> tb
```

```
predicted
actual   1   2   3   4   5
      1   0   5   0   0   0
      2   0  16   5   1   0
      3   0  13   8   5   0
      4   0   2   3   7   0
      5   0   0   2   5   0
> error <- 1-(sum(diag(tb))/sum(tb))
> round (error ,3)
[1] 0.569
```

Decision Trees for Survival Analysis

Studies involving time to event data are numerous and arise in all areas of research. For example, survival times or other time to failure related measurements, such as relapse time, are major concerns of modeling medical data. The Cox proportional hazard regression model and its extensions have been the traditional tool of the data scientist for modeling survival variables with censoring. These parametric (and semi-parametric) models remain useful staples as they allow simple interpretations of the covariate effects and can readily be used for statistical inference. However, such models force a specific link between the covariates and the response. Even though interactions between covariates can be incorporated, they must be specified by the analyst.

Survival decision trees allow the data scientist to carry out their analysis without imposing a specific link function or know aprori the nature of variable interaction. Survival trees offer great flexibility because they can automatically detect certain types of interactions without the need to specify them beforehand. prognostic groupings are a natural output from survival trees this is because the basic idea of a decision tree is to partition the covariate space recursively to form groups (nodes in the tree) of subjects which are similar according to the outcome of interest.

Technique 15

Exponential Algorithm

A decision tree for survival data, where the time values are assumed to fit an exponential model³² can be built using the package `rpart` with the `rpart` function:

```
rpart(z ~ ., data, method = "exp")
```

Key parameters include `z` the survival response variable (note we set `z=Surv(time, status)` where `Surv` is a survival object constructed using the `survival` package); `data` the data set of explanatory variable; and `method = "exp"` is used to indicate a survival decision tree.

Step 1: Load Required Packages

We build a survival decision tree using the `rhDNase` data frame contained in the `simexaft` package. The required packages and data are loaded as follows:

```
> library (rpart)
> library("simexaft")
> library("survival")
> data("rhDNase")
```

NOTE... ↗

Respiratory disease in patients with cystic fibrosis is characterized by airway obstruction caused by the accumulation of thick, purulent secretions. The viscoelasticity of these secretions can be reduced in-vitro by recombinant human deoxyribonuclease I (rhDNase), a bioengineered copy of the human enzyme. The `rhDNase` data set contained in the `simexaft` package contains a subset of the original data collected by Fuchs et al³³ who performed a randomized, double-blind, placebo-controlled study on 968 adults and children with cystic fibrosis to determine the effects of once-daily and twice-daily administration of rhDNase. The patients were treated for 24 weeks as outpatients. The `rhDNase` data frame contains data on the occurrence and resolution of all exacerbations for 641 patients.

Step 2: Prepare Data & Tweak Parameters

The forced expiratory volume (FEV) was considered a risk factor and was measured twice at randomization (`rhDNase$fev` and `rhDNase$fev2`). We take the average of the two measurements as an explanatory variable. The response is defined as the logarithm of the time from randomization to the first pulmonary exacerbation measured in the object `survreg(Surv(time2, status))`.

```
> rhDNase$fev.ave <- (rhDNase$fev + rhDNase$fev2)/2  
> z<-Surv(rhDNase$time2, rhDNase$status)
```

Step 3: Estimate the Decision Tree & Assess fit

Now we are ready to fit the decision tree.

```
> fit<-rpart(z ~ trt + fev.ave,data= rhDNase, method = "exp")
```

To see a plot of the tree simply type its name.

```
> plot(fit); text(fit, use.n=TRUE, cex=.8, all=TRUE)
```

Figure 15.1 shows a plot of the fitted tree. Notice that the terminal nodes report the estimated rate as well as the number of events and observations available. For example for the rule `fev.ave <80.03` the estimated event rate is 0.35 with 27 events out of a total of 177.

☛ PRACTITIONER TIP ☛

To see the rules that lead to a specific node use the function `path.rpart(fitted_Model, node = x)`. For example to see the rules associated with node 7 enter:

```
> path.rpart(fit, node = 7)

node number: 7
root
fev.ave< 80.03
fev.ave< 46.42
```

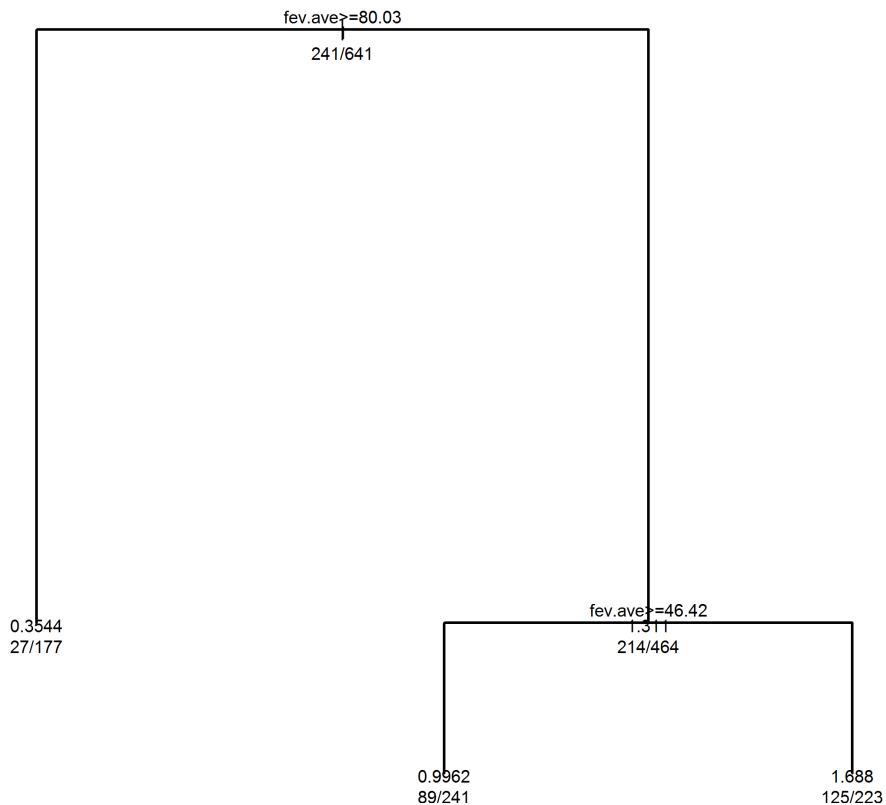


Figure 15.1: SurvivalDecision Tree using the **rhDNase** data frame

To help assess the decision tree we use the `plotcp` function, shown in Figure 15.2. Since the tree is relatively parsimonious there is little need to prune.

```
> plotcp(fit)
```

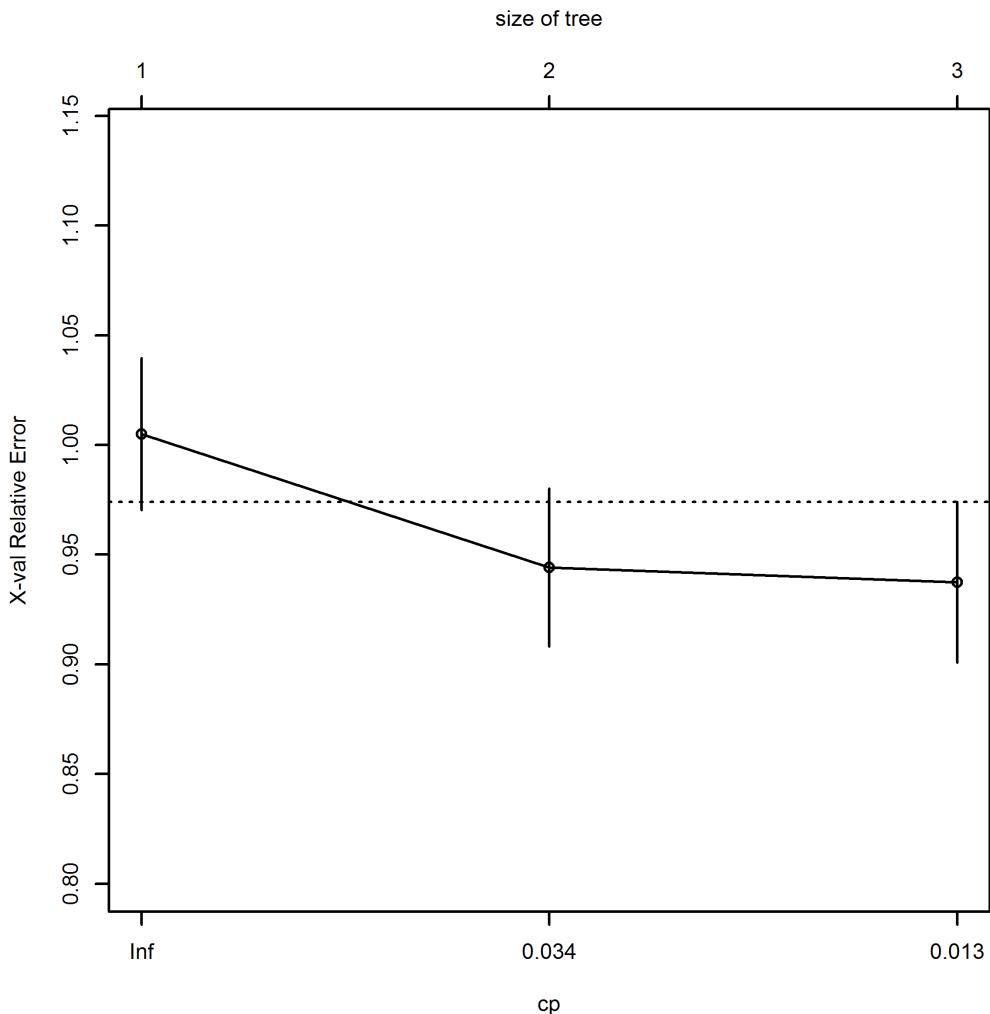


Figure 15.2: Complexity parameter and error for the Survival Decision Tree using the `rhDNase` data frame

Survival times can vary greatly between subjects. Decision tree analysis is a useful tool to homogenize the data by separating it into different subgroups based on treatments and other relevant characteristics. In other words, a single tree will group subjects according to their survival behavior based on their covariates. For a final summary of the model, it can be helpful to plot the probability of survival based on the final nodes in which the individual patients landed as shown in Figure 15.3.

We see that the node 2 appears to have the most favorable survival characteristics.

```
> km <- survfit(z ~ fit$where, data= rhDNase)
```

```
> plot(km, lty = 1:3, mark.time = FALSE, xlab = "Time", ylab = "Status")
> legend(150, 0.2, paste('node', c(2,4,5)), lty = 1:3)
```

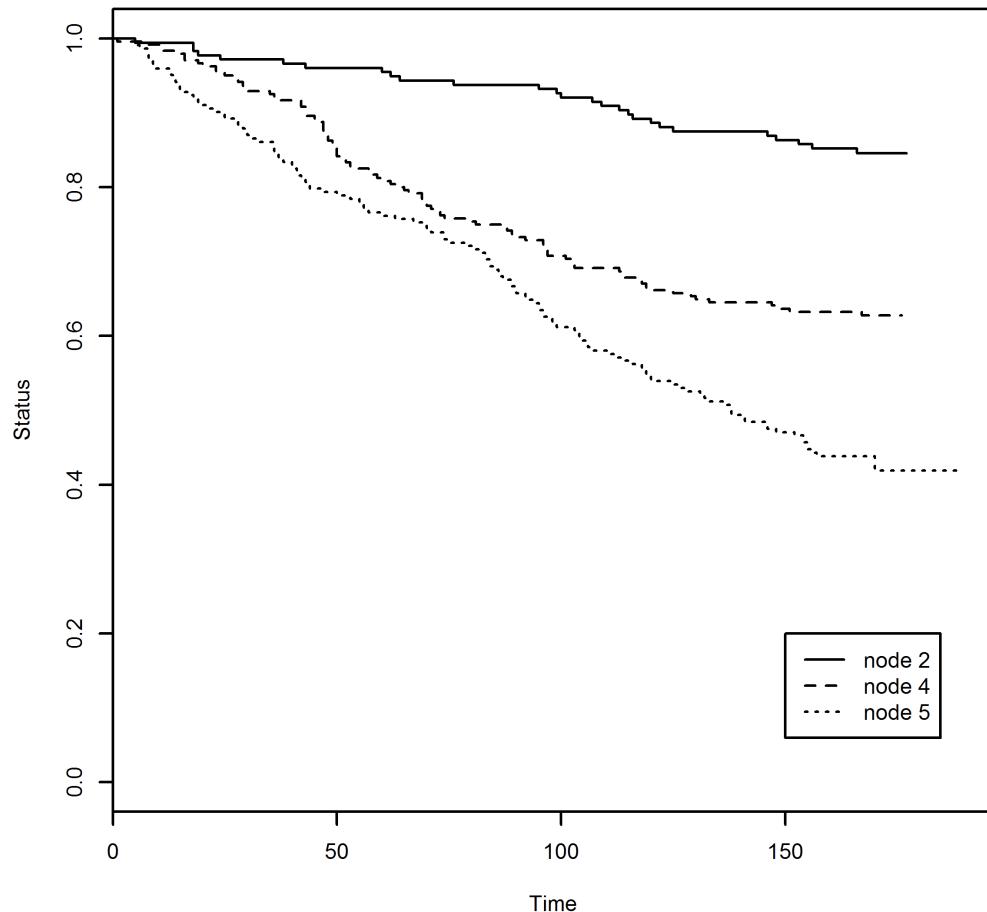


Figure 15.3: Survival plot by terminal node for rhDNase

Technique 16

Conditional Inference Survival Tree

A conditional inference survival tree is a non-parametric regression tree embedding tree-structured regression models. This is essentially a decision tree but with extra information about survival in the terminal nodes³⁴. It can be built using the package `party` with the `ctree` function:

```
ctree(z ~ ., data, ...)
```

Key parameters include `z` the survival response variable (note we set `z=Surv(time, status)` where `Surv` is a survival object constructed using the `survival` package); and `data` the sample of explanatory variables.

Step 1: Load Required Packages

We build a conditional inference survival decision tree using the `rhDNase` data frame contained in the `simexaft` package. The required packages and data are loaded as follows:

```
> library ("party")
> library("simexaft")
> library("survival")
> data("rhDNase")
```

Details of step 2 are given on page 100.

Step 3: Estimate the Decision Tree & Assess fit

Next we fit and plot the decision tree. Figure 16.1 shows the resultant tree.

```
> fit<-ctree(z ~ trt + fev.ave, data= rhDNase)
> plot(fit)
```

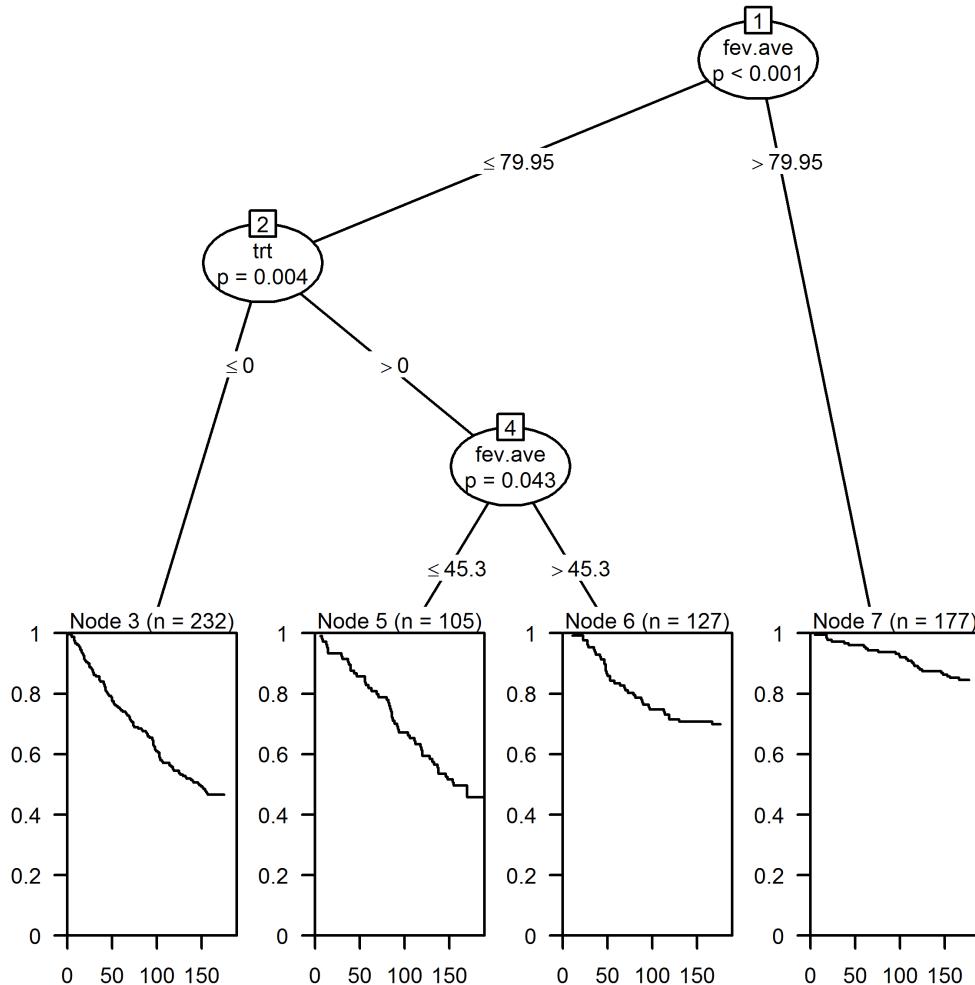


Figure 16.1: Conditional Inference Survival Tree for rhDNase

Notice that the internal nodes report the p-value for split, whilst the leaf nodes give the number of subjects and a plot of the estimated survival curve. We can obtain a summary of the tree by typing:

```
> print(fit)
```

```
Conditional inference tree with 4 terminal
nodes
```

```
Response: z
Inputs: trt, fev.ave
Number of observations: 641

1) fev.ave <= 79.95; criterion = 1, statistic =
   58.152
2) trt <= 0; criterion = 0.996, statistic = 9.34
   3)* weights = 232
2) trt > 0
   4) fev.ave <= 45.3; criterion = 0.957, statistic
      = 5.254
   5)* weights = 105
   4) fev.ave > 45.3
      6)* weights = 127
1) fev.ave > 79.95
   7)* weights = 177
```

Node 7 is a terminal or leaf node (the symbol “*” signifies this) with the decision rule `fev.ave > 79.95`. At this node there are 177 observations.

We grab the fitted responses using the function `treeresponse` and store them in `stree`. Notice that every `stree` component is a survival object of class "survfit"

```
> stree <- treeresponse(fit)

> class(stree[[2]])
[1] "survfit"

> class(stree[[7]])
[1] "survfit"
```

For this particular tree we have four terminal nodes, so there are only four unique survival objects. You can use the `where` method to see which nodes the observations are in.

```
> subjects <- where(fit)

> table(subjects)
subjects
 3   5   6   7
232 105 127 177
```

So we have 232 subjects in node 3 and 127 subjects in node 6 which agree with the numbers reported in Figure 16.1.

We end our initial analysis by plotting in Figure 16.2 the survival curve for node 3 with a 95% confidence interval, and also mark on the plot the time of individual subject events.

```
> plot(stree[[3]], conf.int = TRUE, mark.time = TRUE  
      , ylab="Cumulative Survival (%)", xlab="Days Elapsed  
      ")
```

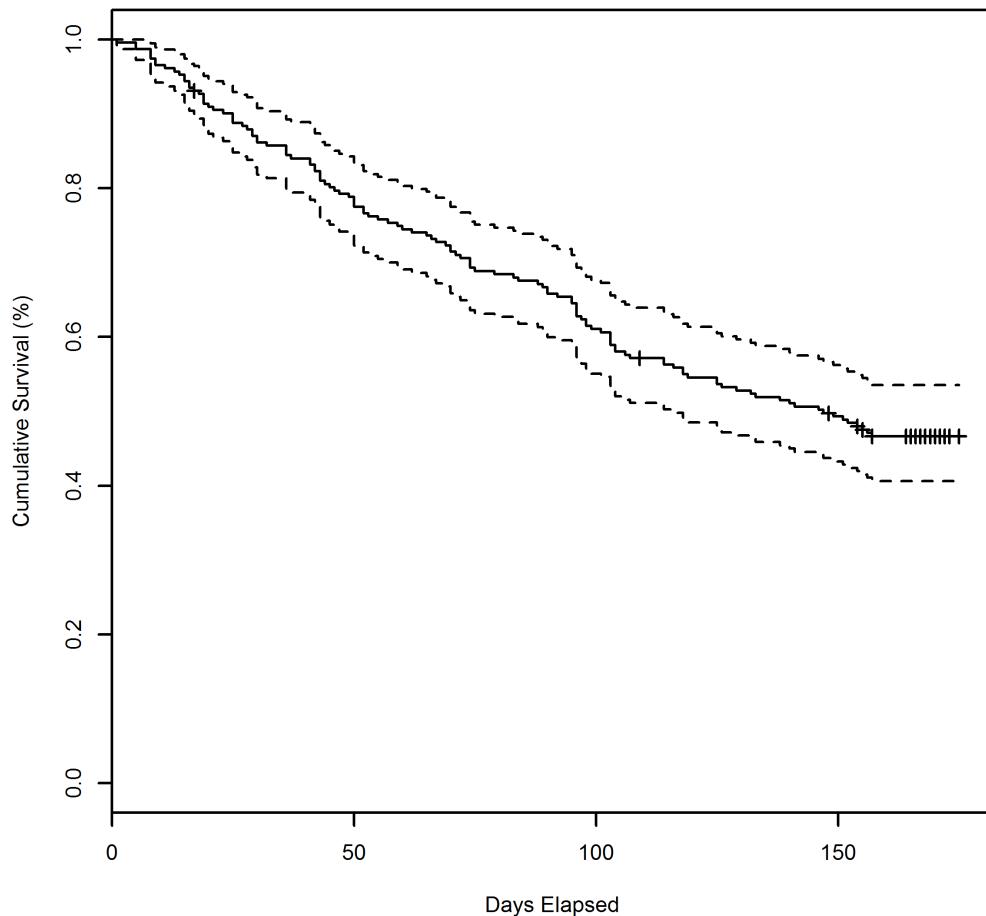


Figure 16.2: Survival curve for node 3

Notes

¹See for example, the top ten list of Wu, Xindong, et al. "Top 10 algorithms in data mining." *Knowledge and Information Systems* 14.1 (2008): 1-37.

²Koch Y, Wolf T, Sorger PK, Eils R, Brors B (2013) Decision-Tree Based Model Analysis for Efficient Identification of Parameter Relations Leading to Different Signaling States. *PLoS ONE* 8(12): e82593. doi:10.1371/journal.pone.0082593

³Ebenezer Hailemariam, Rhys Goldstein, Ramtin Attar & Azam Khan. (2011). Real-Time Occupancy Detection using Decision Trees with Multiple Sensor Types SimAUD 2011 Conference Proceedings: Symposium on Simulation for Architecture and Urban Design.

⁴Taken from Stiglic G, Kocbek S, Pernek I, Kokol P (2012) Comprehensive Decision Tree Models in Bioinformatics. *PLoS ONE* 7(3): e33812. doi:10.1371/journal.pone.0033812

⁵Bohanec M, Bratko I (1994) Trading accuracy for simplicity in decision trees, *Machine Learning* 15: 223–250.

⁶See for example J. R. Quinlan, "Induction of decision trees," *Mach. Learn.*, vol. 1, no. 1, pp. 81–106, 1986

⁷See J. R. Quinlan, C4.5: programs for machine learning. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993.

⁸See Breiman, Leo, et al. Classification and regression trees. CRC press, 1984.

⁹See R. L. De M'antaras, "A distance-based attribute selection measure for decision tree induction," *Mach. Learn.*, vol. 6, no. 1, pp. 81–92, 1991.

¹⁰Zhang, Ting, et al. "Using decision trees to measure activities in people with stroke." Engineering in Medicine and Biology Society (EMBC), 2013 35th Annual International Conference of the IEEE. IEEE, 2013.

¹¹See for example J. Liu, M. A. Valencia-Sánchez, G. J. Hannon, and R. Parker, "MicroRNA-dependent localization of targeted mRNAs to mammalian P-bodies," *Nature Cell Biology*, vol. 7, no. 7, pp. 719–723, 2005.

¹²Williams, Philip H., Rod Eyles, and George Weiller. "Plant MicroRNA prediction by supervised machine learning using C5.0 decision trees." *Journal of nucleic acids* 2012 (2012).

¹³Nakayama, Nobuaki, et al. "Algorithm to determine the outcome of patients with acute liver failure: a data-mining analysis using decision trees." *Journal of gastroenterology* 47.6 (2012): 664-677.

¹⁴Hepatic encephalopathy, also known as portosystemic encephalopathy, is the loss of brain function (evident in confusion, altered level of consciousness, and coma) as a result of liver failure.

¹⁵de Oña, Juan, Griselda López, and Joaquín Abellán. "Extracting decision rules from police accident reports through decision trees." *Accident Analysis & Prevention* 50 (2013): 1151-1160.

¹⁶See for example Kashani, A., Mohaymany, A., 2011. Analysis of the traffic injury severity on two-lane, two-way rural roads based on classification tree models. *Safety Science* 49, 1314-1320.

¹⁷Monedero, Iñigo, et al. "Detection of frauds and other non-technical losses in a power utility using Pearson coefficient, Bayesian networks and decision trees." *International Journal of Electrical Power & Energy Systems* 34.1 (2012): 90-98.

¹⁸Maximum and minimum value monthly consumption, Number of meter readings, Number of hours of maximum power consumption and three variables to measure abnormal consumption.

¹⁹Wang, Quan, et al. "Tracking tetrahymena pyriformis cells using decision trees." Pattern Recognition (ICPR), 2012 21st International Conference on. IEEE, 2012.

²⁰This data set comes from the Turing Institute, Glasgow, Scotland.

²¹For further details see <http://www.rulequest.com/see5-comparison.html>

²²For further details see:

1. Hothorn, Torsten, Kurt Hornik, and Achim Zeileis. "ctree: Conditional Inference Trees."
2. Hothorn, Torsten, Kurt Hornik, and Achim Zeileis. "Unbiased recursive partitioning: A conditional inference framework." Journal of Computational and Graphical statistics 15.3 (2006): 651-674.
3. Hothorn, Torsten, et al. "Party: A laboratory for recursive partytioning." (2010).

²³<http://www.niddk.nih.gov/>

²⁴Garcia, Ada L., et al. "Improved prediction of body fat by measuring skinfold thickness, circumferences, and bone breadths." Obesity Research 13.3 (2005): 626-634.

²⁵The reported p-value at a node is equal to 1-criterion

²⁶Oates T, Jensen D (1997) The effects of training set size on decision tree complexity. In: Proceedings of the Fourteenth International Conference on Machine Learning. pp 254–262.

²⁷John GH (1995) Robust decision trees: removing outliers from databases. In: Proceedings of the First Conference on Knowledge Discovery and Data Mining. pp 174–179.

²⁸Brodley CE, Friedl MA (1999) Identifying mislabeled training data. J Artif Intell Res 11: 131–167. 19.

²⁹Cano JR, Herrera F, Lozano M (2007) Evolutionary stratified training set selection for extracting classification rules with trade off precision-interpretability. Data & Knowledge Engineering 60(1): 90–108.

³⁰Deb, Partha, and Pravin K. Trivedi. "Demand for medical care by the elderly: a finite mixture approach." Journal of applied Econometrics 12.3 (1997): 313-336.

³¹See Randall, J (1989). The analysis of sensory data by generalised linear model. Biometrical journal 7, pp. 781–793.

³²See Atkinson, Elizabeth J., and Terry M. Therneau. "An introduction to recursive partitioning using the RPART routines." Rochester: Mayo Foundation (2000).

³³See Henry J. Fuchs, Drucy S. Borowitz, David H. Christiansen, Edward M. Morris, Martha L. Nash, Bonnie W. Ramsey, Beryl J. Rosenstein, Arnold L. Smith, and Mary Ellen Wohl for the Pulmozyme Study Group N Engl J Med 1994; 331:637-642 September 8, 1994

³⁴For further details see:

1. Hothorn, Torsten, Kurt Hornik, and Achim Zeileis. "ctree: Conditional Inference Trees."
2. Hothorn, Torsten, Kurt Hornik, and Achim Zeileis. "Unbiased recursive partitioning: A conditional inference framework." Journal of Computational and Graphical statistics 15.3 (2006): 651-674.
3. Hothorn, Torsten, et al. "Party: A laboratory for recursive partytioning." (2010).

Part II

Support Vector Machines

The Basic Idea

The support vector machine (SVM) is a supervised machine learning algorithm³⁵ that can be used for both regression and classification. Let's take a quick look at how SVM performs classification. The core idea of SVMs is that they construct hyperplanes in a multidimensional space that separates objects which belong to different classes. A decision plane is then used to define the boundaries between different classes. Figure 16.3 visualizes this idea. .

The decision plane separates a set of observations into their respective classes using a straight line. In this example, the observations belong either to class “solid circle” or class “edged circle”. The separating line defines a boundary on the right side of which all objects are “solid circle” and to the left of which all objects are “edged circle”.

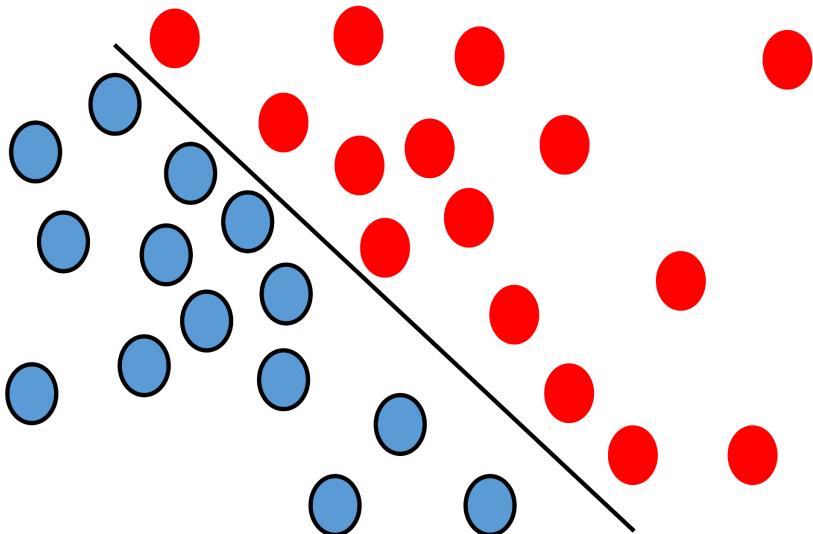


Figure 16.3: Schematic illustration of a decision plane determined by a linear classifier

In practice, as illustrated in Figure 16.4, the majority of classification problems require nonlinear boundaries in order to determine the optimal separation³⁶. Figure 16.5 illustrates how SVMs solve this problem. The left side of the figure represents the original sample (known as the input space) which is mapped using a set of mathematical functions, known as kernels, to the feature space. The process of rearranging the objects for optimal separation is known as transformation. Notice that in mapping from the input space to the feature space the mapped objects are linearly separable. Thus, although SVM uses linear learning methods due to its nonlinear kernel function, it is in effect a nonlinear classifier.

NOTE... ↗

Since intuition is better built from examples that are easy to imagine, lines and points are drawn in the Cartesian plane instead of hyperplanes and vectors in a high dimensional space. Remember that the same concepts apply where the examples to be classified lie in a space whose dimension is higher than two.

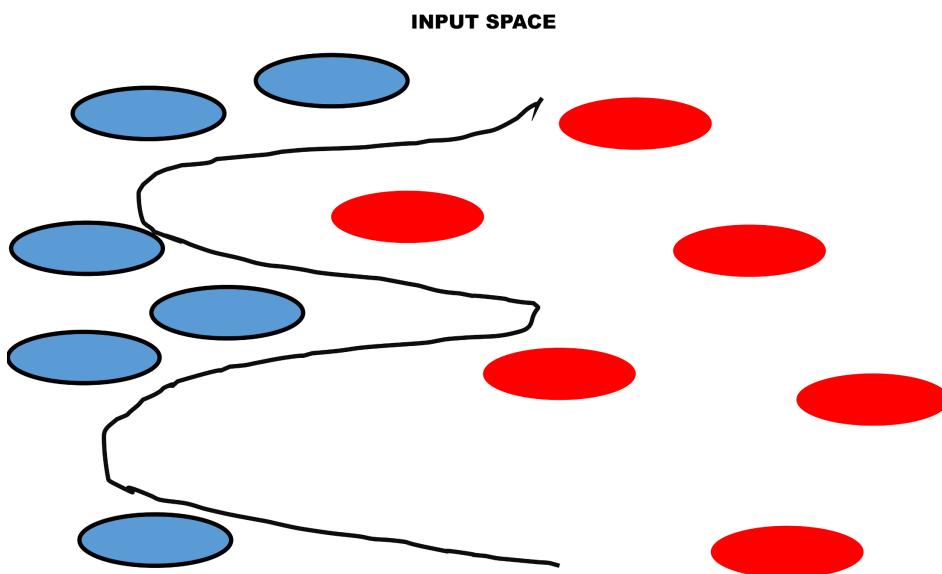


Figure 16.4: Nonlinear boundary required for correct classification

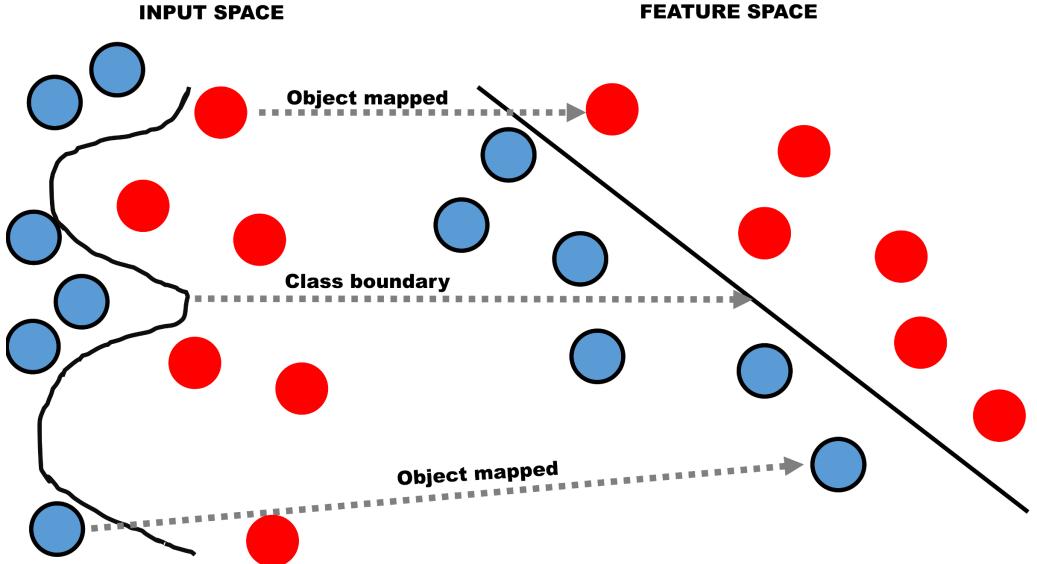


Figure 16.5: Mapping from input to feature space in an SVM

Overview of SVM Implementation

The SVM finds the decision hyperplane leaving the largest possible fraction of points of the same class on the same side, while maximizing the distance of either class from the hyperplane. This minimizes the risk of misclassifying not only the examples in the training data set but also the yet-to-be seen examples of the test set.

To construct an optimal hyperplane, SVM employs an iterative training algorithm, which is used to minimize an error function.

Let's take a look at how this is achieved. Given a set of feature vectors $x_i (i = 1, 2, \dots, N)$ a target $y_i \in \{-1, +1\}$ with corresponding binary labels is associated with each feature vector x_i . The decision function for classification of unseen examples is given as:

$$y = f(x; \alpha) = \text{sign} \left(\sum_{i=1}^{N_s} \alpha_i y_i K(s_i, x) + b \right) \quad (16.1)$$

Where, s_i are the N_s support vectors and $K(s_i, x)$ is the kernel function. The parameters are determined by maximizing the margin hyperplane (see Figure 16.6):

$$\sum_{i=1}^N \alpha_i - \frac{1}{2} \sum_{i=1}^N \sum_{j=1}^N \alpha_i \alpha_j y_i y_j K(x_i \cdot x_j) \quad (16.2)$$

subject to the constraints:

$$\sum_{i=1}^N \alpha_i y_i = 0 \text{ and } 0 \leq \alpha_i \leq C \quad (16.3)$$

To build a SVM classifier, the user needs to tune the cost parameter C and choose a kernel function and optimize its parameters.

☛ PRACTITIONER TIP ☛

I once worked for an Economist who was trained (essentially) in one statistical technique (linear regression and its variants). Whenever there was an empirical issue, this individual always tried to frame it in terms of his understanding of linear models and economics. Needless to say this archaic approach to modeling lead to all sorts of difficulties! The data scientist is pragmatic in their modeling approach, linear, non-linear, Bayesian, boosting, they are guided by statistical theory, machine learning insights and unshackled from the vagueness of economic theory.

Note on the Slack Parameter

The variable C, known as the slack parameter, serves as the cost parameter that controls the trade-off between the margin and classification error. If no slack is allowed (often known as a hard margin) and the data are linearly separable the support vectors are the points which lie along the supporting hyperplanes as shown in Figure 16.6. In this case all of the support vectors lie exactly on the margin.

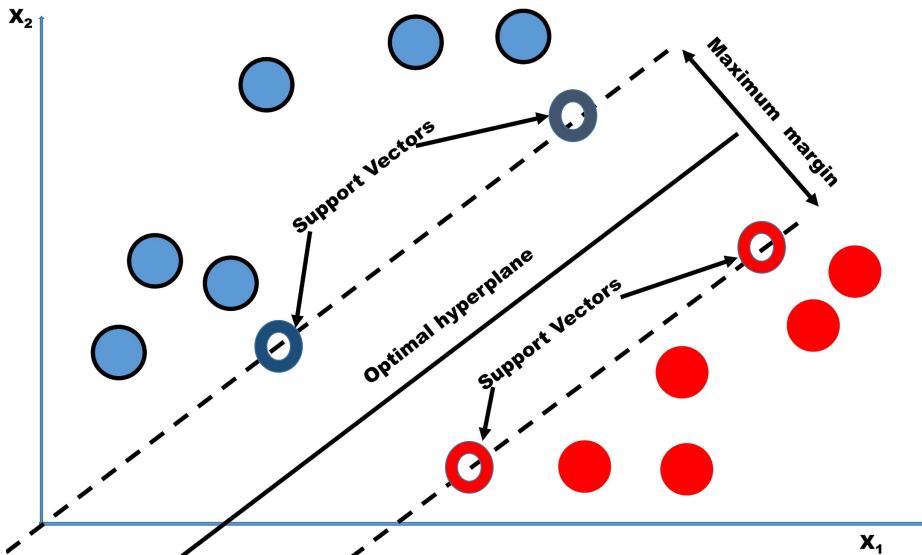


Figure 16.6: Support vectors for linearly separable data and a hard margin

In many situations this will not yield useful results and a soft margin will be required. In this circumstance some proportion of data points are allowed to remain inside the margin. The slack parameter C is used to control this proportion³⁷. A soft margin results in a wider margin and greater error on the training data set, however it improves the generalization of data and reduces the likelihood of over fitting.

NOTE... ↗

The total number of support vectors depends on the amount of allowed slack and the distribution of the data. If a large amount of slack is permitted, there will be a larger number of support vectors than the case where very little slack is permitted. Fewer support vectors means faster classification of test points. This is because the computational complexity of the SVM is linear in the number of support vectors.

Practical Applications

NOTE... ↗

The kappa coefficient³⁸ is a measure of agreement between categorical variables. It is similar to the correlation coefficient in that higher values indicate greater agreement. It is calculated as:

$$k = \frac{P_o - P_e}{1 - P_e} \quad (16.4)$$

P_o is the observed proportion correctly classified. P_e is the proportion correctly classified by chance.

Classification of Dengue Fever Patients

The Dengue virus is a mosquito-borne pathogen that infects millions of people every year. Gomes et al³⁹ use the support vector machine algorithm to classify 28 dengue patients from the Recife metropolitan area, Brazil (13 with dengue fever (DF) and 15 with dengue haemorrhagic fever (DHF)) based on mRNA expression data of 11 genes involved in the innate immune response pathway (MYD88, MDA5, TLR3, TLR7, TLR9, IRF3, IRF7, IFN-alpha, IFN-beta, IFN-gamma, and RIGI).

A radial basis function is used and the model built using leave-one-out cross-validation repeated fifteen times under different conditions to analyze the individual and collective contributions of each gene expression data to DF/DHF classification.

A different gene was removed during the first twelve cross-validations. During the last three cross-validations multiple genes were removed. Figure 16.7 shows the overall accuracy for the support vector machine for differing values of its parameter C.

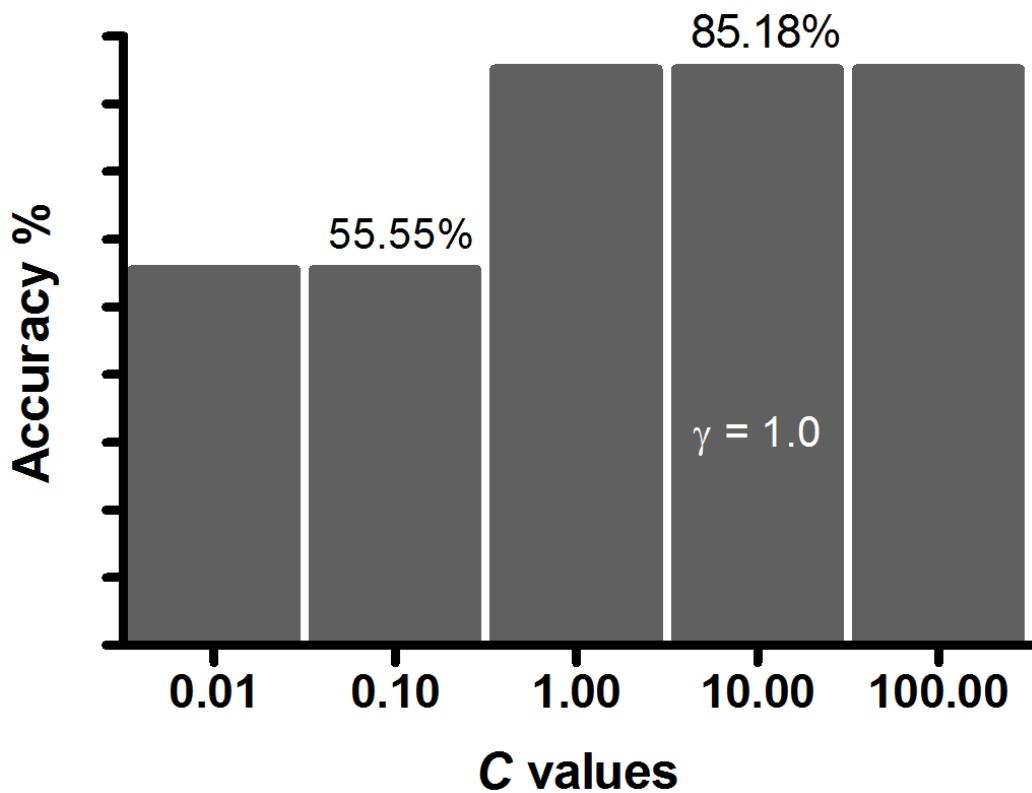


Figure 16.7: SVM optimization. Optimization of the parameters C and c of the SVM kernel RBF. Source Gomes et al. doi:10.1371/journal.pone.0011267.g003

• PRACTITIONER TIP •

To transform the gene expression data to a suitable format for support vector machine training and testing Gomes et al designate each gene as either “10” (for observation of up-regulation) or “01” (for observation of downregulation). Therefore, the collective gene expressions observed in each patient was represented by a 24-dimension vector (12 genes \times 2 gene states: up- or down-regulated). Each of the 24-dimension vectors was labeled as either “1” for DF patients or “-1” for DHF patients. Notice this is a different classification structure than that used in traditional statistical modeling of binary variables. Typically, binary observations are measured by Statisticians using 0 and 1. Be sure you have the correct classification structure when moving between traditional statistical models and those developed out of machine learning.

Forecasting Stock Market Direction

Huang et al⁴⁰ use a support vector machine to predict the direction of weekly changes in the NIKKEI 225 Japanese stock market index. The index is composed of 225 stocks of the largest Japanese publicly traded companies.

Two independent variables are selected as inputs to the model, weekly changes in the S&P500 index, and weekly changes in the US dollar - Japanese Yen exchange rate.

Data was collected from January 1990 to December 2002, yielding a total sample size of 676 observations. The researchers use 640 observations to train their support vector machine; and perform an out of sample evaluation on the remaining 36 observations.

As a benchmark the researchers compare the performance of their model to four other models, a random walk, linear discriminant analysis, quadratic discriminant analysis and a neural network.

The random walk correctly predicts the direction of the stock market 50% of the time, linear discriminant analysis 55%, quadratic discriminant analysis and the neural network 69% and the support vector machine 73%.

The researchers also observe that an information weighted combination of the models correctly predicts the direction of the stock market 75% of the time.

Bankruptcy Prediction

Min et al⁴¹ develop a support vector machine to predict bankruptcy and compare its performance to a neural network, logistic regression and multiple discriminant analysis.

Data on 1888 firms is collected from Korea's largest credit guarantee organization. The data set contains 944 bankruptcy and 944 surviving firms. The attribute set consisted of 38 popular financial ratios and was reduced by principal component analysis to two "fundamental" factors.

The training set consists of 80% of the observations with the remaining 20% of observations used for the hold out / test sample. A radial basis function is used for the kernel. Its parameters are optimized using a grid search procedure and 5- fold cross-validation.

In rank order (for the hold out data) the support vector machine had a prediction accuracy of 83%, the neural network 82.5%, multiple discriminant analysis 79.1% and the logistic regression 78.3%.

Early Onset Breast Cancer

Breast cancer is often classified according to the number of estrogen receptors present on the tumor. Tumors with a large numbers of receptors are termed estrogen receptor positive (ER+) and estrogen receptor negative (ER-) for few or no receptors. ER status is important because ER+ cancers grow under the influence of estrogen, and may respond well to hormone suppression treatments. This is not the case for ER- cancers as they do not respond to hormone suppression treatments.

Upstill-Goddard et al⁴² investigate whether patients who develop ER+ and ER- tumors show distinct constitutional genetic profiles using genetic single nucleotide polymorphisms data. At the core of their analysis was a support vector machines with linear, normalized quadratic polynomial, quadratic polynomial, cubic and radial basis kernels. The researchers opt for a 10- fold cross-validation.

All five kernel models had an accuracy rate in excess of 93%, see Table 11.

Kernal Type	%Correctly Classified
Linear	93.28 \pm 3.07
Normalized quadratic polynomial	93.69 \pm 2.69
Quadratic polynomial	93.89 \pm 3.06
Cubic polynomial	94.64 \pm 2.94
Radial basis function	95.95 \pm 2.61

Table 11: Upstill-Goddard et al's kernels and classification results.

Flood Susceptibility

Tehrany et al⁴³ evaluate support vector machines with different kernel functions for spatial prediction of flood occurrence in the Kuala Terengganu basin, Malaysia. Model attributes were constructed using ten geographic factors altitude, slope, curvature, stream power index (SPI), topographic wetness index (TWI), distance from the river, geology, land use/cover (LULC), soil, and surface runoff.

Four kernels, linear (LN), polynomial (PL), radial basis function (RBF), and sigmoid (SIG) were used to assess factor importance. This was achieved by eliminating the factor and then measuring the Cohen's kappa index of the model. The overall rank of each factor⁴⁴ is shown in Table 12. Overall, slope was the most important factor, followed by distance from river and then altitude.

Factor	Average	Rank
Altitude	0.265	3
Slope	0.288	1
Curvature	0.225	6
SPI	0.215	8
TWI	0.235	4
Distance	0.268	2
Geology	0.223	7
LULC	0.215	8
Soil	0.228	5
Runoff	0.140	10

Table 12: Variable importance calculated from data reported in Tehrany et al.

► PRACTITIONER TIP ◄

Notice how both Upstill-Goddard et al and Tehrany et al use multiple kernels in developing their models. This is always a good strategy because it is not always obvious which kernel is optimal at the onset of a research project.

Signature Authentication

Radhika et al⁴⁵ consider the problem of automatic signature authentication using a variety of algorithms including a support vector machine (SVM). The other algorithms considered included a Bayes classifier (BC), fast Fourier transform (FT), linear discriminant analysis (LD) and principal component analysis (PCA).

Their experiment used a signature database containing 75 subjects with 15 genuine samples and 15 forged samples for each subject. Features were extracted from images drawn from the database and used as inputs to train and test the various methods.

The researchers report a false rejection rate of 8% for SVM and 13% for FT, 10% for BC, 11% for PCA and 12% for LD.

Prediction of Vitamin D Status

The accepted bio marker of vitamin D status is serum 25-hydroxyvitamin D (25(OH)D) concentration. Unfortunately, in large epidemiological studies direct measurement is often not feasible. However, useful proxies for 25(OH)D are available by using questionnaire data.

Guo et al⁴⁶ develop a support vector machine to predict serum 25(OH)D concentration in large epidemiological studies using questionnaire data. A total of 494 participants were recruited onto the study and asked to complete a questionnaire which included sun exposure and sun protection behaviors, physical activity, smoking history, diet and the use of supplements. Skin types were defined by spectrophotometric measurements of skin reflectance to calculate melanin density for exposed skin sites (dorsum of hand, shoulder) and non-exposed skin sites (upper inner arm, buttock).

A multiple regression model (MLR) estimated using 12 explanatory variables⁴⁷ was used to benchmark the support vector machine. The researchers selected a radial basis function for the kernel with identical explanatory fac-

tors used in the MLR. The data were randomly assigned to a training sample ($n= 294$) and validation sample ($n= 174$).

The researchers report a correlation of 0.74 between predicted scores and measured 25(OH)D concentration for the support vector machine. They also note that it performed better than MLR in correctly identifying individuals with vitamin D deficiency. Overall, they conclude: “*RBF SVR [radial basis function support vector machine] method has considerable promise for the prediction of vitamin D status for use in chronic disease epidemiology and potentially other situations.*”

☛ PRACTITIONER TIP ☛

The performance of the SVM is very closely tied to the choice of the kernel function. There exist many popular kernel functions that have been widely used for classification e.g., linear, Gaussian radial basis function, polynomial and so on. Data Scientists can spend a considerable amount of time tweaking the parameters of a specified kernel function via trial-and-error. Here are four general approaches that can speed up the process:

1. Cross-validation⁴⁸.
2. Multiple kernel learning⁴⁹ attempts to construct a generalized kernel function so as to solve all classification problems through combining different types of standard kernel functions;
3. Evolution & Particle swarm optimization Thadani et al⁵⁰ use gene expression programming algorithms to evolve the kernel function of SVM. An analogous approach has been proposed using particle swarm optimization⁵¹.
4. Automatic kernel selection using the C5.0 algorithm which attempts to select the optimal kernel function based on the statistical data characteristics and distribution information.

Support Vector Classification

Technique 17

Binary Response Classification with C-SVM

A C-SVM for binary response classification can be estimated using the package `svmpath` with the `svmpath` function:

```
svmpath(x, y, kernel.function)
```

Key parameters include the response variable `y` coded as (-1,+1); the covariates `x`; and the specified kernel using `kernel.function`.

☛ PRACTITIONER TIP ☛

Although there are an ever growing number of kernels, four workhorses of applied research are:

- Linear: $K(x_i, x_j) = x_i^T x_j.$
- Polynomial: $K(x_i, x_j) = (\gamma x_i^T x_j + r)^d, \gamma > 0.$
- Radial basis function: $K(x_i, x_j) = \exp(-\gamma \|x_i - x_j\|^2), \gamma > 0.$
- Sigmoid: $\tanh(\gamma x_i^T x_j + r)$

Here γ , r and d are kernel parameter.

Step 1: Load Required Packages

We build the C-SVM for binary response classification using the data frame `PimaIndiansDiabetes2` contained in the `mlbench` package. For additional details on this data set see page 52.

```
> data("PimaIndiansDiabetes2", package = "mlbench")
> require(svmpath)
```

Step 2: Prepare Data & Tweak Parameters

The `PimaIndiansDiabetes2` has a large number of misclassified values (recorded as `NA`) particularly for the attributes of `insulin` and `triceps`. We remove these two attributes from the sample and use the `na.omit` method to remove any remaining misclassified values. The cleaned data is stored in `temp`.

```
> temp<-(PimaIndiansDiabetes2)
> temp$insulin  <- NULL
> temp$triceps <- NULL
> temp<-na.omit(temp)
```

The response `diabetes` is a factor containing the labels "`pos`" and "`neg`". However, the `svmpath` method requires the response to be numeric taking the values `-1` or `+1`. The following converts `diabetes` into a format usable by `svmpath`.

```
> y<-(temp$diabetes)
> levels(y) <- c("-1", "1")
> y<-as.numeric(as.character(y))
> y <-as.matrix(y)
```

Support vector machine kernels generally depend on the inner product of attribute vectors. Therefore very large values might cause numerical problems. We scale the attributes to lie in the range $[0,1]$ using the `scale` method the results are stored in the matrix `x`.

```
> x<-temp
> x$diabetes  <- NULL variable
> x<-scale(x)
```

We use `nrow` to measure the remaining observations (as a check it should equal 724). We then set the training sample to select at random without replacement 600 observations. The remaining 124 observations form the test sample.

```
> set.seed(103)
> n=nrow(x)
> train <- sample(1:n, 600, FALSE)
```

Step 3: Estimate & Evaluate Model

The `svmpath` function can use two popular kernels, the polynomial and radial basis function. We will assess both beginning with the polynomial kernel. It is selected by setting `kernel.function = poly.kernel`. We also set `trace=FALSE` to prevent `svmpath` from printing results to the screen at each iteration.

```
> fit<-svmpath(x[train,], y[train,], kernel.function
= poly.kernel, trace=FALSE)
```

A nice feature of `svmpath` is that it computes the entire regularization path for the SVM cost parameter along with the associated classification error. We use the `with` method to identify the minimum error.

```
> with(fit, Error[Error == min(Error)])
[1] 140 140 140 140 140 140 140 140 140 140 140 140 140
140 140
```

Two things are worth noting here, first the number of misclassified observations is 140 out of the 600 observations, which is approximately 23%, Second, each observation of 140 is associated with a unique regularization value. Since the regularization value is the penalty parameter of the error term (and `svmpath` reports the inverse of this parameter) we will select for our model the minimum value and store it in `lambda`. This is achieved via the following steps:

1. Store the minimum error values in `error`.
2. Grab the row numbers of these minimum errors using the `which` method.
3. Obtain the regularization parameter values associated with the minimum errors and store in `temp_lambda`.
4. Identify which value in `temp_lambda` has the minimum value, store it in `lambda` and print it to the screen.

```

> error<-with(fit, Error[Error == min(Error)])
> min_err_row<-which(fit$Error == min(fit$Error))
> temp_lambda<-fit$lambda[min_err_row]

> loc<-which(fit$lambda[min_err_row] == min(fit$lambda[min_err_row]))
> lambda<-temp_lambda[loc]
> lambda
[1] 73.83352

```

The method `svmpath` actually reports the inverse of the kernel regularization parameter (often and somewhat confusingly called gamma in the literature). We obtain a value of 73.83352 which corresponds to a gamma of $\frac{1}{73.83352} = 0.0135$.

Next we follow the same procedure for radial basis function kernel. In this case the estimated regularization parameter is stored in `lambdaR`.

```

> fitR<-svmpath(x[train,], y[train,], kernel =
+   function = radial.kernel, trace=FALSE)
> error<-with(fitR, Error[Error == min(Error)])

> min_err_row<-which(fitR$Error == min(fitR$Error))
> temp_lambda<-fitR$lambda[min_err_row]

> loc<-which(fitR$lambda[min_err_row] == min(fitR$lambda[min_err_row]))
> lambdaR<-temp_lambda[loc]

> lambdaR
[1] 0.09738556
> error[1]/600
[1] 0.015

```

Two things are noteworthy about this result. First, the regularization parameter is estimated as: $\frac{1}{0.09738556} = 10.268$; Second the error is estimated at only 1.5%. This is very likely an indication that the model has been over fit.

☛ PRACTITIONER TIP ☛

It often helps intuition to visualize data. Let's use a few lines of code to estimate a simple model and visualize it. We will fit a sub-set of the model we are already working on using the first twelve patient observations on the attributes of `glucose` and `age`, storing the result in `xx`. We standardize `xx` by using the `scale` method. Then we grab the first twelve observations of the response variable `diabetes`.

```
> xx<-cbind(temp$glucose[1:12],temp$age  
[1:12])  
> scale(xx)  
> yy<-y[1:12]
```

Next, we use the method `svmpath` to estimate the model and use `plot` to show the resultant data points. We use `step =1` to show the results at the first value of the regularization parameter and `step =8`, to show the results at the last step. The dotted line in Figure 17.1 represents the margin, notice it gets narrower from step 1 to 8 as the cost parameter increases. The support vectors are represented by the open dots. Notice that the number of support vectors increase as we move from step 1 to step 8. Also notice at step 8 only one point is misclassified (point 7).

```
> example <- svmpath(xx,yy,trace=TRUE,plot  
=FALSE)  
> par(mfrow = c(1, 2))  
> plot(example,xlab="glucose", ylab="age",  
step=1)  
> plot(example,xlab="glucose", ylab="age",  
step=8)
```

Step: 1 Errors: 2 Elbow Size: 2 Sum Eps: 1

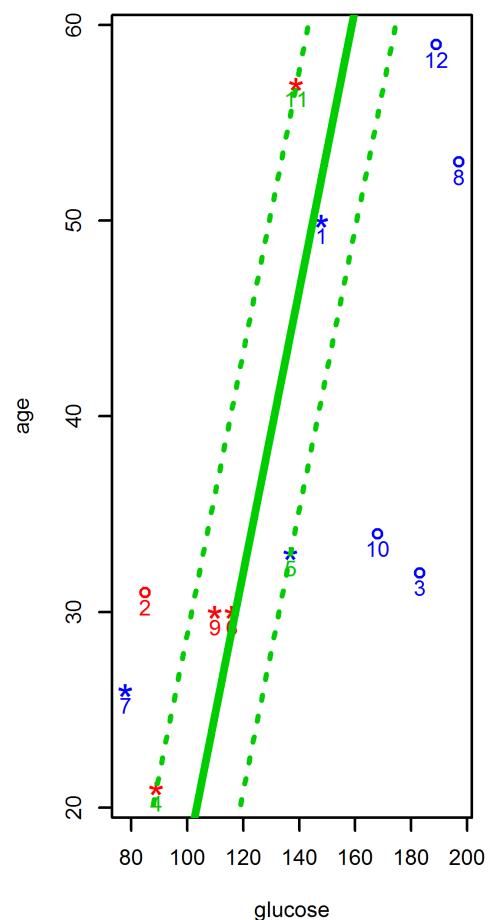
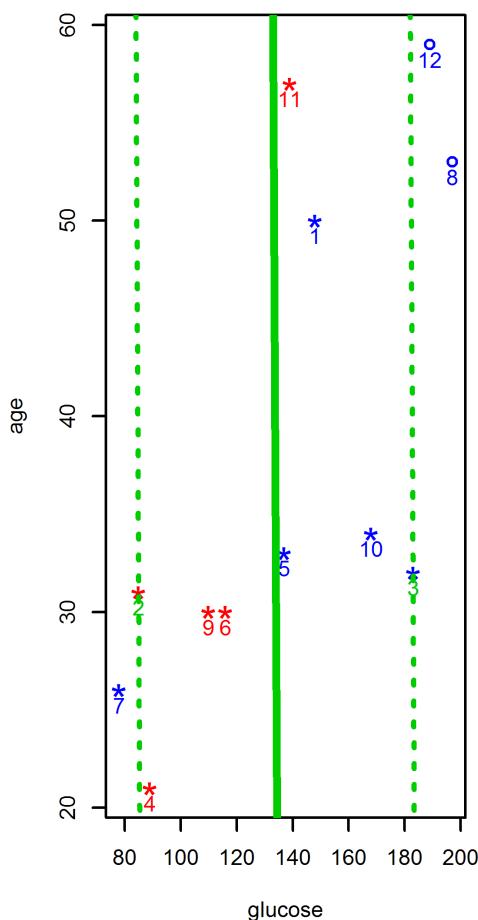


Figure 17.1: Plot of support vectors using `svmpath` with the response variable `diabetes` and attributes `glucose` & `age`

Step 4: Make Predictions

NOTE...

Automated choice of kernel regularization parameters is challenging. This is because it is extremely easy to over fit a SVM model on the validation sample if you only consider the misclassification rate. The consequence is that you end up with a model that is not generalizable to the test data and/or a model that performs considerably worse than the disguard models with higher test sample error rates⁵².

Although we suspect the radial basis kernel results in an over fit, we will compare its predictions to that of the optimal polynomial kernel. First, we use the test data and the radial basis kernel via the `predict` method. The confusion matrix is printed using the `table` method. The error rate is then calculated. It is 35% and considerably higher than the 1.5% indicated during validation!

```
> pred<-predict(fitR,newx=x[-train,], lambda=lambdaR  
+ ,type="class")  
  
> table( pred,y[-train,] , dnn =c("Predicted" , "  
Observed"))  
          Observed  
Predicted -1   1  
           -1 65 27  
            1 16 16  
> error_rate = (1- sum( pred == y[-train,] )/ 124)  
> round(error_rate ,2)  
[1] 0.35
```

• PRACTITIONER TIP •

Degradation in performance due to over-fitting can be surprisingly large! The key is to remember the primary goal of the validation sample is to provide a reliable indication of the expected error on the test sample and future as yet unseen samples. Throughout this book we have used the `set.seed()` method to help ensure replicability of the results. However, given the stochastic nature of the validation-test sample split we should always expect variation in performance for different realisations. This suggests that evaluation should always involve multiple partitions of the data to form training/ validation and test sets, as the sampling of data for a single partition might arbitrarily favour one classifier over another.

Let's now look at the polynomial kernel model.

```
> pred<-predict(fit,newx=x[-train,], lambda=lambda,
+ type="class")
>
> table( pred,y[-train,] ,   dnn =c("Predicted" , "Observed"))
            Observed
Predicted -1   1
           -1 76 13
           1  5 30
> error_rate = (1- sum( pred == y[-train,] )/ 124)

> round( error_rate ,2)
[1] 0.15
```

It seems the error rate for this choice of kernel is around 15%. This is less than half the error of the radial basis kernel SVM!

Technique 18

Multicategory Classification with C-SVM

A C-SVM for multicategory response classification can be estimated using the package `e1071` with the `svm` function:

```
svm(y ~ ., data, kernel, cost, ...)
```

Key parameters include `kernel` - the kernel function, `cost` - the cost parameter, multicategory response variable `y` and the covariates `data`.

Step 1: Load Required Packages

We build the C-SVM for multicategory response classification using the data frame `Vehicle` contained in the `mlbench` package. For additional details on this sample see page 23.

```
> require(e1071)
> library (mlbench)
> data(Vehicle)
```

Step 2: Prepare Data & Tweak Parameters

We use 500 out of the 846 observations in the training sample with the remaining saved for testing. The variable `train` selects the random sample without replacement from the 500 observations.

```
> set.seed(107)
> N=nrow(Vehicle)
> train <- sample(1:N, 500, FALSE)
```

Step 3: Estimate & Evaluate Model

We begin by estimating the support vector machine using the default settings. This will use a radial basis function as the kernel with a cost parameter value set equal to 1.

```
> fit<- svm(Class ~., data = Vehicle[train,])  
The summary function provides details of the estimated model:  
> summary(fit)  
  
Call:  
svm(formula = Class ~ ., data = Vehicle[train, ])  
  
Parameters:  
  SVM-Type: C-classification  
  SVM-Kernel: radial  
    cost: 1  
   gamma: 0.05555556  
  
Number of Support Vectors: 376  
  
( 122 64 118 72 )  
  
Number of Classes: 4  
  
Levels:  
bus opel saab van
```

The function provides details of the type of support vector machine (C-classification), kernel, cost parameter and gamma model parameter. Notice the model estimates 376 support vectors, with 122 in the first class (**bus**), and 72 in the fourth class (**van**).

A nice feature of the **e1071** package is that contains **tune.svm**, a support vector machine tuning function. We use the method to identify the best model for **gamma** ranging between 0.25 and 4, and **cost** between 4 and 16. We store the results in **obj**.

```
> obj <- tune.svm(Class~., data = Vehicle[train,],  
  gamma = 2^(-2:2), cost = 2^(2:4))
```

Once again we can use the **summary** function to see the result.

```
> summary(obj)
```

Parameter tuning of svm:

- sampling method: 10-fold cross validation
- best parameters:
`gamma cost
0.25 16`
- best performance: 0.254

The method automatically performs a 10-fold cross validation. It appears the best model has a `gamma` of 0.25 and a `cost` parameter equal to 16 with 25.4% misclassification rate.

☛ PRACTITIONER TIP ☛

Greater control over model tuning can be achieved using the `tune.control` method inside of `tune.svm`. For example, to perform a 20-fold cross validation you would use

```
tunecontrol = tune.control(sampling = "cross", cross=20)
```

Your call using `tune.svm` would look something like this:

```
obj <- tune.svm(Class~, data = Vehicle[  
  train,], gamma = 2^(-2:2), cost =  
  2^(2:4), tunecontrol = tune.control(  
  sampling = "cross", cross=20))
```

Visualization of the output of `tune.svm` using `plot` is often useful for fine tuning.

```
> plot(obj)
```

Figure 18.1 illustrates the resultant plot. To interpret the image note that the darker the shading, the better the fit of the model. Two things are noteworthy about this image. First, a larger cost parameter seems to indicate a better fit; Second, a `gamma` of 0.5 or less also indicates a better fitting model.

Using this information we re-tune the model with `gamma` ranging from 0.01 to 0.5 and `cost` ranging from 16 to 256. Now, the best performance occurs

with `gamma` set to 0.03 with `cost` equal to 32.

```
> obj <- tune.svm(Class~, data = Vehicle[train,],  
+ gamma = seq(0.01,0.5,by=0.01), cost = 2^(4:8))  
> summary(obj)
```

Parameter tuning of svm:

- sampling method: 10-fold cross validation
- best parameters:

<code>gamma</code>	<code>cost</code>
0.03	32
- best performance: 0.17

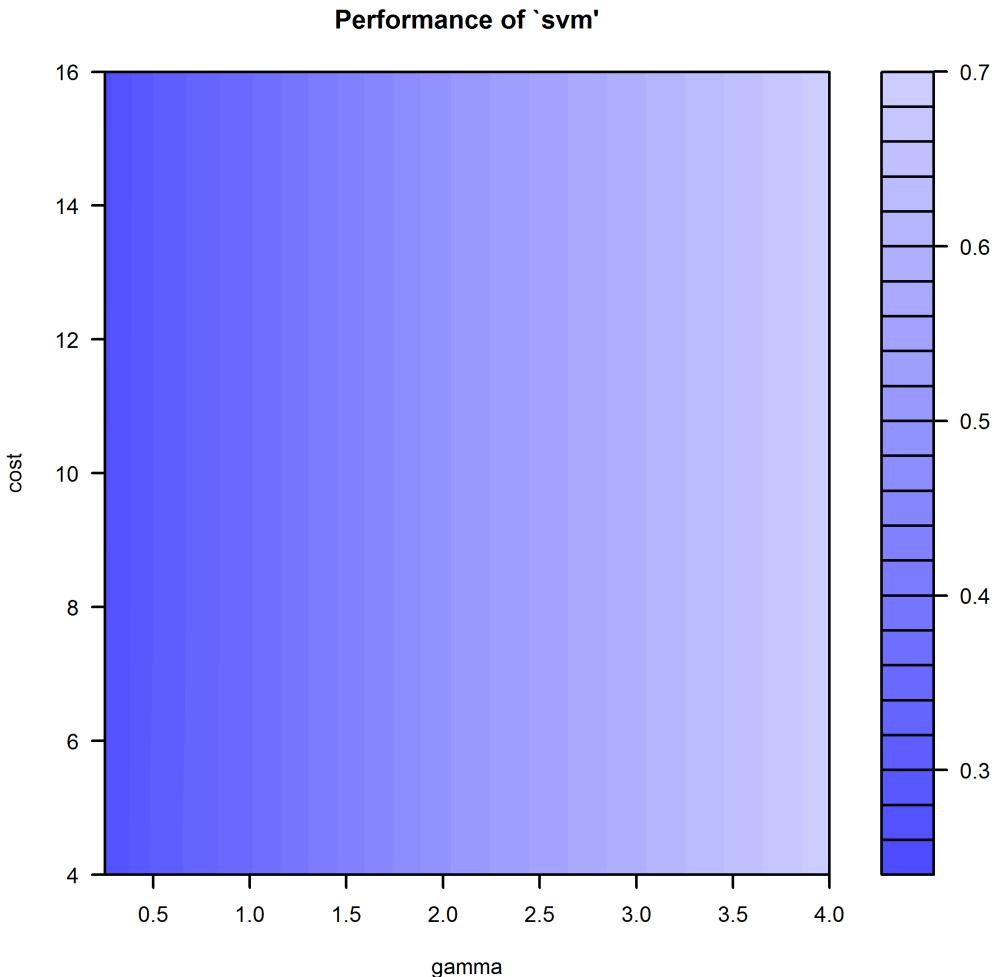


Figure 18.1: Tuning Multicategory Classification with C-SVM using Vehicle data set

We store the results for the optimal model in the objects `bestC` and `bestGamma`; and refit the model:

```
> bestC <- obj$best.parameters[[2]]  
> bestGamma<-obj$best.parameters[[1]]  
> fit<- svm(Class ~., data = Vehicle[train,], cost=  
bestC, gamma=bestGamma, cross=10)
```

Details of the `fit` can be viewed using the `print` method.

```
> print(fit)
```

```
Call:  
svm(formula = Class ~ ., data = Vehicle[train, ],  
    cost = bestC, gamma = bestGamma,  
    cross = 10)
```

Parameters:

```
SVM-Type: C-classification  
SVM-Kernel: radial  
cost: 32  
gamma: 0.03
```

Number of Support Vectors: 262

The fitted model now has 262 support vectors, down from 376 for the original model.

The `summary` method provides additional details. Reported are the support vectors by classification, and the results from the 10-fold cross validation. Overall, the model has a total accuracy of 81%.

```
> summary(fit)
```

```
Call:  
svm(formula = Class ~ ., data = Vehicle[train, ],  
    cost = bestC, gamma = bestGamma,  
    cross = 10)
```

Parameters:

```
SVM-Type: C-classification  
SVM-Kernel: radial  
cost: 32  
gamma: 0.03
```

Number of Support Vectors: 262

(100 32 94 36)

Number of Classes: 4

Levels:

```
bus opel saab van

10-fold cross-validation on training data:

Total Accuracy: 81
Single Accuracies:
 84 86 88 76 82 78 76 84 82 74
```

It can be fun to visualize a two dimensional projection of the fitted data. To do this we use the `plot` function with variables `Elong` and `Max.L.Ra` for the x and y axis whilst holding all the other variables in `Vehicle` at their median value. Figure 18.2 shows the resultant plot.

```
> plot(fit, Vehicle[train,], Elong ~ Max.L.Ra,
  svSymbol = "v" , slice = list(
  Comp = median(Vehicle$Comp),
  Circ = median(Vehicle$Circ),
  D.Circ = median(Vehicle$D.Circ),
  Rad.Ra = median(Vehicle$Rad.Ra),
  Pr.Axis.Ra = median (Vehicle$Pr.Axis.Ra)

  ,
  Scat.Ra=median(Vehicle$Scat.Ra) ,
  Pr.Axis.Rect =median(Vehicle$Pr.Axis.Rect) ,
  Max.L.Rect =median(Vehicle$Max.L.Rect ) ,
  Sc.Var.Maxis =median(Vehicle$Sc.Var.Maxis) ,
  Sc.Var.maxis =median(Vehicle$Sc.Var.maxis) ,
  Ra.Gyr=median(Vehicle$Ra.Gyr) ,
  Skew.Maxis =median(Vehicle$Skew.Maxis) ,
  Skew.maxis =median(Vehicle$Skew.maxis) ,
  Kurt.maxis =median(Vehicle$Kurt.maxis),
  Kurt.Maxis =median(Vehicle$Kurt.Maxis) ,
  Holl.Ra =median(Vehicle$Holl.Ra )
))
```

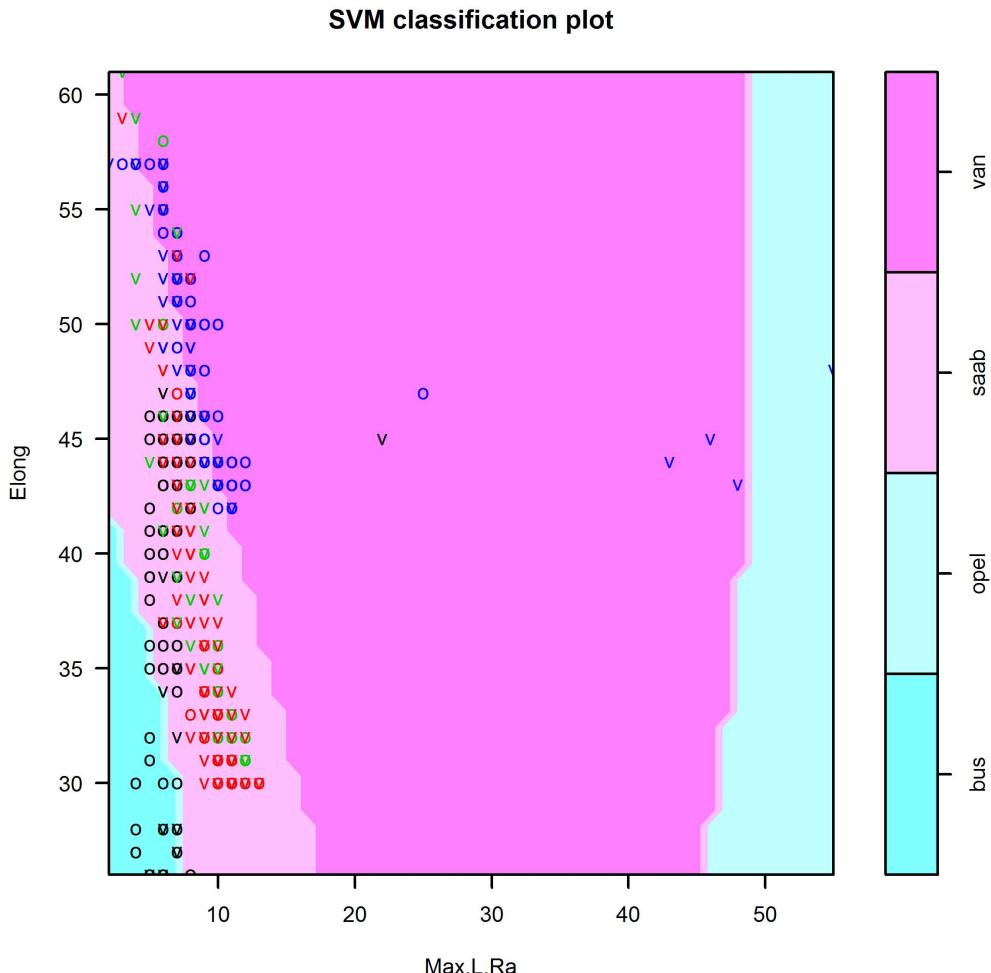


Figure 18.2: Multicategory Classification with C-SVM two dimensional projection of Vehicle.

Step 4: Make Predictions

The `predict` method with the test data and the fitted model fit are used as follows.

```
> pred <- predict(fit, Vehicle[-train,])
```

Classification results alongside the actual observed values can be obtained using the `table` method.

```
> table(Vehicle$Class[-train], pred, dnn=c( "Observed Class", "Predicted Class" ))
```

		Predicted Class			
Observed Class	bus	opel	saab	van	
bus	87	2	0	5	
opel	0	55	28	2	
saab	0	20	68	0	
van	1	1	1	76	

The error / misclassification rate can then be calculated. Overall, the model achieves an error rate of 17% on the test sample.

```
> error_rate = (1 - sum(pred == Vehicle$Class[-train])) /  
  346)  
> round(error_rate, 3)  
[1] 0.173
```

Technique 19

Multicategory Classification with nu-SVM

A nu-SVM for multicategory response classification can be estimated using the package `e1071` with the `svm` function:

```
svm(y ~ ., data, kernel, type="nu-classification" ...)
```

Key parameters include `kernel` - the kernel function, `type` set to "nu-classification", multicategory response variable `y` and the covariates `data`.

Step 3: Estimate & Evaluate Model

Step 1 and 2 are outlined beginning on page 134.

We estimate the support vector machine using the default settings. This will use a radial basis function as the kernel with a `nu` parameter value set equal to 0.5.

```
> fit<- svm(Class ~ ., data = Vehicle[train,], type="nu-classification")
```

The `summary` function provides details of the estimated model:

```
> summary(fit)
```

`Call:`

```
svm(formula = Class ~ ., data = Vehicle[train, ],  
    type = "nu-classification")
```

Parameters:

```
SVM-Type: nu-classification
SVM-Kernel: radial
gamma: 0.05555556
nu: 0.5
```

Number of Support Vectors: 403

```
( 110 88 107 98 )
```

Number of Classes: 4

Levels:

```
bus opel saab van
```

The function provides details of the type of support vector machine (nu-classification), kernel, nu parameter and gamma parameter. Notice the model estimates 403 support vectors, with 110 in the first class (**bus**), and 98 in the fourth class (**van**). The total number of support vectors is slightly higher than estimated in the C-SVM discussed on page 134.

We use the **tune.svm** method to identify the best model for **gamma** and **nu** ranging between 0.05 and 0.45. We store the results in **obj**.

```
> obj <- tune.svm(Class~, data = Vehicle[train,],
+ type="nu-classification",
+ gamma = seq(0.05,0.45,by=0.1),
+ nu=seq(0.05,0.45,by=0.1))
```

We can use the **summary** function to see the result.

```
> summary(obj)
```

Parameter tuning of svm:

- sampling method: 10-fold cross validation
- best parameters:
gamma **nu**
0.05 0.05
- best performance: 0.178

The method automatically performs a 10-fold cross validation. We see that the best model has a `gamma` and `nu` equal to 0.05 with a 17.8% misclassification rate. Visualization of the output of `tune.svm` can be achieved using the `plot` function.

```
> plot(obj)
```

Figure 19.1 illustrates the resultant plot. To interpret the image note that the darker the shading, the better the fit of the model. It seems that a smaller `nu` and `gamma` lead to a better fit of the training data.

Using this information we re-tune the model with `gamma` and `nu` ranging from 0.01 to 0.05. The best performance occurs with both parameters set to 0.02, and an overall misclassification error rate of 16.4%.

```
> obj <- tune.svm(Class~, data = Vehicle[train,],  
+ type="nu-classification",  
+ gamma = seq(0.01,0.05,by=0.01),  
+ nu=seq(0.01,0.05,by=0.01))  
  
> summary(obj)
```

Parameter tuning of svm:

- sampling method: 10-fold cross validation
- best parameters:
`gamma` `nu`
0.02 0.02
- best performance: 0.164

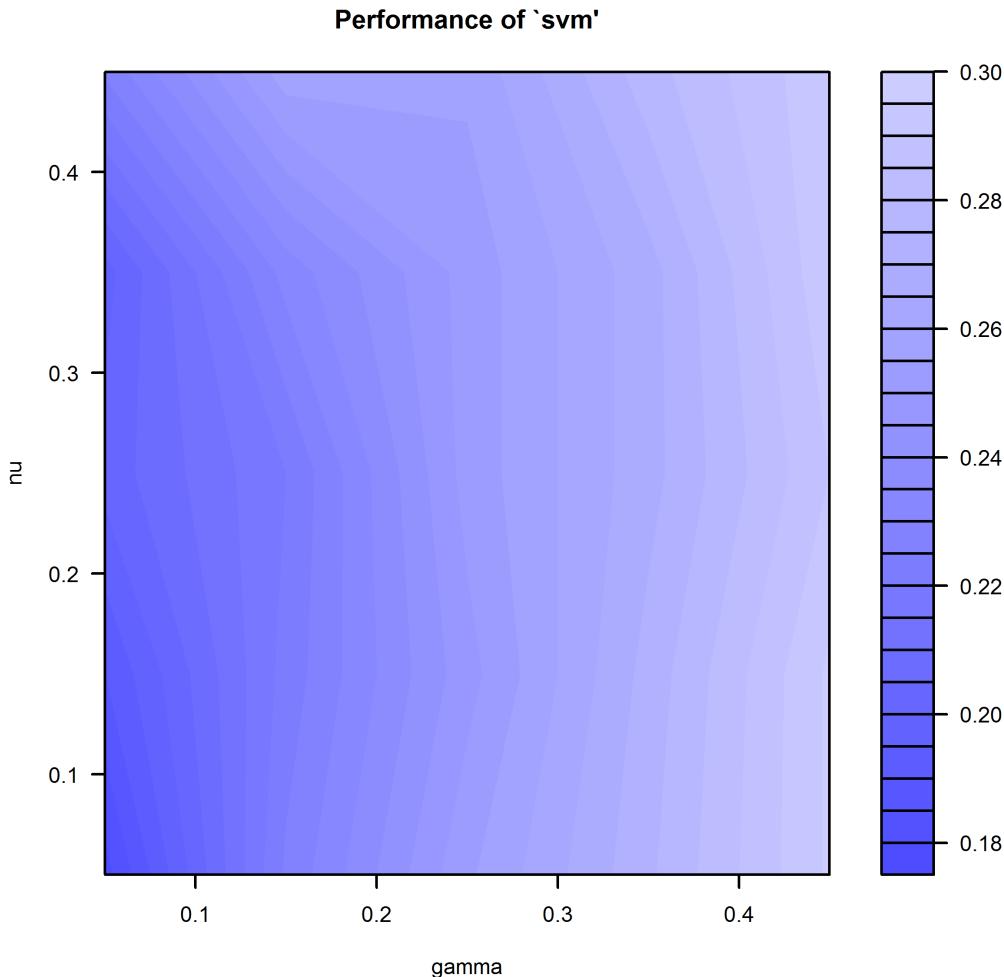


Figure 19.1: Tuning Multicategory Classification with NU-SVM using **Vehicle** data set

We store the results for the optimal model in the objects `bestC` and `bestGamma` and then refit the model:

```
> bestNU <- obj$best.parameters[[2]]  
> bestGamma<-obj$best.parameters[[1]]  
>fit<- svm(Class ~., data = Vehicle[train,],  
type="nu-classification",  
nu=bestNU,  
gamma=bestGamma ,cross=10)
```

Details of the `fit` can be viewed using the `print` method.

```
print(fit)

Call:
svm(formula = Class ~ ., data = Vehicle[train, ],
     type = "nu-classification",
     nu = bestNU, gamma = bestGamma, cross = 10)

Parameters:
  SVM-Type: nu-classification
  SVM-Kernel: radial
  gamma: 0.02
  nu: 0.02
```

Number of Support Vectors: 204

The fitted model now has 204 support vectors, less than half required for the model we initially build.

The **summary** method provides additional details. Reported are the support vectors by classification, and the results from the 10-fold validation. Overall, the model has a total accuracy of 75.6%.

```
> summary(fit)
```

```
Call:
svm(formula = Class ~ ., data = Vehicle[train, ],
     type = "nu-classification",
     nu = bestNU, gamma = bestGamma, cross = 10)
```

Parameters:

```
  SVM-Type: nu-classification
  SVM-Kernel: radial
  gamma: 0.02
  nu: 0.02
```

Number of Support Vectors: 204

(71 29 72 32)

Number of Classes: 4

Levels:

bus opel saab van

10-fold cross-validation on training data:

Total Accuracy: 75.6

Single Accuracies:

82 80 84 80 72 64 74 76 76 68

Next we visualize a two dimensional projection of the fitted data. To do we use the `plot` function with variables `Elong` and `Max.L.Ra` for the x and y axis whilst holding all the other variables in `Vehicle` at their median value. Figure 19.2 shows the resultant plot.

```
> plot(fit,Vehicle[train,],Elong ~ Max.L.Ra,
  svSymbol = "v" ,slice = list(
  Comp = median(Vehicle$Comp),
  Circ = median(Vehicle$Circ),
  D.Circ = median(Vehicle$D.Circ),
  Rad.Ra = median(Vehicle$Rad.Ra),
  Pr.Axis.Ra = median (Vehicle$Pr.Axis.Ra)
  ,
  Scat.Ra=median(Vehicle$Scat.Ra) ,
  Pr.Axis.Rect =median(Vehicle$Pr.Axis.Rect) ,
  Max.L.Rect =median(Vehicle$Max.L.Rect) ,
  Sc.Var.Maxis =median(Vehicle$Sc.Var.Maxis) ,
  Sc.Var.maxis =median(Vehicle$Sc.Var.maxis) ,
  Ra.Gyr=median(Vehicle$Ra.Gyr) ,
  Skew.Maxis =median(Vehicle$Skew.Maxis) ,
  Skew.maxis =median(Vehicle$Skew.maxis) ,
  Kurt.maxis =median(Vehicle$Kurt.maxis),
  Kurt.Maxis =median(Vehicle$Kurt.Maxis) ,
  Holl.Ra =median(Vehicle$Holl.Ra)
))
```

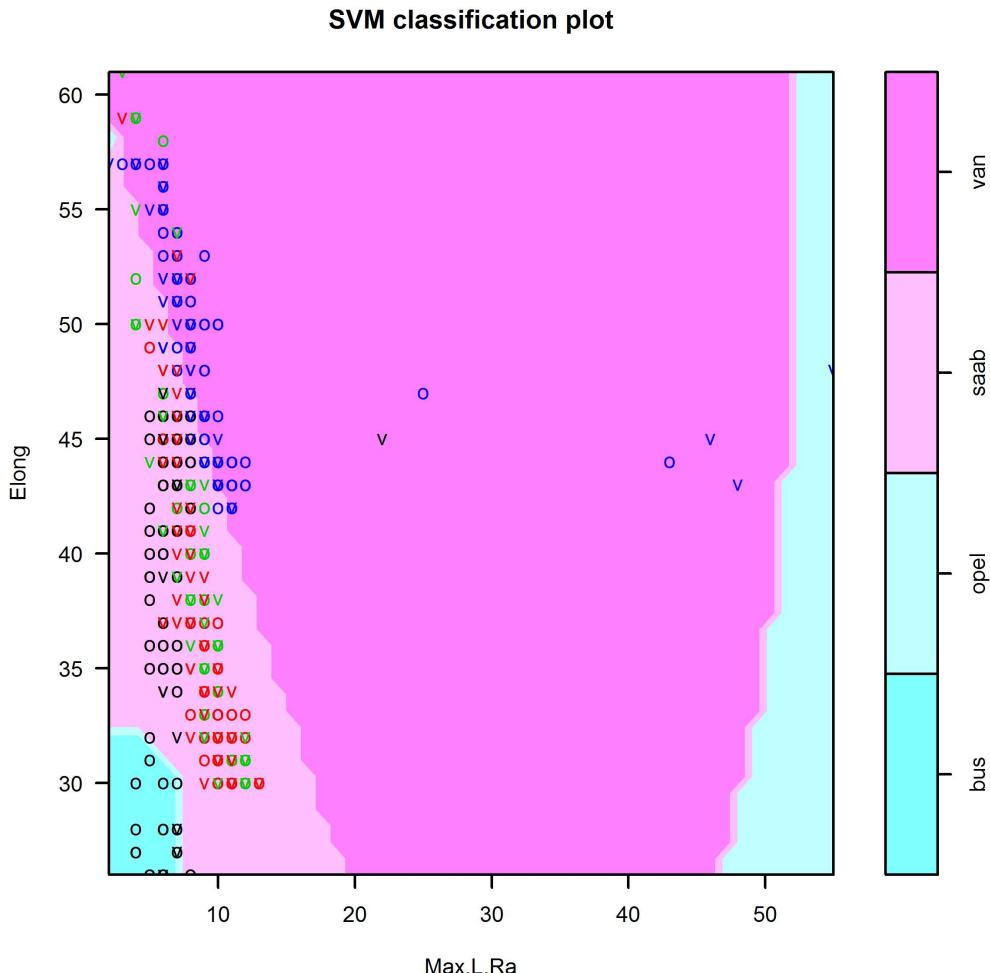


Figure 19.2: Multicategory Classification with NU-SVM two dimensional projection of Vehicle

Step 4: Make Predictions

The `predict` method with the test data and the fitted model fit are used as follows.

```
> pred <- predict(fit, Vehicle[-train,])
```

Classification results alongside the actual observed values can be obtained using the `table` method.

```
> table(Vehicle$Class[-train],  
pred, dnn=c( "Observed Class", "Predicted Class" ))
```

		Predicted Class			
Observed Class	bus	opel	saab	van	
bus	87	1	1	5	
opel	0	41	42	2	
saab	0	30	58	0	
van	1	1	2	75	

The error / misclassification rate can then be calculated. Overall, the model achieves an error rate of 24.6% on the test sample.

```
> error_rate = (1 - sum(pred == Vehicle$Class[-train])) /  
  346)  
> round(error_rate, 3)  
[1] 0.246
```

Technique 20

Bound-constraint C-SVM classification

A bound-constraint C-SVM for multicategory response classification can be estimated using the package `kernlab` with the `ksvm` function:

```
ksvm(y ~ ., data, kernel, type="C-bsvc", kpar, ...)
```

Key parameters include `kernel` - the kernel function, `type` set to "C-bsvc", `kpar` which contains parameter values, multicategory response variable `y` and the covariates `data`.

Step 1: Load Required Packages

We build the bound-constraint C-SVM for multicategory response classification using the data frame `Vehicle` contained in the `mlbench` package. For additional details on this sample see page 23. The `scatterplot3d` packaged will be used to create a three dimensional scatter plot.

```
> library(kernlab)
> library(mlbench)
> data(Vehicle)
> library(scatterplot3d)
```

Step 2: Prepare Data & Tweak Parameters

We use 500 out of the 846 observations in the training sample with the remaining saved for testing. The variable `train` selects the random sample without replacement from the 500 observations.

```
> set.seed(107)
> N=nrow(Vehicle)
> train <- sample(1:N, 500, FALSE)
```

Step 3: Estimate & Evaluate Model

We estimate the bound-constraint support vector machine using a radial basis kernel (`type = rbfdot`) and parameter `sigma` equal to 0.05. We also set the cross validation parameter `cross =10`.

```
> fit<- ksvm(Class ~ ., data = Vehicle[train,],
  type="C-bsvc",kernel=rbfdot,kpar=list(sigma=0.05),
  cross=10)
```

The `print` function provides details of the estimated model:

```
> print(fit)
Support Vector Machine object of class "ksvm"

SV type: C-bsvc (classification)
parameter : cost C = 1

Gaussian Radial Basis kernel function.
Hyperparameter : sigma = 0.05

Number of Support Vectors : 375

Objective Function Value : -49.1905 -53.3552
                           -48.0924 -193.2145 -48.9507 -59.7368
Training error : 0.164
Cross validation error : 0.248
```

The function provides details of the type of support vector machine (C-bsvc), model cost parameter and kernel parameter (`C=1, sigma =0.05`). Notice the model estimates 375 support vectors, with a training error of 16.4% and cross validation error of 24.8%. In this case we observe a relatively large difference between the training error and cross validation error. In practice the cross validation error is often a better indicator of the expected performance on the test sample.

Since the output of `ksvm` (in our example values stored in `fit`) is an S4 object, both errors can be accessed directly using “@”. Although the “preferred” approach is to use an accessor function such as `cross(fit)` and

`error(fit)`. If you don't know the accessor functions names you can always use `attributes(fit)` and access the required parameter using @ (for S4 objects) or \$ (for S3 objects) .

```
> fit@error  
[1] 0.164
```

```
> fit@cross  
[1] 0.248
```

☛ PRACTITIONER TIP ☛

The package `Kernlab` supports a wide range of kernels. Here are nine popular choices:

- `rbfdot` - Radial Basis kernel function
- `polydot` - Polynomial kernel function
- `vanilladot` - Linear kernel function
- `tanhdot` - Hyperbolic tangent kernel function
- `laplacedot` - Laplacian kernel function
- `besseldot` - Bessel kernel function
- `anovadot` - ANOVA RBF kernel function
- `splinedot` - Spline kernel
- `stringdot` - String kernel

We need to tune the model to obtain the optimum parameters. Let's create a few lines of R code to do this for us. First we set up the ranges for the `cost` and `sigma` parameters.

```
> cost <- 2^(2:8)  
> sigma<-seq(0.1,0.5,by=0.1)  
> n_cost<-length(cost)  
> n_sigma<-length(sigma)
```

The total number of models to be estimated is stored in `runs`. Check to see if it contains the value 35 (it should!).

```
> runs<-n_sigma*n_cost
```

```
> count.cost <- 0  
> count.sigma<-0  
> runs  
[1] 35
```

The result in terms of cross validation error, `cost` and `sigma` are stored in `results`.

```
> results<-1:(3*runs  
> dim(results) <- c(runs,3)  
> colnames(results) <- c("cost","sigma","error")
```

The objects `i`, `j` and `count` are loop variables.

```
> i=1  
> j=1  
> count=1
```

The main loop for tuning is as follows.

```
> for (val in cost) {  
  
  for (val in sigma) {  
    cat("iteration = ", count, " out of ", runs, "\n")  
  
    fit<- ksvm(Class ~ ., data = Vehicle[train,],  
    type="C-bsvc",  
    C=cost[j],  
    kernel=rbfdot,  
    kpar=list(sigma=sigma[i]),  
    cross=45)  
  
    results[count,1]=fit@param$C  
    results[count,2]=sigma[i]  
    results[count,3]= fit@cross  
    count.sigma = count.sigma+1  
    count=count+1  
  
    i=i+1  
  } # end sigma loop  
  
  i=1  
  j=j+1  
  
}# end cost loop
```

Notice we set `cross = 45` to perform leave one out validation.

When you execute the above code, as it is running, you should see output along the lines of:

```
iteration = 1 out of 35
iteration = 2 out of 35
iteration = 3 out of 35
iteration = 4 out of 35
iteration = 5 out of 35
```

We turn `results` into a data frame using the `as.data.frame` method.

```
> results<-as.data.frame(results)
```

Take a peek and you should see something like this.

```
> results
  cost sigma      error
1     4   0.1 0.2358586
2     4   0.2 0.2641414
3     4   0.3 0.2744108
4     4   0.4 0.2973064
```

Now let's find the best cross validation performance and its associated row number.

```
> with(results, error[error == min(error)])
[1] 0.2074074
```

```
> which(results$error== min(results$error))
[1] 11
```

The optimal values may need to be stored for later use. We save them in `best_result` using a few lines of code.

```
> best_per_row<-which(results$error== min(results$error))

> fit_cost<-results[best_per_row,1]
> fit_sigma<-results[best_per_row,2]

> fit_xerror<-results[best_per_row,3]
> best_result<-cbind(fit_cost,fit_sigma,fit_xerror)

> colnames(best_result)<- c("cost","sigma","error")
> best_result
```

```
cost sigma      error
[1,]    16   0.1  0.2074074
```

So we see the optimal results occur for `cost =16`, `sigma = 0.1` with a cross validation error of 20.7%.

Figure 20.1 presents a three dimensional visualization of the tuning numbers. It was created using `scatterplot3d` as follows:

```
> scatterplot3d(results$cost ,
  results$sigma ,
  results$error ,
  xlab="cost",
  ylab="sigma",
  zlab="Error")
```

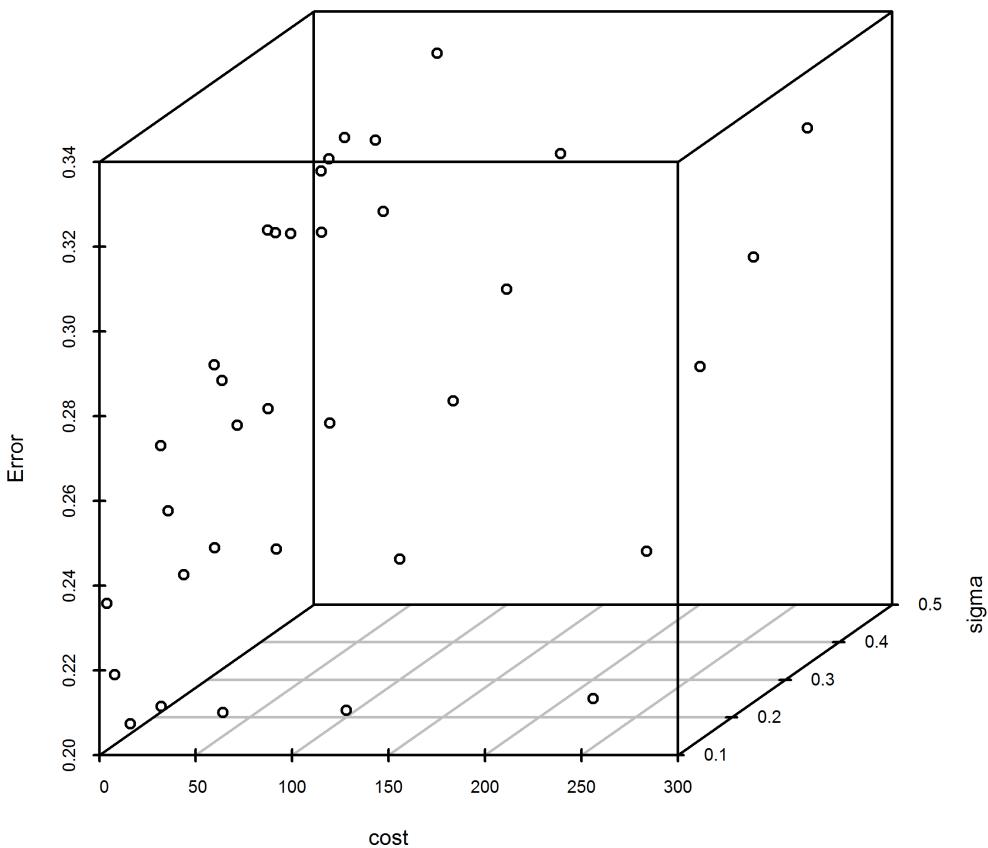


Figure 20.1: Bound-constraint C-SVM tuning 3d scatterplot using `Vehicle`

After all that effort we may as well estimate the optimal model using the training data and show the cross validation error. It is around 24.5%.

```
> fit<- ksvm(Class ~ ., data = Vehicle[train,],
  type="C-bsvc",
  cost=fit_cost,
  kernel=rbfdot,
  kpar=list(sigma=fit_sigma),
  cross=45)

> fit@cross
[1] 0.2457912
```

Step 4: Make Predictions

The `predict` method with the test data and the fitted model fit are used as follows.

```
> pred <- predict(fit, Vehicle[-train,])
```

Classification results alongside the actual observed values can be obtained using the `table` method.

```
> table(Vehicle$Class[-train],  
pred,  
dnn=c( "Observed Class","Predicted Class" ))
```

		Predicted Class			
Observed Class	bus	opel	saab	van	
bus	88	1	2	3	
opel	1	31	50	3	
saab	1	24	61	2	
van	1	0	3	75	

The error / misclassification rate can then be calculated. Overall, the model achieves an error rate of 26.3% on the test sample.

```
> error_rate = (1-sum(pred== Vehicle$Class[-train]))/  
346)  
> round(error_rate,3)  
[1] 0.263
```

Technique 21

Weston - Watkins Multi-Class SVM

A bound-constraint Weston - Watkins SVM for multiclass response classification can be estimated using the package `kernlab` with the `ksvm` function:

```
ksvm(y ~ ., data, kernel, type = "kbb-svc", kpar, ...)
```

Key parameters include `kernel` - the kernel function, `type` set to "`kbb-svc`", `kpar` which contains parameter values, multiclass response variable `y` and the covariates `data`.

Step 3: Estimate & Evaluate Model

Step 1 and step 2 are outlined beginning on page 151.

We estimate the Weston - Watkins support vector machine using a radial basis kernel (`type = rbf`) and parameter `sigma` equal to 0.05. We also set the cross validation parameter `cross = 10`.

```
fit <- ksvm(Class ~ ., data = Vehicle[train,],  
            type = "kbb-svc",  
            kernel = rbf,  
            kpar = list(sigma = 0.05),  
            cross = 10)
```

The `print` function provides details of the estimated model:

```
> print(fit)  
Support Vector Machine object of class "ksvm"
```

```
SV type: kbb-svc  (classification)
parameter : cost C = 1

Gaussian Radial Basis kernel function.
Hyperparameter : sigma = 0.05

Number of Support Vectors : 356

Objective Function Value : 0
Training error : 0.148
Cross validation error : 0.278
```

The function provides details of the type of support vector machine (kbb-svc), model cost parameter, kernel type and parameter (`sigma =0.05`), number of support vectors (356), training error of 14.8% and cross validation error of 27.8%. Both types of error can be individually accessed as follows:

```
> fit@error
[1] 0.148

> fit@cross
[1] 0.278
```

Let's see if we can tune the model using our own grid search. First we set up the ranges for the `cost` and `sigma` parameters.

```
> cost <- 2^(2:8)
> sigma<-seq(0.1,0.5,by=0.1)
> n_cost<-length(cost)
> n_sigma<-length(sigma)
```

The total number of models to be estimated is stored in `runs`.

```
> runs<-n_sigma*n_cost
> count.cost <- 0
> count.sigma<-0
```

The results in terms of cross validation error, `cost` and `sigma` are stored in `results`.

```
> results<-1:(3*runs)
> dim(results) <- c(runs,3)
> colnames(results) <- c("cost","sigma","error")
```

The objects `i`, `j` and `count` are loop variables.

```
> i=1
```

```
> j=1
> count=1

The main loop for tuning is as follows.

> for (val in cost) {
  for (val in sigma) {
    cat("iteration = ", count , " out of ", runs , "\n")

    fit<- ksvm(Class ~ ., data = Vehicle[train,],
type="kbb-svc",
C=cost[j],
kernel=rbfdot,
kpar=list(sigma=sigma[i]),
cross=45)

    results[count,1]=fit@param$C
    results[count,2]=sigma[i]
    results[count,3]= fit@cross

    count.sigma = count.sigma+1
    count=count+1

    i=i+1
  } # end sigma loop

  i=1
  j=j+1

}# end cost loop
```

When you execute the above code, as it is running, you should see output along the lines of:

```
iteration = 1 out of 35
iteration = 2 out of 35
iteration = 3 out of 35
iteration = 4 out of 35
iteration = 5 out of 35
```

We turn `results` into a data frame using the `as.data.frame` method.

```
> results<-as.data.frame(results)
```

Take a peek, you should see something like this.

```
> results
  cost sigma      error
1     4   0.1 0.2356902
2     4   0.2 0.2639731
3     4   0.3 0.3228956
4     4   0.4 0.2973064
5     4   0.5 0.2885522
```

Now let's find the best cross validation performance and its associated row number.

```
> with(results, error[error == min(error)])
[1] 0.2122896
```

```
> which(results$error== min(results$error))
[1] 21
```

So the optimal cross validation error is 21% and located in row 21 of `results`.

The optimal values may need to be stored for later use. We save them in `best_result` using a few lines of code.

```
> best_per_row<-which(results$error== min(results$error))

> fit_cost<-results[best_per_row,1]
> fit_sigma<-results[best_per_row,2]
> fit_xerror<-results[best_per_row,3]

> best_result<-cbind(fit_cost,fit_sigma,fit_xerror)
> colnames(best_result)<- c("cost","sigma","error")

> best_result
  cost sigma      error
[1,]    64   0.1 0.2122896
```

Figure 21.1 presents a three dimensional visualization of the tuning numbers. It was created using `scatterplot3d` as follows:

```
> scatterplot3d(results$cost,
  results$sigma,
  results$error,
  xlab="cost",
  ylab="sigma",
  zlab="Error")
```

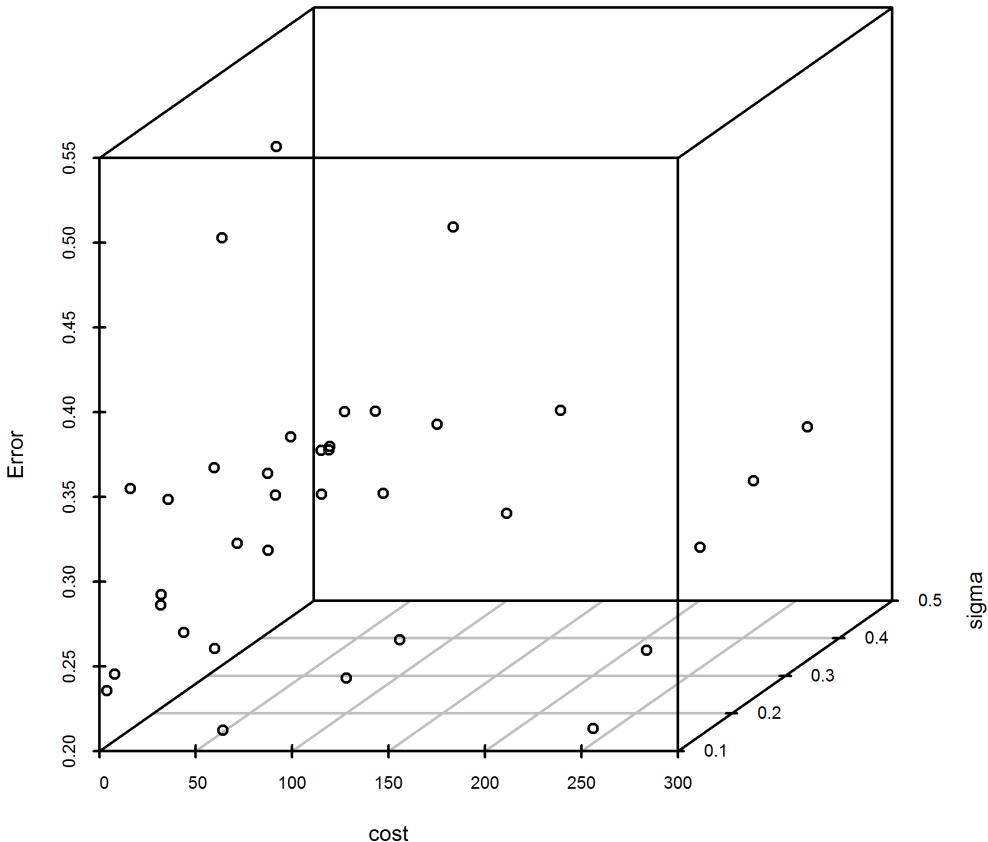


Figure 21.1: Weston - Watkins Multi-Class SVM tuning 3d scatterplot using Vehicle

After all that effort we may as well estimate the optimal model using the training data and show the cross validation error. It is around 27%.

```
> fit<- ksvm(Class ~ ., data = Vehicle[train,],  
+ type="kbb-svc",  
+ cost=fit_cost,  
+ kernel=rbfdot,  
+ kpar=list(sigma=fit_sigma),  
+ cross=45)  
  
> fit@cross
```

```
[1] 0.2762626
```

Step 4: Make Predictions

The `predict` method with the test data and the fitted model fit are used as follows.

```
> pred <- predict(fit, Vehicle[-train,])
```

Classification results alongside the actual observed values can be obtained using the `table` method.

```
> table(Vehicle$Class[-train],  
pred,  
dnn=c( "Observed Class","Predicted Class" ))
```

		Predicted Class			
Observed Class	bus	opel	saab	van	
bus	88	1	0	5	
opel	1	37	43	4	
saab	1	33	53	1	
van	1	1	2	75	

The error / misclassification rate can be calculated.

```
> error_rate = (1-sum(pred== Vehicle$Class[-train]))/  
346  
> round(error_rate,3)  
[1] 0.269
```

Overall, the model achieves an error rate of 26.9% on the test sample.

Technique 22

Crammer - Singer Multi-Class SVM

A Crammer - Singer SVM for multicategory response classification can be estimated using the package `kernlab` with the `ksvm` function:

```
ksvm(y ~ ., data, kernel, type="spoc-svc", kpar, ...)
```

Key parameters include `kernel` - the kernel function, `type` set to "spoc-svc", `kpar` which contains parameter values, multicategory response variable `y` and the covariates `data`.

Step 3: Estimate & Evaluate Model

Step 1 and step 2 are outlined beginning on page 151.

We estimate the Crammer- Singer Multi-Class support vector machine using a radial basis kernel (`type = rbfdot`) and parameter `sigma` equal to 0.05. We also set the cross validation parameter `cross = 10`.

```
fit<- ksvm(Class ~ ., data = Vehicle[train,] ,  
type="spoc-svc" ,  
kernel=rbfdot ,  
kpar=list(sigma=0.05) ,  
cross=10)
```

The `print` function provides details of the estimated model:

```
> print(fit)  
Support Vector Machine object of class "ksvm"
```

```
SV type: spoc-svc (classification)
parameter : cost C = 1

Gaussian Radial Basis kernel function.
Hyperparameter : sigma = 0.05

Number of Support Vectors : 340

Objective Function Value : 0
Training error : 0.152
Cross validation error : 0.244
```

The function provides details of the type of support vector machine (spoc-svc), model cost parameter and kernel parameter (`sigma =0.05`), number of support vectors (340), training error of 15.2% and cross validation error of 24.4%. Both types of error can be individually accessed as follows:

```
> fit@error
[1] 0.152

> fit@cross
[1] 0.244
```

We need to tune the model. First we set up the ranges for the `cost` and `sigma` parameters.

```
> cost <- 2^(2:8)
> sigma<-seq(0.1,0.5,by=0.1)
> n_cost<-length(cost)
> n_sigma<-length(sigma)
```

The total number of models to be estimated is stored in `runs`.

```
> runs<-n_sigma*n_cost
> count.cost <- 0
> count.sigma<-0
```

The results in terms of cross validation error, `cost` and `sigma` are stored in `results`.

```
> results<-1:(3*runs)
> dim(results) <- c(runs,3)
> colnames(results) <- c("cost","sigma","error")
```

The objects `i`, `j` and `count` are loop variables.

```
> i=1
```

```
> j=1  
> count=1
```

The main loop for tuning is as follows.

```
> for (val in cost) {  
  for (val in sigma) {  
    cat("iteration = ", count , " out of ", runs , "\n")  
  
    fit<- ksvm(Class ~ ., data = Vehicle[train,],  
    type="kbb-svc",  
    C=cost[j],  
    kernel=rbfdot,  
    kpar=list(sigma=sigma[i]),  
    cross=45)  
  
    results[count,1]=fit@param$C  
    results[count,2]=sigma[i]  
    results[count,3]= fit@cross  
  
    count.sigma = count.sigma+1  
    count=count+1  
  
    i=i+1  
  } # end sigma loop  
  
  i=1  
  j=j+1  
  
} # end cost loop
```

When you execute the above code, as it is running you should see output along the lines of:

```
iteration = 1 out of 35  
iteration = 2 out of 35  
iteration = 3 out of 35  
iteration = 4 out of 35  
iteration = 5 out of 35
```

We turn `results` into a data frame using the `as.data.frame` method.

```
> results<-as.data.frame(results)
```

Take a peek at the results and you should see something like this.

```
> results
  cost sigma      error
1     4   0.1 0.2299663
2     4   0.2 0.2378788
3     4   0.3 0.2582492
4     4   0.4 0.2730640
```

Now let's find the best cross validation performance and its associated row number. So the optimal cross validation error is 20.7% and located in row 11 of `results`.

```
> with(results, error[error == min(error)]) [1]
0.2075758

> which(results$error== min(results$error))
[1] 11
```

We save in `best_result` using a few lines of code.

```
> best_per_row<-which(results$error== min(results$error))

> fit_cost<-results[best_per_row,1]
> fit_sigma<-results[best_per_row,2]
> fit_xerror<-results[best_per_row,3]

> best_result<-cbind(fit_cost,fit_sigma,fit_xerror)
> colnames(best_result)<- c("cost","sigma","error")

> best_result
  cost sigma      error
[1,]    16   0.1 0.2075758
```

Figure 22.1 presents a three dimensional visualization of the tuning numbers. It was created using `scatterplot3d` as follows:

```
> scatterplot3d(results$cost,
  results$sigma,
  results$error,
  xlab="cost",
  ylab="sigma",
  zlab="Error")
```

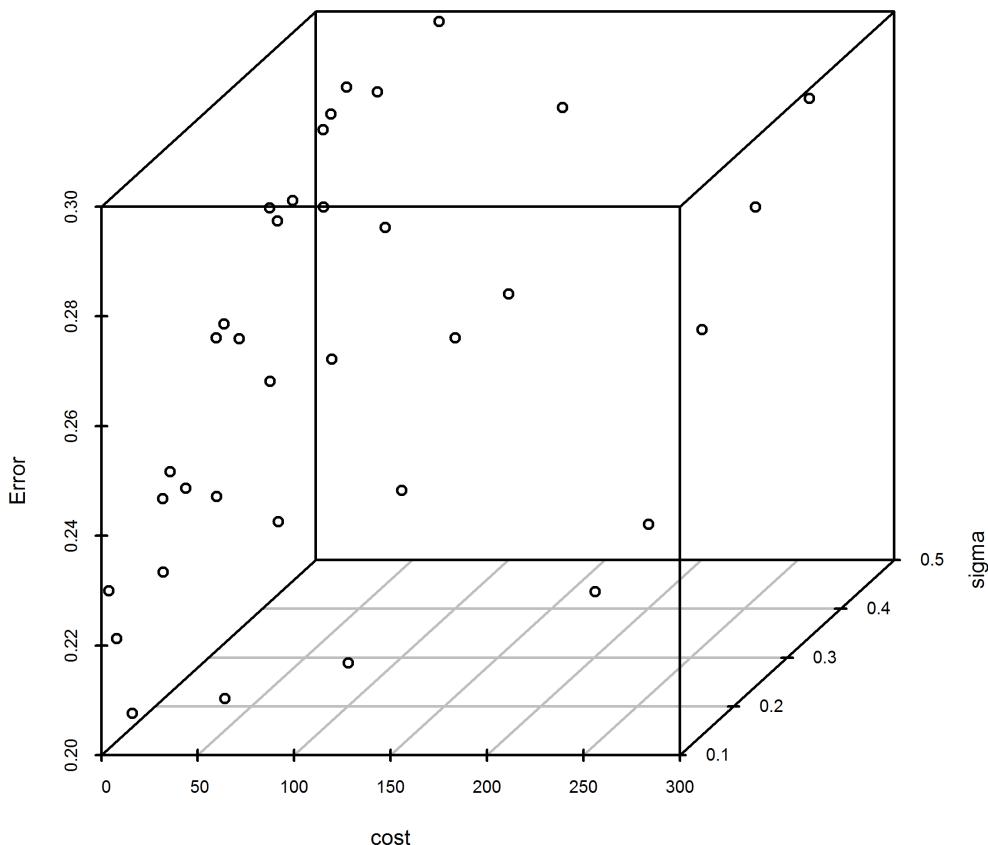


Figure 22.1: Crammer- Singer Multi-Class SVM tuning 3d scatterplot using Vehicle

After all that effort we may as well estimate the optimal model using the training data and show the cross validation error. It is around 25%.

```
> fit<- ksvm(Class ~ ., data = Vehicle[train,],
  type="spoc-svc",
  cost=fit_cost,
  kernel=rbfdot,
  kpar=list(sigma=fit_sigma),
  cross=45)

> fit@cross
```

```
[1] 0.2478114
```

Step 4: Make Predictions

The `predict` method with the test data and the fitted model fit are used as follows.

```
> pred <- predict(fit, Vehicle[-train,])
```

Classification results alongside the actual observed values can be obtained using the `table` method.

```
> table(Vehicle$Class[-train],  
pred,  
dnn=c( "Observed Class","Predicted Class" ))
```

		Predicted Class			
Observed Class	bus	opel	saab	van	
bus	88	0	0	6	
opel	6	43	30	6	
saab	3	29	52	4	
van	1	0	1	77	

The error / misclassification rate can then be calculated. Overall, the model achieves an error rate of 24.9% on the test sample.

```
> error_rate = (1-sum(pred== Vehicle$Class[-train]))/  
346)  
> round(error_rate,3)  
[1] 0.249
```

Support Vector Regression

Technique 23

SVM eps-Regression

A eps-regression can be estimated using the package `e1071` with the `svm` function:

```
svm(y ~., data, kernel, cost, type = "eps-regression",
     epsilon, gamma, cost, ...)
```

Key parameters include `kernel` - the kernel function the parameters `cost`, `gamma` and `epsilon`, `type` set to "`eps-regression`", continuous response variable `y` and the covariates `data`.

Step 1: Load Required Packages

We build the support vector machine eps-regression using the data frame `bodyfat` contained in the `TH.data` package. For additional details on this sample see page 62.

```
> library(e1071)
> data("bodyfat", package = "TH.data")
```

Step 2: Prepare Data & Tweak Parameters

We use 45 out of the 71 observations in the training sample with the remaining saved for testing. The variable `train` selects the random sample without replacement from the 71 observations.

```
> set.seed(465)
> train <- sample(1:71, 45, FALSE)
```

Step 3: Estimate & Evaluate Model

We begin by estimating the support vector machine using the default settings. This will use a radial basis function as the kernel with a cost parameter value set equal to 1.

```
> fit<- svm(DEXfat ~ ., data = bodyfat[train,],type= "eps-regression")
```

The **summary** function provides details of the estimated model:

```
> summary(fit)
```

Call:

```
svm(formula = DEXfat ~ ., data = bodyfat[train, ], type = "eps-regression")
```

Parameters:

```
SVM-Type: eps-regression  
SVM-Kernel: radial  
cost: 1  
gamma: 0.1111111  
epsilon: 0.1
```

Number of Support Vectors: 33

The function provides details of the type of support vector machine (eps-regression), cost parameter, kernel type and associated **gamma** and **epsilon** parameters, and the number of support vectors, in this case 33.

A nice feature of the **e1071** package is that contains **tune.svm**, a support vector machine tuning function. We use the method to identify the best model for **gamma** ranging between 0.25 and 4, and **cost** between 4 and 16 and **epsilon** between 0.01 to 0.09. The results are stored in **obj**.

```
> obj <- tune.svm(DEXfat ~ ., data = bodyfat[train, ],  
+ type="eps-regression",  
+ gamma = 2^(-2:2),  
+ cost = 2^(2:4),  
+ epsilon=seq(0.01,0.9,by=0.01))
```

Once again we can use the **summary** function to see the result.

```
> summary(obj)

Parameter tuning of svm:

- sampling method: 10-fold cross validation

- best parameters:
  gamma cost epsilon
  0.25     8      0.01

- best performance: 26.58699
```

The method automatically performs a 10-fold cross validation. We see that the best model has a `gamma` of 0.25, `cost` parameter equal to 8, `epsilon` of 0.01 with a 26.6% misclassification rate. These metrics can also be accessed calling `$best.performance` and `$best.parameters`.

```
> obj$best.performance
[1] 26.58699

> obj$best.parameters
  gamma cost epsilon
6  0.25     8      0.01
```

We use these optimum parameters to refit the model using leave one out cross validation as follows.

```
> bestEpi <- obj$best.parameters[[3]]
> bestGamma<-obj$best.parameters[[1]]
> bestCost<-obj$best.parameters[[2]]

> fit<- svm(DEXfat ~ ., data = bodyfat[train,],
  type="eps-regression",
  epsilon=bestEpi,
  gamma=bestGamma,
  cost=bestCost,
  cross=45)
```

The fitted model now has 44 support vectors.

```
> print(fit)

Call:
svm(formula = DEXfat ~ ., data = bodyfat[train, ],
  type = "eps-regression",
```

```
epsilon = bestEpi, gamma = bestGamma, cost =
bestCost, cross = 45)
```

Parameters:

```
SVM-Type: eps-regression
SVM-Kernel: radial
cost: 8
gamma: 0.25
epsilon: 0.01
```

Number of Support Vectors: 44

The summary method provides additional details. It also reports the mean square error for each cross validation (not shown here).

```
> summary(fit)
```

Call:

```
svm(formula = DEXfat ~ ., data = bodyfat[train, ],
type = "eps-regression",
epsilon = bestEpi, gamma = bestGamma, cost =
bestCost, cross = 45)
```

Parameters:

```
SVM-Type: eps-regression
SVM-Kernel: radial
cost: 8
gamma: 0.25
epsilon: 0.01
```

Number of Support Vectors: 44

45-fold cross-validation on training data:

```
Total Mean Squared Error: 28.0095
Squared Correlation Coefficient: 0.7877644
```

Step 4: Make Predictions

The `predict` method with the test data and the fitted model fit are used as follows.

```
> pred <- predict(fit, bodyfat[-train,])
```

A plot of the predicted and observed values, shown in Figure 23.1, is obtained using the `plot` function.

```
> plot(bodyfat$DEXfat[-train],  
pred, xlab="DEXfat",  
ylab="Predicted Values",  
main="Training Sample Model Fit")
```

We calculate the squared correlation coefficient using the `cor` function. It reports a value of 0.712.

```
round(cor(pred, bodyfat$DEXfat[-train])^2, 3)  
[1] 0.712
```

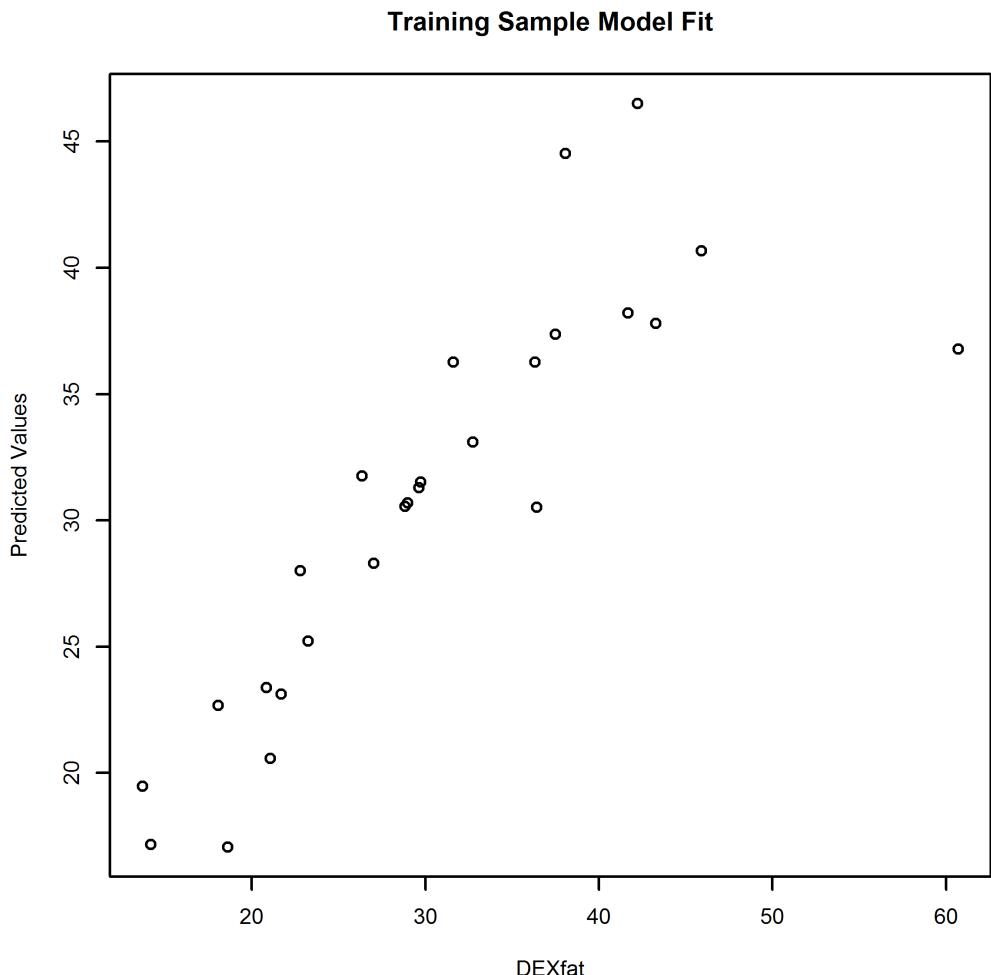


Figure 23.1: Predicted and observed values using SVM eps regression for bodyfat

Technique 24

SVM nu-Regression

A nu-regression can be estimated using the package `e1071` with the `svm` function:

```
svm(y ~ ., data, kernel, cost, type = "nu-regression", nu,  
     gamma, cost, ...)
```

Key parameters include `kernel` - the kernel function the parameters `cost`, `gamma` and `nu`, `type` set to "nu-regression", continuous response variable `y` and the covariates `data`.

Step 3: Estimate & Evaluate Model

Step 1 and 2 are outlined beginning on page 172.

We begin by estimating the support vector machine using the default settings. This will use a radial basis function as the kernel with a cost parameter value set equal to 1.

```
> fit<- svm(DEXfat ~ ., data = bodyfat[train,], type =  
  "nu-regression")
```

The `summary` function provides details of the estimated model:

```
> summary(fit)
```

Call:

```
svm(formula = DEXfat ~ ., data = bodyfat[train, ],  
  type = "nu-regression")
```

```
Parameters:  
  SVM-Type: nu-regression  
  SVM-Kernel: radial  
    cost: 1  
   gamma: 0.1111111  
     nu: 0.5
```

Number of Support Vectors: 33

The function provides details of the type of support vector machine (nu-regression) , cost parameter, kernel type and associated **gamma** and **nu** parameters, and the number of support vectors, in this case 33 which happens to be the same as estimated using the SVM eps-regression (see page 172). We use the **tune.svm** method to identify the best model for **gamma** ranging between 0.25 and 4, and **cost** between 4 and 16 and **nu** between 0.01 to 0.09. The results are stored in **obj**.

```
> obj <- tune.svm(DEXfat ~ .,  
data = bodyfat[train,],  
type="nu-regression",  
gamma = 2^(-2:2),  
cost = 2^(2:4),  
nu=seq(0.1,0.9,by=0.1))
```

We use the **summary** function to see the result.

```
> summary(obj)
```

Parameter tuning of svm:

- sampling method: 10-fold cross validation
- best parameters:
gamma **cost** **nu**
0.25 8 0.9
- best performance: 26.42275

The method automatically performs a 10-fold cross validation. We see that the best model has a **gamma** of 0.25, **cost** parameter equal to 8, **nu** equal to 0.9 with a 26.4% misclassification rate. These metrics can also be accessed calling **\$best.performance** and **\$best.parameters**.

```
> obj$best.performance  
[1] 26.42275
```

```
> obj$best.parameters
  gamma  cost   nu
126  0.25     8 0.9
```

We use these optimum parameters to refit the model using leave one out cross validation as follows.

```
> bestNU <- obj$best.parameters[[3]]
> bestGamma<-obj$best.parameters[[1]]
> bestCost<-obj$best.parameters[[2]]

> fit<- svm(DEXfat ~ .,
  data = bodyfat[train,],
  type="nu-regression",
  nu=bestNU,
  gamma=bestGamma,
  cost=bestCost,
  cross=45)
```

The fitted model now has 45 support vectors.

```
> print(fit)

Call:
svm(formula = DEXfat ~ ., data = bodyfat[train, ],
  type = "nu-regression",
  nu = bestNU, gamma = bestGamma, cost = bestCost,
  cross = 45)
```

Parameters:

```
SVM-Type: nu-regression
SVM-Kernel: radial
cost: 8
gamma: 0.25
nu: 0.9
```

Number of Support Vectors: 45

The summary method provides additional details. It also reports the mean square error for each cross validation (not shown here).

```
> summary(fit)
```

Call:

```
svm(formula = DEXfat ~ ., data = bodyfat[train, ],  
    type = "nu-regression",  
    nu = bestNU, gamma = bestGamma, cost = bestCost,  
    cross = 45)
```

Parameters:

```
SVM-Type: nu-regression  
SVM-Kernel: radial  
cost: 8  
gamma: 0.25  
nu: 0.9
```

Number of Support Vectors: 45

45-fold cross-validation on training data:

```
Total Mean Squared Error: 27.98642  
Squared Correlation Coefficient: 0.7871421
```

Step 4: Make Predictions

The `predict` method with the test data and the fitted model fit are used as follows.

```
> pred <- predict(fit, bodyfat[-train,])
```

A plot of the predicted and observed values, shown in Figure 24.1, is obtained using the `plot` function.

```
> plot(bodyfat$DEXfat[-train],  
pred, xlab = "DEXfat",  
ylab = "Predicted Values",  
main = "Training Sample Model Fit")
```

We calculate the squared correlation coefficient using the `cor` function. It reports a value of 0.712.

```
round(cor(pred, bodyfat$DEXfat[-train])^2, 3)  
[1] 0.712
```

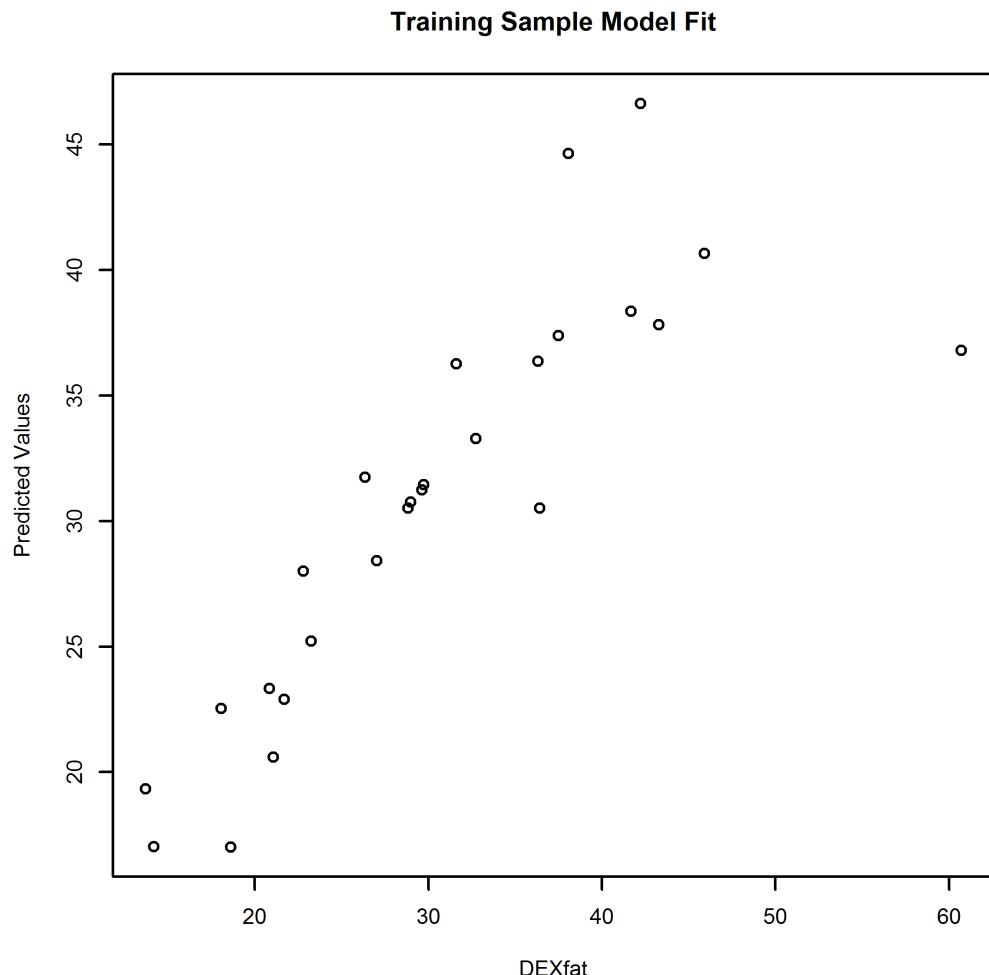


Figure 24.1: Predicted and observed values using SVM nu regression for bodyfat

Technique 25

Bound-constraint SVM eps-Regression

A bound-constraint SVM eps-regression can be estimated using the package `kernlab` with the `ksvm` function:

```
ksvm(y ~ ., data, kernel, type="eps-bsvr", kpar...)
```

Key parameters include `kernel` - the kernel function, `type` set to `type="eps-bsvr"`, `kpar` which contains parameter values, multicategory response variable `y` and the covariates `data`.

Step 1: Load Required Packages

We build the model using the data frame `bodyfat` contained in the `TH.data` package. For additional details on this sample see page 62.

```
library(kernlab)
data("bodyfat", package="TH.data")
```

Step 2: Prepare Data & Tweak Parameters

We use 45 out of the 71 observations in the training sample with the remaining saved for testing. The variable `train` selects the random sample without replacement from the 71 observations.

```
set.seed(465)
train <- sample(1:71, 45, FALSE)
```

Step 3: Estimate & Evaluate Model

We estimate the bound-constraint support vector machine using a radial basis kernel (`type = rbfdot`) and parameter `sigma` equal to 0.05. We also set the cross validation parameter `cross =10`.

```
fit<- ksvm(DEXfat ~ ., data = bodyfat[train,],  
type="eps-bsvr",  
kernel=rbfdot,  
kpar=list(sigma=0.05),cross=10)
```

The `print` function provides details of the estimated model:

```
> print(fit)  
Support Vector Machine object of class "ksvm"  
  
SV type: eps-bsvr  
Gaussian Radial Basis kernel function.  
Hyperparameter : sigma = 0.05  
  
Number of Support Vectors : 35  
  
Objective Function Value : -7.7105  
Training error : 0.096254  
Cross validation error : 18.77613
```

The function provides details of the type of support vector machine (eps-bsvr), kernel parameter (`sigma =0.05`). Notice the model estimates 35 support vectors, with a training error of 9.6% and cross validation error of 18.7%. Since the output of `ksvm` (in this case values are stored in `fit`) is an S4 object, both errors can be accessed directly using “@” .

```
> fit@param$C  
[1] 1  
  
> fit@param$epsilon  
[1] 0.1  
  
> fit@error  
[1] 0.09625418  
  
> fit@cross  
[1] 18.77613
```

We need to tune the model to obtain the optimum parameters. Let's create a small bit of code to do this for us. First we set up the ranges for the `cost` and `sigma` and `epsilon` parameters.

```
> epsilon <- seq(0.01,0.1,by=0.01)
> sigma<-seq(0.1,1,by=0.01)

> n_epsilon<-length(epsilon)
> n_sigma<-length(sigma)
```

The total number of models to be estimated is stored in `runs`. Since it contains 910 models it may take a little while to run.

```
> runs<-n_sigma*n_epsilon
> runs
[1] 910
```

The results in terms of cross validation error, `cost`,`epsilon` and `sigma` are stored in `results`.

```
> results<-1:(4*runs)
> dim(results) <- c(runs,4)
> colnames(results) <- c("cost","epsilon","sigma",
  "error")
```

The loop variables are as follows.

```
> count.epsilon<-0
> count.sigma<-0

> i=1
> j=1
> k=1
> count=1
```

The main loop for tuning is as follows.

```
> for (val in epsilon) {

  for (val in sigma) {
fit<- ksvm(DEXfat ~ ., data = bodyfat[train,],
type="eps-bsvr",
epsilon=epsilon[k],
kernel=rbfdot,
kpar=list(sigma=sigma[i]),
cross=45)
```

```
results[count,1]=fit@param$C
results[count,2]=sigma[i]
results[count,3]= fit@param$epsilon
results[count,4]= fit@cross
count.sigma = count.sigma+1
count=count+1
i=i+1

} # end sigma loop
i=1
k=k+1
cat("iteration (%) = ", round((count/runs)*100,0)
, "\n")

} # end epsilon loop
i=1
k=1
j=j+1
```

Notice we set `cross = 45` to perform leave one out validation.

When you execute the above code, as it is running, you should see output along the lines of:

```
iteration (%) = 10
iteration (%) = 20
iteration (%) = 30
```

We turn `results` into a data frame using the `as.data.frame` method.

```
> results<-as.data.frame(results)
```

Take a peek at the results and you should see something like this.

```
> results
   cost epsilon sigma      error
1      1    0.10  0.01 22.15108
2      1    0.11  0.01 22.76689
3      1    0.12  0.01 23.24192
4      1    0.13  0.01 23.78274
5      1    0.14  0.01 24.35676
```

Now let's find the best cross validation performance (21.8%) and its associated row number (274).

```
> with(results, error[error == min(error)])
```

```
[1] 21.86181
```

```
> which(results$error == min(results$error))
[1] 274
```

The optimal values may need to be stored for later use. We save them in `best_result` using a few lines of code.

```
> best_per_row<-which(results$error == min(results$error))

> fit_cost<-results[best_per_row,1]
> fit_sigma<-results[best_per_row,2]
> fit_epsilon<-results[best_per_row,3]
> fit_xerror<-results[best_per_row,4]

> best_result<-cbind(fit_cost,fit_epsilon,fit_sigma,
  fit_xerror)
> colnames(best_result)<- c("cost","epsilon","sigma",
  "error")

> best_result
      cost   epsilon   sigma     error
[1,]    1     0.04    0.1 21.86181
```

After all that effort we may as well estimate the optimal model using the training data and show the cross validation error. It is around 21.8%.

```
> fit<- ksvm(DEXfat ~ ., data = bodyfat[train,],
  type="eps-bsvr",
  cost=fit_cost,
  epsilon=fit_epsilon,
  kernel=rbfdot,
  kpar=list(sigma=fit_sigma),
  cross=45)

> fit@cross
[1] 21.86834
```

Step 4: Make Predictions

The `predict` method with the test data and the fitted model `fit` are used as follows.

```
pred <- predict(fit2, bodyfat[-train,])
```

We fit a linear regression using `pred` as the response variable and the observed values as the covariate. The regression line alongside the predicted and observed values, shown in Figure 25.1, are visualized using the `plot` method combined with the `abline` method to show the linear regression line.

```
> linReg<-lm(pred ~ bodyfat$DEXfat [-train])
```

```
> plot(bodyfat$DEXfat [-train],
pred,
xlab="DEXfat",
ylab="Predicted Values",
main="Training Sample Model Fit")
```

```
> abline(linReg,col="darkred")
```

The correlation between the test sample predicted and observed values is 0.83.

```
> round(cor(pred,bodyfat$DEXfat [-train])^2,3)
[1,] 0.834
```

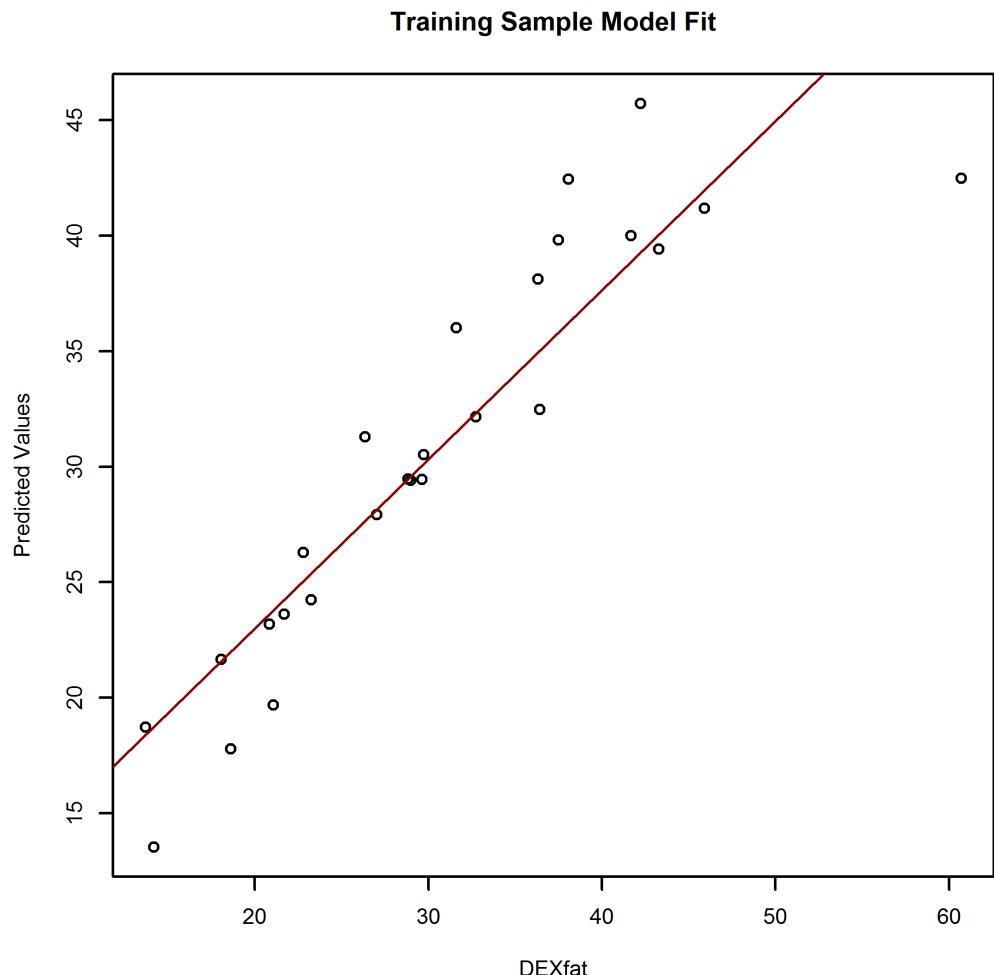


Figure 25.1: Bound-constraint SVM eps-Regression observed and fitted values using `bodyfat`

Support Vector Novelty Detection

Technique 26

One-Classification SVM

A One-Classification SVM can be estimated using the package `e1071` with the `svm` function:

```
svm(x, kernel, type = "one-classification", ...)
```

Key parameters `x` - the attributes you use to train the model, `type = "one-classification"` and `kernel` - the kernel function.

Step 1: Load Required Packages

We build the one-classification SVM using the data frame `Vehicle` contained in the `mlbench` package. For additional details on this sample see page 23.

```
> library(e1071)
> library(caret)
> library(mlbench)
> data(Vehicle)
> library(scatterplot3d)
```

Step 2: Prepare Data & Tweak Parameters

To begin, we choose the bus category as our one classification class and create the variable `isbus` to store TRUE/FALSE values of vehicle type.

```
> set.seed(103)
> Vehicle$isbus[Vehicle$Class == "bus"] <- TRUE
> Vehicle$isbus[Vehicle$Class != "bus"] <- FALSE
```

As a reminder of the distribution of `Vehicle` we use the `table` method.

```
> table(Vehicle$Class)

bus  opel  saab   van
218   212   217   199
```

As a check note that you should observed 218 observations corresponding to bus in `isbus= TRUE`.

```
> table(Vehicle$isbus)

FALSE   TRUE
628     218
```

Our next step is to create variables `isbusTrue` and `isbusFalse` to hold positive (bus) and negative (non-bus) observations respectively.

```
> isbusTrue<-subset(Vehicle,Vehicle$isbus=="TRUE")
> isbusFalse<-subset(Vehicle,Vehicle$isbus=="FALSE")
```

Next, create a subset of 218 positive values and sample 200 of these at random. The remaining 18 observations will form the test sample.

```
> train <- sample(1:218, 100, FALSE)

> trainAttributes<-isbusTrue[train,1:18]
> trainLabels<-isbusTrue[train,19]
```

Now, as a quick check type `trainLabels` and check that they are all "bus".

Step 3: Estimate & Evaluate Model

We need to tune the model. We begin by setting up the tuning parameters. The variable `runs` contains the total number of models we will estimate during the tuning process.

```
gamma <- seq(0.01,0.05,by=0.01)
nu <- seq(0.01,0.05,by=0.01)

n_gam<-length(gamma)
n_nu<-length(nu)

runs<-n_gam*n_nu

count.gamma <- 0
count.nu<-0
```

Next, set up the variable to store the results and the loop count variables - i, j and count.

```
> results<-1:(3*runs)
>   dim(results) <- c(runs ,3)
>   colnames(results) <- c("gamma","nu","Performance")
""

> i=1
> k=1
> count=1
```

The main tuning loop is created as follows (notice we estimate the model using a 10-fold cross validation (`cross = 10`) and store the results in `fit`).

```
> for (val in nu) {

  for (val in gamma) {
    print(gamma[i])
    print(nu[k])

    fit<-svm(trainAttributes ,
y=NULL ,
type='one-classification' ,
nu=nu[k] ,
gamma=gamma[i] ,
cross=10)

    results[count ,2]=fit$gamma
    results[count ,1]=fit$nu
    results[count ,3]= fit$tot.accuracy
    count.gamma = count.gamma+1
    count=count+1
    i=i+1

  } # end gamma loop
  i=1
  k=k+1
} # end nu loop
```

We turn `results` into a data frame using the `as.data.frame` method.

```
> results<-as.data.frame(results)
```

Take a peek and you should see something like this.

```
> results
```

	gamma	nu	Performance
1	0.01	0.01	94
2	0.01	0.02	88
3	0.01	0.03	87
4	0.01	0.04	81

Now let's find the best cross validation performance.

```
> with(results, Performance[Performance == max(Performance)])  
[1] 94 94
```

Notice the best performance contains two values both at 94%. Since this is a very high value it is possibly the result of over fitting - a consequence of over tuning a model! The optimal values may need to be stored for later use. We save them in `best_result` using a few lines of code.

```
> best_per_row<-which(results$Performance == max(results$Performance))  
  
> fit_gamma<-results[best_per_row,1]  
> fit_nu<-results[best_per_row,2]  
> fit_per<-results[best_per_row,3]  
  
> best_result<-cbind(fit_gamma,fit_nu,fit_per)  
> colnames(best_result)<- c("gamma","nu","Performance")  
  
> best_result  
      gamma    nu Performance  
[1,] 0.01 0.01        94  
[2,] 0.02 0.01        94
```

The relationship between the parameters and performance is visualized using the `scatterplot3d` function and shown in Figure 26.1.

```
> scatterplot3d(results$nu,results$gamma,results$  
  Performance,xlab="nu",ylab="gamma",zlab="Performance")
```

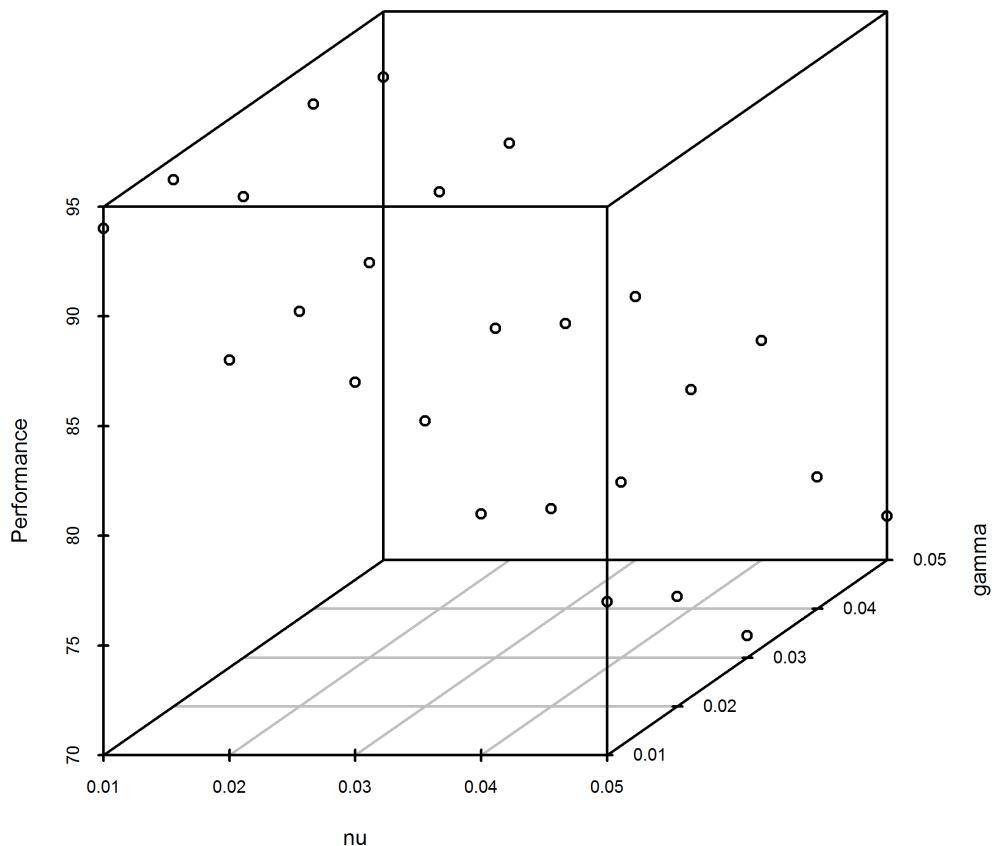


Figure 26.1: Relationship between tuning parameters

Let's fit the optimum model using a leave one out cross validation.

```
> fit<-svm(trainAttributes ,y=NULL ,  
type="one-classification" ,  
nu=fit_nu ,  
gamma=fit_gamma  
,cross=100)
```

Step 4: Make Predictions

To illustrate the performance of the model on the test sample we will use the `confusionMatrix` method from the `caret` package. The first step is to gather

together the required information for the training and test sample using the predict method to make the forecasts.

```
> trainpredictors<-isbusTrue[train,1:18]
> trainLabels<-isbusTrue[train,20]

> testPositive<-isbusTrue[-train,]
> testPosNeg<-rbind(testPositive,isbusFalse)

> testpredictors<-testPosNeg[,1:18]
> testLabels<-testPosNeg[,20]

> svm.predtrain<-predict(fit,trainpredictors)
> svm.predtest<-predict(fit,testpredictors)
```

Next we create two tables, confTrain containing the training predictions and confTest for the test sample predictions.

```
> confTrain<-table(Predicted=svm.predtrain,Reference
+ =trainLabels)
> confTest<-table(Predicted=svm.predtest,Reference=
+ testLabels)
```

Now we call the confusionMatrix method for the test sample.

```
confusionMatrix(confTest,positive="TRUE")
Confusion Matrix and Statistics
```

Predicted	Reference	
FALSE	TRUE	
FALSE	175	5
TRUE	453	113

Accuracy : 0.3861
95% CI : (0.351, 0.4221)

No Information Rate : 0.8418

P-Value [Acc > NIR] : 1

Kappa : 0.093
McNemar's Test P-Value : <2e-16

Sensitivity : 0.9576
Specificity : 0.2787
Pos Pred Value : 0.1996

```
Neg Pred Value : 0.9722
    Prevalence : 0.1582
    Detection Rate : 0.1515
Detection Prevalence : 0.7587
Balanced Accuracy : 0.6181

'Positive' Class : TRUE
```

The method produces a range of test statistics, including an accuracy rate of only 38% and kappa of 0.093. Maybe we over-tuned the model a little! In this example we can see clearly the consequence of over fitting in training is poor generalization in the test sample.

For illustrative purposes we re-estimate the model this time with nu = 0.05 and gamma 0.01.

```
> fit<-svm(trainAttributes ,
y=NULL ,
type="one-classification",
nu=0.05 ,
gamma=0.05 ,
cross=100)
```

A little bit of house keeping as discussed previously.

```
> trainpredictors<-isbusTrue[train,1:18]
> trainLabels<-isbusTrue[train,20]

> testPositive<-isbusTrue[-train,]
> testPosNeg<-rbind(testPositive,isbusFalse)

> testpredictors<-testPosNeg[,1:18]
> testLabels<-testPosNeg[,20]

> svm.predtrain<-predict(fit,trainpredictors)
> svm.predtest<-predict(fit,testpredictors)

> confTrain<-table(Predicted=svm.predtrain,Reference
= trainLabels)
> confTest<-table(Predicted=svm.predtest,Reference=
 testLabels)
```

Now we are ready to pass the necessary information to the confusionMatrix method.

```
> confusionMatrix(confTest,positive="TRUE")
```

Confusion Matrix and Statistics

```
Reference
Predicted FALSE TRUE
  FALSE    568    29
  TRUE      60    89

Accuracy : 0.8807
95% CI  : (0.8552, 0.9031)
No Information Rate : 0.8418
P-Value [Acc > NIR] : 0.001575

Kappa : 0.5952
McNemar's Test P-Value : 0.001473

Sensitivity : 0.7542
Specificity  : 0.9045
Pos Pred Value : 0.5973
Neg Pred Value : 0.9514
Prevalence   : 0.1582
Detection Rate : 0.1193
Detection Prevalence : 0.1997
Balanced Accuracy : 0.8293

'Positive' Class : TRUE
```

Now the accuracy rate is 0.88 with a kappa of around 0.6. An important take away is that support vector machines, as with many predictive analytic methods, can be very sensitive to over fitting!

Notes

³⁵See the paper by Cortes C, Vapnik V (1995) Support-Vector Networks. *Machine Learning* 20: 273–297.

³⁶Note that classification tasks based on drawing separating lines to distinguish between objects of different class memberships are known as hyperplane classifiers. The SVM is well suited for such tasks.

³⁷The nu parameter in nu-SVM.

³⁸For further details see Hoehler, F.K., 2000. Bias and prevalence effects on kappa viewed in terms of sensitivity and specificity. *J. Clin. Epidemiol.* 53, 499–503.

³⁹Gomes, A. L., et al. "Classification of dengue fever patients based on gene expression data using support vector machines." *PLoS one* 5.6 (2010): e11267.

⁴⁰Huang, Wei, Yoshiteru Nakamori, and Shou-Yang Wang. "Forecasting stock market movement direction with support vector machine." *Computers & Operations Research* 32.10 (2005): 2513-2522.

⁴¹Min, Jae H., and Young-Chan Lee. "Bankruptcy prediction using support vector machine with optimal choice of kernel function parameters." *Expert systems with applications* 28.4 (2005): 603-614.

⁴²Upstill-Goddard, Rosanna, et al. "Support vector machine classifier for estrogen receptor positive and negative early-onset breast cancer." (2013): e68606.

⁴³Tehrany, Mahyat Shafapour, et al. "Flood susceptibility assessment using GIS-based support vector machine model with different kernel types." *Catena* 125 (2015): 91-101.

⁴⁴Calculated by the author using the average of all four support vector machines.

⁴⁵Radhika, K. R., M. K. Venkatesha, and G. N. Sekhar. "Off-line signature authentication based on moment invariants using support vector machine." *Journal of Computer Science* 6.3 (2010): 305.

⁴⁶Guo, Shuyu, Robyn M. Lucas, and Anne-Louise Ponsonby. "A novel approach for prediction of vitamin d status using support vector regression." *PLoS one* 8.11 (2013).

⁴⁷Latitude, ambient ultraviolet radiation levels, ambient temperature, hours in the sun 6 weeks before the blood draw (log transformed to improve the linear fit), frequency of wearing shorts in the last summer, physical activity (three levels: mild, moderate, vigorous), sex, hip circumference, height, left back shoulder melanin density, buttock melanin density and inner upper arm melanin density.

⁴⁸For further details see either of the following:

1. Cawley GC. Leave-one-out cross-validation based model selection criteria for weighted LS-SVMs. In: International Joint Conference on Neural Networks. IEEE; 2006. p. 1661–1668.
2. Vapnik V, Chapelle O. Bounds on error expectation for support vector machines. *Neural computation*. 2000;12(9):2013–2036.
3. Muller KR, Mika S, Ratsch G, Tsuda K, Scholkopf B. An introduction to kernel-based learning algorithms. *IEEE Transactions on Neural Networks*.

⁴⁹See:

1. Bach FR, Lanckriet GR, Jordan MI. Multiple kernel learning, conic duality, and the SMO algorithm. In: Proceedings of the twenty-first international conference on Machine learning. ACM; 2004. p. 6–13.
2. Zien A, Ong CS. Multiclass multiple kernel learning. In: Proceedings of the 24th international conference on Machine learning. ACM; 2007. p. 1191–1198.

⁵⁰Thadani K, Jayaraman V, Sundararajan V. Evolutionary selection of kernels in support vector machines. In: International Conference on Advanced Computing and Communications. IEEE; 2006. p. 19–24.

⁵¹See for example:

1. Lin, Shih-Wei, et al. "Particle swarm optimization for parameter determination and feature selection of support vector machines." *Expert systems with applications* 35.4 (2008): 1817-1824.
2. Melgani, Farid, and Yakoub Bazi. "Classification of electrocardiogram signals with support vector machines and particle swarm optimization." *Information Technology in Biomedicine, IEEE Transactions on* 12.5 (2008): 667-677.

⁵²For further discussion on the issues surrounding over fitting see G. C. Cawley and N. L. C. Talbot, Preventing over-fitting in model selection via Bayesian regularization of the hyper-parameters, *Journal of Machine Learning Research*, volume 8, pages 841-861, April 2007.

Part III

Relevance Vector Machine

The Basic Idea

The relevant vector machine (RVM) shares its functional form with the support vector machine (SVM) discussed in Part II. RVMs exploit a probabilistic Bayesian learning framework.

We begin with a data set of N training pairs $\{\mathbf{x}_i, y_i\}$, where \mathbf{x}_i is the input feature vector and, y_i is the target output. RVM make predictions using:

$$\hat{y}_i = \mathbf{w}^T \mathbf{K} + \varepsilon, \quad (26.1)$$

where $\mathbf{w} = [w_1, \dots, w_N]$, is the vector of weights, $\mathbf{K} = [k(x_i, x_1), \dots, k(x_i, x_N)]^T$ is the vector of kernel functions, and ε is the error which for algorithmic simplicity is assumed to be zero-mean independently identically distributed Gaussian with variance σ^2 . Therefore the prediction \hat{y}_i consists of the target output polluted by Gaussian noise.

The Gaussian likelihood of the data is given by:

$$p(\mathbf{y}|\tilde{\mathbf{w}}, \sigma^2) = (2\pi)^{-N/2} \sigma^{-N} \exp \left\{ -\frac{1}{2\sigma^2} \|\mathbf{y} - \Phi \tilde{\mathbf{w}}\|^2 \right\}, \quad (26.2)$$

where $\mathbf{y} = [y_1, \dots, y_N]$, $\tilde{\mathbf{w}} = [\varepsilon, w_1, \dots, w_N]$, and Φ is an $N \times (N+1)$ matrix with $\Phi_{ij} = k(\mathbf{x}_i, \mathbf{x}_{j-1})$ and $\Phi_{i1} = 1$.

A standard approach to estimate the parameters in equation 26.2 is to use a zero-mean Gaussian to introduce an individual hyperparameter α_i on each of the weights w_i so that:

$$\tilde{\mathbf{w}} \sim N \left(0, \frac{1}{\alpha} \right) \quad (26.3)$$

where $\alpha = [\alpha_1, \dots, \alpha_N]$. The prior and posterior probability distributions are then easily derived⁵³.

RVM uses much fewer kernel functions than the SVM. It also yields probabilistic predictions, automatic estimation of parameters and the ability to use arbitrary kernel functions. The majority of parameters are automatically set to zero during the learning process, giving a procedure that is extremely

effective at discerning basis functions which are relevant for making good predictions.

NOTE... ↗

RVM requires the inversion of $N \times N$ matrices. Since it takes $O(N^3)$ operations to invert an $N \times N$ matrix it can quickly become computationally expensive and therefore slow as the sample size increases.

Practical Applications

☛ PRACTITIONER TIP ☛

The relevance vector machine is often assessed using the root mean squared error (RMSE) or the Nash-Sutcliffe efficiency (NS). The larger the value of NS the smaller the value of RMSE.

$$RMSE = \sqrt{\frac{\sum_{i=1}^N (\hat{x}_i - x_i)^2}{N}} \quad (26.4)$$

$$NS = 1 - \frac{\sum_{i=1}^N (x_i - \hat{x}_i)^2}{\sum_{i=1}^N (x_i - \bar{x})^2} \quad (26.5)$$

\hat{x}_i is the predicted value, \bar{x} is the average of the sample and N is the number of observations.

Oil Sand Pump Prognostics

In wet mineral processing operations slurry pumps deliver a mixture of bitumen, sand, and small pieces of rock from one site to another. Due to the harsh environment in which they operate these pumps can fail suddenly. The consequent downtime can lead to a large economic cost for the mining company due to the interruption of the mineral processing operations. Hu and Tse⁵⁴ combine a RVM with an exponential model to predict the remaining useful life (RUL) of slurry pumps.

Data were collected from the inlet and outlet of slurry pumps operating in an oil sand mine using four accelerometers placed at key locations on the pump. In total, the pump was subjected to 904 measurement hours.

Two data-sets were collected from different positions (Site 1 and Site 2) in the same pump and an alternative model of the pump degradation was de-

veloped using the sum of two exponential functions. The overall performance results are shown in Table 13.

Site 1	Site 1	Site 2	Site 2
RVM	Exponential Model	RVM	Exponential Model
70.51%	25.31%	28.85%	7.05%

Table 13: Hu and Tse’s weighted average accuracy of prediction

Precision Agriculture

Precision agriculture involves using aircraft or spacecraft to gather high-resolution information on crops to better manage the growing and harvesting cycle. Chlorophyll concentration, measured in mass per unit leaf area ($\mu\text{g cm}^{-2}$), is an important biophysical measurement retrievable from air or space reflectance data. Elarab et al⁵⁵ use a RVM to estimate spatially distributed chlorophyll concentrations.

Data was gathered from three aircraft flights during the growing season (early growth, mid growth and early flowering). The data set consisted of the dependent variable (Chlorophyll concentration) and 8 attribute or explanatory variables. All attributes were used to train and test the model.

The researchers used six different kernels (Gauss, Laplace, spline, Cauchy, thin plate spline (tps), and bubble) alongside a 5-fold cross validation. A variety of kernel widths ranging from 10^{-5} to 10^5 were also used. The root mean square error and Nash-Sutcliffe efficiency were used to assess model fit. The best fitting trained model used a tps kernel and had a root mean square error of $5.31 \mu\text{g cm}^{-2}$ and Nash-Sutcliffe efficiency of 0.76.

The test data consisted of a sample gathered from an unseen (by the RVM) flight. The model using this data had a root mean square error of $8.52 \mu\text{g cm}^{-2}$ and Nash-Sutcliffe efficiency of 0.71. The researchers conclude: “*This result showed that the [RVM] model successfully performed when given unseen data.*”

Deception Detecting in Speech

Zhou⁵⁶ develop a technique to detect deception in speech by extracting speech dynamics (prosodic and non-linear features) and applying a RVM.

The data set consisted of recorded interviews of 16 male and 16 female participants. A total of 640 deceptive samples and 640 non-deceptive samples were used in the analysis.

Speech dynamics such as pitch frequency, short-term vocal energy and mei-frequency cepstrum coefficients⁵⁷ were used as input/ attribute features.

Classification accuracy of the RVM was assessed relative to a support vector machine and a neural network for various training sample sizes. For example, with a training sample of 400, the RVM correctly classifies 70.37% of male voices whilst the support vector machine and neural network correctly classify 68.14% and 42.13% respectively.

The researchers observe that a combination of prosodic and non-linear features modeled using a RVM is effective for detecting deceptive speech.

Diesel Engine Performance

The mathematical form of diesel engines is highly nonlinear. Because of this they are often modeled using an artificial neural network (ANN). Wong et al⁵⁸ perform an experiment to assess the ability of an ANN and a RVM to predict diesel engine performance.

Three inputs are used in the models (engine speed, load and cooling water temperature), and the output variables are brake specific fuel consumption and exhaust emissions such as nitrogen oxide and carbon dioxide.

A water-cooled, 4-cylinder, direct-injection diesel engine was used to generate data for the experiment. Data was recorded at five engine speeds (1200, 1400, 1600, 1800, and 2000 rpm) with engine torques (28, 70, 140, 210, and 252 Nm). Each test was carried out three times and the average values were used in the analysis. In all 22 sets of data were collected with 18 used as the training data and 4 to assess model performance.

The ANN had a single layer with twenty hidden neurons and a Hyperbolic tangent sigmoid transfer function.

For the training data the researchers report an average root mean square error (RMSE) of 3.27 for the RVM and 41.59 for the ANN. The average RMSE for the test set was 17.73 and 38.56 for the RVM and ANN respectively. The researchers also note that the average R^2 for the test set for the RVM was 0.929 and 0.707 for the ANN.

Gas Layer Classification

Zhao et al⁵⁹ consider the issue of optimal parameter selection for a RVM and identification of gas at various drill depths. To optimize the parameters of a

RVM they use the particle swarm algorithm⁶⁰.

The sample consist of 201 wells, drilled at various depths with 63 gas producing wells and 138 non-gas producing wells. A total of 12 logging attributes are used as features into the RVM model. A prediction accuracy of 93.53% is obtained during training using all the attributes; the prediction accuracy was somewhat lower at 91.75% for the test set.

Credit Risk Assessment

Tong and Li⁶¹ assess the use of RVM and a support vector machine (SVM) in the assessment of company credit risk. The data consist of financial characteristics of 464 Chinese firms listed on China's securities markets. A total of 116 of the 464 firms had experienced a serious credit event and were coded as “0” by the researchers. The remaining firms were coded “1”.

The attribute vector consisted of 25 financial ratios drawn from seven basic categories (cash flow, return on equity, earning capacity, operating capacity, growth capacity, short term solvency and long term solvency). Since many of these financial ratios have a high correlation with each other, the researchers used principal components (PCA) and isomaps to reduce the dimensionality. The models they then compared were a PCA-RVM, Isomap-SVM, and Isomap-RVM. A summary of the results is reported in Table 14.

Model	Accuracy
Isomap-RVM	90.28%
Isomap-SVM	86.11%
PCA-RVM	89.59%

Table 14: Summary of Tong and Li's overall prediction accuracy

◆ PRACTITIONER TIP ◆

You may have noticed that several researchers compared their RVM to a support vector machine or other model. A data scientist I worked with faced an issue where one model (logistic regression) performed marginally better than an alternative model (decision tree). However, the decision was made to go with the decision tree, because, due to a quirk in the software, it was better labeled.

Zhou et al demonstrated the superiority of a RVM over a support vector machine for their data-set. The evidence was less compelling in the case of Tong and Li where Table 14 appears to indicate a fairly close race. In the case where alternative models perform in a similar range, the rationale for which to choose may be based on considerations not related to which model performed best in absolute terms on the test set. This was certainly the case in the choice between logistic regression and decision trees faced by my data scientist co-worker!

Technique 27

RVM Regression

A RVM regression can be estimated using the package `kernlab` with the `rvm` function:

```
rvm(y ~ ., data, kernel, kpar...)
```

Key parameters include `kernel` - the kernel function, `kpar` which contains parameter values, multicategory response variable `y` and the covariates `data`.

Step 1: Load Required Packages

We build the RVM regression using the data frame `bodyfat` contained in the `TH.data` package. For additional details on this sample see page 62.

```
library(kernlab)
data("bodyfat", package = "TH.data")
```

Step 2: Prepare Data & Tweak Parameters

We use 45 out of the 71 observations in the training sample with the remaining saved for testing. The variable `train` selects the random sample without replacement from the 71 observations.

```
set.seed(465)
train <- sample(1:71, 45, FALSE)
```

Step 3: Estimate & Evaluate Model

We estimate the RVM using a radial basis kernel (`type = rbfdot`) and parameter `sigma` equal to 0.05. We also set the cross validation parameter `cross =10`.

```
> fit<- rvm(DEXfat ~ .,
  data = bodyfat[train,],
  kernel=rbfdot,
  kpar=list(sigma=0.5),
  cross=10)
```

The `print` function provides details of the estimated model. The function provides details including the kernel type (radial basis) kernel parameter (`sigma =0.5`), number of relevance vectors (43) and the cross validation error.

```
> print(fit)
Relevance Vector Machine object of class "rvm"
Problem type: regression

Gaussian Radial Basis kernel function.
Hyperparameter : sigma = 0.5

Number of Relevance Vectors : 43
Variance : 121.1944
Training error : 24.802080247
Cross validation error : 1079.221
```

Since the output of `rvm` is an S4 object it can be accessed directly using “@”.

```
> fit@error
[1] 24.80208

> fit@cross
[1] 1079.221
```

OK, let's fit two other models and choose the one with the lowest cross validation error. The first model, `fit1`, is estimated using the default setting of `rvm`. The second model, `fit2`, uses a Laplacian kernel. Ten-fold cross validation is used for both models.

```
> fit1<- rvm(DEXfat ~ .,
  data = bodyfat[train,],
```

```
cross=10)

> fit2<- rvm(DEXfat ~ . ,
  data = bodyfat[train,] ,
  kernel=laplacedot ,
  kpar=list(sigma=0.001) ,
  cross=10)
```

Now we are ready to assess the fit of each of the three models - `fit`, `fit1` and `fit2`.

```
> fit@cross
[1] 1082.592

> fit1@cross
[1] 65.50046

> fit2@cross
[1] 21.9755
```

The model `fit2`, using a Laplacian kernel has by far the best overall cross validation error.

Step 4: Make Predictions

The `predict` method with the test data and the fitted model `fit2` is used as follows.

```
pred <- predict(fit2, bodyfat[-train,])
```

We fit a linear regression using the `pred` as the response variable and the observed values as the covariate. The regression line alongside the predicted and observed values, shown in Figure 27.1, are visualized using the `plot` method combined with `abline` method to show the linear regression line.

```
> linReg<-lm(pred ~ bodyfat$DEXfat [-train])

> plot(bodyfat$DEXfat [-train] ,
  pred ,
  xlab="DEXfat" ,
  ylab="Predicted Values" ,
  main="Training Sample Model Fit")

> abline(linReg , col="darkred")
```

The correlation between the test sample predicted and observed values is 0.813.

```
> round(cor(pred, bodyfat$DEXfat[-train])^2, 3)
[1,] 0.813
```

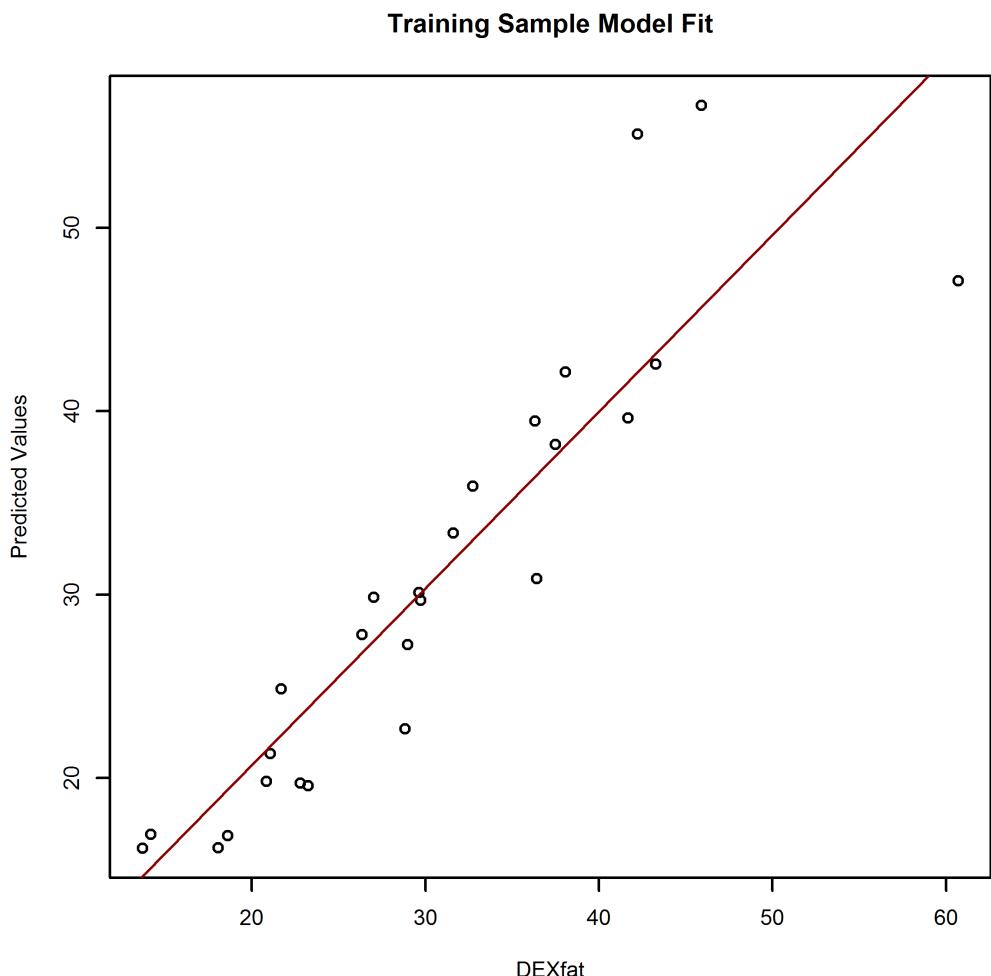


Figure 27.1: RVM regression of observed and fitted values using `bodyfat`

Notes

⁵³For further details see:

1. M. E. Tipping, "Bayesian inference: an introduction to principles and practice in machine learning," in Advanced Lectures on Machine Learning, O. Bousquet, U. von Luxburg, and G. Rätsch, Eds., pp. 41–62, Springer, Berlin, Germany, 2004.
2. M. E. Tipping, "SparseBays: An Efficient Matlab Implementation of the Sparse Bayesian Modelling Algorithm (Version 2.0)," March, 2009, <http://www.relevancevector.com>.

⁵⁴Hu, Jinfei, and Peter W. Tse. "A relevance vector machine-based approach with application to oil sand pump prognostics." Sensors 13.9 (2013): 12663-12686.

⁵⁵Elarab, Manal, et al. "Estimating chlorophyll with thermal and broadband multispectral high resolution imagery from an unmanned aerial system using relevance vector machines for precision agriculture." International Journal of Applied Earth Observation and Geoinformation (2015).

⁵⁶Zhou, Yan, et al. "Deception detecting from speech signal using relevance vector machine and non-linear dynamics features." Neurocomputing 151 (2015): 1042-1052.

⁵⁷Mei-frequency cepstrum is a representation of the short-term power spectrum of a sound commonly used as features in speech recognition systems. See for example, Logan, Beth. "Mel Frequency Cepstral Coefficients for Music Modeling." ISMIR. 2000; and Hasan, Md Rashidul, Mustafa Jamil, and Md Golam Rabbani Md Saifur Rahman. "Speaker identification using Mel frequency cepstral coefficients." variations 1 (2004): 4.

⁵⁸Wong, Ka In, Pak Kin Wong, and Chun Shun Cheung. "Modelling and prediction of diesel engine performance using relevance vector machine." International journal of green energy 12.3 (2015): 265-271.

⁵⁹Zhao, Qianqian, et al. "Relevance Vector Machine and Its Application in Gas Layer Classification." Journal of Computational Information Systems 9.20 (2013): 8343-8350.

⁶⁰See Haiyan Lu, Pichet Sriyanyong, Yong Hua Song, Tharam Dillon, Experimental study of a new hybrid PSO with mutation for economic dispatch with non-smooth cost function [J], International Journal of Electrical Power and Energy Systems, 32 (9), 2012, 921–935.

⁶¹Tong, Guangrong, and Siwei Li. "Construction and Application Research of Isomap-RVM Credit Assessment Model." Mathematical Problems in Engineering (2015).

Part IV

Neural Networks

The Basic Idea

A artificial neural network (ANN) is constructed from a number of interconnected nodes known as neurons. These are arranged into an input layer, a hidden layer and an output layer. The input nodes correspond to the number of features you wish to feed into the ANN and the number of output nodes correspond to the number of items you wish to predict. Figure 27.2 presents an overview of an artificial neural network topology (ANN). It has 2 input nodes, 1 hidden layer with 3 nodes and 1 output node.

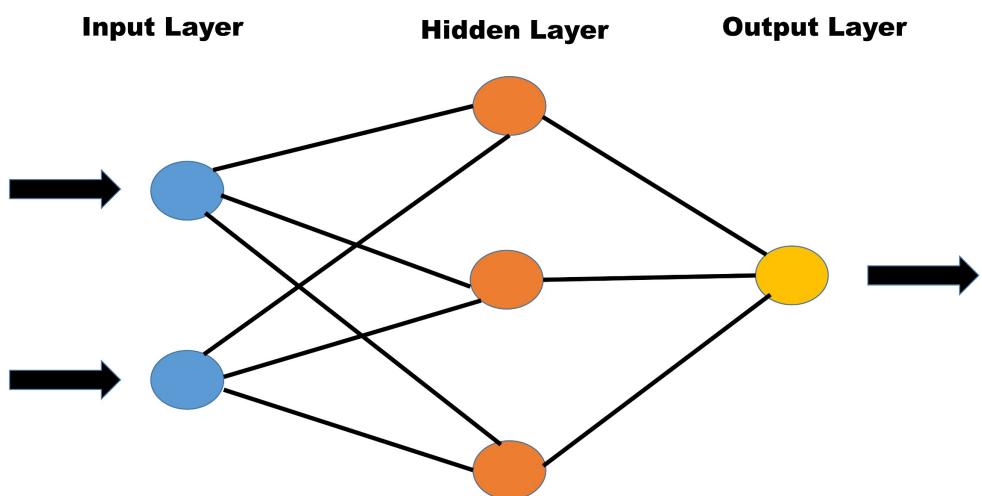


Figure 27.2: A basic neural network

The Neuron

At the heart of a neural network is the neuron. Figure 27.3 outlines the workings of an individual neuron. A weight is associated with each arc into

a neuron and the neuron then sums all inputs according to:

$$S_j = \sum_{i=1}^N w_{ij} a_i + b_j \quad (27.1)$$

The parameter b_j represents the bias associated with the neuron. It allows the network to shift the activation function “upwards” or “downwards”. This type of flexibility is important for successful machine learning.

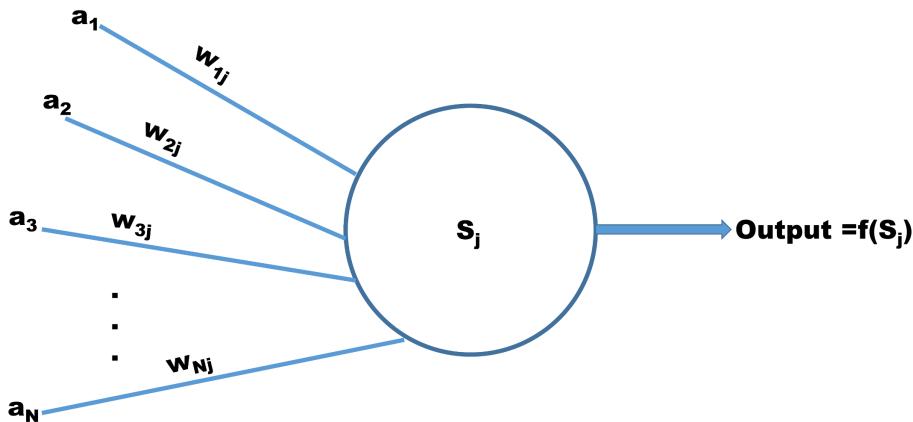


Figure 27.3: An artificial neuron

Activation and Learning

A neural network is generally initialized with random weights. Once the network has been initialized it is then trained. Training consists of two elements - activation and learning.

- STEP 1: An activation function $f(S_j)$ is applied and the output passed to the next neuron(s) in the network. The sigmoid function is a popular activation function:
- $$f(S_j) = \frac{1}{1 + \exp(-S_j)} \quad (27.2)$$
- STEP 2: A learning “law” that describes how the adjustments of the weights are to be made during training. The most popular learning law is backpropagation.

The Backpropagation Algorithm

It consists of the following steps.

1. The network is presented with input attributes and the target outcome
2. The output of the network is compared to the actual known target outcome.
3. The weights and biases of each neuron are adjusted by a factor based on the derivative of the activation function, the differences between the network output and the actual target outcome and the neuron outputs. Through this process the network “learns”.

Two parameters are often used to speed up learning and prevent the system from being trapped in a local minimum. They are known as the learning rate and the momentum.

☛ PRACTITIONER TIP ☛

ANNs are initialized by setting random values to the weights and biases. One rule of thumb is to set the random values to lie in the range (-2 k to 2 k), where k is the number of inputs.

How Many Nodes and Hidden Layers?

One of the very first questions asked about neural networks is how many nodes and layers should be included in the model? There are no fixed rules as to how many nodes to include in the hidden layer. However, as the number of hidden nodes increase so does the time taken for the network to learn from the input data.

Practical Applications

Sheet Sediment Transport

Tayfur⁶² model sediment transport using artificial neural networks (ANNs). The sample consisted of the original experimental hydrological data of Kilinic & Richardson⁶³.

A three-layer feed-forward artificial neural network with two neurons in the input layer, eight neurons in the hidden layer, and one neuron in the output layer was built. The sigmoid function was used as an activation function in the training of the network and the learning of the ANN was accomplished by the back-propagation algorithm. A random value of 0.2, and —1.0 were assigned for the network weights and biases, before starting the training process.

The ANNs performance was assessed relative to popular physical hydrological models (flow velocity, shear stress, stream power, and unit stream power) against a combination of slope types (mild, steep and very steep) and rain intensity (low, high, very high).

The ANN outperformed the popular hydrological models for very high intensity rainfall on both steep and very steep slope, see Table 15.

	Mild slope	Steep slope	Very steep slope
Low Intensity	physical	physical	ANN
High Intensity	physical	physical	physical
Very high intesity	physical	ANN	ANN

Table 15: Which model is best? Model transition framework derived from Tayfur's analysis.

◆ PRACTITIONER TIP ◆

Tayfur's results highlight an important issue facing the data scientist. Not only is it often a challenge to find the best model among competing candidates, it is even more difficult to identify a single model that works in all situations. A great solution, offered by Tayfur is to have a model transition matrix, that is determine which model(s) perform well under specific conditions and then use the appropriate model for a given condition.

Stock Market Volatility

Mantri et al⁶⁴ investigate the performance of a multilayer perceptron relative to standard econometric models of stock market volatility (GARCH, Exponential GARCH, Integrated GARCH, and the Glosten, Jagannathan and Runkle GARCH model).

Since the multilayer perceptron does not make assumptions about the distribution of stock market innovations it is of interest to Financial Analysts and Statisticians. The sample consisted of daily data collected on two Indian stock indices (BSE SENSEX and the NSE NIFTY) over the period January 1995 to December 2008.

The researchers found no statistical difference between the volatility of the stock indices estimated by the multilayer perceptron and standard econometric models of stock market volatility

Trauma Survival

The performance of trauma departments in the United Kingdom is widely audited by applying predictive models that assess the probability of survival, and examining the rate of unexpected survivals and deaths. The standard approach is the TRISS methodology which consists of two logistic regressions, one applied if the patient has a penetrating injury and the other applied for blunt injuries⁶⁵.

Hunter, Henry and Ferguson⁶⁶ assess the performance of the TRISS methodology against alternative logistic regression models and a multilayer perceptron.

The sample consisted of 15,055 cases, gathered from Scottish Trauma departments over the period 1992-1996. The data was divided into two subsets: the training set, containing 7,224 cases from 1992-1994; and the test set, containing 7,831 cases gathered from 1995-1996. The researcher's logistic regression models and the neural network were optimized using the training set.

The neural network was optimized ten times with the best resulting model selected. The researchers conclude that neural networks can yield better results than logistic regression.

☛ PRACTITIONER TIP ☛

The sigmoid function is a popular choice as an activation function. It is good practice to standardize (i.e. convert to the range (0,1)) all external input values before passing them into a neural network. This is because without standardization, large input values require very small weighting factors. This can cause two basic problems:

1. Inaccuracies introduced by very small floating point calculations on your computer.
2. Changes made by the back-propagation algorithm will be extremely small causing training to be slow (the gradient of the sigmoid function at extreme values would be approximately zero).

Brown Trout Reds

Lek et al⁶⁷ compare the ability of multiple regression and neural networks to predict the density of brown trout redds in southwest France. Twenty nine observation stations distributed on six rivers and divided into 205 morphodynamic units collected information on 10 ecological metrics.

The models were fitted using all the ecological variables and also with a sub-set of four variables. Testing consisted of random selection for the training set (75% of observations and the test set 25% of observations). The process was repeated a total of five times.

The average correlation between the observed and estimated values over the five samples is reported in Table 16. The researchers conclude that both multiple regression and neural networks can be used to predict the density of

brown trout redds, however the neural network model had better prediction accuracy.

Neural Network		Multiple Regression	
Train	Test	Train	Test
0.900	0.886	0.684	0.609

Table 16: Let et al's reported correlation coefficients between estimate and observed values in training and test samples

Electric Fish Localization

Weakly electric fish emit an electric discharge used to navigate the surrounding water and to communicate with other members of their shoal. Tracking of individual fish is often carried out using infrared cameras. However, this approach becomes unreliable when there is a visual obstruction⁶⁸.

Kiar et al⁶⁹ develop an non-invasive means of tracking weakly electric fish in real-time using a cascade forward neural network. The data set contained 299 data points which were interpolated to be 29,900 data points. The accuracy of the neural network was 94.3% within 1cm of actual fish location with a mean square error of 0.02 mm and a image frame rate of 213 Hz.

Chlorophyll Dynamics

Wu et al⁷⁰ developed two modeling approaches - artificial neural networks (ANN) and multiple linear regression (MLR), to simulate the daily Chlorophyll *a* dynamics in a northern German lowland river. Chlorophyll absorbs sunlight to synthesize carbohydrates from CO₂ and water. It is often used as a proxy for the amount of phytoplankton present in water bodies.

Daily Chlorophyll *a* samples were taken over an 18 month period. In total 426 daily samples were obtained. Each 10th daily sample was assigned to the validation set resulting in 42 daily observations. The calibration set contained 384 daily observations.

For ANN modelling a three layer back propagation neural network was used. The input layer consisted of 12 neurons corresponding to the independent variables shown in Table 17. The same independent variables were also used in the multiple regression model. The dependent variable in both models was the daily concentration of Chlorophyll *a*.

Air temperature	Ammonium nitrogen
Average daily discharge	Chloride
Chlorophyll a concentration	Daily precipitation
Dissolved inorganic nitrogen	Nitrate nitrogen
Nitrite nitrogen	Orthophosphate phosphorus
Sulfate	Total phosphorus

Table 17: Wu et al's input variables

The results of the ANN and MLR illustrated a good agreement between the observed and predicted daily concentration of Chlorophyll *a*, see Table 18.

Model	R-Square	NS	RMSE
MLR	0.53	0.53	2.75
NN	0.63	0.62	1.94

Table 18: Wu et al's performance metrics (NS = Nash Sutcliffe efficiency, RMSE = root mean square error)

☞ PRACTITIONER TIP ☞

Whilst there are no fixed rules about how to standardize inputs here are four popular choices for your original input variable x_i :

$$z_i = \frac{x_i - x_{min}}{x_{max} - x_{min}} \quad (27.3)$$

$$z_i = \frac{x_i - \bar{x}}{\sigma_x} \quad (27.4)$$

$$z_i = \frac{x_i}{\sqrt{SS_i}} \quad (27.5)$$

$$z_i = \frac{x_i}{x_{max} + 1} \quad (27.6)$$

SS_i is the sum of square of x_i , and \bar{x} and σ_x are the mean and standard deviation of x_i .

Examples of Neural Network Classification

Technique 28

Resilient Backpropagation with Backtracking

A neural network with resilient backpropagation & backtracking can be estimated using the package `neuralnet` with the `neuralnet` function:

```
neuralnet(y ~ ., data, hidden, algorithm = "rprop+"
, ...)
```

Key parameters include the response variable `y`, covariates `data`, `hidden` the number of hidden neurons and `algorithm = "rprop+"` to specify resilient backpropagation with backtracking.

Step 1: Load Required Packages

The neural network is built using the data frame `PimaIndiansDiabetes2` contained in the `mlbench` package. For additional details on this data set see page 52.

```
> library(neuralnet)
> data("PimaIndiansDiabetes2", package="mlbench")
```

Step 2: Prepare Data & Tweak Parameters

The `PimaIndiansDiabetes2` has a large number of misclassified values (recorded as `NA`) particularly for the attributes of `insulin` and `triceps`. We remove these two attributes from the sample and use the `na.omit` method to remove any remaining misclassified values. The cleaned data is stored in `temp`.

```
> temp<-(PimaIndiansDiabetes2)
> temp$insulin  <- NULL
> temp$triceps <- NULL
> temp<-na.omit(temp)
```

Next, we need to convert the response variable and attributes into a matrix and then use the `scale` method to standardize the matrix `temp`.

```
> y<-(temp$diabetes)
> levels(y) <- c("0","1")

> y<-as.numeric(as.character(y))
> y <-as.data.frame(y)

> names(y)<-c("diabetes")

> temp$diabetes<-NULL
> temp<-cbind(temp,y)
> temp<-scale(temp)
```

Now we can select the training sample. We choose to use 600 out of the 724 observations. The variable `f` is used to store the form of the model, where `diabetes` is the dependent or response variable. Be sure to check `n` is equal to 724, the number of observations in the full sample.

```
> set.seed(103)
> n=nrow(temp)

> train <- sample(1:n, 600, FALSE)

> f<- diabetes ~ pregnant+ glucose + pressure +
  mass + pedigree + age
```

Step 3: Estimate & Evaluate Model

The model is fitted using `neuralnet` with four hidden neurons.

```
> fit<- neuralnet(f, data = temp[train,],hidden=4,
  algorithm = "rprop+")
```

The `print` method gives a nice overview of the model.

```
> print(fit)
```

```
Call: neuralnet(formula = f, data = temp[train, ],  
    hidden = 4, algorithm = "rprop+")  
  
1 repetition was calculated.  
  
      Error Reached Threshold Steps  
1 181.242328     0.009057962229     8448
```

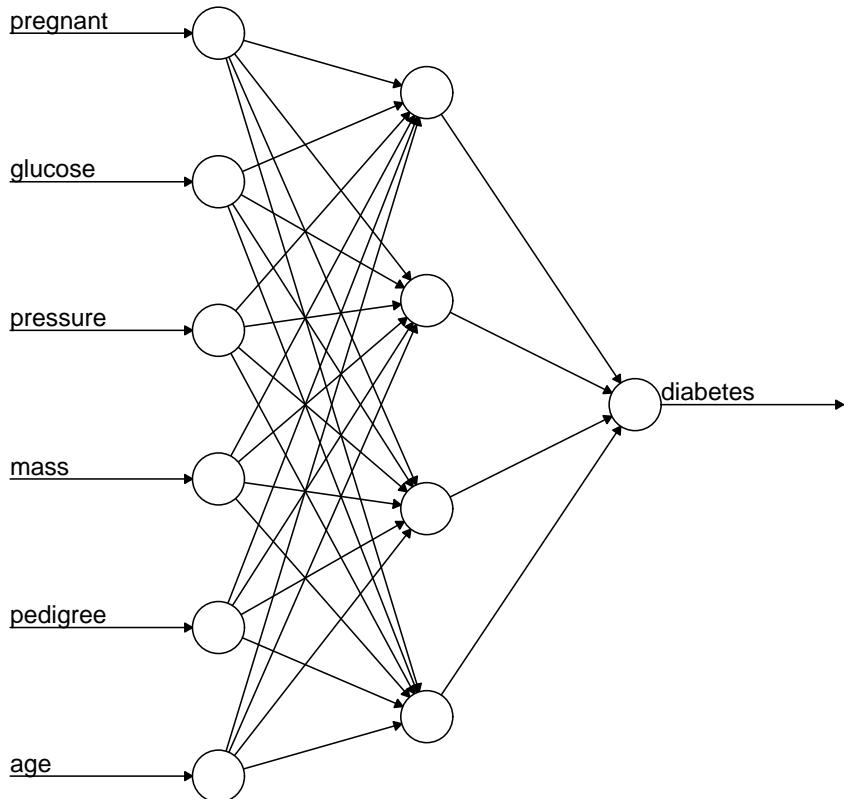
A nice feature of the `neuralnet` package is the ability to visualize a fitted network using the `plot` method, see Figure 28.1.

```
> plot(fit, intercept = FALSE, show.weights = FALSE)
```

☛ PRACTITIONER TIP ☛

It often helps intuition to visualize data. To see the fitted intercepts set `intercept = TRUE`; and to see the estimated neuron weights set `show.weights = TRUE`. For example, try entering:

```
plot(fit, intercept = TRUE,  
show.weights = TRUE)
```



Error: 181.242328 Steps: 8448

Figure 28.1: Resilient Backpropagation and Backtracking neural network using PimaIndiansDiabetes2

Step 4: Make Predictions

We transfer the data into a variable called `z` and use this with the `predict` method and the test sample.

```
> z<-temp
> z<-z[, -7]
> pred <- compute(fit, z[-train, ] )
```

The actual predictions should look something like this.

```
> sign(pred$net.result)
```

```
[ ,1]
4      -1
5      -1
7      -1
12     1
17     1
20     1
```

Let's create a confusion matrix, so we can see how well the neural network performed on the test sample.

```
> table( sign(pred$net.result)
,sign(temp[-train,7])
,dnn =c("Predicted"
," Observed"))

          Observed
Predicted -1   1
           -1 61   6
            1 20  37
```

We also need to calculate the error rate.

```
> error_rate = (1- sum(sign(pred$net.result) ==
sign(temp[-train,7])) / 124)
> round(error_rate,2)
[1] 0.21
```

The misclassification error rate is around 21%.

Technique 29

Resilient Backpropagation

A neural network with resilient backpropagation without backtracking can be estimated using the package `neuralnet` with the `neuralnet` function:

```
neuralnet(y ~ ., data, hidden, algorithm = "rprop-"  
,...)
```

Key parameters include the response variable `y`, covariates `data`, `hidden` the number of hidden neurons and `algorithm = "rprop-"` to specify resilient backpropagation.

Step 3: Estimate & Evaluate Model

Steps 1 and 2 are outlined beginning on page 226.

The model is fitted using `neuralnet` with four hidden neurons.

```
> fit<- neuralnet(f, data = temp[train,],hidden=4,  
algorithm = "rprop+")
```

The `print` method gives a nice overview of the model. The error is close to that observed for the neural network model estimated with backtracking outlined on page 226.

```
> print(fit)
```

```
Call: neuralnet(formula = f, data = temp[train, ],  
hidden = 4, algorithm = "rprop-")
```

```
1 repetition was calculated.
```

```
Error Reached Threshold Steps
1 184.0243459      0.009715675095   6814
```

Step 4: Make Predictions

We transfer the data into a variable called z and use this with the predict method and the test sample.

```
> z<-temp
> z<-z[,-7]
> pred <- compute(fit, z[-train,] )
```

The actual predictions should look something like this.

```
> sign(pred$net.result)
 [1]
4      -1
5       1
7      -1
12      1
17      1
```

Let's create a confusion matrix, so we can see how well the neural network performed on the test sample.

```
> table( sign(pred$net.result) ,
sign(temp[-train,7]) ,
dnn =c("Predicted" , " Observed"))

          Observed
Predicted -1   1
           -1 62   4
            1 19  39
```

We also need to calculate the error rate.

```
> error_rate = (1- sum( sign(pred$net.result) ==
sign(temp[-train,7]) ) / 124)
> round( error_rate ,2)
[1] 0.19
```

The misclassification error rate is around 19%.

Technique 30

Smallest Learning Rate

A neural network using the smallest learning rate can be estimated using the package `neuralnet` with the `neuralnet` function:

```
neuralnet(y ~ ., data, hidden, algorithm = "slr", ...)
```

Key parameters include the response variable `y`, covariates `data`, `hidden` the number of hidden neurons and `algorithm = "slr"` to use of a globally convergent algorithm that uses resilient backpropagation without weight backtracking and additionally the smallest learning rate.

Step 3: Estimate & Evaluate Model

Steps 1 and 2 are outlined beginning on page 226.

The model is fitted using `neuralnet` with four hidden neurons.

```
> fit<- neuralnet(f, data = temp[train,], hidden=4,  
+ algorithm = "slr")
```

The `print` method gives a nice overview of the model. The error is close to that observed for the neural network model estimated with backtracking outlined on page 226, however the algorithm (due to the small learning rate) takes very many more steps to converge.

```
> print(fit)
```

```
Call: neuralnet(formula = f, data = temp[train, ],  
+ hidden = 4, algorithm = "slr")
```

```
1 repetition was calculated.

      Error Reached Threshold Steps
1 179.7865898     0.009813138137 96960
```

Step 4: Make Predictions

We transfer the data into a variable called z and use this with the `predict` method and the test sample.

```
> z<-temp
> z<-z[, -7]
> pred <- compute(fit, z[-train, ] )
```

The actual predictions should look something like this.

```
> sign(pred$net.result)
 [ ,1]
4      -1
5      -1
7      -1
12      1
17      1
20      -1
```

Let's create a confusion matrix, so we can see how well the neural network performed on the test sample.

```
> table(sign(pred$net.result),
sign(temp[-train,7]) ,
dnn =c("Predicted" , " Observed"))

          Observed
Predicted -1   1
           -1 58 10
            1 23 33
```

We also need to calculate the error rate.

```
> error_rate = (1- sum(sign(pred$net.result) ==
sign(temp[-train,7])) / 124)
> round(error_rate ,2)
[1] 0.27
```

The misclassification error rate is around 27%.

Technique 31

Probabilistic Neural Network

A probabilistic neural network can be estimated using the package `pnn` with the `learn` function:

```
learn(y, data, ...)
```

Key parameters include the response variable `y` and the covariates contained in `data`.

Step 1: Load Required Packages

The neural network is built using the data frame `PimaIndiansDiabetes2` contained in the `mlbench` package. For additional details on this data set see page 52.

```
> library(pnn)
> data("PimaIndiansDiabetes2", package = "mlbench")
```

Step 2: Prepare Data & Tweak Parameters

The `PimaIndiansDiabetes2` has a large number of misclassified values (recorded as `NA`) particularly for the attributes of `insulin` and `triceps`. We remove these two attributes from the sample and use the `na.omit` method to remove any remaining misclassified values. The cleaned data is stored in `temp`.

```
> temp <- (PimaIndiansDiabetes2)
> temp$insulin <- NULL
> temp$triceps <- NULL
> temp <- na.omit(temp)
```

Next, we need to convert the response variable and attributes into a matrix and then use the `scale` method to standardize the matrix `temp`.

```
> temp<-(PimaIndiansDiabetes2)
> temp$insulin  <- NULL
> temp$triceps <- NULL
> temp<-na.omit(temp)

> y<-(temp$diabetes)

> temp$diabetes<-NULL
> temp<-scale(temp)
> temp<-cbind(as.factor(y),temp)
```

Now we can select the training sample. We choose to use 600 out of the 724 observations.

```
> set.seed(103)
> n=nrow(temp)

> n_train <- 600
> n_test<-n-n_train

> train <- sample(1:n, n_train, FALSE)
```

Step 3: Estimate & Evaluate Model

The model is fitted using the `learn` method with the fitted model stored in `fit_basic`.

```
> fit_basic <- learn(data.frame(y[train],
temp[train,]))
```

You can use the `attributes` method to identify the slots / characteristics of `fit_basic`.

```
> attributes(fit_basic)
$names
[1] "model"           "set"                "category."
   column" "categories"
[5] "k"               "n"
```

► PRACTITIONER TIP ◄

Remember you can access the contents of a fitted probabilistic neural network by using the \$ notation. For example, to see what is in the "model" slot you would type:

```
> fit_basic$model  
[1] "Probabilistic neural network"
```

The **summary** method provides details on the model.

```
> summary(fit_basic)  
      Length Class      Mode  
model        1   -none- character  
set          8   data.frame list  
category.column 1   -none- numeric  
categories    2   -none- character  
k            1   -none- numeric  
n            1   -none- numeric
```

Next we use the **smooth** method to set the smoothing parameter sigma. We use a value of 0.5.

```
> fit <- smooth(fit_basic, sigma=0.5)
```

► PRACTITIONER TIP ◄

Much of the time you will not have a pre-specified value in mind for the smoothing parameter sigma. However, you can let the **smooth** function find the best value using its inbuilt genetic algorithm. To do that you would type something along the lines of:

```
> smooth(fit_basic)
```

The performance statistics of the fitted model are assessed using the **perf** method. For example, enter the following to see various aspects of the fitted model.

```
> perf(fit)  
> fit$observed
```

```
> fit$guessed  
> fit$success  
> fit$fails  
> fit$success_rate  
> fit$bic
```

Step 4: Make Predictions

Let's take a look at the testing sample. To see the first row of covariates in the testing set enter:

```
> round(temp[-train,][1,],2)  
      pregnant glucose pressure  
    1.00     -0.85     -1.07     -0.52  
  
mass pedigree age  
-0.63     -0.93     -1.05
```

You can see the first observation of the response variable in the test sample in a similar way.

```
> y[-train][1]  
[1] neg  
Levels: neg pos
```

Now let's predict the first response value in the test set using the covariates.

```
> guess(fit, as.matrix(temp[-train,][1,]))$category  
[1] "neg"
```

Take a look at the associated probabilities. In this case there is a 99% probability associated with the neg class.

```
> guess(fit, as.matrix(temp[-train,][1,]))$  
probabilities  
      neg          pos  
0.996915706 0.003084294
```

Here is how to see both prediction and associated probabilities.

```
> guess(fit, as.matrix(temp[-train,][1,]))  
$category  
[1] "neg"  
  
$probabilities
```

```
    neg          pos  
0.996915706 0.003084294
```

OK, now we are ready to predict all the response values in the test sample. We can do this with a few lines of R code.

```
> pred<-1:n_test  
> for (i in 1:n_test)  
{  
  pred[i]<-guess(fit, as.matrix(temp[-train,][i,]))$  
    category  
}
```

Let's create a confusion matrix, so we can see how well the neural network performed on the test sample.

```
> table( pred,y[-train] ,  
dnn =c("Predicted" , " Observed"))  
  
          Observed  
Predicted neg pos  
  neg     79   2  
  pos      2  41
```

We also need to calculate the error rate.

```
> error_rate = (1- sum( pred == y[-train]) / n_test  
)  
> round( error_rate ,3)  
[1] 0.032
```

The misclassification error rate is around 3%.

Technique 32

Multilayer Feedforward Neural Network

A multilayer feedforward neural network can be estimated using the AMORE package with the `train` function:

```
train(net, P,T, error.criterium,...)
```

Key parameters include `net` the neural network you wish to train, `P` training set attributes, `T` the training set response variable/ output values, and the error criteria (Least Mean Squares or Least Mean Logarithm Squared or TAO Error) contained in `error.criterium`.

Step 1: Load Required Packages

The neural network is built using the data frame `PimaIndiansDiabetes2` contained in the `mlbench` package. For additional details on this data set see page 52.

```
> library(AMORE)
> data("PimaIndiansDiabetes2", package = "mlbench")
```

Step 2: Prepare Data & Tweak Parameters

The `PimaIndiansDiabetes2` has a large number of misclassified values (recorded as `NA`) particularly for the attributes of `insulin` and `triceps`. We remove these two attributes from the sample and use the `na.omit` method to remove any remaining misclassified values. The cleaned data is stored in `temp`.

```
> temp<-(PimaIndiansDiabetes2)
> temp$insulin  <- NULL
> temp$triceps <- NULL
> temp<-na.omit(temp)
```

Next, we need to convert the response variable and attributes into a matrix and then use the `scale` method to standardize the matrix `temp`.

```
> y<-(temp$diabetes)
> levels(y) <- c("0","1")

> y<-as.numeric(as.character(y))

> names(y)<-c("diabetes")

> temp$diabetes<-NULL
> temp<-cbind(temp,y)
> temp<-scale(temp)
```

Now we can select the training sample. We choose to use 600 out of the 724 observations.

```
> set.seed(103)
> n=nrow(temp)
> train <- sample(1:n, 600, FALSE)
```

Now we need to create the neural network we wish to train.

```
> net <- newff(n.neurons=c(1,3,2,1),
learning.rate.global=0.01,
momentum.global=0.5,
error.criterium="LMSE", Stao=NA,
hidden.layer="sigmoid",
output.layer="purelin",
method="ADAPTgdwm")
```

I'll explain the code above line by line. In the first line, we're creating an object called `net` that will contain the structure of our new neural network. The first argument to `newff` is `n.neurons` which allows us to specify the number of inputs, number of nodes in each hidden layer, and number of outputs. So in the example above, we have 1 input, 2 hidden layers, the first containing 3 nodes and the second containing 2 nodes, and 1 output.

The `learning.rate.global` argument constrains how much the algorithm is allowed to change the weights from iteration to iteration as the network is trained. In this case, `learning.rate.global = 0.01` which means

that the algorithm can't increase or decrease any one weight in the network by any more than 0.01 from trial to trial.

The `error.criterium` argument specifies the error mechanism used at each iteration. There are three options here: "LMS" (for least mean squares), "LMLS" (for least-mean-logarithm-squared), and "TAO" (for the Tao error method). In general, I tend to choose "LMLS" as my starting point. However, I will often train my networks using all three methods.

The `hidden.layer` and `output.layer` arguments are used to choose the type of activation function you will use to interpret the summations of the inputs and weights for any of the layers in your network. I have set `output.layer="purelin"`, which results in linear output. Other options include, "`tansig`", "`sigmoid`", "`hardlim`" and "`custom`".

Finally, `method` specifies the solution strategy for converging on the weights within the network. For building prototype models I tend to use either "`ADAPTgd`" (adaptive gradient descend) or "`ADAPTgdwm`" (adaptive gradient descend with momentum).

Step 3: Estimate & Evaluate Model

The model is fitted using the `train` method. In this example I have set `report=TRUE` to provide output during the algorithm run, and `n.shows=5` to show a total of 5 training epochs.

```
> fit <- train(net, P=temp[train,],  
T=temp[train,7],  
error.criterium="LMLS",  
report=TRUE,  
show.step=100,  
n.shows=5)  
  
index.show: 1 LMLS 0.337115972269707  
index.show: 2 LMLS 0.335651051328758  
index.show: 3 LMLS 0.335113569553075  
index.show: 4 LMLS 0.334753676125557  
index.show: 5 LMLS 0.334462044665089
```

Step 4: Make Predictions

The `sign` function is used to convert predictions into negative and positive. Here I use the fitted network held in `fit` to predict using the test sample.

```
> pred <- sign(sim(fit$net, temp[-train,]))
```

Let's create a confusion matrix, so we can see how well the neural network performed on the test sample. Notice that `sign(temp[-train,7])` contains the observed output/ response values for the test sample.

```
> table( pred, sign(temp[-train,7]) ,  
dnn =c("Predicted" , " Observed"))  
          Observed  
Predicted -1   1  
      -1 68 27  
      1 13 16
```

We also need to calculate the error rate.

```
> error_rate = (1- sum( pred == sign(temp[-train,7])  
) / 124)  
> round(error_rate ,3)  
[1] 0.323
```

The misclassification error rate is around 32%.

Examples of Neural Network Regression

Technique 33

Resilient Backpropagation with Backtracking

A neural network with resilient backpropagation and backtracking can be estimated using the package `neuralnet` with the `neuralnet` function:

```
neuralnet(y ~ ., data, hidden, algorithm = "rprop+"
, ...)
```

Key parameters include the response variable `y`, covariates `data`, `hidden` the number of hidden neurons and `algorithm = "rprop+"` to specify resilient backpropagation with backtracking.

Step 1: Load Required Packages

The neural network is built using the data frame `bodyfat` contained in the `TH.data` package. For additional details on this data set see page 62.

```
> library(neuralnet)
> data("bodyfat", package="TH.data")
```

Step 2: Prepare Data & Tweak Parameters

We use 45 out of the 71 observations in the training sample with the remaining saved for testing. The variable `train` selects the random sample without replacement from the 71 observations. The response variable and attributes are structured and stored in the variable `f`. In addition the standardized variables are stored in the data frame `scale_bodyfat`.

```
> set.seed(465)
```

```
> train <- sample(1:71, 45 , FALSE)

> f<- DEXfat ~ waistcirc + hipcirc
+age + elbowbreadth + kneebreadth + anthro3a
+ anthro3b + anthro3c +anthro4

> scale_bodyfat<-as.data.frame(scale(bodyfat))
```

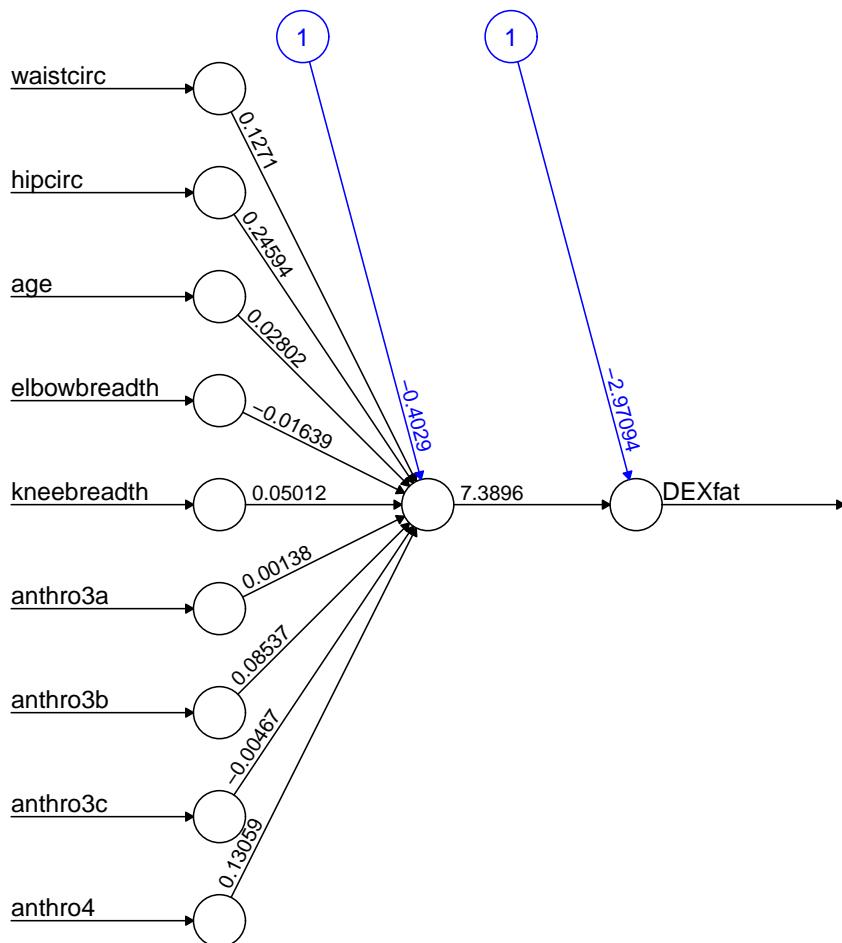
Step 3: Estimate & Evaluate Model

The number of hidden neurons should be determined in relation to the needed complexity. For this example we use one hidden neuron.

```
> fit<- neuralnet(f, data = scale_bodyfat[train,] ,
  hidden=1,algorithm = "rprop+")
```

The `plot.nn` method can be used to visualize the network, as shown in Figure 33.1.

```
> plot.nn(fit)
```



Error: 1.711438 Steps: 2727

Figure 33.1: Neural network used for the bodyfat regression

The `print` method gives a nice overview of the model.

```
> print(fit)
Call: neuralnet(formula = f, data = scale_bodyfat[
  train, ],
hidden = 1,      algorithm = "rprop+")

1 repetition was calculated.
```

	Error	Reached Threshold	Steps
1	1.711437513	0.00955254228	2727

A nice feature of the `neuralnet` package is the ability to print a summary of the results matrix.

```
> round(fit$result.matrix,3)
              1
error          1.711
reached.threshold 0.010
steps          2727.000
Intercept.to.1layhid1 -0.403
waistcirc.to.1layhid1  0.127
hipcirc.to.1layhid1   0.246
age.to.1layhid1      0.028
elbowbreadth.to.1layhid1 -0.016
kneebreadth.to.1layhid1  0.050
anthro3a.to.1layhid1   0.001
anthro3b.to.1layhid1   0.085
anthro3c.to.1layhid1   -0.005
anthro4.to.1layhid1    0.131
Intercept.to.DEXfat   -2.971
1layhid.1.to.DEXfat   7.390
```

Step 4: Make Predictions

We predict using the scaled values, then remove the response variable `DEXfat` storing the predictions in `pred` via the `compute` method.

```
> without_fat<-scale_bodyfat
> without_fat$DEXfat <-NULL

> pred <- compute(fit, without_fat[-train,] )
```

Next we build a linear regression between the predicted and observed values in the training set, storing the model in `linReg`. Next the `plot` method is used to visualize the relationship between the predicted and observed values, see Figure 33.2. The `abline` method plots the linear regression line fitted by `linReg`. Finally, we call the `cor` method to calculate the squared correlation coefficient between the predicted and observed values, it returns a value of 0.864.

```
> linReg<-lm(pred$net.result~ scale_bodyfat$DEXfat
[-train])

> plot(scale_bodyfat$DEXfat[-train],
```

```
pred$net.result ,  
xlab="DEXfat" ,  
ylab="Predicted Values" ,  
main="Training Sample Model Fit")  
  
> abline(linReg , col="darkred")  
  
> round(cor(pred$net.result , scale_bodyfat$DEXfat [-  
train])^2 , 6)  
[ , 1]  
[1 , ] 0.864411
```

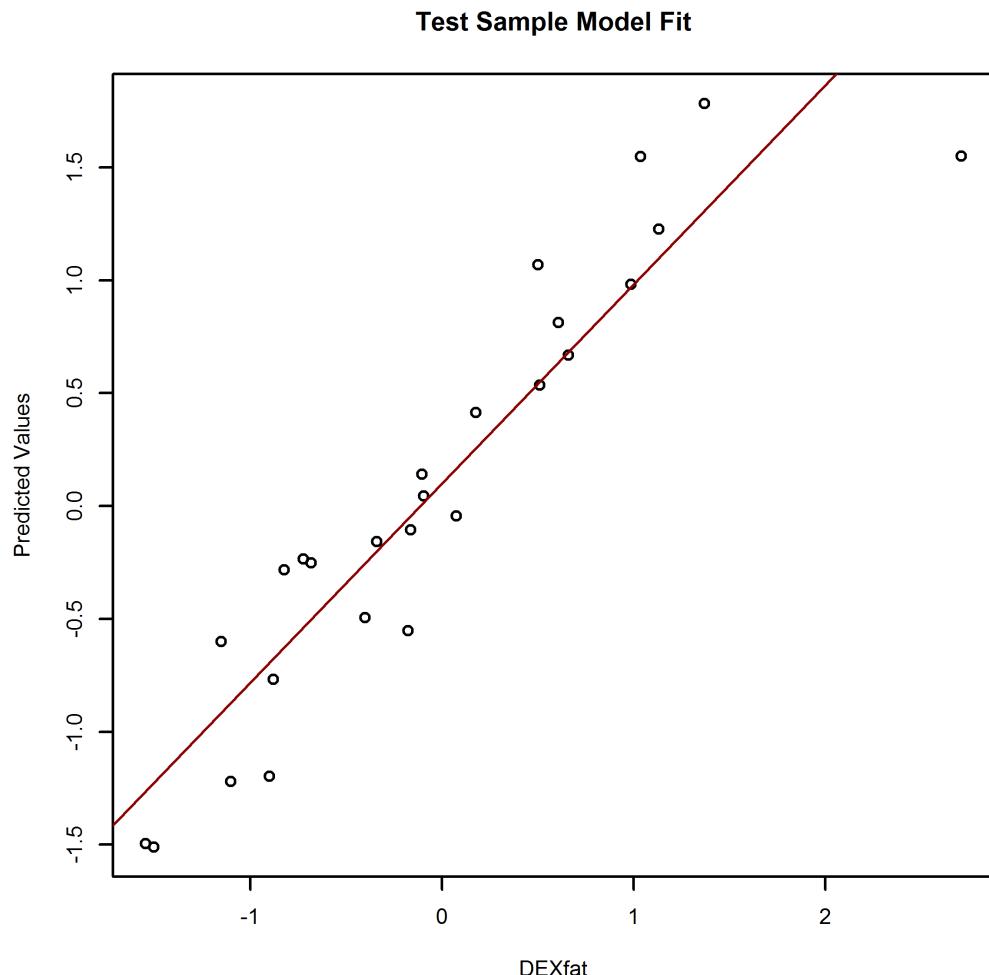


Figure 33.2: Resilient Backpropagation with Backtracking observed and predicted values using `bodyfat`

Technique 34

Resilient Backpropagation

A neural network with resilient backpropagation & backtracking can be estimated using the package **neuralnet** with the **neuralnet** function:

```
neuralnet(y ~ ., data, hidden, algorithm = "rprop-"  
,...)
```

Key parameters include the response variable **y**, covariates **data**, **hidden** the number of hidden neurons and **algorithm = "rprop-"** to specify resilient backpropagation.

Step 3: Estimate & Evaluate Model

Steps 1 and 2 are outlined beginning on page 245.

We fit a network with one hidden neuron as follows.

```
> fit<- neuralnet(f, data = scale_bodyfat[train,],  
+ hidden=1, algorithm = "rprop-")
```

The **print** method gives a nice overview of the model.

```
> print(fit)  
Call: neuralnet(formula = f, data = scale_bodyfat[  
  train, ], hidden = 1,      algorithm = "rprop-")  
  
1 repetition was calculated.
```

	Error	Reached Threshold	Steps
1	1.716273738	0.009965580594	3651

A nice feature of the `neuralnet` package is the ability to print a summary of the results matrix.

```
> round(fit$result.matrix,2)
           1
error      1.72
reached.threshold 0.01
steps      3651.00
Intercept.to.1layhid1 -0.37
waistcirc.to.1layhid1  0.13
hipcirc.to.1layhid1   0.26
age.to.1layhid1       0.03
elbowbreadth.to.1layhid1 -0.02
kneebreadth.to.1layhid1  0.05
anthro3a.to.1layhid1   0.00
anthro3b.to.1layhid1   0.09
anthro3c.to.1layhid1   0.00
anthro4.to.1layhid1    0.14
Intercept.to.DEXfat   -2.86
1layhid.1.to.DEXfat   6.97
```

Step 4: Make Predictions

We predict using the scaled values and the `compute` method.

```
> scale_bodyfat$DEXfat <- NULL
> pred <- compute(fit, scale_bodyfat[-train,] )
```

Next we build a linear regression between the predicted and observed values in the training set, storing the model in `linReg`.

The `plot` method is used to visualize the relationship between the predicted and observed values, see Figure 34.1. The `abline` method plots the linear regression line fitted by `linReg`. Finally, we call the `cor` method to calculate the squared correlation coefficient between the predicted and observed values, it returns a value of 0.865,

```
> linReg<-lm(pred$net.result~ bodyfat$DEXfat [-train])
> plot(bodyfat$DEXfat[-train],
pred$net.result,
xlab="DEXfat",
ylab="Predicted Values",
```

```
main="Test Sample Model Fit")  
  
> abline(linReg,col="darkred")  
  
> round(cor(pred$net.result, bodyfat$DEXfat[-train])^2,3)  
[1,] 0.865
```

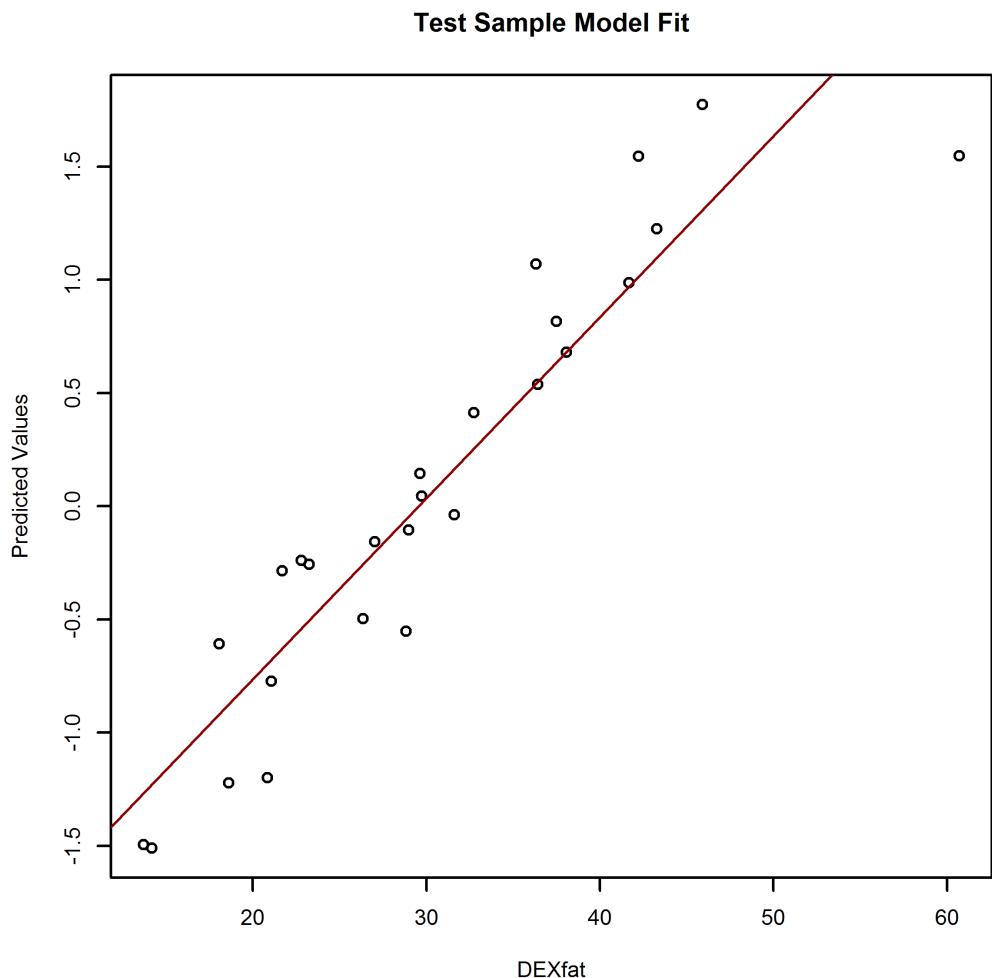


Figure 34.1: Resilient Backpropagation without Backtracking observed and predicted values using `bodyfat`

Technique 35

Smallest Learning Rate

A neural network using the smallest learning rate can be estimated using the package **neuralnet** with the **neuralnet** function:

```
neuralnet(y ~ ., data, hidden, algorithm = "slr", ...)
```

Key parameters include the response variable **y**, covariates **data**, **hidden** the number of hidden neurons and **algorithm = "slr"** which specifies the use of a globally convergent algorithm with resilient backpropagation and the smallest learning rate.

Step 3: Estimate & Evaluate Model

Steps 1 and 2 are outlined beginning on page 245.

The network is built with 1 hidden neuron as follows.

```
> fit<- neuralnet(f, data = scale_bodyfat[train,],  
+ hidden=1,algorithm = "slr")
```

The **print** method gives a nice overview of the model.

```
> print(fit)  
Call: neuralnet(formula = f, data = scale_bodyfat[  
  train, ], hidden = 1,      algorithm = "slr")  
  
1 repetition was calculated.
```

	Error Reached	Threshold	Steps
1	1.714864115	0.009544470987	10272

Here is a summary of the fitted model weights.

```
> round(fit$result.matrix,2)
              1
error          1.71
reached.threshold 0.01
steps         10272.00
Intercept.to.1layhid1 -0.38
waistcirc.to.1layhid1  0.13
hipcirc.to.1layhid1   0.26
age.to.1layhid1      0.03
elbowbreadth.to.1layhid1 -0.02
kneebreadth.to.1layhid1  0.05
anthro3a.to.1layhid1   0.00
anthro3b.to.1layhid1   0.09
anthro3c.to.1layhid1   0.00
anthro4.to.1layhid1    0.13
Intercept.to.DEXfat   -2.88
1layhid.1.to.DEXfat    7.08
```

Step 4: Make Predictions

We predict using the scaled values and the `compute` method.

```
> scale_bodyfat$DEXfat <-NULL
> pred <- compute(fit, scale_bodyfat[-train,] )
```

Next we build a linear regression between the predicted and observed values in the training set, storing the model in `linReg`. The `plot` method is used to visualize the relationship between the predicted and observed values, see Figure 35.1. The `abline` method plots the linear regression line fitted by `linReg`. Finally, we call the `cor` method to calculate the squared correlation coefficient between the predicted and observed values, it returns a value of 0.865,

```
> linReg<-lm(pred$net.result~ bodyfat$DEXfat [-train])
>
> plot(bodyfat$DEXfat[-train],
pred$net.result,
xlab="DEXfat",
ylab="Predicted Values",
main="Test Sample Model Fit")
```

```
> abline(linReg, col="darkred")  
  
> round(cor(pred$net.result, scale_bodyfat$DEXfat[-  
  train])^2, 3)  
[1,] [,1]  
[1,] 0.865
```

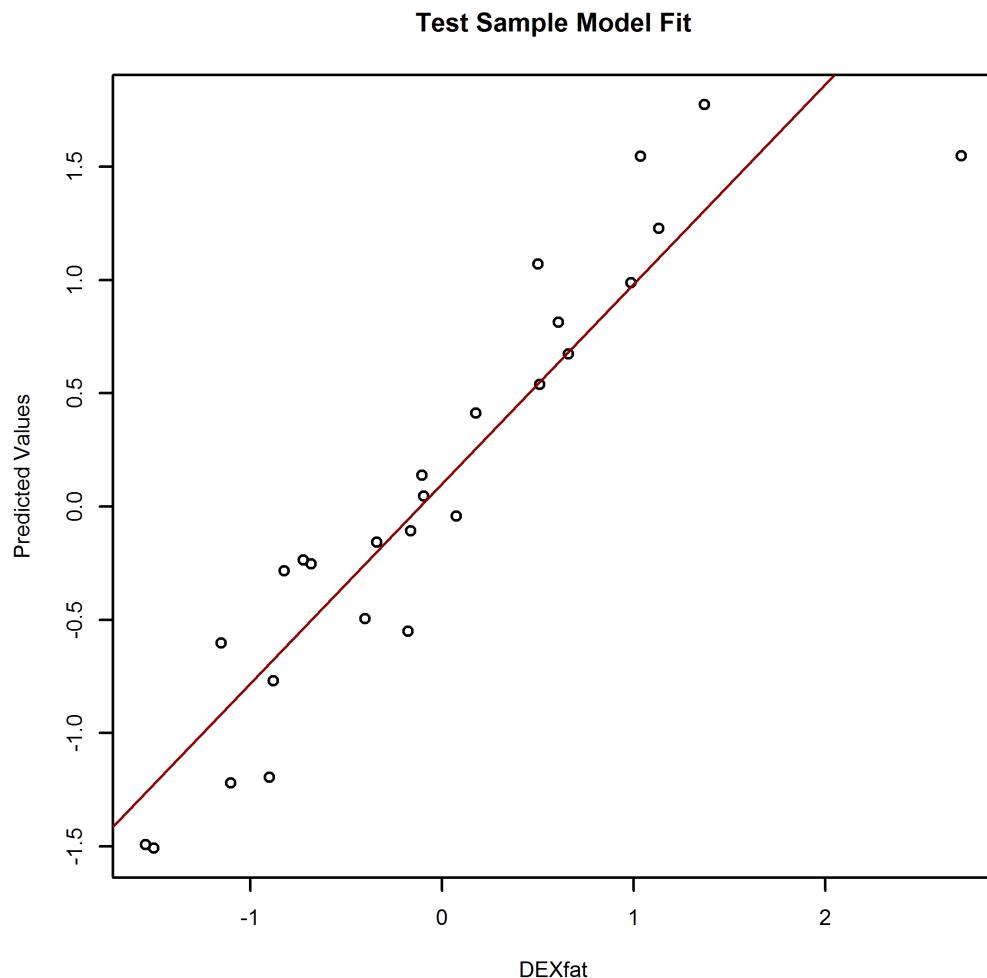


Figure 35.1: Resilient Backpropagation with smallest learning rate and predicted values using bodyfat

Technique 36

General Regression Neural Network

A General Regression Neural Network neural network can be estimated using the package `grnn` with the `learn` function:

```
learn(y, x)
```

Key parameters include the response variable `y` and the explanatory variables contained in `x`.

Step 1: Load Required Packages

The neural network is built using the data frame `bodyfat` contained in the `TH.data` package. For additional details on this data set see page 62.

```
> require(grnn)
> data("bodyfat", package = "TH.data")
```

Step 2: Prepare Data & Tweak Parameters

We use 45 out of the 71 observations in the training sample with the remaining saved for testing. The variable `train` selects the random sample without replacement from the 71 observations.

```
> set.seed(465)
> n <- nrow(bodyfat)

> n_train <- 45
> n_test <- n - n_train
```

```
> train <- sample(1:n, n_train , FALSE)
```

Next, we take the log of the response variable and store it in `y`. We do the same for the attributes variables, storing them in `x`.

```
> y<-log(bodyfat$DEXfat)
> x<-log(bodyfat)
> x$DEXfat<-NULL
```

Step 3: Estimate & Evaluate Model

We use the `learn` method to fit the model and the `smooth` method to smooth the general regression neural network. We set `sigma` to 2.5 in the `smooth` function and store the resultant model in `fit`.

```
> fit_basic <- learn(data.frame(y[train],x[train,]))
> fit <- smooth(fit_basic, sigma = 2.5)
```

Step 4: Make Predictions

Let's take a look at the first row of the covariates in the testing set

```
> round(x[-train,][1,],2)

    age waistcirc hipcirc elbowbreadth
47  4.04       4.61     4.72        1.96

kneebreadth anthro3a anthro3b anthro3c
                2.24      1.49      1.6       1.5

    anthro4
47      1.81
```

To see the first observation of the response variable in the test sample you can use a similar technique.

```
> y[-train][1]
[1] 3.730021
```

Now we are ready to predict the first response value in the test set using the covariates. Here is how to do that.

```
> guess(fit, as.matrix(x[-train,][1,]))
[1] 3.374901
```

Of course, we will want to predict using all the values in the test sample. We can achieve this using the `guess` method and the following lines of R code.

```
> pred<-1:n_test  
  
> for (i in 1:n_test)  
{  
pred[i]<-guess(fit, as.matrix(x[-train,][i,]))  
}
```

Finally we plot in Figure 36.1 the observed and fitted values using the `plot` method and on the same chart plot the regression line of the predicted versus observed values. The overall squared correlation is 0.915.

```
> plot(y[-train],pred,  
xlab="log(DEXfat)",  
ylab="Predicted Values",  
main="Test Sample Model Fit")  
  
> abline(linReg<-lm(pred~y[-train]),col="darkred")  
  
> round(cor(pred,y[-train])^2,3)  
[1] 0.915
```

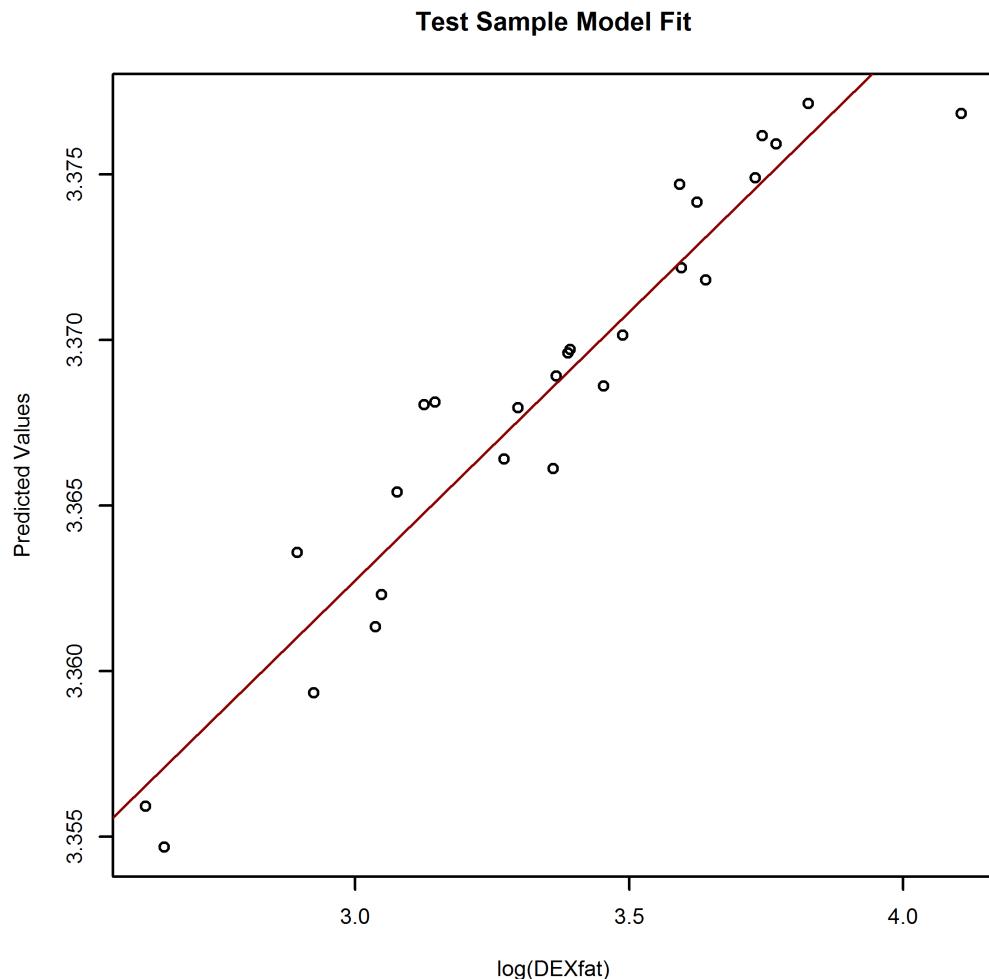


Figure 36.1: General Regression Neural Network observed and predicted values using `bodyfat`

Technique 37

Monotone Multi-Layer Perceptron

On occasion you will find yourself modeling variables for which you require monotonically increasing behavior of the response variables with respect to specified attributes. It is possible to train and make predictions from a multi-layer perceptron neural network with partial monotonicity constraints. The `monmlp` package implements the monotone multi-layer perceptron neural network using the approach of Zhang and Zhang⁷¹. The model can be estimated using the `monmlp.fit` function:

```
monmlp.fit(x, y, hidden1, hidden2, monotone)
```

Key parameters include the response variable `y`, the explanatory variables contained in `x`, the number of neurons in the first and second hidden layers (`hidden1` and `hidden2`) and the monotone constraints on the covariates.

Step 1: Load Required Packages

The neural network is built using the data frame `bodyfat` contained in the `TH.data` package. For additional details on this data set see page 62.

```
> library(monmlp)
> data("bodyfat", package = "TH.data")
```

Step 2: Prepare Data & Tweak Parameters

We use 45 out of the 71 observations in the training sample with the remaining saved for testing. The variable `train` selects the random sample without replacement from the 71 observations.

```
> set.seed(465)
> train <- sample(1:71, 45 , FALSE)
```

Next, we take the log of the response variable and store it in `y`. We do the same for the attributes variables, storing them as a matrix in `x`.

```
> y<-log(bodyfat$DEXfat)
> x<-log(bodyfat)

> x$DEXfat<-NULL
> x<-as.matrix(x)
```

Step 3: Estimate & Evaluate Model

We build a 1 hidden layer network with 1 node with monotone constraints on all 9 attributes contained in `x` (we set `monotone =1:9`).

```
> fit<- monmlp.fit(x=x[train,] , y=as.matrix(y[train])
]) ,hidden1=1,hidden2=0,monotone =1:9)
```

NOTE... ↗

The method `monmlp.fit` takes a number of additional arguments. Three that you will frequently want to set manually include:

- `n.trials` - the number of repeated trials used to avoid local minima.
- `n.ensemble` - the number of ensemble members to fit.
- `bag` - A logical variable indicating whether or not bootstrap aggregation (bagging) should be used.

For example, in the model we are building we could have specified:

```
fit<- monmlp.fit(x=x[train,] ,
y=as.matrix(y[train]),
hidden1=1,hidden2=0,
monotone =1:9,
n.trials=100, n.ensemble=50,bag=TRUE)
```

We take a look at the fitted values using the `plot` method, see Figure 37.1.

```
> plot(attributes(fit)$y, attributes(fit)$y.pred,  
      ylab="Fitted Values", xlab="Observed Values")
```

Since the fit looks particularly good, we better calculate the correlation coefficient. At 0.97 it is pretty high.

```
> cor(attributes(fit)$y, attributes(fit)$y.pred)  
[1,] 0.973589
```

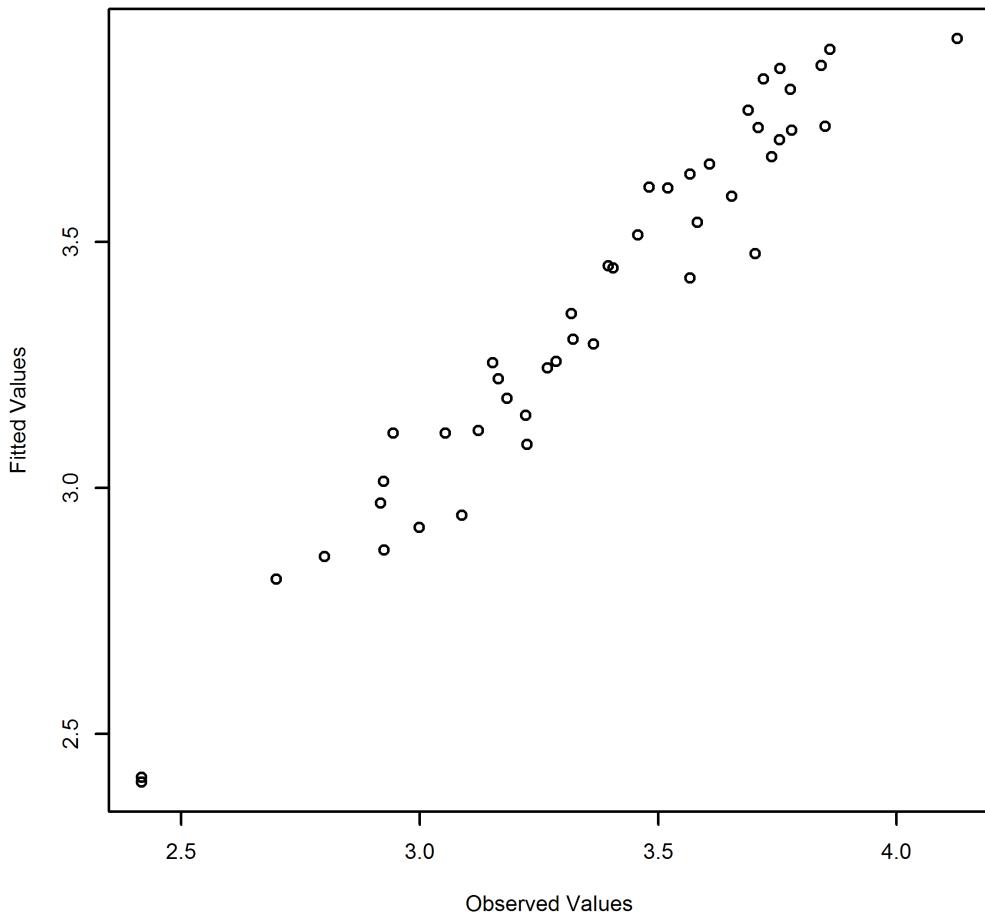


Figure 37.1: Monotone Multi-Layer Perceptron training observed and predicted values using `bodyfat`

Step 4: Make Predictions

We predict using the scaled variables and the test sample with the `monmlp.predict` method.

```
> pred <- monmlp.predict(x = x[-train,], weights = fit)
```

Next, a liner regression model is used to fit the predicted and observed values for the test sample.

```
> linReg<-lm(pred~y[-train])
```

Now plot the result (see Figure 37.2) and calculate the squared correlation coefficient.

```
> plot(y[-train],pred,xlab="DEXfat", ylab="Predicted Values", main="Test Sample Model Fit")
> abline(linReg,col="darkred")

> round(cor(pred,y[-train])^2,3)
[1] 0.958
```

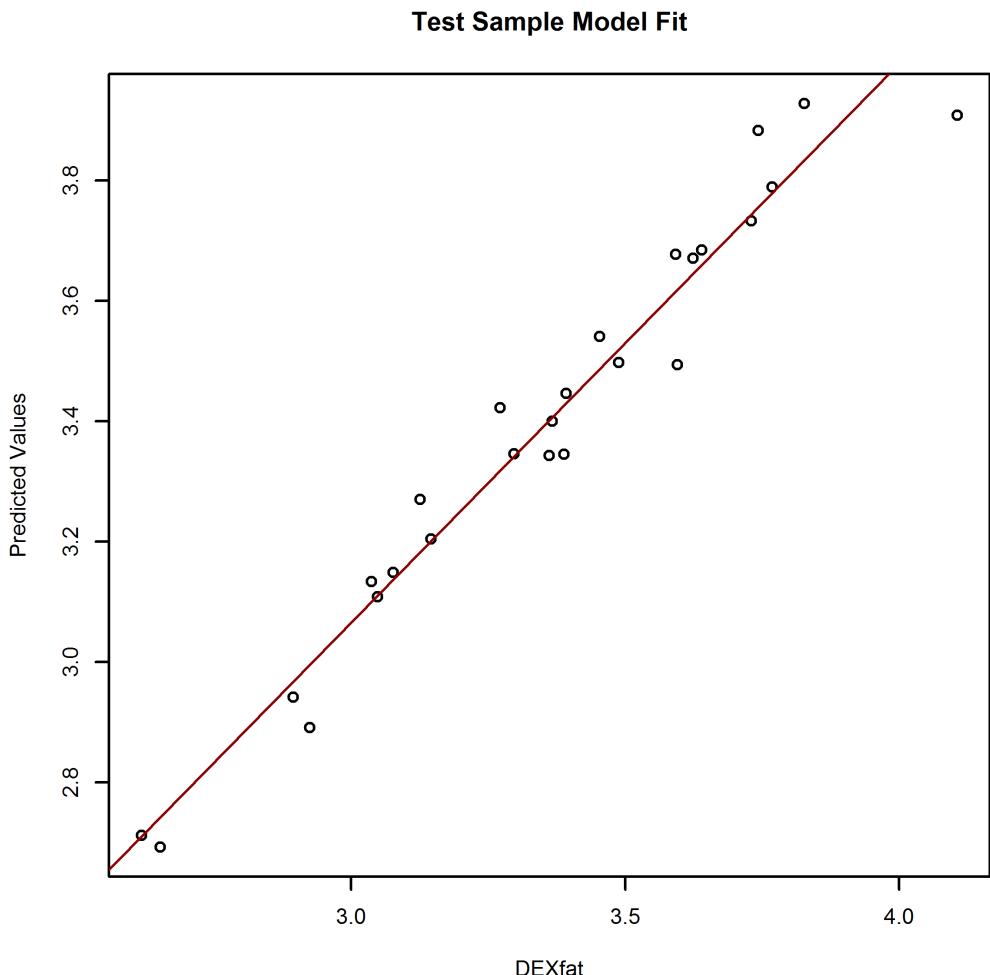


Figure 37.2: Monotone Multi-Layer Perceptron observed and predicted values using `bodyfat`

Technique 38

Quantile Regression Neural Network

Quantile regression models the relationship between the independent variables and the conditional quantiles of the response variable. It provides a more complete picture of the conditional distribution of the response variable when both lower and upper or all quantiles are of interest. The `qrnn` package combined with the `qrnn.fit` function can be used to estimate a quantile regression neural network.

```
qrnn.fit(x, y, n.hidden, tau, ...)
```

Key parameters include the response variable `y`, the explanatory variables contained in `x`, the number of hidden neurons `n.hidden` and the quantiles to be fitted held in `tau`.

Step 1: Load Required Packages

The neural network is built using the data frame `bodyfat` contained in the `TH.data` package. For additional details on this data set see page 62.

```
> data("bodyfat", package = "TH.data")
> library(qrnn)
> library(scatterplot3d)
```

Step 2: Prepare Data & Tweak Parameters

We use 45 out of the 71 observations in the training sample with the remaining saved for testing. The variable `train` selects the random sample without replacement from the 71 observations.

```
> set.seed(465)
> train <- sample(1:71, 45 , FALSE)
```

Next, we take the log of the response variable and store it in `y`. We do the same for the attributes variables, storing them as a matrix in `x`.

```
> y<-log(bodyfat$DEXfat)
> x<-log(bodyfat)

> x$DEXfat<-NULL
> x<-as.matrix(x)
```

We will estimate the model using the 5th, 50th and 95th quantiles.

```
probs <- c(0.05, 0.50, 0.95)
```

Step 3: Estimate & Evaluate Model

We estimate the model using the following:

```
> fit <- pred <- list()

> for(i in seq_along(probs))
{
  fit[[i]] <- qrnn.fit(x = x[train,],
y = as.matrix(y[train]),
n.hidden = 4,
tau = probs[i],
iter.max = 1000,
n.trials = 1,
n.ensemble=20)
}
```

Let's go through it line by line. The first line sets up as a list `fit` which will contain the fitted model, and `pred` which will contain the predictions. Because we have three quantiles held in `probs` to estimate we use a for loop as the core work engine. The function `qrnn.fit` is used to fit the model for each quantile.

A few other things are worth pointing out here. First, we build the model with four hidden neurons (`n.hidden = 4`). Second, we set the maximum number of iterations of the optimization algorithm at 1000. Third we set `n.trials = 1`, this parameter controls the number of repeated trials used to avoid local minima. We set it to 1 for illustration. In practice you should set

this to a much higher value. The same is true for the parameter `n.ensemble` which we set to 20. It controls the number of ensemble members to fit. You will generally want to set this to a high value also.

Step 4: Make Predictions

For each quantile the `qrnn.predict` method is used alongside the fitted model. Again, we choose to use a for loop to capture each quantiles predicted value.

```
> for(i in seq_along(probs))
{
  pred[[i]] <- qrnn.predict(x[-train,], fit[[i]])
}
```

Next, we create three variables to hold the predicted values. Note there are 26 sample values in total.

```
> pred_high<-pred_low<-pred_mean<-1:26
```

Now we are ready to store the predictions. Since we use 20 ensembles for each quantile, we store the average prediction using the `mean` method.

```
> for (i in 1:26)
{
  pred_low[i]<- mean(pred[[1]][i,])
  pred_mean[i]<- mean(pred[[2]][i,])
  pred_high[i]<- mean(pred[[3]][i,])
}
```

Now we fit a linear regression model using the 50th percentile values held in `pred_mean` and the actual observed values. The result is visualized using the `plot` method and shown in Figure 38.1. The squared correlation coefficient is then calculated. It is fairly high at 0.885.

```
> linReg2<-lm(pred_mean~ y [-train])

> plot(y[-train],pred_mean,xlab="log(DEXfat)",
       ylab="Predicted Values",
       main="Test Sample Model Fit")

> abline(linReg2,col="darkred")

> round(cor(pred_mean,y[-train])^2,3) [1] 0.885
```

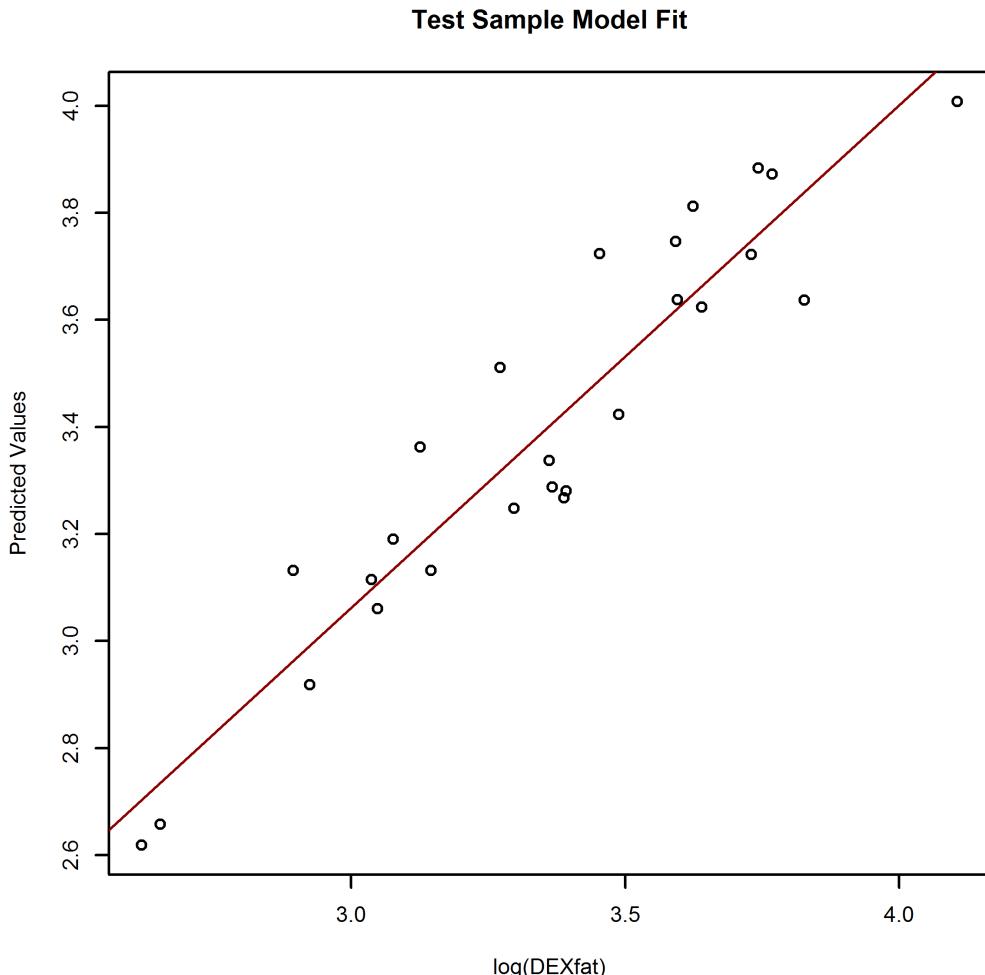


Figure 38.1: Quantile Regression Neural Network observed and predicted values (`pred_mean`) using `bodyfat`

It is often useful to visualize the results of a model. In Figure 38.2 we use `scatterplot3d` to plot the relationship between the `pred_low`,`pred_high` and the observed values in the test sample.

```
> scatterplot3d(pred_low,
+ y [-train],
+ pred_high,xlab="quantile = 0.05",
+ ylab="Test Sample",
+ zlab="quantile = 0.95")
```

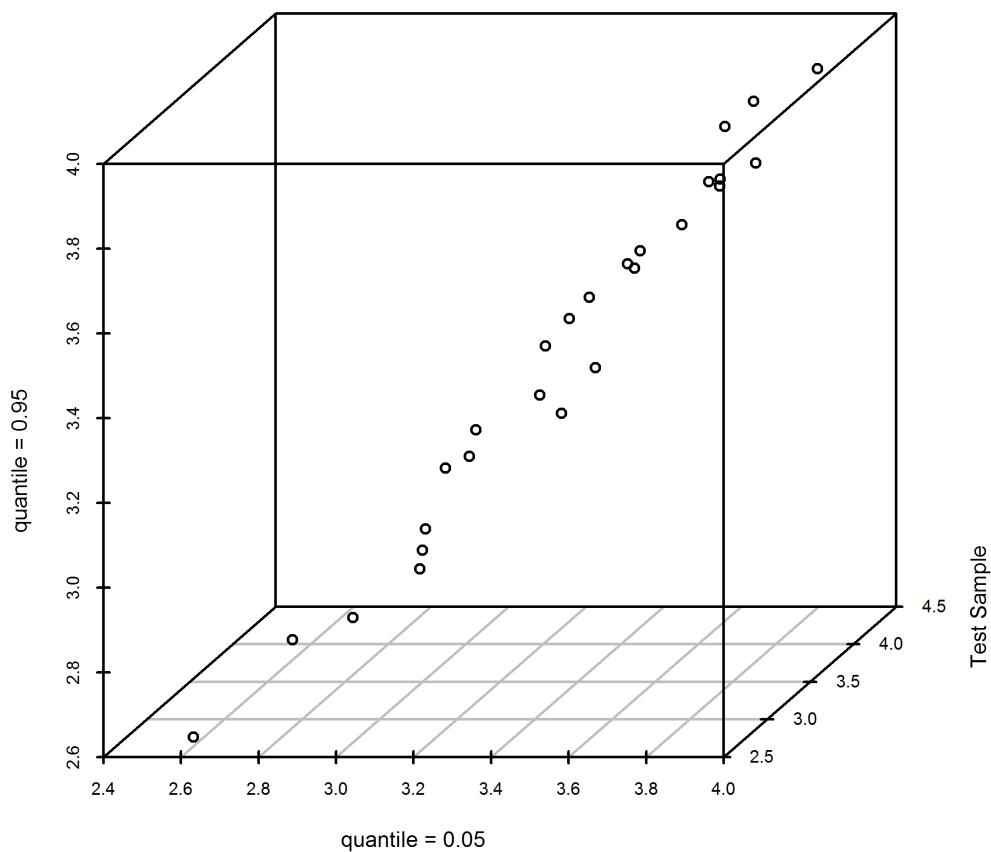


Figure 38.2: Quantile regression neural Network with `scatterplot3d` of 5th and 95th predicted quantiles

Notes

⁶²Tayfur, Gokmen. "Artificial neural networks for sheet sediment transport." *Hydrological Sciences Journal* 47.6 (2002): 879-892.

⁶³Kilinc, Mustafa. "Mechanics of soil erosion from overland flow generated by simulated rainfall." Colorado State University. *Hydrology Papers* (1973).

⁶⁴Mantri, Jibendu Kumar, P. Gahan, and Braja B. Nayak. "Artificial neural networks—An application to stock market volatility." *Soft-Computing in Capital Market: Research and Methods of Computational Finance for Measuring Risk of Financial Instruments* (2014): 179.

⁶⁵For further details see H.R. Champion et.al, Improved Predictions from a Severity Characterization of Trauma (ASCOT) over Trauma and Injury Severity Score (TRISS): Results of an Independent Evaluation. *Journal of Trauma: Injury, Infection and Critical Care*, 40 (1), 1996.

⁶⁶Hunter, Andrew, et al. "Application of neural networks and sensitivity analysis to improved prediction of trauma survival." *Computer methods and programs in biomedicine* 62.1 (2000): 11-19.

⁶⁷Lek, Sovan, et al. "Application of neural networks to modelling nonlinear relationships in ecology." *Ecological modelling* 90.1 (1996): 39-52.

⁶⁸See for example Jun, J. J., Longtin, A., and Maler, L. (2011) Precision measurement of electric organ discharge timing from freely moving weakly electric fish. *J. Neurophysiol* 107, pp. 1996-2007.

⁶⁹Kiar, Greg, et al. "Electrical localization of weakly electric fish using neural networks." *Journal of Physics: Conference Series*. Vol. 434. No. 1. IOP Publishing, 2013.

⁷⁰Wu, Naicheng, et al. "Modeling daily chlorophyll a dynamics in a German lowland river using artificial neural networks and multiple linear regression approaches." *Limnology* 15.1 (2014): 47-56.

⁷¹See Zhang, H. and Zhang, Z., 1999. Feedforward networks with monotone constraints. In: International Joint Conference on Neural Networks, vol. 3, p. 1820-1823. doi:10.1109/IJCNN.1999.832655

Part V

Random Forests

The Basic Idea

Suppose you have a sample of size N with M features. Random forests (RF) build multiple decision trees with each grown from a different set of training data. For each tree, K training samples are randomly selected with replacement from the original training data set.

In addition to constructing each tree using a different random sample (bootstrap) of the data, the RF algorithm differs from a traditional decision tree because at each decision node, the best splitting feature is determined from a randomly selected subspace of m features where m is much smaller than the total number of features M . Traditional decision trees split each node using the best split among all features.

Each tree in the forest is grown to the largest extent possible without pruning. To classify a new object, each tree in the forest gives a classification, which is interpreted as the tree ‘voting’ for that class. The final prediction is determined by majority votes among the classes decided by the forest of trees.

NOTE... ↗

Although each tree in a RF tends to be less accurate than a classical decision tree, combining multiple predictions into one aggregate prediction a more accurate forecast is often obtained. Part of the reason is that prediction of a single decision tree tend to be highly sensitive to noise in its training set. This is not the case for the average of many trees provided they are uncorrelated. For this reason the RF often decrease overall variance without increasing bias relative to a single decision tree.

Random Ferns

Random ferns are an ensemble of constrained trees originally used in image processing for identifying textured patches surrounding key-points of interest across images taken from different viewpoints. This machine learning method gained widespread attention in the image processing community because it is extremely fast, easy to implement and appeared to outperform random forests in image processing classification tasks provided the training set was sufficiently large⁷².

While a tree applies a different decision function at each node, a fern systematically applies the same decision function for each node of the same level. Each fern consists of a small set of binary tests and returns the probability that a object belongs to any one of the classes that have been learned during training. A naive Bayesian approach is used to combine the collection of probabilities. Ferns differ from decision trees in two ways:

- In decision trees the binary tests are organized hierarchically (hence the term tree) ferns are flat. Each fern consists of a small set of binary tests and returns the probability that a observation belongs to any one of the classes that have been learned during training.
- Ferns are more compact than trees. In fact, $2^{(N-1)}$ operations are needed to grow a tree of 2^N leaves, only N operations are needed to grow a fern of 2^N leaves.
- For decision trees the posterior distributions are computed additively in ferns they are multiplicative.

Practical Applications

Tropical Forest Carbon Mapping

Accurate and spatially-explicit maps of tropical forest carbon stocks are needed to implement carbon offset mechanisms such as REDD+ (Reduced Deforestation and Degradation Plus⁷³). Parametric statistics have traditionally dominated the discipline⁷⁴. Remote sensing technologies such as Light Detection and Ranging (LiDAR) have been used successfully to estimate spatial variation in carbon stocks⁷⁵. However, due to cost and technological limitations it is not yet feasible to use it to map the entire worlds tropical forests.

Mascaro et al⁷⁶ evaluated the performance of the Random Forest algorithm in up scaling airborne LiDAR based carbon estimates compared to the traditionally accepted stratification-based sampling approach over a 16-million hectare focal area of the Western Amazon.

Two separate Random Forest models were investigated, each using an identical set of 80,000 randomly selected input pixels. The first Random Forest model used the same set of input variables as the stratification approach. The second Random Forest used additional “position” parameters: x and y coordinates, combined with two diagonal coordinates, see Table 19.

The resultant maps of ecosystem carbon stock are shown in Figure 38.3. Notice that several regions exhibit pronounced differences when using the Random Forest with the additional four position’ parameters (image B).

The stratification method had a root mean square error (RMSE) of 33.2 Mg C ha⁻¹ and adjusted R² of 0.37 (predicted versus observed). The first Random Forest model had a RMSE =31.6 Mg C ha⁻¹and an adjusted R² of 0.43. The Random Forest model with the additional four position’ parameters had a RMSE 26.7 Mg Cha⁻¹ and an adjusted R² of 0.59. The researchers conclude there is an improvement when using Random Forest with position information which is consistent at all distances (see Figure 38.4).

Variable	Explanation	S	RF1	RF2
easting	UTM X coordinate	-	-	x
northing	UTM Y coordinate	-	-	x
diagx	X coordinate after 45 degree clockwise image rotation	-	-	x
diagy	Y coordinate after 45 degree clockwise image rotation	-	-	x
frac_soil	Percent cover of soil as determined by Landsat image processing with CLASlite	-	-	x
frac_pv	Percent cover of photosynthetic vegetation as determined by Landsat image processing with CLASlite	x	x	x
frac_npv	Percent cover of non-photosynthetic vegetation as determined by Landsat image processing with CLASlite	x	x	x
elevation	SRTM elevation above sea level	x	x	x
slope	SRTM slope	x	x	x
aspect	SRTM aspect	x	x	x
geoeco	Habitat class as determined by synthetic integration of national geological map, NatureServe and other sources.	x	x	x

Table 19: Input variables used Mascaro et al. S = Stratification-based sampling, RF = Random Forest.

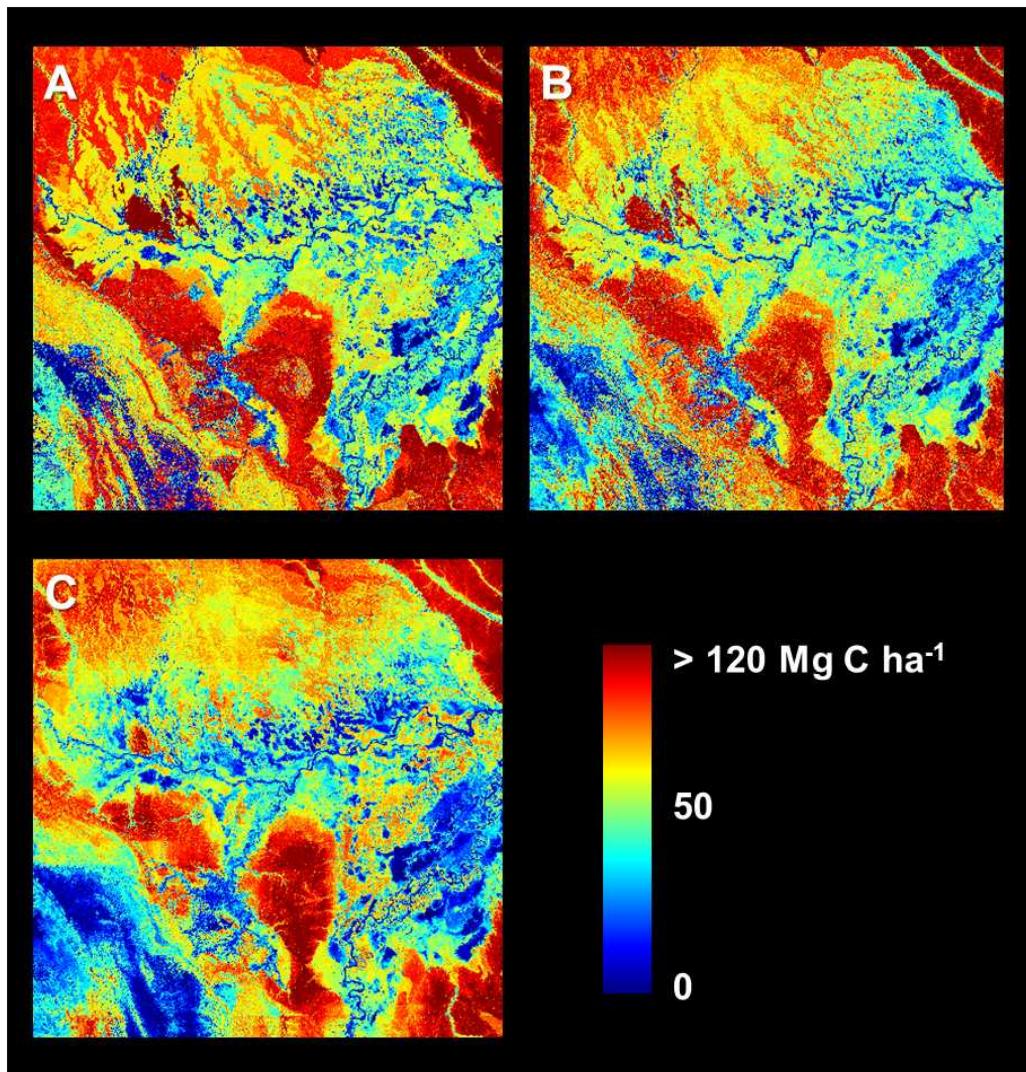


Figure 38.3: Mascaro et al's Predicted carbon stocks using three different methodologies. (a) Stratification and mapping of median carbon stocks in each class, (b) Random Forest without the inclusion of position information, (c) Random Forest using additional model inputs for position. Source of figure Mascaro et al.

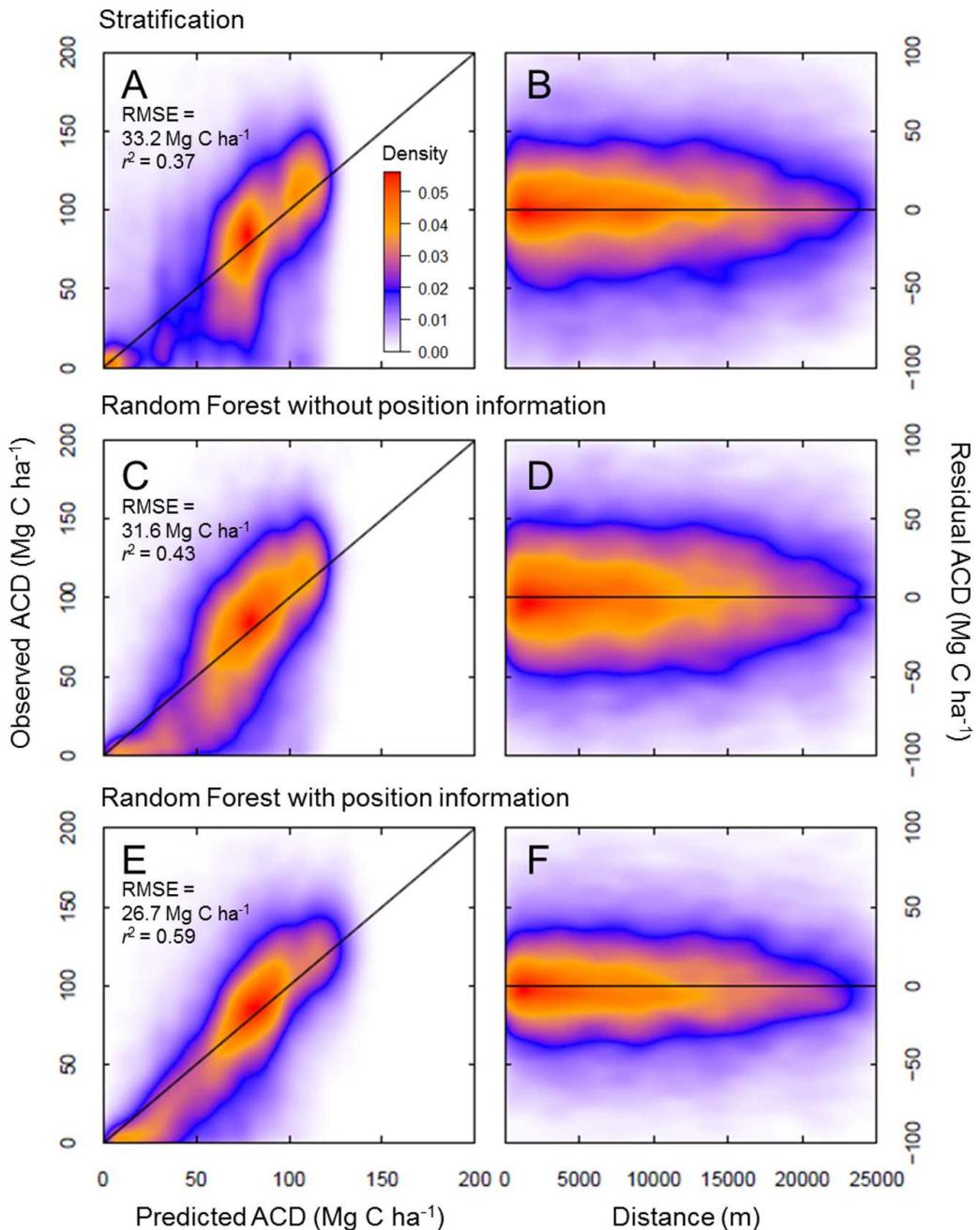


Figure 38.4: Mascaro et al's performance of three modeling techniques as assessed in 36 validation cells. Left panels highlight model performance against LiDAR-observed above ground carbon density from CAO aircraft data (Mg C ha^{-1}), while right panels highlight the model performance by increasing distance from CAO aircraft data. The color-scale reflects the two-dimensional density of observations, adjusted to one dimension using a square root transformation. Source of figure Mascaro et al.

Opioid Dependency & Sleep Disordered Breathing

Patients using chronic opioids are at elevated risk for potentially lethal disorders of breathing during sleep. Farney et al⁷⁷ investigate sleep disordered breathing in patients receiving therapy with buprenorphine/ naloxone.

A total of 70 patients admitted for therapy with buprenorphine/naloxone were recruited into the study. Indices were created for Apnoea/hypopnoea (AHI), obstructive apnoea (OAI), central apnoea (CAI) and hypopnoea (HI).

Each index was computed as the total of defined respiratory events divided by the total sleep time in hours scored simultaneously by two onlooking researchers. Oximetry data were analyzed to calculate mean SpO₂, lowest SpO₂ and time spent < 90% SpO₂ during sleep.

For each of these response metrics a Random Forest model was built using the following attributes - buprenorphine dose, snoring, tiredness, witnessed apnoeas, hypertension, body mass index, age, neck circumference, gender, use of benzodiazepines, antidepressants, antipsychotics and smoking history. The researchers find that the random forests showed little relationship between the explanatory variables and the response variables.

Waste-water Deterioration Modeling

Vitorino et al⁷⁸ build a Random Forest model to predict the condition of individual sewers and to determine the expected length of sewers in poor condition. The data consisted of two sources, a sewer data table and a inspection data table.

The sewer table contained information on the sewer identification code, zone, construction material, diameter, installation date and a selection of user defined covariates. The inspection data table contained information on the sewer identification code, date of last inspection and condition of sewer at date of last inspection.

The Random Forest was trained using all available data and limited to 50 trees. It was then used to predict the condition of individual sewer pipes. The researchers predict the distribution of sewer pipe in poor condition by type of material used for construction. The top three highest ranked materials were CIP (31.83%) followed by unknown material (27.94%) and RPM (23.89%).

Optical Coherence Tomography & Glaucoma

Glaucoma is the second most common cause of blindness. As glaucomatous visual field (VF) damage is irreversible, the early diagnosis of glaucoma is essential. Sugimoto et al⁷⁹ develop a random forest classifier to predict the presence of (VF) deterioration in glaucoma suspects using optical coherence tomography (OCT) data.

The study investigated 293 eyes of 179 live patients referred to the University of Tokyo Hospital for glaucoma between August 2010 and July 2012. The Random Forest algorithm with 10,000 trees was used to classify the presence or absence of glaucomatous VF damage using 237 different OCT measurements ; Age, gender, axial length and eight other right/left eye metrics were also included as training attributes.

The researchers report a receiver operating characteristic curve value of 0.9 for the random forest. This compared well to the value of 0.75 for an individual tree and between 0.77 and 0.86 for individual right/left eye metrics.

Obesity Risk Factors

Kanerva et al⁸⁰ investigate 4720 Finnish subjects who completed health questionnaires about leisure time physical activity, smoking status, and educational attainment. Weight and height were measured by experienced nurses. A random forest algorithm using the `randomForest` package in R and the collected indicator variables was used to predict overweight or obesity. The results were compared with a logistic regression model.

The researchers observe that the random forest and logistic regression had very similar classification power, for example the estimated error rates for the models were equal - 42% for men (RF) versus logistic regression 43% for men. The researchers tentatively conclude: “*Machine learning techniques may provide a solution to investigate the network between exposures and eventually develop decision rules for obesity risk estimation in clinical work.*”

Protein Interactions

Chen et al⁸¹ develop a novel random forest model to predict protein-protein interactions. They choose a traditional classification framework with two classes. Either a protein pair interacts with each other or they do not. Each protein pair is characterized by a very large attribute space, in total 4,293 unique attributes.

The training and test set each contained 8,917 samples, 4917 positive and 4,000 negative samples. Five-fold cross validation is used and the maximum size of a tree is 450 levels. The researchers report an overall sensitivity of 79.78% and specificity of 64.38%; this compares well to an alternative maximum likelihood technique which had a sensitivity of 74.03% and specificity of 37.53%.

Technique 39

Classification Random Forest

A classification random forest can be built using the package `randomForest` with the `randomForest` function:

```
randomForest(z ~., data, ntree, importance=TRUE,  
proximity=TRUE, ...)
```

Key parameters include `ntree` which controls the number of trees to grow in each iteration, `importance` a logical variable if set to `TRUE` assesses the importance of predictors, `proximity` another logical variable which if set to `TRUE` calculates a proximity measure among the rows, `z` the data-frame of classes; `data` the data set of attributes with which you wish to build the forest.

Step 1: Load Required Packages

We build the classification random forest using the `Vehicle` data frame (see page 23) contained in the `mlbench` package:

```
> library ("randomForest")  
> library(mlbench)  
> data(Vehicle)
```

Step 2: Prepare Data & Tweak Parameters

The variable `num.trees` is used to contain the number of trees grown at each iteration. We set it to 1000. A total 500 of the 846 observations in `Vehicle` are used to create a randomly selected training sample.

```
> set.seed(107)
```

```
> N=nrow(Vehicle)
> num.trees=1000
> train <- sample(1:N, 500, FALSE)
```

The data frame `attributes` is used to hold the complete set of attributes contained in `Vehicle`.

```
> attributes<-Vehicle[train,]
> attributes$Class <- NULL
```

Step 3: Estimate the Random Forest

We estimate the random forest using the training sample and the function `randomForest`. The parameter `ntree = 1000` and we set both `importance` and `proximity` equal to `TRUE`.

```
> fit<- randomForest(Class ~., data = Vehicle[train
   ,],ntree=num.trees,importance=TRUE,proximity=TRUE)
```

The print function returns details of the random forest:

```
> print(fit)

Call:
randomForest(formula = Class ~ ., data = Vehicle[
  train, ], ntree = num.trees,           importance =
TRUE, proximity = TRUE)
  Type of random forest: classification
  Number of trees: 1000
  No. of variables tried at each split: 4

  OOB estimate of  error rate: 27.2%
Confusion matrix:
      bus  opel  saab  van class.error
bus    121     1     1     1  0.02419355
opel     1    60    60     6  0.52755906
saab     5    49    66     9  0.48837209
van      1     0     2   117  0.02500000
```

Important information include the formula used to fit the model, the number of trees grown at each iteration, the number of variables tried at each split, the confusion matrix, class errors and out of the bag error estimate.

The random forest appears to identify both **van** and **bus** with a low error (less than 3%) but **saab** (49%) appears to be essentially a random guess whilst **opel** (53%) is marginally better than random.

Step 4: Assess Model

A visualization of the error rate by iteration for each **bus**, **opel**, **saab**, **van** and overall out of the bag error is obtained using **plot**, see Figure 39.1.

```
> plot(fit)
```

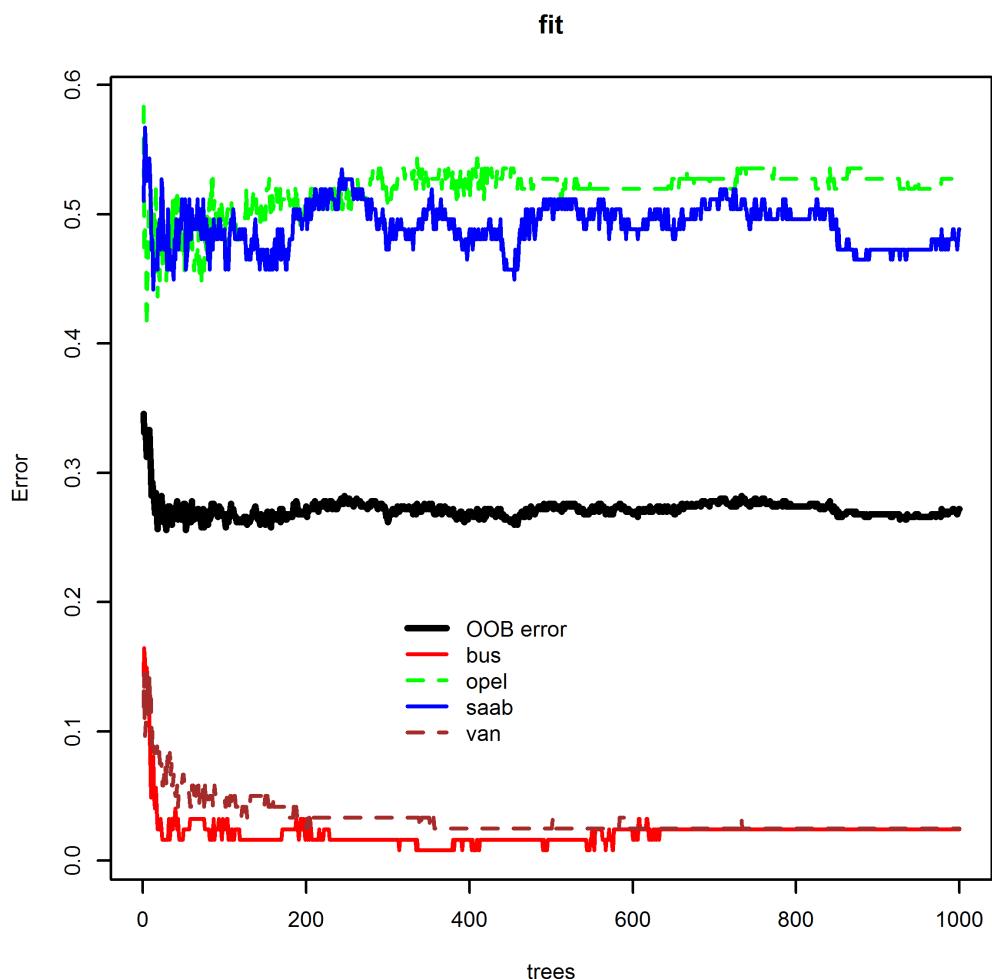


Figure 39.1: Error estimate across iterations for each class and overall using **Vehicle**

The misclassification error on **van** and **bus** settle as well as the out of bag error settle down to a stable range within 200 trees. For **opel** stability occurs around 400 trees, whilst for **saab** around 800 trees.

The **rfcv** function is used to perform a 10-fold cross validation (**cv.fold=10**) with variable importance re-assessed at each step (**reductionrecursive=TRUE**). The result is stored in **cv**.

```
> cv <- rfcv(trainx = attributes, trainy = Vehicle
   $Class[train], ,ntree=num.trees, cv.fold=10,
   recursive=TRUE)
```

☛ PRACTITIONER TIP ☛

This function **rfcv** calculates cross-validated prediction performance with a sequentially reduced number of attributes ranked by variable importance using a nested cross-validation procedure. The minimum value of the cross validation risk can be used to select the optimum number of attributes (known also as predictors) in a random forest model.

A plot of cross validated error versus number of predictors can be calculated as follows:

```
> with(cv, plot(n.var, error.cv,
  log="x", type="o", lwd=2,
  xlab="number of predictors",
  ylab="cross-validated error (%)"))
```

Figure 39.2 shows the resultant plot. It indicates a steady decline in the cross validated error for the first nine attributes and then it levels off.

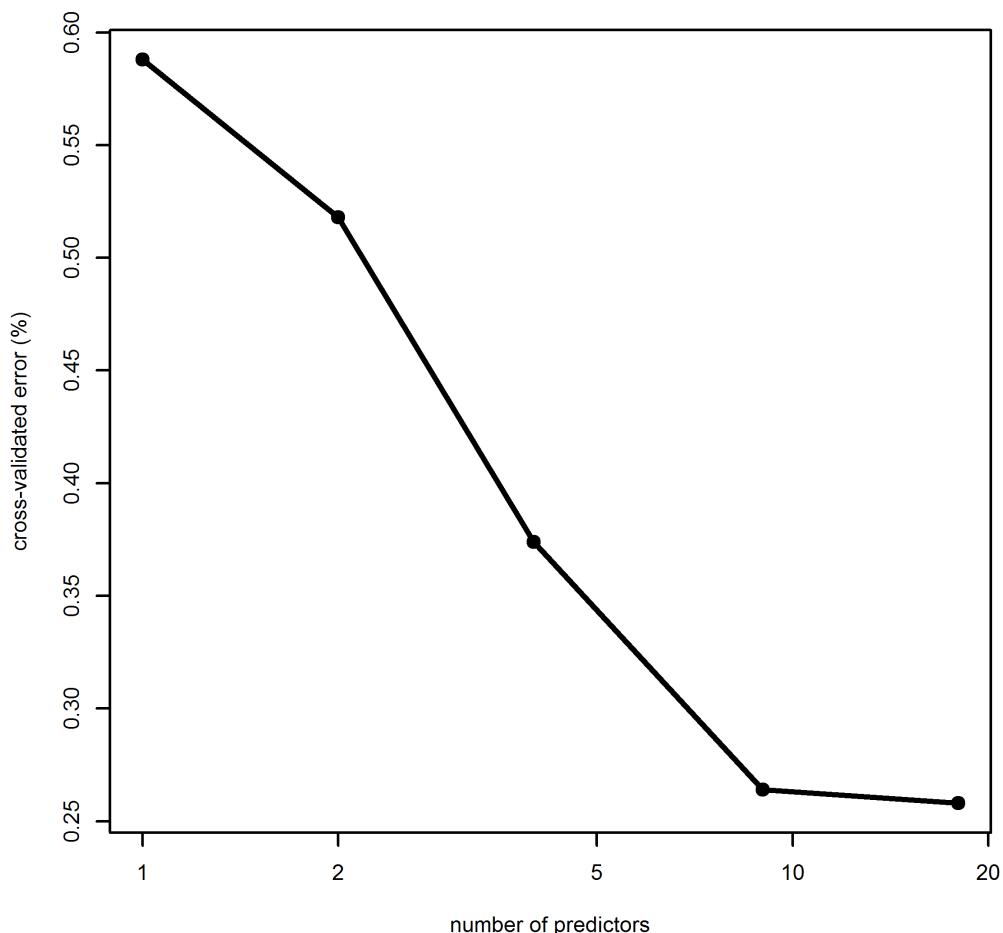


Figure 39.2: Cross-validated prediction performance by number of attributes for **Vehicle**

Variable importance, shown in Figure 39.3, is obtained using the function `varImpPlot`:

```
> varImpPlot(fit)
```

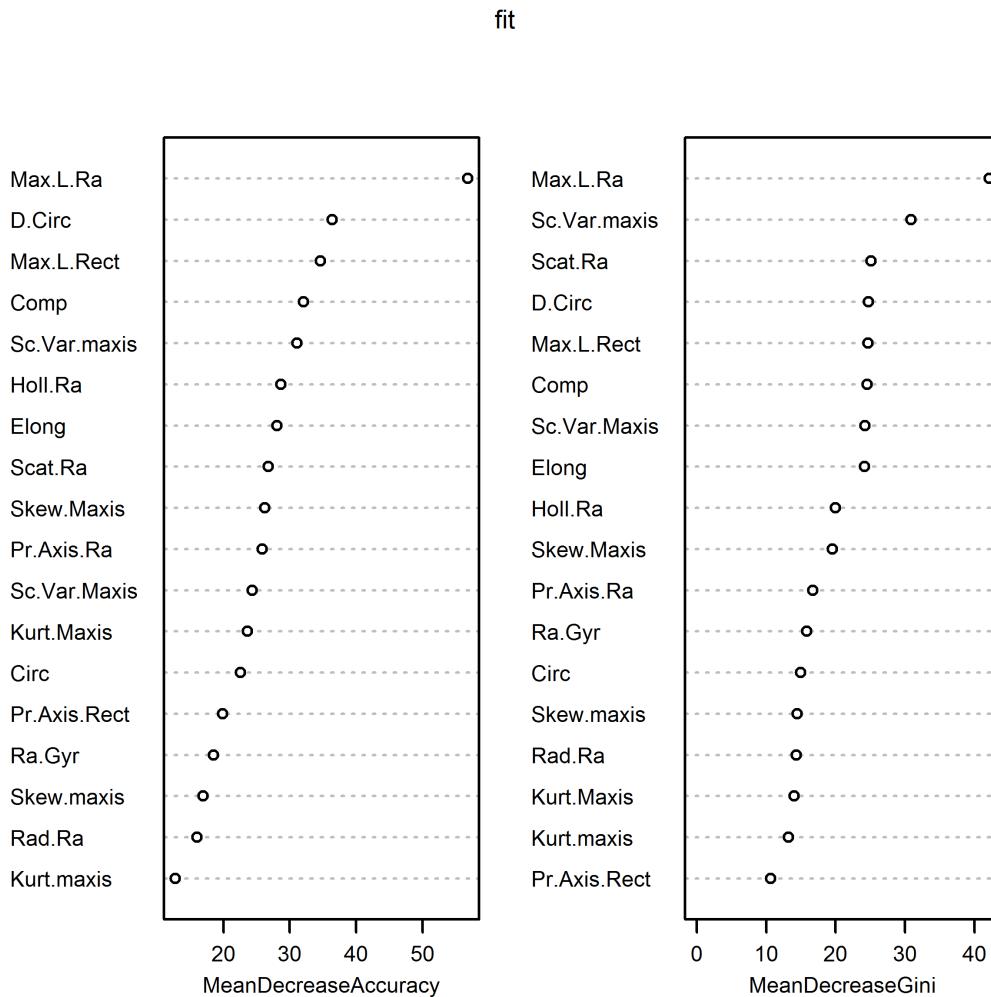


Figure 39.3: Random Forrest variable importance plot for `Vehicle`

`Max.L.Ra` and `D.Circ` seem to be important whether measured by average decrease in the accuracy or reduction in node impurity (gini).

Partial dependence plots provide a way to visualize the marginal relationship between the response variable and the covariates / attributes. Let's use this idea combined with the importance function to investigate the top four attributes.

```
> imp <- importance(fit)

> impvar <- rownames(imp)[order(imp[, 1], decreasing = TRUE)]
```

```
> op <- par(mfrow=c(2, 2))

> for (i in seq_along(1:4)) {
  partialPlot(fit, Vehicle[train,], impvar[i], xlab=
    impvar[i],
  main=paste("Partial Dependence on", impvar[i]))
}
> par(op)
```

Figure 39.4 shows the partial dependence plots for Max.L.Ra, D.Circ,Pr.Axis.Ra and Max.L.Ra.

☛ PRACTITIONER TIP ☛

The **importance** function takes the **type** argument. Set **type** = 1 if you only want to see importance by average decrease in accuracy; and set **type** = 2 to see only importance by average decrease in node impurity measured by the gini coefficient.

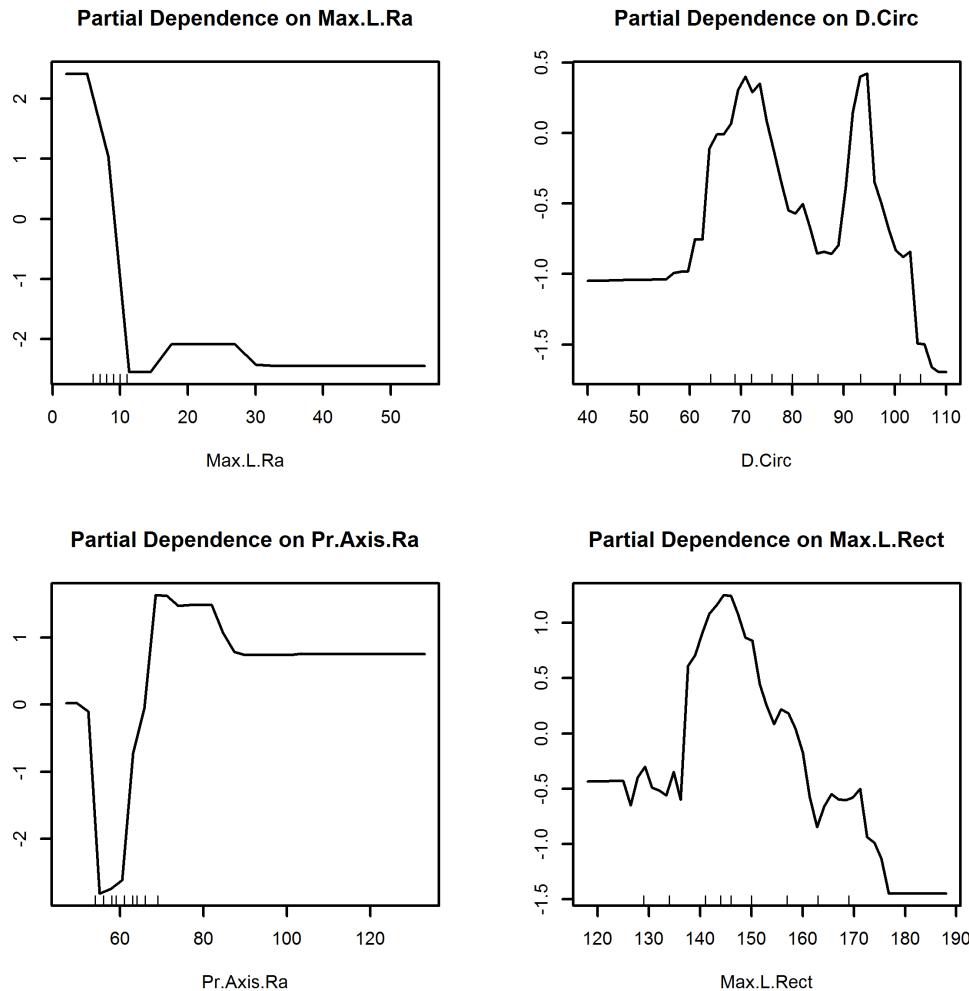


Figure 39.4: Random forest partial dependence plot for Max.L.Ra, D.Circ, Pr.Axis.Ra and Max.L.Rect

It is often helpful to look at the margin. This is measured as the proportion of votes for the correct class minus the maximum proportion of votes for the other classes. A positive margin implies correct classification and a negative margin incorrect classification.

```
> plot(margin(fit))
```

The margin, for `fit` is show in Figure 39.5.

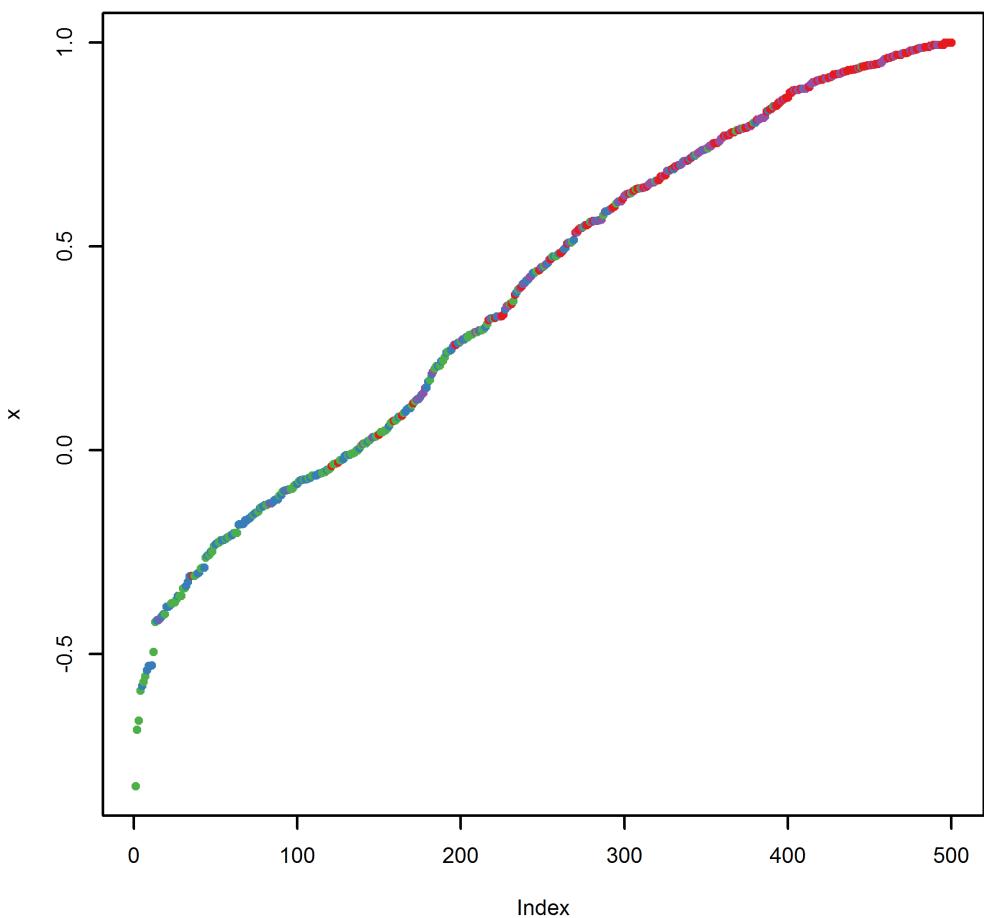


Figure 39.5: Random forest margin plot for Vehicle

Step 5: Make Predictions

Next we use the fitted model on the test sample.

```
>fit<-randomForest(Class~.,data=Vehicle[-train,],  
+ntree=num.trees,importance=TRUE,proximity=TRUE)  
  
> print(fit)  
  
Call:
```

```
randomForest(formula = Class ~ ., data = Vehicle[-  
train, ], ntree = num.trees, importance =  
TRUE, proximity = TRUE)  
Type of random forest: classification  
Number of trees: 1000  
No. of variables tried at each split: 4  
  
OOB estimate of error rate: 26.59%  
Confusion matrix:  
bus opel saab van class.error  
bus 92 0 1 1 0.02127660  
opel 4 35 37 9 0.58823529  
saab 5 25 50 8 0.43181818  
van 1 1 0 77 0.02531646
```

We notice a sharp deterioration in the classification error for **saab** over the validation set, although **opel** has improved dramatically. The misclassification rates for **van** and **bus** are in line with those observed in the training sample. The overall out of the bag estimate is around 27%

An important aspect of building random forest models is tuning the model. We can attempt to do this by finding the optimal value of **mtry** and then fitting that model on the test data. The parameter **mtry** controls the number of variables randomly sampled as candidates at each split. The **tuneRF** function returns the optimal value of **mtry**. The smallest out of the bag error occurs at **mtry = 3**, this is confirmed from Figure 39.6.

```
>best_mytry <- tuneRF(attributes, Vehicle$Class[  
  train], ntreeTry=num.trees, stepFactor=1.5, improve  
  =0.01, trace=TRUE, plot=TRUE, dobest=FALSE)  
  
mtry = 4 OOB error = 27%  
Searching left ...  
mtry = 3 OOB error = 26.6%  
0.01481481 0.01  
mtry = 2 OOB error = 26.8%  
-0.007518797 0.01  
Searching right ...  
mtry = 6 OOB error = 26.6%
```

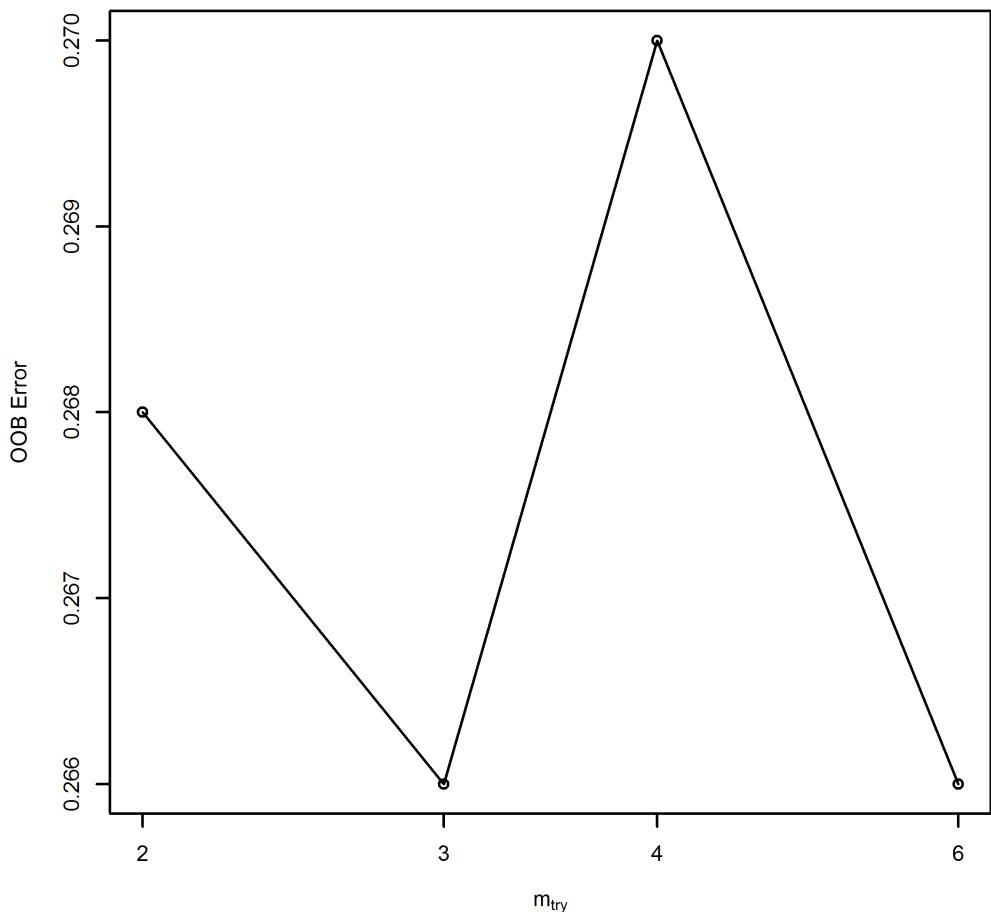


Figure 39.6: Random forest OOB error by mytry for Vehicle

Finally, we fit a random forest model on the test data using `mtry=3`.

```
>fit.best<-randomForest(Class~, data = Vehicle[-  
  train,], ntree=num.trees, importance=TRUE, proximity=  
  TRUE ,mtry=3)  
  
> print(fit.best)  
  
Call:  
  randomForest(formula = Class ~ ., data = Vehicle[-  
    train, ], ntree = num.trees,           importance =  
    TRUE, proximity = TRUE, mtry = 3)
```

```
Type of random forest: classification
Number of trees: 1000
No. of variables tried at each split: 3

OOB estimate of error rate: 25.14%
Confusion matrix:
      bus  opel  saab  van  class.error
bus     93     0     0     1  0.01063830
opel     3    39    36     7  0.54117647
saab     4    26    50     8  0.43181818
van      1     1     0    77  0.02531646
```

The model `fit.best` has a slightly lower out of the bag error at 25.14%. However, relative to `fit`, the accuracy of `opel` has declined somewhat.

Technique 40

Conditional Inference Classification Random Forest

A conditional inference classification random forest can be built using the package `party` with the `cforest` function:

```
cforest(z ~ ., data, , controls, ...)
```

Key parameters include `controls` which controls parameters such as the number of trees grown in each iteration, `z` the data-frame of classes; `data` the data set of attributes with which you wish to build the forest.

Step 1: Load Required Packages

We build the classification random forest using the `Vehicle` data frame (see page 23) contained in the `mlbench` package. We also load the `caret` package as it provides handy performance metrics for random forests.

```
> library ("party")
> library(caret)
> library(mlbench)
> data(Vehicle)
```

☛ PRACTITIONER TIP ☛

If you have had your R session open for a lengthy period of time you may have objects hogging memory. To see what objects are being held in memory type:

```
ls()
```

To remove all objects from memory use:

```
rm(list=ls())
```

To remove a specific object from memory use:

```
rm(object_name)
```

Step 2: Prepare Data & Tweak Parameters

A total 500 of the 846 observations are in `Vehicle` are used to create a randomly selected training sample.

```
> set.seed(107)
> N=nrow(Vehicle)
> train <- sample(1:N, 500, FALSE)
```

Step 3: Estimate and Assess the Random Forest

We estimate the random forest using the training sample and the function `cforest`. The number of trees is set to 1000 (`ntree = 1000`) with `mtry` set to 5.

```
> fit<-cforest(Class ~., data = Vehicle[train,],
  controls =cforest_unbiased(ntree = 1000,mtry=5))
```

The accuracy of the model and the kappa statistic can be retrieved using the `caret` package:

```
> caret:::cforestStats(fit)
Accuracy      Kappa
0.86600    0.82146
```

Let's do a little memory management and remove fit:

```
> rm(fit)
```

Let's re-estimate the random forest using the default settings in cforest.

```
> fit<-cforest(Class ~., data = Vehicle[train,])
> caret:::cforestStats(fit)
  Accuracy      Kappa
0.8700000 0.8267831
```

Since both the accuracy and kappa are slightly higher we will use this as the fitted model.

☞ PRACTITIONER TIP ☜

Remember that all simulation based models including the random forest are subject to random variation. This variation can be important when investigating variable importance. It is advisable before interpreting a specific importance ranking to check whether the same ranking is achieved with a different model run (random seed) or a different number of trees.

Variable importance is calculated using the function `varimp`. First we estimate variable importance using the default approach which calculates the mean decrease in accuracy using the method outlined in Hapfelmeier et al.⁸²

```
> ord1<-varimp(fit)
> imp1<-ord1[order(-ord1[1:18])]

> round(imp1[1:3],3)
  Elong  Max.L.Ra  Scat.Ra
  0.094    0.069    0.057

> rm(ord1, imp1)
```

Next we try the unconditional approach using the ‘mean decrease in accuracy’ outline by Breiman⁸³. Interestingly, the attributes `Elong`, `Max.L.Ra` and `Scat.Ra` are rank ordered exactly the same by both methods and they have very similar scores.

```
> ord2<-varimp(fit, pre1.0_0 = TRUE)
> imp2<-ord2[order(-ord2[1:18])]

> round(imp2[1:3],3)
  Elong  Max.L.Ra  Scat.Ra
```

```
0.100      0.083      0.057  
> rm(ord2, imp2)
```

☛ PRACTITIONER TIP ☛

In addition to the two unconditional measures of variable importance already discussed a conditional version is also available. It is conditional in the sense that it adjusts for correlations between predictor variables. For the random forest estimated by `fit` conditional variable importance can be obtained by:

```
> varimp(fit, conditional = TRUE)
```

Step 5: Make Predictions

First we estimate accuracy and kappa using the fitted model and the test sample.

```
> fit_test<-cforest(Class ~., data = Vehicle[-train ,])  
> caret:::cforestStats(fit_test)  
Accuracy      Kappa  
0.8179191  0.7567866
```

Both accuracy and kappa values are close to the values estimated on the validation sample. This is encouraging in terms of model fit.

The `predict` function calculates predicted values over the testing sample. These values, along with the original observations, are used to construct the test sample confusion matrix and overall misclassification error. We observe an error rate of 28.6%.

```
> pred<-predict(fit,newdata=Vehicle[-train ,],type =  
"response")  
  
> table(Vehicle$Class[-train],pred,dnn=c( "Observed_  
Class","Predicted_Class" ))  
          Predicted Class  
Observed Class bus opel saab van  
    bus     91     0     1     2  
    opel     8    30    33   14
```

saab	6	23	51	8
van	2	0	2	75

```
> error_rate = (1 - sum(pred == Vehicle$Class[-train])) /  
346)  
  
> round(error_rate, 3)  
[1] 0.286
```

Technique 41

Classification Random Ferns

A classification random ferns model can be built using the package `rFerns` with the `rFerns` function:

```
rFerns(z ~., data, ferns, ...)
```

Key parameters include `ferns` which controls the number of ferns grown, at each iteration, `z` the data-frame of classes; `data` the data set of attributes with which you wish to build the forest.

Step 1: Load Required Packages

We construct a classification random ferns model using the `Vehicle` data frame (see page 23) contained in the `mlbench` package.

```
> library("rFerns")
> library(mlbench)
> data(Vehicle)
```

Step 2: Prepare Data & Tweak Parameters

A total 500 of the 846 observations in `Vehicle` are used to create a randomly selected training sample.

```
> set.seed(107)
> N=nrow(Vehicle)
> train <- sample(1:N, 500, FALSE)
```

Step 3: Estimate and Assess the Model

☛ PRACTITIONER TIP ☛

To print the out of bag error as the model iterates add the parameter `reportErrorEvery` to `rFerns`. For example, `reportErrorEvery = 50` returns the error every 50th iteration. Your estimated model would look something like this:

```
> fit<- rFerns(Class ~.,
  data = Vehicle[train,],
  ferns=300,
  reportErrorEvery=50)

Done fern 50/300; current OOB error
  0.39200
Done fern 100/300; current OOB error
  0.37800
Done fern 150/300; current OOB error
  0.35000
Done fern 200/300; current OOB error
  0.36000
Done fern 250/300; current OOB error
  0.35400
Done fern 300/300; current OOB error
  0.34800
```

The number of ferns is set to 5000 with the parameter `saveErrorPropagation=TRUE` to save the out of bag errors at each iteration.

```
> fit<- rFerns(Class ~., data = Vehicle[train,],
  ferns=5000, saveErrorPropagation=TRUE)
```

The details of the fitted model can be viewed using the `print` method. The method returns the number of ferns in the forest, the fern depth, the out of bag error estimate and the confusion matrix.

```
> print(fit)
```

```
Forest of 5000 ferns of a depth 5.

OOB error 33.20%; OOB confusion matrix:
  True
Predicted bus opel saab van
  bus   109     2     7     0
  opel    7    65    44     0
  saab    0    32    40     0
  van     8    28    38   120
```

It is often useful to view the out of bag error by fern size, this can be achieved using the `plot` method and `modelname$oobErr`.

```
> plot(fit$oobErr, xlab="Number of Ferns",
       ylab="OOB Error",
       type="l")
```

Figure 41.1 shows the resultant chart. Looking closely at the figure it appears the error is minimized at approximately 30% somewhere between 2000 and 3000 ferns. To get the exact number use the `which.min` method.

```
> which.min(fit$oobErr)
[1] 2487
> fit$oobErr[which.min(fit$oobErr)]
[1] 0.324
```

So the error reaches a minimum at 32.4% for 2487 ferns.

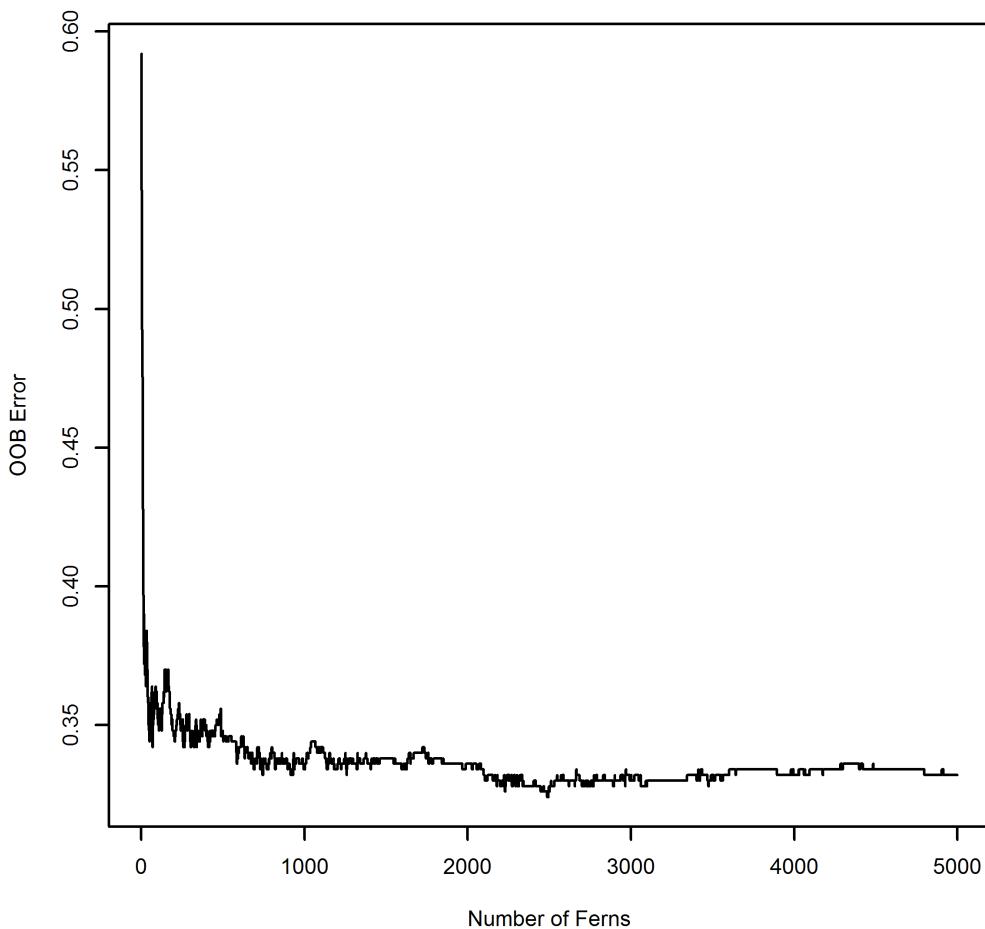


Figure 41.1: Classification Random Ferns out of bag error estimate by iteration for Vehicle

☛ PRACTITIONER TIP ☛

To determine the time taken to run the model you would use `$timeTaken` as follows:

```
> fit$timeTaken  
Time difference of 0.314466 secs
```

Other useful components of the `rFerns` object are given in Table 20.

Parameter	Description
\$model	The estimated model
\$oobErr	Out of bag error
\$importance	Importance scores
\$oobScores	class scores for each object
\$oobPreds	Class predictions for each object
\$timeTaken	Time used to train the model

Table 20: Some key components a `rFerns` object

Let's compare re-estimate with the number of ferns equal to 2487.

```
> fit<- rFerns(Class ~., data = Vehicle[train,], ferns=2487, importance=TRUE, saveForest=TRUE)
```

```
> print(fit)
```

```
Forest of 2487 ferns of a depth 5.
```

```
OOB error 33.20%; OOB confusion matrix:
```

```
    True
```

Predicted	bus	opel	saab	van
bus	107	1	6	0
opel	8	65	43	0
saab	0	33	42	0
van	9	28	38	120

Alas we ended up with the same out of bag error as our original model! Nevertheless we will keep this version of the model.

The influence of variables can be assessed using `$importance`. The six most important variables and their standard deviations are as follows:

```
> imp<-fit$importance[order(-fit$importance[1]),]
```

```
> round(head(imp),3)
```

	MeanScoreLoss	SdScoreLoss
Max.L.Ra	0.353	0.008
Elong	0.219	0.006
Sc.Var.Maxis	0.205	0.005
Sc.Var.maxis	0.201	0.005
Pr.Axis.Rect	0.196	0.005

D.Circ	0.196	0.004
--------	-------	-------

Step 5: Make Predictions

Predictions using the test sample can be obtained using the `predict` method. The results are stored in `predClass` and combined using the `table` method to create the confusion matrix.

```
> predClass<-predict(fit,Vehicle[-train,])  
  
> table( predClass ,Vehicle$Class[-train] , dnn =c(  
" Predicted Class " , " Observed Class "))  
                                Observed Class  
Predicted Class   bus  opel  saab  van  
    bus      85     5     4     0  
    opel      4    38    36     0  
    saab      0    13    28     0  
    van       5    29    20    79
```

Finally, we calculate the misclassification error rate. At 33.5% it is fairly close to the error estimate for the validation sample.

```
> error_rate = (1- sum( Vehicle$Class[-train] ==  
predClass )/ 346)  
> round( error_rate ,3)  
[1] 0.335
```

Technique 42

Binary Response Random Forest

On many occasions the response variable is binary. In this chapter we show how to build a random forest model in this situation. A binary response random forest can be built using the package `randomForest` with the `randomForest` function:

```
randomForest(z ~., data, ntree, importance=TRUE,  
proximity=TRUE, ...)
```

Key parameters include `ntree` which controls the number of trees to grow in each iteration, `importance` a logical variable if set to `TRUE` assesses the importance of predictors, `proximity` another logical variable which if set to `TRUE` calculates a proximity measure among the rows, `z` the binary response variable; `data` the data set of attributes with which you wish to build the forest.

Step 1: Load Required Packages

We build the binary response random forest using the `Sonar` data frame (see page 482) contained in the `mlbench` package. The package `ROCR` is also loaded. We will use this package during the final stage of our analysis.

```
> library("randomForest")  
> library(mlbench)  
> data(Sonar)  
> require(ROCR)
```

Step 2: Prepare Data & Tweak Parameters

The variable `num.trees` is used to contain the number of trees grown at each iteration. We set it to 1000. A total 157 out of the 208 observations in `Sonar` are used to create a randomly selected training sample.

```
> set.seed(107)
> N=nrow(Sonar)
> num.trees=1000
> train <- sample(1:N, 157, FALSE)
```

The data frame `attributes` holds the complete set of attributes contained in `Sonar`.

```
> attributes<-Sonar[train,]
> attributes$Class <- NULL
```

Step 3: Estimate the Random Forest

We estimate the random forest using the training sample and the function `randomForest`. The parameter `ntree = 1000` and we set `importance` equal to `TRUE`.

```
> fit<- randomForest(Class~, data = Sonar[train,],
  ntree=num.trees, importance=TRUE)
```

The `print` function returns details of the random forest:

```
> print(fit)

Call:
randomForest(formula = Class ~ ., data = Sonar[
  train, ], ntree = num.trees,           importance =
  TRUE)
              Type of random forest: classification
                      Number of trees: 1000
No. of variables tried at each split: 7

          OOB estimate of error rate: 18.47%
Confusion matrix:
      M   R class.error
M 70 12    0.1463415
R 17 58    0.2266667
```

The classification error on M is less than 20%, whilst for R is around 23%. The out of the bag estimate of error is also less than 20%.

Step 4: Assess Model

A visualization of the error rate by iteration for each M and R with the overall out of the bag error is obtained using `plot`, see Figure 42.1.

```
> plot(fit)
```

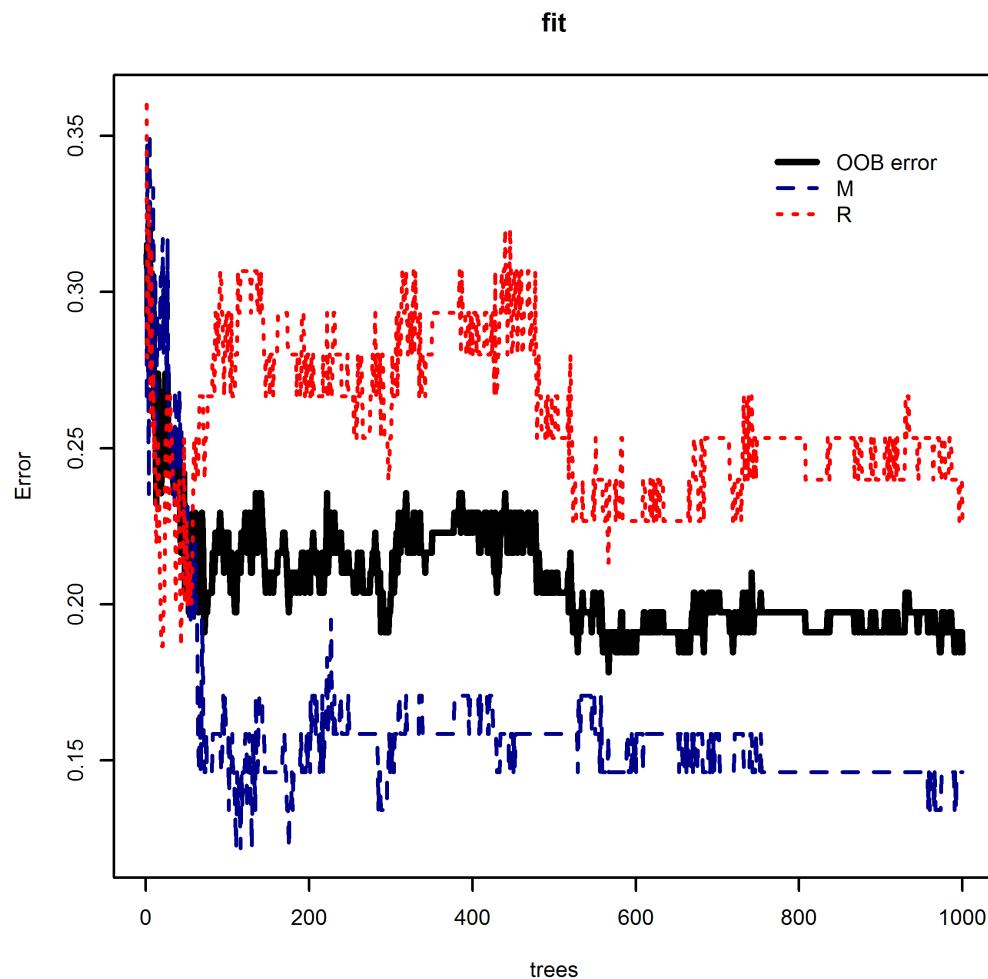


Figure 42.1: Random Forest Error estimate across iterations for M, R and overall using Sonar

The misclassification errors settle down to a stable range by 800 trees.

Next the variable importance scores are calculated and the partial dependence plot, see Figure 42.2, is plotted.

```
> imp <- importance(fit)
> impvar <- rownames(imp)[order(imp[, 1], decreasing =TRUE)]
> op <- par(mfrow=c(2, 2))

> for (i in seq_along(1:4)) {
  partialPlot(fit, Sonar[train,], impvar[i], xlab=impvar[i],
  main=paste("Partial Dependence on", impvar[i]))
}

> par(op)
```

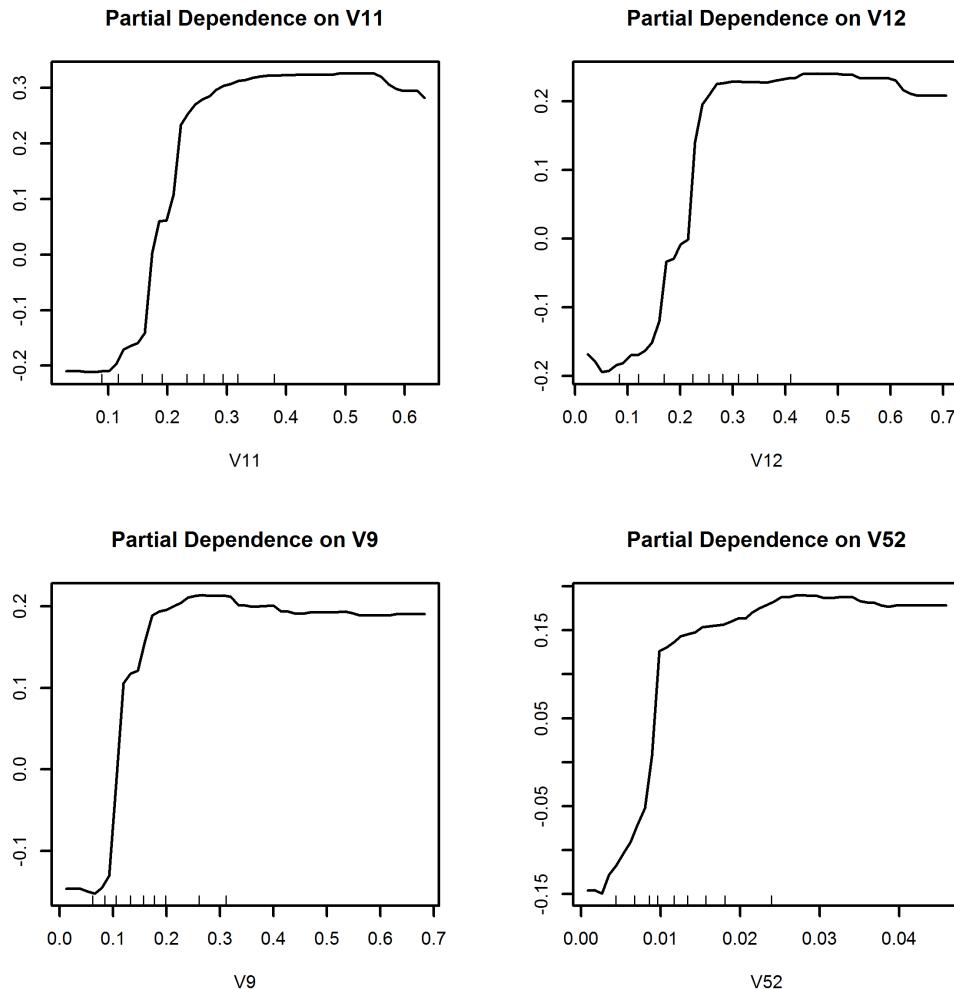


Figure 42.2: Binary random forest partial dependence plot using Sonar

Figure 42.2 seems to indicate that V11, V12, V9 and V52 have similar shapes, in terms of partial dependence. Looking closely at the diagram you will notice that all except V52 appear to be responding on a similar scale. Given the similar pattern we calculate a number of combinations of these three variables and add them to our group of attributes.

```
> temp<-cbind(Sonar$V11,Sonar$V12,Sonar$V9)

> Vmax<- apply(temp, 1, function(x) max(x))
> Vmin<- apply(temp, 1, function(x) min(x))
> Vmean<- apply(temp, 1, function(x) mean(x))
> Vmed<- apply(temp, 1, function(x) median(x))
```

```
> attributes<-cbind(attributes ,Vmax[train] ,Vmin[  
  train] ,Vmean[train] ,Vmed[train])  
> Sonar<-cbind(Sonar ,Vmax ,Vmin ,Vmean ,Vmed)
```

Now our list of attributes in `Sonar` also includes the maximum, minimum, mean and median of V11, V12, V9. Let's refit the model and calculate variable importance of all attributes.

```
> fit<- randomForest(Class ~., data = Sonar[train,],  
  ntree=num.trees, importance=TRUE)  
> varImpPlot(fit)
```

Figure 42.3 shows variable importance using two measures - the mean decrease in accuracy and the mean decrease in the gini score. Notice the consistency in the identification of the top eight variables between these two methods. It seems the most influential variables are `Vmax`, `Vmin`, `Vmean`, `Vmed`, `V11`, `V12`, `V9` and `V52`.

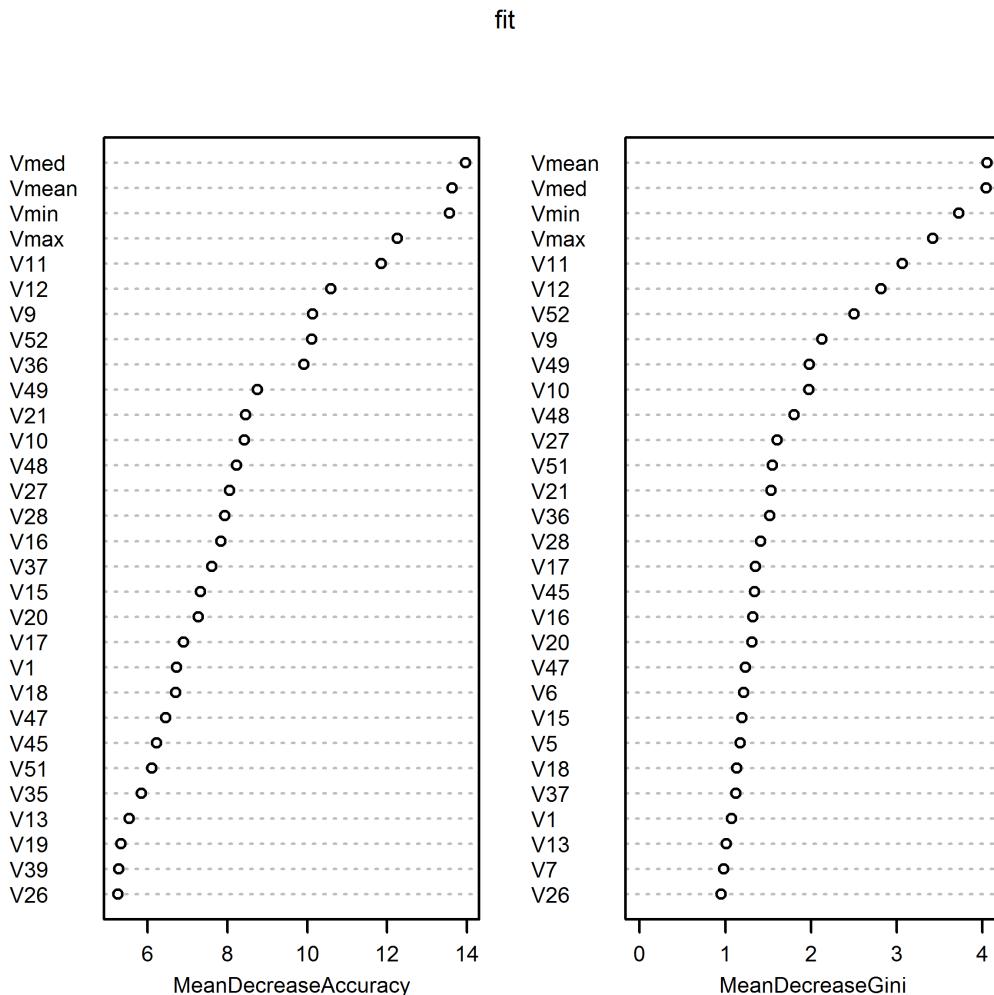


Figure 42.3: Binary Random Forest variable Importance plots for Sonar

A ten-fold cross validation is performed using the function `rfcv` followed with a plot, shown in Figure 42.4, of the cross validated error by number of predictors (attributes):

```
> cv <- rfcv(trainx = attributes, ,ntree=num.trees
   ,trainy = Sonar$Class[train], cv.fold=10,recursive
   =TRUE)
> with(cv, plot(n.var, error.cv, log="x", type="o",
   lwd=2,xlab="number of predictors",ylab="cross-
   validated error (%)))
```

The error falls sharply from 2 to around 9 predictors and continues to decline leveling out by 50 predictors.

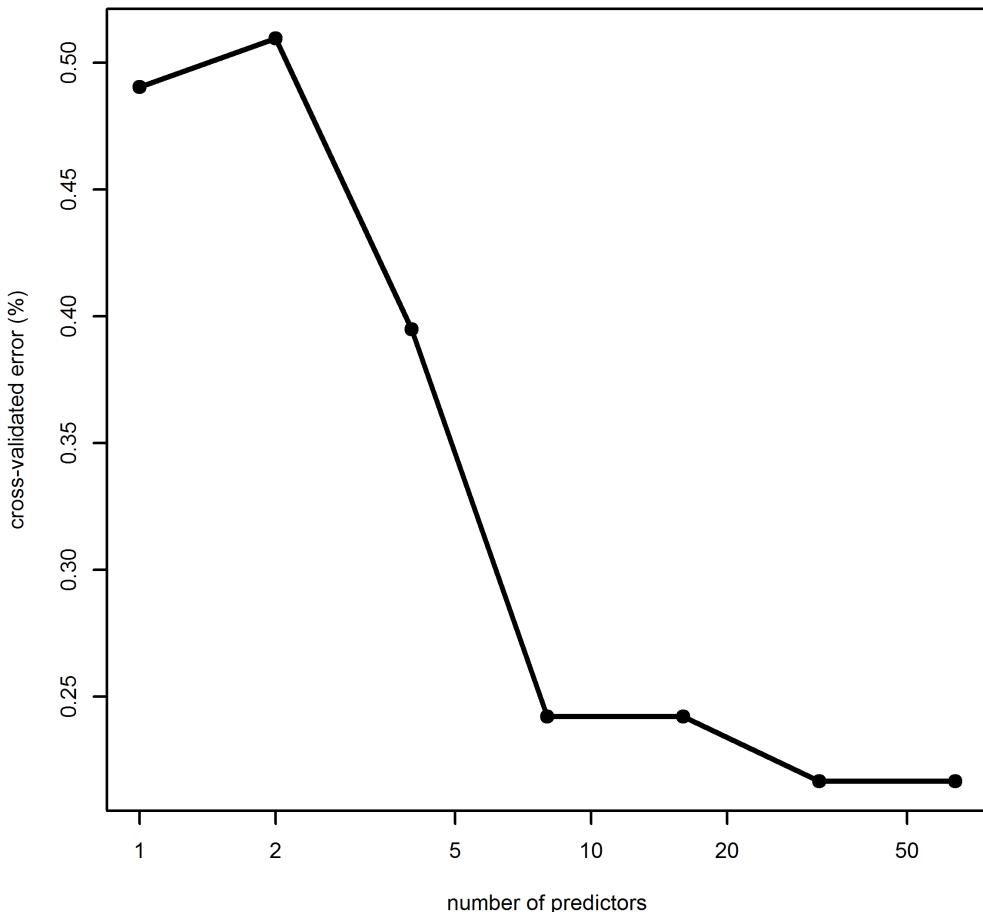


Figure 42.4: Binary random forest cross validation error by number of predictors using `Sonar`

Step 5: Make Predictions

It is always worth experimenting with the tuning parameter `mytry`. It controls the number of variables randomly sampled as candidates at each split. The function `tuneRF` can be used to visualize the out of bag error associated with different values of `mytry`.

```
>best_mytry <- tuneRF(attributes, Sonar$Class[train], ntreeTry=num.trees, stepFactor=1.5, improve = 0.01, trace=TRUE, plot=TRUE, dobest=FALSE)
```

Figure 42.5 illustrates the output of `tuneRF`. The optimal value occurs at `mytry = 8` with an out of the bag error around 19.75%.

We use this value of `mytry` and refit the model using the test sample.

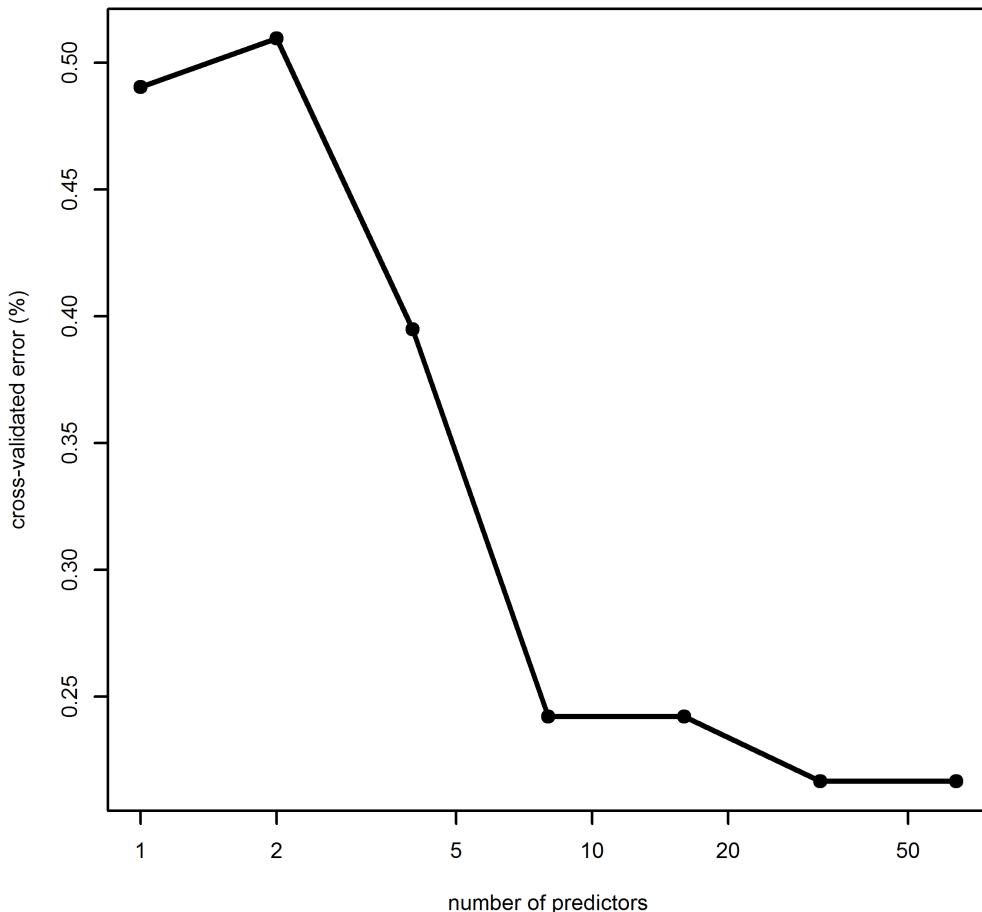


Figure 42.5: Using `tuneRF` to obtain optimal `mytry` value

```
>fit.best<-randomForest(Class~., data=Sonar[-train,], ntree=num.trees, importance=TRUE, proximity=TRUE, mtry=8)
```

```
> print(fit.best)

Call:
randomForest(formula = Class ~ ., data = Sonar[-
  train, ], ntree = num.trees,           importance =
  TRUE, proximity = TRUE, mtry = 8)
  Type of random forest: classification
  Number of trees: 1000
No. of variables tried at each split: 8

  OOB estimate of error rate: 23.53%
Confusion matrix:
      M   R class.error
M 26   3    0.1034483
R  9  13    0.4090909
```

The misclassification error for M and R is 10% and 41% respectively with an out of the bag error estimate of around 24%. Finally, we calculate and plot the Receiver Operating Characteristic for the training set, see Figure 42.6.

```
> fit.preds <- predict(fit.best, data=Sonar$Class[-
  train,-61], type = 'prob')
> preds <- fit.preds[,2]
> plot(performance(prediction(preds, Sonar$Class[-
  train]), 'tpr', 'fpr'))
> abline(a=0,b=1,lwd=2,lty=2,col="gray")
```

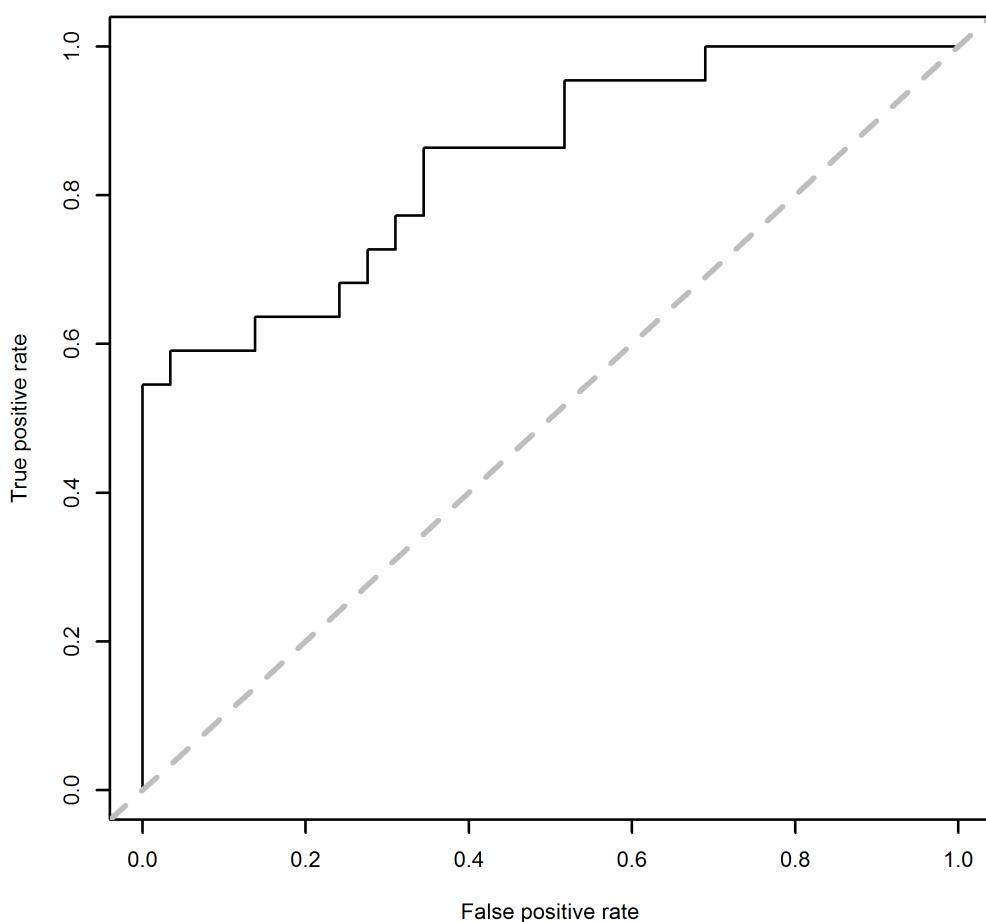


Figure 42.6: Binary Random Forest ROC for test set using Sonar

Technique 43

Binary Response Random Ferns

A binary response random ferns model can be built using the package **rFerns** with the **rFerns** function:

```
rFerns(z ~ ., data, ferns, saveErrorPropagation=TRUE  
...)
```

Key parameters include **z** the binary response variable; **data** the data set of attributes with which you wish to build the ferns; and **ferns** which controls the number of ferns to grow in each iteration, **saveErrorPropagation** a logical variable if set to **TRUE** calculates and saves the out of bag error approximation.

Step 1: Load Required Packages

We build the binary response random ferns using the **Sonar** data frame (see page 482) contained in the **mlbench** package. The package **ROCR** is also loaded. We will use this package during the final stage of our analysis to build a ROC curve.

```
> library("rFerns")  
> library(mlbench)  
> data(Sonar)  
> require(ROCR)
```

Step 2: Prepare Data & Tweak Parameters

A total 157 out of the 208 observations in **Sonar** are used to create a randomly selected training sample.

```
> set.seed(107)
> N=nrow(Sonar)
> train <- sample(1:N, 157, FALSE)
```

Step 3: Estimate and Assess the Random Ferns

We use the function `rFerns` with 1000 ferns. The parameter `saveErrorPropagation=TRUE` so that we can save the out of bag error estimates.

```
> fit<- rFerns(Class ~., data = Sonar[train,], ferns
=1000, saveErrorPropagation=TRUE)
```

The print function returns details of the fitted model:

```
> print(fit)

Forest of 1000 ferns of a depth 5.

OOB error 20.38%; OOB confusion matrix:
      True
Predicted   M   R
      M 67 17
      R 15 58
```

The forest consists of 1000 ferns of a depth 5 with an out of the bag error rate of around 20%. Let's plot the error rate.

```
> plot(fit$oobErr,xlab="Number of Ferns",ylab="OOB
Error",type="l")
```

From Figure 43.1 it looks as if the error is minimized around 335 ferns with an approximate out of the bag error rate of around 20%, To get the exact value use:

```
> which.min(fit$oobErr)
[1] 335

> fit$oobErr[which.min(fit$oobErr)]
[1] 0.1910828
```

It seems the error is minimized at 335 ferns with a oob error of 19%.

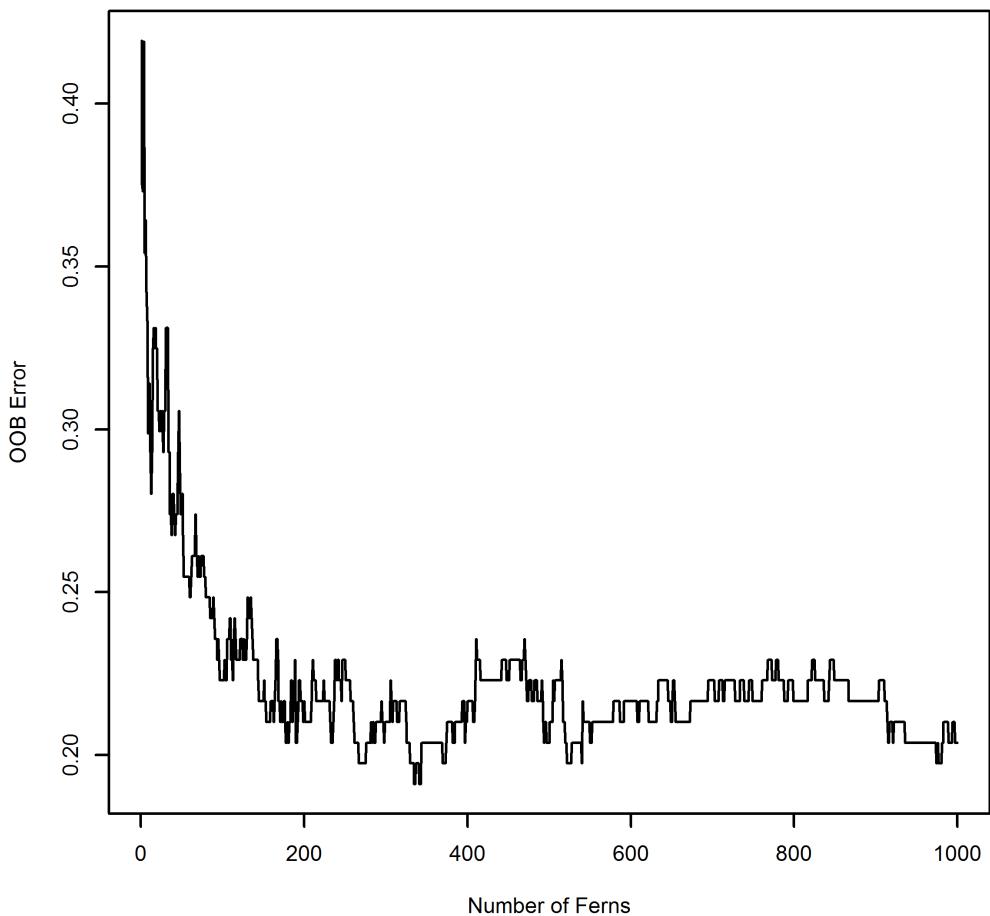


Figure 43.1: Binary Random Ferns out of bag error by number of Ferns for Sonar

Now setting `ferns = 335` we re-estimate the model.

```
> fit<- rFerns(Class ~., data = Sonar[train,], ferns  
=335, importance=TRUE, saveForest=TRUE)  
  
> print(fit)  
  
Forest of 335 ferns of a depth 5.  
  
OOB error 18.47%; OOB confusion matrix:  
True
```

```
Predicted   M   R
      M 70 17
      R 12 58
```

☛ PRACTITIONER TIP ☛

To see a print out of the error estimate as the models iterates add the parameter `reportErrorEvery` to `rFerns`. For example, `reportErrorEvery=50` returns an out of the bag error estimate at every 50th iteration. Try this:

```
test<- rFerns(Class ~., data = Sonar[train
  ,],
ferns=100,
reportErrorEvery=20)
```

Now we turn our attention to investigating variable importance. The values can be extracted from the fitted model using `fit$importance`. Since larger values indicate greater importance we use the `order` function to sort from largest to smallest, but only show the top six.

```
> imp<-fit$importance[order(-fit$importance[1]),]
> round(head(imp),3)
```

	MeanScoreLoss	SdScoreLoss
V11	0.125	0.028
V12	0.109	0.021
V10	0.093	0.016
V49	0.068	0.015
V47	0.060	0.017
V9	0.060	0.017

The output reports the top six attributes by order of importance. It also provides a measure of their variability (`SdScoreLoss`). It turns out that `V11` has the highest importance score, followed by `V12` and `V10`. Notice that `V49`, `V47` and `V9` are all clustered around 0.06.

Step 4: Make Predictions

We use the `predict` function to predict the classes using the test sample, create and print the confusion matrix and calculate the misclassification error. The

misclassification error is close to 18%.

```
> predClass<-predict(fit,Sonar[-train,])
> table( predClass ,Sonar$Class[-train] , dnn =c("Predicted Class " , " Observed Class "))

```

		Observed Class	
Predicted Class	M	R	
M	24	4	
R	5	18	

```
> error_rate = (1- sum( Sonar$Class[-train] ==
+ predClass )/ 51)
> round( error_rate ,3)
[1] 0.176
```

It can be of value to calculate the Receiver Operating Characteristic (ROC). This can be constructed using the using the `ROCR` package. First we convert the raw scores estimated using the test sample from the model `fit` into probabilities or at least values that lie in the zero to one range.

```
> predScores<-predict(fit,Sonar[-train,],scores=TRUE)
> predScores<-predScores+abs(min(predScores))
```

Next we need to generate the `prediction` object. As inputs we use the predicted class probabilities in `predScores`, and the actual test sample classes (`Sonar$Class[-train]`). The predictions are stored in `pred` and passed to the `performance` method which calculates the true positive rate and false positive rate. The result is visualized as shown in Figure 43.3 using `plot`.

```
> pred <- prediction(predScores[,2],Sonar$Class[-train])
> perf <- performance( pred , "tpr" , "fpr" )

> plot( perf )
> abline(a=0,b=1,lwd=2,lty=2,col="gray")
```

To interpret the ROC curve, let us remember that perfect classification implies a 100% true positive rate (and therefore a 0% false positive rate). This classification only happens at the upper left-hand corner of the graph and results in plot as shown in Figure 43.2.

Now looking at Figure 43.3 we see that the closer it gets to the upper left corner, the higher the classification rate. The diagonal line shown in Figure 43.3 represents random chance, so the distance of our graph over the diagonal line represents how much better it is over a random guess. The area

under the curve, which takes the value 1 for a 100% true positive rate, can be calculated using a few lines of code:

```
> auc <- performance(pred, "auc")
> auc_area <- slot(auc, "y.values")[[1]]
> round(auc_area,3)
[1] 0.929
```

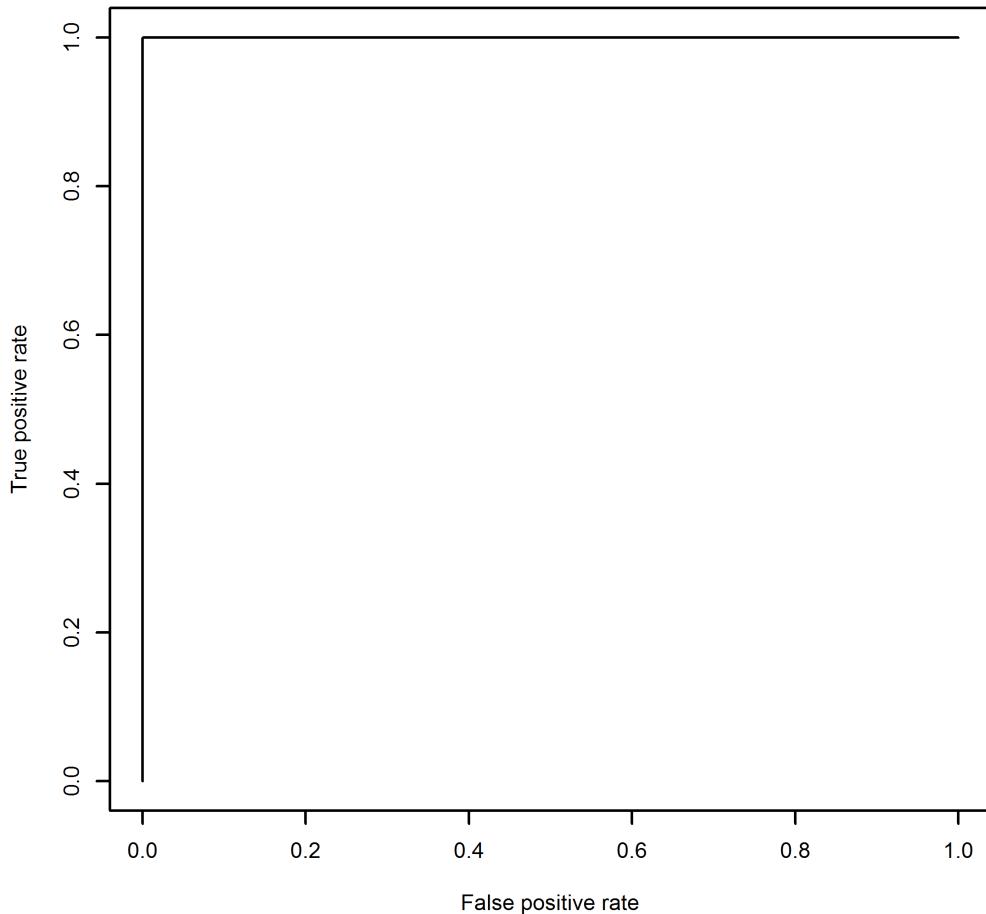


Figure 43.2: 100% true positive rate ROC curve

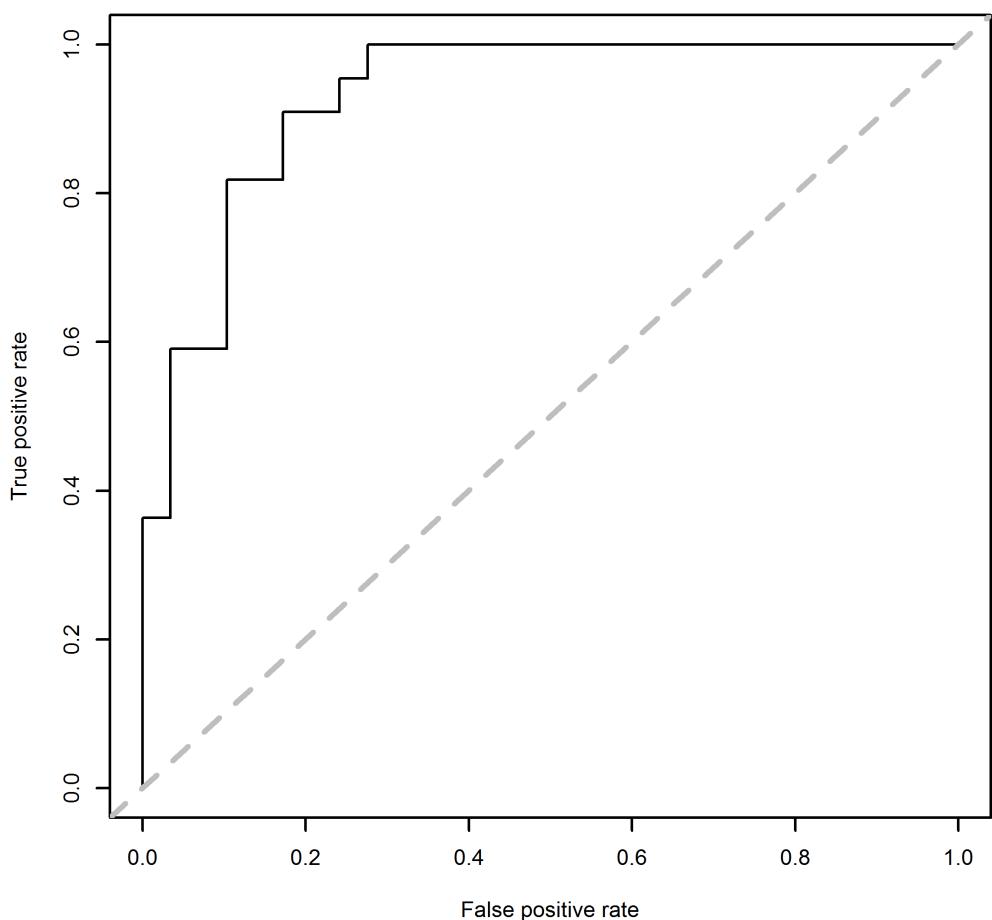


Figure 43.3: Binary Random Ferns Receiver Operating Characteristic for Sonar

Technique 44

Survival Random Forest

A survival random forest can be built using the package `randomForestSRC` with the `rfsrc` function:

```
rfsrc(z ~., data, ...)
```

Key parameters include `z` the survival response variable (note we set `z=Surv(time, status)` where `Surv` is a survival object constructed using the `survival` package) and `data` the data set of attributes with which you wish to build the forest.

Step 1: Load Required Packages

We construct a survival random forest model using the `rhDNase` data frame (see page 100) contained in the `mlbench` package.

```
> library("randomForestSRC")
> library("simexaft")
> library("survival")
> data("rhDNase")
```

Step 2: Prepare Data & Tweak Parameters

A total 600 of the 641 observations in `rhDNase` are used to create a randomly selected training sample. The rows in `rhDNase` have “funky” numbering. We use the `rownames` method to create a sequential number for each patient (1 for the first patient and 641 for the last patient).

```
> set.seed(107)
> N=nrow(rhDNase)
```

```
> rownames(rhDNase) <- 1:nrow(rhDNase)
> train <- sample(1:N, 600, FALSE)
```

The forced expiratory volume (FEV) is considered a risk factor and was measured twice at randomization (`rhDNase$fev` and `rhDNase$fev2`). We take the average of the two measurements as an explanatory variable. The variable response is defined as the logarithm of the time from randomization to the first pulmonary exacerbation measured in the object `survreg(Surv(time2, status))`.

```
> rhDNase$fev.ave <- (rhDNase$fev + rhDNase$fev2)/2
```

Step 3: Estimate and Assess the Model

We estimate the random forest using the training sample and the function `rfsrc`. The number of trees is set to 1000 and the number of variables randomly sampled as candidates at each split set equal to 3 (`mytry=3`) with a maximum of 3 (`nsplit =3`) split points chosen randomly.

```
> fit<-rfsrc(Surv(time2, status) ~ trt + fev+fev2+
  fev.ave,data= rhDNase[train,], nsplit = 3, ntree =
  1000 ,mtry=3)
```

The error rate by number of trees and variable importance are given using the `plot` method - see Figure 44.1. The error rate levels out around 400 trees, remaining relatively constant until around 600 trees. Our constructed variable `fev.ave` is indicated as the most important variable, followed by `fev2`.

```
> plot(fit)
```

	Importance	Relative Imp
fev.ave	0.0206	1.0000
fev2	0.0197	0.9586
fev	0.0073	0.3529
trt	0.0005	0.0266

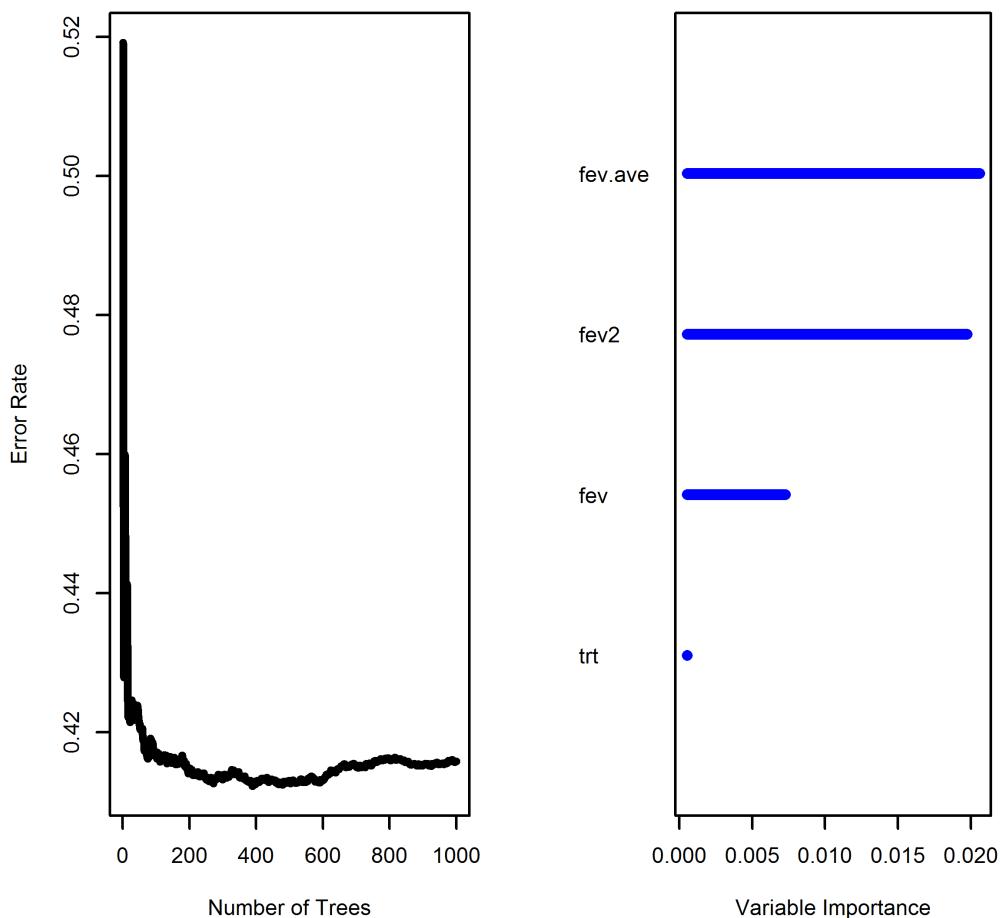


Figure 44.1: Survival random forest error rate and variable importance for rhDNase

Further details of variable importance can be explored using a range of methods via the `vimp` function and the parameter `importance`. This parameter can take four distinct values "`permute`", "`random`", "`permute.ensemble`", "`random.ensemble`". Let's calculate all four methods.

```
> permute<-vimp(fit, importance="permute")$importance
> random<-vimp(fit, importance="random")$importance
> permute.ensemble<-vimp(fit, importance="permute.
  ensemble")$importance
> random.ensemble<-vimp(fit, importance="random .
```

```
ensemble")$importance
```

We combine the results using `rbind` into the matrix `tab`. Overall we see that `fev2` and `fev.average` appear to be the most important attributes.

```
> tab<-rbind(permute,random,permute.ensemble,random.ensemble)
> round(tab,3)
```

	trt	fev	fev2	fev.ave
permute	0.001	0.007	0.021	0.021
random	0.001	0.010	0.015	0.018
permute.ensemble	-0.002	-0.027	-0.015	-0.016
random.ensemble	-0.004	-0.025	-0.027	-0.026

☛ PRACTITIONER TIP ☛

Another way to assess variable importance is using the `var.select` method. It takes the form `var.select(object = fitted model, method="md")`. Method can take the values "md" for minimal depth (default), "vh" for variable hunting and "vh.vimp" for variable hunting. As an illustration using the fitted model `fit` type:

```
> md <- var.select(object = fit,method="md")
> vh <- var.select(object = fit,method="vh")
> vh.vimp <- var.select(object = fit,
  method="vh.vimp")
```

Searching for interaction effects is an important task in many areas of statistical modeling. A useful tool is the method `find.interaction`. Key parameters for this method include the number of variables to be used (`nvar`) and the method (`method`). We set `method = "vimp"` to use the approach of Ishwaran⁸⁴. In this method the importance of each variable is calculated. Their paired variable importance is also calculated. The sum of these two values is known as “Additive” importance. A large positive or negative difference between “Paired” and “Additive” indicates a potential interaction provided the individual variable importance scores are large.

```
> find.interaction(fit, nvar = 8,method="vimp")
               Method: vimp
```

```
          No. of variables: 4
Variables sorted by VIMP?: TRUE
No. of variables used for pairing: 4
Total no. of paired interactions: 6
Monte Carlo replications: 1
Type of noising up used for VIMP: permute
```

	Paired	Additive	Difference
fev.ave:fev2	0.0346	0.0405	-0.0058
fev.ave:fev	0.0296	0.0284	0.0012
fev.ave:trt	0.0217	0.0222	-0.0005
fev2:fev	0.0283	0.0270	0.0013
fev2:trt	0.0196	0.0208	-0.0012
fev:trt	0.0134	0.0090	0.0045

Since the differences across all variable pairs appear rather small, we conclude there is little evidence of an interaction effect.

Now we investigate interactions effects using `method="maxsubtree"`. This invokes a maximal subtree analysis⁸⁵.

```
> find.interaction(fit, nvar = 8, method="maxsubtree"
  )
```

```
Method: maxsubtree
          No. of variables: 4
Variables sorted by minimal depth?: TRUE

          fev.ave  fev2   fev   trt
fev.ave      0.07  0.12  0.13  0.21
fev2        0.12  0.08  0.13  0.20
fev         0.12  0.12  0.09  0.23
trt         0.13  0.13  0.16  0.15
```

In this case, reading across the rows in the resultant table, small [i][j] values with small [i][i] values are an indication of a potential interaction between attribute i and attribute j.

Reading across the rows, we do not see any such values, and again we do not find evidence supporting an interaction effect.

Step 4: Make Predictions

Predictions using the test sample can be obtained using the `predict` method. The results are stored in `pred`.

```
> pred <- predict(fit, data= rhDNase[train,])
```

Finally, we use the `plot` and `plot.survival` methods to visualize the predicted outcomes, see Figure 44.2 and Figure 44.3.

```
> plot(pred)
> plot.survival(pred)
```

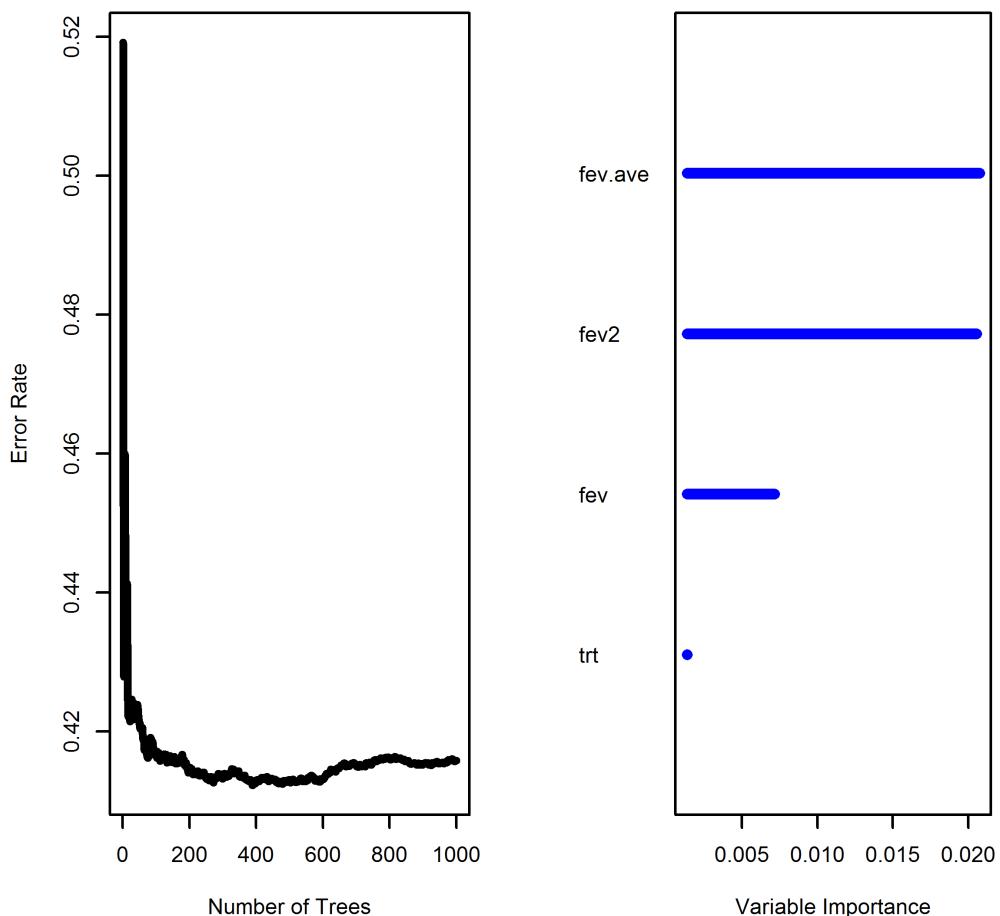


Figure 44.2: Survival random forest visualization of `plot(pred)` using `rhDNase`

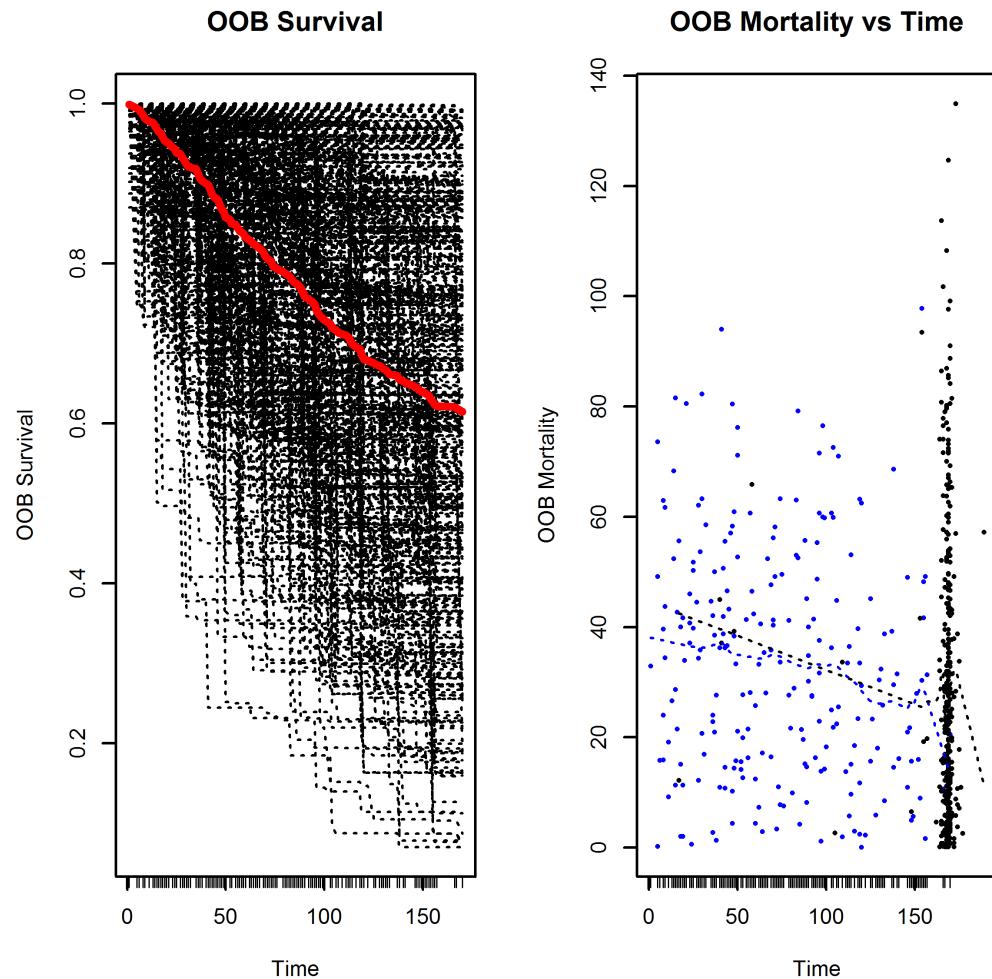


Figure 44.3: Survival random forest visualization of prediction

Technique 45

Conditional Inference Survival Random Forest

A conditional inference Survival random forest can be built using the package `party` with the `cforest` function:

```
cforest(z ~ ., data, controls, ...)
```

Key parameters include `controls` which controls parameters such as the number of trees grown in each iteration, `z` the survival response variable (note we set `z=Surv(time, status)` where `Surv` is a survival object constructed using the `survival` package) and `data` the data set of attributes with which you wish to build the forest.

Step 1: Load Required Packages

We construct a conditional inference survival random forest model using the `rhDNase` data frame (see page 100) contained in the `mlbench` package.

```
> library("party")
> library("simexaft")
> library("survival")
> data("rhDNase")
```

Step 2: Prepare Data & Tweak Parameters

We will use all 641 observations in `rhDNase` for our analysis. The rows in `rhDNase` have “funky” numbering. Therefore we use the `rownames` method to create a sequential number for each patient (1 for the first patient and 641 for the last patient).

```
> set.seed(107)
> rownames(rhDNase)
```

The forced expiratory volume (FEV) is considered a risk factor and was measured twice at randomization (`rhDNase$fev` and `rhDNase$fev2`). We take the average of the two measurements as an explanatory variable. The response is defined as the logarithm of the time from randomization to the first pulmonary exacerbation measured in the object `survreg(Surv(time2, status))`.

```
> rhDNase$fev.ave <- (rhDNase$fev + rhDNase$fev2)/2
> z<-Surv(rhDNase$time2, rhDNase$status)
```

Step 3: Estimate and Assess the Model

We estimate the random forest using the training sample and the function `cforest`. The control `cforest_unbiased` is used to set the number of trees (`ntree = 10`) and to set the number of variables randomly sampled as candidates at each split (`mtry = 2`).

```
fit<-cforest(z ~ trt + fev.ave,data= rhDNase ,
  controls =cforest_unbiased(ntree = 10,mtry=2))
```

Step 4: Make Predictions

We will estimate conditional Kaplan-Meier survival curves for individual patients. First we use the `predict` method.

```
> pred<-predict(fit,newdata=rhDNase ,type = "prob")
```

Next we use the `plot` method to graph the conditional Kaplan-Meier curves for the individual patients (patient 1, 137, 205 and 461), see Figure 45.1.

```
> op <- par(mfrow=c(2, 2))
> plot(pred[[1]],main="Patient 1",xlab="Status",ylab
  ="Time")
> plot(pred[[137]],main="Patient 137",xlab="Status",
  ylab="Time")
> plot(pred[[205]],main="Patient 205",xlab="Status",
  ylab="Time")
> plot(pred[[461]],main="Patient 461",xlab="Status",
  ylab="Time")
```

```
> par(op)
```

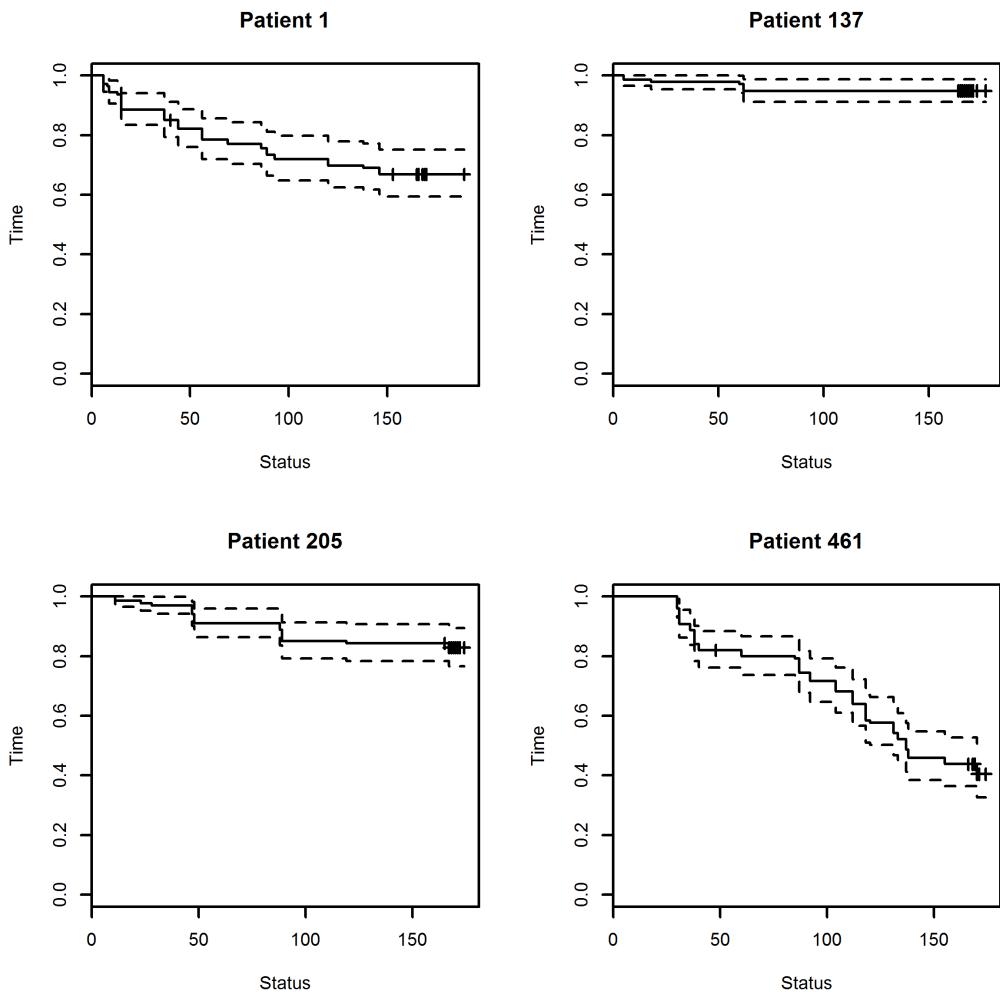


Figure 45.1: Survival random forest Kaplan-Meier curves for the individual patients using rhDNase

Technique 46

Conditional Inference Regression Random Forest

A conditional inference regression random forest can be built using the package `party` with the `cforest` function:

```
cforest(z ~ ., data, , controls, ...)
```

Key parameters include `controls` which controls parameters such as the number of trees grown in each iteration, `z` the continuous response variable and `data` the data set of attributes with which to build the forest.

Step 1: Load Required Packages

We build the conditional inference regression random forest using the `bodyfat` (see page 62) data frame contained in the `TH.data` package. We also load the `caret` package as it provides handy performance metrics for random forests.

```
> library ("party")
> data("bodyfat", package = "TH.data")
> library(caret)
```

Step 2: Prepare Data & Tweak Parameters

Following the approach taken by Garcia et al, we use 45 of the 71 observations to build the regression tree. The remainder will be used for prediction. The 45 training observations were selected at random without replacement.

```
> set.seed(465)
> train <- sample(1:71, 45, FALSE)
```

Step 3: Estimate and Assess the Decision Tree

We estimate the random forest using the training sample and the function `cforest`. The control `cforest_unbiased` is used to set the number of trees (`ntree = 100`) and to set the number of variables randomly sampled as candidates at each split (`mtry = 5`).

```
> fit<-cforest(DEXfat ~ ., data = bodyfat[train,],  
  controls =cforest_unbiased(ntree = 100,mtry=5))
```

The fitted model root mean square error and R-squared are obtained using the `caret` package:

```
> round(caret:::cforestStats(fit),4)  
  RMSE Rsquared  
 5.1729   0.8112
```

We calculate variable importance using three alternate methods. The first method calculates the unconditional mean decrease in accuracy using the approach of Hapfelmeier et al ⁸⁶.

```
> ord1<-varimp(fit)  
> imp1<-ord1[order(-ord1[1:3])]  
  
> round(imp1,3)  
 hipcirc waistcirc      age  
 69.957     47.366     0.000
```

The second approach calculates the unconditional mean decrease in accuracy using the approach of Breiman⁸⁷.

```
> ord2<-varimp(fit,pre1.0_0 = TRUE)  
  
> imp2<-ord2[order(-ord2[1:3])]  
  
> round(imp2,3)  
 hipcirc waistcirc      age  
 72.527     40.501     0.000
```

The third approach calculates the conditional mean decrease in accuracy using the approach of Breiman.

```
> ord3<-varimp(fit, conditional = TRUE)  
  
> imp3<-ord3[order(-ord2[1:3])]
```

```
> round(imp3,3)
  hipcirc  waistcirc      age
  66.600    47.218     0.000
```

It is informative to note that all three approaches identify `hipcirc` as the most important variable, followed by `waistcirc`.

Step 5: Make Predictions

We use the test sample observations and the fitted regression forest to predict `DEXfat`. The scatter plot between predicted and observed values is shown in Figure 46.1. The squared correlation coefficient between predicted and observed values is 0.756.

```
> pred<-predict(fit,newdata=bodyfat[-train],type =
  "response")

> plot(bodyfat$DEXfat[-train],pred,xlab="DEXfat",
  ylab="Predicted Values",, main="Training Sample
  Model Fit")

> round(cor(pred,bodyfat$DEXfat[-train])^2,3)
  [,1]
DEXfat 0.756
```

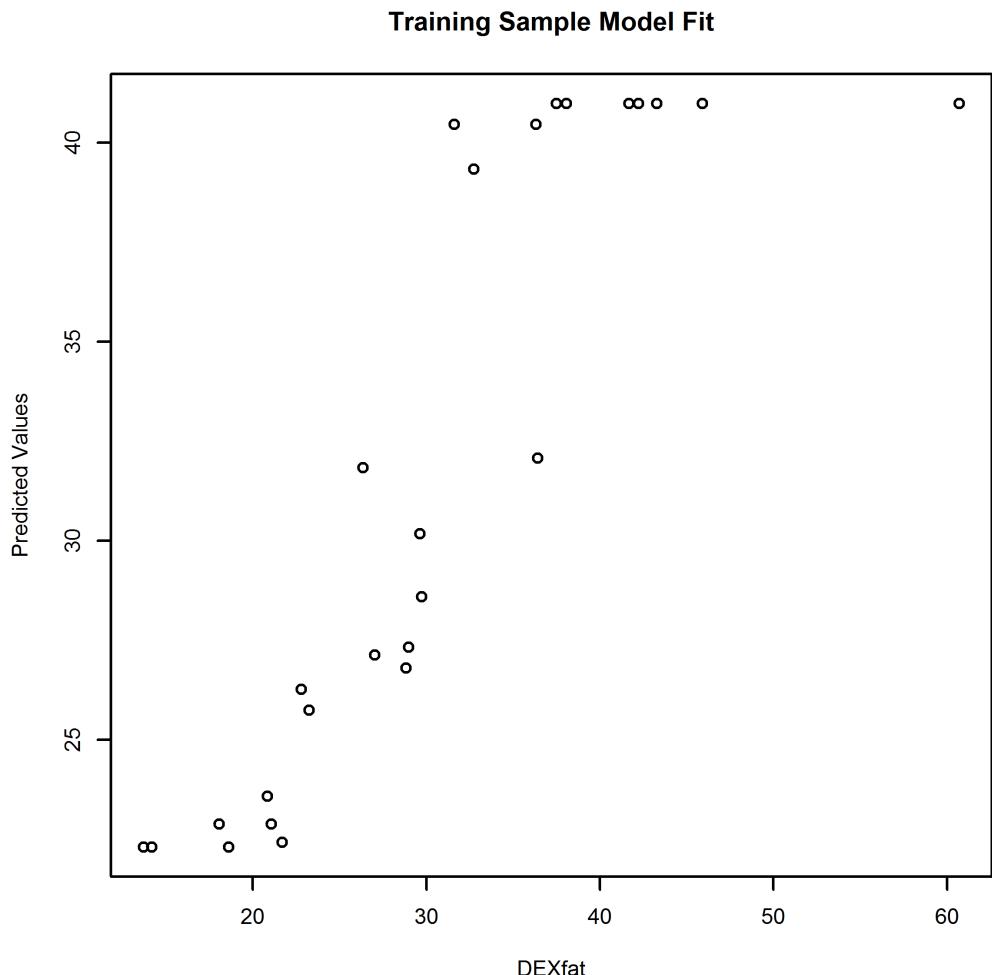


Figure 46.1: Conditional Inference regression forest scatter plot for DEXfat

Technique 47

Quantile Regression Forests

Quantile regression forests are a generalization of random forests and offer a non-parametric way of estimating conditional quantiles of predictor variables⁸⁸. Quantile regression forests can be built using the package `quantregForest` with the `quantregForest` function:

```
quantregForest(y,x, data,,ntree,quantiles...)
```

Key parameters include `y` the continuous response variable, `x` the data set of attributes with which you wish to build the forest, `ntree` the number of trees and `quantiles` the quantiles you wish to include.

Step 1: Load Required Packages

We build the Quantile Regression Forests using the `bodyfat` (see page 62) data frame contained in the `TH.data` package.

```
> require(quantregForest)
> data("bodyfat",package="TH.data")
```

Step 2: Prepare Data & Tweak Parameters

Following the approach taken by Garcia et al, we use 45 of the 71 observations to build the regression tree. The remainder will be used for prediction. The 45 training observations were selected at random without replacement. We use `train` to partition the data into a validation and test sample.

```
> train <- sample(1:71,45 , FALSE)
> num.trees=2000
> xtrain<-bodyfat[train,]
```

```
> xtrain$DEXfat <- NULL  
> xtest<-bodyfat[-train,]  
> xtest$DEXfat <- NULL  
> DEXtrain<-bodyfat$DEXfat[train]  
> DEXtest<-bodyfat$DEXfat[-train]
```

Step 3: Estimate and Assess the Model

We use the `quantregForest` method setting the number of variables randomly sampled as candidates at each split and node size equal to 5 (`mytry = 5, nodesize = 5`). We use `fit` to contain the results of the fitted model.

```
> fit<- quantregForest(y=DEXtrain, x=xtrain, mtry= 5  
 ,ntree=num.trees, nodesize= 5,importance=TRUE,  
 quantiles=c(0.25,0.5,0.75))
```

Use the `print` method to see a summary of the estimated model.

```
> print(fit)
```

Call:

```
quantregForest(x = xtrain, y = DEXtrain, mtry = 5,  
nodesize = 5, ntree = num.trees, importance  
= TRUE, quantiles = c(0.25,0.5, 0.75))
```

Number of trees: 2000

No. of variables tried at each split: 5

To see variable importance across all fitted quantiles use the method `varImpPlot.qrf`. A visualization using this method for `fit` is shown in Figure 47.1. It appears `hipcirc` and `waistcirc` are the most influential variables for the median and 75th quantile.

```
> varImpPlot.qrf(fit)
```

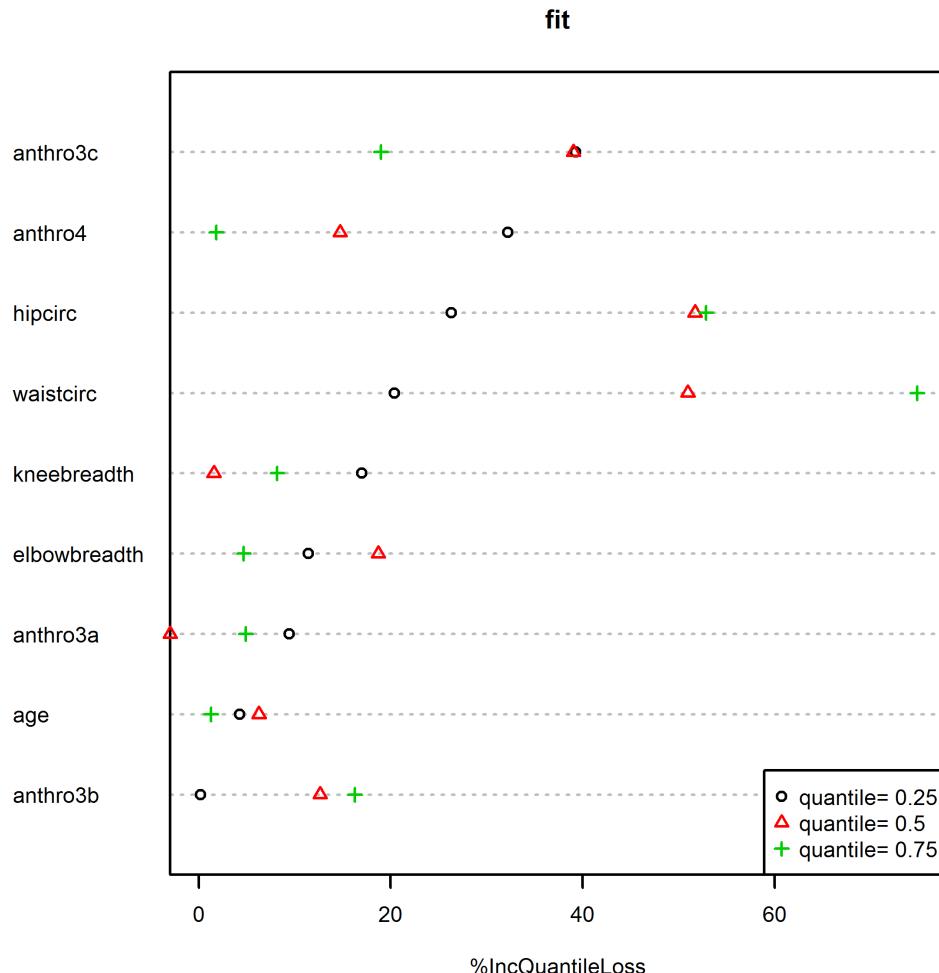


Figure 47.1: Quantile regression forests variable importance across all fitted quantiles for **bodyfat**

Let's look in a little more detail at the median. This is achieved using the `varImpPlot.qrf` method and setting `quantile=0.5`. The result is illustrated in Figure 47.2.

```
> varImpPlot.qrf(fit, quantile=0.5)
```

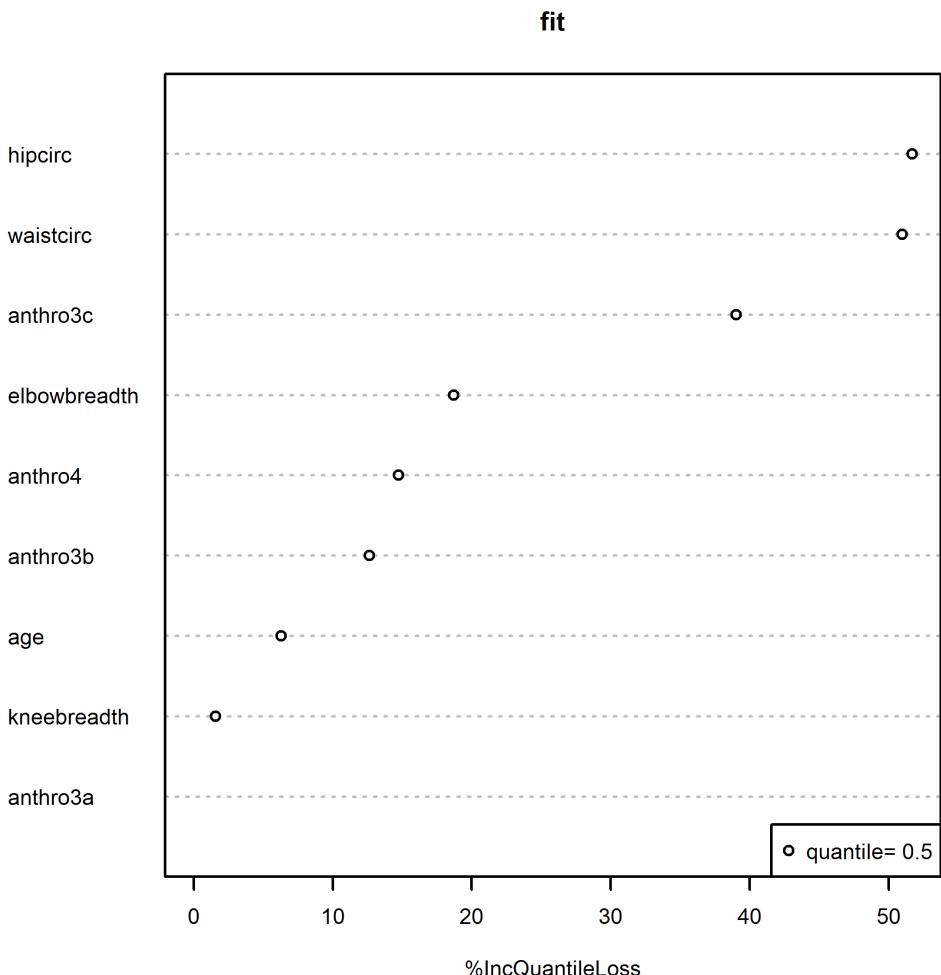


Figure 47.2: Quantile regression forests using the `varImpPlot.qrf` method and setting `quantile=0.5` for `bodyfat`

The `plot` method returns the 90% prediction interval on the estimated data, see Figure 47.3. It appears the model is well calibrated to the data.

```
> plot(fit)
```

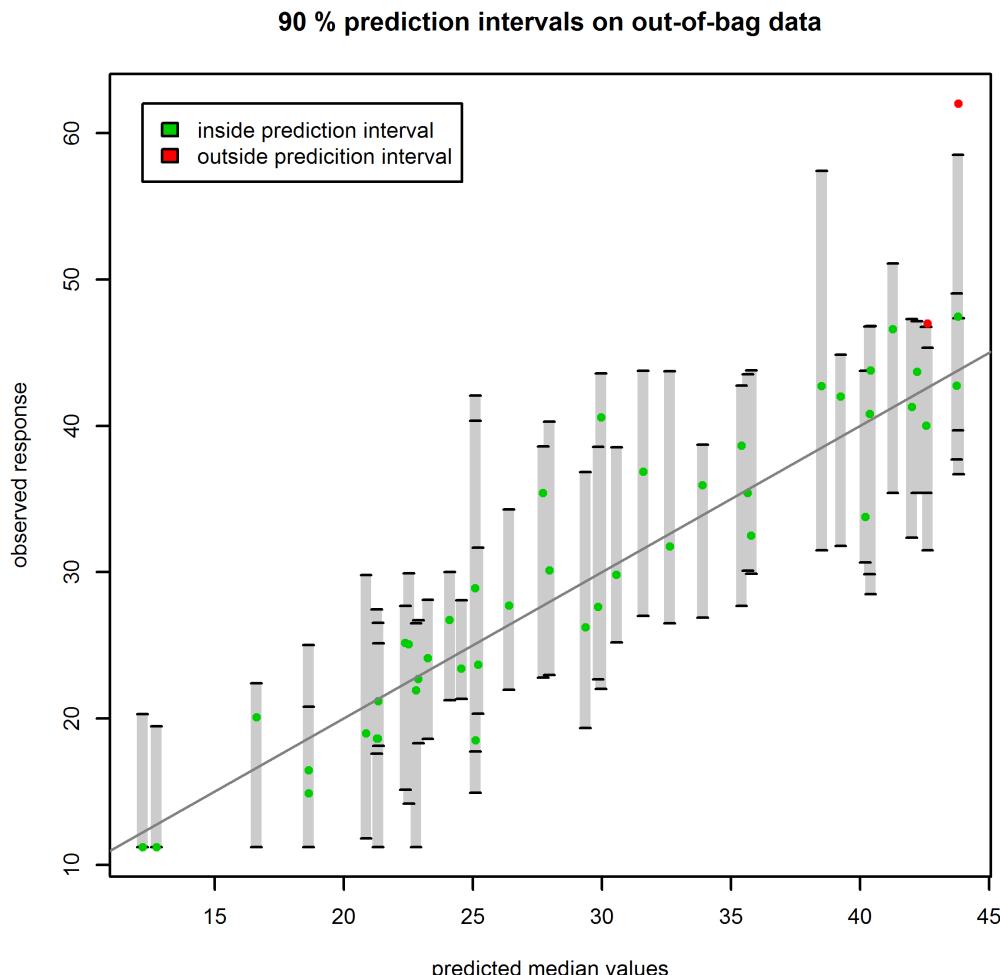


Figure 47.3: Quantile Regression Forests 90% prediction interval using bodyfat

Step 4: Make Predictions

We use the test sample observations and the fitted regression forest to predict DEXfat. Notice we set `all = TRUE` to use all observations for prediction.

```
> pred<- predict(fit, newdata= xtest, all=TRUE)
```

To take a closer look at the value of `pred` you can enter.

```
> head(pred)
    quantile= 0.1 quantile= 0.5 quantile= 0.9
[1,]      34.22470      41.53230      47.14400
```

[2,]	35.40183	41.53541	52.41859
[3,]	22.50604	27.00303	35.40398
[4,]	27.64735	35.56909	42.73546
[5,]	21.30879	26.20480	35.33114
[6,]	32.49331	38.86571	44.52887

The plot of the fitted and observed values is shown in Figure 47.4. The squared correlation coefficient between the predicted and observed values is 0.898.

```
> plot(DEXtest, pred[,2], xlab="DEXfat", ylab="Predicted Values (median)", , main="Training Sample Model Fit")  
  
> round(cor(pred[,2], DEXtest)^2, 3)  
[1] 0.898
```

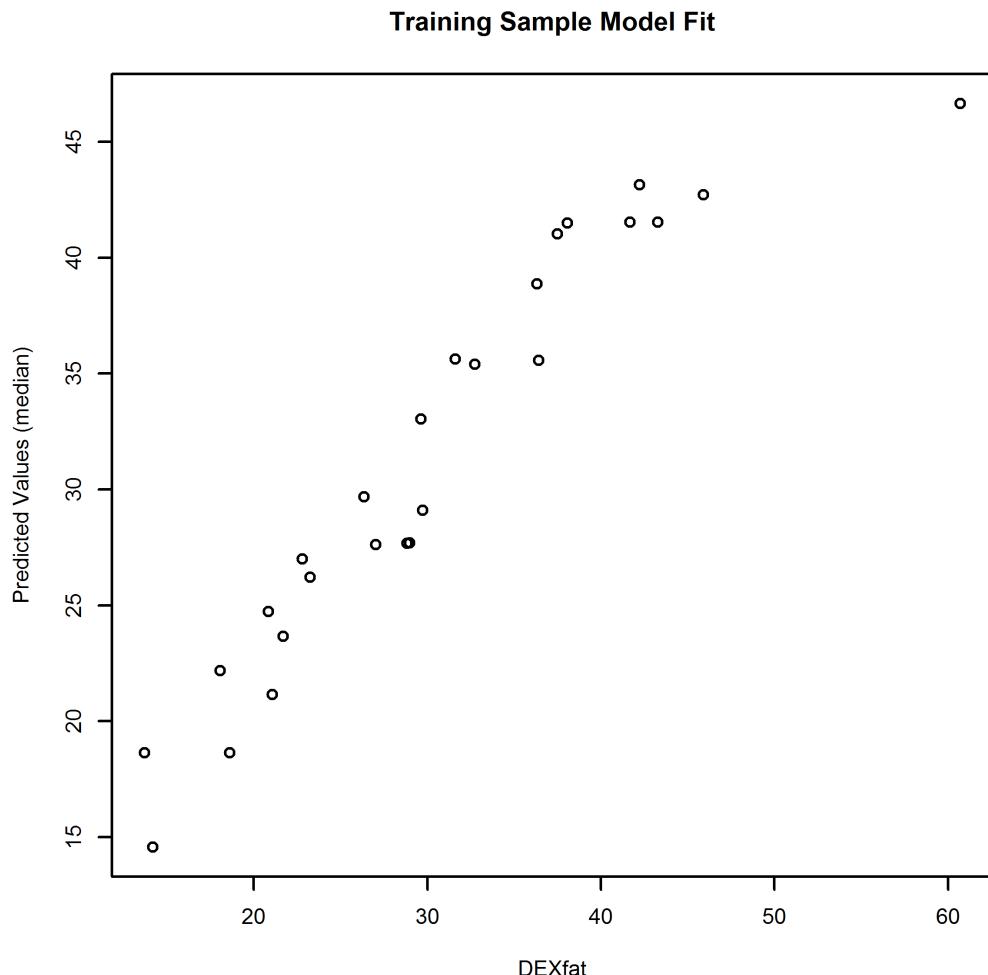


Figure 47.4: Quantile regression forests scatter plot between predicted and observed values using `bodyfat`

Technique 48

Conditional Inference Ordinal Random Forest

Conditional inference ordinal random forests are used when the response variable is measured on an ordinal scale. In marketing, for instance, we often see consumer satisfaction measured on an ordinal scale - “very satisfied”, “satisfied”, “dissatisfied” and “very dissatisfied”. In medical research constructs such as self-perceived health are often measured on an ordinal scale - “very unhealthy,” “unhealthy”, “healthy”, “very healthy”. Conditional inference ordinal random forests can be built using the package `party` with the `cforest` function:

```
cforest(z ~ ., data, , controls, ...)
```

Key parameters include `controls` which controls parameters such as the number of trees grown in each iteration, response variable `Z` an ordered factor and `data` the data set of attributes with which you wish to build the forest.

Step 1: Load Required Packages

We build our random forest using the `wine` data frame contained in the `ordinal` package (see page 95). We also load the `caret` package as it provides handy performance metrics for random forests.

```
> library(caret)
> library("ordinal")
> data("wine")
```

Step 2: Prepare Data & Tweak Parameters

We use `train` to partition the validation set using fifty observations to build the model and the remainder as the test set.

```
> set.seed(107)
> N=nrow(wine)
> train <- sample(1:N, 50, FALSE)
```

Step 3: Estimate and Assess the Model

We use the `cforest` method with 100 trees and set the number of variables randomly sampled as candidates at each split equal to 5 (`mtry = 5`). We use `fit` to contain the results of the fitted model.

```
> fit<-cforest(rating ~., data = wine[train,],
  controls =cforest_unbiased(ntree = 100,mtry=5))
```

Accuracy and kappa are obtained using `caret`.

```
> round(caret:::cforestStats(fit),4)
Accuracy      Kappa
 0.8600      0.7997
```

The fitted model has reasonable accuracy and kappa statistics. We next calculate variable importance using three alternate methods. The first method calculates the unconditional mean decrease in accuracy using the approach of Hapfelmeier et al (See page 335 for further details of these methods).

```
> ord1<-varimp(fit)

> imp1<-ord1[order(-ord1[1:5])]
> round(imp1,3)
response      temp    contact     bottle     judge
  0.499      0.000      0.000      0.000      0.000
```

The second approach calculates the unconditional mean decrease in accuracy using the approach of Breiman.

```
> ord2<-varimp(fit,pre1.0_0 = TRUE)

> imp2<-ord2[order(-ord2[1:5])]
> round(imp2,3)
response      temp    contact     bottle     judge
  0.448      0.000      0.000      0.000      0.000
```

The third approach calculates the conditional mean decrease in accuracy using the approach of Breiman.

```
> ord3<-varimp(fit, conditional = TRUE)

> imp3<-ord3[order(-ord2[1:5])]

> round(imp3,3)
response      temp   contact     bottle      judge
0.481       0.000     0.000     0.000     0.000
```

For all three methods it seems the response (wine bitterness score) is the only informative attribute in terms of importance.

Step 4: Make Predictions

We next use the test sample observations and `cforest` and display the accuracy and kappa using `caret`.

```
> fit_test<-cforest(rating ~., data = wine[-train,])
> round(caret:::cforestStats(fit_test),4)
Accuracy      Kappa
0.3182      0.0000
```

Accuracy is now around 32% and kappa has fallen to zero! Although these number are not very encouraging, nevertheless we investigate the predictive performance using the `predict` method and storing the results in `pred`.

```
> pred<-predict(fit,newdata=wine[-train,],type =
  "response")
```

Now we compare the predicted values to those actually observed by using the `table` method to create a confusion matrix.

```
> tb<-table(wine$rating[-train],pred, dnn=c("actual",
  , "predicted"))
> tb
  predicted
actual 1 2 3 4 5
  1 0 3 0 0 0
  2 0 6 0 0 0
  3 0 0 7 0 0
  4 0 0 0 4 0
  5 0 0 0 2 0
```

Finally, the misclassification rate can be calculated. We see it is around 23% for the test sample.

```
> error <- 1-(sum(diag(tb))/sum(tb))
> round(error,3)
[1] 0.227
```

Notes

⁷²See Ozuysal M, Fua P, Lepetit V (2007). Fast Keypoint Recognition in Ten Lines of Code." In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2007), pp. 1{8.

⁷³Please visit <http://www.un-redd.org/>

⁷⁴See for example:

1. Evans JS, Murphy MA, Holden ZA, Cushman SA (2011) Modeling species distribution and change using random forest. In: Drew CA, Wiersma YF, Huettmann F, editors. Predictive species and habitat modeling in landscape ecology: concepts and applications. New York City, NY, USA: Springer Science+Business Media. pp. 139–159;
2. Breiman L (2001) Statistical modeling: The two cultures (with comments and a rejoinder by the author). *Statistical Science* 16: 199–231.

⁷⁵See for example:

1. Baccini A, Goetz SJ, Walker WS, Laporte NT, Sun M, et al. (2012) Estimated carbon dioxide emisclassificationions from tropical deforestation improved by car-bondensity maps. *Nature Climate Change* 2: 182–185;
2. Drake JB, Knox RG, Dubayah RO, Clark DB, Condit R, et al. (2003) Above ground biomass estimation in closed canopy neotropical forests using lidar remote sensing: Factors affecting the generality of relationships. *Global Ecology and Biogeography* 12: 147–159;
3. And Hudak AT, Strand EK, Vierling LA, Byrne JC, Eitel JUH, et al. (2012) Quantifying above ground forest carbon pools and fluxes from repeat LiDAR surveys. *Remote Sensing of Environment* 123: 25–40.

⁷⁶Mascaro, Joseph, et al. "A tale of two “forests”: Random Forest machine learning aids tropical forest carbon mapping." (2014): e85993.

⁷⁷Farney, Robert J., et al. "Sleep disordered breathing in patients receiving therapy with buprenorphine/naloxone." *European Respiratory Journal* 42.2 (2013): 394-403.

⁷⁸Vitorino, D., et al. "A Random Forest Algorithm Applied to Condition-based Waste-water Deterioration Modeling and Forecasting." *Procedia Engineering* 89 (2014): 401-410.

⁷⁹Sugimoto, Koichiro, et al. "Cross-sectional study: Does combining optical coherence tomography measurements using the ‘Random Forest’decision tree classifier improve the prediction of the presence of perimetric deterioration in glaucoma suspects?." *BMJ open* 3.10 (2013): e003114.

⁸⁰Kanerva, N., et al. "Random forest analysis in identifying the importance of obesity risk factors." *European Journal of Public Health* 23.suppl 1 (2013): ckt124-042.

⁸¹Chen, Xue-Wen, and Mei Liu. "Prediction of protein–protein interactions using random decision forest framework." *Bioinformatics* 21.24 (2005): 4394-4400.

⁸²Alexander Hapfelmeier, Torsten Hothorn, Kurt Ulm, and Carolin Strobl (2012). A New Variable Importance Measure for Random Forests with misclassified Data. *Statistics and Computing*, <http://dx.doi.org/10.1007/s11222-012-9349-1>

⁸³Leo Breiman (2001). Random Forests. *Machine Learning*, 45(1), 5–32.

⁸⁴See Ishwaran H. (2007). Variable importance in binary regression trees and forests, *Electronic J. Statist.*, 1:519-537.

⁸⁵For more details of this method see:

1. Ishwaran H., Kogalur U.B., Gorodeski E.Z, Minn A.J. and Lauer M.S. (2010). High-dimensional variable selection for survival data. *J. Amer. Statist. Assoc.*, 105:205-217.
2. Ishwaran H., Kogalur U.B., Chen X. and Minn A.J. (2011). Random survival forests for highdimensional data. *Statist. Anal. Data Mining*, 4:115-132.

⁸⁶See Alexander Hapfelmeier, Torsten Hothorn, Kurt Ulm, and Carolin Strobl (2012). A New Variable Importance Measure for Random Forests with misclassified Data. *Statistics and Computing*, <http://dx.doi.org/10.1007/s11222-012-9349-1>

⁸⁷see Leo Breiman (2001). Random Forests. *Machine Learning*, 45(1), 5–32.

⁸⁸See Meinshausen, Nicolai. "Quantile regression forests." *The Journal of Machine Learning Research* 7 (2006): 983-999.

Part VI

Cluster Analysis

The Basic Idea

Cluster analysis is an exploratory data analysis tool which aims to group a set of objects in such a way that objects in the same group are more similar to each other than to those in other groups. The groups of similar objects are called clusters.

Cluster analysis itself is not one specific algorithm but consists of a wide variety of approaches. Two of the most popular are hierarchical clustering and partition based clustering. We discuss each briefly below.

Hierarchical Clustering

Hierarchical clustering methods attempt to maximize the distance between clusters. They begin by assigning each object to its own cluster. At each step the algorithm merges together the least distant pair of clusters, until only one cluster remains. At every step the distance between clusters, say cluster A and cluster B is updated.

There are two basic approaches used in hierarchical clustering algorithms. The first is known as agglomerative and the second divisive.

Agglomerative Approach

The agglomerative approach begins with each observations considered a separate cluster. These observations are then merged successively into larger clusters until a hierarchy of clusters, known as a dendrogram emerges.

Figure 54.4 shows a typical dendrogram using the `thyroid` data set. Further details of this data are discussed on page 385. Because agglomerative clustering algorithms begin with the individual observations then grow to larger clusters it is known as a bottom up approach to clustering.

Divisive Approach

The divisive approach begins with the entire sample of observations and then proceeds to divide it into smaller clusters. Two approaches are generally taken. The first uses a single attribute to determine the split at each step. This approach is called monothetic. The second approach uses all attributes at each step and is known as polythetic. Since for a sample of n observations there are $2^{n-1}-1$ possible divisions of clusters the divisive approach is computationally intense. Figure 55.2 shows an example of a dendrogram produced using the divisive approach.

Partition Based Clustering

Partitioning clustering methods attempt to minimize the “average” distance within clusters. They typically proceed as follows:

1. Select at random group centers and assign objects to the nearest group.
2. Form new centers at the center of these groups.
3. Move individual objects to a new group if it is closer to that group than the center of its present group.
4. Repeat step 2 and 3 until no (or minimal) change in groupings occurs.

K- Means

The most popular partition based clustering algorithm is the k-means method. It requires a distance metric and the number of centroids (clusters) to be specified. Here is how it works:

1. Determine how many clusters you expect in your sample, say for example k .
2. Pick a group of k random centroids. A centroid is the center of a cluster.
3. Assign sample observations to the closest centroid. The closeness is determined by a distance metric. They become part of that cluster.
4. Calculate the center (mean) of each cluster.
5. Check assignments for all the sample observations. If another center is closer to an observation, reassign the it to that cluster.

-
6. Repeat step 3-5 until no reassignment occur.

Much of the popularity of k-means lies in the fact that the algorithm is extremely fast and can therefore be easily applied to large data-sets. However, it is important to note that the solution is a local maximum, so in practice several starting points should be used.

Practical Applications

NOTE... ↗

Different clustering algorithms may produce very different clusters. Unfortunately, there is no generally accepted “best” method. One way to assess an algorithm is to take a data set with a known structure and see whether the algorithm is able to reproduce this structure. This is the approach taken by Musmeci et al.

Financial Market Structure

Musmeci et al⁸⁹ study the structure of the New York Stock Exchange using various clustering techniques. A total of 4026 daily closing prices from 342 stocks are collected for their analysis. Both trended and detrended log price returns are considered for analysis. Four of the clustering methods considered are Single Linkage (SL), Average Linkage (AL), Complete Linkage (CL) and k-medoids using the Industrial Classification Benchmark⁹⁰ (ICB) as a know classification outcome reference.

Figure 48.1 illustrates the findings of their analysis for a predefined cluster size equal to 23. SL appears to generate one very large cluster which contains stocks in all the ICB sectors; this is clearly inaccurate. AL, CL and k-medoids show more structured clustering, although for AL there remain a number of clusters with less than 5 stocks⁹¹.

The Adjusted Rand index and number of sectors identified by each method are presented in Table 21. ICB has 19 sectors, and one would hope that the clustering methods would report a similar number of clusters. For the trended stock returns k-medoids comes closet with 17 sectors and a Adjusted Rand index of 0.387. For the detrended case k-medoids has a lower Adjusted Rand index than CL and AL, but it again comes closet to recovering the actual number of sectors in ICB.

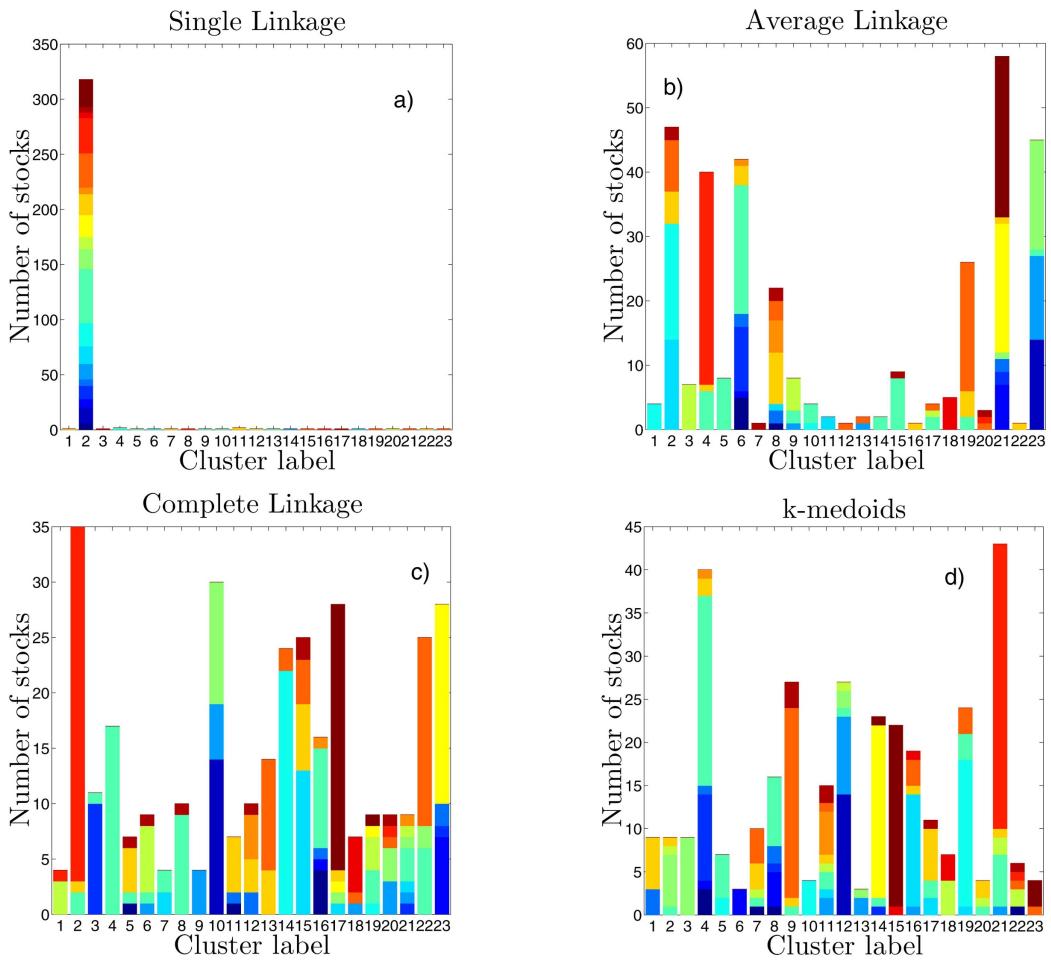


Figure 48.1: Musmeci et al's composition of clustering in terms of ICB super-sectors. The x-axis represents the cluster labels, the y-axis the number of stocks in each cluster. Color and shading represent the ICB super-sectors. Source Musmeci et al.

	with trend			
	k-medoids	CL	AL	SL
Adjusted Rand index	0.387	0.387	0.352	0.184
Clusters	17	39	111	229
	detrended			
Adjusted Rand index	0.467	0.510	0.480	0.315
Clusters	25	50	60	101

Table 21: Adjusted Rand index and number of estimated sectors by clustering model reported by Musmeci et al

☛ PRACTITIONER TIP ☛

Musmeci et al use the Adjusted Rand index as part of their analysis. The index is a measure of the similarity between two data partitions or clusters using the same sample. This index has zero expected value in the case of a random partition, and takes the value 1 in the case of perfect agreement between two clusters. Negative values of the index indicate anti-correlation between the two clusters. You can calculate it in R by the function `adjustedRandIndex` contained in the `mclust` package.

Understanding Rat Talk

So called “Rat Talk” consists of ultrasonic vocalizations (USVs) emitted by individual rats to members of their colony. It is possible that USVs convey both emotional and/or environmental information. Takahashi et al⁹² use cluster analysis to categorize the USVs of pairs of rats as they go about their daily activities.

The data set consisted of audio and video recording. The audio recording data was analyzed 50 ms at a time. Where a USVs was emitted the corresponding video sequence was also analyzed. USV calls were categorized according to Wright et al’s rat communication classification system⁹³. This consists of seven call types - upward, downward, flat, short, trill, inverted U and 22-kHz.

A two-step cluster analysis was performed. In the first step frequency and duration data were formed into preclusters. The number of clusters was automatically determined on the basis of Schwarz Bayesian Criterion, with the log-likelihood criterion used as a distance measure. In the second step a hierarchical clustering algorithm was applied to the preclusters.

Table 22 presents some of the findings. Notice that three clusters were obtained which the researchers labeled as “feeding”, “moving” and “fighting”. Feeding is associated with a low frequency USV, moving with a medium frequency and fighting with the highest frequency. Overall, the analysis tends to indicate that there is an association between USVs and rat behavior.

Cluster	Frequency	Duration	Dominant call type
1. Feeding	$24.56 \pm 2.18 \text{ kHz}$	$628.70 \pm 415.45 \text{ ms}$	22-kHz
2. Moving	$41.78 \pm 5.88 \text{ kHz}$	$31.18 \pm 32.40 \text{ ms}$	Flat
3. Fighting	$59.18 \pm 4.91 \text{ kHz}$	$9.16 \pm 10.08 \text{ ms}$	Short

Table 22: Summary of findings of Takahashi et al

Identification of Asthma Clusters

Kim et al⁹⁴ use cluster analysis to help define asthma sub-types as a prelude to searching for better asthma management. The researchers conduct their analysis using two cohorts of asthma patients. The first, known as the COREA cohort consists of 724 patients. The second, the SCH cohort consists

of 1843 patients. We focus our discussion on the COREA cohort as similar results were found for the SCH cohort.

Six primary variables, FEV₁⁹⁵, body mass index, age at onset, atopic status, smoking history and history of hospital use, were used to help characterize the asthma clusters. All measurements were standardized using z-scores for continuous variables and as 0 or 1 for categorical variables. Hierarchical cluster analysis using Ward's method was used to generate a dendrogram for estimation of the number of potential clusters. This estimate was then used in a k-means cluster analysis.

The researchers observed four clusters. The first cluster contained 81 patients and was dominated by male patients with the greatest number of smokers and a mean onset age of 46 years. The second cluster contained 151 patients and around half of the patients had atopy. The third cluster had 253 patients with the youngest mean age at onset (21 years) and about two-thirds of patients had atopy. The final group had 239 patients and the highest FEV₁ at 97.9 with mean age at onset of 48 years.

Automated Determination of the Arterial Input Function

Cerebral perfusion, also referred to as cerebral blood flow (CBF), is one of the most important parameters related to brain physiology and function. The technique of dynamic-susceptibility contrast (DSC) MRI⁹⁶ is a popular method to measure perfusion. It relies on the intravenous injection of a contrast agent and the rapid measurement of the transient signal changes during the passage of the contrast agent passes through the brain.

Central to quantification of CBF is the arterial input function (AIF), which describes the contrast agent input to the tissue of interest.

The creation of quantitative maps of cerebral blood flow, cerebral blood volume (CBV), and mean transit time (MTT) are created using a deconvolution method^{97 98}.

Yin et al⁹⁹ consider the use of two clustering techniques (k-means and fuzzy c-means (FCM)) for AIF detection. Forty-two volunteers were recruited onto the study. They underwent DSC MRI imaging. After suitable transformation of the image data both clustering techniques were applied with the number of clusters pre-set to 5.

For the mean curve of each cluster, the peak value (PV), the time to peak (TPP), and the full-width half maximum (FWHM) were computed from which a measure $M = \frac{PV}{(TPP \times FWHM)}$ was calculated. Following the approach

of Murase et al¹⁰⁰ the researchers select the cluster with the highest M to determine the AIF.

Figure 48.2 shows the AIF's for each clustering method for a 37 year old male patient. Relative to FCM, the researchers observe K-means-based AIF shows similar TTP, higher PV, and narrower FWHM. The researchers conclude by stating “...the K-means method yields more accurate and reproducible AIF results compared to FCM cluster analysis. The execution time is longer for the K-means method than for FCM, but acceptable because it leads to more robust and accurate follow-up hemodynamic maps.”

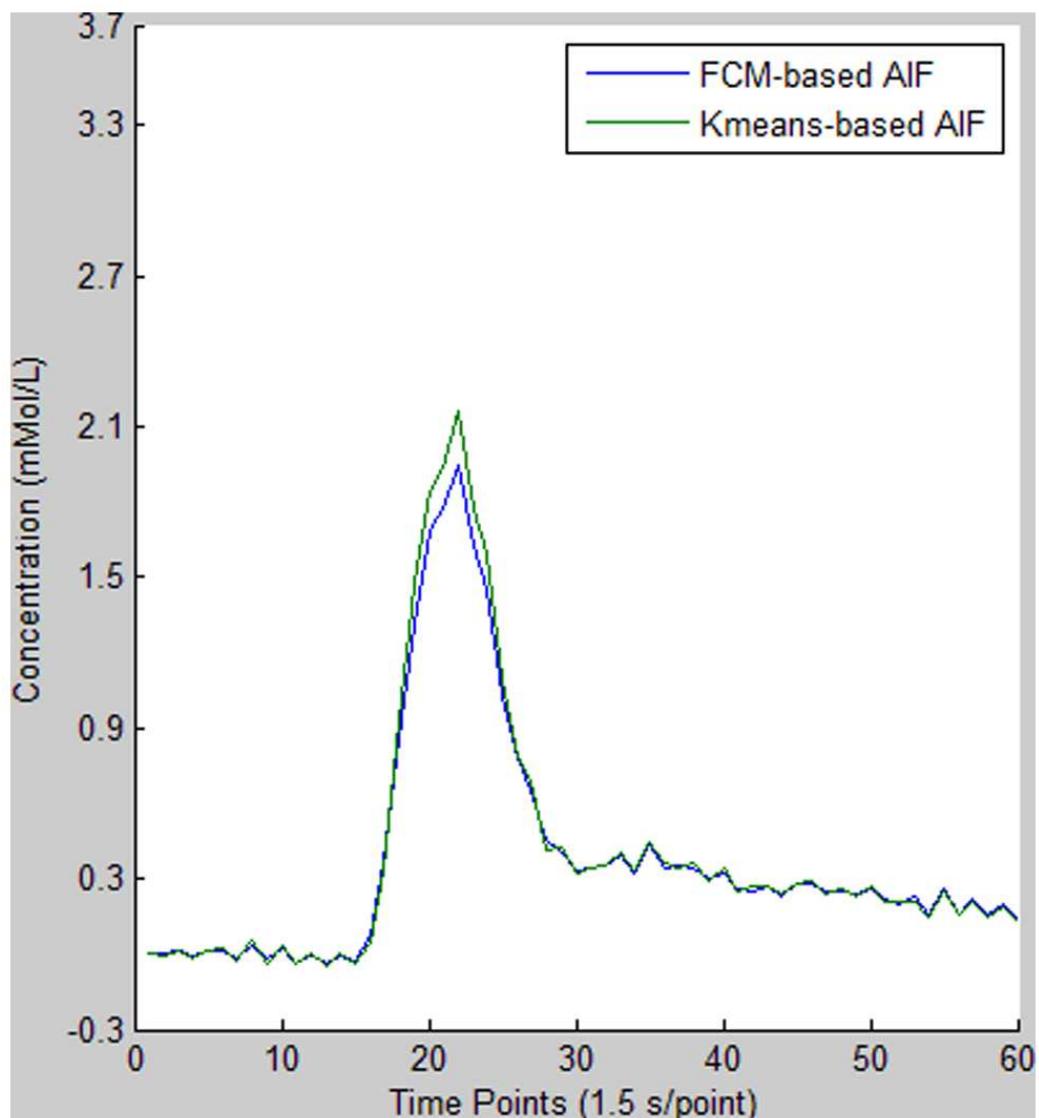


Figure 48.2: Comparison of AIFs derived from the FCM and Kmeans clustering methods. Source Yin et al. doi:10.1371/journal.pone.0085884.g007.

• PRACTITIONER TIP •

The question of how many clusters to pre-specify in methods such as k-means is a common issue. Kim et al solve this problem by using the dendrogram from hierarchical cluster analysis to generate the appropriate number of clusters. Given a sample of size N, An alternative and very crude rule of thumb is¹⁰¹:

$$k \approx \sqrt{\frac{N}{2}} \quad (48.1)$$

Choice Tests for a Wood Sample

Choice tests are frequently used by researchers to determine the food preference of insects. For insects which are sensitive to taste natural variations in the same species of wood might be sufficient to confound experimental results¹⁰². Therefore researchers have spent considerable effort attempting to reduce this natural variation¹⁰³.

Oberst et al¹⁰⁴ test the similarity of wood samples cut sequentially from different Monterey pine (*Pinus radiata*) sources, by applying fuzzy c-means clustering.

Veneer discs, from different trees/ geographical locations, were collected for two samples. The small data set consisted of 505 discs cut from 10 sheets of wood; the large data set consisted of 1417 discs cut from 22 sheets. Fuzzy-c means clustering using three physical properties (dry weight, moisture absorption and reflected light intensity) was used to evaluate both data-sets.

Six clusters were identified for each data set. For the small data set all six cluster centers were in regions of negative mode-skewness, which simply means that most of the wood veneer had more bright (early wood) than dark (late wood) regions. For the large data set only four of the six cluster centers were in regions of negative mode-skewness.

The researchers found that the difference between the small and large data set for the mode skewness of the reflected light intensity was statistically significant. This was not the case for the other two properties (dry weight and moisture absorption).

Oberst et al conclude by observing that “...the clustering algorithm was able to place the veneer discs into clusters that match their original source

veneer sheets by using just the three measurements of physical properties.”

Evaluation of Molecular Descriptors

Molecular descriptors play a fundamental role in chemistry, pharmaceutical sciences, environmental protection policy, and health research. They are believed to map molecular structures into numbers, allowing some mathematical treatment of the chemical information contained in the molecule. As Todeschini and Consonni state¹⁰⁵:

"The molecular descriptor is the final result of a logic and mathematical procedure which transforms chemical information encoded within a symbolic representation of a molecule into a useful number or the result of some standardized experiment."

Dehmer et al¹⁰⁶ evaluate 919 descriptors of 6 different categories (connectivity, edge adjacency, topological, walk path counts, information and 2D matrix based) by means of clustering. The samples for their analysis came from three data sets¹⁰⁷ MS₂₂₆₃, C₁₅ and N₈. For each of the 6 categories 24, 301, 57, 28, 40 and 469 descriptors were acquired.

In order to evaluate the descriptors seven hierarchical clustering algorithms (Ward, Single, Complete, Average, Mcquitty, Median and the Centroid) were applied to each of the three data-sets.

Dehmer et al report the cophenetic correlation coefficients for the average clustering solutions for the three data-sets as 0.84, 0.89 and 0.93. The researchers also plot the hierarchical clusters, see Figure 48.3, and they observe that “*The figure indicates that the descriptors of each categories have not been clustered correctly regarding their respective groups.*”

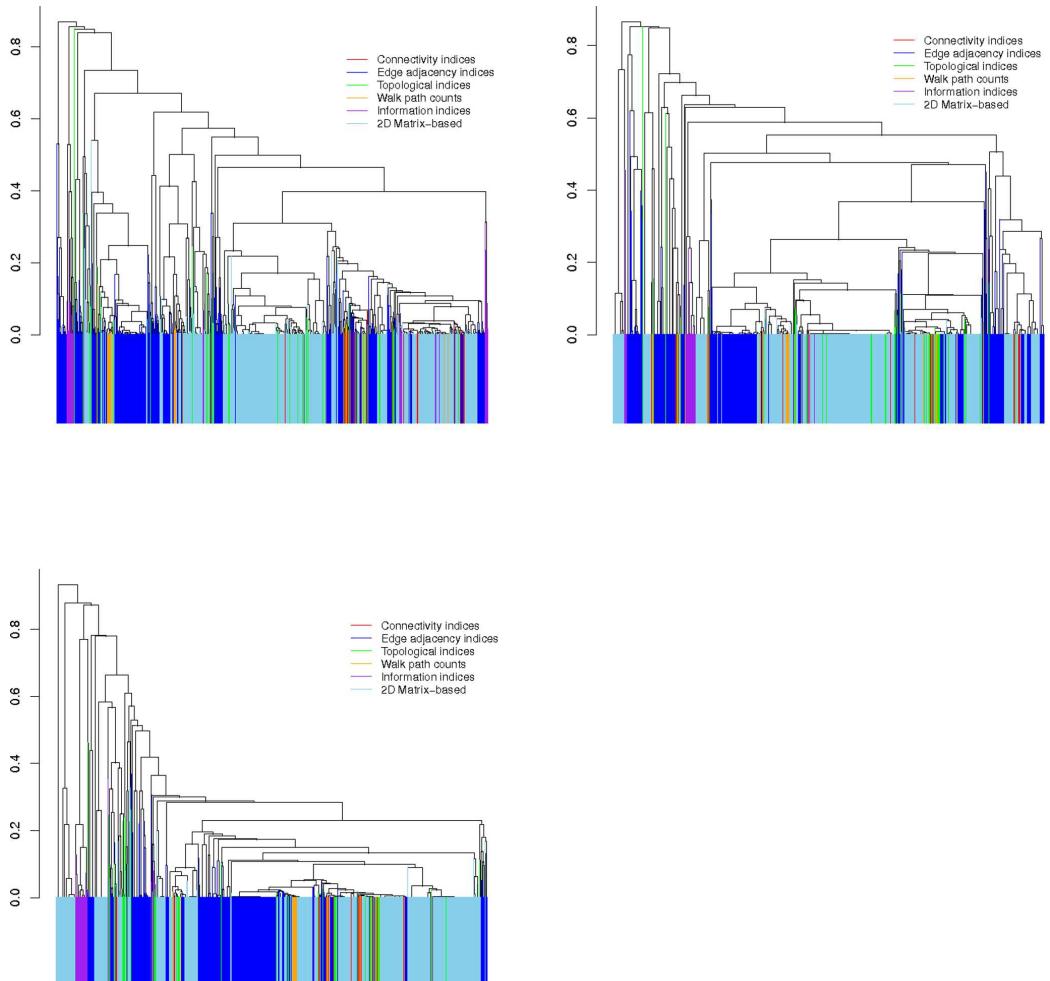


Figure 48.3: Dehme et al's hierarchical clustering using the average algorithm, MS₂₂₆₅ (left), C₁₅(middle), N₈ (right). The total number of descriptors equals 919. They belong to 6 different categories which are as follows: connectivity indices (24), edge adjacency indices (301), topological indices (57), walk path counts (28), information indices (40) and 2D Matrix-based (469). Source Dehme et al. doi:10.1371/journal.pone.0083956.g001

☛ PRACTITIONER TIP ☛

Although often viewed as graphical summary of a sample it is important to remember that dendrograms actually impose structure on the data. For example, the same matrix of pairwise distances between observations will be represented by a different dendrogram depending on the distance function (e.g., complete or average linkage) that is used.

One way to measure how faithfully a dendrogram preserves the pairwise distances between the original data points is to use the cophenetic correlation coefficient.

It is defined as the correlation between the $n(n - 1)/2$ pairwise dissimilarities between observations and the between cluster dissimilarities at which two observations are first joined together in the same cluster (often known as cophenetic dissimilarities). It takes a maximum value of 1. Higher values correspond to greater preservation of the pairwise distances between the original data points.

It can be calculated using the `cophenetic` function in the `stats` package.

Partition Based Methods

Technique 49

K-Means

For our initial analysis we will use the `kmeansruns` function in the `fpc` package. This uses the `kmeans` method in the `stats` package.

```
kmeansruns(x,krange,criterion,...)
```

Key parameters include `x` the data set of attributes for which you wish to find clusters, `krange` the suspected minimum and maximum number of clusters and `criterion` which determines the metric used to assess the clusters. .

Step 1: Load Required Packages

First we load the required packages.

```
> require(fpc)
> data("Vehicle",package="mlbench")
```

We use the `Vehicle` data frame contained in the `mlbench` package for our analysis; see page 23 for additional details on this data.

Step 2: Prepare Data & Tweak Parameters

We store the `Vehicle` data set in `x`.

```
> set.seed(98765)
> x<-Vehicle[, -19]
```

Step 3: Estimate and Assess the Model

K-means requires you specify the exact number of clusters. However, in practice you will rarely know this number precisely. One solution is to specify

a range of possible clusters and use a metric such as the Calinski-Harabasz index to choose the appropriate number.

Let us suppose we expect the number of clusters to be between 3 and 8. In this case we could call the `kmeansruns` method setting `krange` to range between 4 and 8, and the `criterion` parameter = "ch" for the Calinski-Harabasz index.

```
> no_k <- kmeansruns(x,krange=4:8,critout=TRUE,runs  
=10,criterion="ch")  
4 clusters 2151.267  
5 clusters 2528.895  
6 clusters 2571.504  
7 clusters 2375.042  
8 clusters 2305.053
```

The optimal number of clusters is the solution with the highest Calinski-Harabasz index value. In this case the Calinski-Harabasz index selects 6 clusters.

We can also use the silhouette average width as our decision criterion.

```
> no_k <- kmeansruns(x,krange=4:8,critout=TRUE,runs  
=10,criterion="asw")  
4 clusters 0.4423919  
5 clusters 0.4716047  
6 clusters 0.4420139  
7 clusters 0.4483106  
8 clusters 0.3405826
```

The widest width occurs are 5 clusters. This is a smaller number than obtained by the Calinski-Harabasz index, however using both methods we have narrowed down the range of possible clusters to lie between 5 to 6. Visualization often helps in making the final selection. To do this let's build a sum of squared error (SSE) scree plot. I explain line by line below.

```
> wgs <- (nrow(x)-1)*sum(apply(x,2,var))  
  
> for (i in 2:8)  
{  
  fit_temp<-kmeans(as.matrix(x), centers=i)  
  wgs[i]<-sum(fit_temp$withinss)  
}
```

```
> plot(1:8, wgs, type="b", xlab="Number of  
Clusters", ylab="Within groups sum of squares")
```

Our first step is to assign the first observation in `wgs` to a large number. This is followed by a loop which calls `kmeans` in the `stats` package for $k = 2$ to 8 clusters. This is followed by using the `plot` method to create a scree plot, shown in Figure 49.1. Typically in a scree plot we look for a sharp elbow to indicate the appropriate number of clusters. In this case the elbow is gentle, but 4 clusters appears to be a good number to try.

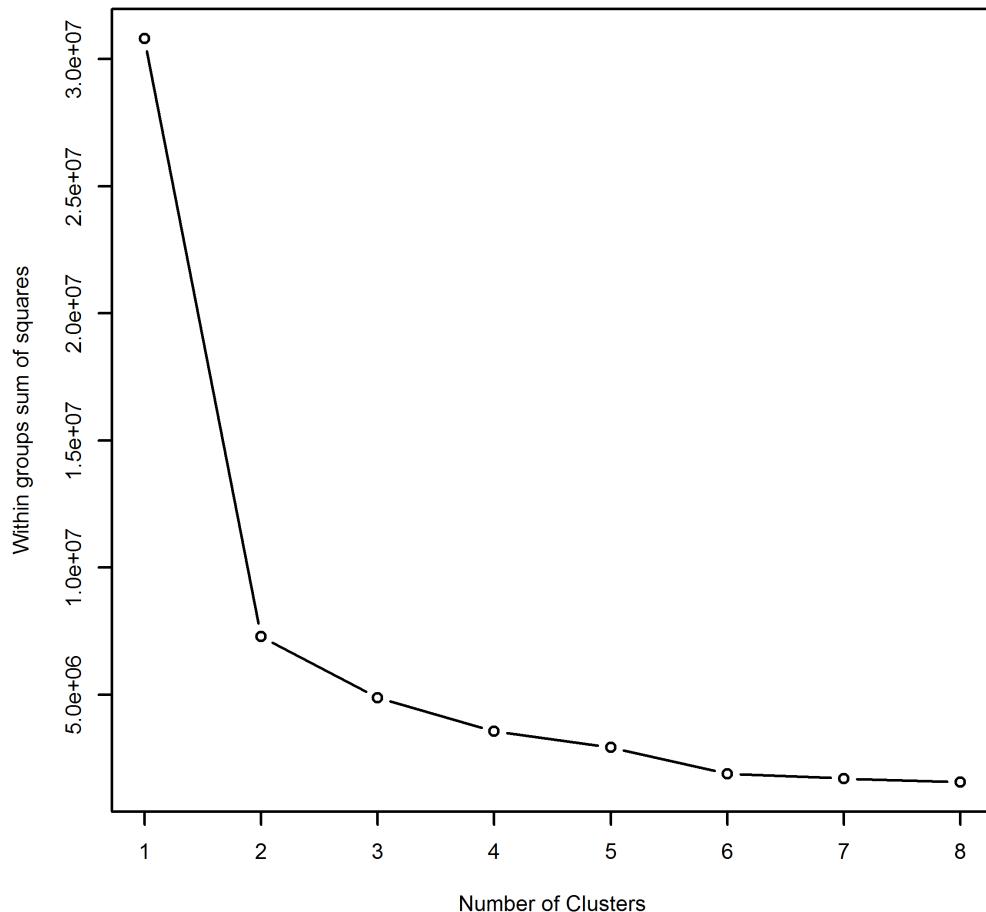


Figure 49.1: k-means clustering scree plot using `Vehicle`

Let's fit the model with 4 clusters using the `kmeans` function in the `stats` package and plotting the results in Figure 49.2.

```
> fit<-kmeans(x, 4, algorithm = "Hartigan-Wong")
> plot(x, col = fit$cluster)
```



Figure 49.2: k-means pairwise plot of clusters using Vehicle

We can also narrow down our visualization to a single pair. To illustrate this let's look at the relationship between **Comp** (column 1 in **x**) and **Scat.Ra** (column 7 in **x**). The plot is shown in Figure 49.3.

```
> plot(x[,c(1,7)], col = fit$cluster)
> points(fit$centers[,c(1,7)], col = 1:4, pch = 8,
  cex = 2)
```

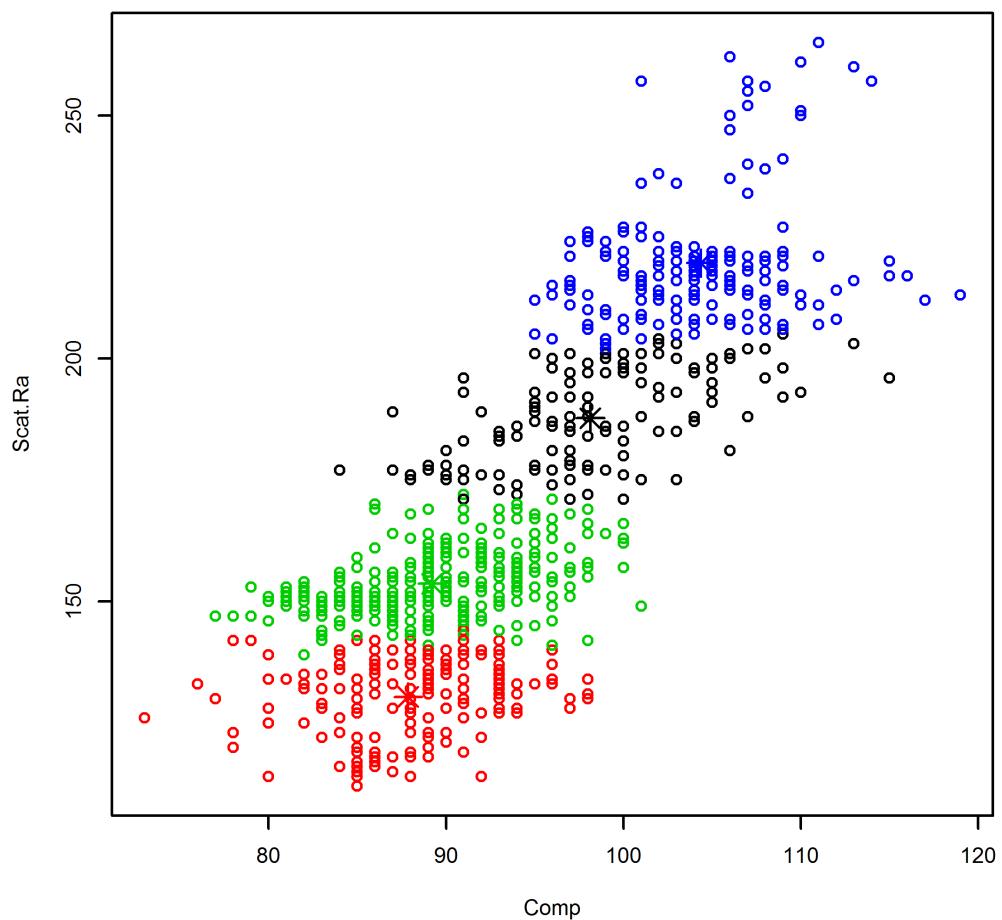


Figure 49.3: k-means pairwise cluster plot of Comp and Scat.Ra

Technique 50

Clara Algorithm

For our analysis we will use the `clara` function in the `cluster` package.

```
clara(x,k)
```

Key parameters include `x` the dissimilarity matrix, and `k` the number of clusters.

Step 1: Load Required Packages

First we load the required packages.

```
> require (cluster)
> require(fpc)
> data("thyroid", package="mclust")
```

We use the `thyroid` data frame contained in the `mclust` package for our analysis; see page 385 for additional details on this data. The `fpc` package will be used to help select the optimum number of clusters.

Step 2: Prepare Data & Tweak Parameters

The thyroid data is stored in `x`. We also drop the class labels stored in the variable `Diagnosis`. Finally, we use the `daisy` method to create a dissimilarity matrix.

```
> set.seed(1432)
> x<-thyroid
> x$Diagnosis  <- NULL
> dissim<-daisy(x)
```

Step 3: Estimate and Assess the Model

To use of the Clara algorithm you have to specify the exact number of clusters in your sample. However, in practice you will rarely know this number precisely. One solution is to specify a range of possible clusters and use a metric such as the average silhouette width to choose the appropriate number.

Let us suppose we expect the number of clusters to be between 1 and 6. In this case we could call the `pamk` method setting `krange` to lie between 1 and 6, and the `criterion` parameter = "asw" for the average silhouette width. Here is how to do that.

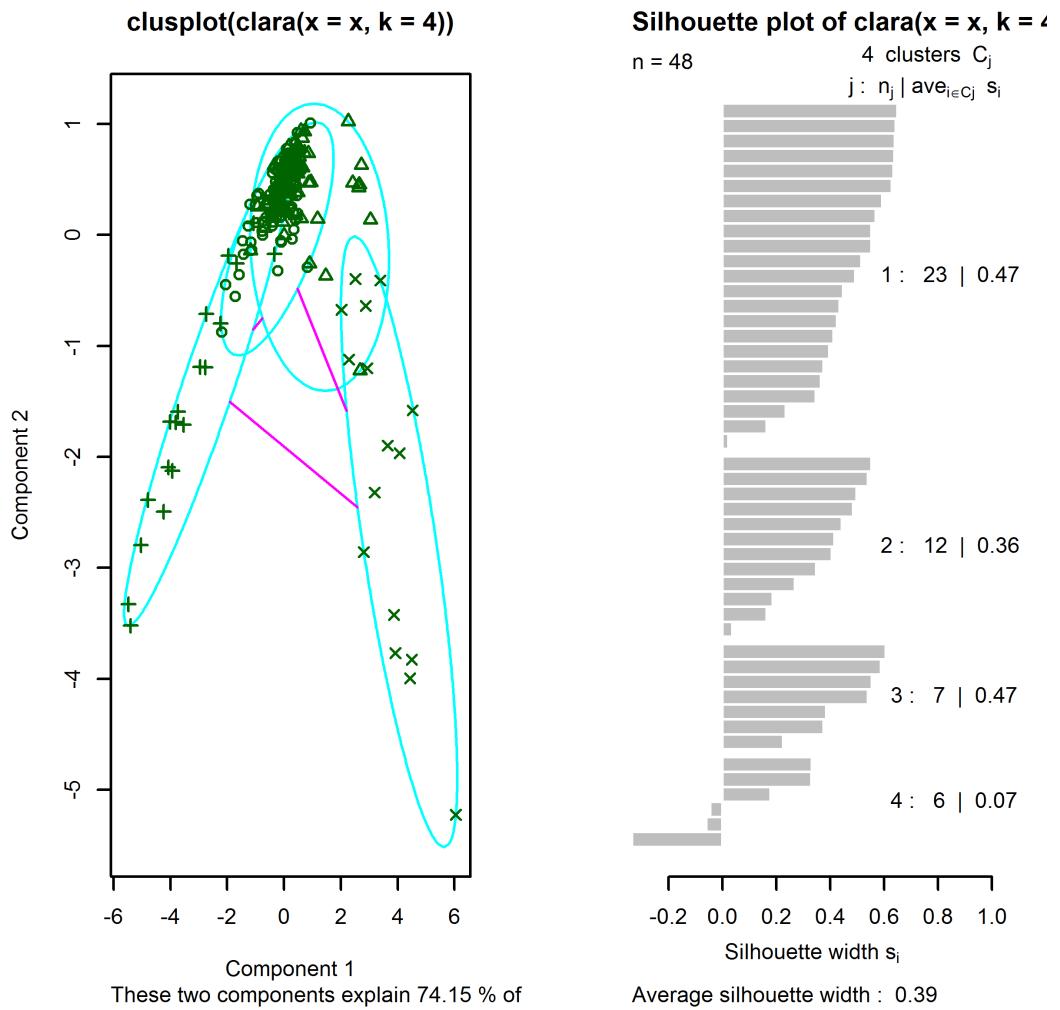
```
> pk1 <- pamk(dissim, krange=1:6,  
+ criterion="asw",  
+ critout=TRUE,  
+ usepam=FALSE)  
  
1 clusters 0  
2 clusters 0.5172889  
3 clusters 0.4031142  
4 clusters 0.4325106  
5 clusters 0.4845567  
6 clusters 0.4251244
```

The optimal number of clusters is the solution with the largest average shillotte width. So in this example 2 clusters has largest width.

However, we fit the model with 4 clusters and use the `plot` method to visualize the result.

```
> fit<-clara(x,k=4)  
  
> par( mfrrow = c(1, 2))  
> plot(fit)
```

Figure 50.1 shows the resultant plots.

Figure 50.1: Plot using the clara algorithm with $k = 4$ and data set thyroid

Technique 51

PAM Algorithm

For our analysis we will use the `pam` function in the `cluster` package.

```
pam(x, k)
```

Key parameters include `x` the dissimilarity matrix, and `k` is the number of clusters.

Step 1: Load Required Packages

First we load the required packages.

```
require(cluster)
require(fpc)
data("wine", package = "ordinal")
```

We use the `wine` data frame contained in the `ordinal` package for our analysis; see page 95 for additional details on this data.

Step 2: Prepare Data & Tweak Parameters

Since the `wine` data set contains ordinal variables we use the `daisy` method with `metric` set to `"gower"` to create a dissimilarity matrix. The gower metric can handle nominal, ordinal, and binary data¹⁰⁸. The `wine` sample is stored in `data`. We also drop the wine rating column and then pass data to the `daisy` method storing the result in `x`.

```
> set.seed(1432)
> data<-wine
> data<-data[, -2]
> x<-daisy(data, metric = "gower")
```

Step 3: Estimate and Assess the Model

The PAM algorithm requires you to specify the exact number of clusters in your sample. However, in practice you will rarely know this number precisely. One solution is to specify a range of possible clusters and use a metric such as the average silhouette width to choose the appropriate number. Let us suppose we expect the number of clusters to be between 1 and 5. In this case we could call the `pamk` method setting `krange` to lie between 1 and 5, and the criterion parameter = "asw" for the average silhouette width. Here is how to do that.

```
> pk1 <- pamk(x, krange=1:5,  
+ criterion="asw",  
+ critout=TRUE,  
+ usepam=TRUE)  
  
1 clusters 0  
2 clusters 0.2061618  
3 clusters 0.2546821  
4 clusters 0.4053708  
5 clusters 0.3318729
```

The optimal number of clusters is the solution with the largest average silhouette width. So in this example the largest width occurs at 4 clusters.

Now we fit the model with 4 clusters and use the `clusplot` method to visualize the result. Figure 51.1 shows the resultant plot.

```
> fit<-pam(x,4)  
> clusplot(fit)
```

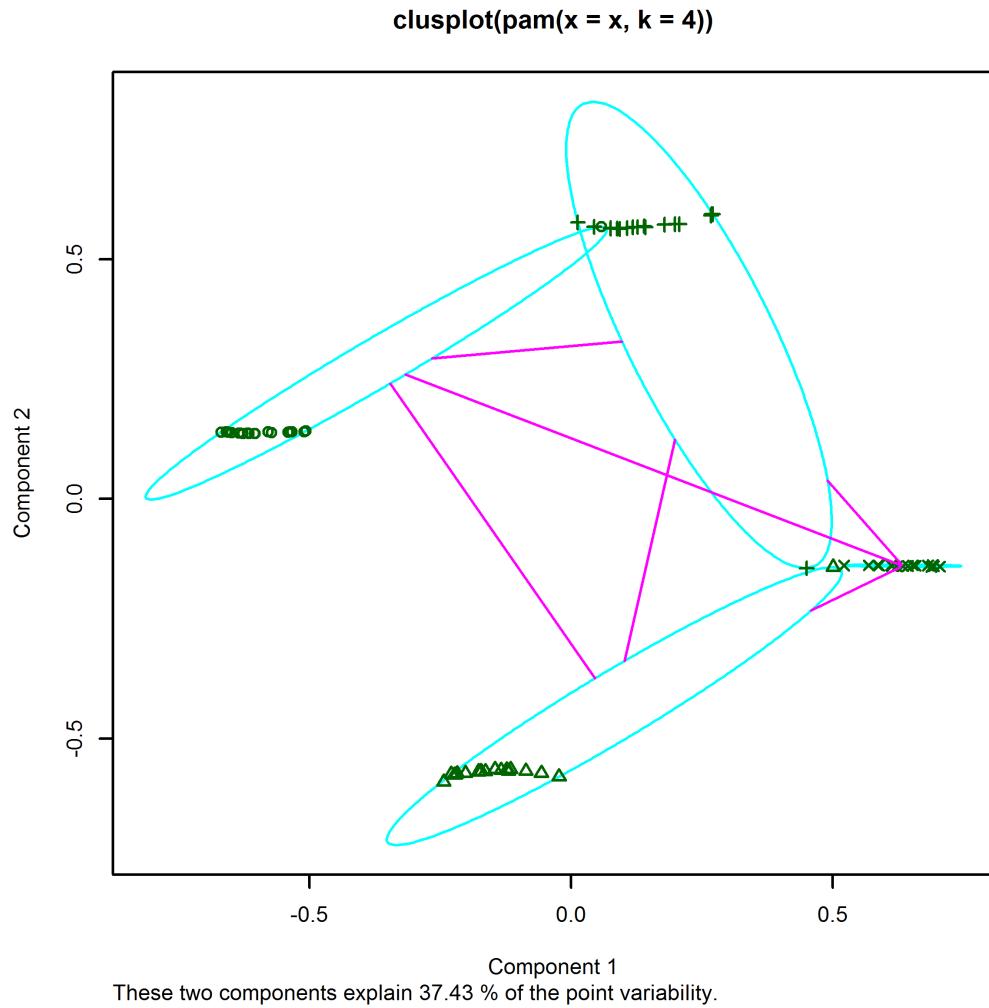


Figure 51.1: Partitioning around medoids using `pam` with $k = 4$ for `wine`

Technique 52

Kernel Weighted K-Means

The kernel weighted version of the k-means algorithm projects the sample data into a non-linear feature space by use of a kernel, see Part II. It has the benefit that it can identify clusters which are not linearly separable in the input space. For our analysis we will use the `kkmeans` function in the `kernlab` package.

```
kkmeans(x, centers, ...)
```

Key parameters include `x` the matrix of data to be clustered, and `centers` the number of clusters.

Step 1: Load Required Packages

First we load the required packages.

```
> require(kernlab)
> data("Vehicle", package = "mlbench")
```

We use the `Vehicle` data frame contained in the `mlbench` package for our analysis; see page 95 for additional details on this data.

Step 2: Prepare Data & Tweak Parameters

We drop the vehicle class column and store the result in `x`.

```
> set.seed(98765)
> x <- Vehicle[, -19]
```

Step 3: Estimate and Assess the Model

We estimate the model using four clusters (`centers = 4`), storing the result in `fit`. The plot method is then used to visualize the result, see Figure 52.1.

```
> fit <- kkmeans(as.matrix(x), centers=4)
> plot(x,col=fit)
```

Let's focus in on the pairwise clusters associated with `Comp` (column 1 in `x`) and `Scat.Ra` (column 7 in `x`). We can visualize this relationship using the `plot` method. Figure 52.2 shows the resultant plots.

```
> plot(x[,c(1,7)], col = fit)
> points(centers(fit)[,c(1,7)], col = 1:4, pch = 8,
  cex = 2)
```

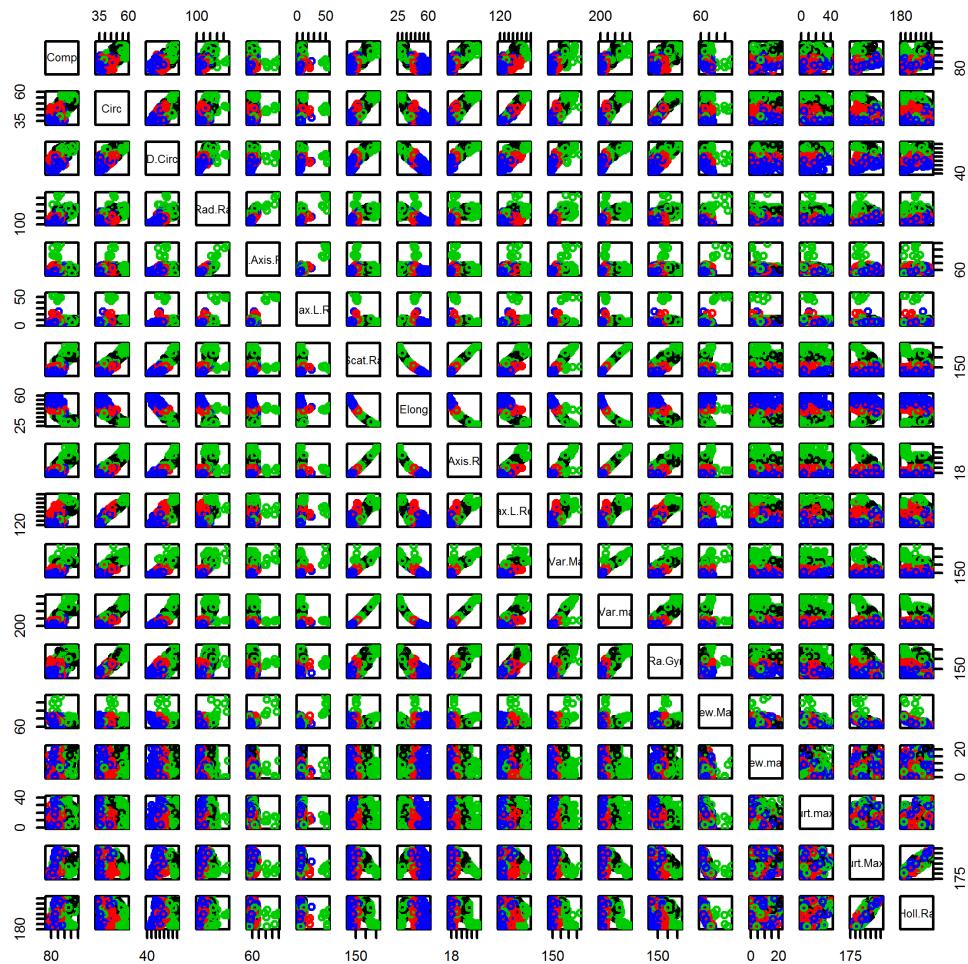


Figure 52.1: Kernel Weighted K-Means with centers = 4 using Vehicle

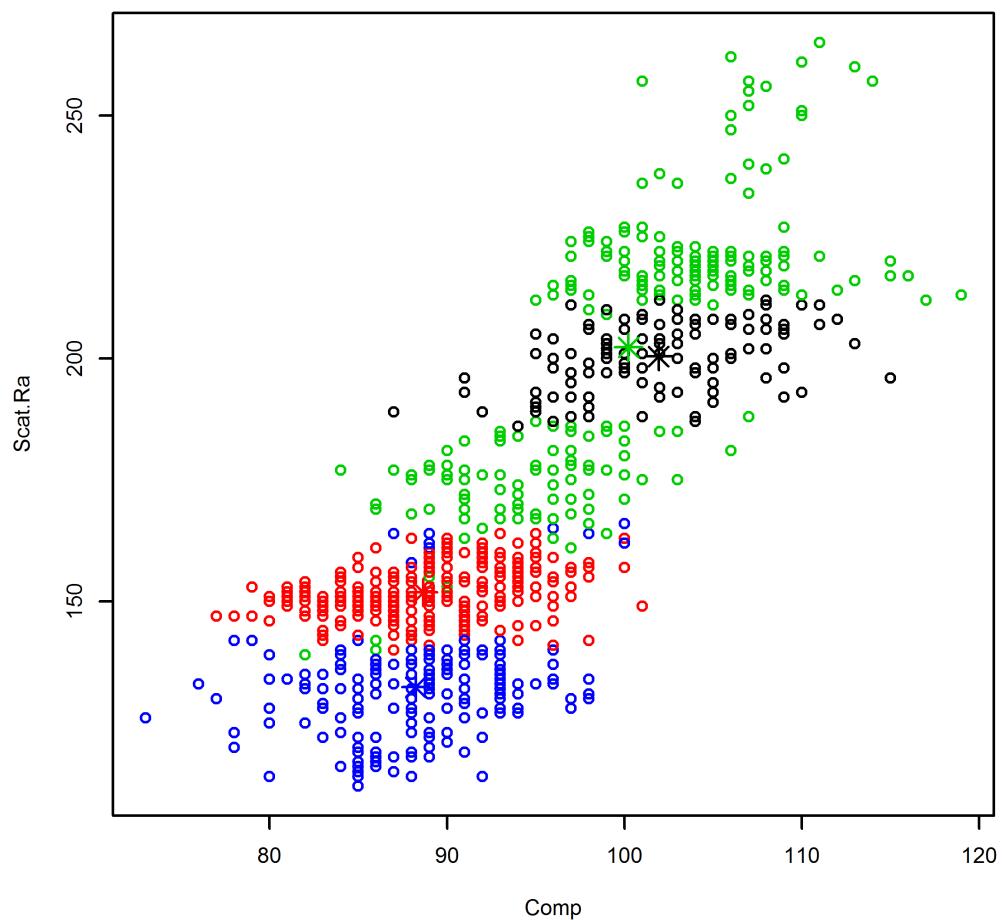


Figure 52.2: Kernel Weighted K-Means pairwise clusters of Comp and Scat.Ra using Vehicle

Hierarchy Based Methods

Technique 53

Hierarchical Agglomerative Cluster Analysis

Hierarchical Cluster Analysis is available with the basic installation of R. It is obtained using the `stats` package with the `hclust` function:

```
hclust(d, method,...)
```

Key parameters include `d` the dissimilarity matrix and `method` the agglomeration method to be used.

Step 1: Load Required Packages

First we load the required packages. I'll explain each below.

```
> library(colorspace)
> library(dendextend)
> require(qgraph)
> require(cluster)
> data("thyroid", package="mclust")
```

We will use the `thyroid` data frame contained in the `mclust` package in our analysis. For color output we use the `colorspace` package. The `dendextend` and `qgraph` packages will help us better visualize our results. We will also use the `bannerplot` method from the `cluster` package.

NOTE... ↗

The thyroid data frame¹⁰⁹ was constructed from five laboratory tests administered to a sample of 215 patients. The tests were used to predict whether a patient's thyroid function could be classified as euthyroidism (normal), hypothyroidism (under active thyroid) or hyperthyroidism. The thyroid data frame contains the following variables:

- **Class variable:**

- Diagnosis: Diagnosis of thyroid operation: Hypo, Normal, and Hyper.
 - * Distribution of Diagnosis (number of instances per class)
 - * Class 1: (normal) 150
 - * Class 2: (hyper) 35
 - * Class 3: (hypo) 30

- **Continuous attributes:**

- RT3U: T3-resin uptake test (percentage).
- T4: Total Serum thyroxin as measured by the isotopic displacement method.
- T3: Total serum triiodothyronine as measured by radioimmuno assay.
- TSH: Basal thyroid-stimulating hormone (TSH) as measured by radioimmuno assay.
- DTSH: Maximal absolute difference of TSH value after injection of 200 micro grams of thyrotropin-releasing hormone as compared to the basal value.

Step 2: Prepare Data & Tweak Parameters

We store the thyroid data set in `thyroid2` and remove the class variable (`Diagnosis`). The variable `diagnosis_col` will be used to color the dendograms using `rainbow_hcl`.

```
> thyroid2<-thyroid[,-1]
> diagnosis_labels <- thyroid[,1]

> diagnosis_col <- rev(rainbow_hcl(3))
[as.numeric(diagnosis_labels)]
```

Step 3: Estimate and Assess the Model

Our first step is to create a dissimilarity matrix. We use the `manhattan` method storing the results in `dthyroid`.

```
> dthyroid<-dist(thyroid2,method="manhattan")
```

Now are ready to fit our basic model. I have had some success with ward's method so let's try it first.

```
> fit <- hclust(dthyroid,method="ward.D2")
```

NOTE... ↗

I have had good practical success using ward's method. However, it is interesting to notice that in the academic literature there are two different algorithms associated with the method! The method we selected ("ward.D2") squares the dissimilarities before clustering. An alternative ward method "ward.D" does not¹¹⁰.

The print method provides a useful overview of the fitted model.

```
> print(fit)

Call:
hclust(d = dthyroid, method = "ward.D2")

Cluster method : ward.D2
Distance       : manhattan
Number of objects: 215
```

Now we create a banneplot of the fitted model.

```
> bannerplot(fit,main="Euclidean")
```

The result is shown in Figure 53.1.

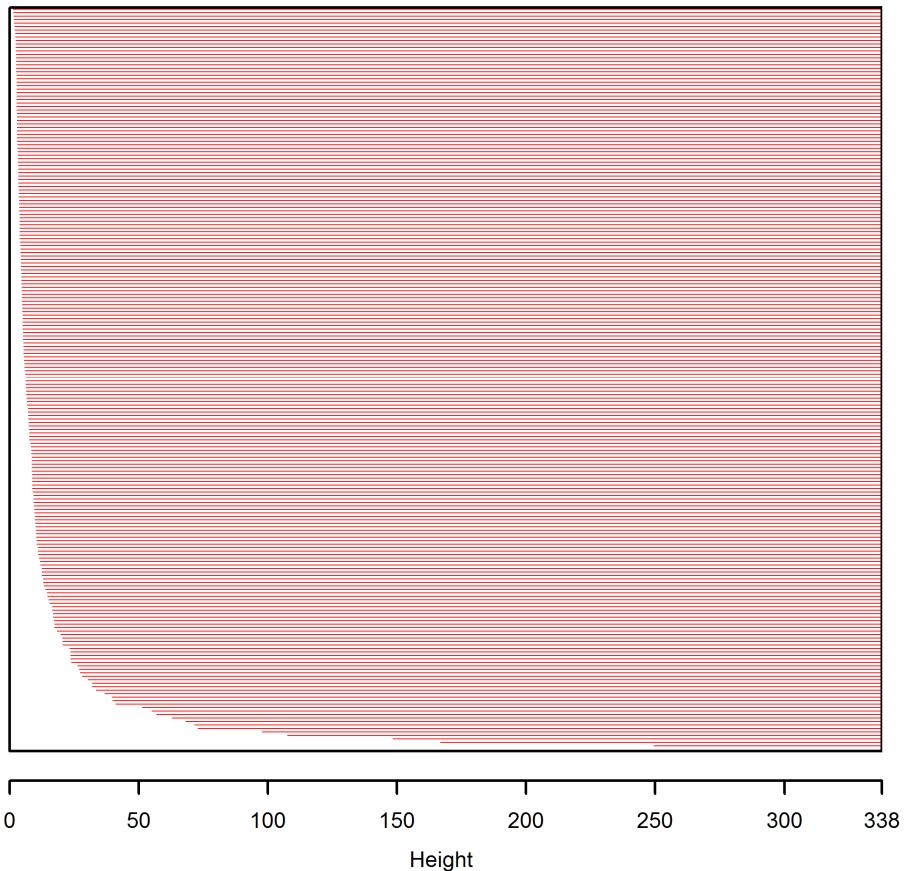


Figure 53.1: Hierarchical Agglomerative Cluster Analysis bannerplot using `thyroid` and Wards method

OK, we should take a look at the fitted model. It will be in the form of a dendrogram. First we set `diagnosis` to capture the classes. Then we save `fit` as a dendrogram into `dend`, followed by ordering of the observations somewhat using the `rotate` method.

```
> diagnosis<- rev(levels(diagnosis_labels))
> dend <- as.dendrogram(fit)
> dend <- rotate(dend, 1:215)
```

We set `k=3` in `color_branches` for each of the three diagnosis types (hyper, normal, hypo).

```
> dend <- color_branches(dend, k=3)
```

Next, we match the labels, to the actual classes.

```
> labels_colors(dend) <-  
rainbow_hcl(3)[sort_levels_values(  
as.numeric(diagnosis_labels)  
[order.dendrogram(dend)]  
)]
```

We still need to add the diagnosis classes to the labels. This is achieved as follows.

```
> labels(dend) <- paste(as.character(diagnosis_  
labels)  
[order.dendrogram(dend)],  
"(",labels(dend),")",  
sep = "")
```

A little administration is required in our next step; reduce the size of the labels to 75% of their original size. Then we create the actual plot shown in Figure 53.2.

```
> dend <- set(dend, "labels_cex", 0.75)  
> par(mar = c(3,3,3,7))  
> plot(dend,  
       main = "Clustered Thyroid data where the  
       labels give the true diagnosis",  
       horiz = TRUE, nodePar = list(cex = .007))  
> legend("topleft", legend = diagnosis, fill =  
rainbow_hcl(3))
```

Overall the data seems to have been well separated. However, there is some mixing between **normal** and **hypo**.

Clustered Thyroid data where the labels give the true diagnosis

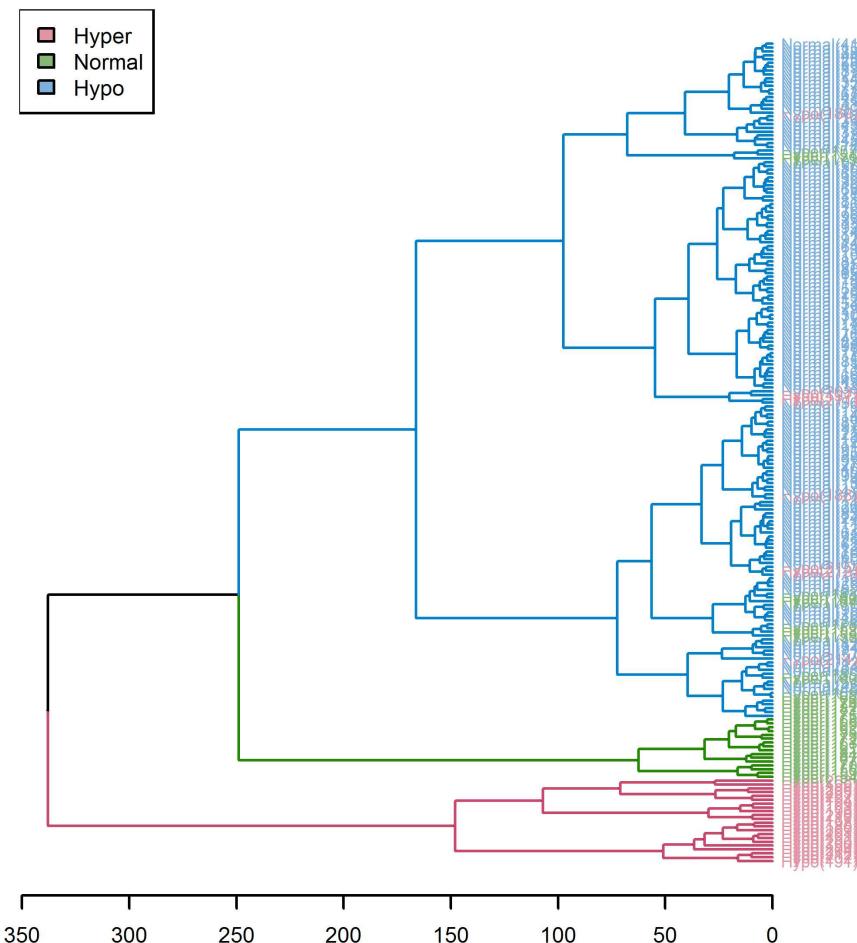


Figure 53.2: Hierarchical Agglomerative Cluster Analysis dendrogram using thyroid

Since our initial analysis used ward's method we may as well take a look at the other methods. We choose six of the most popular methods and investigate their correlation. We begin by capturing the methods in `hclust_methods` and creating a list in `hclust_list`.

```
> hclust_methods=c("ward.D",
+ "ward.D2", "single",
+ "complete", "average",
+ "mcquitty", "median", "centroid")

> hclust_list<-dendlist()
```

Here is the main loop. Notice we fit the model's using `temp_fit` with the dissimilarity matrix `dthyroid`.

```
> for(i in seq_along(hclust_methods))
{
  print(hclust_methods[i])
  temp_fit<-hclust(dthyroid,method=hclust_methods
  [i])

  hclust_list <- dendlist(hclust_list , as.
    dendrogram(temp_fit))
}
```

Let's add names to the list.

```
> names(hclust_list) <- hclust_methods
```

Next we take a look at the output. Notice the wide variation in height produced by the different methods.

```
> hclust_list
$ward.D
'dendrogram' with 2 branches and 215 members total,
 at height 1233.947

$ward.D2
'dendrogram' with 2 branches and 215 members total,
 at height 337.8454

$single
'dendrogram' with 2 branches and 215 members total,
 at height 33.5

$complete
'dendrogram' with 2 branches and 215 members total,
 at height 160.1

$average
'dendrogram' with 2 branches and 215 members total,
 at height 84.66573

$mcquitty
'dendrogram' with 2 branches and 215 members total,
 at height 103.8406
```

```
$median
'dendrogram' with 2 branches and 215 members total,
 at height 41.14692

$centroid
'dendrogram' with 2 branches and 215 members total,
 at height 63.73702
```

Now we investigate the correlation between methods. Note that `method="common"` measures the commonality between members of nodes.

```
> cor<-cor.dendlist(hclust_list,method="common")
> par( mfrow = c(2, 2))
> qgraph(cor,minimum=0.70,title= "Correlation = 0.70")
> qgraph(cor,minimum=0.75,title= "Correlation = 0.75")
> qgraph(cor,minimum=0.80,title= "Correlation = 0.80")
> qgraph(cor,minimum=0.85,title= "Correlation = 0.85")
```

Figure 53.3 shows the resultant plot for varying levels of correlation. It gives us another perspective on the clustering algorithms. We can see that most methods have around 75% commonality within nodes with one another.

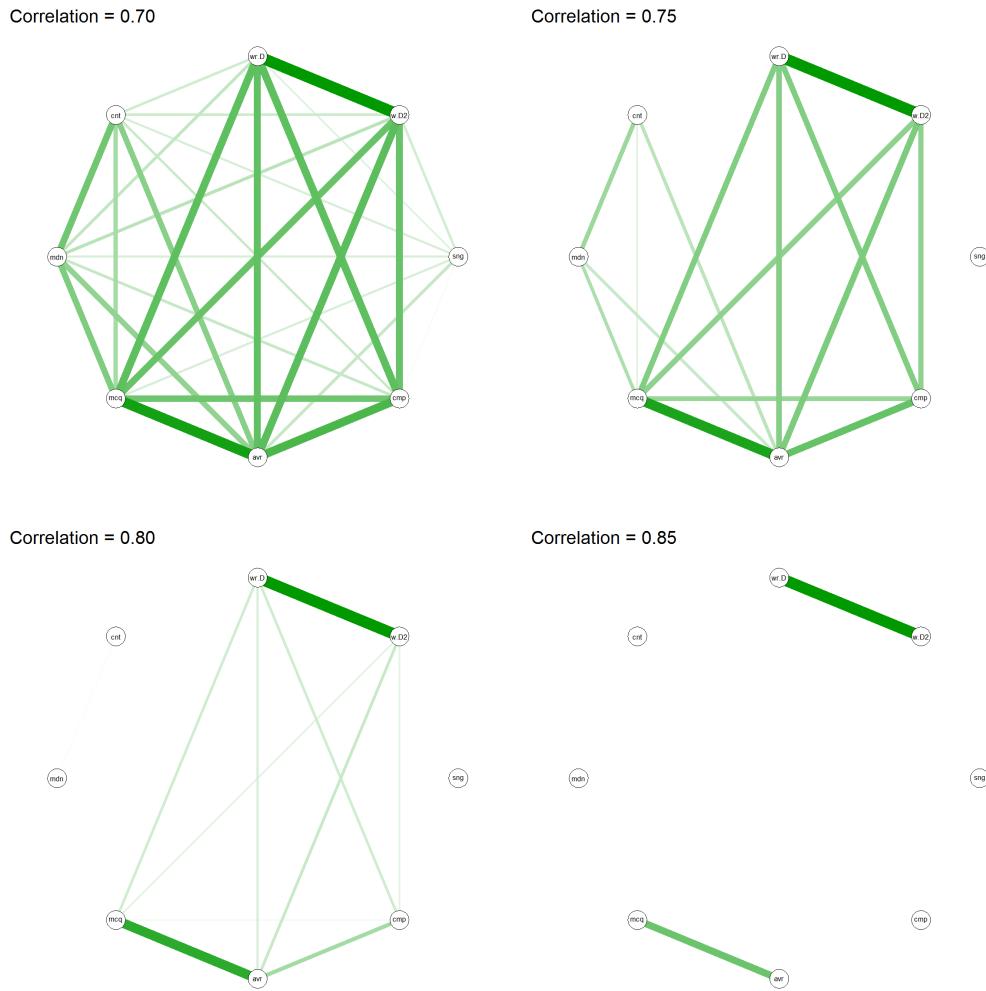


Figure 53.3: Hierarchical Agglomerative Cluster Analysis correlation between methods using thyroid

Since `ward.D` and `ward.D2` have high commonality let's look in detail using a tanglegram visualization. The "which" parameter allows us to pick the elements in the list to compare. Figure 53.4 shows the result.

```
> hclust_list %>% dendlist(which = c(1,2)) %>%
  ladderize %>%
  set("branches_k_color", k=3) %>%
  tanglegram(faster = TRUE)
```

We can use a similar approach for `average` and `mcquitty`, see Figure 53.5.

```
> hclust_list %>% dendlist(which = c(5,6)) %>%
```

```
ladderize %>%
  set("branches_k_color", k=3) %>%
  tanglegram(faster = TRUE)
```

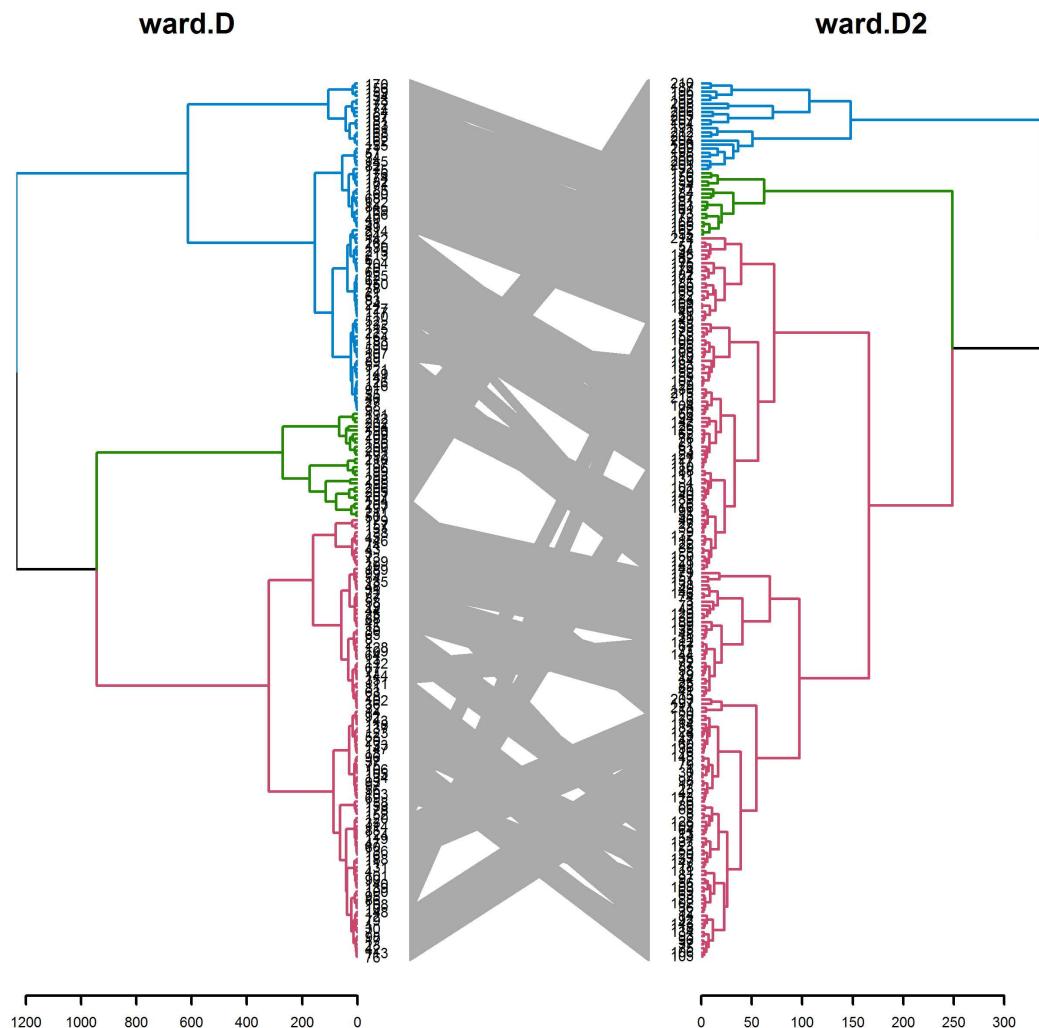


Figure 53.4: Tanglegram visualization between `ward.D` and `ward.D2` using `thyroid`

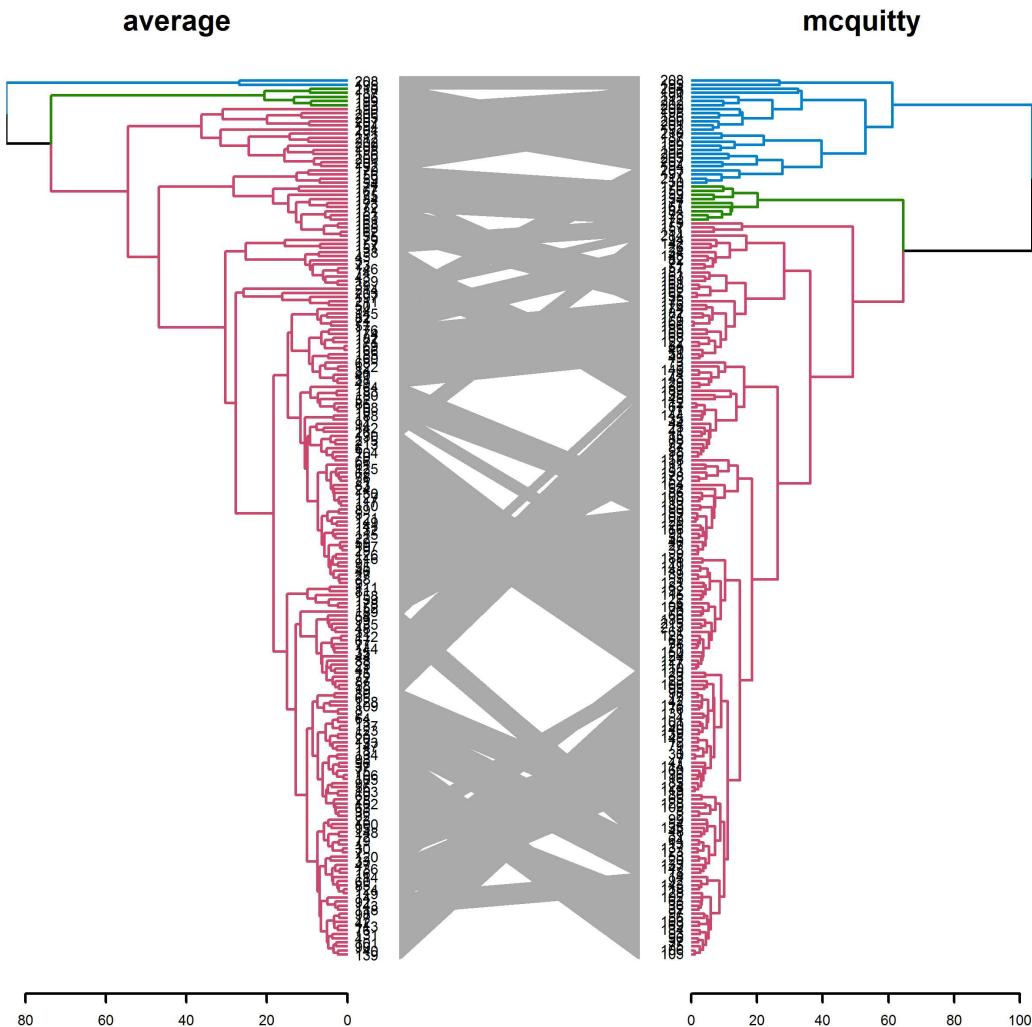


Figure 53.5: Tanglegram visualization between `average` and `mcquitty` using `thyroid`

Finally we plot the dendograms for all of the methods. This is shown in Figure 53.6.

```
> par( mfrow = c(4, 2))

> for(i in seq_along(hclust_methods))
{
  fit <- hclust(dthyroid, method=hclust_methods[i])

dend <- as.dendrogram(fit)
```

```
dend <- rotate(dend, 1:215)
dend <- color_branches(dend, k=3)

labels_colors(dend) <-
rainbow_hcl(3)[sort_levels_values(
    as.numeric(diagnosis_labels)[order.dendrogram(
        dend)]]

labels(dend) <- paste(as.character(diagnosis_labels),
[order.dendrogram(dend)],
      "(",labels(dend),")",
      sep = "")

dend <- set(dend, "labels_cex", 0.75)

par(mar = c(3,3,3,7))
plot(dend,
      main = hclust_methods[i],
      horiz = TRUE, nodePar = list(cex = .007))
legend("bottomleft", legend = diagnosis, fill =
      rainbow_hcl(3))
}
```

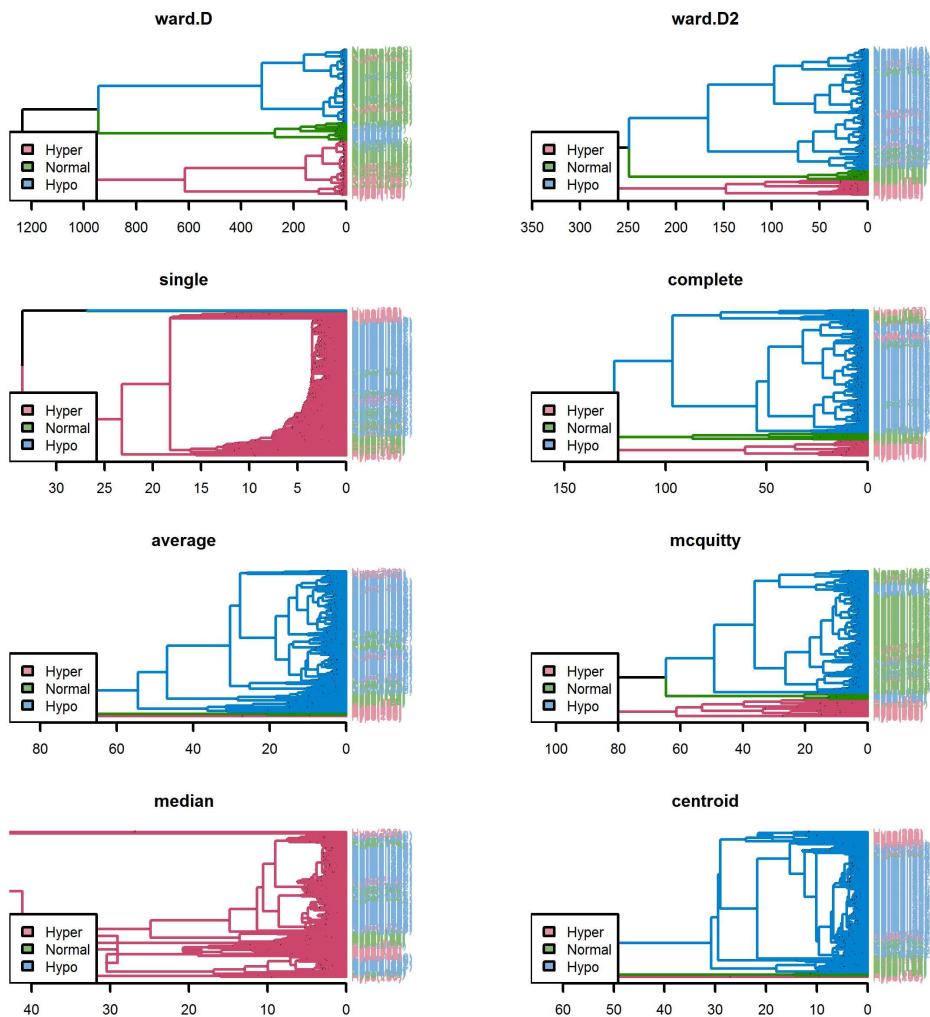


Figure 53.6: Hierarchical Agglomerative Cluster Analysis for all methods using `thyroid`

Technique 54

Agglomerative Nesting

Agglomerative nesting is available in the `cluster` package using the function `agnes`:

```
agnes(x, metric, method, ...)
```

Key parameters include `x` the data set of attributes for which you wish to find clusters, `metric` the metric used for calculating dissimilarities whilst `method` defines the clustering method to be used.

Step 1: Load Required Packages

First we load the required packages. I'll explain each below.

```
> require(cluster)
> library(colorspace)
> library(dendextend)
> require(corrplot)
> data("thyroid", package="mclust")
```

The package `cluster` contains the `agnes` function. For color output we use the `colorspace` package and `dendextend` to create fancy dendograms. The `corrplot` package is used to visualize correlations. Finally, we use the `thyroid` data frame contained in the `mclust` package for our analysis.

We also make use of two user defined functions. The first we call `panel.hist` which we will use to plot histograms. It is defined as follows.

```
> panel.hist <- function(x, ...)
{
  usr <- par("usr"); on.exit(par(usr))
  par(usr = c(usr[1:2], 0, 1.5) )
```

```
h <- hist(x, plot = FALSE)
breaks <- h$breaks; nB <- length(breaks)
y <- h$counts; y <- y/max(y)
rect(breaks[-nB], 0,
      breaks[-1], y,
      density = 50,
      border = "blue")
}
```

The second function will be used for calculating the Pearson correlation coefficient. Here is the R code:

```
> panel.cor <- function(x, y, digits = 3, prefix = "
  ", cex.cor, ...)
{
  usr <- par("usr"); on.exit(par(usr))
  par(usr = c(0, 1, 0, 1))
  r <- (cor(x, y,
method = "pearson"))

  txt <- format(c(r, 0.123456789),
digits = digits)[1]

  txt <- paste0(prefix, txt)

  if(missing(cex.cor)) cex.cor <- 0.8/strwidth(txt)
  text(0.5, 0.5, txt, cex = cex.cor)
}

}
```

Step 2: Prepare Data & Tweak Parameters

We store the thyroid data set in `thyroid2` and remove the class variable (`Diagnosis`). The variable `diagnosis_col` will be used to color the dendograms using `rainbow_hcl`.

```
> thyroid2<-thyroid[,-1]
> diagnosis_labels <- thyroid[,1]
> diagnosis_col <- rev(rainbow_hcl(3))[as.numeric(
  diagnosis_labels)]
```

Let's spend a few moments investigating the data set. We will use visualization to assist us. We begin by looking at the pairwise relationships

between the attributes, as shown in Figure 54.1. This plot was built using the pairs method as follows.

```
> par(oma = c(4, 1, 1, 1))

> pairs(thyroid2, col = diagnosis_col,
  upper.panel = panel.cor,
  cex.labels=2,
  pch=19,
  cex = 1.2,
  panel = panel.smooth,
  diag.panel = panel.hist)

> par(fig = c(0, 1, 0, 1),
  oma = c(0, 0, 0, 0),
  mar = c(0, 0, 0, 0),
  new = TRUE)

> plot(0, 0, type = "n",
  bty = "n", xaxt = "n", yaxt = "n")

> legend("bottom", cex = 1, horiz = TRUE,
  inset = c(1,0), bty = "n",
  xpd = TRUE,
  legend = as.character(levels(diagnosis_labels)),
  fill = unique(diagnosis_col))

> par(xpd = NA)
```

The top panels (above the diagonal) give the correlation coefficient between the attributes. We see that RT3U is negatively correlated with T4 and T3. It is moderately correlated with TSH and DTSH. We also see that T3 and T4 are negatively correlated with TSH and DTSH. Whilst TSH and DTSH are positively correlated.

The diagonal in Figure 54.1 shows the distribution of each attribute; and bottom panels are colored by diagnosis using `diagnosis_col`. Notice that `Hypo`, `Normal` and `Hyper` appear to be distinctly different each other (as measured by the majority of attributes). However, all three groups diagnosis types cannot be easily separated if measured by RT3U and DTSH alone.

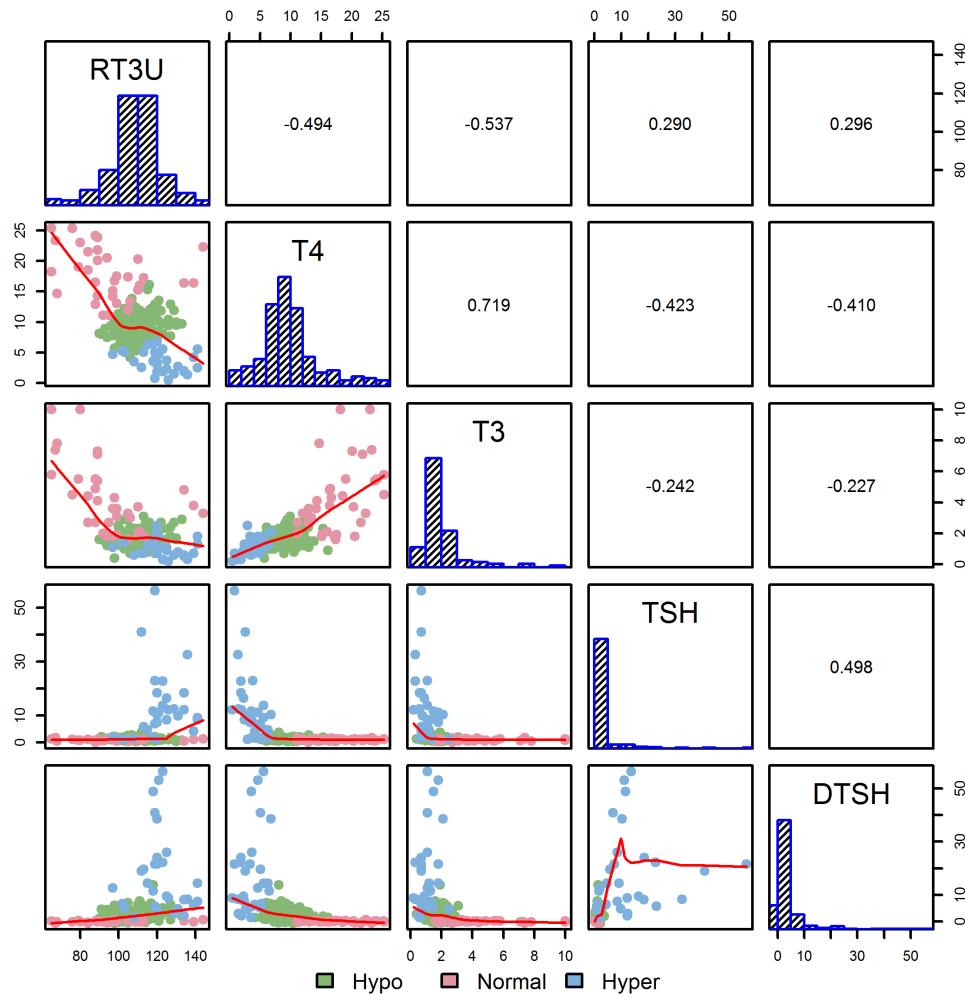


Figure 54.1: Pairwise relationships, distributions and correlations for attributes in thyroid

The same conclusion that the diagnosis types are distinct can be made by looking at the parallel coordinates plot of the data shown in Figure 54.1. It can be calculated as follows:

```
> par(oma = c(4, 1, 1, 1))

> MASS::parcoord(thyroid2,
+ col = diagnosis_col,
+ var.label = TRUE,
+ lwd = 2)
```

```
> par(fig = c(0, 1, 0, 1),
  oma = c(0, 0, 0, 0),
  mar = c(0, 0, 0, 0),
  new = TRUE)

> plot(0, 0, type = "n",
  bty = "n",
  xaxt = "n", yaxt = "n")

> legend("bottom", cex = 1.25,
  horiz = TRUE, inset = c(1,0),
  bty = "n", xpd = TRUE,
  legend = as.character(levels(diagnosis_labels)),
  fill = unique(diagnosis_col))

par(xpd = NA)
```

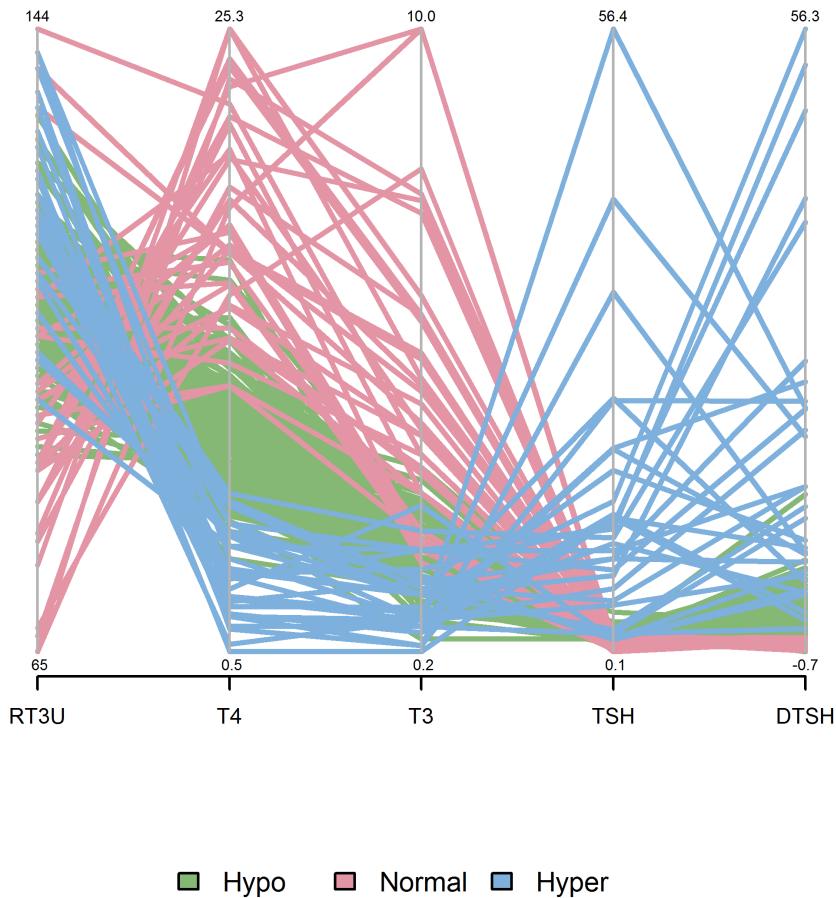


Figure 54.2: Parallel coordinates plot using thyroid

Step 3: Estimate and Assess the Model

The `agnes` function offers a number of methods for agglomerative nesting. These include "average", "single" (single linkage), "complete" (complete linkage), "ward" (Ward's method) and "weighted". The default is "average", however I have had some success with this ward's method in the past so I usually give it a try first. I also set the parameter `stand = TRUE` to standardize the attributes.

```
> fit <- agnes(thyroid2, stand = TRUE, metric = "
```

```
euclidean", method = "ward")
```

The `print` method provides an overview of `fit` (we only show the first few lines of output below).

```
> print(fit)
Call: agnes(x = thyroid2,
metric = "euclidean",
stand = TRUE,
method = "ward")
```

```
Agglomerative coefficient: 0.9819448
```

The agglomerative coefficient measures the amount of clustering structure. Higher values indicate more structure. It can also be viewed as the average width of the banner plot shown in Figure 54.3. A banner plot is calculated using the `bannerplot` method.

```
> bannerplot(fit)
```

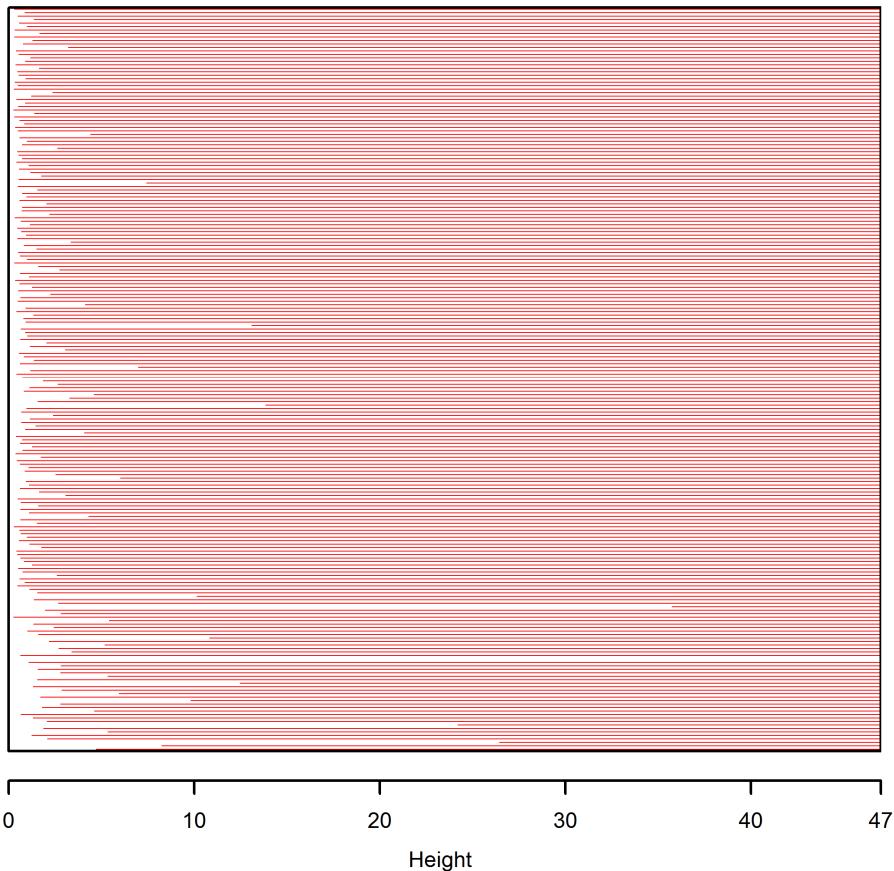


Figure 54.3: Agglomerative nesting bannerplot using thyroid

Now we should take a look at the fitted model. It will be in the form of a dendrogram. First we set `diagnosis` to capture the classes. Then we save `fit` as dendrogram in `dend`, followed by ordering of the observations somewhat using the `rotate` method.

```
> diagnosis<- rev(levels(diagnosis_labels))
> dend <- as.dendrogram(fit)
> dend <- rotate(dend, 1:215)
```

Our next step is to color the branches based on the three clusters. We set `k=3` in `color_branches` for each of the three diagnosis types (hyper, normal, hypo)

```
> dend <- color_branches(dend, k=3)
```

Then we match the labels, to the actual classes.

```
> labels_colors(dend) <-
rainbow_hcl(3)[sort_levels_values(
as.numeric(diagnosis_labels)
[order.dendrogram(dend)]
)]
```

We still need to add the diagnosis classes to the labels. This is achieved as follows.

```
> labels(dend) <- paste(as.character(diagnosis_
  labels)
[order.dendrogram(dend)],
"(",labels(dend),")",
sep = "")
```

A little administration is our next step; reduce the size of the labels to 75% of their original size. This assists in making the eventual plot look less cluttered.

```
> dend <- set(dend, "labels_cex", 0.75)
```

And now to the visualization using the `plot` method. .

```
> par(mar = c(3,3,3,7))

> plot(dend,
main = "Clustered Thyroid data where the labels give
the true diagnosis",
horiz = TRUE,
nodePar = list(cex = .007))

legend("bottomleft", legend = diagnosis, fill =
rainbow_hcl(3))
```

The result is show in Figure 54.4. Overall Hyper seems well separated with some mixing between Normal and Hypo.

Clustered Thyroid data where the labels give the true diagnosis

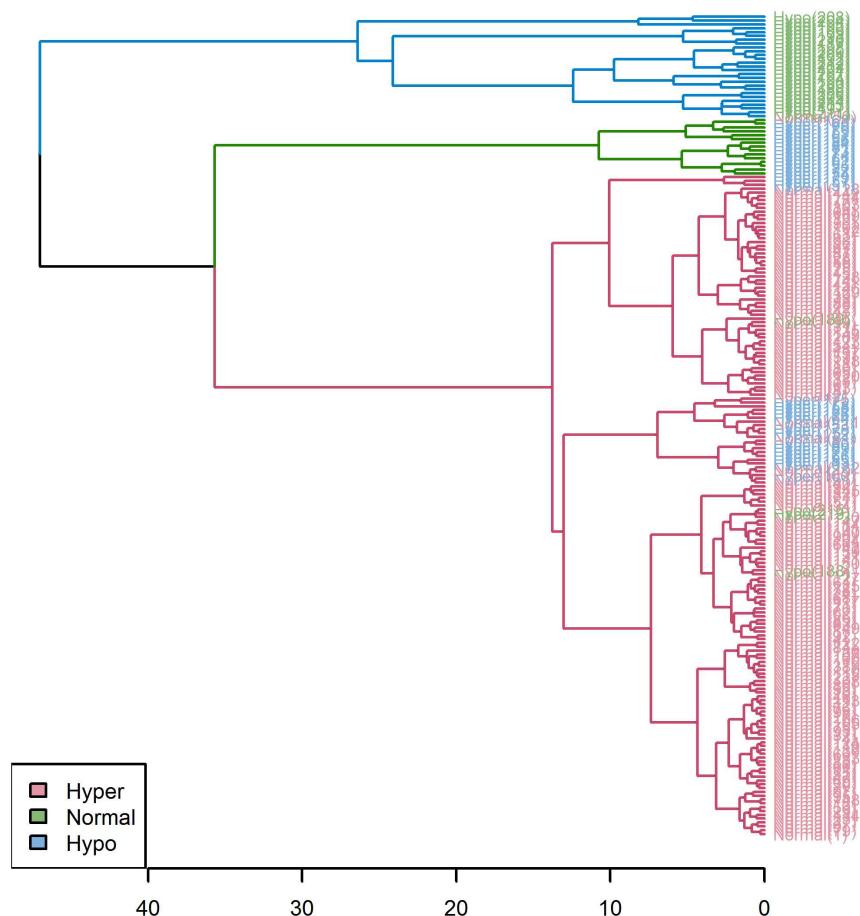


Figure 54.4: Agglomerative Nesting dendrogram using `thyroid`. The label color corresponds to the true diagnosis

Since the `agnes` function provides multiple methods, in the spirit of predictive analytics empiricist analysis let's look at the correlation between the five most popular methods. We assign these methods to `agnes_methods`.

```
> agnes_methods=c("ward",
+ "average", "single",
+ "complete", "weighted")
```

```
> agnes_list<-dendlist()
```

Our main look for the calculations is as follows.

```
> for(i in seq_along(agnes_methods))
{
temp_fit<-agnes(thyroid2,
stand = TRUE,
metric="euclidean",
method=agnes_methods[i])

agnes_list <- dendlist(agnes_list ,
as.dendrogram(temp_fit))
}
```

Need to add the names of the methods as a final touch.

```
> names(agnes_list ) <- agnes_methods
```

Now let's take a look at the output.

```
> agnes_list
$ward
'dendrogram' with 2 branches
and 215 members total,
at height 47.01778

$average
'dendrogram' with 2 branches
and 215 members total,
at height 11.97675

$single
'dendrogram' with 2 branches
and 215 members total,
at height 5.450811

$complete
'dendrogram' with 2 branches
and 215 members total,
at height 24.03689

$weighted
'dendrogram' with 2 branches
and 215 members total,
at height 16.12223
```

Notice that all methods produce two branches, however there is considerable variation in the height of the dendrogram ranging from 5.4 for "single" to 47 for "ward".

We use `corrplot` with `method="common"` to visualize the correlation between the methods.

```
> corrplot(corr.dendlist(agnes_list, method="common"),  
  "pie", type= "lower")
```

The resultant plot shown in Figure 54.5 gives us another perspective on our clustering algorithms. We can see that most of methods have around 75% nodes in common with one another.

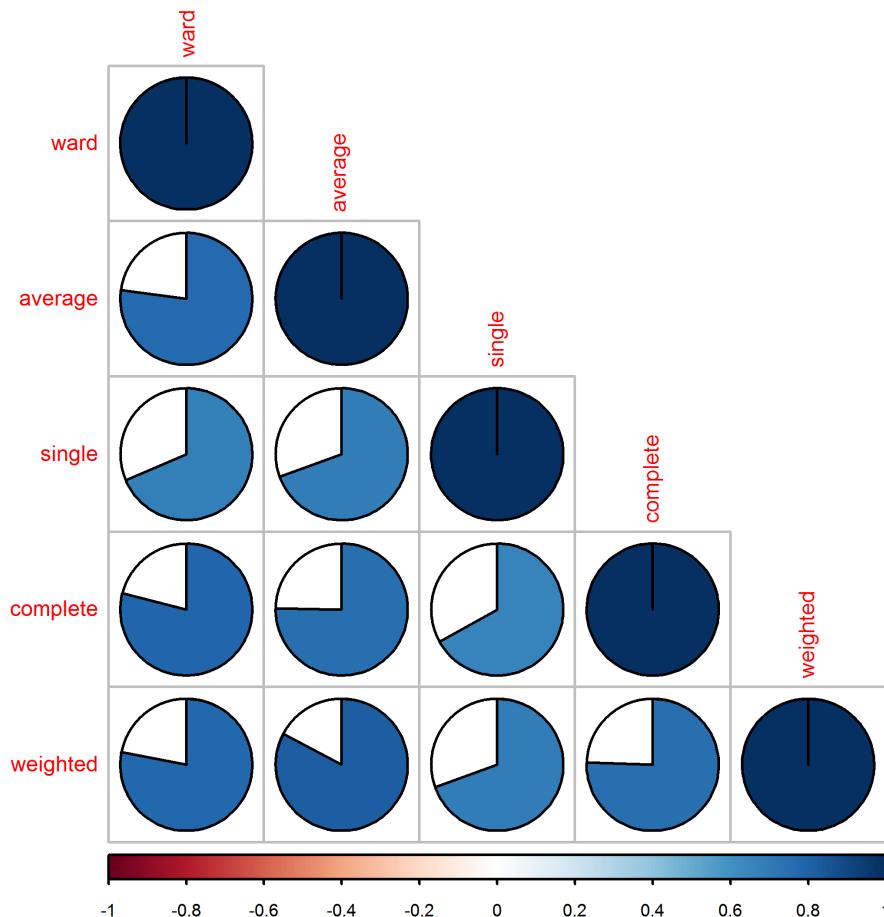


Figure 54.5: Agglomerative Nesting correlation plot using `thyroid`

Finally let's plot the all five dendograms. Here is how.

```
> par( mfrow = c(3, 2))

> for(i in seq_along(agnes_methods))
{
  fit <- agnes(thyroid2, stand = TRUE,
metric="euclidean",
method=agnes_methods[i])

dend <- as.dendrogram(fit)
dend <- rotate(dend, 1:215)

dend <- color_branches(dend, k=3)

labels_colors(dend) <-
rainbow_hcl(3)[sort_levels_values(
as.numeric(diagnosis_labels)
[order.dendrogram(dend)]
)]}

labels(dend) <- paste(as.character(diagnosis_labels)
[order.dendrogram(dend)],
"(",labels(dend),")",
sep = "")

dend <- set(dend, "labels_cex", 0.75)

par(mar = c(3,3,3,7))

plot(dend,
main = agnes_methods[i],
horiz = TRUE,
nodePar = list(cex = .007))
legend("bottomleft",
legend = diagnosis,
fill = rainbow_hcl(3))
}
```

Wow! That is a nice bit of typing, the result is well worth it. Take a look at Figure 54.6 (It still appears as ward's method works the best).

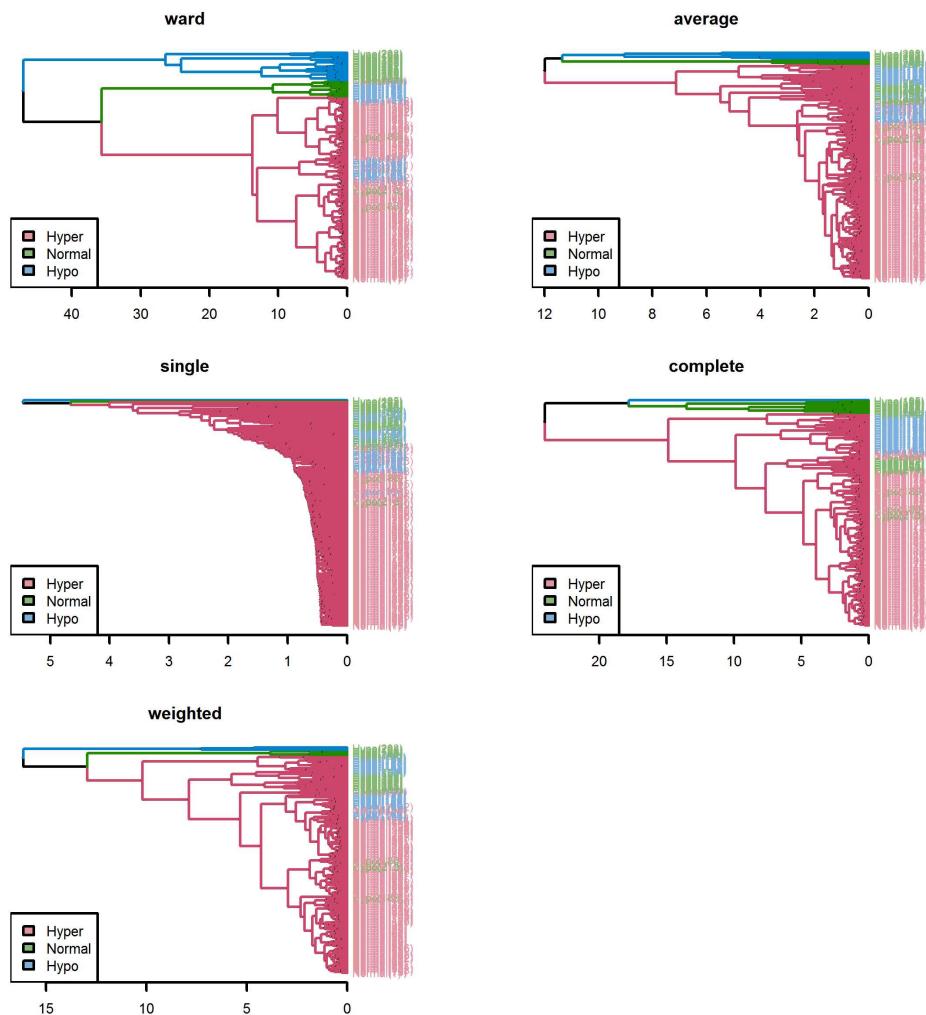


Figure 54.6: Agglomerative nesting dendogram for various methods using thyroid

Technique 55

Divisive Hierarchical Clustering

Divisive hierarchical clustering is available in the `cluster` package using the function `diana`:

```
diana(x, metric, ...)
```

Key parameters include `x` the data set of attributes for which you wish to find clusters and `metric` the metric used for calculating dissimilarities.

Step 1: Load Required Packages

First we load the required packages. I'll explain each below.

```
> require(cluster)
> library(colorspace)
> library(dendextend)
> require("circlize")
```

We use the `thyroid` data frame contained in the `mclust` package for our analysis; see 385 page for additional details on this data. For color output we use the `colorspace` package and `dendextend` to create fancy dendograms. The `circlize` package is used to visualize a circular dendogram.

Step 2: Prepare Data & Tweak Parameters

We store the thyroid data set in `thyroid2` and remove the class variable (`Diagnosis`). The variable `diagnosis_col` will be used to color the dendograms using `rainbow_hcl`.

```
> thyroid2<-thyroid[, -1]
> diagnosis_labels <- thyroid[, 1]
```

```
> diagnosis_col <- rev(rainbow_hcl(3))[as.numeric(  
  diagnosis_labels)]
```

Step 3: Estimate and Assess the Model

The model can be fitted as follows.

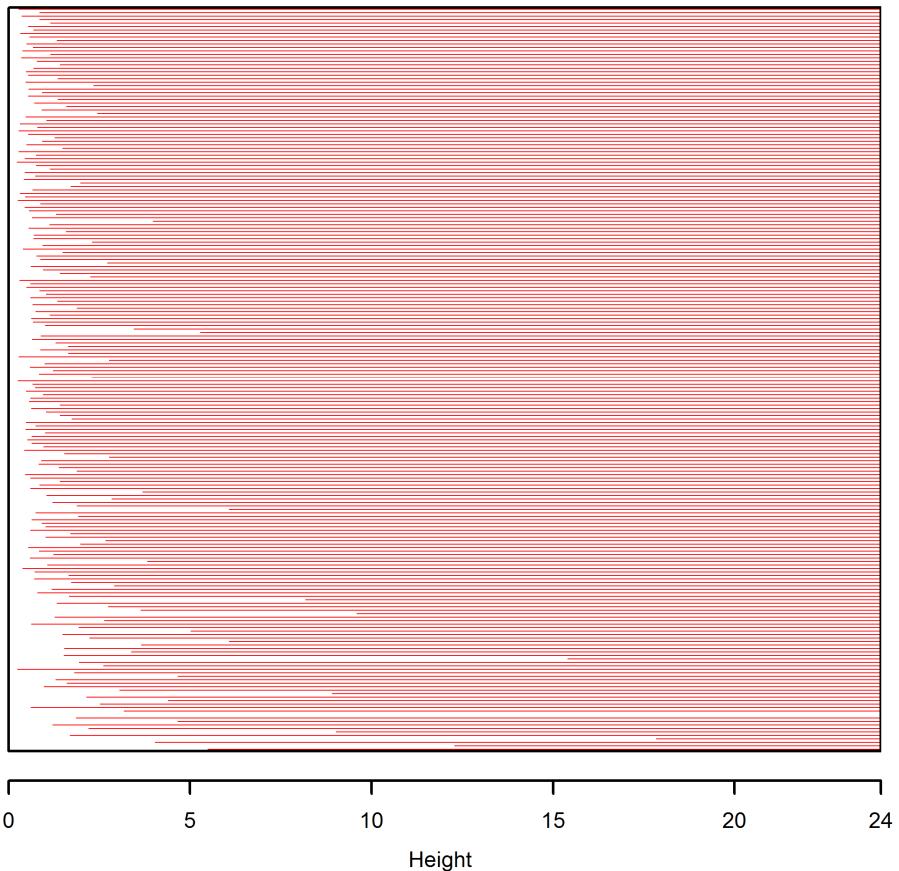
```
> fit <- diana(thyroid2, stand = TRUE, metric =  
  "euclidean")
```

The parameter `stand = TRUE` is used to standardize the attributes. The parameter `metric="euclidean"` is used for calculating dissimilarities; a popular alternative is "`manhattan`".

Now we can use the `bannerplot` method.

```
> bannerplot(fit, main = "Euclidean")
```

The result is shown in Figure 55.1.

EuclideanFigure 55.1: Divisive hierarchical clustering bannerplot using `thyroid`

We should take a look at the fitted model. It will be in the form of a dendrogram. First we set `diagnosis` to capture the classes. Then we save `fit` as a dendrogram into `dend`, followed by ordering of the observations somewhat using the `rotate` method.

```
> diagnosis<- rev(levels(diagnosis_labels))
> dend <- as.dendrogram(fit)
> dend <- rotate(dend, 1:215)
```

Our next step is to color the branches based on the three clusters. We set `k=3` in `color_branches` for each of the three diagnosis types (hyper, normal, hypo)

```
> dend <- color_branches(dend, k=3)
```

We match the labels, to the actual classes.

```
> labels_colors(dend) <-
rainbow_hcl(3)[sort_levels_values(
as.numeric(diagnosis_labels)
[order.dendrogram(dend)]
)]
```

We still need to add the diagnosis classes to the labels. This is achieved as follows.

```
> labels(dend) <- paste(as.character(diagnosis_
  labels)
[order.dendrogram(dend)],
"(",labels(dend),")",
sep = "")
```

A little administration is our next step; reduce the size of the labels to 70% of their original size. This assists in making the eventual plot look less cluttered. Then circlize the dendrogram using method `circlize_dendrogram`.

```
> dend <- set(dend, "labels_cex", 0.7)
> circlize_dendrogram(dend)
> legend("bottomleft", legend = diagnosis, fill =
  rainbow_hcl(3), bty="n")
```

The result is show in Figure 55.2. Overall Hyper seems well separated with some mixing between Normal and Hypo.

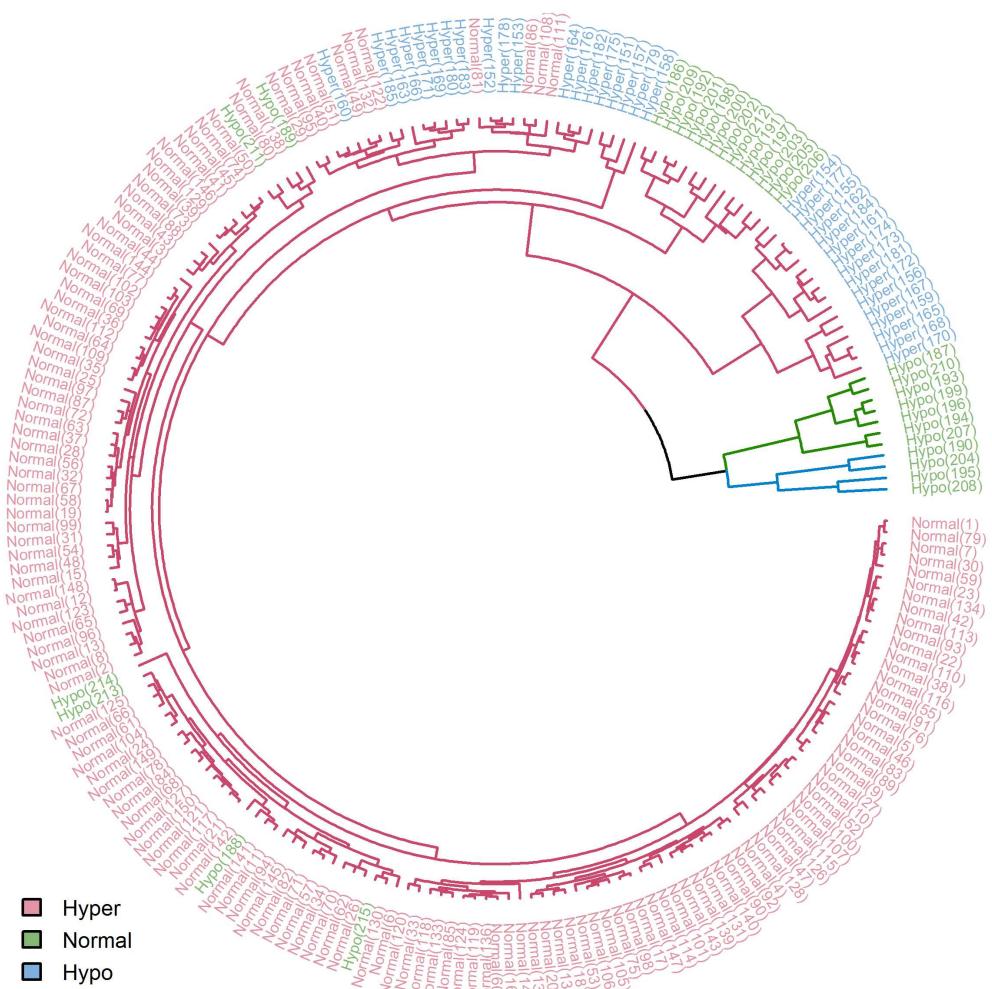


Figure 55.2: Divisive hierarchical clustering dendrogram using thyroid. The label color corresponds to the true diagnosis

Technique 56

Exemplar Based Agglomerative Clustering

Exemplar Based Agglomerative Clustering is available in the `apcluster` package using the function `aggExCluster`:

```
aggExCluster(d, x)
```

Key parameters include `x` the data set of attributes for which you wish to find clusters and `d` the similarity matrix.

Step 1: Load Required Packages

First we load the required packages. I'll explain each below.

```
> require(apcluster)
> data("bodyfat", package = "TH.data")
```

We use the `bodyfat` data frame contained in the `TH.data` package for our analysis; see page 62 for additional details on this data.

Step 2: Prepare Data & Tweak Parameters

We store the `bodyfat` data set in `x`.

```
> set.seed(98765)
> x <- bodyfat
```

Step 3: Estimate and Assess the Model

The model can be fitted as follows.

```
> fit <- aggExCluster(negDistMat(r=10), x)
```

Next we visualize the dendrogram using the `plot` method.

```
> plot(fit, showSamples=TRUE)
```

The result is shown in Figure 56.1.

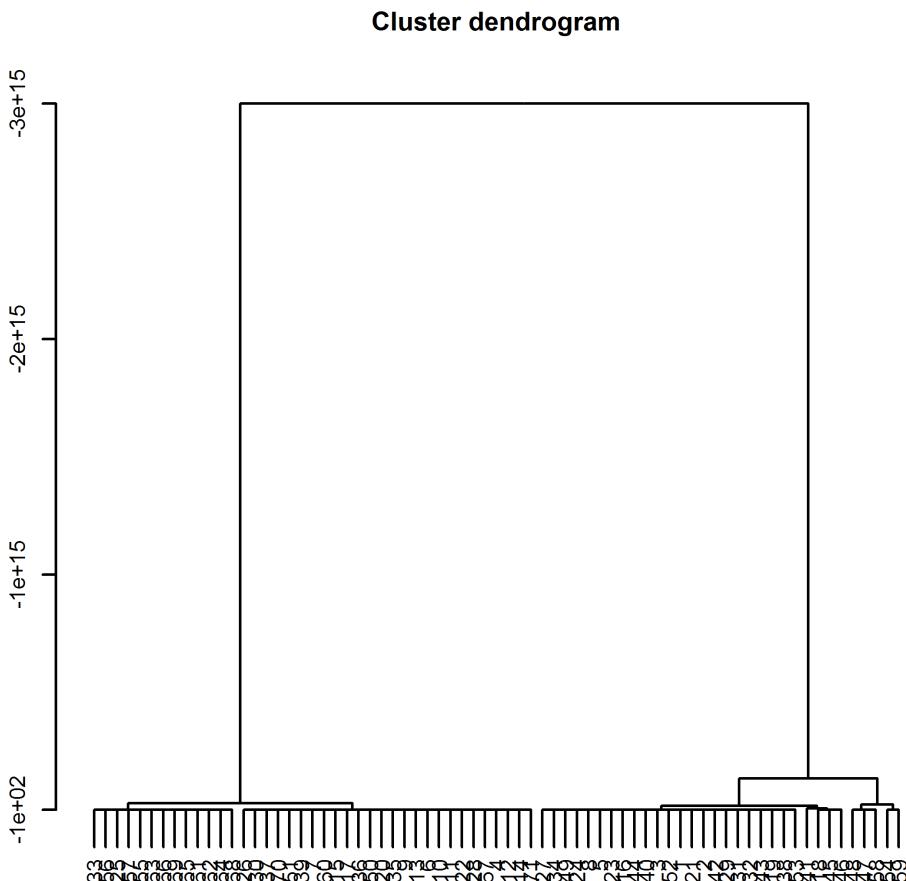


Figure 56.1: Agglomerative Clustering dendrogram using `bodyfat`

Let's look at the relationship between `dexfat` and `waistcirc` for different cluster sizes. To do this we use a `while` loop, see Figure 56.2. There are two main branches with possibly two sub-branches each suggesting 4 or 5 clusters overall.

```

> i=2
> par(mfrow = c(2, 2))

> while (i<=5)
{
  plot(fit, x[,c(2,3)], k=i,xlab="DEXfat", ylab=
    "waistcirc",main=c("Number Clusters ",i))
  i=i+1
}

```

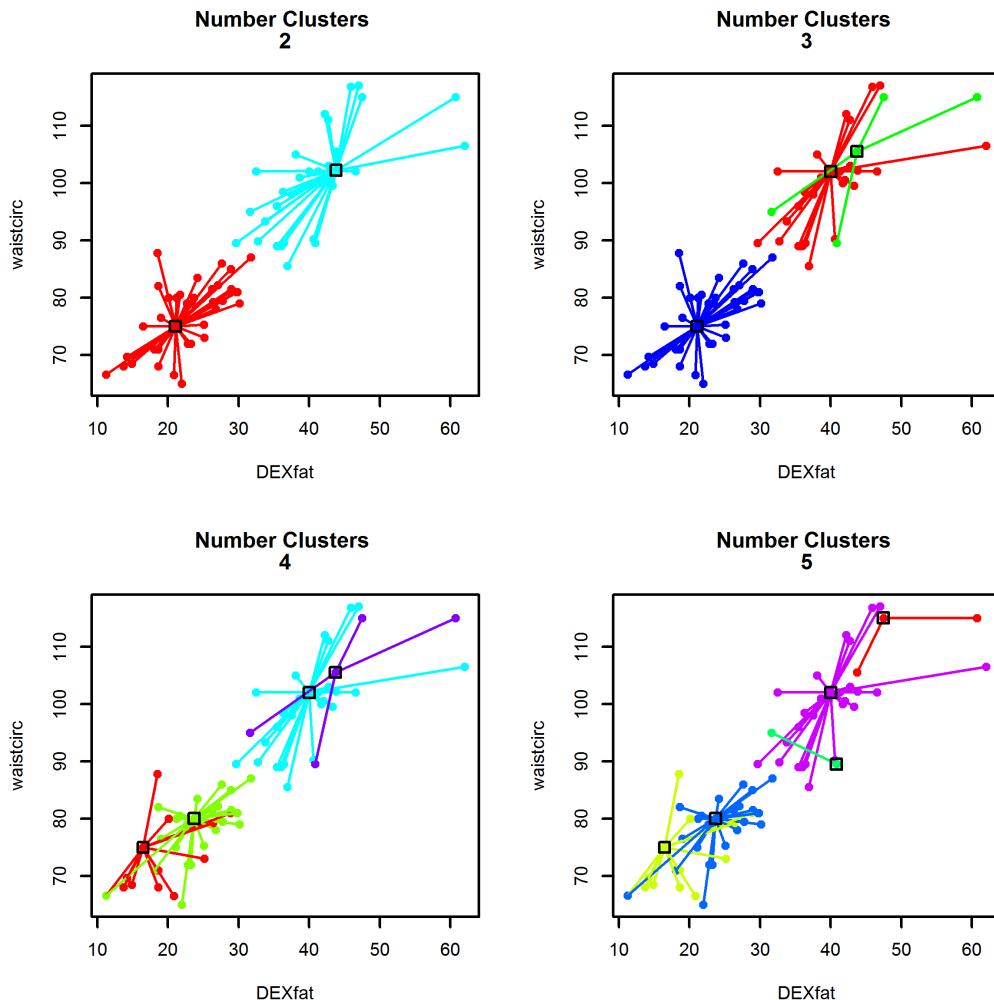


Figure 56.2: Exemplar Based Agglomerative Clustering - `dexfat` and `waistcirc` for various cluster sizes

Fuzzy Methods

Technique 57

Rousseeuw-Kaufman's Fuzzy Clustering Method

The method outlined by Rousseeuw and Kaufman¹¹¹ for fuzzy clustering is available in the `cluster` package using the function `fanny`:

```
fanny(x, k, memb.exp, metric, ...)
```

Key parameters include `x` the data set of attributes for which you wish to find clusters, `k` the number of clusters, `memb.exp` the membership exponent used in the fit criterion, and `metric` the measure to be used for calculating dissimilarities between observations.

Step 1: Load Required Packages

We use the `wine` data frame contained in the `ordinal` package (see page 95). We also load the `smacof` package which offers a number of approaches for multidimensional scaling.

```
> require(cluster)
> require("smacof")
> data("wine", package = "ordinal")
```

Step 2: Prepare Data & Tweak Parameters

We store the `wine` data set in `data` and then remove the wine ratings by specifying `data<-data[,-2]`. One of the advantages of Rousseeuw-Kaufman's fuzzy clustering method over other fuzzy clustering techniques is that it can handle dissimilarity data. Since `wine` contains ordinal variables we use the

daisy method to create a dissimilarity matrix with `metric` set to the general dissimilarity coefficient of Gower¹¹².

```
> set.seed(1432)
> data<-wine
> data<-data[,-2]
> x<-daisy(data,metric = "gower")
```

Step 3: Estimate and Assess the Model

We need to specify the number of clusters. Let's begin with 3 clusters (`k=3`) and setting `metric = "gower"`. We set the parameter `memb.exp = 1.1`. The higher the value in `memb.exp` the more fuzzy the clustering in general.

```
> fit<-fanny(x,k=3,memb.exp = 1.1,metric="gower")
```

The cluster package provides silhouette plots via the `plot` method. These can help determine the viability of the clusters. Figure 57.1 presents the silhouette plot for `fit`. We see one large cluster containing 36 observations with a width of 0.25 and two smaller clusters each with 18 observations and an average width of 0.46.

```
> plot(fit)
```

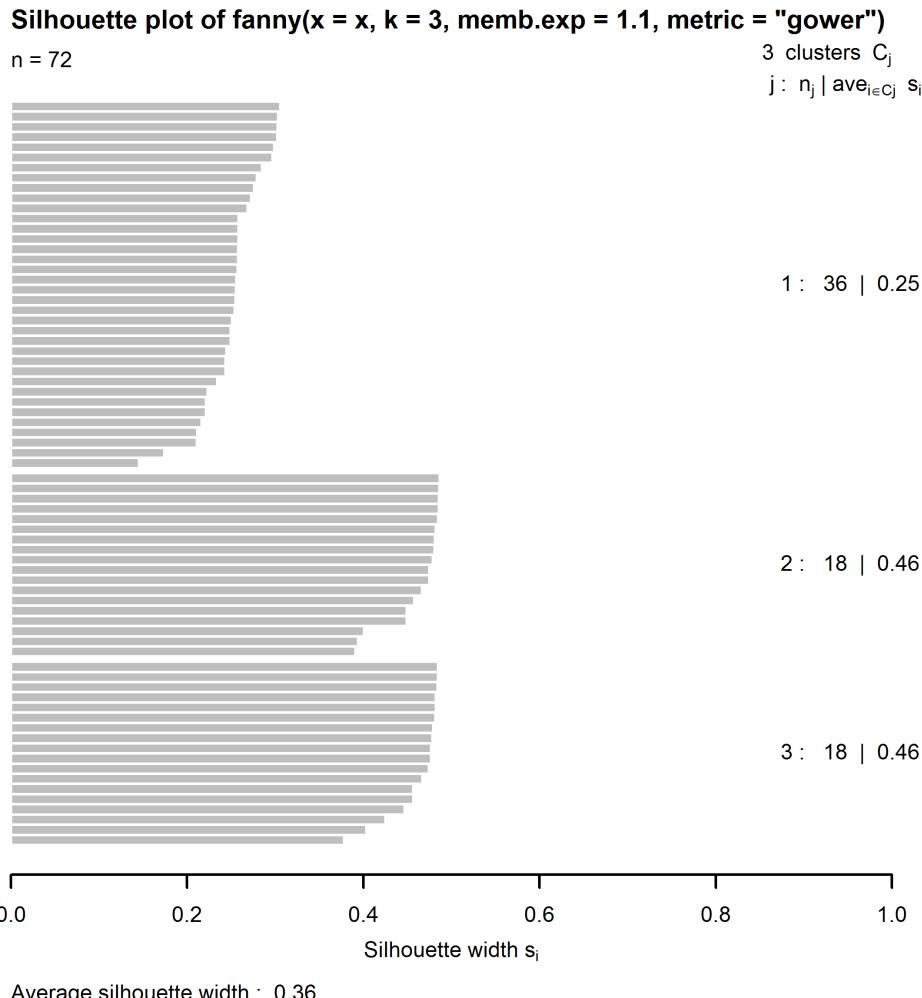


Figure 57.1: Rousseeuw-Kaufman's fuzzy clustering method silhouette plot using `wine`

It can be instructive to investigate the relationship between the fuzziness parameter in `memb.exp` and the average silhouette width. To do this we use a small `while` loop and call `fanny` with values of `memb.exp` ranging from 1.1 to 1.8.

```
> fuzzy=1.1
> while (fuzzy<1.9)
{
  temp_fit<-fanny(x,k=3,memb.exp = fuzzy,metric =
    "euclidean")
```

```
cat("Fuzz = ", fuzzy , " Average Silhouette Width =  
" ,round(temp_fit$silinfo[[3]] ,1),"\n")  
  
fuzzy=fuzzy+0.1  
}
```

The resultant output is shown below. Notice how the average silhouette width declines as the fuzziness increases.

```
Fuzz = 1.1 Average Silhouette Width = 0.4  
Fuzz = 1.2 Average Silhouette Width = 0.3  
Fuzz = 1.3 Average Silhouette Width = 0.3  
Fuzz = 1.4 Average Silhouette Width = 0.3  
Fuzz = 1.5 Average Silhouette Width = 0.3  
Fuzz = 1.6 Average Silhouette Width = 0.3  
Fuzz = 1.7 Average Silhouette Width = 0.3  
Fuzz = 1.8 Average Silhouette Width = 0.2
```

We still need to determine the optimal number of clusters. Let's try multidimensional scaling to investigate further. We use the `smacofSym` method from the package `smacof` and store the result in `scaleT`.

```
> scaleT<- smacofSym(x)$conf  
  
> plot(scaleT, type = "n",main="Fuzzy = 1.1 with k =  
3")  
  
> text(scaleT, label = rownames(scaleT),col = rgb(  
fit$membership))
```

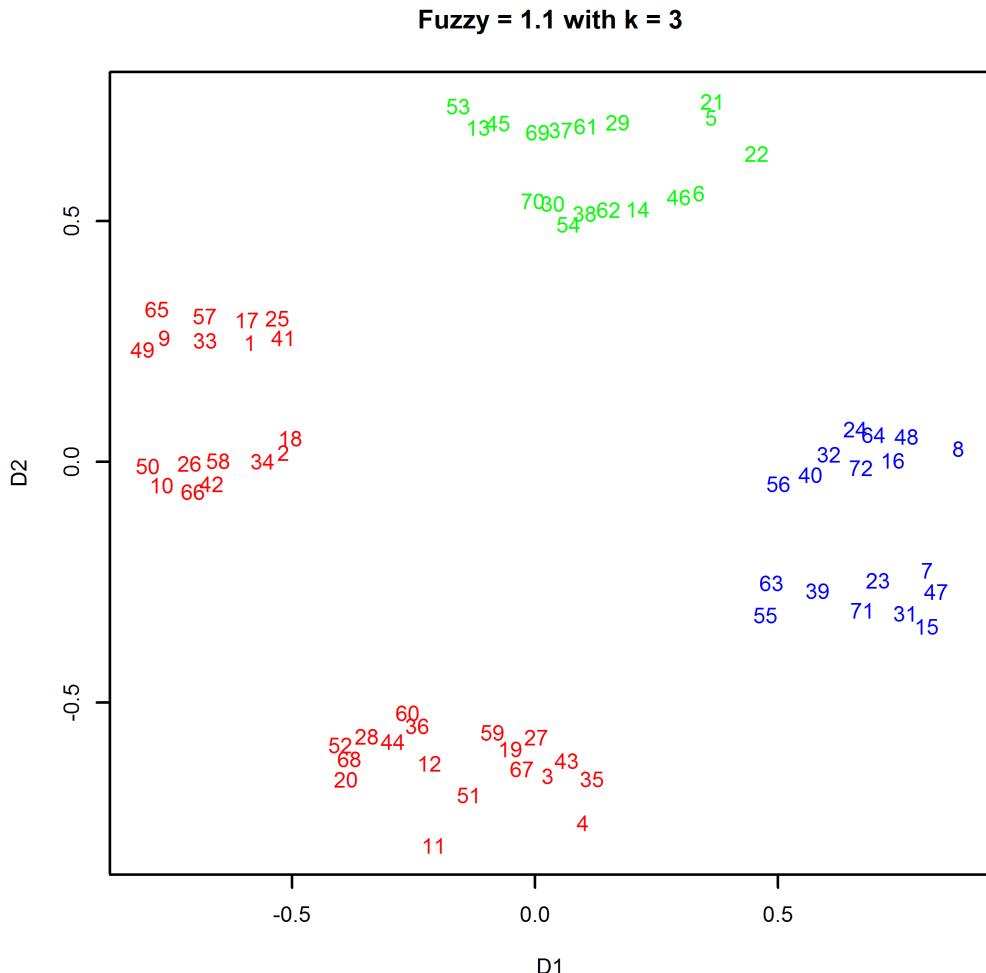


Figure 57.2: Use the `smacofSym` method to help identify clusters using `wine`

The result is visualized using the `plot` method and shown in Figure 57.2. The image appears to identify four clusters. Even though five official ratings were used to assess the quality of the wine we will refit using $k=4$

```
> fit<-fanny(x,k=4,memb.exp = 1.1,metric="gower")
```

Finally, we create a silhouette plot.

```
> plot(fit)
```

Figure 57.3 shows the resultant plot. A rule of thumb¹¹³ is to choose as the number of clusters the silhouette plot with the largest average. Here we see the average silhouette plot is now 0.46, compared to 0.36 for $k=3$.

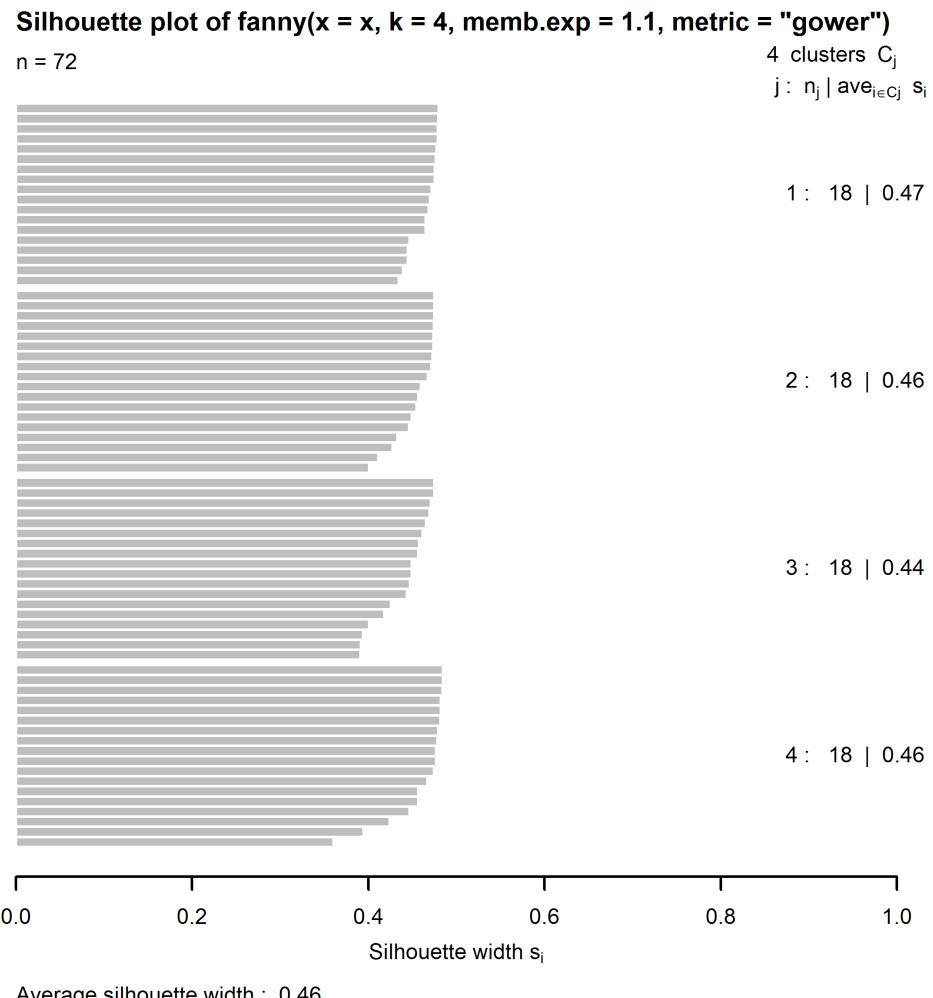


Figure 57.3: Rousseeuw-Kaufman's method silhouette plot with four clusters using `wine`

Technique 58

Fuzzy K-Means

Fuzzy K-means is available in the `fclust` package using the function `FKM`:

```
FKM(x, k=3, m=1.5, RS=1, ...)
```

Key parameters include `x` the data set of attributes for which you wish to find clusters, `k` the number of clusters, `m` the fuzziness parameter and `RS` the number of random starts.

Step 1: Load Required Packages

We use the `Vehicle` data frame contained in the `mlbench` package (see page 23).

```
> require (fclust)
> data("Vehicle", package="mlbench")
```

Step 2: Prepare Data & Tweak Parameters

We store the `Vehicle` data set in `x` and remove the vehicle types stored in column 19.

```
> set.seed(98765)
> x<-Vehicle[, -19]
```

Step 3: Estimate and Assess the Model

Suppose we believe the actual number of clusters will be between 3 and 6. We can estimate a model with 3 clusters using the following.

```
> fit3<-FKM(x, k=3, m=1.5, RS=1, stand=1)
```

We set the fuzzy parameter `m` to 1.5 and for illustrative purposes only set `RS = 1`. In practice you will want to set a higher number. The parameter `stand = 1` instructs the algorithm to standardize the observations in `x` before any analysis.

The other three models can be estimated in a similar manner.

```
> fit4<-FKM(x, k=4, m=1.5, RS=1, stand=1)
> fit5<-FKM(x, k=5, m=1.5, RS=1, stand=1)
> fit6<-FKM(x, k=6, m=1.5, RS=1, stand=1)
```

The question we now face is how to determine which of our models is optimal. Fortunately we can use the `fclust.index` function to help us out. This function returns the value of six clustering indices often used for choosing the optimal number of clusters. The indices include "PC" (partition coefficient), "PE" (partition entropy), "MPC" (modified partition coefficient), "SIL" (silhouette), "SIL.F" (fuzzy silhouette) and "XB" (Xie and Beni index). The key thing to remember is that the optimal number of clusters occurs at the maximum value of each of these indices except for PE, where the optimal number of clusters occurs at the minimum.

Here is how to calculate the index for each model.

```
> f3<-Fclust.index(fit3)
> f4<-Fclust.index(fit4)
> f5<-Fclust.index(fit5)
> f6<-Fclust.index(fit6)
```

Now let's take a look at the output of each `Fclust.index`.

```
> round(f3,2)
   PC      PE      MPC      SIL    SIL.F      XB
 0.76    0.43    0.64    0.44    0.52    0.65

> round(f4,2)
   PC      PE      MPC      SIL    SIL.F      XB
 0.66    0.64    0.54    0.37    0.47    0.99

> round(f5,2)
   PC      PE      MPC      SIL    SIL.F      XB
 0.59    0.78    0.49    0.32    0.41    1.02

> round(f6,2)
   PC      PE      MPC      SIL    SIL.F      XB
 0.52    0.93    0.42    0.26    0.35    1.97
```

Overall, $K = 3$ is the optimum choice for the majority of methods.

It is fun to visualize clusters. Let's focus on `fit3` and focus on the two important variables `Comp` and `Scat.Ra`. It can be instructive to plot clusters using the first two principal components. We use the `plot` method with the parameter `v1v2=c(1,7)` to select `Comp` (first variable in `x` and `Scat.Ra` (seventh variable in `x`); and `pca=TRUE`, in the second plot, to use the principal components as the `x` and `y` axis.

```
> par(mfrow = c(1, 2))
> plot(fit3, v1v2=c(1,7))
> plot(fit3, v1v2=c(1,7), pca=TRUE)
```

Figure 58.1 shows both plots. There appears to be reasonable separation between the clusters.

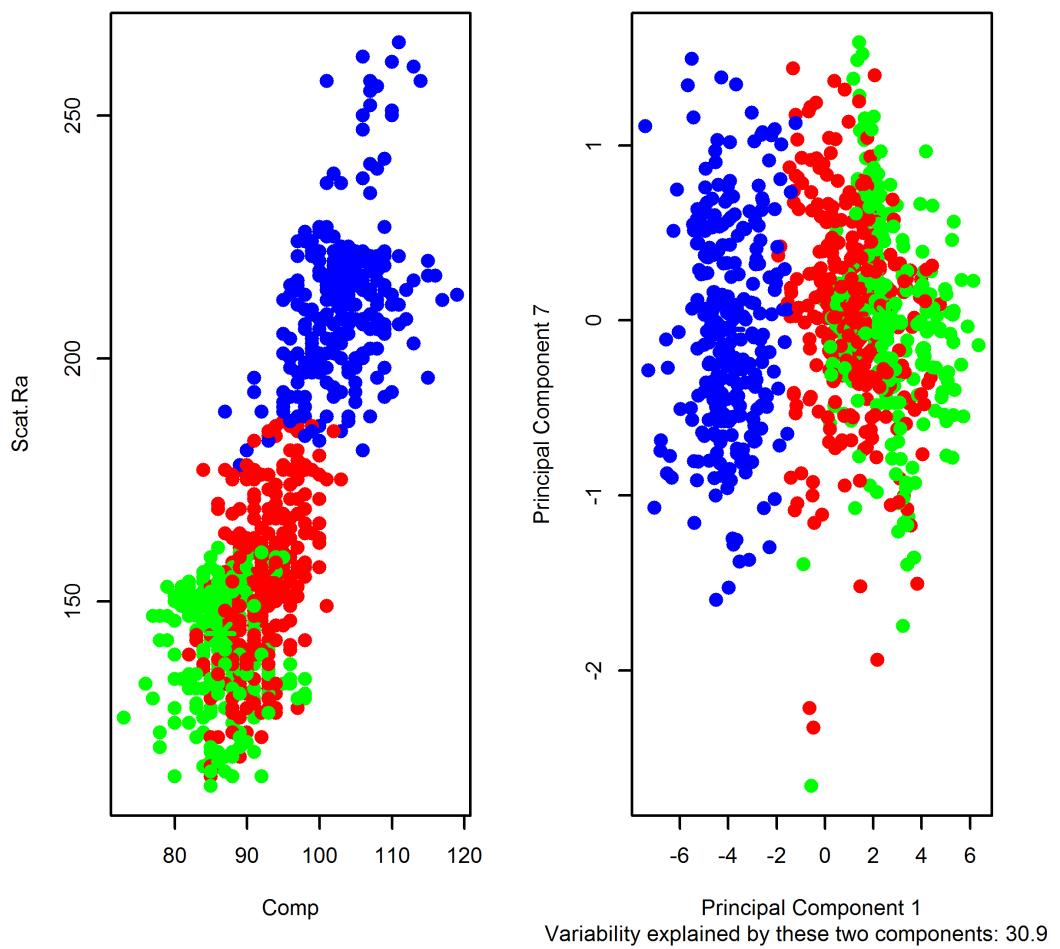


Figure 58.1: Fuzzy K-Means clusters with k=3 using Vehicle

Technique 59

Fuzzy K-Medoids

Fuzzy K-means is available in the `fclust` package using the function `FKM.med`:

```
FKM.med(x, k=3, m=1.5, RS=1, ...)
```

Key parameters include `x` the data set of attributes for which you wish to find clusters, `k` the number of clusters, `m` the fuzziness parameter and `RS` the number of random starts. Note that the difference between fuzzy K-means and fuzzy K-Medoids is that the cluster prototypes (centroids) are artificial randomly computed in fuzzy K-means. In the fuzzy K-Medoids the cluster prototypes (medoids) are a subset of the actual observed objects.

Step 1: Load Required Packages

We use the `Vehicle` data frame contained in the `mlbench` package (see page 23).

```
> require(fclust)
> data("Vehicle", package = "mlbench")
```

Step 2: Prepare Data & Tweak Parameters

We store the `Vehicle` data set in `x` and remove the vehicle types stored in column 19.

```
> set.seed(98765)
> x <- Vehicle[, -19]
```

Step 3: Estimate and Assess the Model

Suppose we believe the actual number of clusters will be between 3 and 6. We can estimate a model with 3 clusters using the following.

```
> fit3<-FKM.med(x, k=3, m=1.5, RS=10, stand=1)
```

We set the fuzzy parameter `m` to 1.5 and `RS = 10`. The parameter `stand = 1` instructs the algorithm to standardize the observations in `x` before any analysis.

The other three models can be estimated in a similar manner.

```
> fit4<-FKM.med(x, k=4, m=1.5, RS=10, stand=1)
> fit5<-FKM.med(x, k=5, m=1.5, RS=10, stand=1)
> fit6<-FKM.med(x, k=6, m=1.5, RS=10, stand=1)
```

The questions we now face is how to determine which of our models is optimal? Fortunately, we can use the `fclust.index` function to help us out. This function returns the value of six clustering indices often used for choosing the optimal number of clusters. The indices include "PC" (partition coefficient), "PE" (partition entropy), "MPC" (modified partition coefficient), "SIL" (silhouette), "SIL.F" (fuzzy silhouette) and "XB" (Xie and Beni index). The key thing to remember is that the optimal number of clusters occurs at the maximum value of each of these indices except for PE, where the optimal number of clusters occurs at the minimum.

Here is how to calculate the index for each model.

```
> f3<-Fclust.index(fit3)
> f4<-Fclust.index(fit4)
> f5<-Fclust.index(fit5)
> f6<-Fclust.index(fit6)
```

Now let's take a look at the output of each `Fclust.index`.

```
> round(f3,2)
    PC      PE      MPC      SIL   SIL.F       XB
  0.37    1.04    0.06    0.13    0.15    6.96

> round(f4,2)
    PC      PE      MPC      SIL   SIL.F       XB
  0.29    1.30    0.06    0.09    0.11    6.05

> round(f5,2)
    PC      PE      MPC      SIL   SIL.F       XB
  0.24    1.51    0.05    0.07    0.11    8.63
```

```
> round(f6,2)
  PC      PE      MPC      SIL  SIL.F      XB
0.21   1.68   0.05   0.07   0.08   7.90
```

Overall, K=3 is the optimum choice for the majority of methods.

Now we use `fit3` and focus on the two important variables `Comp` and `Scat.Ra`. It can also be instructive to plot clusters using the first two principal components. We use the `plot` method with the parameter `v1v2=c(1,7)` to select `Comp` (first variable in `x` and `Scat.Ra` (seventh variable in `x`); and `pca=TRUE`, in the second plot, to use the principal components as the `x` and `y` axis.

```
> par(mfrow = c(1, 2))
> plot(fit3,v1v2=c(1,7))
> plot(fit3,v1v2=c(1,7),pca=TRUE)
```

Figure 59.1 shows both plots.

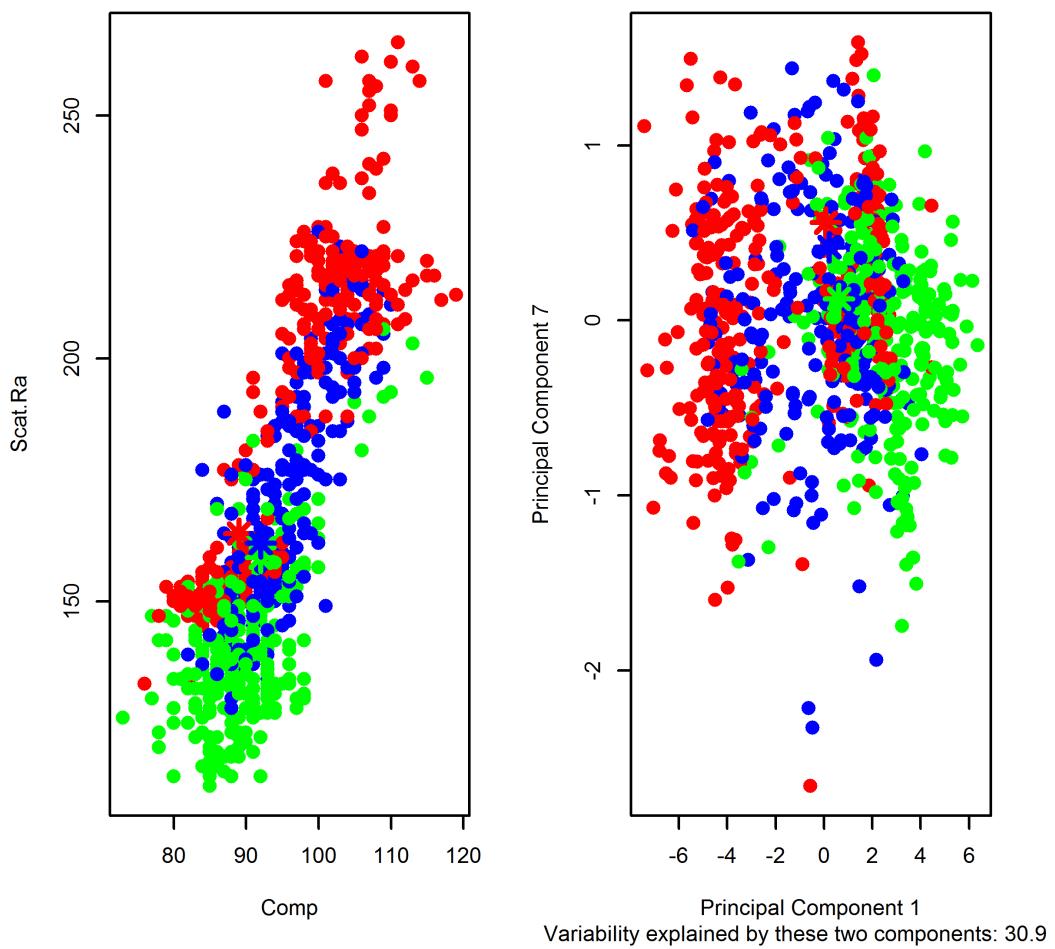


Figure 59.1: Fuzzy K-Medoids clusters with $k=3$ using Vehicle

Other Methods

Technique 60

Density-Based Cluster Analysis

In density based cluster analysis two parameters are important: "eps" defines the radius of neighborhood of each point, and "minpts" is the minimum number of points within a radius (or "reach") associated with the clusters of interest. For our analysis we will use the `dbSCAN` function in the `cluster` package.

```
dbSCAN(data, eps, MinPts)
```

Key parameters include `data` the data matrix, `data.frame` or dissimilarity matrix.

Step 1: Load Required Packages

First we load the required packages.

```
> require(fpc)
> data("thyroid", package = "mclust")
```

We use the `thyroid` data frame contained in the `mclust` package for our analysis; see page 385 for additional details on this data.

Step 2: Estimate and Assess the Model

We estimate the model using the `dbSCAN` method. Note in practice `MinPts` is chosen by trial and error. We set `MinPts = 4` and `eps = 8.5`. We also set `showplot = 1` which will allow you to see visually how `dbSCAN` works.

```
> fit <- dbSCAN(thyroid[, -1], eps = 8.5, MinPts = 4,
+ showplot = 1)
```

Next we take a peek at the fitted model.

```
> fit  
dbscan Pts=215 MinPts=4 eps=8.5  
  
      0   1 2 3  
border 18   3 3 2  
seed     0 185 1 3  
total    18 188 4 5
```

Unlike other approaches to cluster analysis the density based cluster algorithm can identify outliers (data points that do not belong to any clusters). Points in cluster 0 are unassigned outliers - 18 in total.

The pairwise plot of clusters, shown in Figure 60.1, is calculated as follows.

```
> plot(fit,thyroid)
```

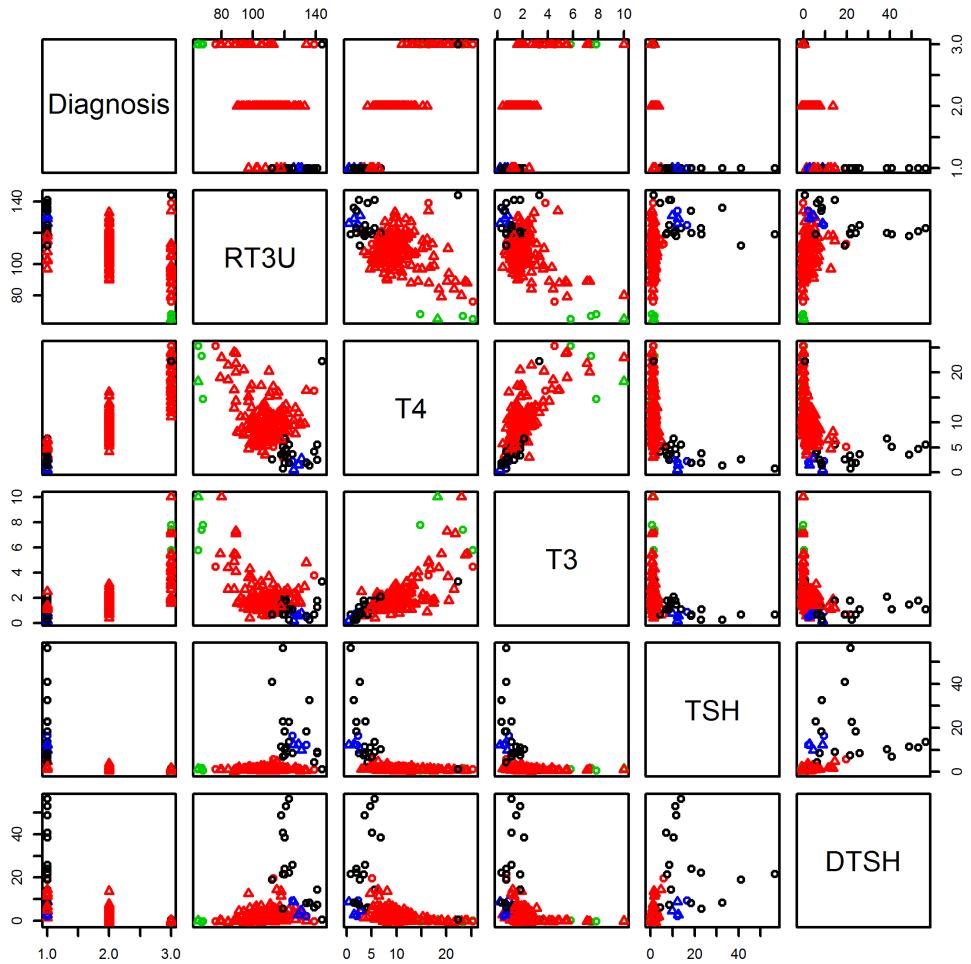


Figure 60.1: Density-Based Cluster Analysis pairwise cluster plot using thyroid

Let's take a closer look at the pairwise cluster plot between the attributes T4 (column 3 in thyroid) and RT3U (column 2 in thyroid). To do this we use the plot method, see Figure 60.2.

```
> plot(fit, thyroid[,c(3,2)])
```

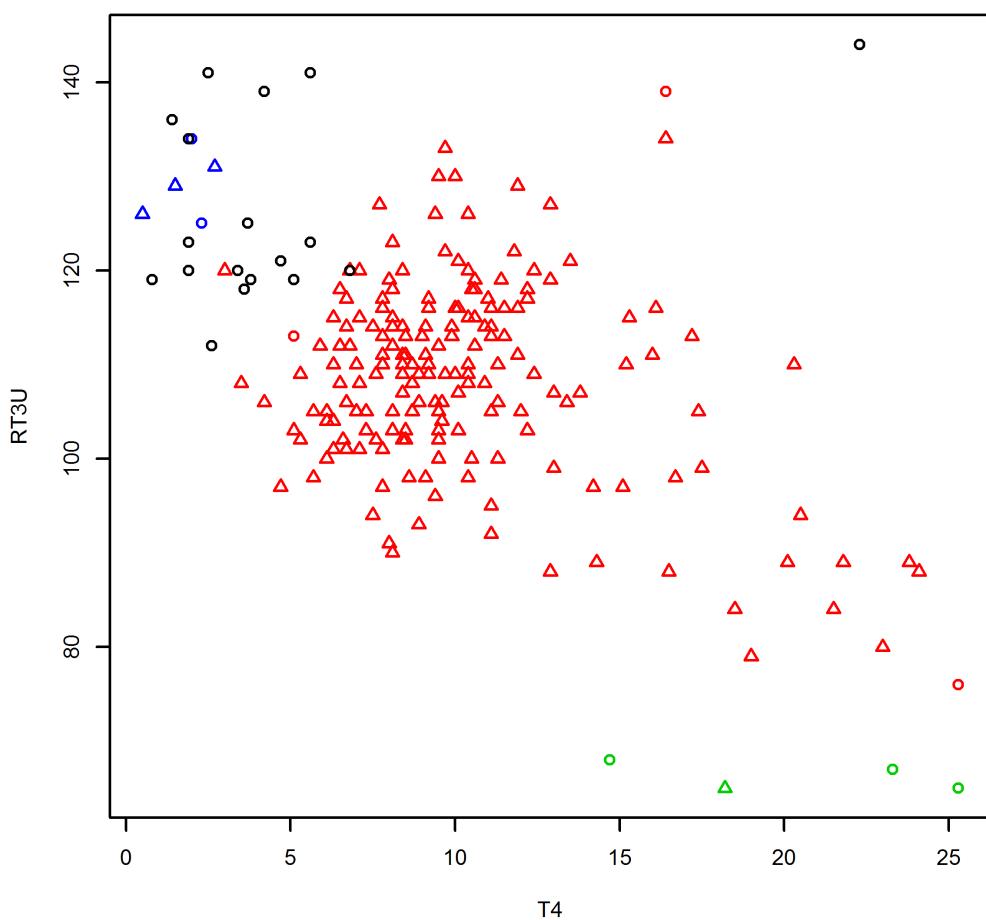


Figure 60.2: Density-Based Cluster Analysis pairwise plot of T4 and RT3U

Technique 61

K-Modes Clustering

K-modes clustering is useful for finding clusters in categorical data. It is available in the `klaR` package using the function `kmodes`:

```
kmodes(x, k)
```

Key parameters include `x` the data set of categorical attributes for which you wish to find clusters and `k` is the number of clusters.

Step 1: Load Required Packages

We use the `housing` data frame contained in the `MASS` package. We also load the `plyr` package which we will use to map attributes in `housing` to numerical categories.

```
> require("klaR")
> data("housing", package = "MASS")
> library(plyr)
```

NOTE... ↗.

The housing data frame contains data from an investigation of satisfaction with housing conditions in Copenhagen carried out by the Danish Building Research Institute and the Danish Institute of Mental Health Research¹¹⁴. It contains 72 rows with measurements on the following 5 variables:

- **Sat:** Satisfaction of householders with their present housing circumstances, (High, Medium or Low, ordered factor).
- **Inf:1:** Perceived degree of influence householders have on the management of the property (High, Medium, Low).
- **Type:** Type of rental accommodation, (Tower, Atrium, Apartment, Terrace).
- **Cont:** Contact residents are afforded with other residents, (Low, High).
- **Freq:** The numbers of residents in each class.

Step 2: Prepare Data & Tweak Parameters

We store the housing data set in `x`. We then convert the attributes into numerical values using `mapvalues` and `as.numeric` methods.

```
> set.seed(98765)
> x<-housing [,-5]

> x$Sat <- mapvalues(x$Sat ,
from = c("Low" ,
"Medium" ,
"High"),
to = c("1" , "2" , "3"))

> x$Sat <-as.numeric(x$Sat)

> x$Infl <- mapvalues(x$Infl ,
```

```
from = c("Low" ,  
"Medium" ,  
"High") ,  
to = c("1" , "2" , "3"))  
  
> x$Infl <- as.numeric(x$Infl)  
  
> x>Type <- mapvalues(x>Type ,  
from = c("Tower" ,  
"Apartment" ,  
"Atrium" ,  
"Terrace") ,  
to = c("1" , "2" , "3" , "4"))  
  
> x>Type <- as.numeric(x>Type)  
  
> x$Cont <- mapvalues(x$Cont ,  
from = c("Low" , "High") ,  
to = c("1" , "3"))  
  
> x$Cont <- as.numeric(x$Cont)
```

Step 3: Estimate and Assess the Model

Suppose we believe the actual number of clusters is 3. We estimate the model and plot the result (see Figure 61.1) as follows.

```
> fit <- kmodes(x , 3)  
> plot(x , col = fit$cluster)
```

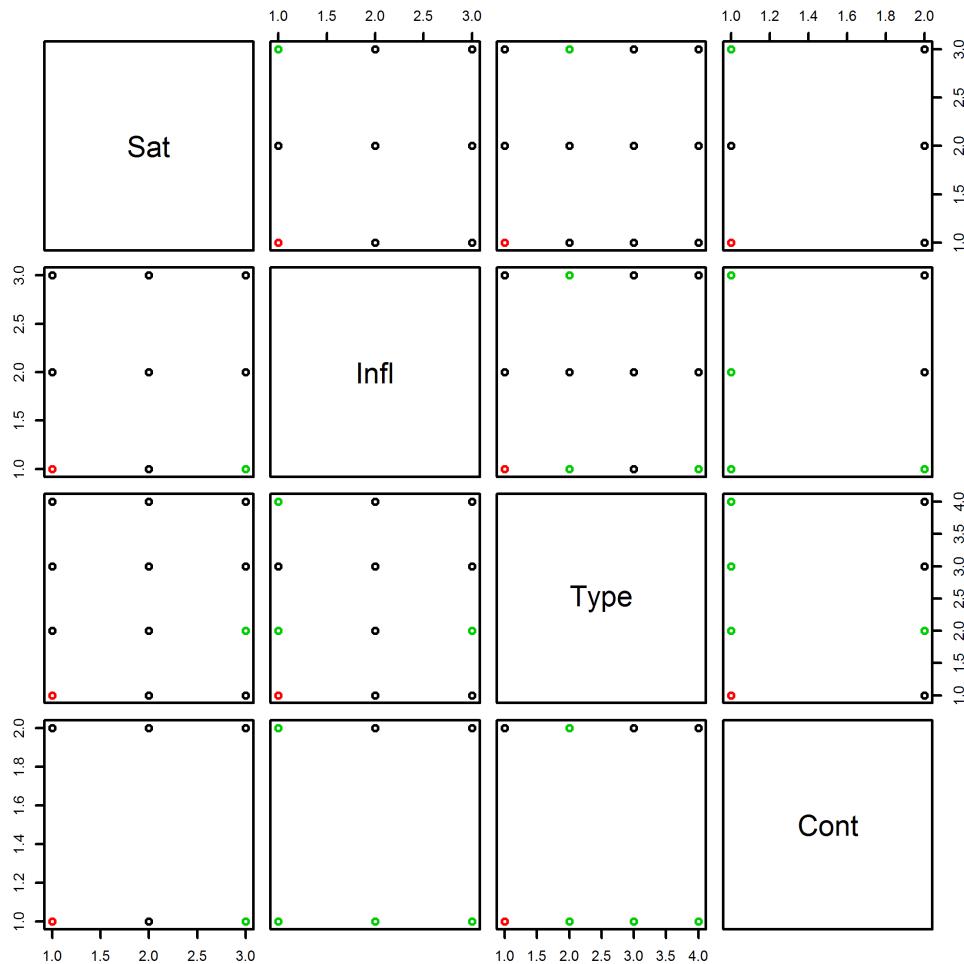


Figure 61.1: K-Modes Clustering with $k = 3$ using housing

It is also possible to narrow down our focus. For example, let's investigate the pairwise relationship between `type` and `sat`. To do this we use the `plot` method with the `jitter` method which adds a small amount of noise to the attributes. The result is shown in Figure 61.2.

```
> plot(jitter(as.matrix(x[,c(1,3)])),  
+       col = fit$cluster)  
  
> points(fit$modes, col = 1:5, pch = 8, cex=4)
```

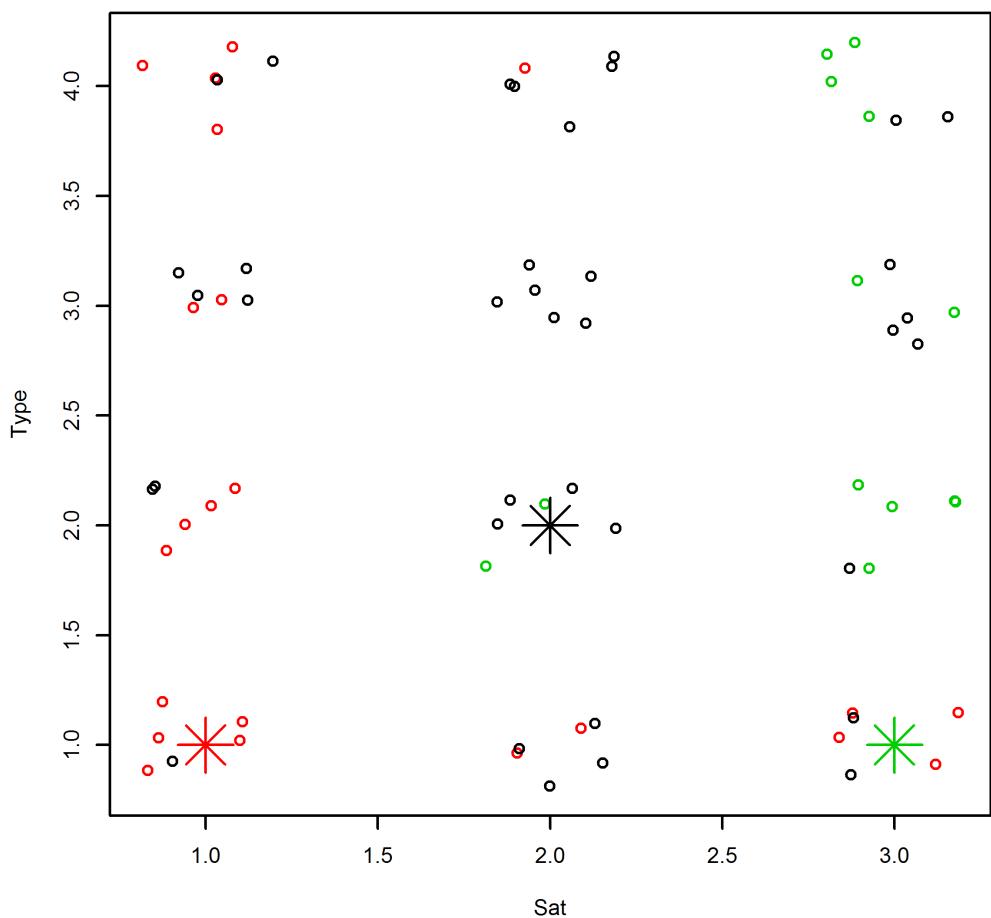


Figure 61.2: K-Modes Clustering pairwise relationship between `type` and `sat`

Technique 62

Model-Based Clustering

Model based clustering identifies clusters using the Bayesian information criterion (BIC) for Expectation-Maximization initialized by hierarchical clustering for Gaussian mixture models. It is available in the `Mclust` package using the function `mclust`:

```
Mclust(x, G)
```

Key parameters include `x` the data set of attributes for which you wish to find clusters and `g` the maximum and minimum expected number of clusters.

Step 1: Load Required Packages

We use the `thyroid` data frame (see page 23).

```
> require("mclust")
> data("thyroid")
```

Step 2: Prepare Data & Tweak Parameters

We store the `thyroid` data set in `x` and remove the first column which contains the `Diagnosis` values.

```
> set.seed(98765)
> x=thyroid[,-1]
```

Step 3: Estimate and Assess the Model

Suppose we believe the number of clusters lies between 1 and 20. We can use the `Mclust` method to find the optimal number as follows:

```
> fit <- Mclust(as.matrix(x), G=1:20)
```

NOTE... 

Notice that for multivariate data the **Mclust** methods computes the optimum Bayesian information criterion for each of the following Gaussian mixture models.

1. "EII" = spherical, equal volume
2. "VII" = spherical, unequal volume
3. "EEI" = diagonal, equal volume and shape
4. "VEI" = diagonal, varying volume, equal shape
5. "EVI" = diagonal, equal volume, varying shape
6. "VVI" = diagonal, varying volume and shape
7. "EEE" = ellipsoidal, equal volume, shape, and orientation
8. "EVE" = ellipsoidal, equal volume and orientation
9. "VEE" = ellipsoidal, equal shape and orientation
10. "VVE" = ellipsoidal, equal orientation
11. "EEV" = ellipsoidal, equal volume and equal shape
12. "VEV" = ellipsoidal, equal shape
13. "EVV" = ellipsoidal, equal volume
14. "VVV" = ellipsoidal, varying volume, shape, and orientation.

Let's take a look at the result.

```
> fit  
'Mclust' model object:  
best model: diagonal, varying volume and shape (VVI  
 ) with 3 components
```

The optimal number of clusters using all 14 Gaussian mixture models

appears to be 3. Here is another way to access the optimum number of clusters.

```
> k <- dim(fit$z)[2]
> k
[1] 3
```

Since BIC values using 14 models are used for choosing the number of clusters, it can be instructive to plot the result by individual model. The `plot` method `what = "BIC"` achieves this. The result is shown in Figure 62.1. Notice most models reach a peak around 3 clusters.

```
> plot(fit, what = "BIC")
```

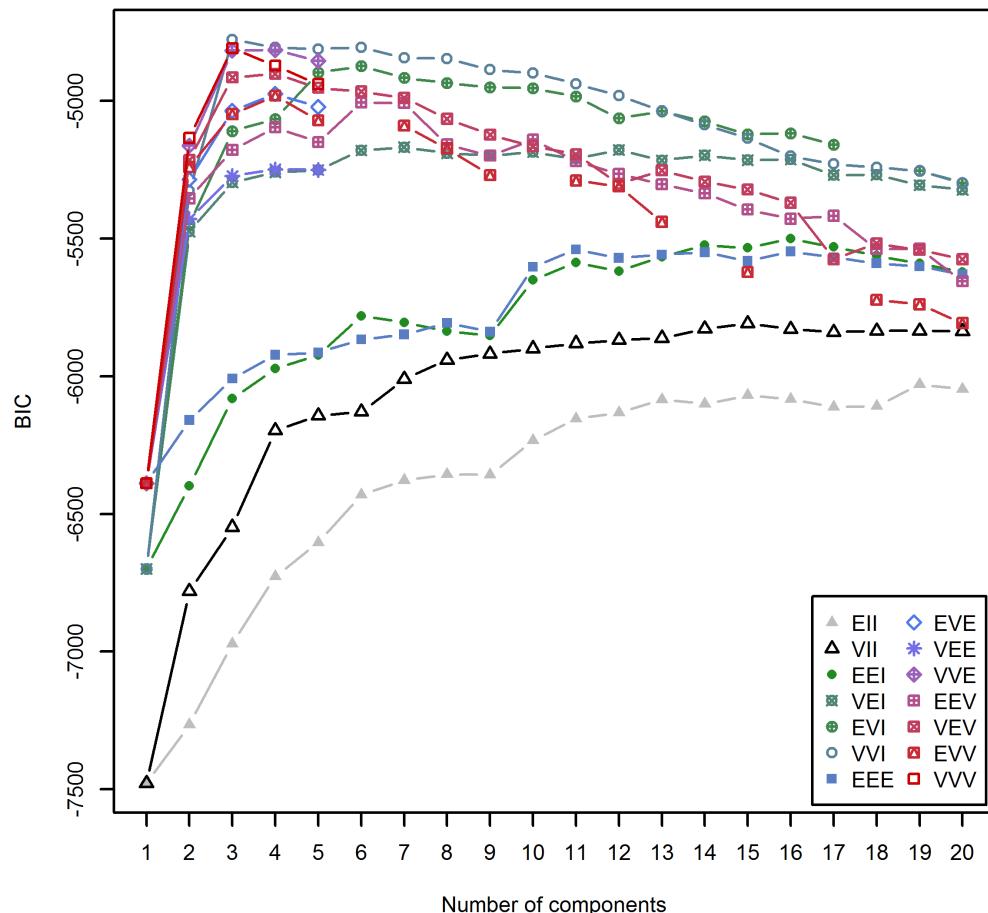


Figure 62.1: Model-Based Clustering BIC by model for `thyroid`

We can also visualize the fitted model by setting `what = "uncertainty"` (see Figure 62.2), `what = "density"` (see Figure 62.3) and `what = "classification"` (see Figure 62.4).

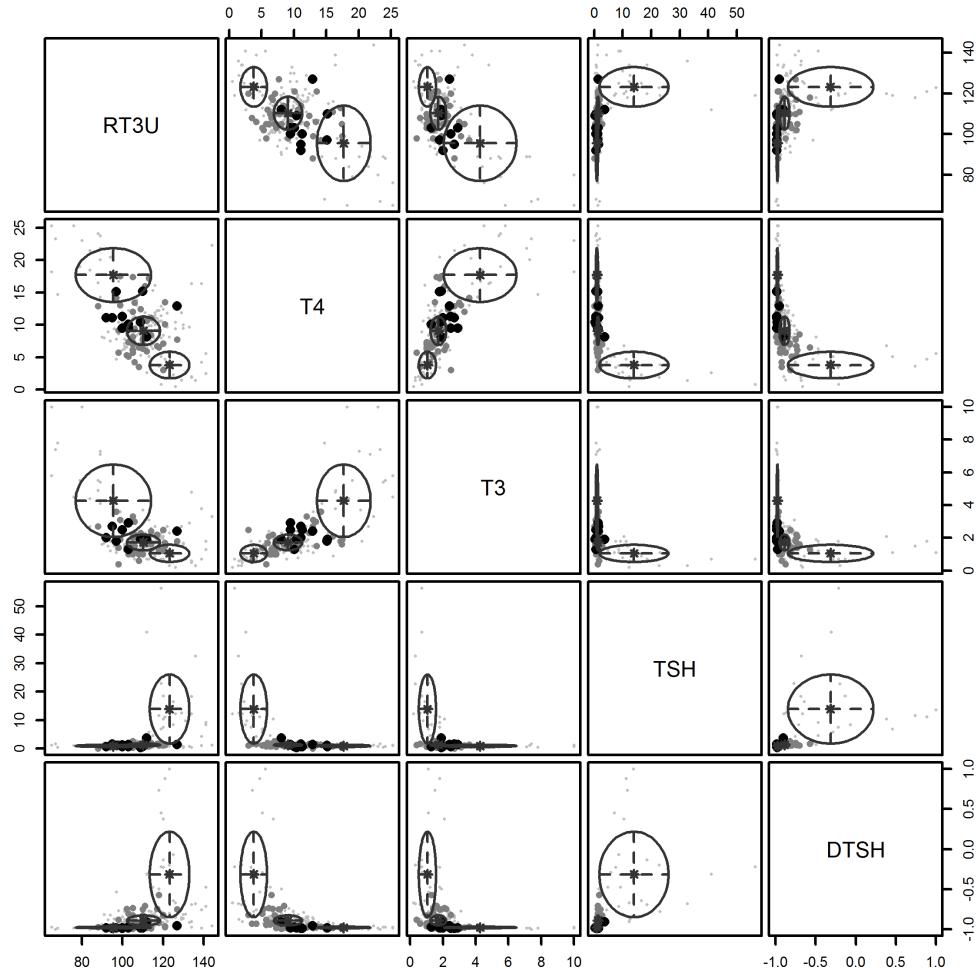


Figure 62.2: Model-Based Clustering plot with `what = "uncertainty"` using thyroid

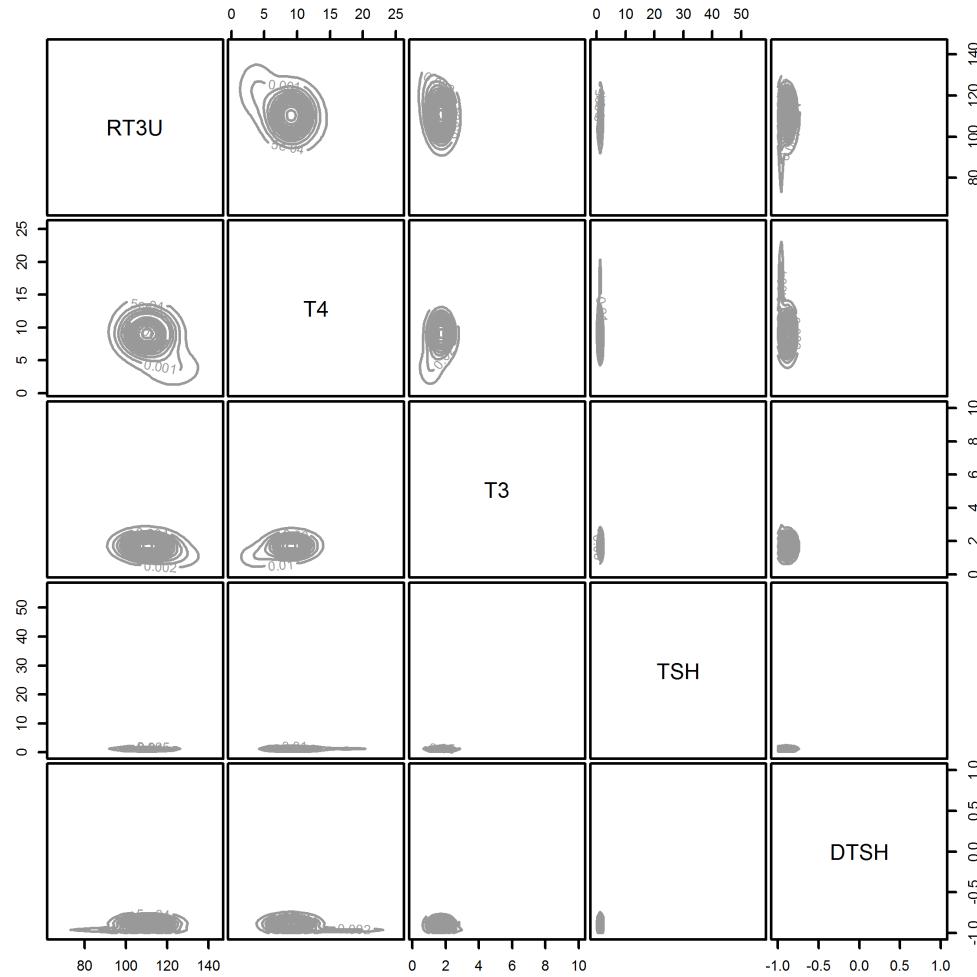


Figure 62.3: Model-Based Clustering plot with `what = "density"` using `thyroid`

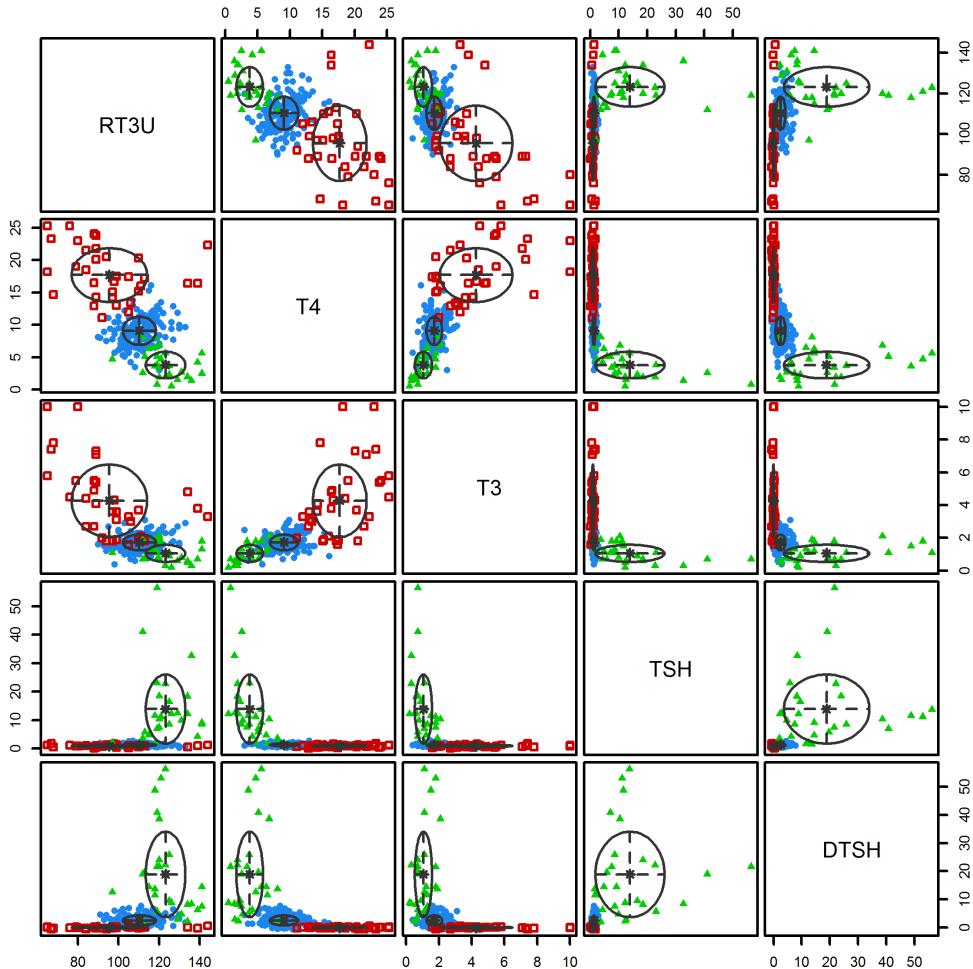


Figure 62.4: Model-Based Clustering plot with `what = "classification"` using `thyroid`

Finally, let's look at the uncertainty and classification pairwise plots using T3 and T4, see Figure 62.5.

```
> par(mfrow = c(1, 2))

> plot(fit, x[c(1,2),c(1,3)], 
       what = "uncertainty")

> plot(fit, x[c(1,2),c(1,3)], 
       what = "classification")
```

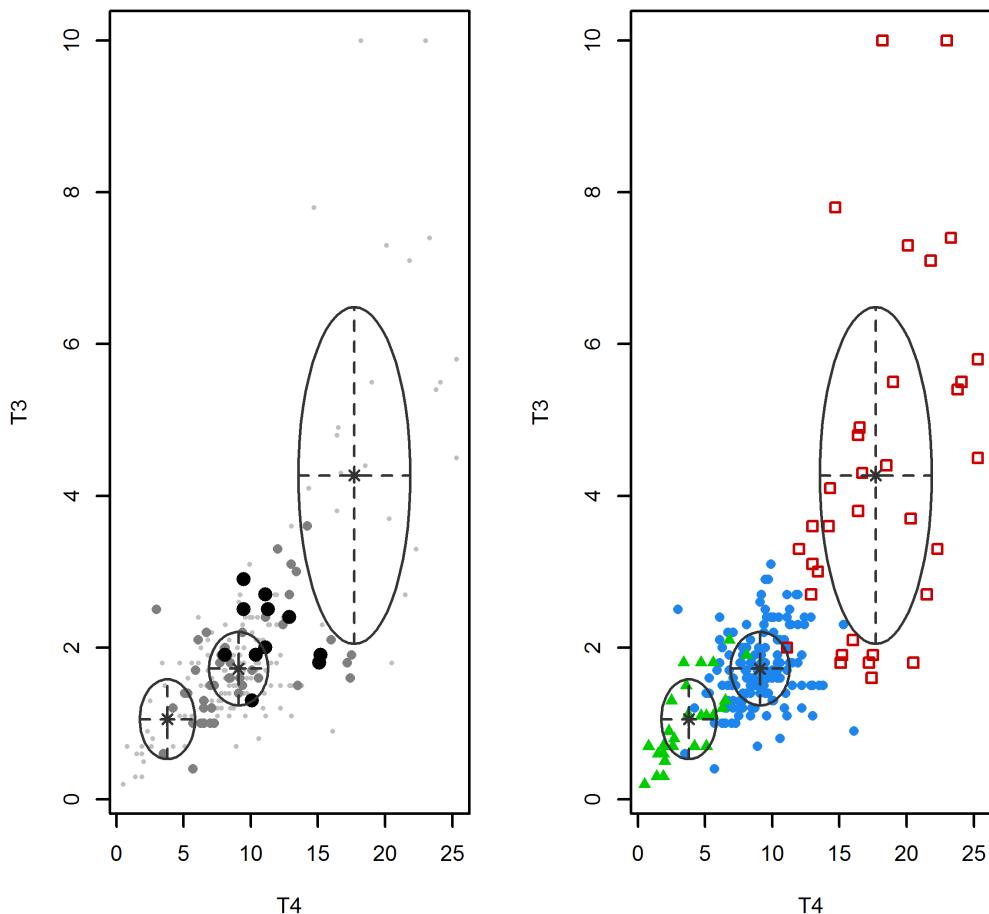
2.3.1 Coordinate Projection showing Uncertainty, **3.1 Coordinate Projection showing Classification**

Figure 62.5: Model based uncertainty and classification pairwise plots using T3 and T4

Technique 63

Clustering of Binary Variables

Clustering of Binary Variables is available in the `cluster` package using the function `mona`:

```
mona(x)
```

Key parameters include `x` the data set of binary attributes for which you wish to find clusters.

Step 1: Load Required Packages

We use the `locust` data frame from the package `bild`.

```
> require(cluster)
> data("locust", package="bild")
```

NOTE... 📈.

The `locust` data frame contains data on the effect of hunger on the locomotory behavior of 24 locust observed at 161 time points. It contains 3864 observations on the following attributes:

1. `id`: a numeric vector that identifies the number of the individual profile.
2. `move`: a numeric vector representing the response variable.
3. `sex`: a factor with levels 1 for "male" and 0 for "female".
4. `time`: a numeric vector that identifies the number of the time points observed. The time vector considered was obtained dividing (1:161) by 120 (number of observed periods in 1 hour).
5. `feed`: a factor with levels 0 "no" and 1 "yes".

Step 2: Prepare Data & Tweak Parameters

Because the `locust` data has a time dimension, our first task is to transform the data into a format that can be used by `mono`. First let's create a variable `x` that will hold the binary attributes.

```
> n<-nrow(locust)
> x<-seq(1,120)
> dim(x) <- c(24,5)
> colnames(x) <- colnames(locust)
```

We create a binary data set where we assign 1 if a move took place by the locust within 20 days, 0 otherwise. We do likewise with feeding. We begin by assigning zero to the move and feed attributes (i.e. `x[i,2]<-0` and `x[i,2]<-0`). Note that `locust[count,1]`, `locust[count,2]` and `locust[count,3]` contain `id`, `time` and `feed` respectively.

```
count=0
k=1
for(i in seq_along(seq(1,24)))
```

```
{  
x[i,2]<-0  
x[i,5]<-0  
  
for (k in seq(1:161))  
{  
count=count+1  
  
if (k==1) {  
x[i,1]<-locust[count,1]  
x[i,3]<-locust[count,3]  
x[i,4]<-locust[count,4]  
}  
  
if (locust[count,2]==1 && k<20)  
{ x[i,2]<-locust[count,2]}  
  
if (locust[count,5]==1 && k<20)  
{x[i,5]<-locust[count,5]  
}  
  
}  
  
k<-k+1  
}  
k =1  
}
```

Finally, we remove the `id` and `time` attributes.

```
> x<-x[, -1]  
> x<-x[, -3]
```

As a check, the contents of `x` should look similar to this:

```
> head(x, 6)  
    move sex feed  
[1,]    0    2    2  
[2,]    1    1    2  
[3,]    0    2    2  
[4,]    1    1    2  
[5,]    0    2    2  
[6,]    0    1    2
```

Step 3: Estimate and Assess the Model

Here is how to estimate the model and print out the associated bannerplot, see Figure 63.1.

```
fit<-mona(x)
plot(fit)
```

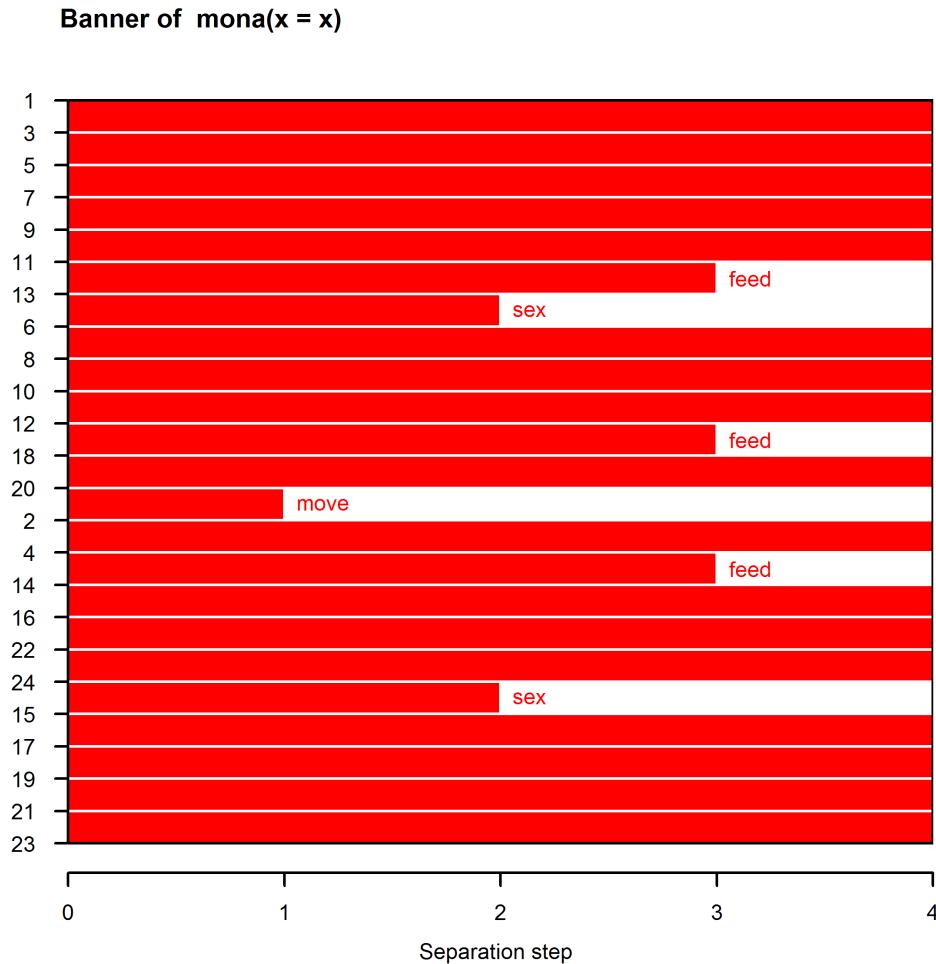


Figure 63.1: Clustering of Binary Variables bannerplot using locust

Technique 64

Affinity Propagation Clustering

Affinity propagation takes a given similarity matrix and simultaneously considers all data points as potential exemplars. Real-valued messages are then exchanged between data points until a high-quality set of exemplars and corresponding clusters emerges. It is available in the `apcluster` package using the function `apcluster`:

```
apcluster(s, x)
```

Key parameters include `x` the data set of attributes for which you wish to find clusters; and `s` the similarity matrix.

Step 1: Load Required Packages

We use the `Vehicle` data frame from the package `mlbench`. See page 23 for additional details on this data set.

```
> require(apcluster)
> data("Vehicle", package = "mlbench")
```

Step 2: Prepare Data & Tweak Parameters

We store the sample in the variable `x` and remove the Vehicle types (`Class`) attribute. .

```
> set.seed(98765)
> x <- Vehicle[, -19]
```

Step 3: Estimate and Assess the Model

We fit the model as follows, where r refers to the number of columns in **x**:

```
> fit <- apcluster(negDistMat(r=18), x)
```

The model determines the optimal number of clusters. To see the actual numbers use.

```
> length(fit@clusters)
[1] 5
```

So for this data set the algorithm identifies five clusters. We can visualize the results using the plot method as shown in Figure 64.1. Note we only plot 15 attributes.

```
> plot(fit, x[,1:15])
```

Finally, here is how to zoom in on a specific pairwise relationship. We choose **Circ** and **Sc.Var.Maxis**, as shown in Figure 64.2.

```
> plot(fit, x[,c(2,11)])
```

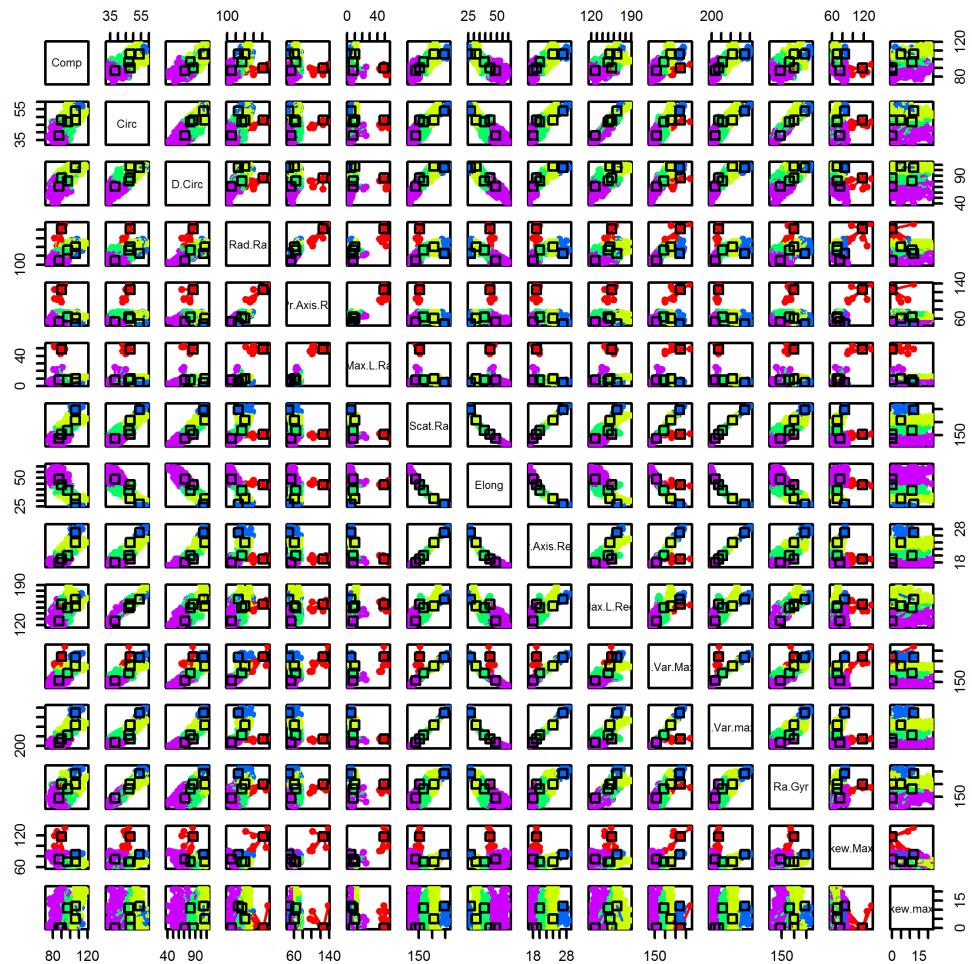


Figure 64.1: Affinity Propagation pairwise cluster plot using Vehicle

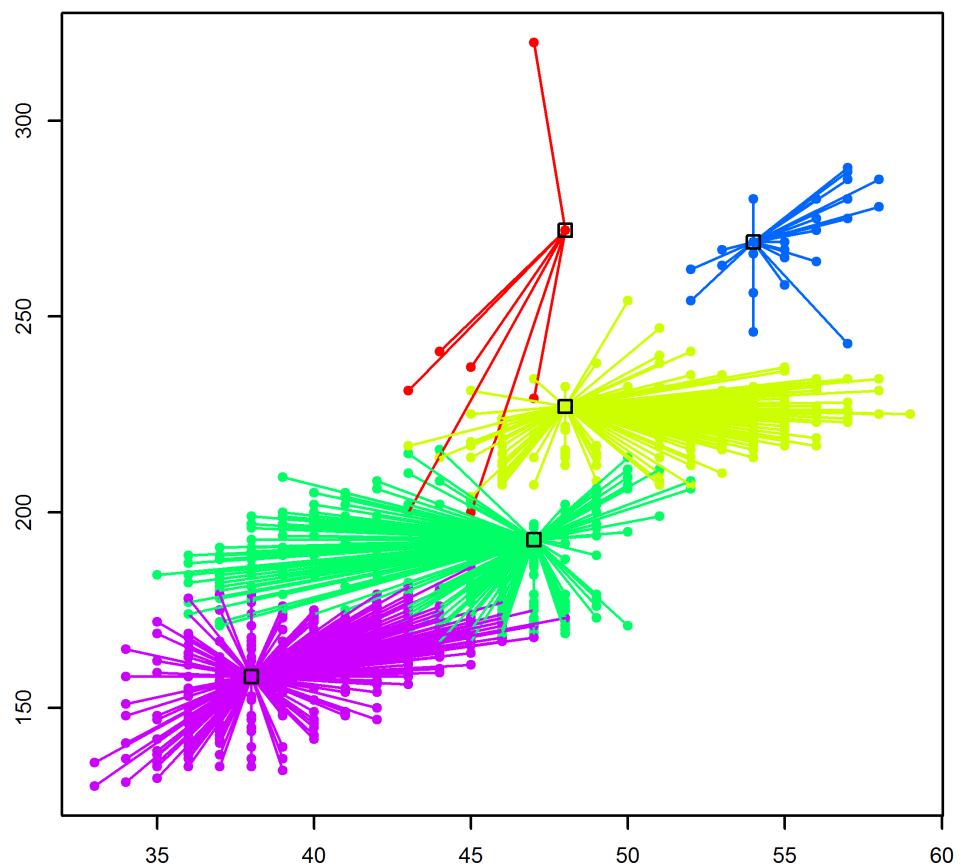


Figure 64.2: Affinity Propagation pairwise cluster plot between `Circ` and `Sc.Var.Maxis`

Technique 65

Exemplar-Based Agglomerative Clustering

Exemplar-Based Agglomerative Clustering is available in the `apcluster` package using the function `aggExCluster`:

```
aggExCluster(s, x)
```

Key parameters include `x` the data set of attributes for which you wish to find clusters; and `s` the similarity matrix.

Step 1: Load Required Packages

We use the `bodyfat` data frame from the package `TH.data`. See page 62 for additional details on this data set.

```
> require(apcluster)
> data("bodyfat", package="TH.data")
```

Step 2: Prepare Data & Tweak Parameters

We store the sample data in the variable `x`.

```
> set.seed(98765)
> x<-bodyfat
```

Step 3: Estimate and Assess the Model

We fit the model using a two-step approach. First we use affinity propagation to determine the number of clusters via the `apcluster` method. For additional details on affinity propagation and `apcluster` see page 455.

```
> fit1 <- apcluster(negDistMat(r=10), x)
```

Here are some of the details of the fitted model. It appears to have four clusters.

```
> fit1
```

```
APResult object
```

```
Number of samples      = 71
Number of iterations   = 152
Input preference       = -6.482591e+14
Sum of similarities     = -4.886508e+14
Sum of preferences      = -2.593037e+15
Net similarity          = -3.081687e+15
Number of clusters      = 4
```

Next we create a hierarchy of the four clusters using exemplar-based agglomerative clustering via the `aggExCluster` method.

```
> fit <- aggExCluster(x=fit1)
```

We can plot the resultant dendrogram as shown in Figure 65.1.

```
> plot(fit, showSamples=FALSE)
```

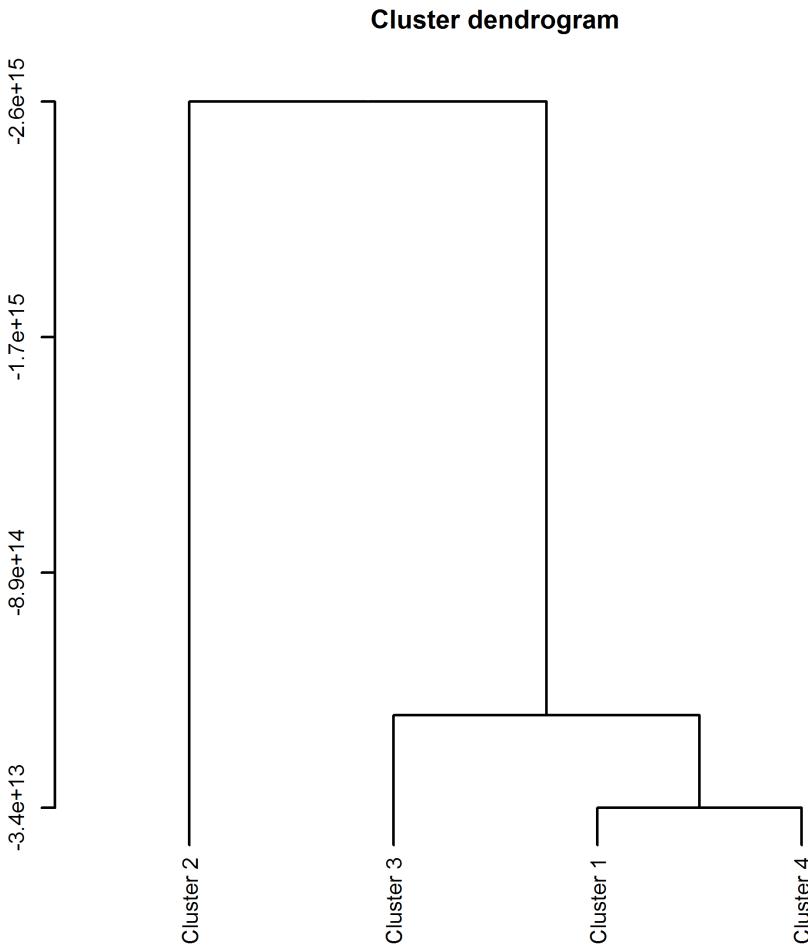


Figure 65.1: Exemplar-Based Agglomerative Clustering dendrogram using `bodyfat`

Finally, we zoom in on the pairwise cluster plots between `DEXfat` and `waistcirc`, see page 462. To do this we use a small `while` loop as follows:

```
> i=2
> par(mfrow = c(2, 2))
> while (i<=5)
{
  plot(fit, x[,c(2,3)], k=i,xlab="DEXfat", ylab=
    "waistcirc",main=c("Number Clusters ",i))
  i=i+1
}
```

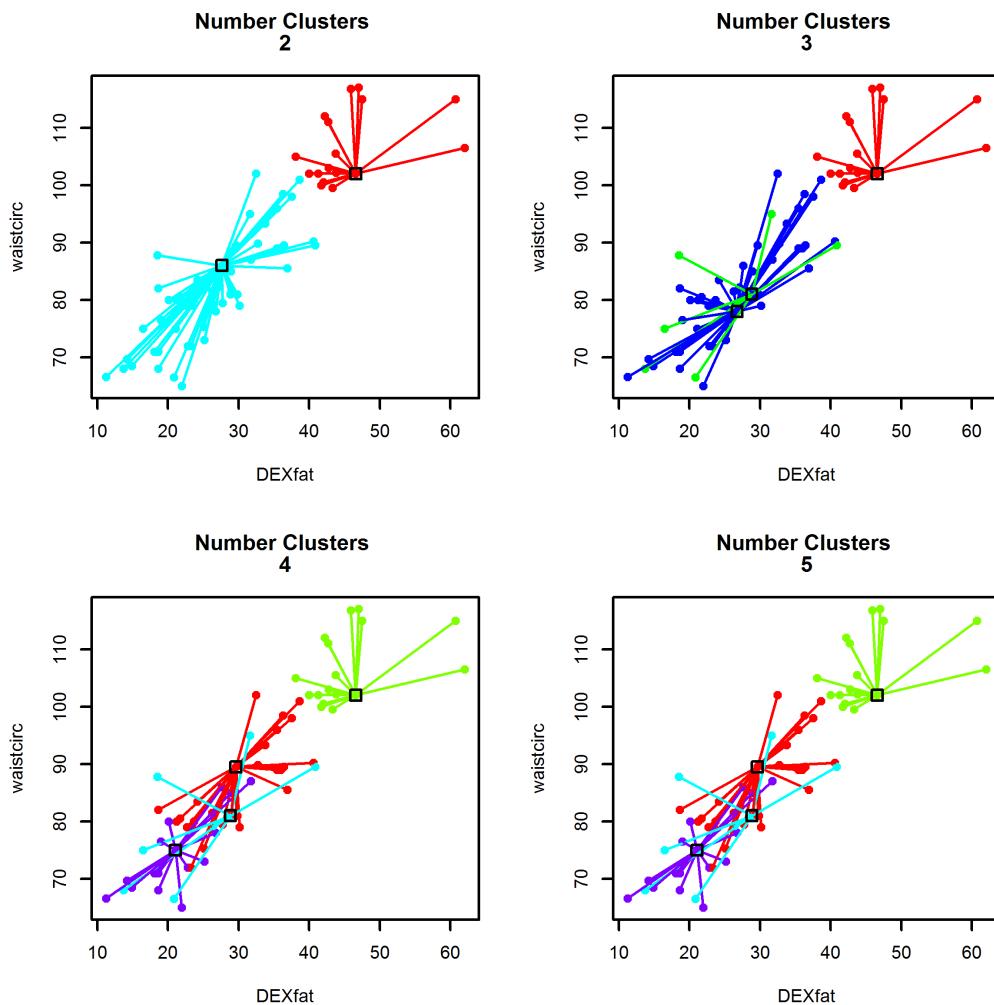


Figure 65.2: Exemplar-Based Agglomerative Clustering relationship between DEXfat and waistcirc

Technique 66

Bagged Clustering

During bagged clustering a partitioning cluster algorithm such as k-means is run repeatedly on bootstrap samples from the original data. The resulting cluster centers are then combined using hierarchical agglomerative cluster analysis (see page 384). The approach is available in the `e1071` package using the function `bclust`:

```
bclust(x, centers, base.centers, dist.method, ...)
```

Key parameters include `x` the data set of attributes for which you wish to find clusters; `centers` the number of clusters; `base.centers` the number of centers used in each repetition and `dist.method` the distance method used for hierarchical clustering.

Step 1: Load Required Packages

We use the `thyroid` data frame from the package `mclust`. See page 385 for additional details on this data set.

```
> require (e1071)
> data("thyroid", package="mclust")
```

Step 2: Prepare Data & Tweak Parameters

We store the standardized sample data in the variable `x`.

```
> x<-thyroid[, -1]
> x<-scale(x)
```

Step 3: Estimate and Assess the Model

We fit the model using the `bclust` method with 3 centers.

```
> fit <- bclust(x,
  centers=3,
  base.centers=5,
  dist.method="manhattan")
```

We can use the `plot` method to visualize the dendrogram. Figure 66.1 shows the result.

```
> plot(fit)
```

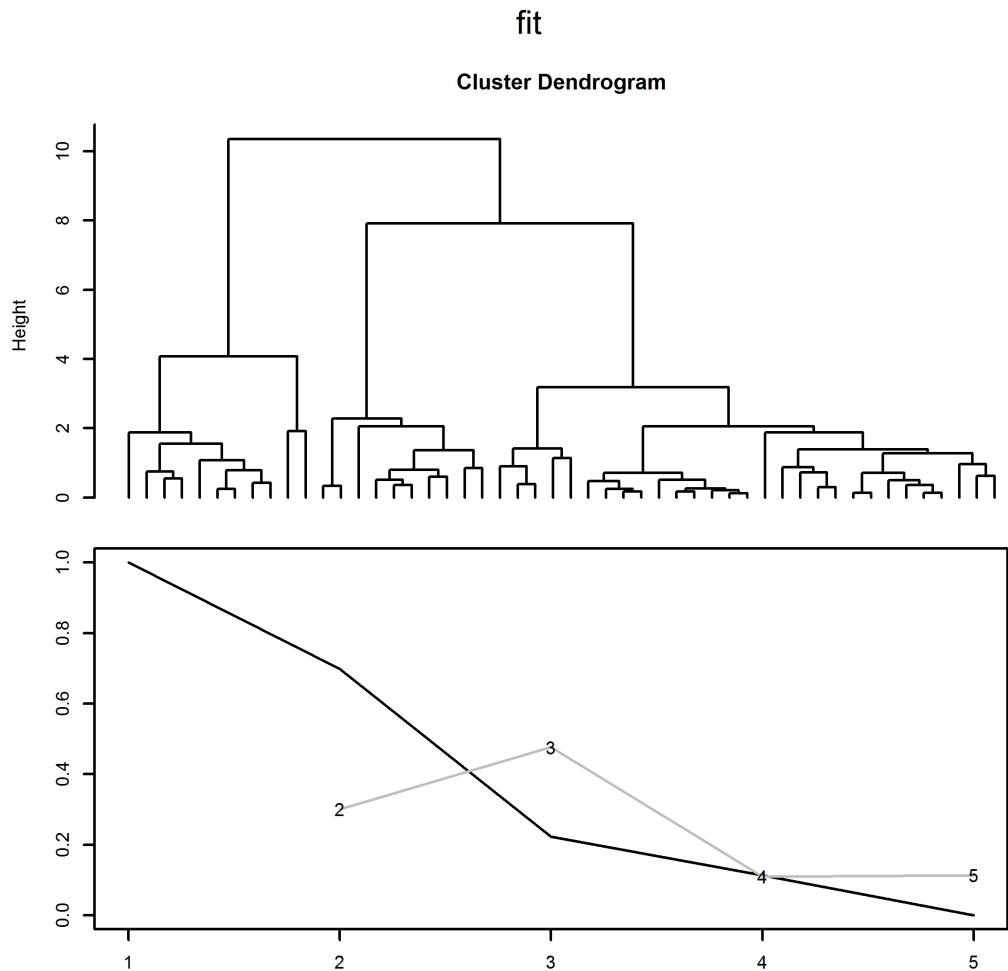


Figure 66.1: Bagged Clustering dendrogram (top) and scree style plot (bottom). Gray line is slope of black line.

We can also view a box plot of `fit` as follows, see Figure 66.2.

```
> boxplot(fit)
```

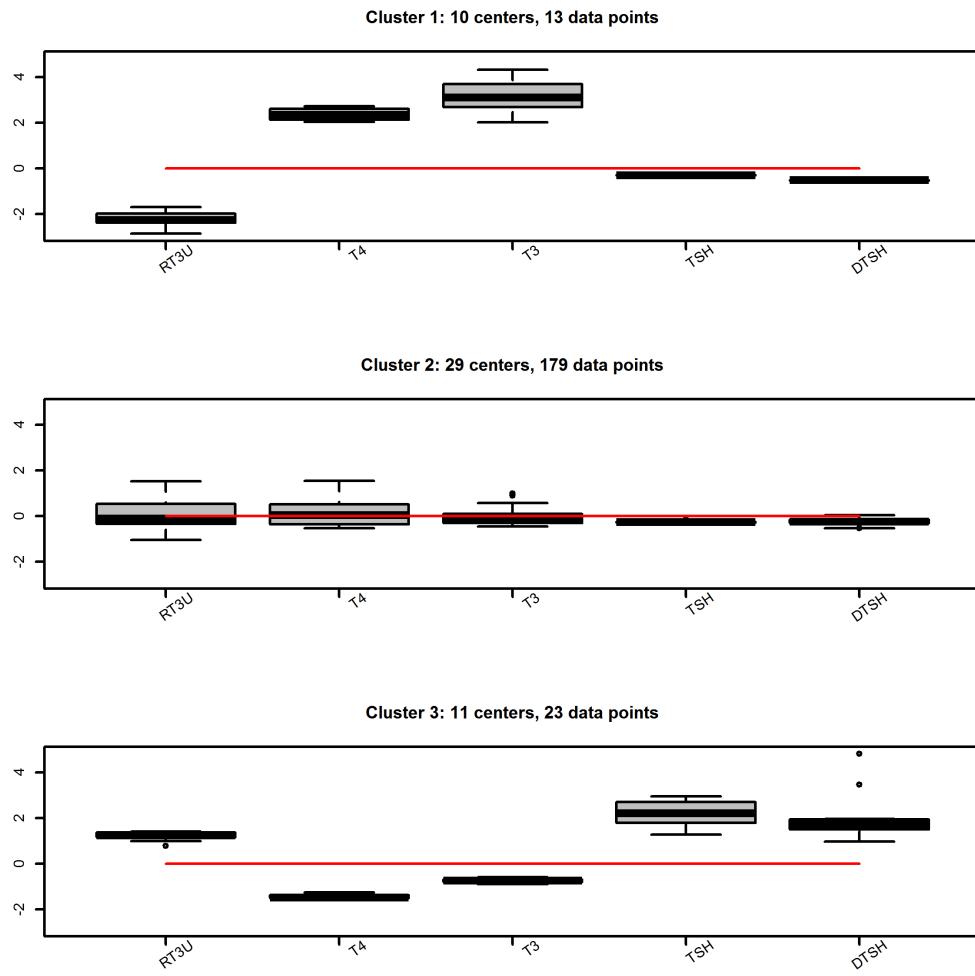


Figure 66.2: Bagged Clustering boxplots

We can also view **boxplots** by attribute as shown in Figure 66.3.

```
> boxplot(fit, bycluster=FALSE)
```

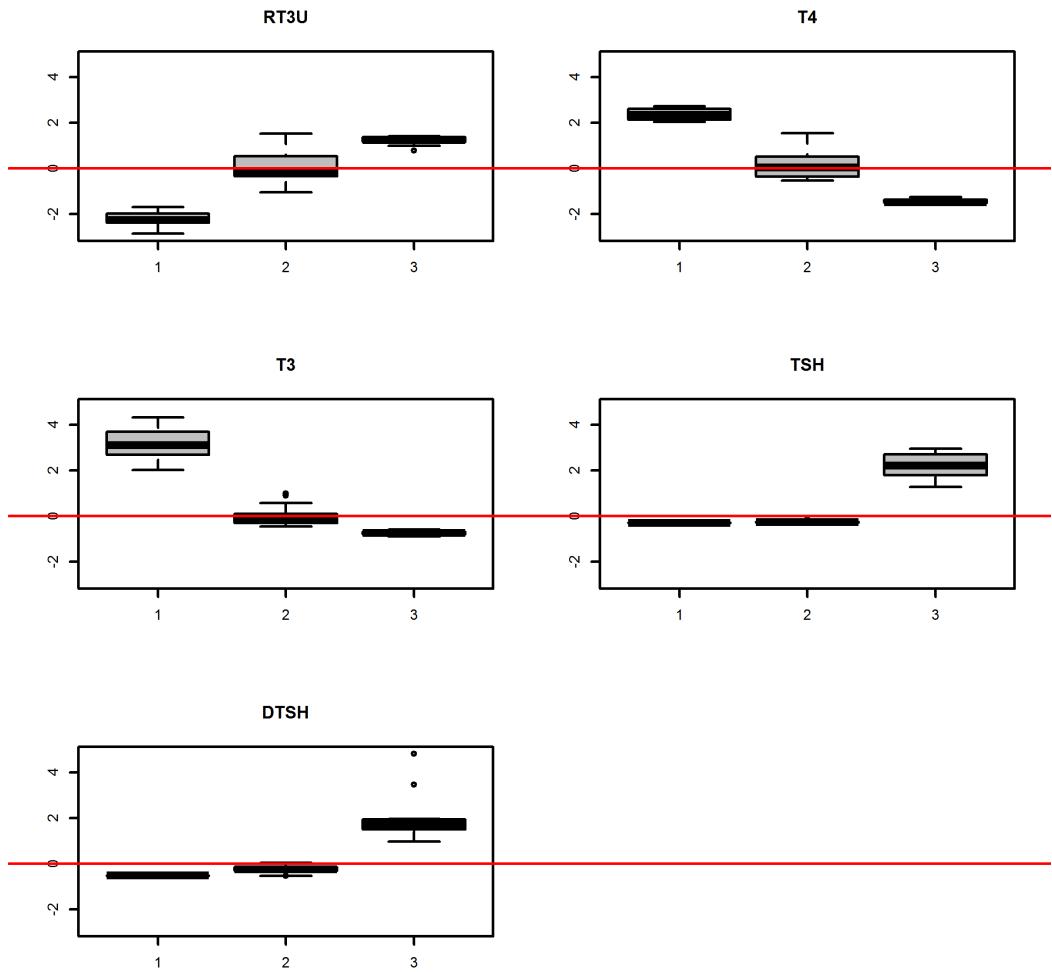


Figure 66.3: Bagged Clustering boxplots by attribute

Notes

⁸⁹Musmeci, Nicoló, Tomaso Aste, and Tiziana Di Matteo. "Relation between financial market structure and the real economy: comparison between clustering methods." (2015): e0116201.

⁹⁰The Industry Classification Benchmark (ICB) is a definitive system categorizing over 70,000 companies and 75,000 securities worldwide, enabling the comparison of companies across four levels of classification and national boundaries. For more information see <http://www.icbenchmark.com/>

⁹¹Since correlations between stocks and stock market sectors change over time, the time period over which the analysis is carried out may also impact the number of clusters recovered by a specific technique.

⁹²Takahashi, Nobuaki, Makio Kashino, and Naoyuki Hironaka. "Structure of rat ultrasonic vocalizations and its relevance to behavior." PloS one 5.11 (2010): e14115-e14115.

⁹³Wright JM, Gourdon JC, Clarke PB (2010) Identification of multiple call categories within the rich repertoire of adult rat 50-kHz ultrasonic vocalizations: effects of amphetamine and social context. Psychopharmacology 211: 1–13.

⁹⁴Kim, Tae-Bum, et al. "Identification of asthma clusters in two independent Korean adult asthma cohorts." The European respiratory journal 41.6 (2013): 1308-1314.

⁹⁵FEV₁ is the volume exhaled during the first second of a forced expiratory activity.

⁹⁶In recent years, there has been growing interest in using perfusion imaging in the initial diagnosis and management of many conditions. Magnetic resonance imaging (MRI) uses a powerful magnetic field, radio frequency pulses and a computer to produce high resolution images of organs, soft tissues and bone.

⁹⁷See for example:

1. Karonen JO, Liu Y, Vanninen RL, et al. Combined perfusion- and diffusion-weighted MR imaging in acute ischemic stroke during the 1st week: a longitudinal study. Radiology. 2000;217:886–894;
2. Rother J, Jonetz-Mentzel L, Fiala A, et al. Hemodynamic assessment of acute stroke using dynamic single-slice computed tomographic perfusion imaging. Arch Neurol. 2000;57:1161–1166;
3. Nabavi DG, Cenic A, Henderson S, Gelb AW, Lee TY. Perfusion mapping using computed tomography allows accurate prediction of cerebral infarction in experimental brain ischemia. Stroke. 2001;32:175–183;
4. Eastwood JD, Lev MH, Azhari T, et al. CT perfusion scanning with deconvolution analysis: pilot study in patients with acute MCA stroke. Radiology;
5. Kealey SM, Loving VA, Delong DM, Eastwood JD. User-defined vascular input function curves: influence on mean perfusion parameter values and signal- to-noise ratio. Radiology. 2004;231:587–593.

⁹⁸The computation of these parametric maps is often performed as a semi automated process. The observer selects an appropriate artery to represent the arterial input function and a vein to represent the venous function. From these arterial and venous time-attenuation curves, the observer then determines the pre and post enhancement cutoff values for the calculation of the perfusion parameters. During data acquisition it is essential to select slice locations in the brain that contain a major intracranial artery to represent the AIF.

⁹⁹Yin, Jiandong, et al. "Comparison of K-Means and fuzzy c-Means algorithm performance for automated determination of the arterial input function." PloS one 9.2 (2014): e85884.

¹⁰⁰Murase K, Kikuchi K, Miki H, Shimizu T, Ikezoe J (2001) Determination of arterial input function using fuzzy clustering for quantification of cerebral blood flow with dynamic susceptibility contrast-enhanced MR imaging. J Magn Reson Imaging 13: 797–806.

¹⁰¹See Mardia et al. (1979). Multivariate Analysis. Academic Press.

¹⁰²For example see:

1. Rudman P (1967) The causes of natural durability in timber. Pt. XXI The antitermitic activity of some fatty acids, esters and alcohols, Holzforschung 21(1): 24. 12;
2. Rudman P, Gay F (1963) Causes of natural durability in timber. X. Deterrent properties of some three-ringed carboxylic and heterocyclic substances to the subterranean termite, Nasutitermes exitiosus. C.S.I.R.O. Div. Forest Prod., Melbourne, Holzforschung 17: 2125. 13;
3. Lukmandaru G, Takahashi K (2008) Variation in the natural termite resistance of teak (*Tectona grandis* Linn. fil.) wood as a function of tree age, Annals of Forest Science 65: 708–708.

¹⁰³See for example:

1. Lenz M (1994) Nourishment and evolution of insect societies, Westview Press, Boulder and Oxford and IBH Publ., New Delhi, chapter Food resources, colony growth and caste development in wood feeding termites, 159–209. 23;
2. Evans TA, Lai JCS, Toledano E, McDowall L, Rakotonarivo S et al. (2005) Termites assess wood size by using vibration signals, Proceedings of the National Academy of Science, 102(10): 3732–3737. 24;
3. Evans TA, Inta R, Lai JCS, Prueger S, Foo NW et al. (2009) Termites eavesdrop to avoid competitors, Proceedings of the Royal Society B: Biological Sciences 276(1675): 4035–4041.

¹⁰⁴Oberst, Sebastian, Theodore A. Evans, and Joseph CS Lai. "Novel Method for Pairing Wood Samples in Choice Tests." (2014): e88835.

¹⁰⁵Todeschini, Roberto, and Viviana Consonni. Handbook of molecular descriptors. Vol. 11. John Wiley & Sons, 2008.

¹⁰⁶Dehmer, Matthias, Frank Emmert-Streib, and Shailesh Tripathi. "Large-scale evaluation of molecular descriptors by means of clustering." (2013): e83956.

¹⁰⁷For more details see:

1. Dehmer M, Varmuza K, Borgert S, Emmert-Streib F (2009) On entropy-based molecular descriptors: Statistical analysis of real and synthetic chemical structures. Journal of Chemical Information and Modeling 49: 1655–1663. 21;
2. Dehmer M, Grabner M, Varmuza K (2012) Information indices with high discriminative power for graphs. PLoS ONE 7: e31214.

¹⁰⁸For further details see Gower, J. C. (1971) A general coefficient of similarity and some of its properties, Biometrics 27, 857–874.

¹⁰⁹Further details at <ftp://ftp.ics.uci.edu/pub/machine-learning-databases/thyroid-disease/new-thyroid.names>

¹¹⁰For further details see - Murtagh, Fionn, and Pierre Legendre. "Ward's hierarchical agglomerative clustering method: Which algorithms implement ward's criterion?." *Journal of Classification* 31.3 (2014): 274-295.

¹¹¹Kaufman, Leonard, and Peter J. Rousseeuw. *Finding groups in data: an introduction to cluster analysis*. Vol. 344. John Wiley & Sons, 2009.

¹¹²Gower, J. C. (1971) A general coefficient of similarity and some of its properties, *Biometrics* 27, 857–874.

¹¹³For further details see Struyf, Anja, Mia Hubert, and Peter J. Rousseeuw. "Integrating robust clustering techniques in S-PLUS." *Computational Statistics & Data Analysis* 26.1 (1997): 17-37.

¹¹⁴For further details see Madsen, M. (1976) Statistical analysis of multiple contingency tables. Two examples. *Scand. J. Statist.* 3, 97–106.

Part VII

Boosting

The Basic Idea

Boosting is a powerful supervised classification learning concept. It combines the performance of many “weak” classifiers to produce a powerful committee. A weak classifier is only required to be better than chance. It can therefore be very simple and computationally inexpensive. The basic idea is to iteratively apply simple classifiers and to combine their solutions to obtain a better prediction result. If classifiers misclassify some data, train another copy of it mainly on this misclassified part with the hope that it will develop a better solution. Thus the algorithm increasingly focuses on new strategies for classifying difficult observations.

Here is how it works in a nutshell:

1. A boosting algorithm manipulates the underlying training data by iteratively re-weighting the observations so that at every iteration the classifier finds a new solution from the data.
2. Higher accuracy is achieved by increasing the importance of “difficult” observations so that observations that were misclassified receive higher weights. This forces the algorithm to focus on those observations that are increasingly difficult to classify.
3. In the final step, all previous results of the classifier are combined into a prediction committee where the weights of better performing solutions are increased via an iteration-specific coefficient.
4. The resulting weighted majority vote selects the class most often chosen by the classifier while taking the error rate in each iteration into account.

The Power of the Boost

Boosting is one of the best techniques in the data scientist toolkit. Why? Because it often yields the best predictive models and can often be relied upon to produce satisfactory results. Academic studies and applied studies have found similar finding. Here we cite just two:

- Bauer and Kohavi¹¹⁵ performed an extensive comparison of boosting with several other competitors on 14 data-sets. They found boosting outperformed all other algorithms. They concluded “*For learning tasks where comprehensibility is not crucial, voting methods are extremely useful, and we expect to see them used significantly more than they are today.*”
- Friedman, Hastie, and Tibshirani¹¹⁶ using eight data-sets compare a range of boosting techniques with the very popular classification and regression tree. They find all the boosting methods outperform.

NOTE... ↗

During boosting the target function is optimized with some implicit penalization whose impact varies with the number of boosting iterations. The lack of an explicit penalty term in the target function is the main difference between boosting and other popular penalization methods such as Lasso¹¹⁷.

Practical Applications

NOTE... ↗

The first boosting technique was developed by Robert Schapire¹¹⁸ working out of the MIT Laboratory for Computer Science. His research article “*Strength of Weak Learnability*” theorem showed that a weak base classifier always improve its performance by training two additional classifiers on filtered versions of the classification data stream. The author observes “*A method is described for converting a weak learning algorithm into one that achieves arbitrarily high accuracy. This construction may have practical applications as a tool for efficiently converting a mediocre learning algorithm into one that performs extremely well.*” He was right! boosting is now used in a wide range of practical applications.

Reverberation Suppression

Reverberation suppression is a critical problem in sonar communications. This is because as an acoustic signal is radiated degradation occurs due to reflection from the surface, bottom, and via the volume of water. Cheepurupalli et al¹¹⁹ study of the propagation of sound in water.

One popular solution is to use the empirical mode decomposition (EMD) algorithm¹²⁰ as a filtering technique. A noise corrupted signal is applied to EMD and intrinsic mode functions are generated. The key is to separate the signal from the noise. It turns out that noisy intrinsic mode functions are high frequency component signals whilst signal-led intrinsic mode functions are low frequency component signals. The selection of the appropriate intrinsic mode functions, which are used for signal reconstruction, is often done manually¹²¹.

The researchers use Ada Boost to automatically classify “noise” & “signal” intrinsic mode functions. Over the signal to noise ratio from -10 dB to 10 dB.

The results were very encouraging as they found that combining Ada Boost with EMD increases the likelihood of correct detection. The researchers conclude “*...that the reconstruction of the chirp signal even at low input SNR [signal to noise] conditions is achieved with the use of Ada Boost based EMD as a de-noising technique.*”

Cardiotocography

Fetal state (normal; suspect; pathologic) is assessed by Karabulut and Ibrikci¹²² using Ada Boost combined with six individual machine learning algorithms (Naive Bayes, Radial Basis Function Network, Bayesian Network, Support Vector Machine, Neural Network and C4.5 Decision Tree).

The researchers use the publicly available cardiotocography data set¹²³ which includes a total of 2126 samples of which the fetal state is normal in 1655 cases, suspicious in 295 and pathologic in 176. The estimated Ada Boost confusion matrix is shown in Table 23. Note that the error rate from this table is around 5%.

		Predicted values		
		Normal	Suspicious	Pathologic
Actual values	Normal	1622	26	7
	Suspicious	55	236	4
	Pathologic	7	7	162

Table 23: Confusion matrix of Karabulut and Ibrikci

Stock Market Trading

Creamer and Freund¹²⁴ develop a multi-stock automated trading system that relies in part on model based boosting. Their trading system consists of a logitboost algorithm, an expert weighting algorithm and a risk management overlay.

Their sample consisted of 100 stocks chosen at random from the S&P 500 index using daily data over a four year period. The researchers used 540

trading days for the training set, 100 trading days for the validation set and a 411 trading days for the test set.

Four variants of their trading system are compared to a buy and hold strategy. This is a common investment strategy and involves buying a portfolio of stocks and holding them without selling for the duration of the investment period. Transaction costs are assumed to vary from \$0 to \$0.003. Creamer and Freund report all four variants of their trading system outperformed the buy and hold strategy.

Vehicle Logo Recognition

Sam et al¹²⁵ consider the problem of automatic vehicle logo recognition. The researchers select the modest adaboost algorithm. A total of 500 vehicle images were used in the training procedure. The detection of vehicle logo was carried out by sliding a sub-window across the image at multiple scales and locations. A total of 200 images were used in the test set with 184 images recognized successfully, see Table 24. This implies a misclassification error rate of around 9%.

Manufacturer	Correct	Mistaken
Audi	20	0
BMW	16	4
Honda	19	1
KIA	17	3
Mazda	18	2
Mitsubishi	20	0
Nissan	17	3
Suzuki	18	2
Toyota	19	1
Volkswagen	20	0
Total	184	16

Table 24: Logo recognition rate reported by Sam et al

Basketball Player Detection

Markoski et al¹²⁶ investigate player detection during basketball games using the gentle adaboost algorithm. A total of 6000 positive examples that contain the players entire body and 6000 positive examples of players upper body only were used to train the algorithm.

The researchers observe for images that contain players whole body the algorithm was unable to reduce the level of false positives below 50%. In other words the flipping a coin would have been more accurate! However, a set of testing images using players upper body obtained an accuracy of 70.5%. The researchers observe the application of the gentle adaboost algorithm for detecting players upper body results in a relatively large number of false positive observations.

Magnetoencephalography

Magnetoencephalography (MEG) is used to study how stimulus features are processed in the human brain. Takiguchi et al¹²⁷ use an Ada Boost algorithm to find the sensor area contributing to the accurate discrimination of vowels.

Four volunteers with normal hearing were recruited for the experiment. Two distinct Japanese sounds were delivered to the subject's right ear. On hearing the sound the volunteer was asked to press a reaction key. MEG amplitudes were measured from 61 pairs of sensors¹²⁸.

The ada-Boost algorithm was applied to every latency range with the classification decision made at each time instance. The researchers observe that the Ada-Boost classification accuracy first increased as a function of time, reaching a maximum value of 91.0% in the latency range between 50 and 150 ms. These results outperformed their pre-stimulus baseline.

Binary Boosting

NOTE... ↗

A weak classifier $h(x)$ is slightly better than random chance (50%) at guessing which class an object belongs in. A strong classifier has a high probability ($>95\%$) of choosing correctly. Decision trees (see Part I) are often used as the basis for weak classifiers.

Classical binary boosting is founded on the discrete Ada Boost algorithm in which a sequentially generated weighted set of weak base classifiers are combined to form an overall strong classifier. Ada Boost was the first adaptive boosting algorithm. It automatically adjusts its parameters to the data based on actual performance at each iteration.

How Ada Boost Works

Given a set of training feature vectors $x_i (i = 1, 2, \dots, N)$ a target which represents the binary class $y_i \in \{-1, +1\}$ the algorithm attempts to find the optimal classification by making the individual error ε_m at iteration m as small as possible given a iteration specific distribution of weights on the features.

Incorrectly classified observations receive more weight in the next iteration. Correctly classified observations receive less weight in the next iteration. The weights at iteration m are calculated using the iteration specific learning coefficient α_m multiplied by a classification function $\eta(h_m(x))$.

As the algorithm iterates it focuses more weight on misclassified objects and attempts to correctly classify them in the next iteration. In this way, classifiers that are accurate predictors of the training data receive more weight, whereas, classifiers that are poor predictors receive less weight. The procedure is repeated until a predefined performance requirement is satisfied.

NOTE... ↗

Differences in the nature of classical boosting algorithms are often driven by the functional form of α_m and $\eta(h_m(x))$. For example:

1. Ada Boost.M1: $\alpha_m = 0.5 \log\left(\frac{1-\varepsilon_m}{\varepsilon_m}\right)$ with $\eta(x) = \text{sign}(x)$.
2. Real Ada Boost and Real L₂ set $\eta(p) = \log(\frac{p}{1-p})$, $p \in [0, 1]$.
3. Gentle Ada Boost and Gentle L₂ Boost: $\eta(x) = x$
4. Discrete L₂, Real L₂ and Gentle L₂ use a logistic loss function as opposed to the exponential loss function used by Real Ada Boost, Ada Boost.M1 and Gentle Ada Boost.

Technique 67

Ada Boost.M1

The Ada Boost.M1 algorithm (also known as Discrete Ada Boost) uses discrete boosting with a exponential loss function¹²⁹. In this algorithm $\eta(x) = \text{sign}(x)$ and $\alpha_m = 0.5 \log\left(\frac{1-\varepsilon_m}{\varepsilon_m}\right)$. It can be run using the package **ada** with the **ada** function:

```
ada(z ~., data , iter , loss = "e", type = "discrete  
" , . . . )
```

Key parameters include **iter**, the number of iterations used to estimate the model; **z** the data-frame of binary classes; **data** the data set of attributes with which you wish to train the model.

NOTE... ↗

Ada Boost is often used with simple decision trees as weak base classifiers. This can result in significantly improved performance over a single decision tree¹³⁰. The package **ada** creates a classification model as an ensemble of **rpart** trees. It therefore uses the **rpart** library as its engine (see page 4 and page 272).

Step 1: Load Required Packages

We build our Ada Boost.M1 model using **Sonar** a data frame in the **mlbench** package:

```
> library (ada)  
> library(mlbench)  
> data(Sonar)
```

NOTE... ↗

Sonar contains 208 observations on sonar returns collected from a metal cylinder and a cylindrical shaped rock positioned on a sandy ocean floor¹³¹. Returns were collected at a range of 10 meters and obtained from the cylinder at aspect angles spanning 90° and from the rock at aspect angles spanning 180°. In total 61 variables were collected, all numerical with the exception of the sonar return (111 cylinder returns and 97 rock returns) which are nominal. The label associated with each record is contained in the letter "R" if the object is a rock and "M" if it is metal.

Step 2: Prepare Data & Tweak Parameters

To view the classification for the 95th to 105th record type:

```
> Sonar$Class[95:105]  
[1] R R R M M M M M M M M  
Levels: M R
```

The 95th to 97th observation are labeled "R" for rock, whilst the 98th to 105th observations are labeled "M" for metal. The **ada** package uses the **rpart** function to generate decision trees. So we need to supply an **rpart.control** object to the model:

```
> default <- rpart.control(cp = -1 , maxdepth = 2 ,  
  minsplit = 0)
```

☛ PRACTITIONER TIP ☛

The **maxdepth** parameter is raised to power of 2. In the code that follows we set **maxdepth = 2** which is equivalent to $2^2=4$.

We use 157 of the 208 observations to train the model:

```
> set.seed(107)  
> n=nrow(Sonar)  
> indtrain <- sample(1:n, 157, FALSE)  
> train <- data[indtrain,]  
> test <- data[-indtrain,]
```

Since `Sonar$Class` is non-numeric we change it to a numeric scale. This will be useful later when we plot the training and test data:

```
> z<-Sonar$Class  
> z<-as.numeric(z)
```

Next we create a new data frame combining the numeric variable `z` with the original data in `Sonar`. This is followed by a little tidying up by removing `Class` (which contained the “M” & “R” labels) from the data frame:

```
> data<-(cbind(z, Sonar))  
> data$Class <- NULL
```

Step 3: Estimate Model

Now we are ready to run the Ada Boost.M1 model on the training sample. We choose to perform 50 boosting iterations by setting the parameter `iter = 50`:

```
> set.seed(107)  
> output_train <- ada(z ~ ., data = train, iter = 50,  
    loss = "e", type = "discrete", control = default)
```

Take a look at the output:

```
> output_train  
Call:  
ada(z ~ ., data = train, iter = 50, loss = "e", type  
= "discrete",  
control = default)  
  
Loss: exponential Method: discrete Iteration: 50  
  
Final Confusion Matrix for Data:  
          Final Prediction  
True value  1   2  
      1 80   2  
      2   3 72  
  
Train Error: 0.032
```

```
Out-Of-Bag Error: 0.045 iteration= 45

Additional Estimates of number of iterations:

train.err1 train.kap1
        44         44
```

Notice the output gives the training error, confusion matrix and three estimates of the number of iterations. In this example, you could use the Out-Of-Bag estimate for 45 iterations, training error estimate or the kappa error estimate for 44 iterations. As you can see, the model achieved low error rates. The training error is 3.2%. The Out-Of-Bag error rate is around 4.5%.

Step 4: Assess Model Performance

We use the `addtest` function to evaluate the testing set without having to refit the model:

```
> set.seed(107)

> output_train<-addtest(x=output_train, test.x=test
[, -1], test.y=test[, 1])

> output_train
Call:
ada(z ~ ., data = train, iter = 50, loss = "e", type
= "discrete",
control = default)

Loss: exponential Method: discrete      Iteration: 50

Final Confusion Matrix for Data:
          Final Prediction
True value  1   2
           1 80   2
           2   3 72

Train Error: 0.032

Out-Of-Bag Error: 0.045 iteration= 45
```

```
Additional Estimates of number of iterations:
```

```
train.err1 train.kap1 test,errs2 test.kaps2  
        44          44          32          32
```

Notice the estimate of iterations for the training error and kapa declined from 44 (for training) to 32 for the test sample.

It can be instructive to plot both training and test error results by iteration on one chart:

```
> plot(output_train,test=TRUE)
```

The resulting plot, shown in Figure 67.1, shows that the training error steadily decreases across iterations. This shows that boosting can learn the features in the data set. The testing error also declines somewhat across iterations, although not as dramatically as for the training error.

It is sometimes helpful to examine the performance at a specific iteration. Here is what happened at iteration 25 for `train`:

```
> summary(output_train,n.iter=25)  
Call:  
ada(z ~ ., data = train, iter = 50, loss = "e", type  
= "discrete",  
control = default)
```

```
Loss: exponential Method: discrete Iteration: 25
```

Training Results

```
Accuracy: 0.955 Kappa: 0.911
```

Testing Results

```
Accuracy: 0.824 Kappa: 0.654
```

The accuracy of the training and testing samples is above 0.80. To assess the overall performance enter:

```
> summary(output_train)  
Call:  
ada(z ~ ., data = train, iter = 50, loss = "e", type  
= "discrete",  
control = default)
```

```
Loss: exponential Method: discrete Iteration: 50
```

Training Results

Accuracy: 0.968 Kappa: 0.936

Testing Results

Accuracy: 0.804 Kappa: 0.613

Notice that accuracy is in excess of 80% for both test and training datasets. Kappa declines from 94% in training to 61% for the test data set.

☛ **PRACTITIONER TIP** ☛

In very many circumstances you may find your training set unbalanced in the sense that one class has very many more observations than the other. Ada Boost will tend to focus on learning the larger set with resultant low errors. Of course the low error is related to the focus on the larger class. The Kappa coefficient¹³² provides an alternative measure of absolute classification error which adjusts for class imbalances. As with the correlation coefficient, higher values indicate a stronger fit.

In Figure 67.2 the relative importance of each variable is plotted. This can be obtained by typing:

```
>varplot(output_train)
```

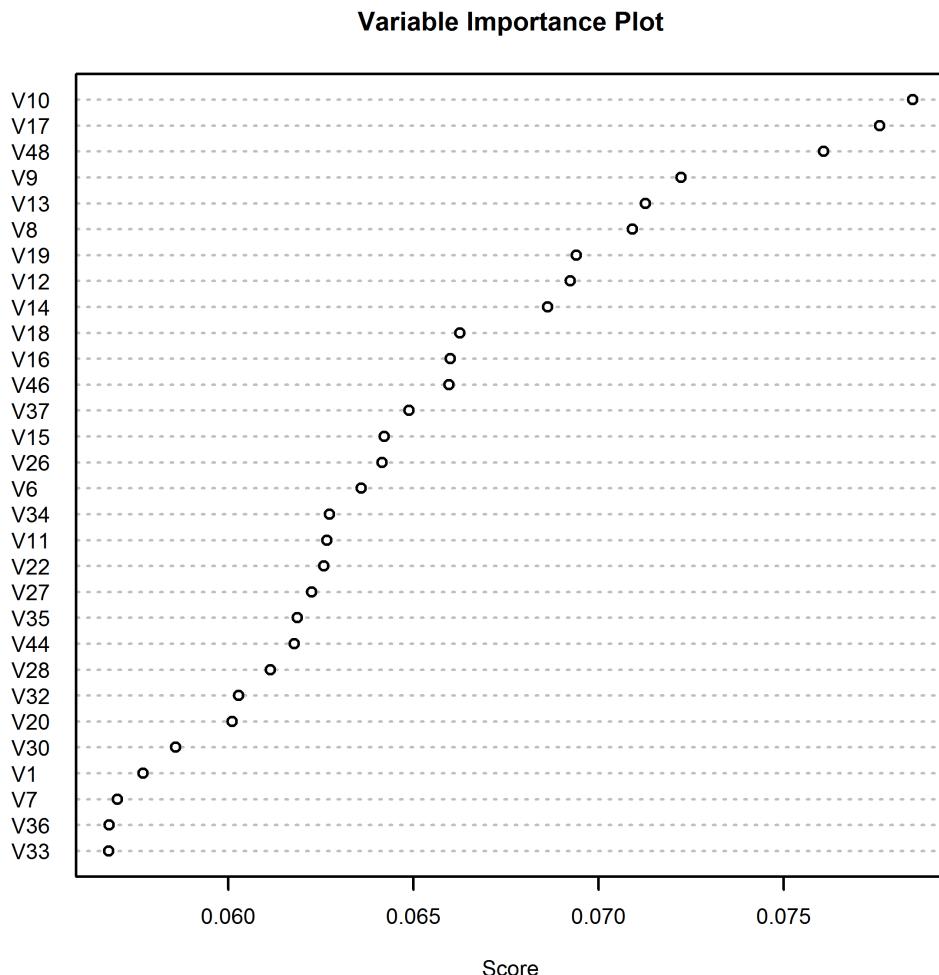


Figure 67.2: Variable importance plot for Ada Boost using Sonar data set

The largest five individual scores can be printed out by entering:

```
> scores<-varplot(output_train,plot.it=FALSE, type="scores")  
  
> round(scores[1:5],3)  
V10    V17    V48    V9    V13  
0.078  0.078  0.076  0.072  0.071
```

Step 5: Make Predictions

Next we predict using the fitted model and the test sample. This can be easily achieved by:

```
> pred<-predict(output_train,test[,-1],type="both")
```

A summary of the class predictions and the associated plot provides performance information:

```
> summary(pred$class)
 1   2
23  28
```

```
> plot(pred$class)
```

The model classifies 23 observations into class 1 (recall “M” if object metal) and 28 observations into class 2 (“R” if the object is a rock). This is reflected in Figure 67.3.

A nice feature of this model is the ability to see the probabilities of an observation and the class assignment. For example, the first observation has associated probabilities and class assignment:

```
> pred$probs [[1,1]]
[1] 0.5709708
> pred$probs [[1,2]]
[1] 0.4290292
> pred$class [1]
[1] 1
Levels: 1 2
```

The probability of the observation belonging to class 1 is 57% and to class 2 around 43% and therefore the predicted class is class 1. The second and third observations can be assessed in a similar fashion:

```
> pred$probs [[2,1]]
[1] 0.07414183

> pred$probs [[2,2]]
[1] 0.9258582

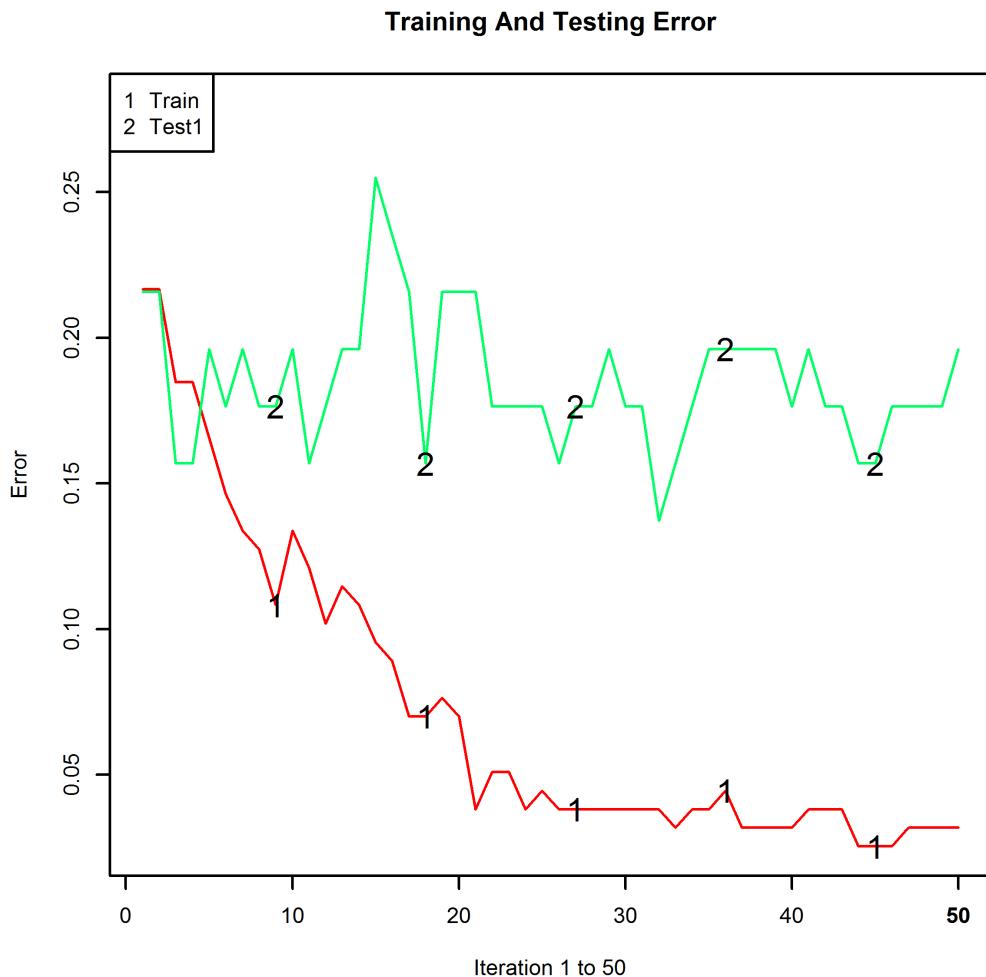
> pred$class [2]
[1] 2
Levels: 1 2
```

```
> pred$probs [[3 ,1]]  
[1] 0.7416495
```

```
> pred$probs [[3 ,2]]  
[1] 0.2583505
```

```
> pred$class [3]  
[1] 1  
Levels: 1 2
```

Notice that the second observation, is assigned to class 2 with an associated probability of approximately 93%; and the third observation is assigned to class 1 with an associated probability of approximately 74%.

Figure 67.1: Error by iteration for `train` and `test`

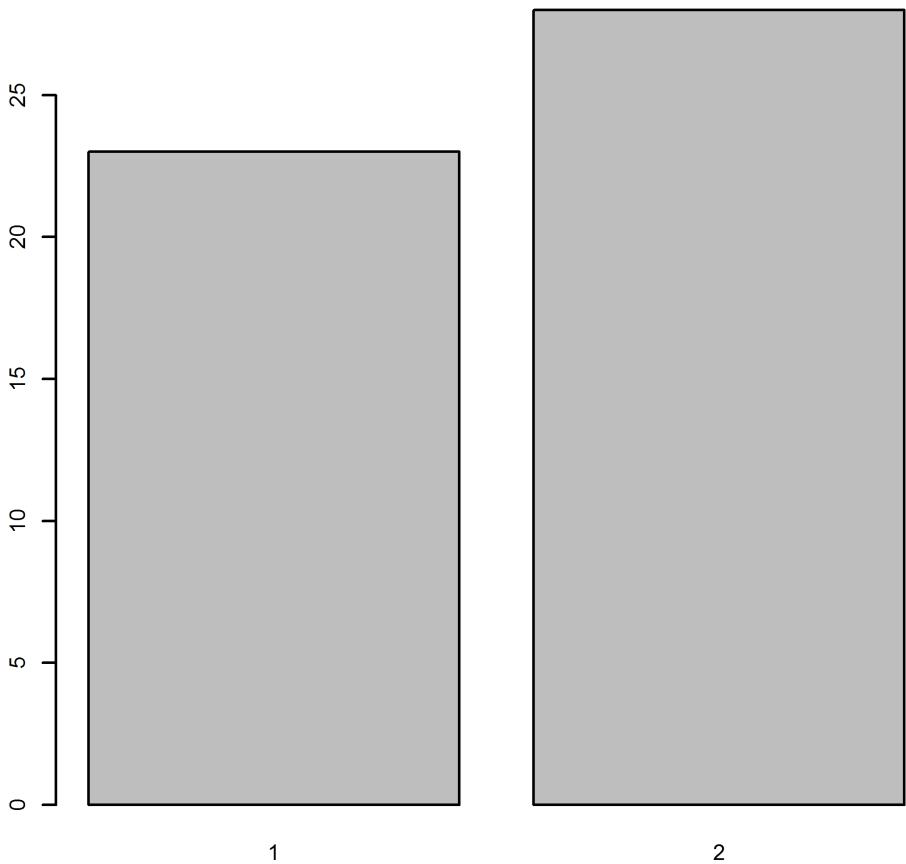


Figure 67.3: Model Predictions for class 1 ("M") and class 2 ("R")

Technique 68

Real Ada Boost

The Real Ada Boost algorithm uses discrete boosting with an exponential loss function and $\eta(p) = \log(\frac{p}{1-p})$, $p \in [0, 1]$. It can be run using the package **ada** with the **ada** function:

```
ada(z ~., data , iter , loss = "e", type = "real",
    bag.frac=1,...)
```

Key parameters include **iter**, the number of iterations used to estimate the model; **z** the data-frame of binary classes; **data** the data set of attributes with which you wish to train the model. To perform Stochastic Gradient boosting you set the **bag.frac** argument less than 1(default is **bag.frac** = 0.5). Pure ϵ -boosting only happens if **bag.frac** = 1.

Steps 1-2 are outlined beginning on page 481.

Step 3 & 4: Estimate Model & Assess Performance

We choose to perform 50 boosting iterations by setting the parameter **iter** = 50.

NOTE... ↗

The main control to avoid over fitting in boosting algorithms is the stopping iteration. A very high number of iterations may favor over-fitting. However, stopping the algorithm too early might lead to poorer prediction on new data. In practice over fitting appears to be less of a risk than under-fitting and there is considerable evidence that Ada Boost is quite resistant to overfitting¹³³.

```
> set.seed(107)
> output_train <- ada(z ~.,
  data = train, iter = 50,
  loss = "e", type = "real",
  bag.frac=1, control = default)

> output_train
Call:
ada(z ~ ., data = train, iter = 50, loss = "e", type
    = "real",
    bag.frac = 1, control = default)

Loss: exponential Method: real      Iteration: 50

Final Confusion Matrix for Data:
          Final Prediction
True value   1     2
           1 82   0
           2   0 75

Train Error: 0

Out-Of-Bag Error: 0 iteration= 6

Additional Estimates of number of iterations:

train.err1 train.kap1
        27         27
```

In this case the model perfectly fits the training data and the training error is zero.

☛ PRACTITIONER TIP ☛

Although the out-of-bag error rate is also zero it is of little value here because there are no subsamples in pure ϵ -boosting (we set `bag.frac = 1`).

We next fit the test data using Stochastic Gradient boosting with `bag.frac` at its default value. The overall error rate remains a modest 1.3% with an Out-Of-Bag Error of 4.5%.

```
> set.seed(107)
> output_SB <- ada(z ~ ., data = train,
iter = 50, loss = "e", type = "real",
bag.frac=0.5,control = default)

> output_SB
Call:
ada(z ~ ., data = train, iter = 50, loss = "e", type
= "real",
bag.frac = 0.5, control = default)

Loss: exponential Method: real      Iteration: 50

Final Confusion Matrix for Data:
          Final Prediction
True value   1    2
           1 82  0
           2  2 73

Train Error: 0.013

Out-Of-Bag Error: 0.045 iteration= 46

Additional Estimates of number of iterations:
```

```
train.err1 train.kap1
        49         49
```

To assess variable importance we enter:

```
> scores<-varplot(output_train,
plot.it=FALSE,
type="scores")

> round(scores[1:5],3)
V46     V47     V48     V35     V20
0.087  0.081  0.078  0.076  0.072
```

The top five variables differ somewhat from those obtained with Ada Boost.M1 (see page 487).

Step 5: Make Predictions

Now we fit the data to the test data set, make predictions and compare the results to the actual values observed in the sample using the command `table(pred)` and `table (test$z)` (recall `test$z` contains the class values)

```
> set.seed(107)
> output_test <- ada(z ~., data = test, iter = 50,
  loss = "e", type = "real", control = default)

> pred<-predict(output_test,newdata=test)

> table(pred)
pred
 1  2
29 22

> table (test$z)

 1  2
29 22
```

The model fits the observations perfectly!

Technique 69

Gentle Ada Boost

The Gentle Ada Boost algorithm uses discrete boosting with an exponential loss function and $\eta(x) = x$. It can be run using the package `ada` with the `ada` function:

```
ada(z ~., data = train, iter = 50, loss = "e", type  
= "gentle", nu=0.01, control = default,...)
```

Key parameters include `iter`, the number of iterations used to estimate the model; `z` the data-frame of binary classes; `data` the data set of attributes with which you wish to train the model, `nu` is the shrinkage parameter for boosting. Steps 1-2 are outlined beginning on page 481.

NOTE... ↗

The idea behind the shrinkage parameter `nu` is to slow down learning and reduce the likelihood of over-fitting. Smaller learning rates (such as `nu < 0.1`) often produce dramatic improvements in a model's generalization ability over gradient boosting without shrinking (`nu = 1`)¹³⁴. However, small values of `nu` increase computational time by increasing the number of iterations required to learn.

Step 3: Estimate Model

We choose to perform 50 boosting iterations by setting the parameter `iter = 50`.

```
> set.seed(107)
```

```
> output_train <- ada(z ~ ., data = train,
iter = 50, loss = "e",
type = "gentle",
nu=0.01,control = default)

> output_train
Call:
ada(z ~ ., data = train, iter = 50, loss = "e", type
= "gentle",
nu = 0.01, control = default)

Loss: exponential Method: gentle Iteration: 50

Final Confusion Matrix for Data:
      Final Prediction
True value   1   2
      1 72   8
      2 13  64

Train Error: 0.134

Out-Of-Bag Error: 0.121 iteration= 28

Additional Estimates of number of iterations:

train.err1 train.kap1
      5          23
```

The training error is around 13% with an Out-Of-Bag Error of 12%. Let's see if we can do better by using the parameter `bag.shift=TRUE` to estimate a ensemble shifted towards bagging.

```
> output_bag<- ada(z ~ ., data = train, iter = 50,
loss = "e", type = "gentle", nu=0.01, bag.shift=
TRUE,control = default)

> output_bag
Call:
ada(z ~ ., data = train, iter = 50, loss = "e", type
= "gentle",
nu = 0.01, bag.shift = TRUE, control = default)
```

```
Loss: exponential Method: gentle Iteration: 50

Final Confusion Matrix for Data:
      Final Prediction
True value   1   2
      1 70 10
      2 20 57

Train Error: 0.191

Out-Of-Bag Error: 0.178 iteration= 17

Additional Estimates of number of iterations:

train.err1 train.kap1
      9          9
```

The training error has risen to 19%. We continue our analysis using both models.

Step 4: Assess Model Performance

We fit the test data to both models. First out boosted model.

```
> set.seed(107)
> output_test <- ada(z ~ ., data = test, iter = 50,
  loss = "e", type = "gentle", nu=0.01, control =
  default)

> output_test
Call:
ada(z ~ ., data = test, iter = 50, loss = "e", type
= "gentle",
  nu = 0.01, control = default)
```

```
Loss: exponential Method: gentle Iteration: 50
```

```
Final Confusion Matrix for Data:
      Final Prediction
True value   1   2
      1 31 0
      2 3 17
```

```
Train Error: 0.059
```

```
Out-Of-Bag Error: 0.059 iteration= 10
```

```
Additional Estimates of number of iterations:
```

```
train.err1 train.kap1  
8 8
```

The training error rate is certainly lower than for the test set at 5.9% and the number of iterations for kappa is only 8. The ensemble shifted towards bagging results are as follows.

```
> output_test_bag <- ada(z ~ ., data = train, iter =  
  50, loss = "e", type = "gentle", nu=0.01, bag.  
  shift=TRUE, control = default)  
  
> output_test_bag  
Call:  
ada(z ~ ., data = train, iter = 50, loss = "e", type  
  = "gentle",  
  nu = 0.01, bag.shift = TRUE, control = default)
```

```
Loss: exponential Method: gentle Iteration: 50
```

```
Final Confusion Matrix for Data:
```

```
          Final Prediction
```

True value	1	2
1	38	42
2	11	66

```
Train Error: 0.338
```

```
Out-Of-Bag Error: 0.166 iteration= 44
```

```
Additional Estimates of number of iterations:
```

```
train.err1 train.kap1  
35 46
```

The training error is higher at 34%! The Out-Of-Bag Error remains stubbornly in the 17% range.

Step 5: Make Predictions

We use the test sample to make predictions for both models and compare the results to the known values.

```
> pred<-predict(output_test,newdata=test)
> table(pred)
pred
 1  2
34 17

> pred<-predict(output_test_bag,newdata=test)
> table(pred)
pred
 1  2
31 20

> table (test$z)

 1  2
31 20
```

Note that `test$z` contains the actual values. It turns out that the results for both models are within an acceptable range. The out of bag model predicts the classes perfectly.

Technique 70

Discrete L₂ Boost

Discrete L₂ Boost uses discrete boosting with a logistic loss function. It can be run using the package `ada` with the `ada` function:

```
ada(z ~., data , iter , loss = "l", type = "discrete  
", control,...)
```

Key parameters include `iter`, the number of iterations used to estimate the model; `z` the data-frame of binary classes; `data` the data set of attributes with which you wish to train the model.

Step 1: Load Required Packages

We will build the model using the `soldat` data frame contained in the `ada` package. The objective is to use the Discrete L₂ Boost model to predict the relationship between the structural descriptors (aka features or attributes) and solubility/ insolubility:

```
> library(ada)  
> data("soldat")
```

NOTE... ↗

The `soldat` data frame consists of 5631 compounds tested to assess their ability to dissolve in a water/solvent mixture. Compounds were categorized as either insoluble ($n=3493$) or soluble ($n=2138$). Then, for each compound, 72 continuous, noisy structural descriptors were computed. Notice that one of the descriptors contain 787 misclassified values.

Step 2: Prepare Data & Tweak Parameters

We partition the data into a training set containing 60% of the observations, a test set containing 30% of the observations and a validation set containing 10% of the observations.

```
> n<-nrow(soldat)
> set.seed(103)
> random_sample<-(1:n)
> train_sample<-ceiling(n*0.6)
> test_sample <-ceiling(n*0.3)
> valid_sample <-ceiling(n*0.1)
> train<-soldat[random_sample[1:train_sample],]
> test<-soldat[random_sample[(train_sample+1):(train_
sample+test_sample)],]
> valid<-soldat[random_sample[(test_sample+train_
sample+1):n],]
```

Wow that is a lot of typing! Better check we have the right number of observations (5631):

```
> nrow(train)+nrow(test)+nrow(valid)
[1] 5631
```

Now we set the `rpart.control`. The `maxdepth` controls the maximum depth of any node in the final tree, with the root node counted as depth 0. It is raised to power of 2; We set the max depth of $2^4 = 16$.

```
> default<-rpart.control(cp=-1,maxdepth=4,maxcompete
=5,minsplit=0)
```

Step 3: Estimate Model

Now we are ready to run the model on the training sample. We choose 50 boosting iterations by setting the parameter `iter = 50`:

```
> set.seed(127)
> output_test <- ada(y ~., data = train, iter = 50,
  loss = "l", type = "discrete",control = default)
> output_test
Call:
ada(y ~ ., data = train, iter = 50, loss = "l", type
= "discrete",
control = default)
```

```
Loss: logistic Method: discrete    Iteration: 50

Final Confusion Matrix for Data:
      Final Prediction
True value   -1      1
      -1  1918  189
      1   461   811

Train Error: 0.192

Out-Of-Bag Error: 0.207 iteration= 50

Additional Estimates of number of iterations:

train.err1 train.kap1
        49          49
```

Notice the output gives the training error, confusion matrix and three estimates of the number of iterations. The training error estimate is 19% and the Out-Of-Bag Error is 20.7%. Of course you could increase the number of iterations to attempt to push down both error rates. However, for illustrative purposes we will stick with this version of the model.

Step 4: Assess Model Performance

☛ PRACTITIONER TIP ☛

Notice that the actual binary classes (coded -1 and +1) are in column 73 in the test, train and validation datasets. Therefore we add the line `test.x=valid[,-73],test.y=valid[,73]` to the `addtest` function.

We use the `addtest` function to evaluate the testing set without needing to refit the model:

```
> output_test<-addtest(output_test,test.x=test
[,-73],test.y=test[,73])

> summary(output_test)
```

```
Call:  
ada(y ~ ., data = train, iter = 50, loss = "l", type  
= "discrete",  
control = default)
```

```
Loss: logistic Method: discrete Iteration: 50
```

Training Results

```
Accuracy: 0.808 Kappa: 0.572
```

Testing Results

```
Accuracy: 0.759 Kappa: 0.467
```

We see a decline in both the model accuracy and kappa from the training to the test set. Although an accuracy of 75.9% may be sufficient, we would like to see a slightly higher kappa.

Next we plot the training error and kappa by iteration for training and test, see Figure 70.1.

```
> plot(output_test, TRUE, TRUE)
```

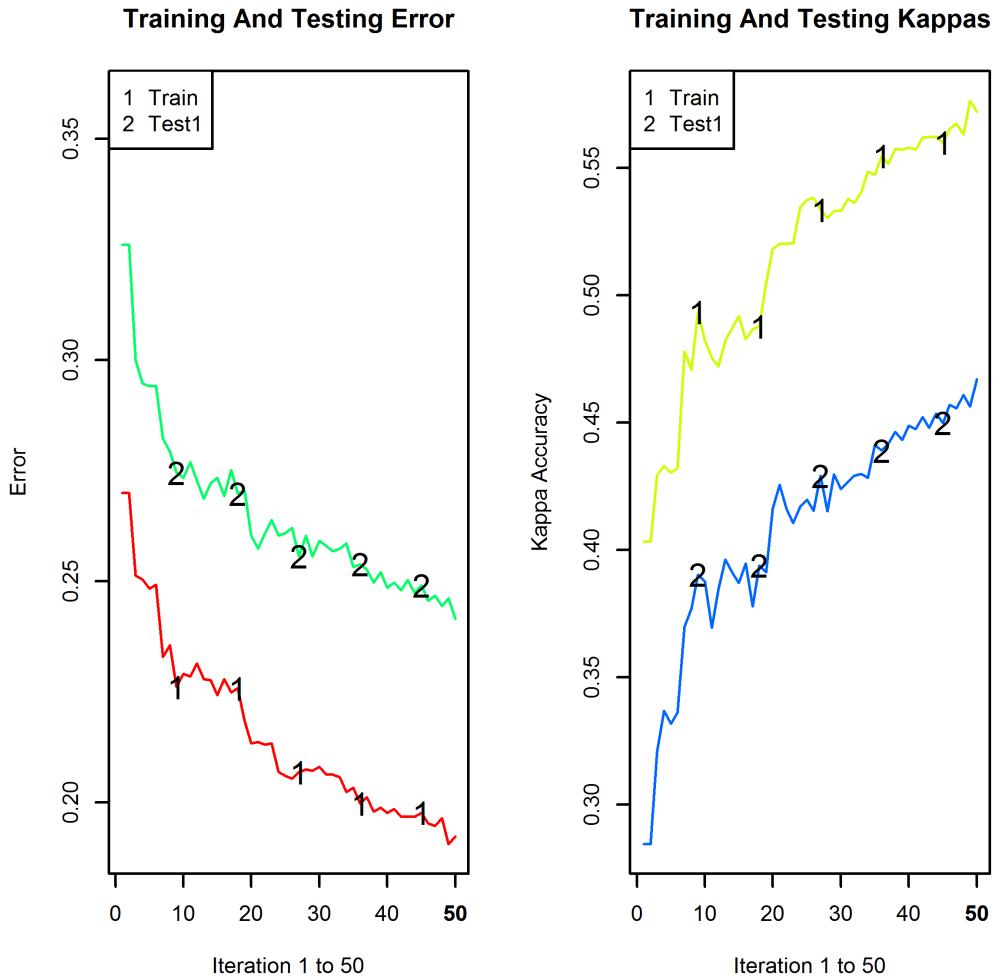


Figure 70.1: Error and kappa by iteration for training and test sets

The error rate of both the test and training set appear to decline as the number of iterations increases. Kappa appears to rise somewhat as the iterations increase for the training set, but it quickly flattens out for the testing data set.

We next look at the attribute influence scores and plot the 10 most influential variables, see Figure 70.2.

```
> scores<-varplot(output_test,plot.it=FALSE, type="scores")
> barplot(scores[1:10])
```

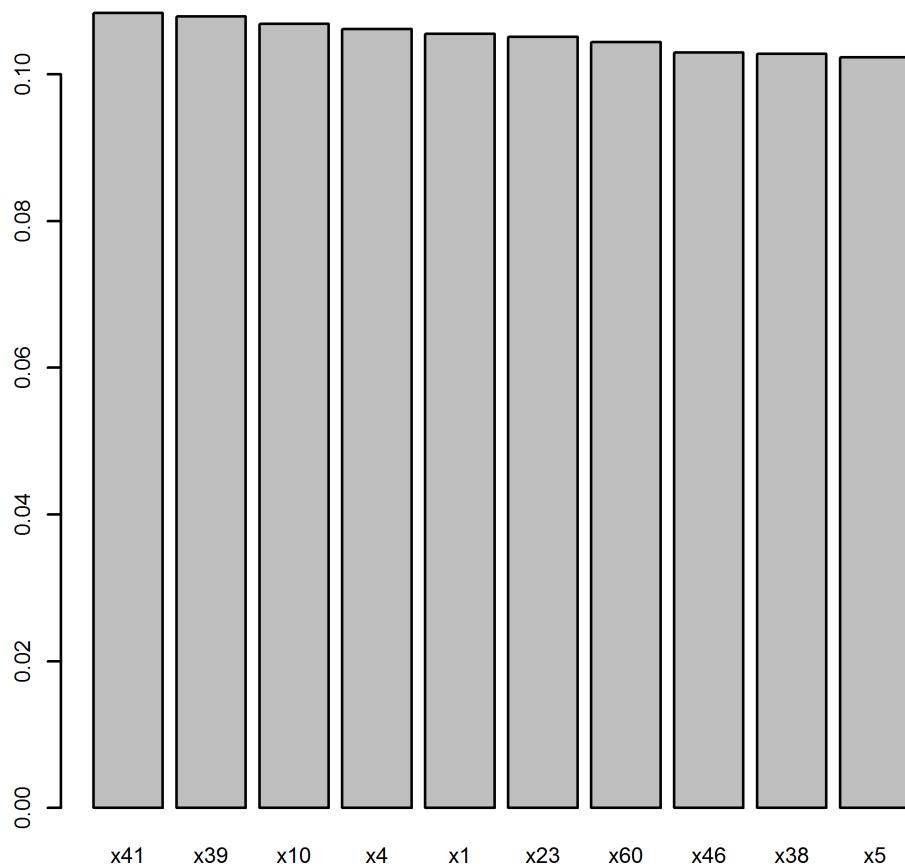


Figure 70.2: Ten most influential variables

For this example there is not a lot of variation between the top ten influential variables.

Step 5: Make Predictions

Next we re-estimate using the validation sample to predict the classes and compare to the actual observed classes.

```
> set.seed(127)
> output_valid <- ada(y ~., data = valid, iter = 50,
  loss = "l", type = "discrete", control = default)
```

```
> pred<-predict(output_valid,newdata=valid)

> table(pred)
pred
-1     1
378 184
> table(valid$y)

-1     1
356 206
```

The model predicts 378 in “class -1” and 184 in “class 1” whilst we actually observe 356 and 206 in “class -1” and “class 1” respectively. A summary of the class probability based predictions is obtained using:

```
> round(pred,3)
      [,1]   [,2]
[1,] 0.010 0.990
[2,] 0.062 0.938
[3,] 0.794 0.206
[4,] 0.882 0.118
...
[559,] 0.085 0.915
[560,] 0.799 0.201
[561,] 0.720 0.280
[562,] 0.996 0.004
```

Technique 71

Real L₂ Boost

The Real L₂ Boost uses real boosting with a logistic loss function. It can be run using the package **ada** with the **ada** function:

```
ada(z ~., data , iter , loss = "l", type = "real",
    control,...)
```

Key parameters include **iter**, the number of iterations used to estimate the model; **z** the data-frame of binary classes; **data** the data set of attributes with which you wish to train the model. Steps 1-2 are outlined beginning on page 501.

Step 3: Estimate Model

We choose 50 boosting iterations by setting the parameter **iter = 50**:

```
>set.seed(127)
> output_test <- ada(y ~., data = train, iter = 50,
+   loss = "l", type = "real",control = default)

> output_test
Call:
ada(y ~ ., data = train, iter = 50, loss = "l", type
= "real",
control = default)

Loss: logistic Method: real      Iteration: 50

Final Confusion Matrix for Data:
          Final Prediction
```

```
True value -1 1  
-1 1970 137  
1 487 785
```

Train Error: 0.185

Out-Of-Bag Error: 0.211 iteration= 50

Additional Estimates of number of iterations:

```
train.err1 train.kap1  
47 50
```

The confusion matrix for the data yields a training error of 18.5%. The Out-Of-Bag Error is a little over 21%.

Step 4: Assess Model Performance

The `addtest` function is used to evaluate the testing set without needing to refit the model:

```
> output_test<-addtest(output_test,test.x=test  
[, -73], test.y=test[, 73])  
  
> summary(output_test)  
Call:  
ada(y ~ ., data = train, iter = 50, loss = "l", type  
= "real",  
control = default)
```

Loss: logistic Method: real Iteration: 50

Training Results

Accuracy: 0.815 Kappa: 0.584

Testing Results

Accuracy: 0.753 Kappa: 0.451

It is noticeable that the accuracy of the model falls from 81.5%, for the training set, to 75.3% for the test set. A rather sharp decline is also observed in the kappa statistic.

We next calculate the influential variables. As we observed with Discrete L₂ Boost there is not a lot of variation in the top five most influential with all hovering in the approximate range of 10-11%.

```
> scores<-varplot(output_test, plot.it=FALSE, type="scores")  
  
> round(scores[1:5],3)  
x1      x2      x23      x5      x51  
0.114  0.111  0.108  0.108  0.106
```

Step 5: Make Predictions

We re-estimate and use the validation sample to make predictions. Notice that 385 are predicted to be in “class -1” and 177 in “class 1”. These predicted values are very close to those obtained by the Discrete L₂ Boost.

```
> pred<-predict(output_valid, newdata=valid)  
  
> table(pred)  
pred  
-1     1  
385  177  
> table(valid$y)  
  
-1     1  
356  206
```

Technique 72

Gentle L₂ Boost

The Gentle L₂ Boost uses gentle boosting with a logistic loss function. It can be run using the package `ada` with the `ada` function:

```
ada(z ~., data , iter , loss = "l", type = "gentle",
    control,...)
```

Key parameters include `iter`, the number of iterations used to estimate the model; `z` the data-frame of binary classes; `data` the data set of attributes with which you wish to train the model. Steps 1-2 are outlined beginning on page 501.

Step 3: Estimate Model

We choose 50 boosting iterations by setting the parameter `iter = 50`:

```
> set.seed(127)
> output_test <- ada(y ~., data = train, iter = 50,
+   loss = "l", type = "gentle",control = default)

> output_test
Call:
ada(y ~ ., data = train, iter = 50, loss = "l", type
= "gentle",
control = default)

Loss: logistic Method: gentle     Iteration: 50

Final Confusion Matrix for Data:
      Final Prediction
```

```
True value -1 1
      -1 1924 183
      1    493 779

Train Error: 0.2

Out-Of-Bag Error: 0.216 iteration= 50
```

Additional Estimates of number of iterations:

```
train.err1 train.kap1
      50          50
```

The test data set obtained a training error of 20% and a slightly higher Out-Of-Bag Error estimate of close to 22%.

Step 4: Assess Model Performance

Next we begin the assessment of the model's performance using the test data set. Notice that the accuracy declines slightly from 80% for the training sample to 74% for the test data set. We see modest declines in kappa also as we move from the training to the test data set.

```
> output_test<-addtest(output_test,test.x=test
[, -73],test.y=test[,73])

> summary(output_test)
Call:
ada(y ~ ., data = train, iter = 50, loss = "l", type
= "gentle",
control = default)
```

Loss: logistic Method: gentle Iteration: 50

Training Results

Accuracy: 0.8 Kappa: 0.552

Testing Results

Accuracy: 0.742 Kappa: 0.427

The top five most influential variables are given below. It is interesting to observe that they all lie roughly in a range of 9% to 10%.

```
> scores<-varplot(output_test, plot.it=FALSE, type="scores")  
  
> round(scores[1:5],3)  
x39    x12    x56    x70    x41  
0.099  0.097  0.097  0.097  0.095
```

Step 5: Make Predictions

We use the validation sample to make class predictions. Notice that 412 are predicted to be in “class -1” and 150 in “class 1”. These predicted values are somewhat different from those obtained by the Discrete L₂ Boost (see page 506) and Real Discrete L₂ Boost (see page 510) .

```
> output_valid <- ada(y ~., data = valid, iter = 50,  
loss = "l", type = "gentle", control = default)  
  
> pred<-predict(output_valid,newdata=valid)  
  
> table(pred)  
pred  
-1    1  
412 150  
> table(valid$y)  
  
-1    1  
356 206
```

Multi-Class Boosting

Technique 73

SAMME

SAMME¹³⁵ is a extension of the binary Ada Boost algorithm to two or more classes. It uses an exponential loss function with $\alpha_m = \ln\left(\frac{1-\varepsilon_m}{\varepsilon_m}\right) + \ln(k-1)$, where k is the number of classes.

☛ PRACTITIONER TIP ☛

When the number of classes $k = 2$ then $\alpha_m = \ln\left(\frac{1-\varepsilon_m}{\varepsilon_m}\right)$ which is similar to the learning coefficient of Ada Boost.M1. Notice that SAMME can be used for both binary boosting and multiclass boosting. This is also the case for the Breiman and the Freund extension's discussed on page 519 and page 522 respectively. As an experiment re-estimate the Sonar data set (see page 482) using SAMME. What do you notice about the results relative to Ada Boost.M1?

The SAMME algorithm can be run using the package `adabag` with the `boosting` function:

```
boosting(Class ~., data, mfinal, control, coeflearn  
= "Zhu", ...)
```

Key parameters include `mfinal` the number of iterations used to estimate the model; `Class` the data-frame of classes; `data` the data set containing the features over which you wish to train the model and `coeflearn="Zhu"` to specify using the SAMME algorithm.

Step 1: Load Required Packages

NOTE... ↗

The `Vehicle` data set (see page 23) is stored in the `mlbench` package. It is automatically loaded with the `adabag` package. If you need to load `Vehicle` directly type `library(mlbench)` at the R prompt.

We begin by loading the library package and the `Vehicle` data set.

```
> library (adabag)  
> data(Vehicle)
```

Step 2: Prepare Data & Tweak Parameters

The `adabag` package uses the `rpart` function to generate decision trees. So we need to supply an `rpart.control` object to the model. We use 500 observations as the training set and the remainder for testing.

```
> default <- rpart.control(cp = -1 , maxdepth = 4 ,  
  minsplit = 0)  
> set.seed(107)  
> N=nrow(Vehicle)  
> train <- sample(1:N, 500, FALSE)  
> observed<-Vehicle[train,]
```

Step 3 & 4: Estimate Model & Assess Model Performance

We estimate the model, create a table of predicted and observed values and then calculate the error rate.

```
>set.seed(107)  
>fit<-boosting(Class ~., data = Vehicle[train,],  
  mfinal=25,control = default,coeflearn="Zhu")  
  
> table(observed$Class,fit$class,dnn= c("Observed  
  Class","Predicted Class"))  
          Predicted Class  
Observed Class bus opel saab van
```

```
bus    124      0      0      0
opel     0    114     13      0
saab     0     12   117      0
van      0      0      0   120
```

```
> error_rate = (1 - sum(fit$class == observed$Class) /
  500)
> round(error_rate, 2)
[1] 0.05
```

☛ PRACTITIONER TIP ☛

The object returned from using the boosting function contains some very useful information, particularly relevant are - `$trees`, `$weights`, `$votes`, `$prob`, `$class` and `$importance`. For example, if you enter `fit$trees` R will return details of the trees used to estimate the model.

Notice the error rate is 5%, which seems quite low. Perhaps we have over fit the model? As a check we perform a 10-fold cross validation. The confusion matrix is obtained using `$confusion` and the training error by `$error`.

```
> set.seed(107)
> cv<-boosting.cv(Class ~ ., data = Vehicle, mfinal
  =25, v=10, control = default, coeflearn="Zhu")

> cv$confusion
          Observed Class
Predicted Class bus  opel  saab  van
               bus 208    0     3     1
               opel  2   113    91     4
               saab  7    94   116    11
               van   1     5     7  183
> cv$error
[1] 0.2671395
```

The cross validation error rate is much higher at 27%. This is probably a better reflection of what we can expect on the testing sample.

Before we use the test set to make predictions, we take a look at the three most influential features.

```
> influence<-sort(fit$importance, decreasing = TRUE)
```

```
> round(influence[1:3],1)
   Max.L.Ra    Pr.Axis.Ra Sc.Var.Maxis
      14.0          9.0        8.7
```

NOTE... ↗

Cross validation is a powerful concept because it can be used to estimate the error of an ensemble without having to divide the data into a training and testing set. It is often used in small samples, but is equally advantageous in larger datasets also.

Step 5: Make Predictions

We use the test data set to make predictions.

```
> pred<-predict.boosting(fit,newdata=Vehicle[-train
  ,])
> pred$error
[1] 0.2745665
> pred$confusion
            Observed Class
Predicted Class bus opel saab van
  bus       88     1     1     1
  opel      1    42    37     1
  saab      2    40    48     4
  van       3     2     2    73
```

The prediction error at 27% is close to the cross validated error rate.

Technique 74

Breiman's Extension

Breiman's extension assumes $\alpha_m = 0.5 \log\left(\frac{1-\varepsilon_m}{\varepsilon_m}\right)$. It can be run using the package `adabag` with the `boosting` function:

```
boosting(Class ~., data, mfinal, control, coeflearn  
= "Breiman", ...)
```

Key parameters include `mfinal` the number of iterations used to estimate the model; `Class` the data-frame of classes; `data` the data set containing the features over which you wish to train the model and `coeflearn="Breiman"`.

☛ PRACTITIONER TIP ☚

The function `boosting` takes the optional parameter `boos`. By default it is set to `TRUE` and a bootstrap sample is drawn using the weight for each observation. If `boos = FALSE` then every observation is used.

Steps 1 and 2 are outlined beginning on page 516.

Step 3 & 4: Estimate Model & Assess Model Performance

We fit the model using the training sample, calculate the error rate and also calculate the error rate using a 10-fold cross validation.

```
>set.seed(107)  
>fit<-boosting(Class~., data= Vehicle[train,], mfinal  
=25, control = default, coeflearn="Breiman")
```

```
> error_rate = (1-sum(fit$class== observed$Class)/  
+ 500)  
> round(error_rate,2)  
[1] 0.1  
  
> set.seed(107)  
> cv<-boosting.cv(Class~., data = Vehicle, mfinal=25, v  
+ =10, control = default, coeflearn="Breiman")  
> round(cv$error,2)  
[1] 0.26
```

Whilst the error rate of 10% for the fitted model on the training sample is higher than observed for the SAMME algorithm (see text beginning on page 516), it remains considerably lower than that obtained by cross validation. Once again we expect the cross-validation error to better reflect what we expect to observe in the test sample.

The order of the three most influential features is as follows:

```
> influence<-sort(fit$importance, decreasing = TRUE)  
  
> round(influence[1:3],1)  
Max.L.Ra Sc.Var.maxis Max.L.Rect  
23.7 14.0 8.8
```

Max.L.Ra is the most influential variable here and also using the SAMME algorithm (see page 517).

Step 5: Make Predictions

Finally we make predictions using the test data set and print out the observed versus predicted value.

```
> pred<-predict.boosting(fit, newdata=Vehicle[-train  
+ ,])  
  
> round(pred$error,2)  
[1] 0.28  
  
> pred$confusion  
Observed Class  
Predicted Class bus opel saab van  
bus 87 8 4 1  
opel 1 43 33 1
```

saab	2	25	45	4
van	4	9	6	73

The predicted error rate is 28% (close to the cross validated value).

Technique 75

Freund's Adjustment

Freund's adjustment is another direct extension of the binary Ada Boost algorithm to two or more classes where $\alpha_m = \log\left(\frac{1-\varepsilon_m}{\varepsilon_m}\right)$. It can be run using the package **adabag** with the **boosting** function:

```
boosting(Class ~., data, mfinal, control, coeflearn  
= "Freund", ...)
```

Key parameters include **mfinal** the number of iterations used to estimate the model; **Class** the data-frame of classes; **data** the data set containing the features over which you wish to train the model and **coeflearn="Freund"**.

Steps 1 and 2 are outlined beginning on page 516.

Step 3 & 4: Estimate Model & Assess Model Performance

We fit the model using the training sample, calculate the error rate and also calculate the error rate using a 10-fold cross validation.

```
> set.seed(107)  
> fit<-boosting(Class~., data = Vehicle[train,], mfinal  
= 25, control = default, coeflearn="Freund")  
  
> error_rate = (1-sum(fit$class== observed$Class)/  
500)  
> round(error_rate, 2)  
[1] 0.06  
  
> set.seed(107)
```

```
>cv<-boosting.cv(Class~, data = Vehicle, mfinal=25, v  
=10, control = default, coeflearn="Freund")  
> round(cv$error, 2)  
[1] 0.25
```

Whilst the error rate of 6% for the fitted model on the training sample is higher than observed for the SAMME algorithm (see (see text beginning on page 516), it remains considerably lower than that obtained by cross validation.

The order of the three most influential features is as follows:

```
> influence<-sort(fit$importance, decreasing = TRUE)  
  
> round(influence[1:3], 1)  
Max.L.Ra Sc.Var.maxis Pr.Axis.Ra  
20.1         9.7        9.3
```

Max.L.Ra is the most influential variable here (also using the SAMME and Breiman algorithm - see pages 517 and 520).

Step 5: Make Predictions

Finally we make predictions using the test data set and print out the observed versus predicted table.

```
> pred<-predict.boosting(fit, newdata=Vehicle[-train  
,])  
  
> round(pred$error, 2)  
[1] 0.27  
  
> pred$confusion  
          Observed Class  
Predicted Class bus opel saab van  
    bus     86     1     2     0  
    opel     1    47    37     0  
    saab     3    31    45     4  
    van      4     6     4    75
```

The predicted error rate is 27% (close to the cross validated value of 25%).

Continuous Response Boosted Regression

Technique 76

L₂ Regression

NOTE... ↗

L₂ boosting minimizes the least squares error (the sum of the square of the differences between the observed value (y_i) and the estimated values (\hat{y}_i)):

$$L_2 = \sum_{i=1}^N (y_i - \hat{y}_i)^2 \quad (76.1)$$

\hat{y}_i is estimated as a function of the independent variables / features.

L₂ Boosting for continuous response variables can be run using the package **mboost** with the **glmboost** function:

```
glmboost(z ~ ., data ,family=Gaussian(),control,...)
```

Key parameters include **z** the continuous response variable; **data** the data set of independent variables; **family=Gaussian()** implements L₂ boosting; **control** which limits the number of boosting iterations and also controls the shrinkage parameter.

Step 1: Load Required Packages

We begin by loading the **mboost** package and the **bodyfat** data set described on page 62.

```
>library("mboost")
```

```
>data("bodyfat", package="TH.data")
```

Step 2: Prepare Data & Tweak Parameters

In the original study Garcia et al used 45 observations for model validation. We follow the same approach using the remaining 26 observations as the testing sample.

```
set.seed(465)
train <- sample(1:71, 45, FALSE)
```

Step 3: Estimate Model & Assess Fit

☛ PRACTITIONER TIP ☛

Set `trace = TRUE` if you want to see status information during the fitting process.

First we fit the model using the `glmboost` function. We set the number of boosting iterations to 75 and the shrinkage parameter (`nu`) to 0.1. This is followed by using `coef` to show the estimated model coefficients.

```
>fit<-glmboost(DEXfat ~ ., data = bodyfat[train,],
  family = Gaussian(), control = boost_control(mstop
  =75, nu = 0.1, trace = FALSE))

> coef(fit, off2int=TRUE)
(Intercept)           age      waistcirc
-66.27793041     0.02547924    0.17377131

hipcirc elbowbreadth  kneebreadth
0.46430335   -0.63272508    0.86864111

anthro3a      anthro3b
3.36145109    3.52597323
```

• PRACTITIONER TIP •

We use `coef(fit, off2int=TRUE)` to add back the offset to the intercept. To see the intercept without the offset use `coef(fit)`.

A key tuning parameter of boosting is the number of iterations. We set `mstop = 75`. However, to prevent over fitting it is important to choose the optimal stopping iteration with care. We use 10-fold cross validated estimates of the empirical risk help us to choose the optimal number of boosting iterations. Empirical risk is calculated using the `cvrisk` function.

```
> cv10f <- cv(model.weights(fit), type = "kfold", B = 10)

> cvm <- cvrisk(fit, folds = cv10f)

> mstop(cvm)
[1] 40

> fit[mstop(cvm)]
```

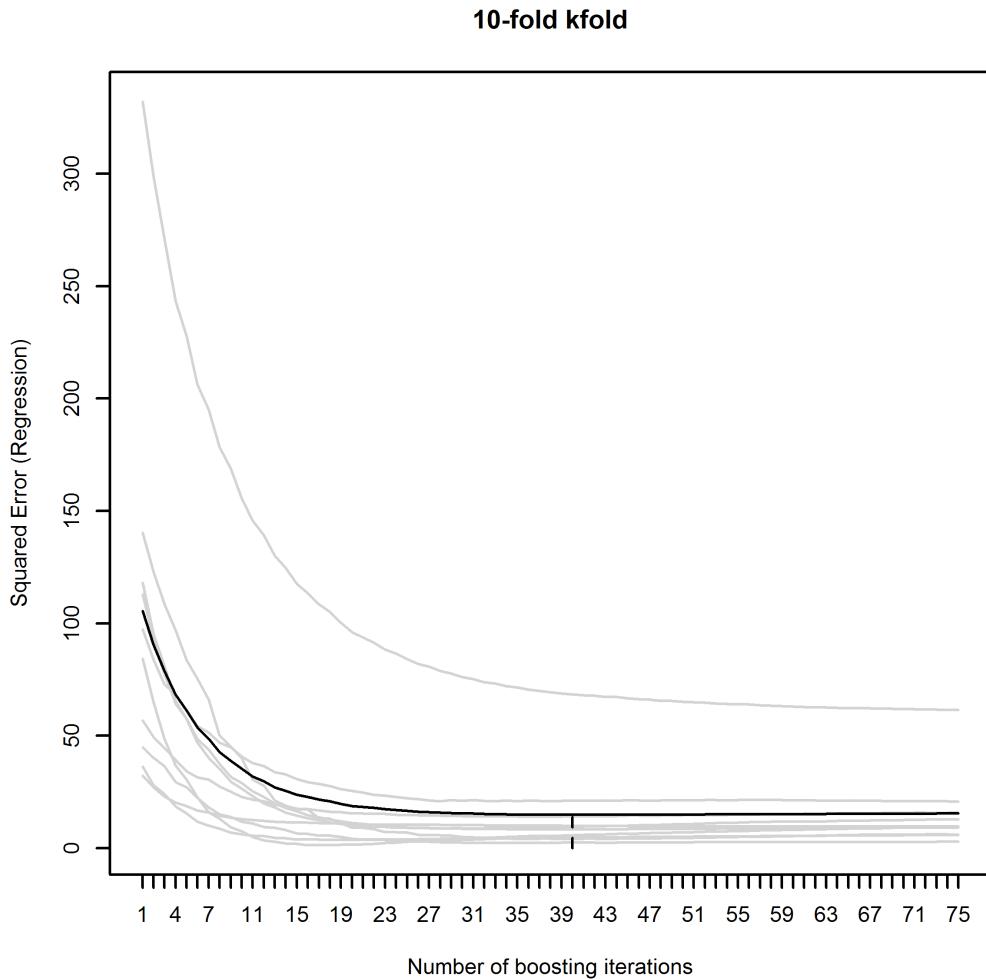


Figure 76.1: Cross-validated predictive risk for bodyfat data set and L₂ Regression

Figure 76.1 displays the predictive risk. The optimal stopping iteration minimizes the average risk over all samples. Notice that `mstop(cvm)` returns the optimal number, in this case 40. We use `fit[mstop(cvm)]` to set the model parameters automatically to the optimal `mstop`.

Given our optimal model we calculate a bootstrapped confidence interval at the 90% level for each of the parameters. For illustration we only use 200 bootstraps, in practice you should use at least 1,000.

```
> CI <- confint(fit, B = 200, level = 0.9)
> CI
```

```
Bootstrap Confidence Intervals
      5%          95%
(Intercept) -74.8992833 -49.16894522
age          0.0000000  0.04193643
waistcirc    0.0250575  0.27216736
hipcirc      0.2656622  0.60493289
elbowbreadth -0.5086485 0.45619227
kneebreadth   0.0000000 1.93441658
anthro3a     0.0000000  8.05412043
anthro3b     0.0000000  5.32118418
anthro3c     0.0000000  3.05345810
anthro4      0.0000000  4.66542476
```

 **PRACTITIONER TIP** 

To compute a confidence interval for another level simply enter the desired level using the `print` function. For example `print(CI, level = 0.8)` returns:

```
> print(CI, level = 0.8)
      Bootstrap Confidence Intervals
      10%          90%
(Intercept) -72.20299829 -51.87095140
age          0.00000000  0.03249837
waistcirc    0.04749293  0.24761814
hipcirc      0.32502314  0.56940817
elbowbreadth -0.30893554 0.00000000
kneebreadth   0.00000000 1.45814706
anthro3a     0.00000000  7.14609005
anthro3b     0.00000000  4.67906130
anthro3c     0.00000000  2.29523786
anthro4      0.00000000  3.59735084
```

The confidence intervals indicate that `waistcirc` and `hipcirc` are statistically significant. Since our goal is to build a parsimonious model we re estimate the model using only these two variables.

```
> fit2 <- gamboost(DEXfat ~ waistcirc+hipcirc, data
= bodyfat[train,],family=Gaussian(),control =
boost_control(mstop =150 ,nu = 0.1,trace = FALSE))
```

```
> CI2 <- confint(fit2, B = 50, level = 0.9)

> par(mfrow = c(1, 2))
> plot(CI2, which = 1)
> plot(CI2, which = 2)
> par(new=TRUE)
```

Notice we use the `gamboost` function rather than `glmboost`. This is because we want to compute and display point-wise confidence intervals using the `plot` function. The confidence intervals are shown in Figure 76.2. For both variables the effect shows an almost linear increase with circumference size.

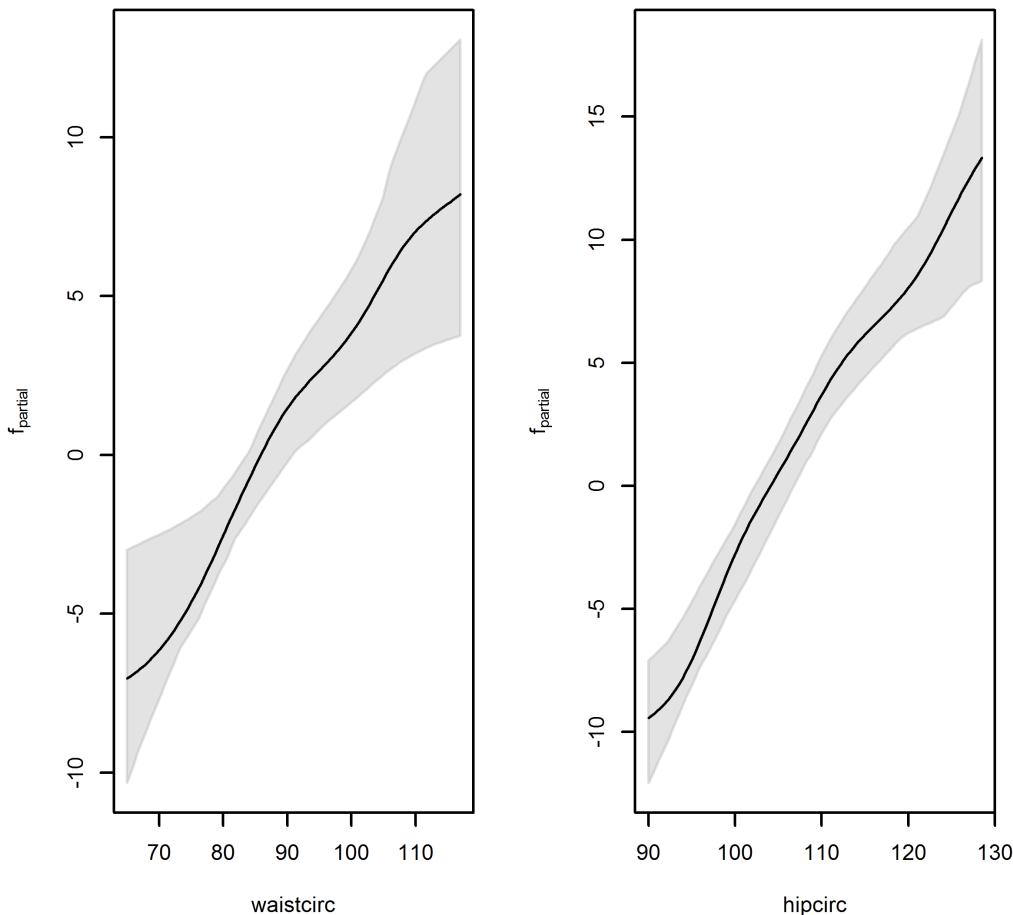


Figure 76.2: Point-wise confidence intervals for `waistcirc` and `hipcirc`

Step 4: Make Predictions

We now make predictions using the testing data set, plot the result (see Figure 76.3) and calculate the squared correlation coefficient. The final model shows a relatively linear relationship with `DEXfat` with a squared correlation coefficient of 0.858 (linear correlation = 0.926).

```
> pred<-predict(fit2, newdata=bodyfat[-train,], type  
= "response")  
  
>plot(bodyfat$DEXfat[-train],pred,xlab="DEXfat",  
ylab="Predicted Values", main="Training Sample  
Model Fit")  
  
> round(cor(pred,bodyfat$DEXfat[-train])^2,3)  
[1] 0.858
```

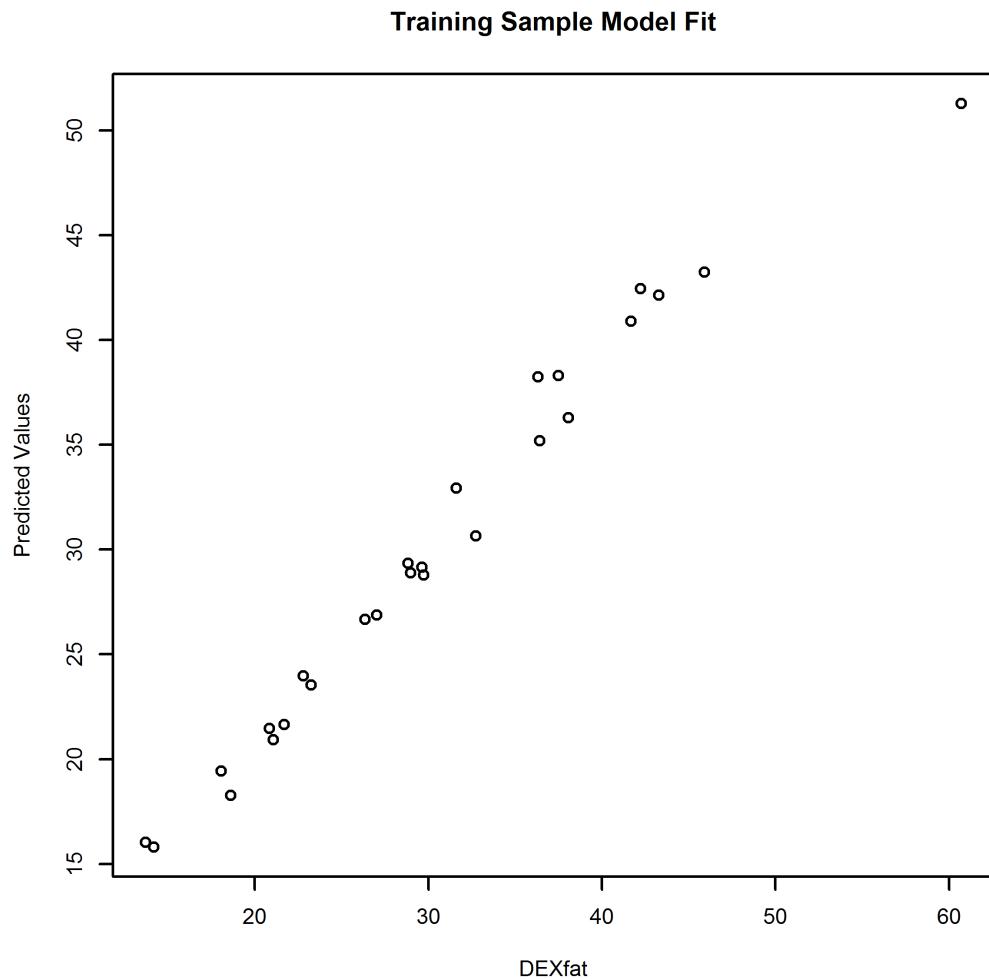


Figure 76.3: Plot of observed versus predicted values for L₂ bodyfat regression.

Technique 77

L₁ Regression

NOTE... ↗

L₁ minimizes the least absolute deviations (also known as the least absolute errors) by minimizing the sum of the absolute differences between the observed value (y_i) and the estimated values (\hat{y}_i):

$$L_1 = \left| \sum_{i=1}^N y_i - \hat{y}_i \right|$$

\hat{y}_i is estimated as a function of the independent variables / features. Unlike L₂ loss with is sensitive to extreme values. L₁ is robust to outliers.

L₁ boosting for continuous response variables can be run using the package **mbnboost** with the **glmboost** function:

```
glmboost(z ~ ., data ,family=Laplace(),control,...)
```

Key parameters include **z** the continuous response variable; **data** the data set of independent variable's, **family=Laplace()** implements L₁loss boosting; **control** which limits the number of boosting iterations and the shrinkage parameter. Steps 1-3 are outlined beginning on page 525.

Step 3: Estimate Model & Assess Fit

First we fit the model using the **glmboost** function. We set the number of boosting iterations to 300 and the shrinkage parameter (**nu**) to 0.1. This is

followed by using `coef` to show the estimated model coefficients.

```
> fit<-glmboost(DEXfat~., data = bodyfat[train,],  
+ family=Laplace(), control = boost_control(mstop  
+ =300, nu = 0.1, trace = FALSE))  
  
> coef(fit, off2int=TRUE)  
 (Intercept) age waistcirc  
 -59.12102158 0.01124291 0.06454519  
  
hipcirc elbowbreadth kneebreadth  
0.49652307 -0.38873368 0.83397012  
  
anthro3a anthro3b anthro3c  
0.58847326 3.64957585 2.00388290
```

All of the variables receive a weight with the exception of `anthro4`. A 10-fold cross validated estimate of the empirical risk is used to choose the optimal number of boosting iterations. The empirical risk is calculated using the `cvrisk` function.

```
> cv10f <- cv(model.weights(fit), type = "kfold", B  
+ =10)  
> cvm <- cvrisk(fit, folds = cv10f)  
  
> mstop(cvm)  
[1] 260  
  
>fit[mstop(cvm)]
```

The optimal stopping iteration is 260 and `fit[mstop(cvm)]` sets the model parameters automatically to the optimal `mstop`.

Step 4: Make Predictions

Predictions using the testing data set are made using the optimal model `fit`. A plot of the predicted and observed values (see Figure 77.1) shows a relatively linear relationship with `DEXfat` with a squared correlation coefficient of 0.906 (linear correlation = 0.95).

```
> pred<-predict(fit, newdata=bodyfat[-train,], type  
+ = "response")
```

```
> plot(bodyfat$DEXfat[-train], pred, xlab="DEXfat",
      ylab="Predicted Values", , main="Training Sample Model Fit")  
  
> round(cor(pred, bodyfat$DEXfat[-train])^2, 3)  
[1,] 0.906
```

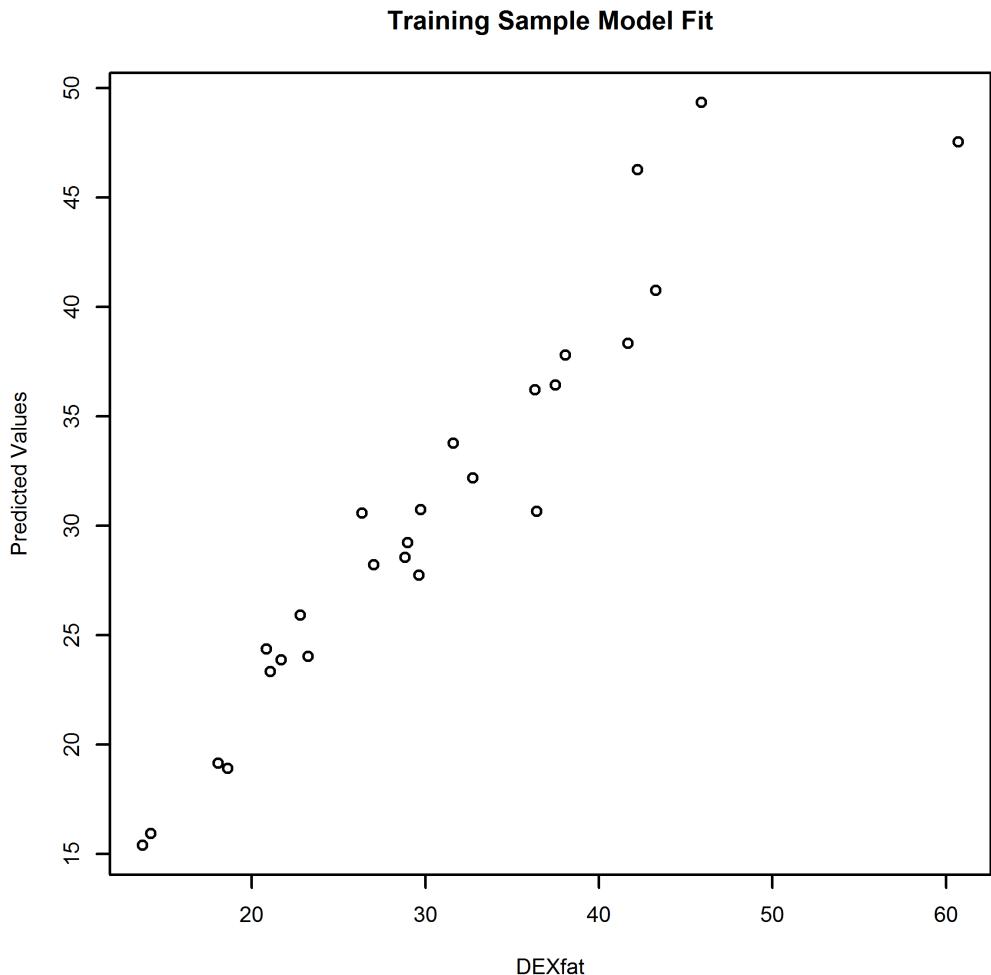


Figure 77.1: Plot of observed versus predicted values for L₁ bodyfat regression.

Technique 78

Robust Regression

Robust boosting regression uses the Huber loss function which is less sensitive to outliers than the L2 loss function discussed on 525. It can be run using the package `mboost` with the `glmboost` function:

```
glmboost(z ~ ., data ,family=Huber(),control ,...)
```

Key parameters include `z` the continuous response variable; `data` the data set of independent variables; `family = Huber()`; `control` which limits the number of boosting iterations and the shrinkage parameter. Steps 1-3 are outlined beginning on page 525.

Step 3: Estimate Model & Assess Fit

First we fit the model using the `glmboost` function. We set the number of boosting iterations to 300 and the shrinkage parameter (`nu`) to 0.1. This is followed by using `coef` to show the estimated model coefficients.

```
> fit <- glmboost(DEXfat ~ ., data = bodyfat[train
  ,],family=Huber(),control = boost_control(mstop
  =300,nu = 0.1,trace = FALSE))

> coef(fit,off2int=TRUE)
(Intercept)           age      waistcirc
-60.83574185    0.02340993    0.17680696

hipcirc elbowbreadth kneebreadth
0.45778183   -0.83925018    0.64471220

anthro3a      anthro3b      anthro3c
```

```
0.26460670    5.14548154    0.78826723
```

All of the variables receive a weight with the exception of `anthro4`. A 10-fold cross validated estimate of the empirical risk is used to choose the optimal number of boosting iterations. The empirical risk is calculated using the `cvrisk` function.

```
> cv10f <- cv(model.weights(fit), type = "kfold", B  
=10)  
> cvm <- cvrisk(fit, folds = cv10f)  
  
> mstop(cvm)  
[1] 100  
  
>fit[mstop(cvm)]
```

The optimal stopping iteration is 100 and `fit[mstop(cvm)]` sets the model parameters automatically to the optimal `mstop`.

Step 4: Make Predictions

Predictions using the testing data set are made using the optimal model `fit`. A plot of the predicted and observed values (see Figure 78.1) shows a relatively linear relationship with `DEXfat` with a squared correlation coefficient of 0.913.

```
> pred<-predict(fit, newdata=bodyfat[-train,], type  
= "response")  
  
>plot(bodyfat$DEXfat[-train],pred,xlab="DEXfat",  
ylab="Predicted Values",, main="Training Sample  
Model Fit")  
  
> round(cor(pred,bodyfat$DEXfat[-train])^2,3)  
[,1]  
[1,] 0.913
```

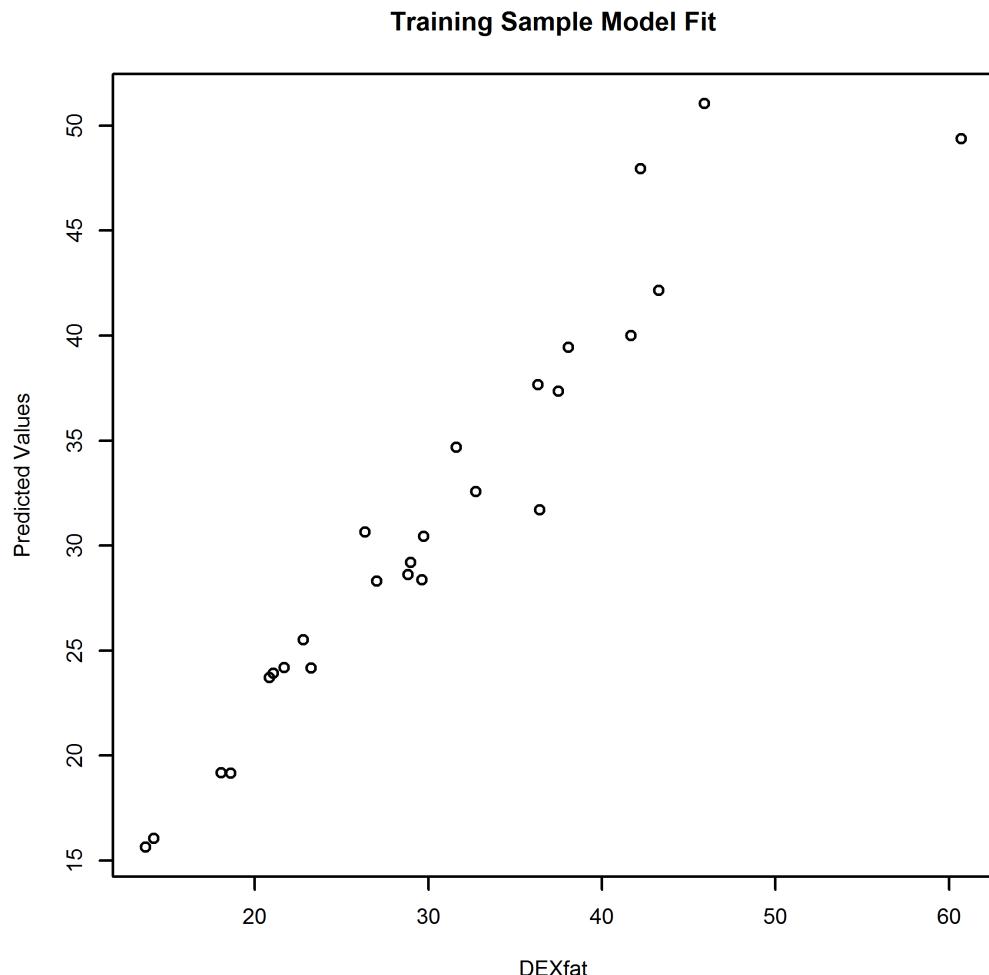


Figure 78.1: Plot of observed versus predicted values for Robust bodyfat regression.

Technique 79

Generalized Additive Model

The generalized additive model (GAM) is a generalization of the linear regression model in which the coefficients can be estimated as smooth functions of covariates. The GAM model can account for non-linear relationships between response variables and covariates:

$$E(Y | X_1, X_2, \dots, X_K) = \alpha + f_1(X_1) + f_2(X_2) + \dots + f_K(X_K) \quad (79.1)$$

A boosted version of the model can be run using the package `mboost` with the `gamboost` function:

```
gamboost(z ~ ., data, control, ...)
```

Key parameters include `z` the continuous response variable; `data` the data set of independent variables, `control` which limits the number of boosting iterations and controls the shrinkage parameter. Steps 1-3 are outlined beginning on page 525.

NOTE...

For more than 100 years, what data scientists could do was limited by asymptotic theory. Consumers of statistical theory such as Economists focused primarily on linear models and residual analysis. With boosting there is no excuse anymore for using overly restrictive statistical models.

Step 3: Estimate Model & Assess Fit

We previously saw that `waistcirc` and `hipcirc` are the primary independent variables for predicting body fat (see page 525). We use these two variables

to estimate linear terms (`bols()`), smooth terms (`bbs()`) and interactions between `waistcirc` and `hipcirc` modeled using decision trees (`btree()`).

```
> m<-DEXfat~waistcirc+hipcirc+bols(waistcirc)+bols(  
  hipcirc)+bbs(waistcirc)+bbs(hipcirc)+ btree(  
  hipcirc,  
+ waistcirc, tree_controls = ctree_control(maxdepth  
  = 4, mincriterion = 0))
```

We set the number of boosting iterations to 150 and the shrinkage parameter (`nu`) to 0.1. This is followed by a 10-fold cross validated estimate of the empirical risk to choose the optimal number of boosting iterations. The empirical risk is calculated using the `cvrisk` function. `fit[mstop(cvm)]` assigns the model coefficients at the optimal iteration.

```
> fit<- gamboost(m, data = bodyfat[train,],control =  
  boost_control(mstop =150 ,nu = 0.1,trace = FALSE)  
)  
  
> cv10f <- cv(model.weights(fit), type = "kfold",B  
  =10)  
> cvm <- cvrisk(fit, folds = cv10f)  
  
> mstop(cvm)  
[1] 41  
  
> fit[mstop(cvm)]
```

The optimal stopping iteration is 41 and `fit[mstop(cvm)]` sets the model parameters automatically to the optimal `mstop`.

Plots of the linear effects and the interaction map can be obtained by typing:

```
> plot(fit,which=1)  
  
> plot(fit,which=2)  
  
> plot(fit,which=5)
```

The resulting plots are given in Figure 79.1 and Figure 79.2. Figure 79.1 indicates a strong linear effect for both `waistcirc` and `hipcirc`. Figure 79.2 indicates that a hip circumference larger than about 110 cm leads to increased body fat provided waist circumference is larger than approximately 90cm.

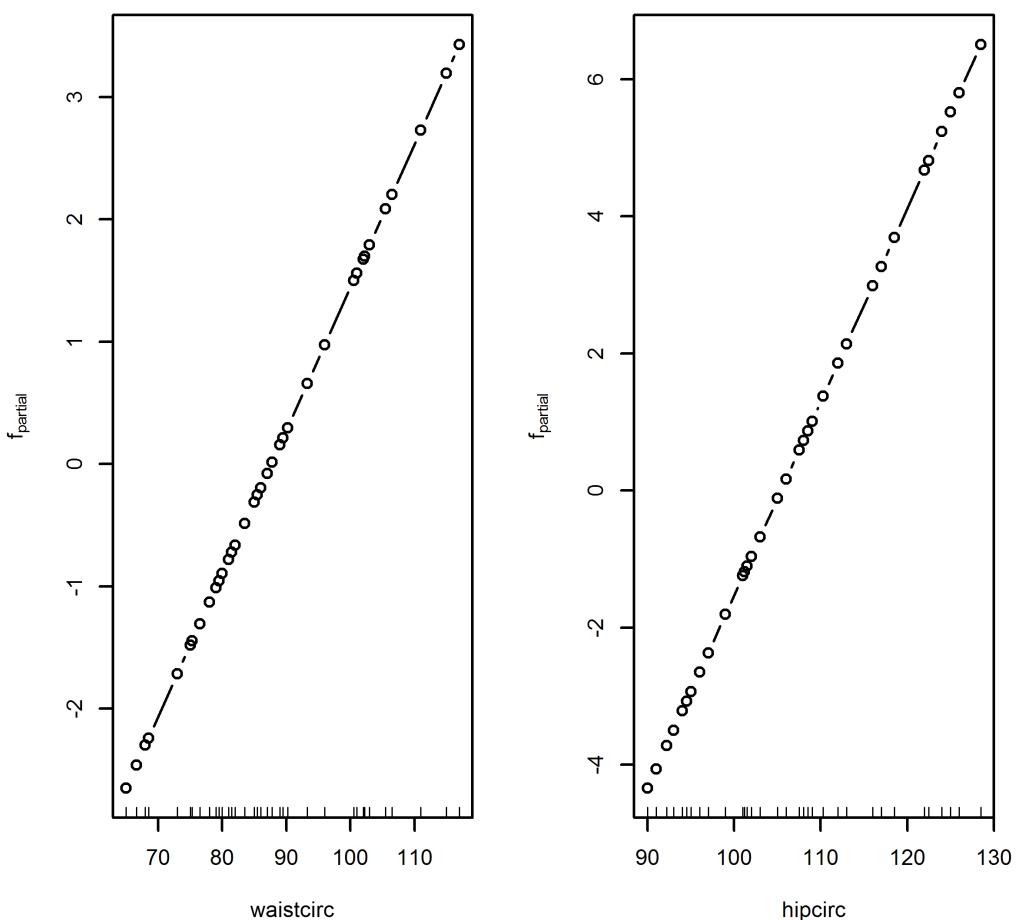


Figure 79.1: GAM regression estimates of linear effects for `waistcirc` and `hipcirc`

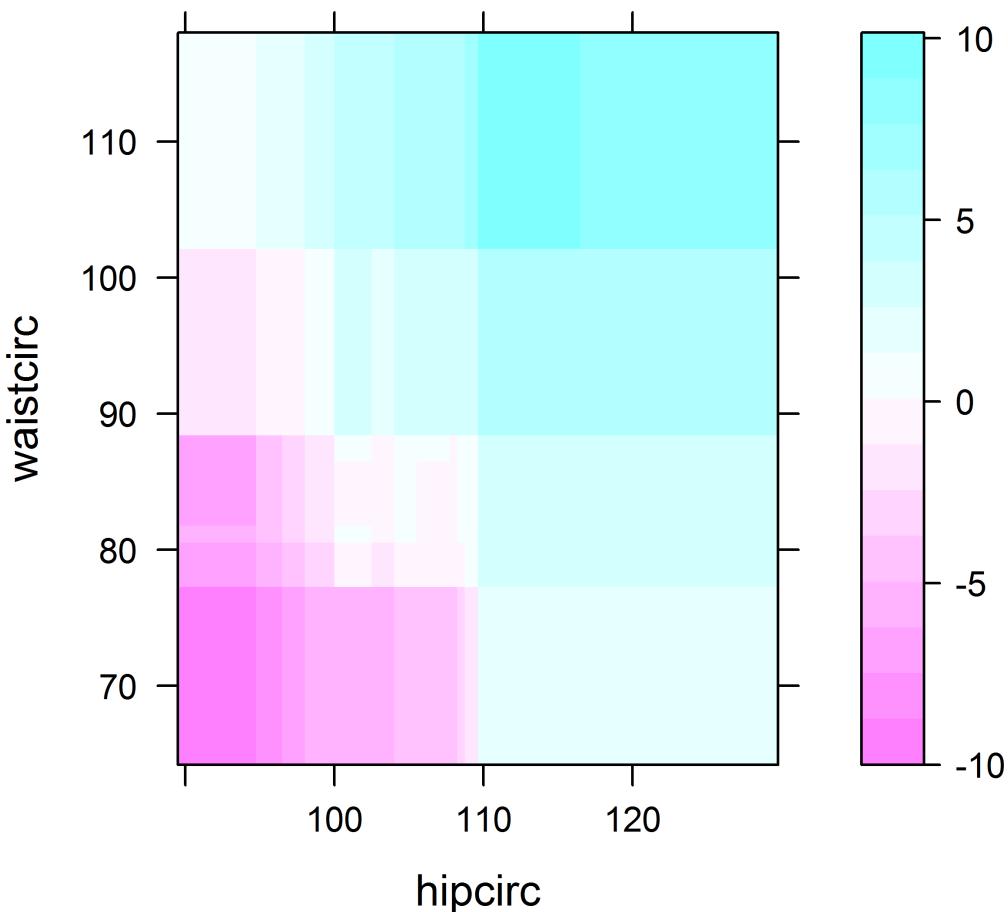


Figure 79.2: Interaction model component between `waistcirc` and `hipcirc`

Step 4: Make Predictions

Predictions using the test data set are made with the optimal model `fit`. A plot of the predicted and observed values (see Figure 79.3) shows a relatively linear relationship with `DEXfat`. The squared correlation coefficient is 0.867 (linear correlation = 0.93).

```
> pred<-predict(fit, newdata=bodyfat[-train,], type = "response")
```

```
> plot(bodyfat$DEXfat[-train], pred, xlab="DEXfat",  
      ylab="Predicted Values", main="Training Sample Model Fit")  
  
> round(cor(pred, bodyfat$DEXfat[-train])^2, 3)  
[1,] 0.867
```

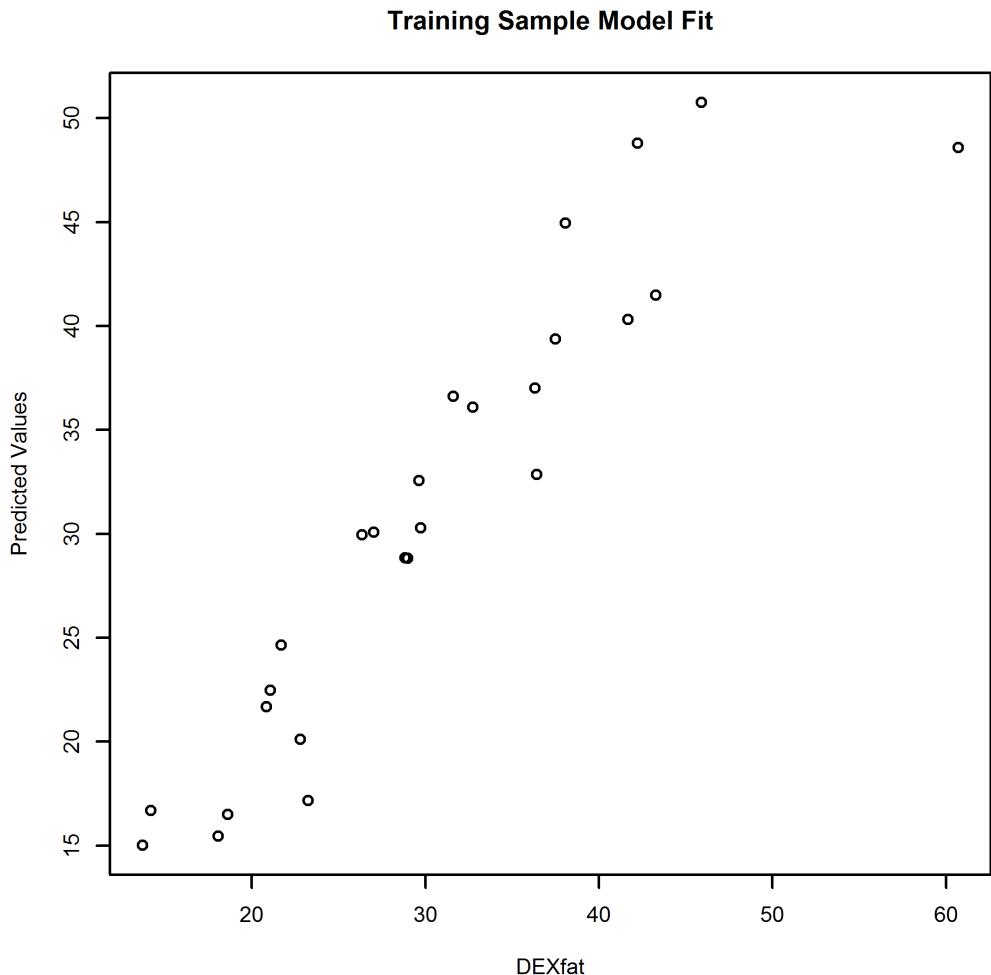


Figure 79.3: Plot of observed versus predicted values for GAM regression

Technique 80

Quantile Regression

Quantile regression models the relationship between the independent variables and the conditional quantiles of the response variable. It provides a more complete picture of the conditional distribution of the response variable when both lower and upper or all quantiles are of interest.

For example, in the analysis of body mass both lower (underweight) and upper (overweight) quantiles are of interest to health professionals and health conscious individuals. We apply to the `bodyfat` data set to illustrate this point.

Quantile regression can be run using the package `mboost` with the `glmboost` function:

```
glmboost(z ~ ., data ,family=QuantReg(),control,...)
```

Key parameters include `z` the continuous response variable; `data` the data set of independent variables; `family=QuantReg`; `control` which limits the number of boosting iterations and the shrinkage parameter.

Step 1 : Load Required Packages

We illustrate the use of this model using the `bodyfat` data frame. Details of this data set are discussed on page 62.

```
library("mboost")
data("bodyfat", package="TH.data")
set.seed(465)
```

Step 2: Estimate Model & Assess Fit

The question we address is do a common set of factors explain the 25th and 75th percentiles of the data set? To solve this issue we estimate two quantile regression models, one at the 25th percentile and the other at the 75th percentile.

```
> fit25 <- glmboost(DEXfat ~ ., data = bodyfat,
  family=QuantReg(tau=0.25),control = boost_control(
  mstop =1200,nu = 0.1,trace = FALSE))
```

```
> fit75 <- glmboost(DEXfat ~ ., data = bodyfat,
  family=QuantReg(tau=0.75),control = boost_control(
  mstop =1200,nu = 0.1,trace = FALSE))
```

The coefficients of the 25th percentile regression are:

```
> coef(fit25,off2int=TRUE)
```

(Intercept)	age	waistcirc
-58.24524124	0.03352251	0.16259137
hipcirc	elbowbreadth	kneebreadth
0.32444309	-0.19165652	0.81968750
anthro3a	anthro3b	anthro3c
1.27773808	1.69581972	3.90369290
anthro4		
0.60159287		

The coefficients of the 75th percentile regression are:

```
> coef(fit75,off2int=TRUE)
```

(Intercept)	age	waistcirc
-64.4564045	0.0130879	0.2779561
hipcirc	elbowbreadth	kneebreadth
0.3954600	-0.1283315	0.8464489
anthro3a	anthro3b	anthro3c
2.5044739	2.2238765	0.9028559

There appears to be considerable overlap in the set of explanatory variables which explain both percentiles. However, it is noticeable that `anthro4` only appears in the 25th percentile regression. To investigate further we use a 10-fold cross validated estimate of the empirical risk to choose the optimal number of boosting iterations for each quantile regression model. The empirical risk is calculated using the `cvrisk` function.

```
> cv10f25 <- cv(model.weights(fit25), type = "kfold"
+   ,B=10)
> cvm25 <- cvrisk(fit25, folds = cv10f25)

> cv10f75 <- cv(model.weights(fit75), type = "kfold"
+   ,B=10)
> cvm75 <- cvrisk(fit75, folds = cv10f75)

> mstop(cvm25)
[1] 994

> mstop(cvm75)
[1] 1200

> fit25[mstop(cvm25)]
> fit75[mstop(cvm75)]
```

The optimal stopping iteration is 994 for the 25th percentile model and 1200 for the 75th percentile model.

Given our optimal model we calculate a bootstrapped confidence interval (90% level) around each of the parameters. For illustration we only use 50 bootstraps, in practice you should use at least 1,000.

```
> CI25 <- confint(fit25, B =50 , level = 0.9)
> CI75 <- confint(fit75, B =50 , level = 0.9)
```

The results for the 25th percentile model are:

```
> CI25
```

	Bootstrap Confidence Intervals	
	5%	95%
(Intercept)	-62.91485919	-43.84055225
age	-0.02032722	0.06116936
waistcirc	0.02054691	0.36351636
hipcirc	0.14107346	0.48121885
elbowbreadth	-1.02553929	0.23022726

kneebreadth	-0.03557848	1.37440528
anthro3a	0.00000000	4.11999837
anthro3b	0.00000000	3.18697032
anthro3c	2.09103968	6.49277249
anthro4	0.00000000	1.36671922

The results for the 75th percentile model are:

> CI75

	Bootstrap	Confidence Intervals	
		5%	95%
(Intercept)	-72.77094640	-49.1977452	
age	-0.00093535	0.0549349	
waistcirc	0.10970068	0.3596424	
hipcirc	0.16935863	0.5051908	
elbowbreadth	-1.36476008	0.5104004	
kneebreadth	0.00000000	1.9214064	
anthro3a	-0.76595108	6.1881146	
anthro3b	0.00000000	6.6016240	
anthro3c	-0.31714787	5.6599846	
anthro4	-0.27462792	2.8194127	

Both models have `waistcirc` and `hipcirc` as significant explanatory variables. `anthro3c` is the only other significant variable and then only for the lower weight percentiles.

Technique 81

Expectile Regression

Expectile regression¹³⁶ is used for estimating the conditional expectiles of a response variable given a set of attributes. Having multiple expectiles at different levels provides a more complete picture of the conditional distribution of the response variable.

A boosted version of Expectile regression can be run using the package **mboost** with the **glmboost** function:

```
glmboost(z ~ ., data ,family=ExpectReg(),control  
,...)
```

Key parameters include **z** the continuous response variable; **data** the data set of independent variables; **family=ExpectReg()**; **control** which limits the number of boosting iterations and the shrinkage parameter.

We illustrate the use of Expectile regression using the **bodyfat** data frame. Details of this data set are discussed on page 62. We continue our analysis of page 544. Taking the final models developed for the quantile regression and fitting them using **ExpectReg**, (note Step 1 is outlined on page 544).

Step 2: Estimate Model & Assess Fit

We set **fit25** as the 25th expectile regression using the three significant variables identified using a quantile regression (**waistcirc**, **hipcirc**, **anthro3c**, see page 544). We also use **fit75** as the 75th expectile regression using the two significant variables (**waistcirc** and **hipcirc**) identified via quantile regression.

Ten-fold cross validation is used to determine the optimal number of iterations for each of the models. The optimal number is captured using **mstop(cvm25)** and **mstop(cvm75)** for the 25th expectile regression and 75th expectile regression respectively.

```
> fit25<-glmboost(DEXfat~ waistcirc+hipcirc+anthro3c ,  
+ data = bodyfat,family=ExpectReg(tau=0.25),control  
+ = boost_control(mstop =1200,nu = 0.1,trace =  
+ FALSE))  
  
> fit75 <- glmboost(DEXfat ~ waistcirc+hipcirc , data  
+ = bodyfat,family=ExpectReg(tau=0.25),control =  
+ boost_control(mstop =1200,nu = 0.1,trace = FALSE))  
  
> cv10f25 <- cv(model.weights(fit25), type = "kfold"  
+ ,B=10)  
> cvm25 <- cvrisk(fit25, folds = cv10f25)  
> cv10f75 <- cv(model.weights(fit75), type = "kfold"  
+ ,B=10)  
> cvm75 <- cvrisk(fit75, folds = cv10f75)  
  
> mstop(cvm25)  
[1] 219  
  
> mstop(cvm75)  
[1] 826  
> fit25[mstop(cvm25)]
```

Notice that the optimal number of iterations is 219 and 826 for the 25th expectile regression and 75th expectile regression respectively. The 90% confidence intervals are estimated for each model (we use a small bootstrap of $B = 50$, in practice you will want to use at least 1000).

```
> CI25 <- confint(fit25, B =50 , level = 0.9)  
> CI75 <- confint(fit75, B =50 , level = 0.9)  
  
> CI25  
      Bootstrap Confidence Intervals  
              5%          95%  
(Intercept) -61.0691823 -48.9044440  
waistcirc     0.1035435   0.2891155  
hipcirc       0.3204619   0.5588739  
anthro3c      4.4213144   7.2163618  
  
> CI75  
      Bootstrap Confidence Intervals  
              5%          95%
```

```
(Intercept) -60.1313272 -41.8993208  
waistcirc      0.2478238   0.4967517  
hipcirc        0.2903714   0.6397651
```

As we might have expected all the variables included in each model are statistically significant. Both models have `waistcirc` and `hipcirc` as significant explanatory variables. It seems `anthro3c` is important in the lower weight percentile. These findings are similar to those observed by the boosted quantile regression models discussed on page 544

Discrete Response Boosted Regression

Technique 82

Logistic Regression

Boosted logistic regression can be run using the package `mboost` with the `glmboost` function:

```
glmboost(z ~ ., data ,family=Binomial(link = c("logit")),control,...)
```

Key parameters include `z` the discrete response variable; `data` the data set of independent variables; `family=Binomial`; `control` which limits the number of boosting iterations and the shrinkage parameter.

Step 1 : Load Required Packages

We will build the model using the `Sonar` data set discussed on page 482.

```
> library("mboost")
> data("Sonar", package="mlbench")
```

Step 2 : Prepare Data & Tweak Parameters

We use the first 157 observations to train the model and the remaining 51 observations as the test set.

```
> set.seed(107)
> n=nrow(Sonar)
> train <- sample(1:n, 157, FALSE)
```

Step 3 : Estimate Model

We fit the model setting the maximum number of iterations to 200. Then a 10-fold cross validated estimate of the empirical risk is used to choose the optimal number of boosting iterations.

Empirical risk is calculated using the `cvrisk` function.

```
> fit <- glmboost(Class~ ., data = Sonar[train,] ,  
+ family=Binomial(link = c("logit")),control = boost  
+ _control(mstop =200 ,nu = 0.1,trace = FALSE))  
  
> cv10f <- cv(model.weights(fit), type = "kfold",B  
+ =10)  
> cvm <- cvrisk(fit, folds = cv10f)  
  
> mstop(cvm)  
[1] 123  
  
> fit[mstop(cvm)]
```

The optimal number is 123 (from `mstop(cvm)`). We use `fit[mstop(cvm)]` to capture the model estimates at this iteration.

Step 4: Classify Data

The optimal model is used to classify the test data set. A threshold parameter (`thresh`) is used to translate the model scores into classes “M” and “R”. Finally, the confusion table is printed out.

```
> pred<-predict(fit, newdata=Sonar[-train,], type =  
+ "response")  
  
> thresh <- 0.5  
> predFac <- cut(pred, breaks=c(-Inf, thresh, Inf),  
+ labels=c("M", "R"))  
  
> table(Sonar$Class[-train],predFac, dnn=c("actual",  
+ "predicted"))  
          predicted  
actual      M   R  
      M 22   7  
      R   4 18
```

The overall error rate of the model is $\frac{7+4}{51} = 21.5\%$.

Technique 83

Probit Regression

Boosted probit regression can be run using the package `mboost` with the `glmboost` function:

```
glmboost(z ~ ., data ,family=Binomial(link = c("probit")),control,...)
```

Key parameters include `z` the discrete response variable; `data` the data set of independent variables; `family=Binomial`; `control` which limits the number of boosting iterations and the shrinkage parameter. Step 1 and 2 are outlined on page 552.

Step 3 : Estimate Model

The model is fit by setting the maximum number of iterations to 200. Then a 10-fold cross validated estimate of the empirical risk is used to choose the optimal number of boosting iterations. The empirical risk is calculated using the `cvrisk` function.

```
> fit <- glmboost(Class~ ., data = Sonar[train,],
+ family=Binomial(link = c("probit")),control =
+ boost_control(mstop =200 ,nu = 0.1,trace = FALSE))

> cv10f <- cv(model.weights(fit), type = "kfold",B
+ =10)
> cvm <- cvrisk(fit, folds = cv10f)

> mstop(cvm)
[1] 199
```

```
> fit[mstop(cvm)]
```

The optimal number is 199 (from `mstop(cvm)`). We use `fit[mstop(cvm)]` to capture the model estimates at this iteration.

Step 4: Classify Data

The optimal model is used to classify the test data set. A threshold parameter (`thresh`) is used to translate the model scores into classes “M” and “R”. Finally, the confusion table is printed out.

```
> pred<-predict(fit, newdata=Sonar[-train,], type =
  "response")

> thresh <- 0.5
> predFac <- cut(pred, breaks=c(-Inf, thresh, Inf),
  labels=c("M", "R"))

> table(Sonar$Class[-train],predFac, dnn=c("actual",
  "predicted"))

  predicted
actual   M   R
      M 21   8
      R   4 18
```

The overall error rate of the model is $\frac{8+4}{51} = 23.5\%$.

Boosted Regression for Count & Ordinal Response Data

Technique 84

Poisson Regression

Modeling count variables is a common task for the data scientist. When the response variable (y_i) is a count variable following a Poisson distribution with a mean that depends on the covariates $\{x_1, \dots, x_k\}$ the Poisson regression model is used. A boosted Poisson regression can be run using the package `mboost` with the `glmboost` function:

```
glmboost(z ~ ., data, family=Poisson(), control, ...)
```

Key parameters include `z` the response variable; `data` the data set of independent variables; `family=Poisson`; `control` which limits the number of boosting iterations and the shrinkage parameter.

NOTE... ↗

If $y_i \sim Poisson(\lambda_i)$ then the mean is equal to λ_i and the variance is also equal to λ_i . Therefore, in a Poisson regression both the mean and the variance depend on the covariates i.e.

$$\ln(\lambda_i) = \beta_0 + \beta_1 x_{1i} + \dots + \beta_k x_{ki} \quad (84.1)$$

Step 1: Load Required Packages

The required packages and data are loaded as follows:

```
> library("mboost")
> library("MixAll")
> data(DebTrivedi)
> set.seed(465)
```

```
> f<-ofp~hosp + health + numchron + gender + school  
+ privins
```

The number of physician office visits `ofp` is the response variable. The covariates are - `hosp` (number of hospital stays), `health` (self-perceived health status), `numchron` (number of chronic conditions), as well as the socioeconomic variables `gender`, `school` (number of years of education), and `privins` (private insurance indicator).

Step 2: Estimate Model & Assess Fit

The model is fit using `glmboost` with the maximum number of iterations equal to 1200. The parameter estimates are shown in Table 25.

```
> fit<-glmboost(f,data=DebTrivedi,family=Poisson(),  
control = boost_control(mstop =1200,nu = 0.1,trace  
= FALSE))  
  
> round(coef(fit,off2int=TRUE),3)
```

(Intercept)	hosp	healthpoor
1.029	0.165	0.248
healthexcellent	numchron	gendermale
-0.362	0.147	-0.112
school	privinsyes	
0.026	0.202	

Table 25: Initial coefficient estimates

A 10-fold cross validated estimate of the empirical risk is used to choose the optimal number of boosting iterations. The empirical risk is calculated using the `cvrisk` function. The parameter estimates are shown in Table 26.

```
> cv10f <- cv(model.weights(fit), type = "kfold",B  
=10)  
> cvm <- cvrisk(fit, folds = cv10f)  
  
> mstop(cvm)  
[1] 36  
  
> fit[mstop(cvm)]
```

```
> round(coef(fit, off2int=TRUE), 3)
```

(Intercept)	hosp	healthpoor
1.039	0.166	0.243
healthexcellent	numchron	gendermale
-0.347	0.147	-0.107
school	privinsyes	
0.026	0.197	

Table 26: Cross validated coefficient estimates

The optimal number of iterations is 36 (from `mstop(cvm)`). We use `fit[mstop(cvm)]` to capture the model estimates at this iteration and print out the estimates using `round(coef(fit, off2int=TRUE), 3)`. We note that all the parameter estimates for the optimal fit model are close to those observed prior to the 10-fold cross validation.

Finally, we estimate a 90% confidence interval for each of the parameters using a small bootstrap sample of 50 (in practice you should use at least 1000).

```
> CI <- confint(fit, B = 50, level = 0.9)
```

```
> CI
      Bootstrap Confidence Intervals
                           5%          95%
(Intercept)       0.9589353  1.18509474
hosp            0.1355033  0.21464596
healthpoor       0.1663162  0.32531872
healthexcellent -0.4494058 -0.21435073
numchron         0.1207087  0.16322440
gendermale        -0.1591000 -0.04314812
school           0.0143659  0.03374010
privinsyes       0.1196714  0.23438877
```

All of the covariates are statistically significant and contribute to explaining the number of physician office visits.

Technique 85

Negative Binomial Regression

Negative binomial regression is often used for over-dispersed count outcome variables. A boosted negative binomial regression can be run using the package `mboost` with the `glmboost` function:

```
glmboost(z ~ ., data ,family=NBinomial(),control  
,...)
```

Key parameters include `z` the count response variable; `data` the data set of independent variables; `family=NBinomial`; `control` which limits the number of boosting iterations and the shrinkage parameter.

☛ PRACTITIONER TIP ☛

A feature of the data frame `DebTrivedi` reported by the original researchers (see page 86) is that the data has a high degree of unconditional over dispersion relative to the standard Poisson model. Overdispersion simply means that the data has greater variability than would be expected based on the given statistical model (in this case Poission). One way to handle over dispersion is to use the Negative Binomial regression model.

Step 1 is outlined on page 558.

Step 2: Estimate Model & Assess Fit

We continue with the `DebTrivedi` data frame and estimate a Negative binomial regression using the same set of covariates as discuss on page 558. The

model is estimated using `glmboost` with the maximum number of iterations equal to 1200. The parameter estimates are shown in Table 27.

```
> fit<-glmboost(f,data=DebTrivedi,family=NBinomial(c(0,100)),control = boost_control(mstop =1200,nu = 0.1,trace = FALSE))

> round(coef(fit,off2int=TRUE),3)
```

(Intercept)	hosp	healthpoor
0.929	0.218	0.305
healthexcellent	numchron	gendermale
-0.342	0.175	-0.126
school	privinsyes	
0.027	0.224	

Table 27: Initial coefficient estimates

Although the values of the estimates are somewhat different from those on page 559, the signs of the coefficient remain consistent.

A 10-fold cross validated estimate of the empirical risk is used to choose the optimal number of boosting iterations. The empirical risk is calculated using the `cvrisk` function. The parameter estimates are shown in Table 28.

```
> cv10f <- cv(model.weights(fit), type = "kfold",B =10)
> cvm <- cvrisk(fit, folds = cv10f)

> mstop(cvm)
[1] 312

> fit[mstop(cvm)]

> round(coef(fit,off2int=TRUE),3)
```

The optimal number of iterations is 312 (from `mstop(cvm)`). Therefore we use `fit[mstop(cvm)]` to capture the model estimates at this iteration and print out the estimates using `round(coef(fit,off2int=TRUE),3)`. We note that all the parameter estimates for the optimal fit model are very close to those observed prior to the 10-fold cross validation.

Finally, we estimate a 90% confidence interval for each of the parameters using a small bootstrap sample of 50 (in practice you should use at least 1000).

(Intercept)	hosp	healthpoor
0.940	0.216	0.300
healthexcellent	numchron	gendermale
-0.336	0.174	-0.122
school	privinsyes	
0.026	0.220	

Table 28: Cross validated coefficient estimates

```
> CI <- confint(fit, B =50 , level = 0.9)

> CI
      Bootstrap Confidence Intervals
                           5%          95%
(Intercept)      0.88730254  1.06806832
hosp            0.17222572  0.24115790
healthpoor       0.21205691  0.35635293
healthexcellent -0.47308517 -0.18802910
numchron         0.14796503  0.19085736
gendermale        -0.18601165 -0.05170944
school           0.01638135  0.03447440
privinsyes       0.13273522  0.25977131
```

All of the covariates are statistically significant.

Technique 86

Hurdle Regression

Hurdle regression is used for modeling count data where there is over dispersion and excessive zero counts in the outcome variable. It can be run using the package `mboost` with the `glmboost` function:

```
glmboost(z ~ ., data ,family=Hurdle(),control,...)
```

Key parameters include `z` the response variable; `data` the data set of independent variables; `family=Hurdle`; `control` which limits the number of boosting iterations and the shrinkage parameter.

NOTE... ↗

A feature of the `DebTrivedi`, reported by the researchers Deb and Trivedi (see page 86) is that the data include a high proportion of zero counts, corresponding to zero recorded demand over the sample interval. One way to handle an excess of zero counts is to fit a negative binomial regression model to the non-zero counts. This can be achieved using the `Hurdle` function. In the hurdle approach the process that determines the zero/nonzero count threshold is different from the process that determines the count once the hurdle (zero in this case) is crossed. Once the hurdle is crossed, the data are assumed to follow the density for a truncated negative binomial distribution.

Step 1 is outlined on page 558.

Step 2: Estimate Model & Assess Fit

We continue with the `DebTrivedi` data frame and estimate a Hurdle regression using the same set of covariates as discuss on page 558. The model is estimated using `glmboost` with the maximum number of iterations equal to 1200. The parameter estimates are shown in Table 29.

```
> fit<-glmboost(f,data=DebTrivedi,family=Hurdle(c  
  (0,100)),control = boost_control(mstop =3000,nu =  
  0.1,trace = FALSE))  
  
> round(coef(fit,off2int=TRUE),3)
```

(Intercept)	hosp	healthpoor
-3.231	0.352	0.533
healthexcellent	numchron	gendermale
-0.586	0.299	-0.206
school	privinsyes	
0.044	0.395	

Table 29: Initial coefficient estimates

(Intercept)	hosp	healthpoor
-3.189	0.341	0.509
healthexcellent	numchron	gendermale
-0.567	0.294	-0.192
school	privinsyes	
0.042	0.378	

Table 30: Cross validated coefficient estimates

Although the sign of the intercept and values of the estimated coefficients are somewhat different from those on 559, the signs of the coefficient are preserved. A 10-fold cross validated estimate of the empirical risk is used to choose the optimal number of boosting iterations. The empirical risk is calculated using the `cvrisk` function. The parameter estimates are shown in Table 30.

```
> cv10f <- cv(model.weights(fit), type = "kfold",B  
  =10)  
> cvm <- cvrisk(fit, folds = cv10f)
```

```
> mstop(cvm)
[1] 1489
```

The optimal number is 1489 (from `mstop(cvm)`). We use `fit[mstop(cvm)]` to capture the model estimates at this iteration and print out the estimates using `round(coef(fit, off2int=TRUE), 3)`. We note that all the parameter estimates for the optimal fit model are very close to those observed prior to the 10-fold cross validation.

Technique 87

Proportional Odds Model

The Proportional Odds Model is a class of generalized linear models used for modelling the dependence of an ordinal response on discrete or continuous covariates.

It is used when it is not possible to measure the response variable on an interval scale. In bio medical research, for instance, constructs such as self-perceived health can be measured on an ordinal scale (“very unhealthy,” “unhealthy”, “healthy”, “very healthy”). A boosted version of the model can be run using the package `mboost` with the `glmboost` function:

```
glmboost(z ~ ., data ,family=PropOdds(),control ,...)
```

Key parameters include the response `Z` which is an ordered factor; `data` the data set of explanatory variable; `family = PropOdds()`; `control` which limits the number of boosting iterations and the shrinkage parameter.

Step 1 : Load Required Packages & Tweak Data

The required packages and data are loaded as follows:

```
> library("mboost")
> library("ordinal")
> data("wine")
> set.seed(125)
```

The package `ordinal` contains the data frame `wine` which is used in the analysis. It also contains the function `clm` which we use later to estimate a non-boosted version of the Proportional Odds Model. Further details of `wine` are given on page 95.

Step 2: Estimate Model & Assess Fit

We estimate the model using `rating` as the response variable and `contact` and `temp` as the covariates. The model is estimated using `glmboost` with the maximum number of iterations equal to 1200. This followed by a five-fold cross validation using the minimum empirical risk to determine the optimal number of iterations.

```
> fit<-glmboost(rating ~ temp + contact,data= wine,
  family=PropOdds(),control = boost_control(mstop
  =1200,nu = 0.1))

> cv5f <- cv(model.weights(fit), type = "kfold",B=5)
> cvm <- cvrisk(fit, folds = cv5f)
> mstop(cvm)
[1] 167

> round(cvm[mstop(cvm)],3)
[1] 1.42

> fit[mstop(cvm)]
```

The optimal number of iterations is 167 (from `mstop(cvm)`) with a empirical risk of 1.42. We use `fit[mstop(cvm)]` to capture the model estimates at this iteration and print out the parameter estimates using `round(coef(fit,off2int=TRUE),3)`.

The estimate of a Proportional Odds Model using the function `clm` is also reported.

```
> round(coef(fit,off2int=TRUE),3)
(Intercept)      tempwarm   contactyes
      -1.508        2.148        1.230

> fit2<- clm(rating ~ temp + contact, data = wine,
  link="logit")

> round(fit2$coefficients [5:6],3)
 tempwarm   contactyes
      2.503        1.528
```

Although the coefficients differ between the boosted and non-boosted version of the model, both models yield the same class predictions:

```
> pred<-predict(fit,type="class")
```

```
> pred2<-predict(fit2,type="class")  
  
> table(pred)  
pred  
1 2 3 4 5  
0 18 36 18 0  
  
> table (pred2)  
pred2  
1 2 3 4 5  
0 18 36 18 0
```

Finally, we compare the fitted model with the actual observations and calculate the overall error rate.

```
> tb<-table(wine$rating,pred, dnn=c("actual", "predicted"))  
  
> tb  
      predicted  
actual 1 2 3 4 5  
      1 0 4 1 0 0  
      2 0 9 12 1 0  
      3 0 5 16 5 0  
      4 0 0 5 7 0  
      5 0 0 2 5 0  
  
> error <- 1-(sum(diag(tb))/sum(tb))  
  
> round (error,3)  
[1] 0.556
```

The error rate for the model is 56%.

Boosted Models for Survival Analysis

NOTE... ↗

Survival analysis is concerned with studying the time between entry to a study and a subsequent event (such as death). The objective is to use a statistical model to simultaneously explore the effects of several explanatory variables on survival. Two popular approaches are the Cox Proportional Hazard Model and Accelerated Failure Time Models.

Cox Proportional Hazard Model

The Cox Proportional Hazard Model is a statistical technique for exploring the relationship between survival (typically of a patient) and several explanatory variables. It takes the form:

$$h_i(t) = \exp(\beta_1 X_{1i} + \dots + \beta_k X_{ki}) h_0(t) \quad (87.1)$$

where $h_i(t)$ is the hazard function for the i th individual at time t , $h_0(t)$ is the baseline hazard function and X_1, \dots, X_k are the explanatory covariates.

The model provides an estimate of the treatment effect on survival after adjustment for other explanatory variables. In addition, it is widely used in medical statistics because it provides an estimate of the hazard (or risk) of death for an individual, given their prognostic variables.

NOTE... ↗

The hazard function is the probability that an individual will experience an event (for example, death) within a small time interval, given that the individual has survived up to the beginning of the interval. In the medical context it can therefore be interpreted as the risk of dying at time t .

Accelerated Failure Time Models

Parametric accelerated failure time (AFT) models provide an alternative to the (semi-parametric) Cox proportional hazards model for statistical modeling of survival data¹³⁷. Unlike the Cox proportional hazards model, the AFT approach models survival times directly and assumes that the effect of a covariate is to accelerate or decelerate the life course of a response by some constant.

The AFT model treats the logarithm of survival time as the response variable and includes an error term that is assumed to follow a particular distribution.

Equation 87.2 shows the log-linear form of the AFT model for the i th individual, where $\log T_i$ is the log-transformed survival time, X_1, \dots, X_k are the explanatory covariates and ε_i represents the residual or unexplained variation in the log-transformed survival times, while μ and σ are intercept and scale parameter, respectively¹³⁸.

$$\log T_i = \mu + \beta_1 X_{1i} + \dots + \beta_k X_{ki} + \sigma \varepsilon_i \quad (87.2)$$

Under the AFT model parameterization, the distribution chosen for T_i dictates the distribution of the error term ε_i . Popular survival time distributions include the Weibull distribution, the log-logistic distribution and the log-normal distribution.

☛ PRACTITIONER TIP ☚

If the baseline hazard function is known to follow a Weibull distribution, accelerated failure and proportional hazards assumptions are equivalent.

Model	Empirical Risk
Weibull AFT	0.937
Lognormal AFT	1.046
Log-logistic AFT	0.934
Cox PH	2.616
Gehan	0.267

Table 31: Estimation of empirical risk using the rhDNase data frame

Assessing Fit

One of the first steps the data scientist faces in fitting survival models is to determine which distribution should be specified for the survival times T_i . One approach is to fit a model for each distribution and choose the model which minimizes the Akaike's Information Criterion (AIC)¹³⁹ or similar criteria. An alternative is to choose the model which minimizes the cross-validated estimation of empirical risk.

As an example, Table 31 shows the cross-validated estimation of empirical risk for various models using the `rhDNase` data frame.

Technique 88

Weibull Accelerated Failure Time Model

The Weibull Accelerated Failure Time Model is one of the most popular distributional choices for modeling survival data. It can be run using the package `mboost` with the `glmboost` function:

```
glmboost(z ~ ., data ,family=Weibull(),control ,...)
```

Key parameters include `z` the survival response variable (note we set `z=Surv(time, status)` where `Surv` is an a survival object constructed using the `survival` package);`data` the data set of explanatory variable; `family = Weibull()`; `control` which limits the number of boosting iterations and the shrinkage parameter.

Step 1 : Load Required Packages & Tweak Data

The required packages and data are loaded as follows:

```
> library("mboost")
> library("simexaft")
> library("survival")
> data("rhDNase")
> set.seed(465)
```

Forced expiratory volume (FEV) was considered a risk factor and was measured twice at randomization (`rhDNase$fev` and `rhDNase$fev2`). We take the average of the two measurements as an explanatory variable. The response is defined as the logarithm of the time from randomization to the first pulmonary exacerbation measured in the object `survreg(Surv(time2, status))`.

```
> rhDNase$fev.ave <- (rhDNase$fev + rhDNase$fev2)/2
> z<-Surv(rhDNase$time2, rhDNase$status)
```

Step 2: Estimate Model & Assess Fit

We estimate a model using `z` as the response variable, `rhDNase$fev.ave` and `trt` (a treatment assignment indicator taking values 1 if patient receive rhDNase and 0 if patient receive placebo). The model is fit using `glmboost` with the maximum number of iterations equal to 1200. This followed by a five-fold cross validation using the minimum empirical risk to determine the optimal number of iterations.

```
> fit<-glmboost(Surv(time2, status) ~ trt + fev.ave,
  data= rhDNase,family=Weibull(),control = boost_
  control(mstop =1200,nu = 0.1))

> cv5f <- cv(model.weights(fit), type = "kfold",B=5)
> cvm <- cvrisk(fit, folds = cv5f)

> mstop(cvm)
[1] 500

> round(cvm[mstop(cvm)],3)
[1] 0.937
```

```
> fit[mstop(cvm)]
```

The optimal number of iterations is 500 (from `mstop(cvm)`) with a empirical risk of 0.937. We use `fit[mstop(cvm)]` to capture the model estimates at this iteration and print out the estimates using `round(coef(fit,off2int=TRUE),3)`.

We then fit a non-boosted Weibull AFT model, using the `survreg` function from the survival package and compare the parameter estimates to the boosted model. This is followed by obtaining an estimate of the boosted models scale parameter using the function `nuisance()`.

```
> fit1 <- survreg(Surv(time2, status) ~ trt + fev.
  ave, data = rhDNase, dist = "weibull")

> round(coef(fit,off2int=TRUE),3)
(Intercept)          trt      fev.ave
```

4.531	0.348	0.019
-------	-------	-------

```
> round(coef(fit1, off2int=TRUE), 3)
(Intercept)          trt      fev.ave
4.518           0.357     0.019

> round(nuisance(fit), 3)
[1] 0.92
```

The boosted model has a similar `fev.ave` to the non-boosted AFT model. There is a small difference in the estimated values of the intercept and `trt`.

Technique 89

Lognormal Accelerated Failure Time Model

The Lognormal Accelerated Failure Time Model can be run using the package `mboost` with the `glmboost` function:

```
glmboost(z ~ ., data ,family=Lognormal(),control  
,...)
```

Key parameters include `z` the survival response variable (note we set `z=Surv(time2, status)` where `Surv` is an a survival object constructed using the `survival` package);`data` the data set of explanatory variable; `family = Lognormal()`; `control` which limits the number of boosting iterations and the shrinkage parameter.

Details on step 1 are given on 573.

Step 2: Estimate Model & Assess Fit

We estimate a model using `z` as the response variable, `rhDNase$fev.ave` and `trt` (a treatment assignment indicator taking values 1 if patient receive rhDNase and 0 if patient receive placebo). The model is fit using `glmboost` with the maximum number of iterations equal to 1200. This followed by a five-fold cross validation using the minimum empirical risk to determine the optimal number of iterations.

```
> fit<-glmboost(z ~ trt + fev.ave,data= rhDNase,  
family=Lognormal(),control = boost_control(mstop  
=1200,nu = 0.1))  
  
> cv5f <- cv(model.weights(fit), type = "kfold",B=5)
```

```
> cvm <- cvrisk(fit, folds = cv5f)

> mstop(cvm)
[1] 456

> round(cvm[mstop(cvm)], 3)
[1] 1.046

> fit[mstop(cvm)]
```

The optimal number of iterations is 456 (from `mstop(cvm)`) with a empirical risk of 1.046. We use `fit[mstop(cvm)]` to capture the model estimates at this iteration and print out the estimates using `round(coef(fit, off2int=TRUE), 3)`.

We then fit a non-boosted Lognormal AFT model, using the `survreg` function from the survival package and compare the parameter estimates to the boosted model. This is followed by obtaining an estimate of the boosted models scale parameter using the function `nuisance()`.

```
> fit1 <- survreg(Surv(time2, status) ~ trt + fev.
  ave, data = rhDNase, dist = "lognormal")

> round(coef(fit, off2int=TRUE), 3)
(Intercept)          trt      fev.ave
    4.103        0.408       0.021

> round(coef(fit1, off2int=TRUE), 3)
(Intercept)          trt      fev.ave
    4.083        0.424       0.022

> round(nuisance(fit), 3)
[1] 1.441
```

The boosted model has a similar `fev.ave` to the non-boosted AFT model. There is a much larger difference in the estimated value `trt` (0.408 boosted AFT versus 0.424 non-boosted Lognormal AFT).

Technique 90

Log-logistic Accelerated Failure Time Model

The Log-logistic Accelerated Failure Time Model can be run using the package `mboost` with the `glmboost` function:

```
glmboost(z ~ ., data ,family=Loglog(),control ,...)
```

Key parameters include `z` the survival response variable (note we set `z=Surv(time2, status)` where `Surv` is an a survival object constructed using the `survival` package); `data` the data set of explanatory variable; `family = Loglog()`; `control` which limits the number of boosting iterations and the shrinkage parameter.

Details on step 1 are given on 573.

Step 2: Estimate Model & Assess Fit

We estimate a model using `z` as the response variable, `rhDNase$fev.ave` and `trt` (a treatment assignment indicator taking values 1 if patient receive rhDNase and 0 if patient receive placebo). The model is fit using `glmboost` with the maximum number of iterations equal to 1200. This followed by a five-fold cross validation using the minimum empirical risk to determine the optimal number of iterations.

```
> fit<-glmboost(z ~ trt + fev.ave,data= rhDNase ,
  family=Loglog(),control = boost_control(mstop
  =1200,nu = 0.1))

> cv5f <- cv(model.weights(fit), type = "kfold",B=5)
> cvm <- cvrisk(fit, folds = cv5f)
```

```
> mstop(cvm)
[1] 445

> round(cvm[mstop(cvm)],3)
[1] 0.934

> fit[mstop(cvm)]
```

The optimal number of iterations is 445 (from `mstop(cvm)`) with a empirical risk of 0.934. We use `fit[mstop(cvm)]` to capture the model estimates at this iteration and print out the estimates using `round(coef(fit, off2int=TRUE),3)`.

We then fit a non-boosted Log-logistic AFT model, using the `survreg` function from the survival package and compare the parameter estimates to the boosted model. This is followed by obtaining an estimate of the boosted models scale parameter using the function `nuisance()`.

```
> fit1 <- survreg(Surv(time2, status) ~ trt + fev.
  ave, data = rhDNase, dist = "loglogistic")

> round(coef(fit, off2int=TRUE),3)
(Intercept)          trt      fev.ave
        4.110       0.384       0.020

> round(coef(fit1, off2int=TRUE),3)
(Intercept)          trt      fev.ave
        4.093       0.396       0.021

> round(nuisance(fit),3)
[1] 0.794
```

The boosted model has a similar `fev.ave` to the non-boosted AFT model. There is a larger difference in the estimated value `trt` (0.384 boosted AFT versus 0.396 non-boosted Log-logistic AFT).

Technique 91

Cox Proportional Hazard Model

The Cox Proportional Hazard Model can be run using the package `mboost` with the `glmboost` function:

```
glmboost(z ~ ., data ,family=CoxPH(),control ,...)
```

Key parameters include `z` the survival response variable (note we set `z=Surv(time2, status)` where `Surv` is an a survival object constructed using the `survival` package); `data` the data set of explanatory variable; `family = CoxPH()`; `control` which limits the number of boosting iterations and the shrinkage parameter.

Details on step 1 are given on 573.

Step 2: Estimate Model & Assess Fit

We estimate a model using `z` as the response variable, `rhDNase$fev.ave` and `trt` (a treatment assignment indicator taking values 1 if patient receive rhDNase and 0 if patient receive placebo). The model is fit using `glmboost` with the maximum number of iterations equal to 1200. This followed by a five-fold cross validation using the minimum empirical risk to determine the optimal number of iterations.

```
> fit<-glmboost(z ~ trt + fev.ave,data= rhDNase ,
  family=CoxPH(),control = boost_control(mstop
  =1200,nu = 0.1))

> cv5f <- cv(model.weights(fit), type = "kfold",B=5)
> cvm <- cvrisk(fit, folds = cv5f)
```

```
> mstop(cvm)
[1] 263

> round(cvm[mstop(cvm)],3)
[1] 2.616

> fit[mstop(cvm)]
```

The optimal number of iterations is 263 (from `mstop(cvm)`) with a empirical risk of 2.616. We use `fit[mstop(cvm)]` to capture the model estimates at this iteration and print out the estimates using `round(coef(fit, off2int=TRUE),3)`.

☛ PRACTITIONER TIP ☛

With boosting the whole business of model diagnostics is greatly simplified. For example, you can quickly boost an additive non-proportional hazards model in order to check if it fits your data better than a linear Cox model. If the two models perform roughly equivalent, you know that assuming proportional hazards and linearity is reasonable.

We then fit a non-boosted Cox Proportional Hazard Model model, using the `coxph` function from the survival package and compare the parameter estimates to the boosted model.

```
> fit1 <- coxph(Surv(time2, status) ~ trt + fev.ave
, data = rhDNase)

> round(coef(fit, off2int=TRUE),3)
(Intercept)          trt      fev.ave
1.431        -0.370       -0.020

> round(coef(fit1, off2int=TRUE),3)
trt fev.ave
-0.378 -0.021
```

Both models have similar treatment effects.

NOTE... ↗

For the Cox Proportional Hazard model a positive regression coefficient for an explanatory variable means that the hazard is higher, and thus the prognosis worse. Conversely, a negative regression coefficient implies a better prognosis for patients with higher values of that variable.

In Figure 91.1 we plot the predicted survivor function.

```
S1 <- survFit(fit)
plot(S1)
```

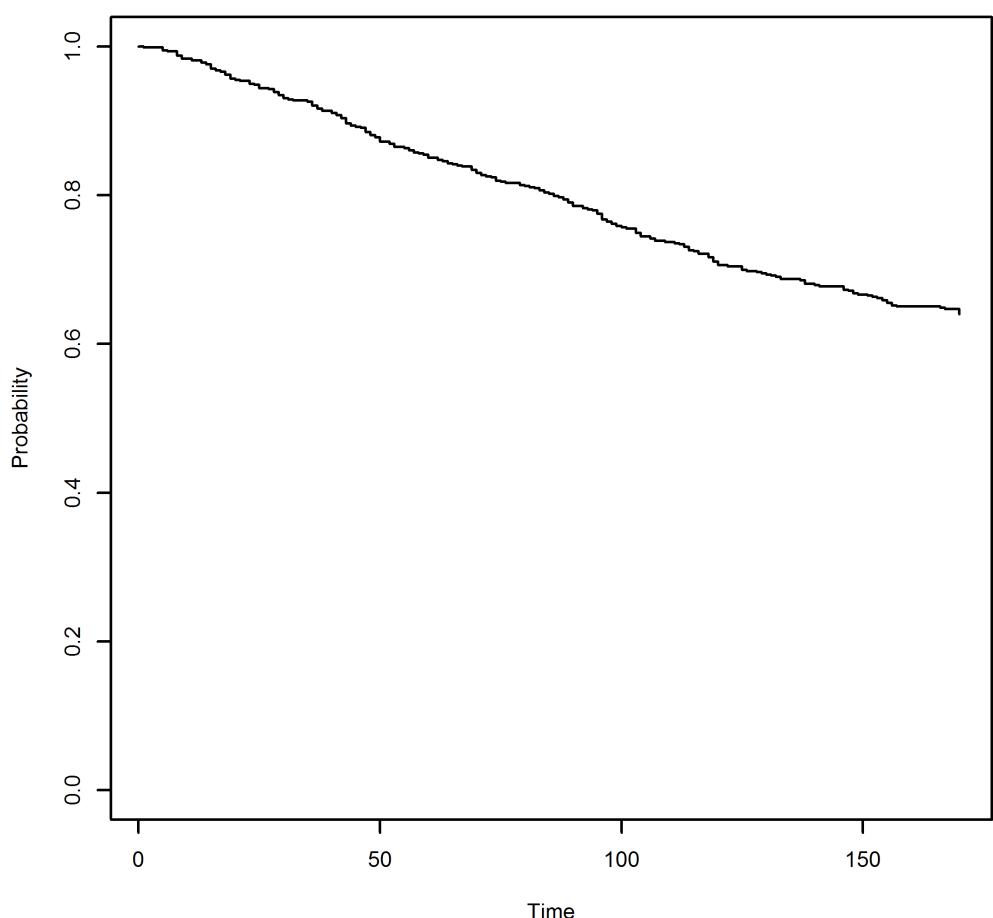


Figure 91.1: Cox Proportional Hazard Model predicted survivor function for rhDNase

Technique 92

Gehan Loss Accelerated Failure Time Model

The Gehan Loss Accelerated Failure Time Model calculates a rank-based estimation of survival data where the loss function is defined as the sum of the pairwise absolute differences of residuals. It can be run using the package `mboost` with the `glmboost` function:

```
glmboost(z ~ ., data ,family=Gehan(),control ,...)
```

Key parameters include `z` the survival response variable (note we `set z=Surv(time2, status)` where `z` is the response variable; `data` the data set of explanatory variable; `family = Gehan()`; `control` which limits the number of boosting iterations and the shrinkage parameter.

Details on step 1 are given on 573.

Step 2: Estimate Model & Assess Fit

We estimate a model using `z` as the response variable, `rhDNase$fev.ave` and `trt` (a treatment assignment indicator taking values 1 if patient receive rhDNase and 0 if patient receive placebo). The model is fit using `glmboost` with the maximum number of iterations equal to 1200. This followed by a five-fold cross validation using the minimum empirical risk to determine the optimal number of iterations.

```
> fit<-glmboost(z ~ trt + fev.ave,data= rhDNase ,
  family=Gehan(),control = boost_control(mstop
  =1200,nu = 0.1))

> cv5f <- cv(model.weights(fit), type = "kfold",B=5)
```

```
> cvm <- cvrisk(fit, folds = cv5f)

> mstop(cvm)
[1] 578

> round(cvm[mstop(cvm)], 3)
[1] 0.267

> fit[mstop(cvm)]

> round(coef(fit, off2int=TRUE), 3)
(Intercept)      trt      fev.ave
      3.718      0.403      0.022
```

The optimal number of iterations is 578 (from `mstop(cvm)`) with a empirical risk of 0.267. We use `fit[mstop(cvm)]` to capture the model estimates at this iteration and print out the parameter estimates using `round(coef(fit, off2int=TRUE), 3)`.

Notes

¹¹⁵Bauer, Eric, and Ron Kohavi. "An empirical comparison of voting classification algorithms: Bagging, boosting, and variants." *Machine learning* 36.1 (1998): 2.

¹¹⁶Friedman, Jerome, Trevor Hastie, and Robert Tibshirani. "Additive logistic regression: a statistical view of boosting (with discussion and a rejoinder by the authors)." *The annals of statistics* 28.2 (2000): 337-407.

¹¹⁷A type of proximal gradient method for learning from data. See for example:

1. Parikh, Neal, and Stephen Boyd. "Proximal algorithms." *Foundations and Trends in optimization* 1.3 (2013): 123-231.
2. Polson, Nicholas G., James G. Scott, and Brandon T. Willard. "Proximal Algorithms in Statistics and Machine Learning." *arXiv preprint arXiv:1502.03175* (2015).

¹¹⁸Schapire, Robert E. "The strength of weak learnability." *Machine learning* 5.2 (1990): 197-227.

¹¹⁹Cheepurupalli, Kusma Kumari, and Raja Rajeswari Konduri. "Noisy reverberation suppression using adaboost based EMD in underwater scenario." *International Journal of Oceanography* 2014 (2014).

¹²⁰For details on EMD and its use see the classic paper by Rilling, Gabriel, Patrick Flandrin, and Paulo Goncalves. "On empirical mode decomposition and its algorithms." *IEEE-EURASIP workshop on nonlinear signal and image processing*. Vol. 3. NSIP-03, Grado (I), 2003.

¹²¹See for example

1. Kusma Kumari and K. Raja Rajeshwari, "Application of EMD as a robust adaptive signal processing technique in radar/sonar communications," *International Journal of Engineering Science and Technology (IJEST)*, vol. 3, no. 12, pp. 8262– 8266, 2011;
2. Kusma Kumari Ch and K. Raja Rajeswari, "Enhancement of performance measures using EMD in noise reduction application," *International Journal of Computer Applications*, vol. 70,no. 5, pp. 10–14, 2013.

¹²²Karabulut, Esra Mahsereci, and Turgay Ibrikci. "Analysis of Cardiotocogram Data for Fetal Distress Determination by Decision Tree Based Adaptive Boosting Approach." *Journal of Computer and Communications* 2.09 (2014): 32.

¹²³See Newman, D.J., Heitttech, S., Blake, C.L. and Merz, C.J. (1998) UCI Repository of Machine Learning Databases. University California Irvine, Department of Information and Computer Science.

¹²⁴Creamer, Germán, and Yoav Freund. "Automated trading with boosting and expert weighting." *Quantitative Finance* 10.4 (2010): 401-420.

¹²⁵Sam, Kam-Tong, and Xiao-Lin Tian. "Vehicle logo recognition using modest adaboost and radial tchebichef moments." *International Conference on Machine Learning and Computing (ICMLC 2012)*. 2012.

¹²⁶Markoski, Branko, et al. "Application of Ada Boost Algorithm in Basketball Player Detection." *Acta Polytechnica Hungarica* 12.1 (2015).

¹²⁷Takiguchi, Tetsuya, et al. "An adaboost-based weighting method for localizing human brain magnetic activity." *Signal & Information Processing Association Annual Summit and Conference (APSIPA ASC)*, 2012 Asia-Pacific. IEEE, 2012.

¹²⁸Note that each sensor weight value calculated by Ada Boost provides an indication of how useful the MEG-sensor pair is for vowel recognition.

¹²⁹See the original paper by Freund Y, Schapire R (1996). “Experiments with a New Boosting Algorithm.” In “International Conference on Machine Learning,” pp. 148–156.

¹³⁰See the seminal work of Bauer E, Kohavi R. An Empirical Comparison of Voting Classification Algorithms: Bagging, Boosting, and Variants. *Journal of Machine Learning*. 1999;36:105.139. They report a 27% relative improvement using Ada Boost compared to a single decision tree.

Also take a look at the following:

1. Ridgeway G. The State of Boosting. *Computing Science and Statistics*. 1999;31:172{181.
2. Meir R, Ratsch G. An Introduction to Boosting and Leveraging. *Advanced Lectures on Machine Learning*. 2003;p. 118{183.

¹³¹Gorman, R. Paul, and Terrence J. Sejnowski. "Analysis of hidden units in a layered network trained to classify sonar targets." *Neural networks* 1.1 (1988): 75-89.

¹³²See Cohen, J. (1960). A coefficient of agreement for nominal data. *Educational and Psychological Measurement* 20: 37–46.

¹³³See for example:

1. Buhlmann P, Hothorn T. Boosting Algorithms: Regularization, Prediction and Model Fitting (with Discussion). *Statistical Science*. 2007;22:477{522.
2. Zhou ZH. Ensemble Methods: Foundations and Algorithms. *CRC Machine Learning & Pattern Recognition*. Chapman & Hall; 2012.
3. Schapire RE, Freund Y. Boosting: Foundations and Algorithms. MIT Press; 2012.

¹³⁴See Hastie, T.; Tibshirani, R.; Friedman, J. H. (2009). "10. Boosting and Additive Trees". *The Elements of Statistical Learning* (2nd ed.). New York: Springer. pp. 337–384.

¹³⁵SAMME stands for stagewise additive modeling using a multi-class exponential loss function.

¹³⁶For additional details see Newey, W. & Powell, J. (1987), ‘Asymmetric least squares estimation and testing’, *Econometrica* 55, 819–847.

¹³⁷See for example Wei LJ. The accelerated failure time model: a useful alternative to the Cox regression model in survival analysis. *Statist Med*. 1992;11:1871–1879.

¹³⁸See Collett D. *Modelling Survival Data in Medical Research*. 2. CRC Press; Boca Raton: 2003

¹³⁹See Akaike A. A new look at the statistical model identification. *IEEE Trans Autom Control*. 1974;19:716–723.

Congratulations!

You made it to the end. Here are three things you can do next.

1. Pick up your free copy of **12 Resources to Supercharge Your Productivity in R** at <http://www.auscov.com/tools.html>
2. Gift a copy of this book to your friends, co-workers, teammates or your entire organization.
3. If you found this book useful and have a moment to spare, I would really appreciate a short review. Your help in spreading the word is gratefully received.

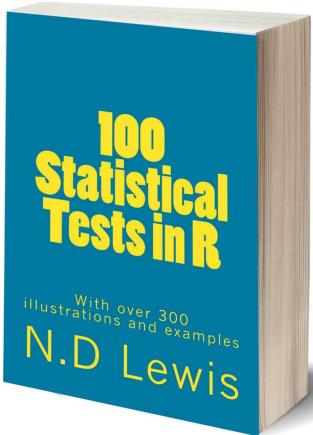
Good luck!

A handwritten signature in black ink that reads "Dr. Nigel D. Lewis". The signature is fluid and cursive, with "Dr." being a small prefix before "Nigel D. Lewis".

P.S. Thanks for allowing me to partner with you on your data analysis journey.

OTHER BOOKS YOU WILL ALSO ENJOY

Over 100 Statistical Tests at Your Fingertips!



100 Statistical Tests in R is designed to give you rapid access to one hundred of the most popular statistical tests.

It shows you, step by step, how to carry out these tests in the free and popular R statistical package.

The book was created for the applied researcher whose primary focus is on their subject matter rather than mathematical lemmas or statistical theory.

Step by step examples of each test are clearly described, and can be typed directly into R as printed on the page.

To accelerate your research ideas, over three hundred applications of statistical tests across engineering, science, and the social sciences are discussed.

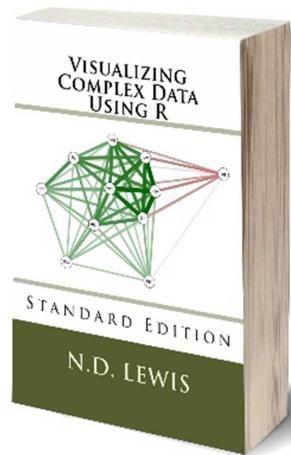
100 Statistical Tests in R - ORDER YOUR COPY TODAY!

"They Laughed As They Gave Me The Data To Analyze...But Then They Saw My Charts!"

Wish you had fresh ways to present data, explore relationships, visualize your data and break free from mundane charts and diagrams?

Visualizing complex relationships with ease using R begins here.

In this book you will find innovative ideas to unlock the relationships in your own data and create killer visuals to help you transform your next presentation from good to great.



**Visualizing Complex
Data Using R - ORDER
YOUR COPY TODAY!**