TIBCO Spotfire S+® 8.1
Guide to Statistics, Volume 2

November 2008

TIBCO Software Inc.

IMPORTANT INFORMATION

SOME TIBCO SOFTWARE EMBEDS OR BUNDLES OTHER TIBCO SOFTWARE. USE OF SUCH EMBEDDED OR BUNDLED TIBCO SOFTWARE IS SOLELY TO ENABLE THE FUNCTIONALITY (OR PROVIDE LIMITED ADD-ON FUNCTIONALITY) OF THE LICENSED TIBCO SOFTWARE. THE EMBEDDED OR BUNDLED SOFTWARE IS NOT LICENSED TO BE USED OR ACCESSED BY ANY OTHER TIBCO SOFTWARE OR FOR ANY OTHER PURPOSE.

USE OF TIBCO SOFTWARE AND THIS DOCUMENT IS SUBJECT TO THE TERMS AND CONDITIONS OF A LICENSE AGREEMENT FOUND IN EITHER A SEPARATELY EXECUTED SOFTWARE LICENSE AGREEMENT, OR, IF THERE IS NO SUCH SEPARATE AGREEMENT, THE CLICKWRAP END USER LICENSE AGREEMENT WHICH IS DISPLAYED DURING DOWNLOAD OR INSTALLATION OF THE SOFTWARE (AND WHICH IS DUPLICATED IN THE *TIBCO SPOTFIRE S+® INSTALLATION AND ADMINISTRATION GUIDE*). USE OF THIS DOCUMENT IS SUBJECT TO THOSE TERMS AND CONDITIONS, AND YOUR USE HEREOF SHALL CONSTITUTE ACCEPTANCE OF AND AN AGREEMENT TO BE BOUND BY THE SAME.

identification purposes only. This software may be available on multiple operating systems. However, not all operating system platforms for a specific software version are released at the same time. Please see the readme.txt file for the availability of this software version on a specific operating system platform.

Reference

The correct bibliographic reference for this document is as follows:

TIBCO Spotfire S+® 8.1 Guide to Statistics Volume 2 TIBCO Software Inc.

Technical Support

For technical support, please visit http://spotfire.tibco.com/support and register for a support account.

ACKNOWLEDGMENTS

TIBCO Spotfire S+ would not exist without the pioneering research of the Bell Labs S team at AT&T (now Lucent Technologies): John Chambers, Richard A. Becker (now at AT&T Laboratories), Allan R. Wilks (now at AT&T Laboratories), Duncan Temple Lang, and their colleagues in the statistics research departments at Lucent: William S. Cleveland, Trevor Hastie (now at Stanford University), Linda Clark, Anne Freeny, Eric Grosse, David James, José Pinheiro, Daryl Pregibon, and Ming Shyu.

TIBCO Software Inc. thanks the following individuals for their contributions to this and earlier releases of TIBCO Spotfire S+: Douglas M. Bates, Leo Breiman, Dan Carr, Steve Dubnoff, Don Edwards, Jerome Friedman, Kevin Goodman, Perry Haaland, David Hardesty, Frank Harrell, Richard Heiberger, Mia Hubert, Richard Jones, Jennifer Lasecki, W.Q. Meeker, Adrian Raftery, Brian Ripley, Peter Rousseeuw, J.D. Spurrier, Anja Struyf, Terry Therneau, Rob Tibshirani, Katrien Van Driessen, William Venables, and Judy Zeh.

TIBCO SPOTFIRE S+ BOOKS

The TIBCO Spotfire S+® documentation includes books to address your focus and knowledge level. Review the following table to help you choose the Spotfire S+ book that meets your needs. These books are available in PDF format in the following locations:

- In your Spotfire S+ installation directory (**SHOME\help** on Windows, **SHOME/doc** on UNIX/Linux).
- In the Spotfire S+ Workbench, from the **Help ► Spotfire S+ Manuals** menu item.
- In Microsoft® Windows®, in the Spotfire S+ GUI, from the **Help ► Online Manuals** menu item.

Spotfire S+ documentation.

Information you need if you...	**See the...**
Are new to the S language and the Spotfire S+ GUI, and you want an introduction to importing data, producing simple graphs, applying statistical models, and viewing data in Microsoft Excel®.	*Getting Started Guide*
Are a new Spotfire S+ user and need how to use Spotfire S+, primarily through the GUI.	*User's Guide*
Are familiar with the S language and Spotfire S+, and you want to use the Spotfire S+ plug-in, or customization, of the Eclipse Integrated Development Environment (IDE).	*Spotfire S+ Workbench User's Guide*
Have used the S language and Spotfire S+, and you want to know how to write, debug, and program functions from the **Commands** window.	*Programmer's Guide*
Are familiar with the S language and Spotfire S+, and you want to extend its functionality in your own application or within Spotfire S+.	*Application Developer's Guide*

Spotfire S+ documentation. (Continued)

Information you need if you...	**See the...**
Are familiar with the S language and Spotfire S+, and you are looking for information about creating or editing graphics, either from a **Commands** window or the Windows GUI, or using Spotfire S+ supported graphics devices.	*Guide to Graphics*
Are familiar with the S language and Spotfire S+, and you want to use the Big Data library to import and manipulate very large data sets.	*Big Data User's Guide*
Want to download or create Spotfire S+ packages for submission to the Comprehensive S-PLUS Archive Network (CSAN) site, and need to know the steps.	*Guide to Packages*
Are looking for categorized information about individual Spotfire S+ functions.	*Function Guide*
If you are familiar with the S language and Spotfire S+, and you need a reference for the range of statistical modelling and analysis techniques in Spotfire S+. Volume 1 includes information on specifying models in Spotfire S+, on probability, on estimation and inference, on regression and smoothing, and on analysis of variance.	*Guide to Statistics, Vol. 1*
If you are familiar with the S language and Spotfire S+, and you need a reference for the range of statistical modelling and analysis techniques in Spotfire S+. Volume 2 includes information on multivariate techniques, time series analysis, survival analysis, resampling techniques, and mathematical computing in Spotfire S+.	*Guide to Statistics, Vol. 2*

GUIDE TO STATISTICS CONTENTS OVERVIEW

Other Topics

CONTENTS

PREFACE

Introduction

Welcome to the *Spotfire S+ 8 Guide to Statistics, Volume 2*.

This book is designed as a reference tool for Spotfire S+ users who want to use the powerful statistical techniques in Spotfire S+. The *Guide to Statistics, Volume 2* covers a wide range of statistical and mathematical modeling. No single user is likely to tap all of these resources, since advanced topics such as survival analysis and time series are complete fields of study in themselves.

All examples in this guide are run using input through the **Commands** window, which is the traditional method of accessing the power of Spotfire S+. Many of the functions can also be run through the **Statistics** dialogs available in the graphical user interface. We hope that you find this book a valuable aid for exploring both the theory and practice of statistical modeling.

Online Version

The *Guide to Statistics, Volume 2* is also available online.

On Microsoft Windows®, from the **Help ► Online Manuals** menu, and in the **/help/statman2.pdf** file of your Spotfire S+ home directory.

On Solaris/Linux, in the **/doc/statman2.pdf** file of your Spotfire S+ home directory.

You can open and view this file using Adobe Acrobat Reader, which is required for reading all online manuals shipped with Spotfire S+.

The online version of the *Guide to Statistics, Volume 2* has particular advantages over print. For example, you can copy and paste example Spotfire S+ code into the **Commands** window and run it without having to type the function calls explicitly. (When doing this, be careful not to paste the greater-than ">" prompt character, and note that distinct colors differentiate between input and output in the online manual.)

A second advantage to the online guide is that you can perform full-text searches. To find information on a certain function, first search, and then browse through all occurrences of the function's name in the guide. A third advantage is in the contents and index entries: all entries are links; click an entry to go to the selected page.

Evolution of SPOTFIRE S+

Spotfire S+ has evolved considerably from its beginnings as a research tool. The contents of this guide have grown steadily, and will continue to grow, as the Spotfire S+ language is improved and expanded. As a result of these changes, some examples in the text might not match exactly the formatting of the output you obtain. However, the underlying theory and computations are as described here.

In addition to the range of functionality covered in this guide, there are additional modules, libraries, and user-written functions available from several sources. Refer to the *User's Guide* for more details.

Companion Guides

The *Guide to Statistics, Volume 2*, together with *Guide to Statistics, Volume 1*, is a companion volume to the *User's Guide* , the *Programmer's Guide*, and the *Application Developer's Guide*. These manuals, as well as the rest of the manual set, are available in electronic form. For a complete list of manuals, see the section TIBCO Spotfire S+ Books on page v.

This volume covers the following topics:

- Tree models
- Multivariate analysis, including factor analysis, principal components analysis, and discriminant analysis
- Cluster analysis
- Survival analysis
- Quality control charts
- Resampling methods (bootstrap and jackknife)
- Mathematical computing

The *Guide to Statistics, Volume 1* covers basic probability, descriptive statistics, statistical inference, regression techniques, and analysis of variance.

CLASSIFICATION AND REGRESSION TREES

19

INTRODUCTION

Tree-based modeling is an exploratory technique for uncovering structure in data, increasingly used for:

- devising prediction rules that can be rapidly and repeatedly evaluated
- screening variables
- assessing the adequacy of linear models
- summarizing large multivariate data sets

Tree-based models are useful for both classification and regression problems. In these problems, there is a set of classification or predictor variables (x), and a single-response variable (y).

If y is a factor, *classification* rules are of the form:

if $x_1 \leq 2.3$ *and* $x_3 \in \{A, B\}$

then y is most likely to be in level 5.

If y is numeric, *regression* rules for description or prediction are of the form:

if $x_2 \leq 2.3$ *and* $x_9 \in \{C, D, F\}$ *and* $x_5 \leq 3.5$

then the predicted value of y is 4.75.

A classification or regression tree is the collection of many such rules displayed in the form of a binary tree, hence the name. The rules are determined by a procedure known as *recursive partitioning*. Tree-based models provide an alternative to linear and additive models for regression problems, and to linear and additive logistic models for classification problems.

Compared to linear and additive models, tree-based models have the following advantages:

- Easier to interpret when the predictors are a mix of numeric variables and factors.
- Invariant to monotone re-expressions of predictor variables.
- More satisfactorily treat missing values.

- More adept at capturing nonadditive behavior.
- Allow more general (that is, other than of a particular multiplicative form) interactions between predictor variables.
- Can model factor response variables with more than two levels.

GROWING TREES

We describe the tree-growing function `tree` by presenting several examples. The `tree` function generates objects of class `"tree"`. This function automatically decides whether to fit a regression or classification tree, according to whether the response variable is numeric or a factor. We also show two types of displays generated by generic functions: a tree display produced by `plot` and a table produced by `print`.

In general, the response y and predictors x may be any combination of numeric or factor types. In fact, the predictors can be a mix of numeric *and* factor. However, no factor predictor can have more than 32 levels, and no factor response can have more than 128 levels. In both of the examples below, the predictors are all numeric. The numeric response example illustrates a regression tree. The factor response example illustrates a classification tree.

Numeric Response and Predictor

In the first example, we grow a *regression* tree relating the numeric response `Mileage` to the predictor variable `Weight` from the data frame `car.test.frame`. The resulting tree is given the name `auto.tree`, which is then plotted by the generic `plot` function and labeled by the generic `text` function (see Figure 19.1).

```
> attach(car.test.frame)
> auto.tree <- tree(Mileage ~ Weight, car.test.frame)
> plot(auto.tree,type = "u")
> text(auto.tree)
> title("A Tree-Based Model\nfor Mileage versus Weight")
```

In describing tree-based models, the terminology mimics real trees.

- *Root*: the top node of the tree
- *Leaf*: A terminal node of the tree
- *Split*: A rule for creating new branches

In growing a tree, the binary partitioning algorithm recursively splits the data in each node until either the node is homogeneous or the node contains too few observations (≤ 5, by default).

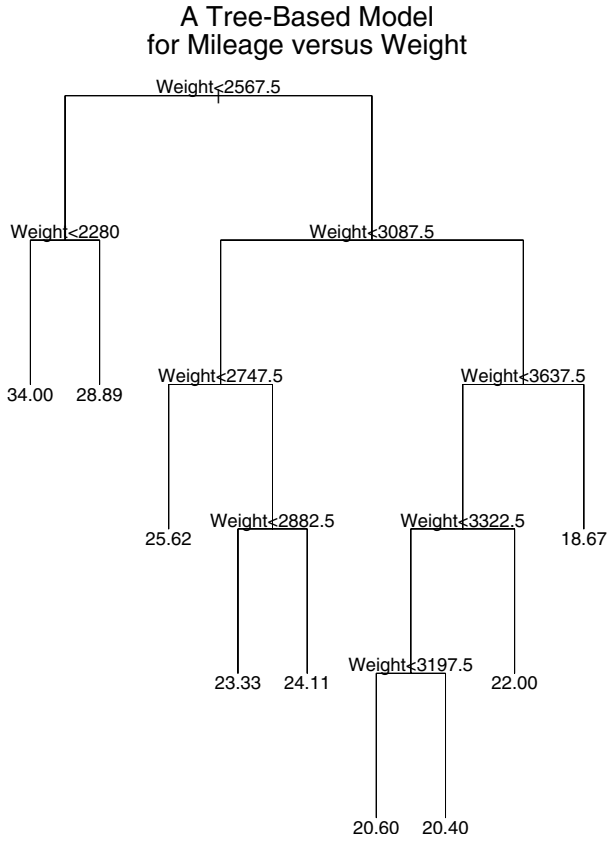

Figure 19.1: *Display of a tree-based model with a numeric response,* `Mileage`*, and one numeric predictor,* `Weight`*.*

In order to *predict* mileage from weight, one follows the path from the root, to a leaf, according to the splits at the interior nodes. The tree in Figure 19.1 is interpreted in the following way:

- Automobiles are first split according to whether they weigh less than 2567.5 pounds.
- If so, they are again split according to weight being less than 2280 pounds.
- Lighter cars (`< 2280` pounds) have a predicted mileage of 34 mpg.
- Heavier cars (`2280 <= Weight <= 2567.5`) have a mileage of 28.9 mpg.

- For those automobiles weighing more than 2567.5 pounds, seven weight classes are formed.
- The predicted mileage ranges from a high of 25.6 mpg to a low of 18.7 mpg.
- Overall, heavier cars get poorer mileage than lighter cars.
- It appears that doubling the weight of an automobile approximately halves its mileage.

Factor Response and Numeric Predictor

In this classification example, we model the probability of developing Kyphosis, using the `kyphosis` data frame with predictors `Age`, `Start`, and `Number`.

First, use boxplots to plot the distributions of the predictor variables as a function of `Kyphosis` in Figure 19.2. `Start` appears to be the single best predictor of `Kyphosis` since Kyphosis is more likely to be present among individuals with Start $\leq$ 12.

```
> kyph.tree <- tree(Kyphosis ~ Age + Number + Start,
+ data = kyphosis)
```

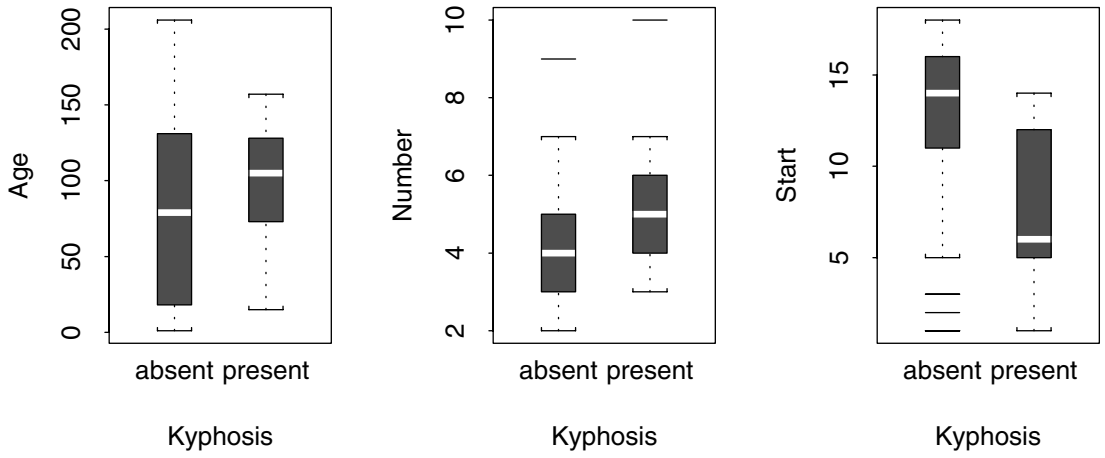

Figure 19.2: *Boxplots of the predictors of* `Kyphosis`.

Since `Kyphosis` is a factor response, the result `kyph.tree` is a *classification* tree.

Either the `formula` or `data` arguments to the `tree` function may be missing. Without the `formula` argument, a tree is constructed from the data frame using the first variable as the response. Hence, the Kyphosis example could have been constructed as follows:

```
> kyph.tree <- tree(kyphosis)
```

Without the `data` argument, the variables named in `formula` are expected to be in the search list. The Kyphosis tree could also have been grown with

```
> attach(kyphosis)
> kyph.tree <- tree(Kyphosis ~ Age + Number + Start)
```

The only meaningful operator on the right side of a formula is "+". Since tree-based models are invariant to monotone re-expressions of individual predictor variables, functions like `log`, `I`, and `^` have little use. Also, tree-based models capture interactions without explicit specification.

This time, we display the fitted tree using the generic function, `print`, which is called automatically simply by typing the name of the tree object. This tabular representation is most useful when the details of the fitting procedure are of interest. Indentation is added as a key to the underlying structure.

```
> kyph.tree

node), split, n, deviance, yval, (yprob)
      * denotes terminal node
1) root 81 83.230 absent ( 0.7901 0.20990 )
   2) Start<12.5 35 47.800 absent ( 0.5714 0.42860 )
     4) Age<34.5 10  6.502 absent ( 0.9000 0.10000 )
       8) Age<16 5  5.004 absent ( 0.8000 0.20000 ) *
       9) Age>16 5  0.000 absent ( 1.0000 0.00000 ) *
     5) Age>34.5 25 34.300 present ( 0.4400 0.56000 )
      10) Number<4.5 12 16.300 absent ( 0.5833 0.41670 )
        20) Age<127.5 7  8.376 absent ( 0.7143 0.28570 ) *
        21) Age>127.5 5  6.730 present ( 0.4000 0.60000 ) *
      11) Number>4.5 13 16.050 present ( 0.3077 0.69230 )
        22) Start<8.5 8  6.028 present ( 0.1250 0.87500 ) *
        23) Start>8.5 5  6.730 absent ( 0.6000 0.40000 ) *
   3) Start>12.5 46 16.450 absent ( 0.9565 0.04348 )
     6) Start<14.5 17 12.320 absent ( 0.8824 0.11760 )
```

```
 12) Age<59 5  0.000 absent ( 1.0000 0.00000 ) *
 13) Age>59 12 10.810 absent ( 0.8333 0.16670 )
   26) Age<157.5 7  8.376 absent ( 0.7143 0.28570 ) *
   27) Age>157.5 5  0.000 absent ( 1.0000 0.00000 ) *
7) Start>14.5 29  0.000 absent ( 1.0000 0.00000 ) *
```

The first number in each row of the output is a node number. The nodes are numbered to index the tree for quick identification. For a full binary tree, the nodes at depth d are integers n, $2^d \le n < 2^{d+1}$. Usually, a tree is not full, but the numbers of the nodes that are present are the same as they would be in a full tree.

In the `print` output, the nodes are ordered according to a depth-first traversal of the tree. Let us first examine one row of the output:

```
2) Start<12.5 35 47.800 absent ( 0.5714 0.42860 )
```

This row is for node 2. Following the node number is the split, `Start < 12.5`. This states the the observations in the parent (root) node with `Start < 12.5` were put into node 2.

The next number after the split is the number of observations, 35. The number 47.8 is the *deviance*, the measure of node heterogeneity used in the tree-growing algorithm. A perfectly homogeneous node has deviance zero. The fitted value, `yval`, of the node is `absent`. Finally, the numbers in parentheses (0.5714 0.42860), `yprob`, are the estimated probabilities of the observations in that node not having, and having, kyphosis. Therefore, the observations with `Start < 12.5` have a 0.5714 chance of not having kyphosis under this tree model.

An interpretation of the table follows:

- The split on `Start` partitions the 81 observations into groups of 35 and 46 individuals (nodes 2 and 3) with probability of `Kyphosis` 0.429 and 0.043, respectively.
- The group at node 2 is then partitioned into groups of 10 and 25 individuals (nodes 4 and 5) depending on whether `Age` is less than 34.5 months or not.
- The group at node 4 is divided in half depending on whether `Age` is less than 16 or not. If `Age > 16` none of the individuals have Kyphosis (probability of Kyphosis is 0). These subgoups are divided no further.

- The group at node 5 is subdivided into groups of size 12 and 13 depending on whether or not `Number` is less than 4.5. The respective probabilities of Kyphosis for these groups is 0.417 and 0.692.
- The procedure continues, yielding 10 distinct groups with probabilities of Kyphosis ranging from 0.0 to 0.875.
- Asterisks signify *terminal* nodes; that is, those that are not split.

DISPLAYING TREES

The generic functions `print`, `plot`, and `summary` work as expected for `tree` objects. We have already encountered the first two functions in the examples above. A further interesting feature of `plot` is that an optional `type` argument controls node placement. The `type` argument can have either of the two values:

- `""` produces nonuniform spacing as the default. The more important the parent split, the further the children node pairs are spaced from their parents.
- `"u"` produces uniform spacing.

In the car mileage example, we used uniform spacing in order to label the tree. However, if the goal is tree simplification, we gain insight into the relative importance of the splits by using the default `type`, that is, nonuniform spacing. This is shown in Figure 19.3.

When you first plot the tree using `plot`, the nodes and splits will be displayed without any text labels. The generic `text` function, described in the *Programmer's Guide*, uses the same arguments to rotate and adjust text in tree plots that it uses with most other types of plots.

The `summary` function has a tree-specific method which indicates the tree type (regression/classification), a record of how the tree was created, the residual mean deviance, and other information. The residual deviance is the sum, over all the observations, of terms which vary according to type (regression/classification) of tree. The residual mean deviance is then obtained after dividing by the degrees of freedom (number of observations minus the number of terminal nodes).

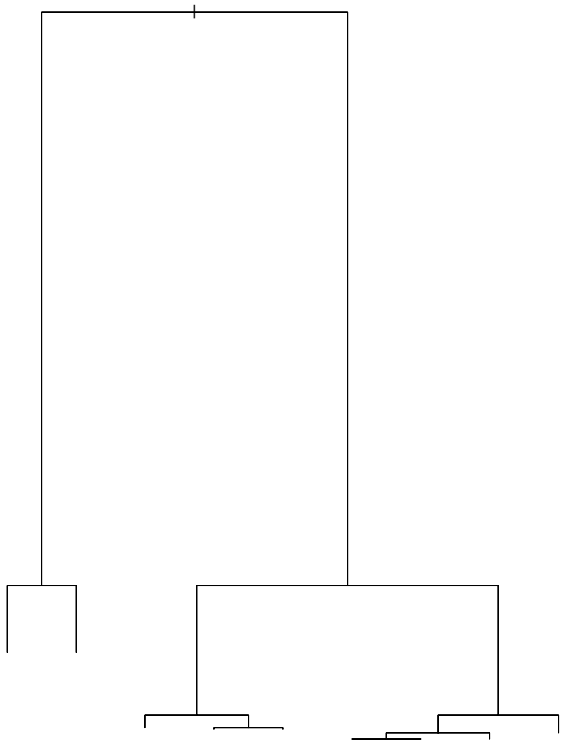

Figure 19.3: *Plot of the car mileage tree with non-uniform node placement.*

The following summary is typical for regression trees:

```
> summary(auto.tree)

Regression tree:
tree(formula = Mileage ~ Weight, data = car.test.frame)
Number of terminal nodes: 9
Residual mean deviance: 4.289 = 218.7 / 51
Distribution of residuals:
   Min. 1st Qu. Median Mean 3rd Qu. Max.
 -3.889 -1.111 0 0 1.083 4.375
```

The regression tree has nine terminal nodes. Under a normal (Gaussian) assumption, the terms in the residual mean deviance are the squared differences between the observations and the predicted values. See the section Prediction and Residuals for a discussion of prediction and residuals. The `summary` function also summarizes the distribution of residuals.

The following summary is typical for classification trees:

```
> summary(kyph.tree)

Classification tree:
tree(formula = Kyphosis ~ Age + Number + Start)
Number of terminal nodes: 10
Residual mean deviance: 0.5809 = 41.24 / 71
Misclassification error rate: 0.1235 = 10 / 81
```

Note that, for classification trees, the `summary` function gives the misclassification error rate instead of distribution of residuals. First, predicted classifications are obtained as described in the section Prediction and Residuals. The error rate is then obtained by counting the number of misclassified observations, and dividing by the number of observations. The terms in the residual mean deviance are based on the multinomial distribution (see Chambers and Hastie (1992)).

PREDICTION AND RESIDUALS

Once a tree is grown, an important use of the fitted tree is to predict the value of the response variable for a set of predictor variables.

For concreteness, consider just one observation x on the predictor variables. In prediction, the splits direct x through the tree. The *prediction* is taken to be the `yval` at the deepest node reached. Usually this corresponds to a leaf node. However, in certain situations, a prediction may reside in a nonterminal node (Chambers and Hastie (1992)). In particular this may happen if missing values occur in x, and the tree was grown with only complete observations.

The generic function `predict` has a tree-specific method. It takes a tree object and, optionally, a data frame as arguments. If the data frame is not supplied, `predict` returns the fitted values for the data originally used to construct the tree. The function returns predicted values either as a vector (the default) or a tree object (`type = "tree"`).

The residuals can then be obtained either by subtracting the fitted values from the response variable, or directly using the function `residuals`. Figure 19.4 presents a plot of the residuals versus the predicted values and a normal probability of the residuals for the `auto.tree` model.

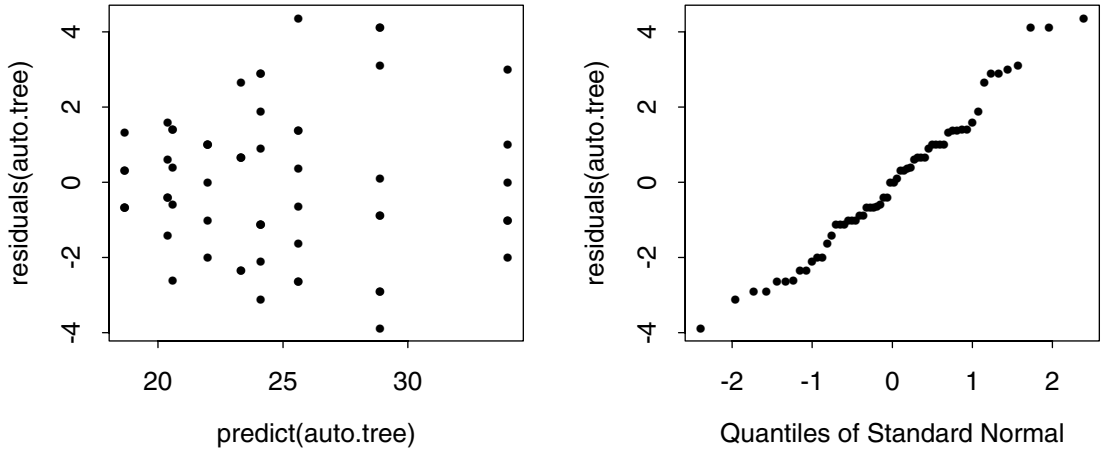

Figure 19.4: *Residuals versus predicted values and a normal probability plot of the residuals for a* `tree` *object.*

MISSING DATA

Missing values, NAs, can occur either in data used to build trees, or in a set of predictors for which the value of the response variable is to be predicted. For data used to build trees, the `tree` function permits NAs only in predictor variables, but only if the argument `na.action = na.tree.replace` or `na.action = na.tree.replace.all`. For any predictor with missing values, the `na.tree.replace` function creates a new factor variable with an added level named `"NA"` for the NAs; it leaves numeric predictors alone, even if they have NAs. The `na.tree.replace.all` function behaves like `na.tree.replace` for factor predictors, and converts numeric predictors with NAs to factors (based on quantiles), adding a separate level for NAs.

In prediction, suppose an observation is missing a value for the variable *V*. Further, suppose there were no missing values for *V* in the training data. The observation follows its path down the tree until it encounters a node whose split is based on *V*. The prediction is then taken to be the `yval` at that node. If values of several variables are missing, the observation stops at the first such variable split encountered.

To clarify this, let us return to the automobile example, where some of the data are missing values on the variable `Reliability`. We first fit a tree on the data with no missing values. The resulting tree is displayed in Figure 19.5. Notice the split on the variable `Reliability`.

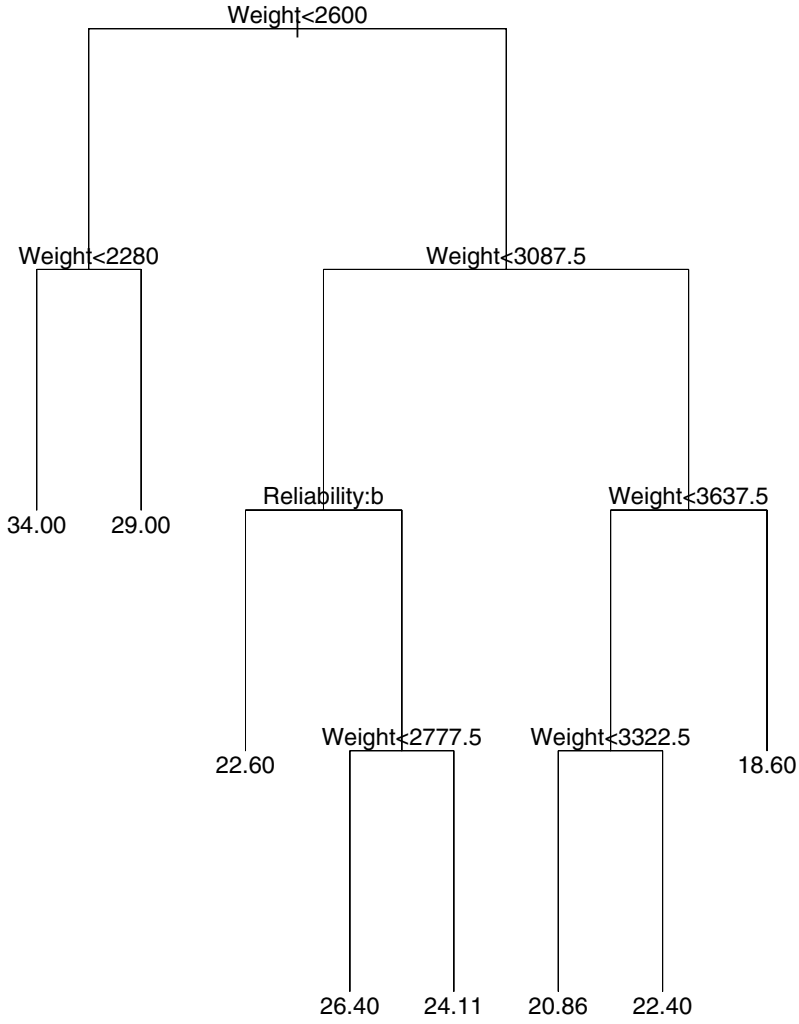

Figure 19.5: *Display of tree relating* `Mileage` *to* `Weight` *and* `Reliability`*. Missing values have been removed from the analysis.*

To create the tree shown in Figure 19.5, first create a new data set from `car.test.frame`, omitting those observations which are missing data for `Reliability`:

```
> car.test.no.miss <-
+ car.test.frame[!is.na(car.test.frame[,3]),]
```

Now grow the tree using the cleansed data:

```
> car.tree <- tree(Mileage ~ Weight + Reliability,
+ car.test.no.miss)
```

Next, we predict the data with values missing on `Reliability`, by extracting those observations that were omitted from `car.test.no.miss`, and then calling `predict` on the resulting data set:

```
> car.test.miss <-
+ car.test.frame[is.na(car.test.frame[,3]),]

> pred.miss <-
+ predict(car.tree, car.test.miss, type = "tree")
> pred.miss

node), split, n, deviance, yval
      * denotes terminal node
1) root 11 245.300 24.80
  2) Weight<2600 3 65.940 30.92
    4) Weight<2280 1 0.000 34.00 *
    5) Weight>2280 2 26.000 29.00 *
  3) Weight>2600 8 81.060 22.58
    6) Weight<3087.5 3 11.770 24.32
     12) Reliability:2 0 0.000 22.60 *
     13) Reliability:1,3,4,5 0 0.000 24.93
       26) Weight<2777.5 0 0.000 26.40 *
       27) Weight>2777.5 0 0.000 24.11 *
    7) Weight>3087.5 5 10.680 20.65
     14) Weight<3637.5 4 17.000 21.50
       28) Weight<3322.5 3 8.918 20.86 *
       29) Weight>3322.5 1 5.760 22.40 *
     15) Weight>3637.5 1 0.160 18.60 *
```

Notice that there are no observations in the nodes (12, 13, 26, 27) at or below the split on `Reliability`.

PRUNING AND SHRINKING

Since tree size is not limited in the growing process, a tree may be more complex than necessary to describe the data. Two functions assess the degree a tree can be simplified without sacrificing goodness of fit. The `prune.tree` function achieves parsimonious description by reducing the nodes on a tree, whereas the `shrink.tree` function *shrinks* each node towards its parent.

Both functions take the arguments listed below.

- `tree`: Fitted model object of class `"tree"`.
- `k`: Cost complexity parameter for `prune.tree` and shrinkage parameter for `shrink.tree`.
- `newdata`: A data frame containing the values at which predictions are required. If missing, the data used to grow the tree are used.

Pruning

Pruning successively snips off the least important splits. *Importance* of a subtree is measured by the cost-complexity measure:

$$D_k(T') = D(T') + k \cdot \text{size}(T')$$

where

$$\begin{aligned} D_k(T') &= \text{the deviance of the subtree } T', \\ \text{size}(T') &= \text{the number of terminal nodes of } T', \\ k &= \text{the cost-complexity parameter.} \end{aligned}$$

Cost-complexity pruning determines the subtree T that minimizes $D_k(T')$ over all subtrees. The larger the k, the fewer nodes there will be.

The `prune.tree` function takes a cost-complexity parameter argument `k`, which can be either a scalar or a vector. A scalar `k` defines one subtree of `tree` whereas a vector `k` defines a sequence of subtrees minimizing the cost-complexity measure. If the `k` argument is not supplied, a nested sequence of subtrees is created by recursively snipping off the least important splits.

Figure 19.6 shows the deviance decreasing as a function of the number of nodes and the cost-complexity parameter `k`.

```
# Establish the margin sizes.
> par(mai = c(1.25, 1.0, 1.25, 1.0))
> plot(prune.tree(kyph.tree))
> mtext("Reduction in Deviance
Continue string: With the Addition of Nodes",
+ line=5, cex=1.5)
```

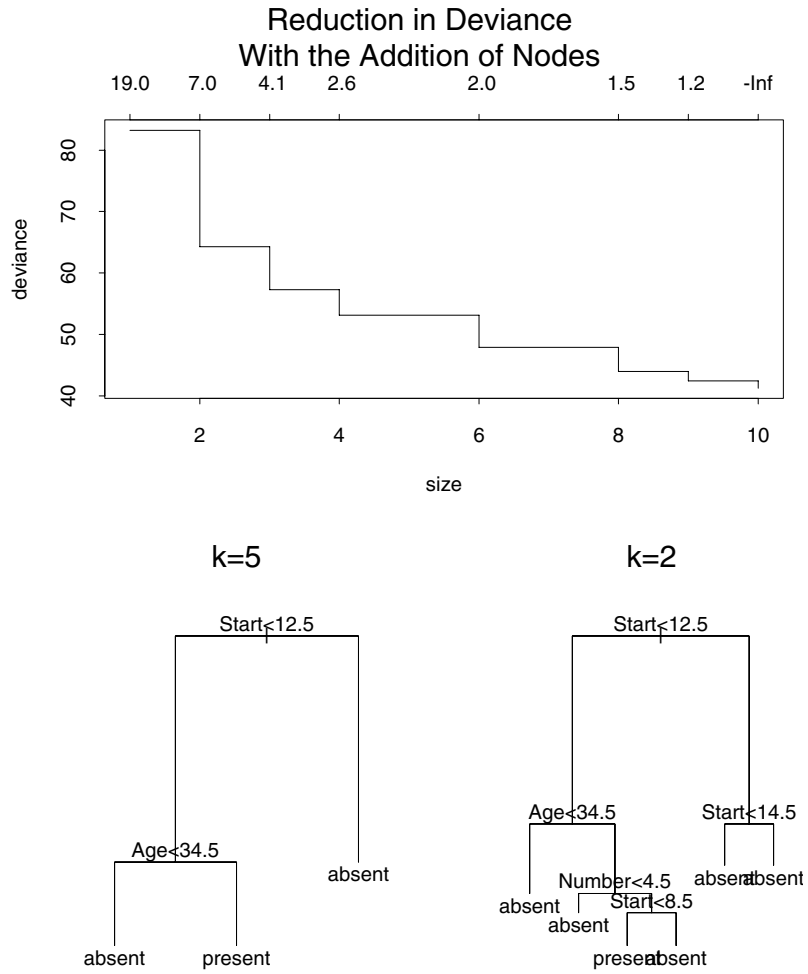

Figure 19.6: *A sequence of plots generated by the* `prune.tree` *function.*

Since over one half of the reduction in deviance is explained by the first three nodes, we limit the tree to three nodes.

```
> plot(prune.tree(kyph.tree, k = 5))
> text(prune.tree(kyph.tree, k = 5))
> title("k=5")
> summary(prune.tree(kyph.tree, k = 5))

Classification tree:
snip.tree(tree = kyph.tree, nodes = c(4, 5, 3))
Variables actually used in tree construction:
[1] "Start" "Age"
Number of terminal nodes:  3
Residual mean deviance:  0.734 = 57.25 / 78
Misclassification error rate: 0.1728 = 14 / 81
```

By comparing this to the summary of the full tree in the section Displaying Trees, we see that reducing the number of nodes from 10 to 3 simplifies the model, but at the cost of increased misclassification.

Increasing the complexity of the tree to 6 nodes drops the misclassification to a rate comparable to that of the full tree with 10 nodes:

```
> summary(prune.tree(kyph.tree, k = 2))

Classification tree:
snip.tree(tree = kyph.tree, nodes = c(10, 4, 6))
Number of terminal nodes:  6
Residual mean deviance:  0.6383 = 47.88 / 75
Misclassification error rate: 0.1358 = 11 / 81
```

Figure 19.6 shows `kyph.tree` pruned to 3 and 6 nodes.

Shrinking

Shrinking reduces the number of *effective* nodes by shrinking the fitted value of each node towards its parent node. Shrunken fitted values, for a shrinking parameter k, are computed according to the recursion:

$$(\text{node}) = k \cdot \bar{y}(\text{node}) + (1 - k) \cdot \hat{y}(\text{parent.}$$

Here,

$$\bar{y}(\text{node}) = \text{the usual fitted value for a node,}$$
$$\hat{y}(\text{parent}) = \text{the } \textit{shrunken} \text{ fitted value for the node's parent.}$$

The `shrink.tree` function *optimally* shrinks children nodes to their parent, based on the magnitude of the difference between $\bar{y}(\text{node})$ and $\bar{y}(\text{parent})$. The shrinkage parameter argument $0 < k < 1$ may be a scalar or a vector. A scalar `k` defines one shrunken version of `tree`, whereas a vector `k` defines a sequence of shrunken trees obtained by optimal shrinking for each value of `k`. If the `k` argument is not supplied, a nested sequence of subtrees is created by recursively shrinking the tree for a default sequence of values (roughly .05 to .91) of `k`.

Figure 19.7 shows the deviance decreasing as a function of the number of effective nodes and the shrinkage parameter, `k`. In the figure, note that there is no change other than a decrease in the residual mean deviance and an increase in the number of effective nodes.

```
# Establish the margin sizes.
> par(mai = c(1.25, 1.0, 1.25, 1.0))
> plot(shrink.tree(kyph.tree))
> mtext("Reduction in Deviance
Continue string: With Sequential Shrinking of Nodes",
+ line=5, cex=1.5)
```

Limit the tree to three effective nodes as done with pruning as follows:

```
> kyph.tree.sh.25 <- shrink.tree(kyph.tree, k = 0.25)
> plot(kyph.tree.sh.25)
> text(kyph.tree.sh.25)
> title("k = 0.25")
> summary(kyph.tree.sh.25)

Classification tree:
shrink.tree(tree = kyph.tree, k = 0.25)
Number of terminal nodes: 10
Effective number of terminal nodes: 2.8
Residual mean deviance: 0.7385 = 57.75 / 78.2
Misclassification error rate: 0.1358 = 11 / 81
```

The lower misclassification rate is maintained even with only three effective nodes.

Expand the tree to three effective nodes as follows:

```
> kyph.tree.sh.47 <- shrink.tree(kyph.tree, k = 0.47)
> plot(kyph.tree.sh.47)
> text(kyph.tree.sh.47)
> title("k = 0.47")
> summary(kyph.tree.sh.47)

Classification tree:
shrink.tree(tree = kyph.tree, k = 0.47)
Number of terminal nodes: 10
Effective number of terminal nodes: 6
Residual mean deviance: 0.6281 = 47.11 / 75
Misclassification error rate: 0.1358 = 11 / 81
```

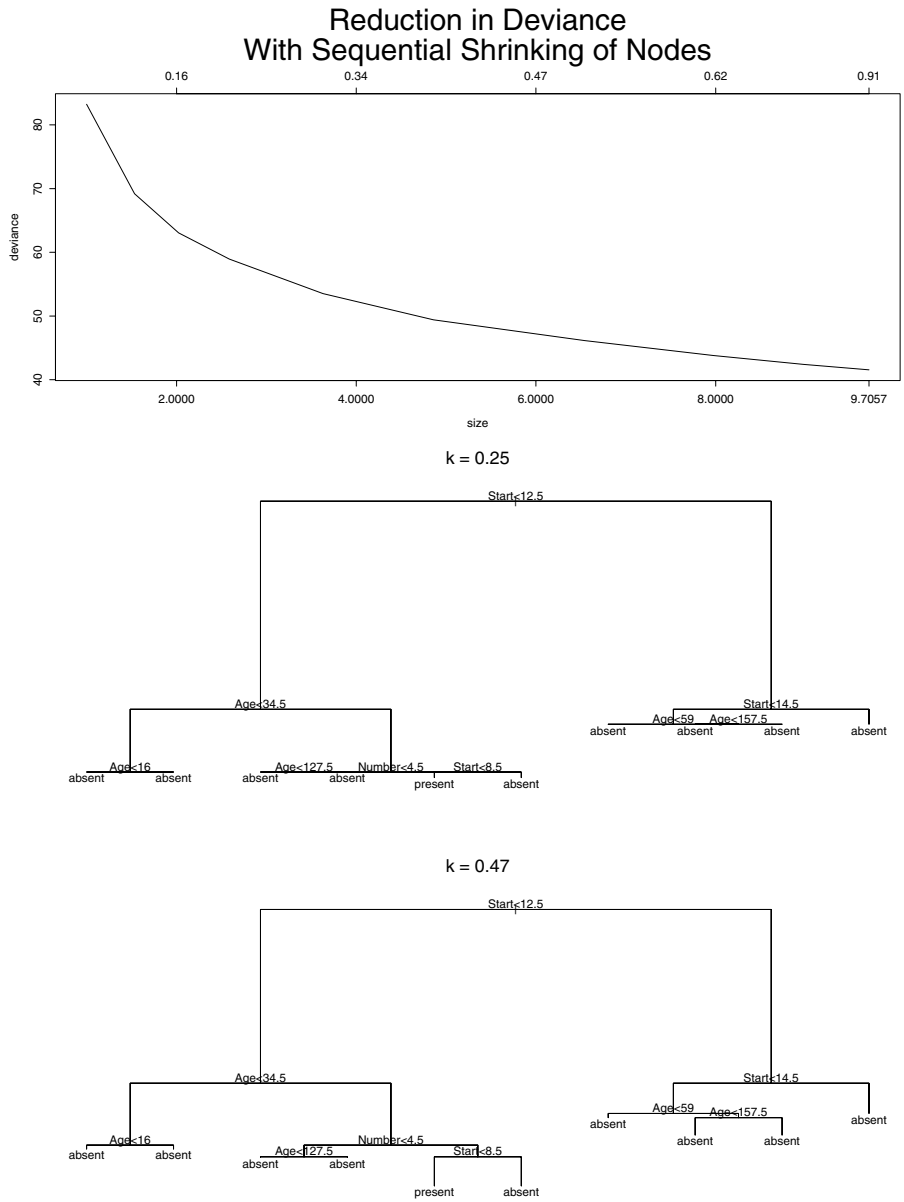

Figure 19.7: *A sequence of plots generated by the* `shrink.tree` *function.*

GRAPHICALLY INTERACTING WITH TREES

A number of Spotfire S+ functions use the tree metaphor to diagnose tree-based model fits. The functions are naturally grouped by components of trees: subtrees, nodes, splits, and leaves. Except for those that are specific to leaves, the functions allow you to interact graphically with trees, to perform a *what-if* analysis. You can also use these functions noninteractively by including a list of nodes as an argument. The goal is to better understand the fitted model, examine alternatives, and interpret the data in light of the model.

You can select subtrees from a large tree, and apply a common function (such as a plot) to the *stand* of resulting trees. Similarly, you can snip subtrees from the large tree, in order to gain resolution and label the top of the tree.

You can browse nodes to obtain important information too bulky to be usefully placed on a tree plot. You can obtain the names of observations that occur in a node. By examining the path (that is, the sequence of splits) that lead to a node, you can characterize the observations in that node.

You can compare optimal splits (generated by the tree-growing algorithm) to other potential splits. This helps to discover splits on variables that may shed light on the nature of the data. Any split divides the observations in a node into two groups. Therefore, you can compare the distribution of observations of a chosen variable in each of the two groups. This helps characterize the two groups, and also find variables with good discriminating abilities. You may regrow the tree, after designating a different split at a node.

The leaves of the trees represent the most homogeneous partitions of the data. You can investigate the differences across leaves by studying the distribution and summary statistics of chosen variables.

Subtrees

You can select or delete subtrees by subscripting the original tree, or by using `snip.tree` or `select.tree`, described below.

The function `snip.tree` function deletes subtrees; that is, it snips branches off a specified tree. One goal may be to gain resolution at the top of the tree so that it can be labeled.

The graphical interface for `snip.tree` proceeds as follows:

- The first left-click informs you of the change in tree deviance if that branch is snipped off.
- The second left-click removes the branch from the tree.
- To end the interactive process, click on either the middle or right mouse button.

Figure 19.8 shows the result of snipping three branches off `kyph.tree`.

```
> par(mfrow = c(3,1))
> plot(kyph.tree)
> plot(kyph.tree)
> kyph.tree.sn <- snip.tree(kyph.tree)

node number: 4
   tree deviance = 41.24
   subtree deviance = 42.74
node number: 10
   tree deviance = 42.74
   subtree deviance = 43.94
node number: 6
   tree deviance = 43.94
   subtree deviance = 47.88

> plot(kyph.tree.sn)
> text(kyph.tree.sn,cex = 1)
```

For noninteractive use, we can equivalently supply the node numbers in `snip.tree(kyph.tree,c(4,10,6))`. Negative subscripting is a convenient shorthand: `kyph.tree[-c(4,10,6)]`.

Similarly, the function `select.tree` function selects subtrees of a specified tree. For each node specified in the argument list or selected interactively, the function returns a tree object rooted at that node. These can in turn be plotted.

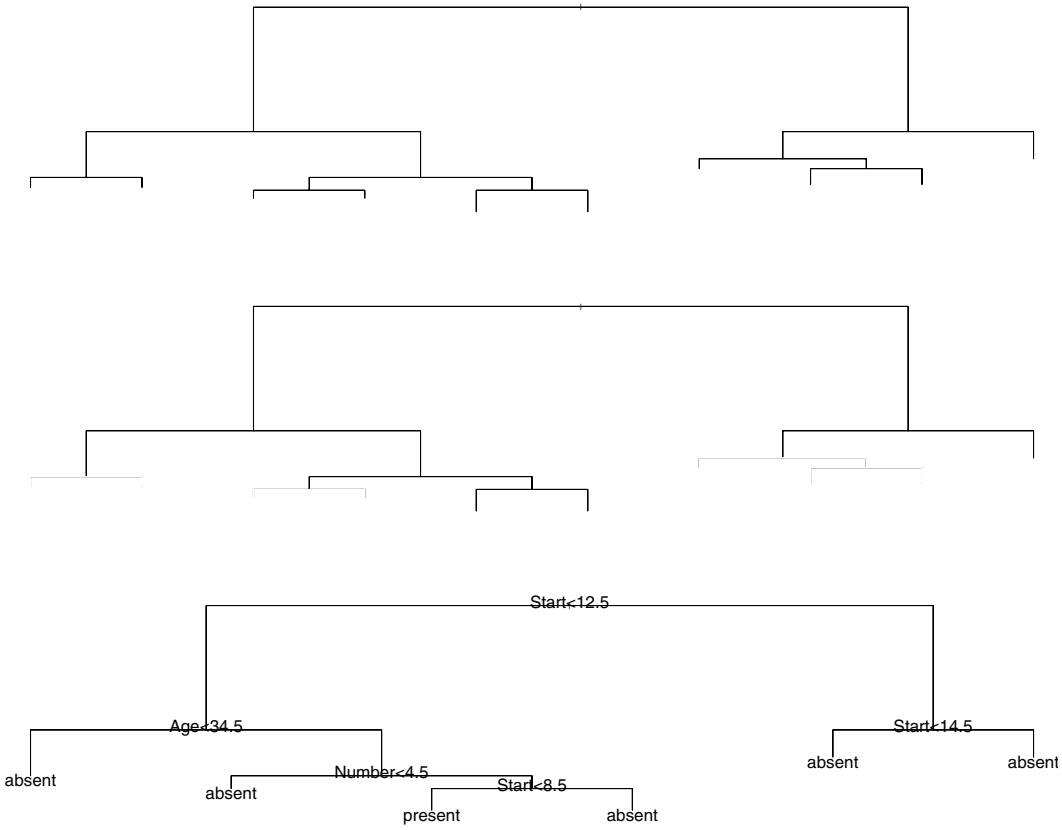

Figure 19.8: *A sequence of plots created by snipping branches from the top tree.*

Nodes

Several Spotfire S+ functions encourage the user to obtain more detailed information about nodes. Each function takes a tree object as a required argument, and accepts a list of nodes as an optional argument. If the node list is omitted, graphical interaction is expected. The functions return a list, with one component for each node.

The graphical interface for the node functions proceeds as follows:

- Left-click to receive information about a particular node in a tree.
- To end the interactive process, click on either the middle or right mouse button.

The `browser` function returns a summary of the information contained in a node. Interactively, you can obtain information on the second and fifth nodes of `kyph.tree` by first plotting the tree, and then calling `browser` as follows:

```
> plot(kyph.tree)
> browser(kyph.tree)

node number: 2
 split: Start<12.5
 n: 35
 dev: 47.800
 yval: absent
        absent present
[1,] 0.5714286 0.4285714
node number: 5
 split: Age>34.5
 n: 25
 dev: 34.300
 yval: present
     absent present
[1,]   0.44    0.56
```

You can also provide a list of nodes to `browser` and obtain the node information noninteractively:

```
> browser(kyph.tree, c(2,5))

     var  n      dev   yval splits.cutleft splits.cutright
2    Age 35 47.80357 absent          <34.5           >34.5
5 Number 25 34.29649 present           <4.5            >4.5
  yprob.absen yprob.present
2   0.5714286     0.4285714
5   0.4400000     0.5600000
```

The `identify` function is another generic function with a tree-specific method. The following noninteractive call lists the observations in the eighth and ninth nodes of `kyph.tree`:

```
> identify(kyph.tree, nodes = c(8,9))

$"8":
[1] "4" "14" "26" "29" "39"
$"9":
[1] "13" "21" "41" "68" "71"
```

The function `path.tree` returns the *path* (sequence of splits) from the root to any node of a tree. This is useful in cases where overplotting results if the tree is labeled indiscriminately. As an example, we interactively look at the path to the rightmost terminal node of the kyphosis tree:

```
> path.tree(kyph.tree)

 node number: 27
   root
   Start>12.5
   Start<14.5
   Age>59
   Age>157.5
```

By examining the path, we can determine that the children in this node are more than 157.5 months old, and the beginnings of the range of vertebrae involved are between 12.5 and 14.5.

Splits

The recursive partitioning algorithm underlying the `tree` function chooses a "best" set of splits that partition the predictor variable space into increasingly homogeneous regions. However, it is important to remember that it is only an algorithm. There may be other splits that also help you understand the data. The functions in this section help to examine alternative splits.

As in previous sections, the graphical interface for functions that examine splits proceed as follows:

- Left-click to receive information about the split at a particular node.
- To end the interactive process, click on either the middle or right mouse button.

With the `burl.tree` function, you can select a node and observe the *goodness of split* for each predictor in the model formula. The goodness-of-split criterion is the difference in deviance between the node and its children (defined by the tentative split). Large differences correspond to important splits.

The `burl.tree` function returns a list with one component for each variable. Each component contains the necessary information for generating the plots. The reduction in deviance is plotted against a quantity that depends upon the form of the predictor:

- If the predictor is numeric, each possible cut-point split is plotted.
- If the predictor is a factor variable, a decimal equivalent of the binary representation of each possible subset split is plotted. The plotting character is a string labeling the left split.

In the following example, competing splits are plotted for each of the four predictor variables in the `cu.summary` data frame. The resulting graph is displayed in Figure 19.9.

```
> reliab.tree <- tree(Reliability ~
+ Price + Country + Mileage + Type,
+ na.action = na.tree.replace.all, data = cu.summary)

> tree.screens() # Establish plotting regions

[1] 1 2

> plot(reliab.tree, type = "u")
> text(reliab.tree)
> burl.tree(reliab.tree) # Now click at the root node
```

The burl plot shows that the most important splits involve the variable `Country`. In the burl plot window for `Country`, the candidate splits are divided into two groups; there is a cluster of splits in the top of the window, and another in the bottom. The top cluster discriminates better than the bottom cluster, and the very best split is the one labeled `ef`. Moreover, the split `ef` occurs in all candidates that are in the top cluster. Therefore, we conclude that this is a meaningful split.

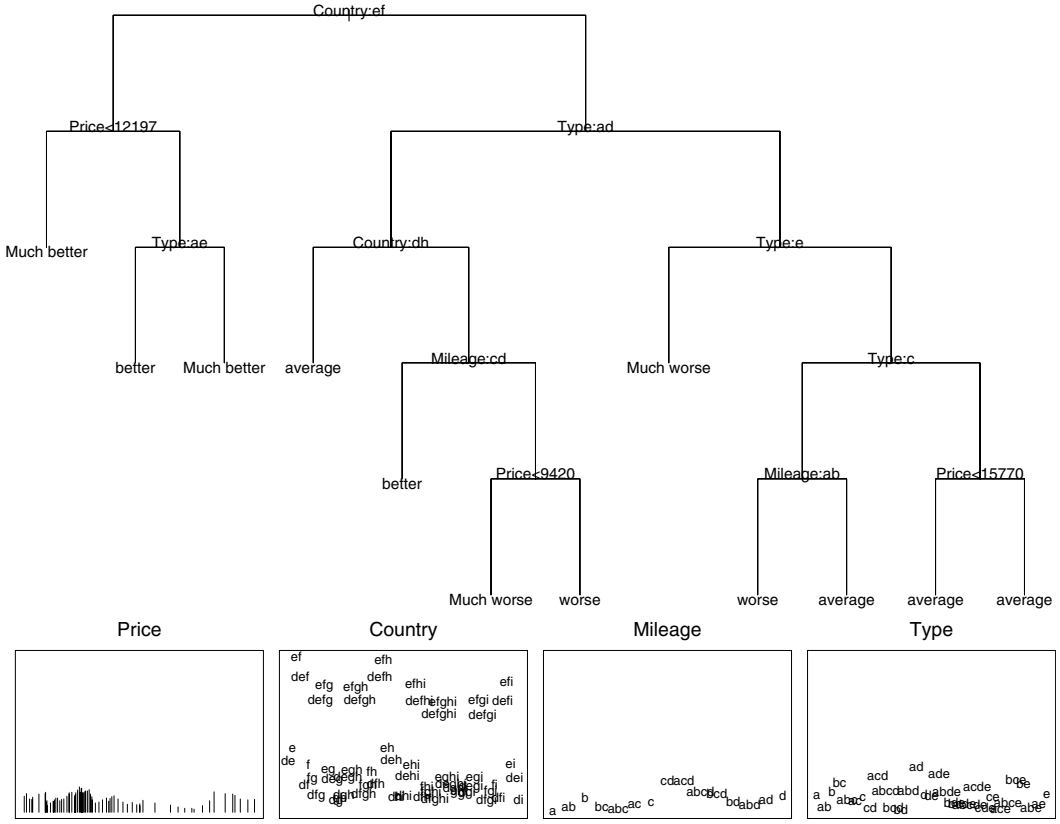

Figure 19.9: *A tree for* `Reliability` *in the* `cu.summary` *data frame with a burl plot of the four predictors for the root node.*

The function `hist.tree` requires a list of variable names in addition to the tree object (and, optionally, a list of nodes). Unlike `burl.tree`, the variables need not be predictors in the tree model. For a given node, a side-by-side histogram is plotted for each variable. The histogram on the left displays the distribution of the observations following the left split. Similarly, the histogram on the right displays the distribution of the observations following the right split.

Figure 19.10 is produced by the expressions below. The figure shows that Japanese cars manufactured in the U.S. or abroad (i.e., the `Country:ef` split) tend to be less expensive and more fuel efficient than other cars. The lower portion of the plot displays side-by-side

histograms for the `Price` and `Mileage` variables. Note that it is possible to obtain a histogram of `Price`, even though the formula for `reliab.tree.2` does not include it as a predictor.

```
> reliab.tree.2 <- tree(Reliability ~
+ Country + Mileage + Type,
+ na.action = na.tree.replace.all, data = cu.summary)
> tree.screens() # Establish plotting regions

[1] 1 2

> plot(reliab.tree.2, type = "u")
> text(reliab.tree.2, cex=0.7)
> hist.tree(reliab.tree.2, Price, Mileage, nodes = 1)
```

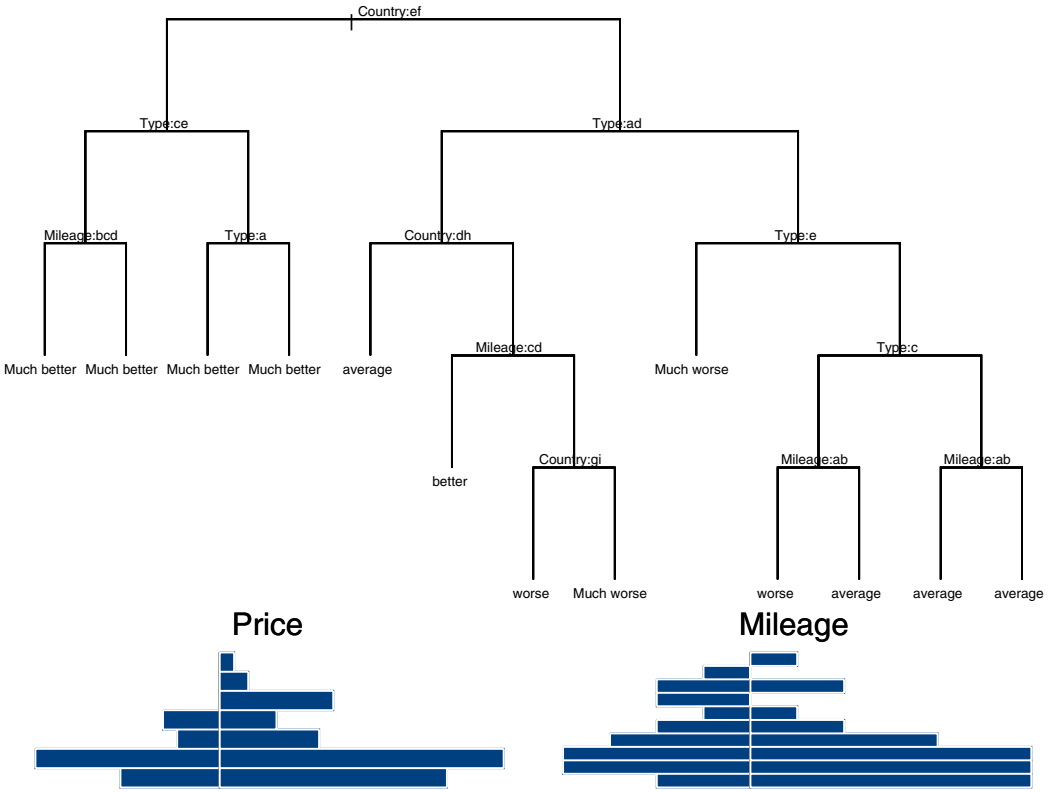

Figure 19.10: *A tree for `Reliability` in the `cu.summary` data frame.*

Manual Splitting and Regrowing

After examining competitor splits at a node, you may wonder what the tree would look like if the node were split differently. You can achieve this by using the `edit.tree` function.

The arguments to `edit.tree` are listed below.

- `object`: Fitted model object of class `"tree"`.
- `node`: Number of the node to edit.
- `var`: Character string naming variable to split on.
- `splitl`: Left split. Numeric for continuous variables; character string of levels that go left for a factor.
- `splitr`: Right split. Character string of levels that go right for a factor.

As an example, look at a burl of `kyph.tree` at the root node for the variable `Start`.

```
> kyph.burl <- burl.tree(kyph.tree, node = 1, plot=F)
> kyph.burl$Start

   Start       dev numl
 1   1.5  1.001008    5
 2   2.5  1.887080    7
 3   4.0  2.173771   10
 4   5.5  5.098140   13
 5   7.0 11.499747   17
 6   8.5 17.946393   19
 7   9.5 12.812267   23
 8  10.5 12.821041   27
 9  11.5 10.136948   30
10  12.5 18.977175   35
11  13.5 13.927629   47
12  14.5 17.508746   52
13  15.5 12.378558   59
14  16.5  2.441679   76
```

Use `edit.tree` to regrow the tree with a designated split at `Start = 8.5`. The result is shown in Figure 19.11.

```
> kyph.tree.edited <- edit.tree(kyph.tree, node = 1,
+ var = "Start", splitl = 8.5)
> plot(kyph.tree.edited)
> text(kyph.tree.edited)
```

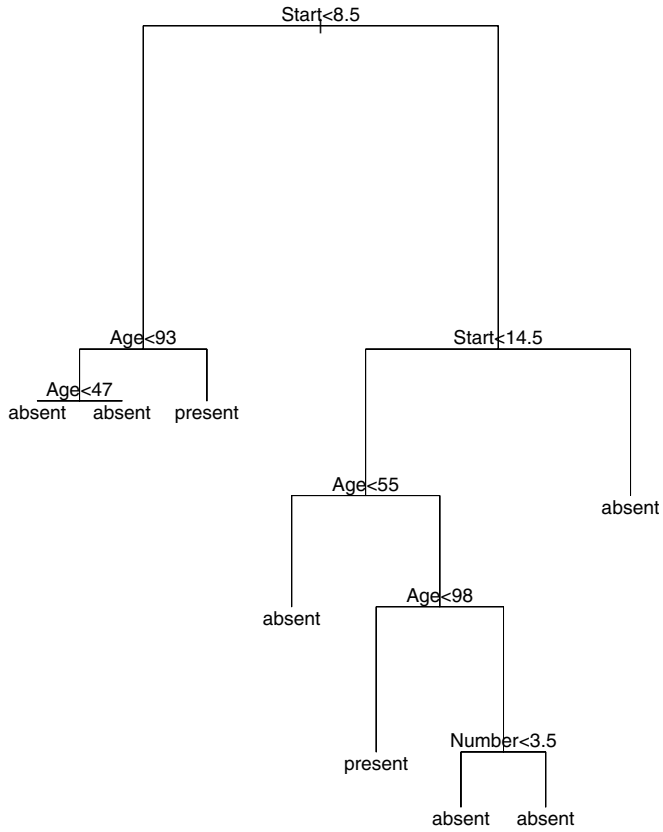

Figure 19.11: *`kyph.tree` regrown at the root node with a split at `Start = 8.5`.*

Leaves

Two noninteractive functions, `tile.tree` and `rug.tree`, show the distribution of a variable over *all* terminal nodes of a tree.

The function `tile.tree` plots histograms of a specified variable for observations in each leaf. This function can be used, for example, to display class probabilities across the leaves of a tree. Figure 19.12 shows the distribution across leaves for the `Kyphosis` variable, as generated by the following commands:

```
> tree.screens() # split plotting screen

[1] 1 2

> plot(kyph.tree)
> text(kyph.tree)
> tile.tree(kyph.tree, Kyphosis)
```

A related function, `rug.tree`, shows the average value of a variable over the leaves of a tree. The optional argument `FUN` allows you to summarize the variable with a measure other than the mean (for example, the trimmed mean or median). Figure 19.13 shows the rug plot of medians for the `Start` variable, as generated by the following commands:

```
> tree.screens() # split plotting screen

[1] 3 4

> plot(kyph.tree)
> text(kyph.tree)
> rug.tree(kyph.tree, Start, FUN = median)
```

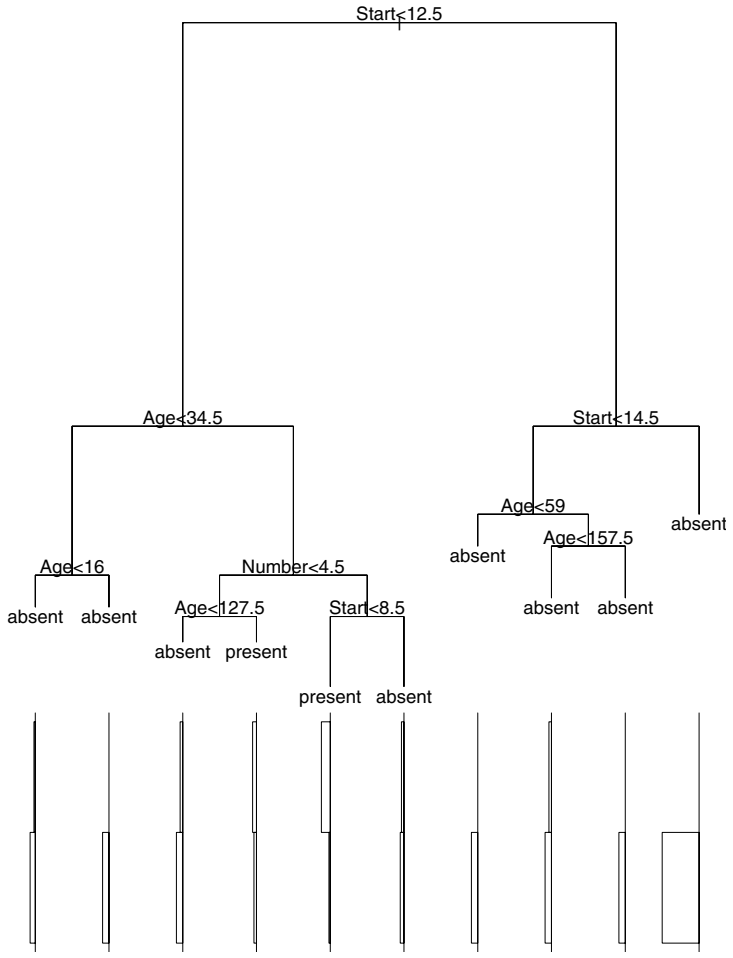

Figure 19.12: *A tree of the* `kyphosis` *data with a tile plot of* `Kyphosis`.

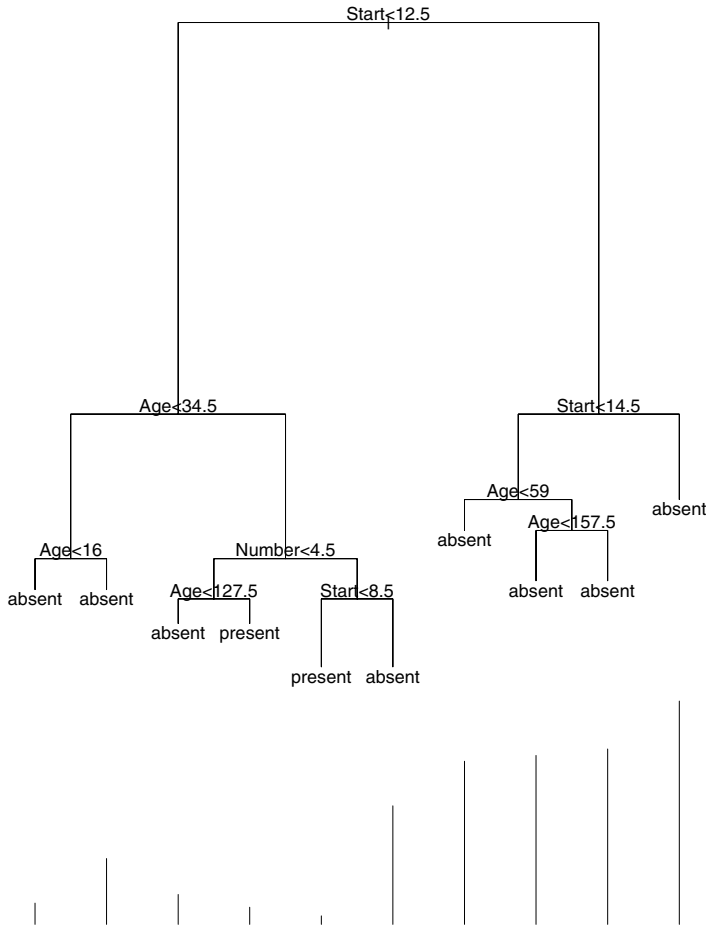

Figure 19.13: *A tree of the* `kyphosis` *data with a rug plot of* `Start`.

REFERENCES

Breiman, L., Friedman, J.H., Olshen, R.A., and Stone, C.J. (1984). *Classification and Regression Trees.* Monterey, CA: Wadsworth and Brooks/Cole.

Chambers, J.M. and Hastie, T.J. (Eds.) (1992). *Statistical Models in S.* London: Chapman and Hall.

PRINCIPAL COMPONENTS ANALYSIS

20

INTRODUCTION

For investigations involving a large number of observed variables, it is often useful to simplify the analysis by considering a smaller number of *linear combinations* of the original variables. For example, scholastic achievement tests typically consist of a number of examinations in different subject areas. In attempting to rate students applying for admission, college administrators frequently attempt to reduce the scores from all subject areas to a single, overall score. If the reduction can be done with minimal information loss, all the better.

One obvious choice for the overall score is the mean over all subject areas. For three subject areas s_1, s_2, and s_3, the mean corresponds to the linear combination $/\ 3 + s_2 /\ 3 + s_3 /$, or equivalently $l's$, where l' is the vector of coefficients $/\ 3,\ 1 /\ 3,\ 1 /$. A linear combination with $\sum l_i^2 = 1$ is called a *standardized linear combination*, or SLC. By restricting attention to SLCs, you can make meaningful comparisons between various choices of linear combinations. For example, with the test scores, you can seek the combination with the greatest variance as a way of ranking the students and separating them.

Principal components analysis finds a set of SLCs, called the principal components, which are orthogonal and taken together explain all the variance of the original data. The principal components are defined as follows (from Mardia, Kent, and Bibby (1979)):

> If x is a random vector with mean μ and covariance matrix Σ, then the *principal component transformation* is the transformation
>
> $$x \rightarrow y = \Gamma'(x - \mu),$$
>
> where Γ is orthogonal, $\Gamma'\Sigma\Gamma = \Lambda$ is diagonal, and $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_p \geq 0 \ldots$. The *ith principal component* of x may be defined as the ith element of the vector y, namely, as
>
> $$y_i = \gamma'_i(x - \mu).$$

Here γ_i is the ith column of Γ and may be called the ith vector of *principal component loadings*.

Note

Some authors define the loadings somewhat differently, as the covariances of the principal components with the original variables. Spotfire S+ follows Mardia, Kent, and Bibby (1979).

The first principal component has the largest variance among all SLCs of x. Similarly, the second principal component has the largest variance among all SLCs of x uncorrelated with the first principal component, and so on.

In general, there are as many principal components as variables. However, because of the way they are calculated, it is usually possible to consider only a few of the principal components, which together explain *most* of the original variation.

CALCULATING PRINCIPAL COMPONENTS

To calculate principal components, use the `princomp` function. In general, the first argument to `princomp` is a numeric matrix or a data frame consisting solely of numeric variables. For example, Table 20.1 shows the results of qualifying examinations for 25 graduate students in mathematics at a fictional university. The students sat for examinations in each of five subject areas–differential geometry, complex analysis, algebra, real analysis, and statistics. The differential geometry and complex analysis examinations were closed book, while the remaining three exams were open book.

Table 20.1: *Examination scores for graduate students in mathematics.*

	diffgeom	**complex**	**algebra**	**reals**	**statistics**
1	36	58	43	36	37
2	62	54	50	46	52
3	31	42	41	40	29
4	76	78	69	66	81
5	46	56	52	56	40
6	12	42	38	38	28
7	39	46	51	54	41
8	30	51	54	52	32
9	22	32	43	28	22
10	9	40	47	30	24
11	32	49	54	37	52
12	40	62	51	40	49

Table 20.1: *Examination scores for graduate students in mathematics. (Continued)*

	diffgeom	complex	algebra	reals	statistics
13	64	75	70	66	63
14	36	38	58	62	62
15	24	46	44	55	49
16	50	50	54	52	51
17	42	42	52	38	50
18	2	35	32	22	16
19	56	53	42	40	32
20	59	72	70	66	62
21	28	50	50	42	63
22	19	46	49	40	30
23	36	56	56	54	52
24	54	57	59	62	58
25	14	35	38	29	20

The data in Table 20.1 is stored in a data set called `testscores`. To perform principal components analysis on these data, use the `princomp` function as follows:

```
> testscores.prc <- princomp(testscores)
```

```
> testscores.prc

Standard deviations:
  Comp. 1  Comp. 2  Comp. 3  Comp. 4  Comp. 5
 28.48968 9.035471 6.600955 6.133582 3.723358

The number of variables is 5
        and the number of observations is 25

Component names:
 "sdev" "loadings" "correlations" "scores" "center"
 "scale" "n.obs" "call" "factor.sdev" "coef"

Call:
princomp(x = testscores)
```

The `princomp` function returns an object of mode `"princomp"`. The printing method for objects of this class shows the standard deviations of the resulting principal components, together with information on the size of the original data set, the names of the components making up the object, and the original call. Use `summary` to produce a summary showing the importance of the calculated principal components:

```
> summary(testscores.prc)

Importance of components:
                          Comp. 1    Comp. 2
    Standard deviation 28.4896795 9.03547104
Proportion of Variance  0.8212222 0.08260135
 Cumulative Proportion  0.8212222 0.90382353
                          Comp. 3    Comp. 4
    Standard deviation 6.60095491 6.13358179
Proportion of Variance 0.04408584 0.03806395
 Cumulative Proportion 0.94790936 0.98597332
                          Comp. 5
    Standard deviation 3.72335754
Proportion of Variance 0.01402668
 Cumulative Proportion 1.00000000
```

In our example, the first principal component explains 82% of the variance, and the first two principal components together explain 90% of the variance.

By default, `princomp` uses a weighted covariance estimation function, `cov.wt`, to perform the principal components analysis. If you want to use a minimum volume ellipsoid covariance estimate, use the `cov.mve` function, which is described in the section Estimating the Model Using a Covariance or Correlation Matrix.

PRINCIPAL COMPONENT LOADINGS

The *principal component loadings* are the coefficients of the principal components transformation. They provide a convenient summary of the influence of the original variables on the principal components, and thus a useful basis for interpretation. A large coefficient (in absolute value) corresponds to a *high* loading, while a coefficient near zero has a *low* loading.

You can view the loadings for a principal components object in either of two ways. First, you can print them as part of the object summary by using the `loadings=T` argument to `summary`:

```
> summary(testscores.prc, loadings = T)

Importance of components:
                          Comp. 1    Comp. 2
    Standard deviation 28.4896795 9.03547104
Proportion of Variance  0.8212222 0.08260135
 Cumulative Proportion  0.8212222 0.90382353
                          Comp. 3    Comp. 4
    Standard deviation 6.60095491 6.13358179
Proportion of Variance 0.04408584 0.03806395
 Cumulative Proportion 0.94790936 0.98597332
                          Comp. 5
    Standard deviation 3.72335754
Proportion of Variance 0.01402668
 Cumulative Proportion 1.00000000

Loadings:
          Comp. 1 Comp. 2 Comp. 3 Comp. 4 Comp. 5
  diffgeom  0.598  -0.675  -0.185  -0.386
   complex  0.361  -0.245   0.249   0.829  -0.247
   algebra  0.302   0.214   0.211   0.135   0.894
     reals  0.389   0.338   0.700  -0.375  -0.321
statistics  0.519   0.570  -0.607          -0.179
```

To see the loadings alone, use the `loadings` function:

```
> loadings(testscores.prc)
```

```
           Comp. 1 Comp. 2 Comp. 3 Comp. 4 Comp. 5
  diffgeom   0.598  -0.675  -0.185  -0.386
   complex   0.361  -0.245   0.249   0.829  -0.247
   algebra   0.302   0.214   0.211   0.135   0.894
     reals   0.389   0.338   0.700  -0.375  -0.321
statistics   0.519   0.570  -0.607          -0.179
```

The `loadings` function returns an object of class `"loadings"`. This class has methods for printing and plotting; a plot of the loadings lets you see at a glance which variables are best explained by each component. For example, consider the loadings plot created by the following call:

```
> plot(loadings(testscores.prc))
```

The plot is shown in Figure 20.1. The loadings for the first principal component are all of the same sign, and of moderate size. A reasonable interpretation is that this component represents an "average" score for the five qualifying examinations. The second component contrasts the two closed book exams with the three open book exams, with the first and last exams weighted most heavily.

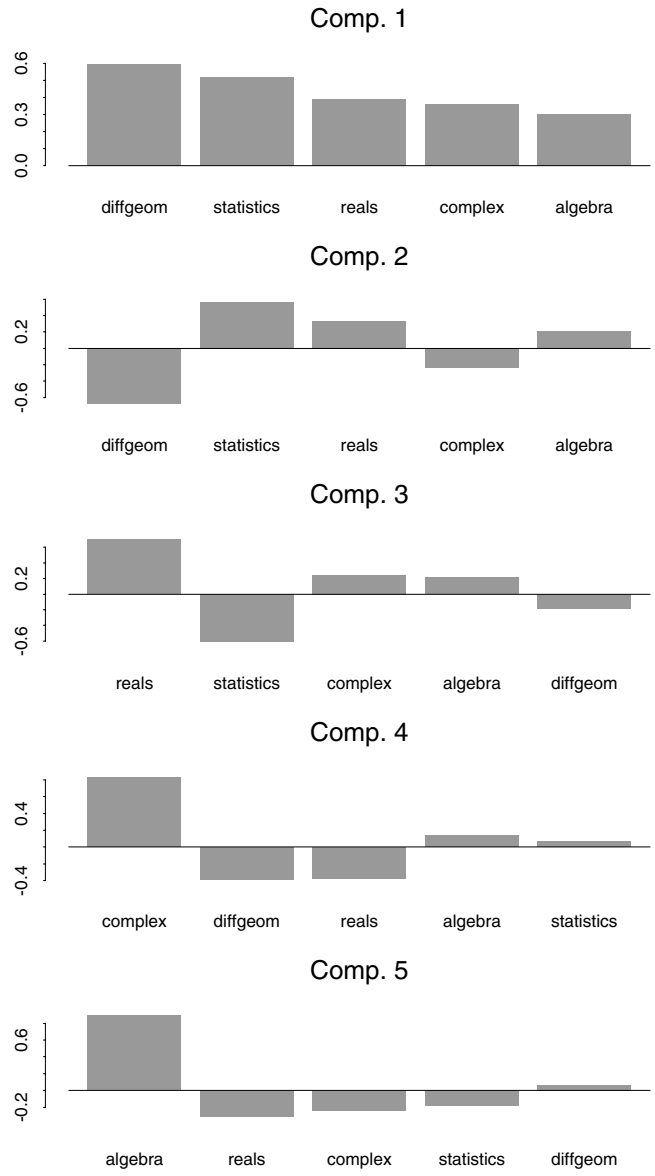

Figure 20.1: *Loadings plot for the test scores data.*

PRINCIPAL COMPONENTS ANALYSIS USING CORRELATION

The principal components decomposition is not scale-invariant. This means that you obtain different decompositions depending on whether you calculate them for the (unscaled) covariance matrix or the (scaled) correlation matrix. In general, you use the covariance matrix when the original observations are on a common scale (as, for example, in the `testscores` data set). You use the correlation matrix when you have observations of different types, such as those in the `state.x77` data set. To calculate principal components for scaled data, use the `cor=T` argument to `princomp`:

```
> state.prc <- princomp(state.x77, cor = T)
> state.prc

Standard deviations:
 Comp. 1  Comp. 2  Comp. 3   Comp. 4   Comp. 5   Comp. 6
1.897076 1.277466 1.054486 0.8411327 0.6201949 0.5544923
  Comp. 7   Comp. 8
0.3800642 0.3364338

The number of variables is 8 and the number of observations
is 50

Component names:
"sdev" "loadings" "correlations" "scores" "center" "scale"
"n.obs" "call"

Call:
princomp(x = state.x77, cor = T)

> summary(state.prc, loadings = T)

Importance of components:
                          Comp. 1   Comp. 2   Comp. 3
    Standard deviation 1.8970755 1.2774659 1.0544862
Proportion of Variance 0.4498619 0.2039899 0.1389926
 Cumulative Proportion 0.4498619 0.6538519 0.7928445
                           Comp. 4    Comp. 5
    Standard deviation 0.84113269 0.62019488
Proportion of Variance 0.08843803 0.04808021
 Cumulative Proportion 0.88128252 0.92936273
```

```
                          Comp. 6   Comp. 7
    Standard deviation 0.55449226 0.3800642
Proportion of Variance 0.03843271 0.0180561
 Cumulative Proportion 0.96779544 0.9858515
                          Comp. 8
    Standard deviation 0.33643379
Proportion of Variance 0.01414846
 Cumulative Proportion 1.00000000

Loadings:
           Comp. 1 Comp. 2 Comp. 3 Comp. 4 Comp. 5
Population -0.126   0.411   0.656   0.409   0.406
    Income  0.299   0.519   0.100          -0.638
Illiteracy -0.468                  -0.353
  Life Exp  0.412           0.360  -0.443   0.327
    Murder -0.444   0.307  -0.108   0.166  -0.128
   HS Grad  0.425   0.299          -0.232
     Frost  0.357  -0.154  -0.387   0.619   0.217
      Area          0.588  -0.510  -0.201   0.499
           Comp. 6 Comp. 7 Comp. 8
Population                  0.219
    Income -0.462
Illiteracy -0.387  -0.620   0.339
  Life Exp -0.219  -0.256  -0.527
    Murder  0.325  -0.295  -0.678
   HS Grad  0.645  -0.393   0.307
     Frost -0.213  -0.472
      Area -0.148   0.286
```

From the loadings for this decomposition, we see that the first principal component contrasts "good" variables such as income and life expectancy with "bad" variables such as murder and illiteracy. It is tempting to interpret this component as a real measure of some nebulous quantity labeled, for example, "Quality of Life." From the importance-of-components summary, however, we see that this component explains only about 45% of the total variance. If we give this "obvious" interpretation to the first principal component, what natural interpretation can we give to the second principal component, which seems to contrast the proportion of frosty days with virtually all of the other variables, and explains another 20% of the variance? This

example shows that, while calculating principal components is straightforward, interpreting the resulting components in physical or social terms is not always so.

ESTIMATING THE MODEL USING A COVARIANCE OR CORRELATION MATRIX

If you do not have raw data, but either a covariance or correlation matrix derived from the original data, you can use the `covlist` argument of the `princomp` function to perform a principal components analysis. The data object that is passed to `princomp` must be a list object with two components, `cov` and `center`.

For example, suppose you have a data object `covmatrix` containing the following covariance matrix:

```
            diffgeom  complex  algebra    reals statistics
   diffgeom 334.8224  174.424 132.0432 169.8096    224.312
    complex 174.4240  139.920  87.6320 104.1360    136.800
    algebra 132.0432   87.632  91.5776 101.8928    129.776
      reals 169.8096  104.136 101.8928 160.2784    160.848
 statistics 224.3120  136.800 129.7760 160.8480    261.760
```

This matrix is created in Spotfire S+ with the following two commands:

```
> covmatrix <- matrix(scan(), ncol=5, byrow=T)

1:  334.8224  174.424 132.0432 169.8096 224.312
6:  174.4240  139.920  87.6320 104.1360 136.800
11: 132.0432   87.632  91.5776 101.8928 129.776
16: 169.8096  104.136 101.8928 160.2784 160.848
21: 224.3120  136.800 129.7760 160.8480 261.760
26:

> dimnames(covmatrix) <- list(c("diffgeom","complex",
+ "algebra","reals","statistics"), c("diffgeom","complex",
+ "algebra","reals","statistics"))
```

Convert `covmatrix` into a list object containing the `cov` and `center` components as follows:

```
> cov.obj <- list(cov = covmatrix, center = c(0,0,0,0,0))
```

```
> cov.obj

$cov:
            diffgeom  complex  algebra    reals statistics
  diffgeom 334.8224  174.424 132.0432 169.8096    224.312
   complex 174.4240  139.920  87.6320 104.1360    136.800
   algebra 132.0432   87.632  91.5776 101.8928    129.776
     reals 169.8096  104.136 101.8928 160.2784    160.848
statistics 224.3120  136.800 129.7760 160.8480    261.760
$center:
[1] 0 0 0 0 0
```

To perform the principal components analysis, pass `cov.obj` to the `princomp` function by using the `covlist` argument:

```
> princov <- princomp(covlist = cov.obj)
> princov

Standard deviations:
  Comp. 1  Comp. 2  Comp. 3  Comp. 4  Comp. 5
 28.48968 9.035471 6.600955 6.133582 3.723358

The number of variables is 5 and the number of
    observations is unknown.

Component names:

 "sdev" "loadings" "correlations" "center" "scale" "call"

Call:
princomp(covlist = cov.obj)
```

If you have a correlation matrix, you can use the `covlist` argument in the same way. For example, suppose you have a data object `cormatrix` containing the following correlation matrix:

```
            diffgeom   complex   algebra     reals statistics
  diffgeom 1.0000000 0.8058590 0.7540744 0.7330229  0.7576935
   complex 0.8058590 0.9999999 0.7741556 0.6953821  0.7148164
   algebra 0.7540744 0.7741556 1.0000000 0.8410298  0.8382009
     reals 0.7330229 0.6953821 0.8410298 1.0000000  0.7852836
statistics 0.7576935 0.7148164 0.8382009 0.7852836  0.9999999
```

As before, the `cormatrix` object is created in Spotfire S+ with the following two commands:

```
> cormatrix <- matrix(scan(), ncol=5, byrow=T)

1: 1.0000000 0.8058590 0.7540744 0.7330229  0.7576935
6: 0.8058590 0.9999999 0.7741556 0.6953821  0.7148164
11: 0.7540744 0.7741556 1.0000000 0.8410298  0.8382009
16: 0.7330229 0.6953821 0.8410298 1.0000000  0.7852836
21: 0.7576935 0.7148164 0.8382009 0.7852836  0.9999999
26:

> dimnames(cormatrix) <- list(c("diffgeom","complex",
+ "algebra","reals","statistics"), c("diffgeom","complex",
+ "algebra","reals","statistics"))
```

Convert `cormatrix` into a list object containing the `cov` and `center` components as follows:

```
> cor.obj <- list(cov = cormatrix, center = c(0,0,0,0,0))
> cor.obj

$cov:
             diffgeom   complex   algebra     reals statistics
   diffgeom 1.0000000 0.8058590 0.7540744 0.7330229  0.7576935
    complex 0.8058590 0.9999999 0.7741556 0.6953821  0.7148164
    algebra 0.7540744 0.7741556 1.0000000 0.8410298  0.8382009
      reals 0.7330229 0.6953821 0.8410298 1.0000000  0.7852836
 statistics 0.7576935 0.7148164 0.8382009 0.7852836  0.9999999

$center:
[1] 0 0 0 0 0
```

To perform the principal components analysis, pass `cor.obj` to the `princomp` function by using the `covlist` argument:

```
> princor <- princomp(covlist = cor.obj)
```

```
> princor

Standard deviations:
  Comp. 1   Comp. 2   Comp. 3   Comp. 4   Comp. 5
 2.020188 0.6114408 0.4653519 0.4525298 0.3516317

The number of variables is 5 and the number of observations
is unknown.

Component names:
 "sdev" "loadings" "correlations" "center" "scale" "call"

Call:
princomp(covlist = cor.obj)
```

By default, `princomp` uses a weighted covariance estimation function, `cov.wt`, to perform the principal components analysis. If you want to use a minimum volume ellipsoid covariance estimate, use the `cov.mve` function by performing the following steps:

1. Use the `cov.mve` function with the raw data (the `rawdataobj` object below), as follows:

   ```
   > mve.object <- cov.mve(rawdataobj)
   ```

   The returned object is a list containing the `cov` and `center` components.

2. Pass the raw data and `mve.object` to `princomp` by using the `covlist` argument as follows:

   ```
   > prin.obj <- princomp(rawdataobj, covlist=mve.object)
   ```

EXCLUDING PRINCIPAL COMPONENTS

The purpose of principal components analysis is to reduce the complexity of multivariate data by transforming the data into the principal components space, and then choosing the first n principal components that explain "most" of the variation in the original variables. Many criteria have been suggested for deciding how many principal components to retain, including the following:

- (Cattell) Plot the eigenvalues λ_j against j. The resulting plot, called a *screeplot* because it resembles a mountainside with a jumble of boulders at its base, often provides a convenient visual method of separating the important components from the less-important components.
- Include just enough components to explain some arbitrary amount (typically, 90%) of the variance.
- (Kaiser) Exclude those principal components with eigenvalues below the average. For principal components calculated from a correlation matrix, this criterion excludes components with eigenvalues less than 1.

Mardia, *et al.* point out that using Cattell's criterion typically results in too many included components, while Kaiser's criterion typically includes too few. The 90% criterion is often a useful compromise.

Creating a Screeplot

A screeplot plots the eigenvalues against their indices, and breaks visually into a steady downward slope (the mountainside) and a gradual tailing away (the scree). The break from the steady downward slope indicates the break between the "important" principal components and the remaining components which make up the scree. The screeplot is the default plot for objects of class `"princomp"`. Thus, to create a screeplot for a principal components object, simply use the `plot` function:

```
> plot(state.prc)

[1] 0.700000 1.900000 3.100000 4.300000 5.500000
[6] 6.700000 7.900000 9.099999
```

By default, the screeplot takes the form of a barplot, and the call to plot returns the x-coordinates of the centers of the bars. The resulting plot is shown in Figure 20.2. Looking for an obvious break between mountainside and scree, you would probably conclude that four or six components should be retained. The 90% criterion retains five components.

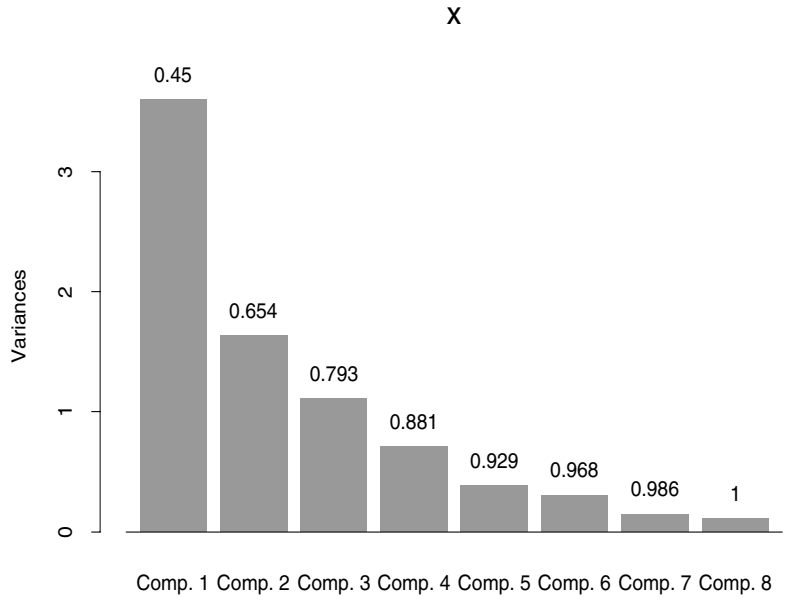

Figure 20.2: *Screeplot for the* `state.x77` *data.*

You can also create a screeplot as a line graph, using the argument `style="lines"`:

```
> plot(testscores.prc, style = "lines")

[1] 1 2 3 4 5
```

The screeplot for the `testscores` data is shown in Figure 20.3. Only the first and second components appear important here, in agreement with the 90% criterion.

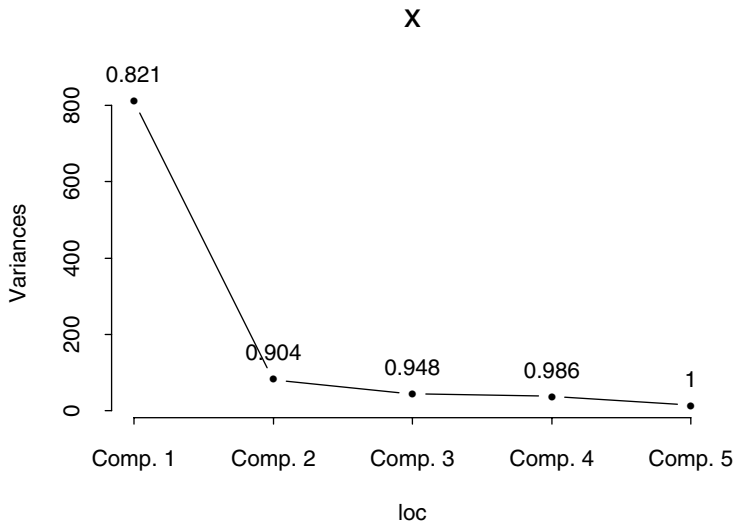

Figure 20.3: *Screeplot for the* `testscores` *data, using* `style="lines"`.

The plot method objects of class `"princomp"` simply calls the `screeplot` function. You can call `screeplot` directly to create the plots in Figure 20.2 and Figure 20.3. Using `screeplot` is particularly useful when writing functions or Spotfire S+ scripts; it clearly indicates what type of plot is being created.

Evaluating Eigenvalues

To apply Kaiser's criterion for excluding eigenvalues:

1. Square the `sdev` component of the principal components object to obtain the vector of eigenvalues.
2. Take the mean of the vector of eigenvalues.
3. Exclude those components with eigenvalues less than the mean.

For the `testscores` data, these steps are:

```
> testscores.eigen <- testscores.prc$sdev^2
> testscores.eigen

 Comp. 1 Comp. 2 Comp. 3 Comp. 4 Comp. 5
 811.662 81.6397 43.5726 37.6208 13.8634

> mean(testscores.eigen)
```

```
[1] 197.672
```

Using Kaiser's criterion, we exclude all components except the first. The 90% criterion suggests keeping the first two.

For principal components objects created from correlation matrices, such as our `state.prc` example, the mean of the eigenvalues is 1. We can therefore look at the eigenvalues to determine which components to exclude:

```
> state.prc$sdev^2

 Comp. 1 Comp. 2 Comp. 3  Comp. 4  Comp. 5  Comp. 6
  3.5989 1.63192 1.11194 0.707504 0.384642 0.307462

  Comp. 7  Comp. 8
0.144449 0.113188
```

Kaiser's criterion suggests including only the first three principal components. The 90% criterion suggests including the first five.

PREDICTION: PRINCIPAL COMPONENT SCORES

One important use of principal components is interpreting the original data in terms of the principal components. For example, the first principal component of the test scores data seems to reflect a weighted average of the test scores. Evaluating this average for each student provides a simple criterion for ranking the students. The images of the original data under the principal components transformation are referred to as *principal component scores*. By default, `princomp` calculates the scores and stores them in the `scores` component of the returned object:

```
> testscores.prc$scores

      Comp. 1    Comp. 2    Comp. 3    Comp. 4
 1  -7.540322 -10.216765  -2.537471   8.670900
 2  20.361037       . . .
```

You can force `princomp` to omit the scores by giving the argument `scores=F`.

Alternatively, if you view the principal components as estimates of interpretable quantities (for example, interpreting the first principal component of the test scores as an estimate of overall ability), it is perhaps more natural to view the principal component scores as predictions from the principal components model. In this case, it is most natural to obtain the scores using the generic `predict` function:

```
> predict(testscores.prc)

      Comp. 1    Comp. 2    Comp. 3    Comp. 4
 1  -7.540322 -10.216765  -2.537471   8.670900
 2  20.361037       . . .
```

You can use `predict` to obtain estimated scores for new data as well. The new data must be in the same form as the original data. For example, suppose you obtained test scores for five additional students and stored them in the matrix `newscores`.

```
> newscores
```

```
  diffgeom complex algebra reals statistics
1       22      50      70    54         30
2       22      46      38    52         62
3       22      42      50    40         62
4       42      49      70    42         50
5       32      35      44    66         32
```

You can obtain the predicted scores for this new data using `predict` as follows:

```
> predict(testscores.prc, newdata = newscores)

     Comp. 1   Comp. 2    Comp. 3     Comp. 4
1  -7.273022  9.070945  20.624141   3.8263656
2  -2.559011 20.754755  -7.975341  -0.7556388
3  -5.044379 20.243279 -14.834342   2.0521791
4  10.041295  3.158848  -3.878835   1.2183456
5  -8.851869  5.635621  16.724818 -20.3311596
      Comp. 5
1  16.4349148
2 -16.2811592
3  -0.7045226
4  18.1853226
5  -6.7149242
```

ANALYZING PRINCIPAL COMPONENTS GRAPHICALLY

We have already seen several graphical views of some portions of the principal components analysis, namely the screeplot and the loadings plot. However, neither of these plots gives a comprehensive view of both the principal components and the original data. The *biplot* (Gabriel (1971)) allows you to represent both the original variables and the transformed observations on the principal components axes. By showing the transformed observations, you can easily interpret the original data in terms of the principal components. By showing the original variables, you can view graphically the relationships between those variables and the principal components.

To create a biplot in Spotfire S+, use the `biplot` function, giving an object of class `"princomp"` as its first argument. For example, to create a biplot for the test scores data, use `biplot` as follows:

```
> biplot(testscores.prc)
```

The resulting plot is shown in Figure 20.4.

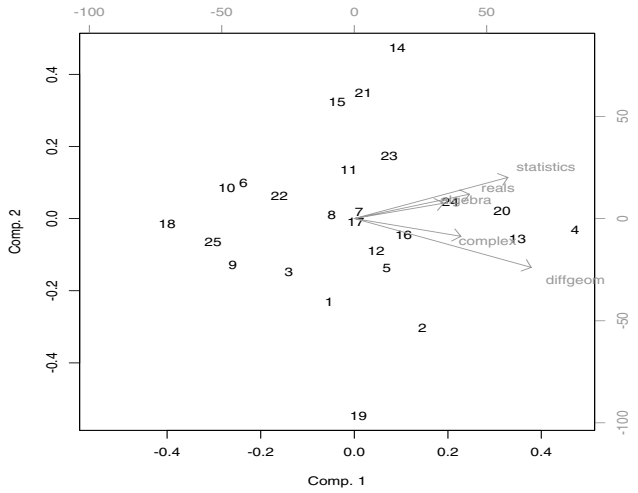

Figure 20.4: *Biplot of test scores data.*

Interpreting the biplot is straightforward: the *x*-axis represents the scores for the first principal component, the *y*-axis the scores for the second principal component. The original variables are represented by arrows which graphically indicate the proportion of the original variance explained by the first two principal components. The direction of the arrows indicates the relative loadings on the first and second principal components. For example, the variable `diffgeom` has the largest loadings in absolute value for both the first and second components, and the loading on the second component has negative sign. Thus `diffgeom` is represented by a longish, downward sloping arrow. The variable `algebra` has the smallest loadings on the first two components, and both loadings have the same sign. Thus, `algebra` is represented by a short, slightly upward-pointing arrow.

While the points plotted on the lower axes in a biplot represent the scores for two principal components, they are not equal to the values returned in the `scores` attribute of a `princomp` object. Likewise, the upper axes do not equal the values returned by the `loadings` function. We devote the remainder of this section to explaining the exact values that are plotted by the Spotfire S+ `biplot.princomp` function. For simplicity, we focus our explanation on the first two principal components only; the derivation is analogous for any two components displayed in a biplot. For additional details, see Gabriel (1971).

An $n \times m$ data matrix X has a singular value decomposition $X = UDV^T$, where D is a diagonal matrix of singular values. For the first two principal components, we calculate the following two matrices:

$$G = \left[U_1 \; U_2\right] D_2^{(1-s)}$$

$$H = \left[V_1 \; V_2\right] D_2^{s},$$

where U_i is the ith column of U, V_i is the ith column of V, D_2 is the upper 2×2 submatrix of D, and $0 \leq s \leq 1$ is a scaling factor. A biplot displays the rows of G on the lower and left axes, and the rows of H on the upper and right axes.

The `princomp` function uses the eigenvalues of covariance or correlation matrices instead of the singular value decomposition, as explained in the section Principal Components Analysis Using Correlation. In this formulation, the loadings matrix L of a principal components analysis is equal to V, the scores matrix S is $XV = UD$, and the standard deviations of the principal components are equal to $D/\sqrt{n-1}$. For the first two principal components, we calculate:

$$G = \left[S_1\ S_2\right] D_2^{-s}$$

$$H = \left[L_1\ L_2\right] D_2^{s},$$

where S_i is the ith column of S and L_i is the ith column of L. The `biplot.princomp` function displays the rows of G on the lower and left axes, and the rows of H on the upper and right axes. The scaling factor s corresponds to the `scale` argument in `biplot.princomp`; by default, $s = 1$.

REFERENCES

Mardia, K.V., Kent, J.T., & Bibby, J.M. (1979). *Multivariate Analysis.* London: Academic Press.

Gabriel, K.R. (1971). The biplot graphical display of matrices with applications to principal component analysis. *Biometrika* **58**:453-467.

FACTOR ANALYSIS 21

INTRODUCTION

In many scientific fields, notably psychology and other social sciences, you are often interested in quantities, such as intelligence or social status, that are not directly measurable. However, it is often possible to measure other quantities which reflect the underlying variable of interest. *Factor analysis* is an attempt to explain the correlations between observable variables in terms of underlying *factors*, which are themselves not directly observable. For example, measurable quantities such as performance on a series of tests can be explained in terms of an underlying factor such as intelligence.

Note

The use of the word "factor" in factor analysis has nothing to do with the usual Spotfire S+ sense of a factor as a categorical data object. In this chapter, we reserve the phrase "Spotfire S+ factor" for this usual sense. The word "factor" alone refers to the traditional meaning in factor analysis: an underlying variable that is not directly observable.

At first glance, factor analysis closely resembles principal components analysis. Both use linear combinations of variables to explain sets of observations of many variables. In principal components analysis, the observed variables are themselves the quantities of interest. The combination of these variables in the principal components is primarily a tool for simplifying the interpretation of the observed variables. In factor analysis, by contrast, the observed variables are of relatively little intrinsic interest; the underlying factors are the quantity of interest.

Formally, if x is a $p \times 1$ random vector with mean μ and covariance matrix Σ, then the k-factor model holds for x if x can be written in the form

$$x = \mu + \Lambda f + u \tag{21.1}$$

where $\Lambda = \{\lambda_{ij}\}$ is a $p \times k$ matrix of constants called the *matrix of factor loadings*. In this equation, f and u are random vectors representing, respectively, the k underlying *common* factors and p

unique factors associated with the original observed variables. Equivalently, the covariance matrix Σ can be decomposed into a *factor covariance matrix* and an *error covariance matrix*:

$$\Sigma = \Lambda \Lambda' + \Psi \tag{21.2}$$

where $\Psi = \text{VAR}(u)$. The diagonal of the factor covariance matrix is called the vector of *communalities* h_i^2, where

$$h_i^2 = \sum_{j=1}^{k} \lambda_{ij}^2 .$$

The communalities represent the common variation in the factors, while the ψ_{ii}, called the *uniquenesses*, represent the variation in the x_i not shared with the other variables.

The *k*-factor model makes sense only if the degrees of freedom $s \geq 0$, where s is given by the equation

$$s = \frac{1}{2}(p-k)^2 - \frac{1}{2}(p+k) .$$

For example, if $p = 5$, $s > 0$ for $k = 1$ and $k = 2$, but $s < 0$ for $k = 3$, $k = 4$, and $k = 5$. Thus, if a factor model is appropriate for a set of five variables, it will have no more than two factors.

ESTIMATING THE MODEL

To perform factor analysis in Spotfire S+, use the `factanal` function. There are two main techniques for estimating the factors in factor analysis: the *principal factor estimate* and the *maximum likelihood estimate*. For a description of these techniques, see Harman (1976) or Mardia, Kent, and Bibby (1979). The principal factor estimate (`method="principal"`) is the default.

For example, consider again the test scores data of Table 20.1. We suppose a two-factor model, one factor representing the overall ability of each student and the second factor representing the relative effects of open vs. closed book exams. We perform the factor analysis as follows, giving `factanal` the raw data `testscores` and specifying the number of factors with the `factors` argument:

```
> testscores.fa <- factanal(testscores, factors =  2)
```

The `factanal` function returns an object of class `"factanal"`. As always, you can look at the object by typing its name. The print method for objects of class `"factanal"` shows the sum of squares of the factor loadings, the size of the data, the names of the components in the returned object, and the call that created the object:

```
> testscores.fa

Sums of squares of loadings:
  Factor1  Factor2
 2.219645 1.866672
The number of variables is 5 and the number of observations
is 25

Component names:

 "loadings" "uniquenesses" "correlation" "criteria"
 "factors" "dof" "method" "center" "scale" "n.obs"
 "scores" "call"

Call:
factanal(x = testscores, factors = 2)
```

By default, `factanal` uses a weighted covariance estimation function, `cov.wt`, to perform the factor analysis. If you want to use a minimum volume ellipsoid covariance estimate, use the `cov.mve` function, which is described in the section Estimating the Model Using a Covariance or Correlation Matrix.

To see a numeric summary of the factor solution, use the `summary` function:

```
> summary(testscores.fa)

Importance of factors:
                Factor1   Factor2
   SS loadings 2.219645 1.8666722
Proportion Var 0.443929 0.3733344
Cumulative Var 0.443929 0.8172634

The degrees of freedom for the model is 1.

Uniquenesses:
  diffgeom   complex   algebra     reals statistics
 0.1970121 0.1879035 0.1201226 0.1984058  0.2102388

Loadings:
           Factor1 Factor2
  diffgeom 0.506   0.739
   complex 0.457   0.777
   algebra 0.787   0.510
     reals 0.775   0.448
statistics 0.730   0.507
```

The table at the top of the summary, labeled `Importance of Factors`, shows the sum of squares of the loadings on each factor, along with the proportion of the total variance explained by each factor, and the cumulative proportion explained after each factor is included. Thus, the two-factor model for the test scores data explains about 80% of the variation in the original data, with the first factor accounting for about 45%.

The summary also shows the number of degrees of freedom in the model, the uniquenesses, and the factor loadings. The factor loadings can also be seen by themselves, using the `loadings` function:

```
> loadings(testscores.fa)

           Factor1 Factor2
  diffgeom 0.506   0.739
   complex 0.457   0.777
   algebra 0.787   0.510
     reals 0.775   0.448
statistics 0.730   0.507
```

Since the uniquenesses and communalities sum to 1 for each variable, you can calculate the communalities h_i^2 from the uniquenesses as follows:

```
> 1 - testscores.fa$uniquenesses

  diffgeom   complex   algebra     reals statistics
 0.8029879 0.8120965 0.8798774 0.8015942  0.7897612
```

ESTIMATING THE MODEL USING MAXIMUM LIKELIHOOD

To use the maximum likelihood factor estimate, specify `method="mle"` in the call to `factanal`:

```
> testscores.fa2 <- factanal(testscores, factors = 2,
+ method = "mle")
> testscores.fa2

Sums of squares of loadings:
 Factor1  Factor2
 2.48222 1.726735

The number of variables is 5 and the number of observations
is 25

Test of the hypothesis that 2 factors are sufficient versus
the alternative that more are required:
The chi square statistic is 0.78 on 1 degree of freedom.
The p-value is 0.378

Component names:

"loadings" "uniquenesses" "correlation" "criteria"
"factors" "dof" "method" "center" "scale" "n.obs" "scores"
"call"

Call:
factanal(x = testscores, factors = 2, method = "mle")
```

With the maximum likelihood method, it is possible to perform a test of the hypothesis that the specified number of factors is adequate to explain the model, and the print method for objects of class `"factanal"` gives the results of this test. In this case, there is no evidence that more factors should be added.

ESTIMATING THE MODEL USING A COVARIANCE OR CORRELATION MATRIX

If you do not have raw data, but either a covariance or correlation matrix derived from the original data, you can use the `covlist` argument of the `factanal` function to estimate the factors. The data object that is passed to `factanal` must be a list object with two components, `cov` and `center`.

For example, suppose you have a data object `covmatrix` containing the following covariance matrix:

```
           diffgeom  complex  algebra    reals statistics
diffgeom   334.8224  174.424 132.0432 169.8096    224.312
complex    174.4240  139.920  87.6320 104.1360    136.800
algebra    132.0432   87.632  91.5776 101.8928    129.776
reals      169.8096  104.136 101.8928 160.2784    160.848
statistics 224.3120  136.800 129.7760 160.8480    261.760
```

Convert `covmatrix` into a list object containing the `cov` and `center` components as follows:

```
> cov.obj <- list(cov = covmatrix, center = c(0,0,0,0,0))
> cov.obj

$cov:
           diffgeom  complex  algebra    reals statistics
diffgeom   334.8224  174.424 132.0432 169.8096    224.312
complex    174.4240  139.920  87.6320 104.1360    136.800
algebra    132.0432   87.632  91.5776 101.8928    129.776
reals      169.8096  104.136 101.8928 160.2784    160.848
statistics 224.3120  136.800 129.7760 160.8480    261.760

$center:
[1] 0 0 0 0 0
```

To perform the factor analysis, pass the `cov.obj` object to the `factanal` function by using the `covlist` argument, as follows:

```
> factcov <- factanal(covlist = cov.obj)
```

```
> factcov

Sums of squares of loadings:
  Factor1
 3.854577

The number of variables is 5 and the number of observations
is unknown.

Component names:

"loadings" "uniquenesses" "correlation" "criteria"
"factors" "dof" "method" "center" "scale" "call"

Call:
factanal(covlist = cov.obj)
```

If you have a correlation matrix, you can use the `covlist` argument in the same way. For example, suppose you have a data object `cormatrix` containing the following correlation matrix:

```
            diffgeom   complex   algebra     reals statistics
diffgeom  1.0000000 0.8058590 0.7540744 0.7330229  0.7576935
complex   0.8058590 0.9999999 0.7741556 0.6953821  0.7148164
algebra   0.7540744 0.7741556 1.0000000 0.8410298  0.8382009
real      0.7330229 0.6953821 0.8410298 1.0000000  0.7852836
statistics 0.7576935 0.7148164 0.8382009 0.7852836  0.9999999
```

Convert `cormatrix` into a list object containing the `cov` and `center` components as follows:

```
> cor.obj <- list(cov = cormatrix, center = c(0,0,0,0,0))
> cor.obj

$cov:
            diffgeom   complex   algebra     reals statistics
diffgeom  1.0000000 0.8058590 0.7540744 0.7330229  0.7576935
complex   0.8058590 0.9999999 0.7741556 0.6953821  0.7148164
algebra   0.7540744 0.7741556 1.0000000 0.8410298  0.8382009
reals     0.7330229 0.6953821 0.8410298 1.0000000  0.7852836
statistics 0.7576935 0.7148164 0.8382009 0.7852836  0.9999999

$center:
[1] 0 0 0 0 0
```

To perform the factor analysis, pass the `cor.obj` object to the `factanal` function by using the `covlist` argument, as follows:

```
> factcor <- factanal(covlist = cor.obj)
> factcor

Sums of squares of loadings:
  Factor1
 3.854577

The number of variables is 5 and the number of observations
is unknown.

Component names:

"loadings" "uniquenesses" "correlation" "criteria"
"factors" "dof" "method" "center" "scale" "call"

Call:
factanal(covlist = cor.obj)
```

By default, `factanal` uses a weighted covariance estimation function, `cov.wt`, to estimate the factors. If you want to use a minimum volume ellipsoid covariance estimate, use the `cov.mve` function by performing the following steps:

1. Use the `cov.mve` function with the raw data, in this example, the `rawdataobj` object, as follows:

   ```
   > mve.object <- cov.mve(rawdataobj)
   ```

   The returned object is a list containing the `cov` and `center` components.

2. Pass the raw data and `mve.object` to `factanal` by using the `covlist` argument as follows:

   ```
   > fact.obj <- factanal(rawdataobj, covlist=mve.object)
   ```

ROTATING FACTORS

The solution to Equation (21.2) is not unique unless the number of factors k is 1. If G is a $k \times k$ orthogonal matrix, then

$$\Sigma = (\Lambda G')(G'\Lambda') + \Psi \tag{21.3}$$

which has the form of Equation (21.2) with $\Delta = \Lambda G$ being the matrix of *rotated factor loadings*. Thus, the factor loadings are inherently indeterminate. Any solution can be rotated arbitrarily to arrive at a new solution. In practice, this indeterminancy is used to arrive at a factor solution that has what Thurstone (1935) named *simple structure*. Loosely, the factor solution has simple structure if each variable is loaded highly on one factor, and all factor loadings are either large (in absolute value) or near zero.

Factor analysts have developed many different criteria for choosing the appropriate rotation. By default, Spotfire S+ uses the "varimax" method. You can specify a different rotation with the `rotation` argument to `factanal`. For example, to compute the factor solution to the test scores data using the `"oblimin"` rotation, call `factanal` as follows:

```
> testscores.fao <- factanal(testscores, factors = 2,
+ rotation = "oblimin")
> summary(testscores.fao)

Importance of factors:
                   Factor1    Factor2
     SS loadings 3.8946361 0.18800271
  Variable Index 0.7789272 0.03760054
Cumulative Index 0.7789272 0.81652776

The degrees of freedom for the model is 1.

Uniquenesses:
  diffgeom   complex   algebra     reals statistics
 0.1970121 0.1879035 0.1201226 0.1984058  0.2102388
```

```
Loadings:
           Factor1 Factor2
  diffgeom  0.855   0.216
   complex  0.839   0.277
   algebra  0.937  -0.143
     reals  0.889  -0.181
statistics  0.889  -0.107

Component/Factor Correlations:
        Factor1 Factor2
Factor1   1.000  -0.067
Factor2  -0.067   1.000
```

You can rotate any object of class `"factanal"` using the `rotate` function:

```
> rotate(testscores.fa, rotation="biquartimin")

Sums of squares of loadings:
  Factor1   Factor2
 3.884609 0.1903185

The number of variables is 5 and the number of observations
is 25

Component names:

 "loadings" "uniquenesses" "correlation" "criteria"
 "factors" "dof" "method" "center" "scale" "n.obs" "call"

Call:
rotate.factanal(x = factanal(x = testscores, factors = 2),
  rotation = "biquartimin")

> loadings(.Last.value)

           Factor1 Factor2
  diffgeom  0.844   0.225
   complex  0.825   0.286
   algebra  0.943  -0.135
     reals  0.897  -0.173
statistics  0.894
```

```
Component/Factor Correlations:
        Factor1 Factor2
Factor1 1.000   0.106
Factor2 0.106   1.000
```

Spotfire S+ recognizes the following character strings as valid `rotation` arguments:

```
"varimax"       "quartimax"  "equamax"
"parsimax"      "orthomax"   "covarimin"
"biquartimin"   "quartimin"  "oblimin"
"procrustes"    "promax"     "none"
"crawford.ferguson"
```

See Harman (1976) for descriptions of the various rotations. See the `rotate` help file for additional information on using the various rotations in Spotfire S+.

VISUALIZING THE FACTOR SOLUTION

The loadings matrix provides a precise, numeric answer to the question of which variables are loaded most strongly on each factor. However, you can get a much more intuitive feel for the answer if you look at the loadings visually. You obtain a loadings plot by calling `plot` on the factor loadings:

```
> plot(loadings(testscores.fa))
```

The resulting plot is shown in Figure 21.1.

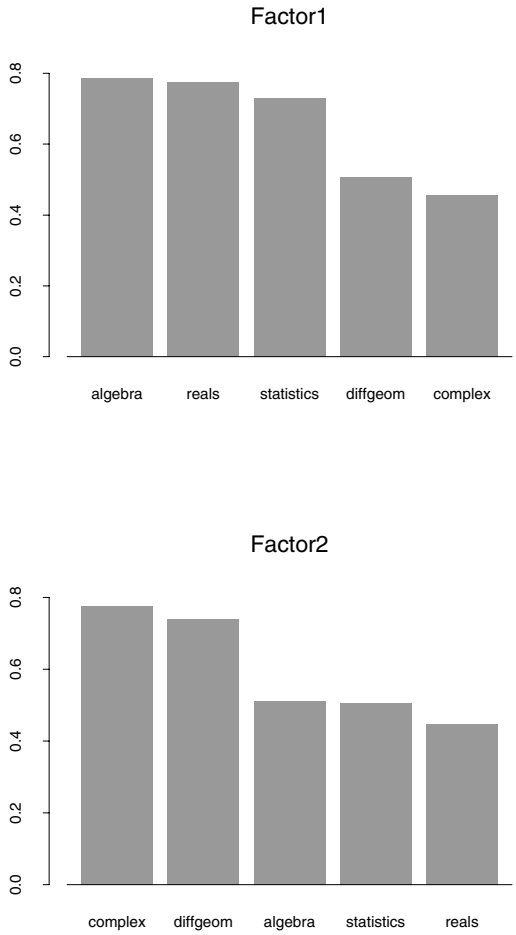

Figure 21.1: *Loadings for the test scores principal factor solution.*

To see the relation of the factors to both the original variables and the original data, use `biplot`:

```
> biplot(testscores.fa)
```

The resulting plot is shown in Figure 21.2.

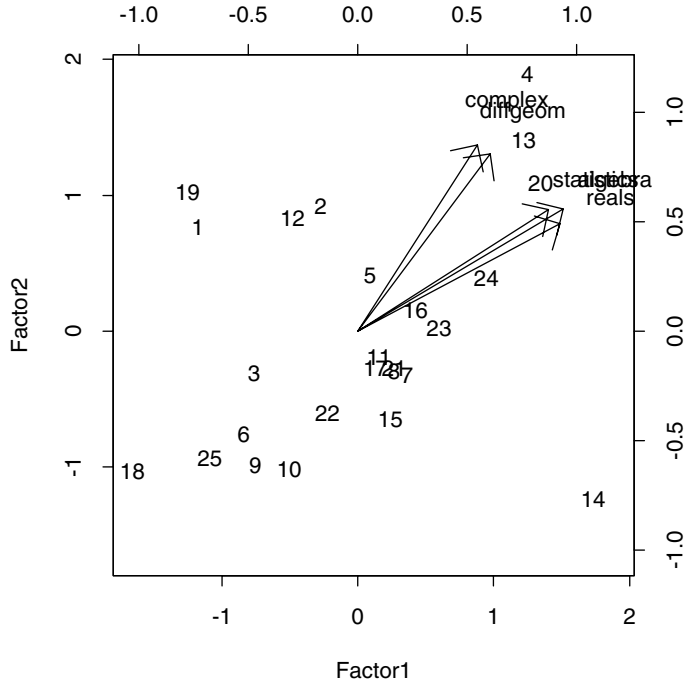

Figure 21.2: *Biplot for the test scores principal factor solution.*

PREDICTION: FACTOR ANALYSIS SCORES

An important use of factor analysis is to translate the original data into the planes of the factors. You view the factors as estimates of interpretable quantities (for example, interpreting the first factor of the test scores as an estimate of overall ability). The images of the original data under the factor analysis transformation are referred to as *factor analysis scores*. By default, `factanal` calculates the scores and stores them in the `scores` component of the returned object:

```
> testscores.fa$scores

       Factor1   Factor2
 1  -1.1778029 0.7612478
 2  -0.2755734     . . .
```

You can force `factanal` to omit the scores by giving the argument `scores=F`.

It is perhaps more natural to view the factor scores as predictions from the factor analysis model. In this case, you can use the generic `predict` function to obtain the scores:

```
> predict(testscores.fa)

       Factor1    Factor2
 1  -1.1778029  0.7612478
 2  -0.2755734      . . .
```

You can use `predict` to obtain estimated scores for new data, as well. The new data must be in the same form as the original data. For example, suppose you obtained test scores for five additional students and stored them in the matrix `newscores`:

```
> newscores

  diffgeom complex algebra reals statistics
1       22      50      70    54         30
2       22      46      38    52         62
3       22      42      50    40         62
4       42      49      70    42         50
5       32      35      44    66         32
```

You can obtain the predicted scores for this new data using `predict` as follows:

```
> predict(testscores.fa, newdata = newscores)

          Factor1    Factor2
[1,]  1.454873272 -0.9626068
[2,] -0.001166622 -0.5764937
[3,]  0.493414880 -0.8808624
[4,]  1.216808651 -0.3201456
[5,]  0.570954434 -1.1814138
attr(, "type"):
[1] "regression"
```

REFERENCES

Harman, H.H. (1976). *Modern Factor Analysis*. Chicago: University of Chicago Press.

Mardia, K.V., Kent, J.T., and Bibby, J.M. (1979). *Multivariate Analysis*. London: Academic Press.

Thurstone, L.L. (1935). *The Vectors of Mind*. Chicago: University of Chicago Press.

DISCRIMINANT ANALYSIS 22

INTRODUCTION

Suppose you have a set of quantitative observations about individuals belonging to two or more groups, such as the three species in Fisher's iris data, or patients infected with or free of some disease. Membership in a given group can be represented by a categorical variable. You can use the quantitative observations to create a model that explains the grouping of the given individuals, and can further be used to assign additional observations to the correct group. Such models can be fit in a variety of ways, all of which are encompassed by the general term *discriminant analysis*.

In the simplest case, assume that all the groups have equal covariance matrices. In this case, called the *homoscedastic* model, you can derive a linear discriminant function of the form:

$$l(\mathbf{x}) = \beta_{i0} + \beta_{i1}\mathbf{x}$$

In the most general case, the various groups have independent covariance matrices, leading to the *heteroscedastic model*, which leads to a quadratic discriminant function of the form:

$$d(\mathbf{x}) = \beta_{i0} + \beta_{i1}\mathbf{x} + \mathbf{x}^T\beta_{i2}\mathbf{x}$$

Relationships among feature variables with respect to the grouping variable can be expressed by their mean values and their variance-covariance matrices. You can quantify these relationships and take advantage of group variance-covariance similarities to reduce the number of parameters estimated.

A SIMPLE EXAMPLE

As a simple example of using the `discrim` function, consider Fisher's iris data in the Spotfire S+ data set `iris`. This data set is an array containing 50 observations of each of three species of iris. We first need to convert it to a data frame:

```
> Species <- factor(c(rep("Setosa", 50),
+     rep("Versicolor", 50), rep("Virginica", 50)))
> exiris <- rbind(iris[,,1], iris[,,2], iris[,,3])
> exiris <- data.frame(Species, exiris)
```

Next we fit a default (homoscedastic) model:

```
> exiris.discrim <- discrim(Species ~ ., data = exiris)
> exiris.discrim

Call:
discrim(Species ~ Sepal.L. + Sepal.W. + Petal.L. +
Petal.W., data = exiris)

Group means:
           Sepal.L. Sepal.W. Petal.L. Petal.W.  N
    Setosa    5.006    3.428    1.462    0.246 50
Versicolor    5.936    2.770    4.260    1.326 50
 Virginica    6.588    2.974    5.552    2.026 50
              Priors
    Setosa 0.3333333
Versicolor 0.3333333
 Virginica 0.3333333

Covariance Structure: homoscedastic
          Sepal.L.  Sepal.W.  Petal.L.   Petal.W.
Sepal.L. 0.2650082 0.0927211 0.1675143 0.03840136
Sepal.W.           0.1153878 0.0552435 0.03271020
Petal.L.                     0.1851878 0.04266531
Petal.W.                               0.04188163

Constants:
    Setosa Versicolor Virginica
 -86.30847  -72.85261 -104.3683
```

```
Linear Coefficients:
             Setosa Versicolor Virginica
Sepal.L.  23.54417   15.69821  12.44585
Sepal.W.  23.58787    7.07251   3.68528
Petal.L. -16.43064    5.21145  12.76654
Petal.W. -17.39841    6.43423  21.07911
```

The "." on the right-hand side of the formula tells Spotfire S+ to fit a model using all the remaining variables in `exiris` as predictor variables. We next obtain predictions for our training data:

```
> exiris.predict <- predict(exiris.discrim)
```

How well did our model do? There were 150 observations in the original data; as the following expression shows, only 3 are misclassified by our simple model:

```
> sum(exiris.predict$groups != Species)

[1] 3
```

MODELS

The various models for discriminating between the groups specify some relationships among the groups' covariance matrices; the two extremes that are typically considered are the heteroscedastic model, in which there is no posited relationship among the covariance matrices, and the homoscedastic model, in which the covariance matrices are assumed to be all alike. For the `discrim` function a model is specified by the `family` argument. Currently, there are three family constructors for the `discrim` function: `Classical`, `CPC`, and `Canonical`. Each family defines a possible hierarchy of models that makes use of the posited similarity among the group covariances.

The `Classical` family includes the following covariance structures, from most general to specific:

- Heteroscedastic
- Equal correlation
- Proportional
- Group spherical
- Homoscedastic
- Spherical

As you move from the heteroscedastic to the spherical model, there is in general a reduction in the number of parameters which have to be estimated. There is some overlap in the number of parameters estimated for the proportional and group spherical models, however, depending on the number of groups and number of feature variables. Models with fewer estimated parameters tend to be more stable in terms of standard errors than models with more parameters. You fit the classical hierarchy of models in Spotfire S+ using the `discrim` function with the argument `family=Classical(cov.structure=`*`structure`*`)`. For example, an equal correlation model is fit by specifying `cov.structure="equal correlation"`.

The family `CPC` is the common principal component family (Flury, 1984). The two covariance structures currently available for this family are the proportional and common principal component. These do not exhaust the possibilities discussed by Flury (1988), but together

with the homoscedastic and heteroscedastic models of the classical family, they complete another logical hierarchy of models. The argument `family=CPC(cov.structure = structure)` to `discrim` provides the two principal component models.

The `Canonical` family consists of just one model, using the homoscedastic covariance structure.

We assume that the feature vectors are p-variate normal random variables $N(\mu_i, \Sigma_i)$, for $i = 1, \ldots, g$. The normality assumption is not required for the canonical discriminant function, however.

Heteroscedastic

The heteroscedastic model is the most general model and requires estimating the maximum number of parameters: $g \cdot p(p+1)/2$ variance-covariance estimates. Here, we have $\Sigma_i \neq \Sigma_j$ for $i \neq j$.

To fit a heteroscedastic model, use `discrim` with the argument `family=Classical(cov="heteroscedastic")`:

```
> exiris.het <- discrim(Species ~ ., data = exiris,
+ family = Classical(cov = "heteroscedastic"))
> exiris.het

Call:
discrim(Species ~ Sepal.L. + Sepal.W. + Petal.L. +
Petal.W., data = exiris, family = Classical(cov =
"heteroscedastic"))

Group means:
           Sepal.L. Sepal.W. Petal.L. Petal.W.  N
    Setosa     5.006    3.428    1.462    0.246 50
Versicolor     5.936    2.770    4.260    1.326 50
 Virginica     6.588    2.974    5.552    2.026 50
              Priors
    Setosa 0.3333333
Versicolor 0.3333333
 Virginica 0.3333333

Covariance Structure: heteroscedastic
```

```
Group: Setosa
             Sepal.L.  Sepal.W.   Petal.L.   Petal.W.
Sepal.L. 0.1242490 0.0992163 0.01635510 0.01033061
Sepal.W.           0.1436898 0.01169796 0.00929796
Petal.L.                     0.03015918 0.00606939
Petal.W.                                0.01110612
...
```

Equal Correlation Matrix

The equal correlation matrix model assumes that the groups have a common correlation structure, but different variances. The covariance matrix of each group is then $\Sigma_i = \mathbf{K}_i \Psi \mathbf{K}_i$, where $\mathbf{K}_i = \text{diag}(\sigma_{i1}, \ldots, \sigma_{ip})$ and Ψ is the common correlation matrix. Here, we estimate $\cdot\ p + p \cdot (p-1)/\ 2$ correlation and variance parameters, a reduction of $\jmath - 1) \cdot p \cdot (p-1)/$ from the heteroscedastic model.

To fit an equal-correlation model, use `discrim` with the argument `family=Classical(cov="equal correlation")`:

```
> exiris.eqcor <- discrim(Species ~ ., data = exiris,
+     family = Classical(cov = "equal"))
```

Common Principal Component

The group covariance matrices for the common principal component model can be written as $\Sigma_i = \mathbf{A}\Lambda_i\mathbf{A}$, where $\Lambda_i = \text{diag}(\lambda_{i1}, \ldots, \lambda_{ip})$ and $\mathbf{A}$ is the matrix of common principal components. The number of parameters estimated here is $\imath \cdot\ p + p \cdot\ \mu$.

To fit a common principal component model, use `discrim` with the argument `family=CPC()` (the common principal component is the default for the CPC family):

```
> exiris.cpc <- discrim(Species ~ ., data = exiris,
+     family = CPC())
```

Proportional Covariances

The proportional covariances model further reduces the number of parameters to estimate to $g-1) + p \cdot\ (p+1)/\ 2$ by assuming each group's covariance is proportional to a common covariance: $\Sigma_i = \kappa_i^2 \Sigma$. Note that one proportionality constant, κ_i, is redundant so

we set $\kappa_1 \equiv 1$. In the common principal component family, the proportional model assumes $\lambda_{ik} = \kappa_i^2 \lambda_{1k}$, for $i = 2, \ldots, g$ and $k = 1, \ldots, p$.

To fit a classical proportional covariances model, use `discrim` with the argument `family=Classical(cov="proportional")`:

```
> exiris.propcov <- discrim(Species ~ ., data = exiris,
+     family = Classical(cov = "proportional"))
```

Spherical

Here we assume that the feature vectors are independent. Two spherical models can be fit. The more general is the *group spherical* model, in which the variances for the feature vectors for each group are different $\Sigma_i = \text{diag}(\sigma_{i1}^2, \ldots, \sigma_{ip}^2)$. To fit a group spherical model, use the argument `family=Classical(cov="group")` in the call to `discrim`:

```
> exiris.gs <- discrim(Species ~ ., data = exiris,
+     family = Classical(cov = "group"))
```

The spherical model, on the other hand, assumes the feature vector variances are the same for each group $\Sigma_i = \Sigma = \text{diag}(\sigma_1^2, \ldots, \sigma_p^2)$, for all $i = 1, \ldots, g$. Thus, the spherical model is the most restrictive model, but also the simplest to compute, with only p variances to be estimated.

To fit a spherical model, use `discrim` with the argument `family=Classical(cov="spherical")`:

```
> exiris.sph <- discrim(Species ~ ., data = exiris,
+     family = Classical("spherical"))
```

Homoscedastic

The homoscedastic model assumes that the group covariance matrices are equal $\Sigma_i = \Sigma$, for all $i = 1, \ldots, g$. Here, $\cdot\cdot (p+1)/\ 2$ variance-covariances are estimated. You can fit a homoscedastic model using either the `Classical` or `Canonical` families; it is the only covariance structure permissible for the `Canonical` family.

To fit a classical homoscedastic model, use `discrim` with the argument `family=Classical(cov="homoscedastic")`:

```
> exiris.homcl <- discrim(Species ~ ., data = exiris,
+     family = Classical(cov = "homoscedastic"))
```

To fit a canonical homoscedastic model, use `discrim` with the argument `family=Canonical()`:

```
> exiris.homcan <- discrim(Species ~ ., data = exiris,
+     family = Canonical())
```

HYPOTHESIS TESTING

A likelihood ratio test can be performed between two or more fitted models to test for a plausible covariance structure for the groups. One hierarchy of models that can be constructed is the heteroscedastic, call it hypothesis H_5, equal correlation, H_4, proportional, H_3, group spherical, H_2, homoscedastic, H_1, and spherical, H_0, covariance structures. A sequence of tests proceeds from the most general, H_4 versus H_5, to the most restrictive, H_0 versus H_1, until a significant likelihood ratio statistic is observed (McLachlan, 1992, pp. 175-178). You perform these tests using the `anova` method for the `discrim` class.

For example, to compare our heteroscedastic model `discrim.het` to our equal-correlation model `discrim.eqcor`, we call `anova` as follows:

```
> anova(exiris.het, exiris.eqcor)

Group Variable: Species
                 Cov.Structure Df     AIC     BIC
  exiris.het   heteroscedastic 46 -838.57 -792.08
exiris.eqcor equal correlation 34 -823.08 -788.72
             Loglik    Test Lik.Ratio     P.value
  exiris.het 511.28
exiris.eqcor 479.54 1 vs. 2    63.489 5.1801e-009
```

The models differ significantly, so in this case we believe the full heteroscedastic model is required.

Within the CPC family, there is just a two level hierarchy. This permits a two level hierarchy for the principal component models: $H_{proportional}$ versus H_{CPC}. For a full hierarchy, you should include the classical heteroscedastic and homoscedastic models.

ESTIMATION

How parameters are estimated depends on the model fitted. The classical homoscedastic and heteroscedastic covariance structures of the `Classical` family require only simple manipulation of the estimated means and covariances. The equal correlation and proportional covariance structures of the `Classical` family require numerical optimization with respect to the Wishart distribution. The canonical discriminant function requires eigenvalue estimation, while the common principal component family requires both optimization and eigenvalue estimation.

Classical Homoscedastic and Heteroscedastic

The classical homoscedastic and heteroscedastic discriminant functions are derived from the log of the normal distribution

$$\begin{aligned} l(\mathbf{x}|\mu_i,\ \Sigma_i) \propto & -\frac{p}{2}\cdot\ \log|\Sigma_i| - \frac{1}{2}(\mathbf{x}-\mu_i)^T\Sigma_i^{-1}(\mathbf{x}-\mu_i)^T \\ = & -\frac{p}{2}\cdot\ \log|\Sigma_i| - \frac{1}{2}\mathbf{x}^T\Sigma_i^{-1}\mathbf{x} + \mu_i^T\Sigma_i^{-1}\mathbf{x} - \frac{1}{2}\mu_i^T\Sigma_i^{-1}\mu_i \end{aligned}$$

For the heteroscedastic model, the quadratic discriminant function is then

$$\begin{aligned} d_i(\mathbf{x}) &= -\frac{1}{2}(p\log|\Sigma_i| + \mu_i^T\Sigma_i^{-1}\mu_i) + \mu_i^T\Sigma_i^{-1}\mathbf{x} + \frac{1}{2}\mathbf{x}^T\Sigma_i^{-1}\mathbf{x} \\ &= \beta_{i0} + \beta_{i1}\mathbf{x} + \mathbf{x}^T\beta_{i2}\mathbf{x} \end{aligned}$$

where $\beta_{i0} = -\frac{1}{2}(p\log|\Sigma_i| + \mu_i^T\Sigma_i^{-1}\mu_i)$, $\beta_{i1} = \mu_i^T\Sigma_i^{-1}$, and $\beta_{i2} = -\frac{1}{2}\Sigma_i^{-1}$. Substituting the unbiased estimates for Σ_i and μ_i results in the estimated quadratic discriminant function.

A linear discriminant function is obtained if we can assume the group covariance matrices are equal, the homoscedastic model. In this case we replace the common covariance matrix Σ with the group covariance matrices Σ_i above. Once done, the quadratic term $\frac{1}{2}\mathbf{x}^T\Sigma_i^{-1}\mathbf{x}$ is constant for all groups and may be discarded from the

discriminant function leaving only the constant terms β_{i0} and the linear terms β_{i1}.

Proportional Covariance and Equal Correlation Matrices

For the proportional covariance matrices model we assume that $\Sigma_i = \kappa_i^2\Sigma$ for $i = 1, \ldots, g$. Under the assumption of p-variate normality of the feature variables, the maximum likelihood estimates (denoted by a 'hat' ^) of κ_i and Σ satisfy $\hat{\kappa}_i = \left(\frac{tr(\hat{\Sigma}^{-1}\hat{\Sigma}_i)}{p}\right)^{-1/2}$ and $\hat{\Sigma} = \sum_{i=1}^{g} \frac{n_i\hat{\Sigma}_i}{n\hat{\kappa}_i}$, where $\kappa_1 \equiv 1$, $tr()$ is the trace function, n_i is the number of observations in the training data from group i and $n = \sum n_i$ (McLachlan, 1992, p. 139). These equations are solved iteratively until convergence.

McLachlan (1992, p. 139) also provides an iterative solution for the equal correlation problem. Instead of working with the common correlation matrix, however, we work with the group covariance matrices such that $\Sigma_i = \mathbf{K}_i\Sigma\mathbf{K}_i$, where the diagonal matrices $\mathbf{K}_i = \text{diag}(\kappa_{i1}, \ldots, \kappa_{ip})$ and for the first group $\kappa_{ij} \equiv 1$, for $j = 1, \ldots, p$. The estimating equations are then $\hat{\Sigma} = \sum_{i=1}^{g}(n_i/n)(\hat{\mathbf{K}}_i^{-1}\hat{\Sigma}_i\hat{\mathbf{K}}_i^{-1})$ and $\hat{\kappa}_{ij} = \sum_{k=1}^{p}\left(\left(\hat{\Sigma}^{-1}\right)_{kj}(\hat{\Sigma}_i)_{kj}\right)/\hat{\kappa}_{ik}$, for $i = 2, \ldots, g$ and $j = 1, \ldots, p$.

Common Principal Components

Flury (1984) developed the common principal component model, which is also discussed in McLachlan (1992, p. 140). Here, the group covariance matrices share the same principal axes, $\mathbf{A}$, which is expressed as $\Sigma_i = \mathbf{A}^T\Lambda_i\mathbf{A}$ where $\Lambda_i = \text{diag}(\lambda_{i1}, \ldots, \lambda_{ip})$.

A special case is the proportional covariance model where $\lambda_{ij} = \kappa_i^2\lambda_{1j}$ for $j = 1, \ldots, p$.

Canonical Variates

The canonical discriminant function is a dimension reduction technique that can be applied only to the homoscedastic model. Define **B** as the between-groups sum of squares product matrix divided by $g-1$,

$$\mathbf{B} = \frac{1}{g-1}\sum_{i=1}^{g}(\mu_i - \bar{\mu})(\mu_i - \bar{\mu})^T$$

where $\bar{\mu} = \sum_i \mu_i$. The canonical variates are then the eigenvectors associated with the eigenvalues of $\Sigma^{-1}\mathbf{B}$. There are at most $d = \min(g-1,\ p)$ nonzero eigenvalues. Denote the canonical variates by the $p \times d$ matrix Γ.

We can then write the constants and linear coefficients of the discriminant function as

$$\beta_{i0} = -\frac{1}{2}\left(p\sum_{f}^{d}\log\lambda_j\right) + \mu_i^T \Gamma\Gamma^T\mu_i,$$

$$\beta_{i1} = \mu_i^T\Gamma\Gamma^T.$$

PREDICTION

We assume that an observation with feature vector $\mathbf{X} = \mathbf{x}$ is drawn randomly from a mixture of g groups with probability density $f_X(\mathbf{x}) = \sum_{i=1}^{g} \pi_i f_i(\mathbf{x})$, where π_i are the mixing proportions and $f_i(\mathbf{x})$ is the probability density function for the observation for each group $i = 1, \ldots, g$. Using the notation of McLachlan (1992), denote the probability of group membership given an observation with feature vector $\mathbf{x}$ as $\tau_i(\mathbf{x}) = (\pi_i f_i(\mathbf{x})) / [\sum_{k=1}^{g} \pi_k f_k(\mathbf{x})]$. The optimal rule, or *Bayes Rule*, is to assign observation $\mathbf{x}$ to group k if $\tau_k(\mathbf{x}) = \max_{i=1}^{g} \tau_i(\mathbf{x})$.

The `discrim` function assumes the group density function for x is multivariate normal. Estimates for the mean, μ_i, and covariance, Σ_i, for the p-variate normal density for group i are estimated from training data, $\mathbf{x}_{ij}$, $i = 1, \ldots, g, j = 1, \ldots, n_i$. Treatment of the mixing proportions, π_i, is dependent on the sampling scheme used to obtain the training data.

There are two sampling schemes in which the training data can be obtained: mixture sampling and group conditional sampling. The mixture sampling design is where a random sample of n observations are obtained and each observation's group membership and feature vector is recorded, thereby making the number of observations from each group, n_i, multinomial random variables so the maximum likelihood estimate for π_i is n_i / n.

In group conditional sampling, the number of individuals sampled from each group is fixed. If the π_i are not known in advance, McLachlan (1992, pp. 31-33) discusses a technique to use an additional unclassified mixture sample to estimate the group proportions using the group conditional error rates obtained from the training data (the confusion matrix).

Plug-In

Plug-in estimates of the population densities are computed by substituting the unbiased estimates of the group means, $\bar{\mathbf{x}}_i$, and covariances, $\mathbf{S}_i$, for the parameters of the densities μ_i and Σ_i, without regard to their being random variables.

To obtain the plug-in estimates, use `predict` on a `discrim` object with the argument `method="plug-in"`:

```
> predict(exiris.het, method = "plug-in")

        groups Setosa Versicolor Virginica
 ...
69 Versicolor      0  0.8130906 0.1869094
70 Versicolor      0  0.9999643 0.0000357
71  Virginica      0  0.3359442 0.6640558
72 Versicolor      0  0.9999898 0.0000102
73 Versicolor      0  0.6993187 0.3006813
74 Versicolor      0  0.9721091 0.0278909
75 Versicolor      0  0.9999794 0.0000206
 ...
```

Unbiased Estimates

An unbiased estimate of the log of the p-variate normal densities is obtained as follows. Denote the estimated squared Mahalanobis distance between an observed feature set $\mathbf{x}$ and the mean of group i to be $\delta_i(\mathbf{x}|\bar{\mathbf{x}}_i, \mathbf{S}_i) = (\mathbf{x} - \bar{\mathbf{x}}_i)^T \mathbf{S}_i^{-1} (\mathbf{x} - \bar{\mathbf{x}}_i)$, where $\mathbf{S}_i$ is the unbiased estimate of the group covariance, Σ_i. Based on the Wishart distribution, its expected value is

$$\frac{n-1}{n-p-2}\left(\delta_i(\mathbf{x}|\mu_i, \Sigma_i) + \frac{p}{n_i}\right) \qquad i = 1, \ldots, g,$$

where $\delta_i(\mathbf{x}|\mu_i, \Sigma_i) = (\mathbf{x} - \mu_i)^T \Sigma_i^{-1} (\mathbf{x} - \mu_i)$. Moreover, the expected value of $\log|\mathbf{S}_i|$ is

$$\log|\Sigma_i| - p\log\left(\frac{n_i - 1}{2}\right) + \sum_{k=1}^{p} \psi\left(\frac{1}{2}(n_i - k)\right) \qquad i = 1, \ldots, g.$$

In the above equation, ψ is the digamma function (McLachlan, 1992, p. 57). The results are used to compute unbiased log density estimates for the heteroscedastic model. Ripley (1996, p. 56) gives the unbiased estimator of the log of the p-variate normal density explicitly.

McLachlan (1992, p. 57) gives similar results for the homoscedastic model. Let **S** be the unbiased estimate of the common covariance Σ, then

$$E[\delta_i(\mathbf{x}|\bar{\mathbf{x}}_i,\ \mathbf{S})] = \frac{n-g}{n-g-p-1}\left(\delta_i(\mathbf{x}|\mu_i,\ \Sigma) + \frac{p}{n_i}\right)$$

To obtain unbiased estimates, use `predict` with `method="unbiased"`:

```
> predict(exiris.het, method = "unbiased")

        groups Setosa Versicolor Virginica
  ...
69 Versicolor      0  0.8052674 0.1947326
70 Versicolor      0  0.9999080 0.0000920
71  Virginica      0  0.3745987 0.6254013
72 Versicolor      0  0.9999702 0.0000298
73 Versicolor      0  0.7020916 0.2979084
74 Versicolor      0  0.9640201 0.0359799
75 Versicolor      0  0.9999439 0.0000561
  ...
```

Predictive

Predictive estimation of group membership is a Bayesian method. Here, we estimate the posterior density function for each group given the training data by taking the product of the p-variate normal probability density $f_i(\mathbf{x}|\mu_i,\ \Sigma_i)$ and the posterior probability density function of the unknown parameters, $\Theta = \{\mu_i,\ \Sigma_i\}_{i=1}^{g}$, $f_\Theta(\Theta|\mathbf{x}_{ij}) \propto l_\Theta(\Theta|\mathbf{x}_{ij})p(\Theta)$, and integrating out the Θ. A non-informative prior for the unknown mean and covariance is derived using Jeffery's rule, and is taken to be $p(\Theta) \propto \prod_{i=1}^{g} |\Sigma_i|^{(p+1)/2}$ and the likelihood of the unknown parameters given the training data is

$$l(\Theta|\mathbf{x}_{ij}) \propto \prod_{i=1}^{g}\prod_{j=1}^{n} |\Sigma_i|^{-1/2} \exp(-\delta_i(\mathbf{x}_{ij})) \quad .$$

The resulting densities are multivariate t. For the heteroscedastic model we have

$$f_i(\mathbf{x}|\bar{\mathbf{x}}_i,\ \mathbf{S}_i) \propto \left(\frac{n_i}{n_i^2-1}\right)^{p/2} \left(\frac{\Gamma\left(\frac{n_i}{2}\right)}{\Gamma\left(\frac{n_i-p}{2}\right)|\mathbf{S}_i|}\right) \left(1+\frac{n_i\hat{\delta}_i(\mathbf{x})}{n_i^2-1}\right)^{-\frac{n_i}{2}}$$

whereas for the homoscedastic model we have

$$f_i(\mathbf{x}|\bar{\mathbf{x}}_i,\ \mathbf{S}_i) \propto \left(\frac{n_i}{n_i+1}\right)^{p/2} \left(1+\frac{n_i\hat{\delta}_{i,\ E}(\mathbf{x})}{(n_i+1)(n-g)}\right)^{-\frac{n-g+1}{2}}$$

Further details and original authors can be found in Krzanowski and Marriot (1995, §9.20 and §9.21), McLachlan (1992, p. 68), and Geisser (1982, pp. 106–108).

If the group proportions are also unknown, estimation of the π_i can be done within the Bayesian framework using a Dirichlet prior proportional to $\prod_{i=1}^{g}\pi_i^{\alpha_i}$ (Krzanowski and Marriot, 1995, p.20). The posterior density is then proportional to

$$p(\pi_1,\ \ldots,\ \pi_g|n_1,\ \ldots,\ n_g,\ \mathbf{x}) \propto \prod_{i=1}^{g}\pi_i^{\alpha_i+n_i} f_{\mathbf{X}}(\mathbf{x})$$

Krzanowski and Marriot (1995) then remove π_i from the posterior probability that $\mathbf{x}$ belongs to group i by multiplying $\pi_i(f_i(\mathbf{x}|\bar{\mathbf{x}}_i,\ \mathbf{S}_i))$ by $\wp(\pi_1,\ \ldots,\ \pi_g|n_1,\ \ldots,\ n_g,\ \mathbf{x})$ and integrating out the π_j, $j = 1,\ \ldots,\ j$. The result is

$$\tau_i^*(\mathbf{x}) = \frac{(n_i+\alpha_i+1)f_i(\mathbf{x}|\bar{\mathbf{x}}_i,\ \mathbf{S}_i)}{\sum_{j=1}^{g}(n_j+\alpha_j+1)f_j(\mathbf{x}|\bar{\mathbf{x}}_j,\ \mathbf{S}_j)}$$

In the case of group condition sampling Krzanowski and Marriot (1995) set $n_i = 0$. A non-informative prior sets $\alpha_i = -1/2$ (Box and Tiao, 1973). As pointed out by Ripley (1996, p. 53), we are left with a Bayes rule that is essentially the same as $\tau_i(\mathbf{x})$.

To obtain predictive estimates, use `predict` with `method="predictive"`:

```
> predict(exiris.het, method = "predictive")

        groups    Setosa Versicolor Virginica
  ...
69 Versicolor       0  0.7970519 0.2029481
70 Versicolor       0  0.9998288 0.0001712
71  Virginica       0  0.3816198 0.6183802
72 Versicolor       0  0.9999235 0.0000765
73 Versicolor       0  0.7021942 0.2978058
74 Versicolor       0  0.9582749 0.0417251
75 Versicolor       0  0.9998786 0.0001214
  ...
```

ERROR ANALYSIS

Apparent Error Rate

An estimate of the misclassification rate provides a quantitative assessment of the discriminating power of an estimated discriminant function. One such estimate is the apparent error rate where each observation in the training data is classified and the number of misclassifications for each group is divided by the group sample size. This estimate provides an overly optimistic assessment of the true error rate (conditioned on the training data). The overall conditional error rate is weighted means of the group error rates where the weights are the mixture proportions.

```
> exiris.plug <- predict(exiris.het)
> tbl <- table(exiris$Species, exiris.plug$groups)
> tbl <- cbind(tbl, error = (apply(tbl,1,sum)-diag(tbl))/
+ exiris.het$counts)
> tbl

           Setosa Versicolor Virginica error
    Setosa     50          0         0  0.00
Versicolor      0         48         2  0.04
 Virginica      0          1        49  0.02

> sum(exiris.het$prior*tbl[,'error'])

[1] 0.02
```

Cross-Validation

Cross-validation is a leave-one-out technique for estimating the error rate conditioned on the training data. Conceptually, each observation is systematically dropped, the discriminant function reestimated, and the excluded observation classified. Fortunately, for the homoscedastic, heteroscedastic, and spherical models, the discriminant function does not need to be reestimated. The leave-one-out formulas for Mahalanobis distance and the determinant of the estimated covariances matrices for the homoscedastic and heteroscedastic models can be found in McLachlan (1992, pp. 342-343) and Ripley (1996, p. 100).

For example, the commands below estimate the error rate of the `exiris.het` object:

```
> exiris.cross <- crossvalidate(exiris.het)
> tbl <- table(exiris$Species, exiris.cross$groups)
> tblx <- table(exiris$Species, exiris.cross$groups)
> tblx <- cbind(tblx, error = (apply(tblx,1,sum)-
+ diag(tblx))/exiris.het$counts)
> tblx

           Setosa Versicolor Virginica error
    Setosa     50          0         0  0.00
Versicolor      0         47         3  0.06
 Virginica      0          1        49  0.02

> sum(exiris.het$prior*tblx[,'error'])

[1] 0.02666667
```

Estimating Error Rates Based on Posterior Probabilities

One can use the posterior probabilities for error rate estimation. Borrowing the discussion from McLachlan (1992, p. 365) or Ripley (1996, pp. 75-76), let $r(\mathbf{x})$ be the discriminant rule for the observation $\mathbf{X} = \mathbf{x}$ randomly chosen from a mixed population that has the mixture distribution $f_{\mathbf{X}}(\mathbf{x}) = \sum_{i=1}^{g} \pi_i f_i(\mathbf{x})$, $r(\mathbf{x}) = i$ if $\max_j(\tau_j(\mathbf{x})) = \tau_i(\mathbf{x})$, where $\tau_i(\mathbf{x}) = \pi_i f_i(\mathbf{x}) / f_{\mathbf{X}}(\mathbf{x})$ is the posterior probability of an observation belonging to group i. Also let $\mathrm{I}(i, j)$ be the indicator function that evaluates to 1 if $i = j$ and 0 otherwise. Then

$$\varepsilon_{ij} = \Pr\{r(\mathbf{X}) = j | \mathbf{X} \in G_i\} = \frac{\Pr(\mathbf{X} \in G_i,\ r(\mathbf{X}) = j)}{\pi_i}$$

$$= \frac{1}{\pi_i} E_{\mathbf{X}}[\tau_i(\mathbf{X}) \mathrm{I}(r(\mathbf{X}),\ j)] = \frac{1}{\pi_i} \sum_{k=1}^{g} \pi_k (E_k[\tau_i(\mathbf{X}) \mathrm{I}(r(\mathbf{X}),\ j)]$$

Substituting the expectation with the averages over the training data gives the posterior-based error rate estimator

$$\hat{e}_{ij} = \frac{1}{\hat{\pi}_i} \sum_{k=1}^{g} \frac{\hat{\pi}_k}{n_k} \sum_{l=1}^{n} \hat{\tau}_i(\mathrm{x}_l) \mathrm{I}(\hat{r}(\mathrm{x}_l), j) \cdot z_{li}$$

where $z_{lk} = 1$ if observation l from the training data came from group G_k. The following example exploits $\hat{\tau}_i = n_i / n$.

```
> Z <- diag(3)[exiris.cross$groups,]
> P <- NULL
> for (i in 1:3)
+ P <- rbind(P, apply(exiris.cross[,i+1]*Z, 2, sum)/
+ exiris.het$counts[i])
> P
```

```
     [,1]       [,2]       [,3]
[1,]    1 0.00000000 0.00000000
[2,]    0 0.93595428 0.02613819
[3,]    0 0.02404572 1.01386181
```

Note that $\sum_j \hat{e}_{ij}$ does not necessarily equal 1 so the estimate can be normalized.

```
> P/apply(P, 1, sum)
```

```
     [,1]      [,2]       [,3]
[1,]    1 0.0000000 0.00000000
[2,]    0 0.9728319 0.02716806
[3,]    0 0.0231675 0.97683250
```

The SAS® system takes a different approach to the formulation of the posterior probability error rate estimates. Here, they define the classification error rate for group i as

$$\begin{aligned} \hat{e}_i &= 1 - \int f_i(\mathbf{x}) d\mathbf{x} \\ &= 1 - \frac{1}{\pi_i} \int \tau_i(\mathbf{x}) f_{\mathbf{X}}(\mathbf{x}) d\mathbf{x} \end{aligned}$$

In the above equation, the interval of integration is over the set of observations such that τ_i is maximum, that is all $\mathbf{x}$ such that $r(\mathbf{x}) = i$ (SAS, 1988). This leads to the unstratified and stratified estimates

$$\hat{\ }_i = 1 - \frac{1}{\hat{\pi}_i \cdot n} \sum_{j=1}^{n} \hat{\tau}_i(x_j) I(\hat{r}(x_j), i)$$

and

$$\hat{\ }_i = \frac{1}{\hat{\pi}_i} \sum_{k=1}^{g} \frac{\hat{\pi}_k}{n_k} \sum_{j=1}^{n} \hat{\tau}_k(x_j) I(\hat{r}(x_j), i) \cdot z_{ji}$$

respectively. Huberty (1994, p. 90) also discusses these estimates. If $\hat{\pi}_i = n_i / n$, the stratified estimate reduces to the unstratified. Note also that negative estimates can occur.

```
> 1-apply(Z*as.matrix(exiris.cross[,-1]), 2, sum)/
+ exiris.het$counts

 Setosa Versicolor   Virginica
      0 0.06404572 -0.01386181
```

Example

```
> summary(exiris.het)

Call:
discrim(Species ~ Sepal.L. + Sepal.W. + Petal.L. +
     Petal.W., data = exiris, family =
     Classical(cov.structure = "heteroscedastic"))

...

Plug-in classification table:
           Setosa Versicolor Virginica Error
    Setosa     50          0         0  0.00
Versicolor      0         48         2  0.04
 Virginica      0          1        49  0.02
   Overall                              0.02
```

```
           Posterior.Error
    Setosa       0.0000000
Versicolor       0.0443544
 Virginica       0.0021883
   Overall       0.0155142
(from=rows,to=columns)

Rule Mean Square Error: 0.02304681
(conditioned on the training data)

Cross-validation table:
           Setosa Versicolor Virginica     Error
    Setosa     50          0         0 0.0000000
Versicolor      0         47         3 0.0600000
 Virginica      0          1        49 0.0200000
   Overall                              0.0266667
           Posterior.Error
    Setosa       0.0000000
Versicolor       0.0640457
 Virginica      -0.0138618
   Overall       0.0167280
(from=rows,to=columns)
```

The error estimates labeled `Posterior.Error` are the same estimates as those computed by SAS.

The rule mean squared error reported above is computed as $MSE = \frac{1}{n}\sum_{j=1}^{g}\sum_{i=1}^{n_j}(\hat{\tau}_j(\mathbf{x}_i) - z_{ij})^2$ where z_{ij} is an indicator variable that is equal to one if observation i is from group j and zero otherwise (McLachlan, 1992, p. 20).

REFERENCES

Box G.E.P. & Tiao G.C. (1973). *Bayesian inference in statistical analyses.* Reading, MA: Addison-Wesley Publishing Co.

Flury, B. (1984). Common principal components in *k* groups. *Journal of the American Statistical Association* **79**: 892-898.

Flury, B. (1988). *Common Principal Components and Related Multivariate Methods*. New York: John Wiley & Sons, Inc.

Geisser, S. (1982). Bayesian discrimination. In *Handbook of Statistics* (Vol. 2), P.R. Krishnaiah and L.N. Kanal (Eds.). Amsterdam: North-Holland.

Huberty, C.H. (1994). *Applied Discriminant Analysis.* New York: John John Wiley & Sons, Inc.

Krzanowski W.J. & Marriott F.H.C. (1995). *Multivariate analyses part 2: Classification, covariance structures and repeated measurements.* London: Arnold.

McLachlan, G.J. (1992). *Discriminant Analysis and Statistical Pattern Recognition.* New York: John Wiley & Sons, Inc.

Ripley, B.D. (1996). *Pattern Recognition and Neural Networks.* Cambridge: Cambridge University Press.

SAS User's Guide, Release 6.03 Edition (1988). Cary, NC: SAS Institute, Inc..

Stuart, A. & Ord, J.K. (1994). *Kendall's Advanced Theory of Statistics, Volume One, Distribution Theory.* New York: Halsted Press.

CLUSTER ANALYSIS 23

INTRODUCTION

Cluster analysis is the searching for groups (*clusters*) in the data, in such a way that objects belonging to the same cluster resemble each other, whereas objects in different clusters are dissimilar.

In two or three dimensions, clusters can be visualized. With more than three dimensions, or in the case of *dissimilarity* data (see below), we need some kind of analytical assistance.

Generally speaking, clustering algorithms fall into two categories:

1. *Partitioning Algorithms.* A partitioning algorithm describes a method that divides the data set into *k* clusters, where the integer *k* needs to specified. Typically, you run the algorithm for a range of *k*-values. For each *k*, the algorithm carries out the clustering and also yields a "quality index," which allows you to select the "best" value of *k* afterwards. Algorithms of this type described in this chapter are used by the functions `kmeans`, `pam`, `clara`, and `fanny`.
2. *Hierarchical Algorithms.* A hierarchical algorithm describes a method yielding an entire hierarchy of clusterings for the given data set. *Agglomerative* methods start with the situation where each object in the data set forms its own little cluster, and then successively merges clusters until only one large cluster remains which is the whole data set. The functions `agnes`, `mclust`, and `hclust` use agglomerative methods. *Divisive* methods start by considering the whole data set as one cluster, and then splits up clusters until each object is separate. Algorithms of this type are used in the functions `diana` and `mona`.

The clustering functions `daisy`, `pam`, `clara`, `fanny`, `agnes`, `diana`, and `mona` make up the *cluster library*, which implements the algorithms described in Kaufman & Rousseeuw (1990). The functions `kmeans`, `mclust`, and `hclust` are not part of the cluster library. They have a slightly different syntax than the cluster library functions.

DATA AND DISSIMILARITIES

Data sets for clustering can have either of the following structures:

1. $n \times p$ data matrix:

$$\begin{bmatrix} x_{11} & \ldots x_{1p} \\ \vdots & \vdots \\ x_{n1} & \ldots x_{np} \end{bmatrix}$$

   where rows stand for objects and columns stand for variables.

2. $n \times n$ dissimilarity matrix:

$$\begin{bmatrix} 0 & & & \\ d(2,\ 1) & 0 & & \\ d(3,\ 1) & d(3,\ 2) & 0 & \\ A & A & A & \\ d(n,\ 1) & d(n,\ 2) & \ldots\ldots & 0 \end{bmatrix}$$

   where $d(i,\ j) = d(j,\ i)$ measures the "difference" or *dissimilarity* between the objects i and j. This kind of data occurs frequently in the social sciences and in marketing.

Many of the clustering algorithms considered here operate on a dissimilarity matrix. If the data consist of an $n \times p$ data matrix, the algorithm first constructs the corresponding dissimilarity matrix.

The functions `kmeans`, `clara`, `mona`, and `mclust` operate on a data matrix. The `hclust` function operates on a dissimilarity matrix. The functions `pam`, `fanny`, `diana`, and `agnes` will take either a data or dissimilarity matrix.

Dissimilarity Matrices

The function `daisy` constructs a dissimilarity matrix. The algorithm used by `daisy` is described in full in Kaufman and Rousseeuw (1990, Chapter 1). Compared to the older function `dist` for which input must be numeric variables, `daisy` accepts other variable types (for example, nominal, ordinal, and asymmetric binary) even when the different types occur in the same data set.

Although we refer to the object produced by `daisy` or `dist` as a dissimilarity matrix, it is actually a vector representing the below-diagonal elements of such a matrix, with additional attributes giving information such as the number of observations.

Dissimilarities

The dissimilarity between two objects measures how different they are. Sometimes we can use an actual metric (distance function) between objects, but a dissimilarity function is not necessarily a metric. Often only the following three axioms of a metric are satisfied:

1. $d(i, i) = 0$
2. $d(i, j) \geq 0$
3. $d(i, j) = d(j, i)$

Computation

How we compute the dissimilarity between two objects depends on the type of the original variables.

By default, numeric columns are treated as interval-scaled variables, factors are treated as nominal variables, and ordered factors are treated as ordinal variables. The `type` argument to `daisy` may be used to specify that a column should be treated in a manner other than the default.

1. Interval-scaled variables

Interval-scaled variables are continuous measurements on a (roughly) linear scale. Typical examples are temperature, height, weight, and energy.

If all variables are interval-scaled, we can use an actual metric such as:

$$d(i, j) = \sqrt{\sum_{f=1}^{p} (x_{if} - x_{jf})^2} \quad \text{(Euclidean distance)} \tag{23.1}$$

or

$$d(i, j) = \sum_{f=1}^{p} |x_{if} - x_{jf}| \quad \text{(Manhattan distance)} \tag{23.2}$$

Note that the choice of measurement units strongly affects the resulting clustering. The variable with the largest dispersion will have the largest impact on the clustering. If all variables are considered equally important, the data need to be standardized first.

Put $m_f = \frac{1}{n} \sum_{i=1}^{n} x_{if}$ and $s_f = \frac{1}{n} \sum_{i=1}^{n} |x_{if} - m_f|$; then the standardized measurements are defined as follows:

$$z_{if} = \frac{x_{if} - m_f}{s_f} \tag{23.3}$$

Here we have used s_f, the *mean absolute deviation* instead of the usual standard deviation, because the former is more robust: since the deviations are not squared, the effect of outliers is somewhat reduced. Of course, there are more robust measures of dispersion, such as the median absolute deviation (the function `mad`). The advantage of using a robust measure of dispersion is that the z-scores of outliers do not become too small, hence the outliers remain detectable and visible in the clustering.

2. Continuous ordinal variables

Continuous ordinal variables are continuous measurements on an unknown scale, or where only the ordering is known but not the actual magnitude. Then the dissimilarities are computed as follows:

1. Replace the x_{if} by their rank $r_{if} \in \{1, \ldots, M_f\}$.
2. Transform the scale to [0,1] as follows: $z_{if} = \frac{r_{if} - 1}{M_f - 1}$.
3. Compute the dissimilarities as for interval-scaled variables.

3. Ratio-scaled variables

Ratio-scaled variables are positive continuous measurements on a nonlinear scale, such as an exponential scale. One example would be the growth of a bacterial population (say, with a growth function Ae^{Bt}). With this model, equal time intervals multiply the population by the same ratio.

There are different ways to compute dissimilarities for ratio-scaled variables:

1. Simply as interval-scaled variables, though this is not recommended as it can distort the measurement scale.
2. As continuous ordinal data.
3. By first transforming the data, perhaps by taking logarithms, and then treating the results as interval-scaled variables.

4. Discrete ordinal variables

A discrete ordinal variable has M possible values (scores) which are ordered. The dissimilarities are computed in the same way as for continuous ordinal variables.

5. Nominal variables

Nominal variables have M possible values, which are not ordered. The dissimilarity between objects i and j is usually defined as:

$$d(i,\ j) = \frac{\#\text{ variables taking different values for } i \text{ and } j}{\text{total number of variables}}$$

This is called the *simple matching coefficient.*

6. Symmetric binary variables

Symmetric binary variables have two possible values, coded 0 and 1, which are *equally* important. Examples include male and female, or vertebrate and invertebrate.

Symmetric binary variables are nominal variables, hence we again use the simple matching coefficient given above for *nominal variables.* Let us also consider the contingency table of the objects i and j:

i\j	1	0
1	a	b
0	c	d

We can then rewrite the simple matching coefficient as

$$d(i,\ j)\ =\ \frac{b+c}{a+b+c+d} \tag{23.4}$$

7. Asymmetric binary variables

Asymmetric binary variables have two possible values, one of which carries *more* importance than the other. The most meaningful outcome is coded as 1, and the less meaningful outcome as 0. Typically, 1 stands for the presence of a certain attribute (for example, a particular disease), and 0 for its absence.

The dissimilarity between i and j is then defined as:

$$d(i,\ j)\ =\ \frac{\#\text{ variables taking different values for } i \text{ and } j}{\text{total number of meaningful comparisons}}$$

Using the contingency table again, this becomes $d(i,\ j)\ =\ \frac{b+c}{a+b+c}$, which is called the *Jaccard coefficient.*

8. Variables of mixed types

The above formulas hold when all variables in the data set are of the same type. However, many data sets contain variables of different types. Therefore, we want a method to compute dissimilarities between objects when the data set contains p variables that may be of different types. For this the function `daisy` uses the formula

$$d(i,\ j)\ =\ \frac{\sum_{f=1}^{p} \delta_{ij}^{(f)}\ d_{ij}^{(f)}}{\sum_{f=1}^{p} \delta_{ij}^{(f)}} \in [0,\ 1]\ . \tag{23.5}$$

In Equation (23.5), $\delta_{ij}^{(f)} = 0$ if x_{if} or x_{jf} is missing, or if $x_{if} = x_{jf} = 0$ and f is an asymmetric binary variable. Otherwise, $\delta_{ij}^{(f)} = 1$. The term $d_{ij}^{(f)}$ is the contribution of variable f, which depends on its type:

1. If f is binary or nominal, $d_{ij}^{(f)} = 0$ if $x_{if} = x_{jf}$, and $d_{ij}^{(f)} = 1$ otherwise.
2. If f is interval-scaled, $d_{ij}^{(f)} = \frac{|x_{if} - x_{jf}|}{max_h(x_{hf}) - min_h(x_{hf})}$.
3. For ordinal and ratio-scaled variables, Spotfire S+ computes ranks r_{if} and $z_{if} = \frac{r_{if} - 1}{M_f - 1}$, and treats the z_{if} as interval-scaled.

Example: Calculating Dissimilarities

As a simple example of using `daisy`, we will calculate dissimilarities for a data frame where the rows are the first five integers:

```
> my.df <- data.frame(inds = 1:5)
> daisy(my.df)

Dissimilarities :
[1] 1 2 3 4 1 2 3 1 2 1

Metric :  euclidean
Number of objects :  5
```

PARTITIONING METHODS

Partitioning methods are based on specifying an initial number of groups, and iteratively reallocating observations between groups until some equilibrium is attained.

K-Means

One of the most well-known partitioning methods is *k-means*. In the k-means algorithm the observations are classified as belonging to one of k groups. Group membership is determined by calculating the centroid for each group (the multidimensional version of the mean) and assigning each observation to the group with the closest centroid.

The k-means algorithm alternates between calculating the centroids based on the current group memberships, and reassigning observations to groups based on the new centroids. Centroids are calculated using least-squares, and observations are assigned to the closest centroid based on least-squares. This use of a least-squares criterion makes k-means less resistant to outliers than the medoid-based methods which will be discussed in later sections.

The `kmeans` function performs k-means clustering. It is an older function that does not have special plot or summary methods. The main arguments to `kmeans` are dissimilarities as produced by `daisy` or `dist` and the number of clusters. Alternatively, a matrix of starting centroids may be specified in place of the number of centroids. If starting values are not specified the initial centroids are obtained using the hierarchical clustering algorithm in `hclust`.

Example: K-Means

The `ruspini` data were originally used by Ruspini (1970) in order to illustrate fuzzy clustering techniques. The data set consists of 75 points; see Figure 23.1, which was created using the function `plot.default(ruspini)`. We will use k-means to cluster the observations into four groups:

```
> kmeans(ruspini, 4)

Centers:
             x        y
[1,] 98.17647 114.8824
[2,] 20.15000  64.9500
[3,] 43.91304 146.0435
[4,] 68.93333  19.4000
```

```
Clustering vector:
 [1] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3
[28] 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 1 1 1 1 1 1 1 1 1 1 1
[55] 1 1 1 1 1 1 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4

Within cluster sum of squares:
[1] 4558.235 3689.500 3176.783 1456.533

Cluster sizes:
[1] 17 20 23 15

Available arguments:
[1] "cluster"  "centers"  "withinss" "size"
```

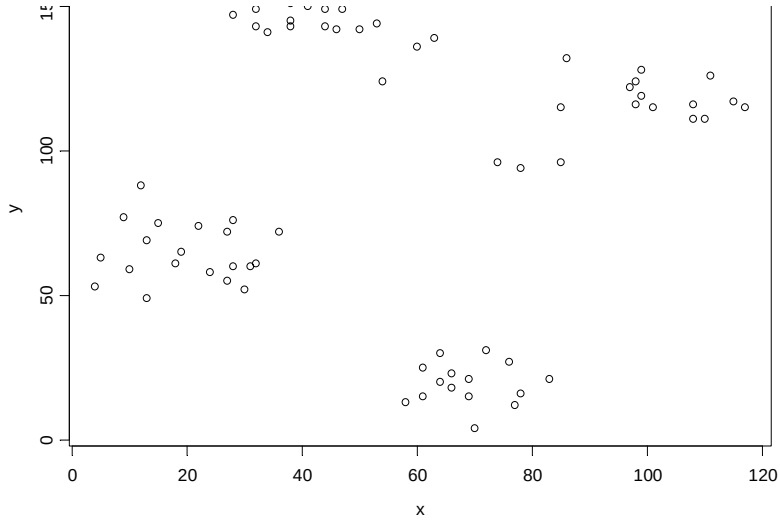

Figure 23.1: *The Ruspini data.*

Partitioning Around Medoids

The *partitioning around medoids* algorithm is similar to k-means but uses medoids rather than centroids.

The method `pam` is fully described in Chapter 2 of Kaufman and Rousseeuw (1990). Compared to the function `kmeans`, the function `pam` has the following features: (a) it accepts a dissimilarity matrix; (b) it is more robust because it minimizes a sum of dissimilarities instead of a sum of squared euclidean distances; (c) it provides novel graphical displays such as silhouette plots and clusplots.

Algorithm

The function `pam` operates on the dissimilarity matrix of the given data set. When it is presented with an $n \times p$ data matrix, `pam` first computes a dissimilarity matrix.

The algorithm computes k *representative objects*, called *medoids*, which together determine a clustering. The number k of clusters is an argument of the function.

Each object is then assigned to the cluster corresponding to the nearest medoid. That is, object i is put into cluster v_i when medoid m_{v_i} is nearer than any other medoid m_w:

$$d(i,\ m_{v_i}) \leq d(i,\ m_w) \text{ for all } w = 1,\ \ldots,\ k$$

The k representative objects should minimize the sum of the dissimilarities of all objects to their nearest medoid:

$$\text{objective function} = \sum_{i=1}^{n} d(i,\ m_{v_i})$$

The algorithm proceeds in two steps:

1. *Build-step*

   This step sequentially selects k *centrally located* objects to be used as initial medoids.

2. *Swap-step*

   If the objective function can be reduced by interchanging (swapping) a selected object with an unselected object, then the swap is carried out. This is continued until the objective function no longer decreases.

Graphical Displays: Silhouette Plots

A partition of the data, such as the clustering found by `pam`, can be displayed by means of the *silhouette plot* (Rousseeuw 1987).

For each object i, the silhouette value $s(i)$ is computed and then represented in the plot as a bar of length $s(i)$. In order to define $s(i)$, A denotes the cluster to which object i belongs, and the calculation proceeds as

$a(i)$ = average dissimilarity of i to all other objects of A

Now consider any cluster C different from A and define

$$d(i,\ C) = \text{average dissimilarity of } i \text{ to all objects of } C$$

After computing $d(i,\ C)$ for all clusters C *not equal to* A, we take the smallest of those:

$$b(i)\ =\ \min_{C \neq A} d(i,\ C)$$

The cluster B which attains this minimum, namely $d(i,\ B)\ =\ b(i)$, is called the *neighbor* of object i. This is the second-best cluster for object i.

The value $s(i)$ can now be defined:

$$s(i)\ =\ \frac{b(i) - a(i)}{max\{a(i),\ b(i)\}} \tag{23.6}$$

We see that $s(i)$ always lies between -1 and 1. The value $s(i)$ may be interpreted as follows:

$$s(i) \approx\ \ 1 \Rightarrow \text{object } i \text{ is well classified}$$

$$s(i) \approx\ \ 0 \Rightarrow \text{object } i \text{ lies between two clusters}$$

$$s(i) \approx\ -1 \Rightarrow \text{object } i \text{ is badly classified}$$

The silhouette of a cluster is a plot of the $s(i)$, ranked in decreasing order, of all its objects i. The entire silhouette plot shows the silhouettes of all clusters next to each other, so the *quality* of the clusters can be compared. The *overall average silhouette width* of the silhouette plot is the average of the $s(i)$ over all objects i in the data set (Figure 23.2).

It is possible to run `pam` several times, each time for a different k, and to compare the resulting silhouette plots (as in Figure 23.3). You can then select that value of k yielding the highest average silhouette width. If even that highest width is below (say) 0.25, one may conclude that no substantial structure has been found.

Graphical Displays: Clusplots

A *clusplot* is a bivariate plot displaying a partition (clustering) of the data (Figure 23.2). All observations are represented by points in the plot, using principal components or multidimensional scaling. Around each cluster an ellipse is drawn. The clusplot provides a convenient projection of the points into a two dimensional space with an indication of cluster membership.

Example: European Countries

The `euro` data set is an extract from the brochure "Cijfers en feiten: Een statistisch portret van de Europese Unie" (1994) published by Eurostat, the European agency for statistics. For each country belonging to the European Union during 1994, it gives the gross national product (`bbp`) in 1992 and the percentage of the gross national product due to agriculture (`landbouw`).

Here, both partitioning and hierarchical methods yield the same division of the European countries into two clusters; with one cluster consisting of four countries that are more oriented towards agriculture and whose gross national product is low relative to the other countries.

Table 23.1: *Countries of the European Union*

Code	Country	Code	Country
B	Belgium	I	Italy
D	Germany	IRL	Ireland
DK	Denmark	L	Luxembourg
E	Spain	NL	Netherlands
F	France	P	Portugal
GR	Greece	UK	United Kingdom

To view the `euro` data and cluster it with the `pam` function, type:

```
> euro

    landbouw  bbp
B       2.7 16.8
DK      5.7 21.3
D       3.5 18.7
GR     22.2  5.9
E      10.9 11.4
F       6.0 17.8
IRL    14.0 10.9
I       8.5 16.6
L       3.5 21.0
NL      4.3 16.4
P      17.4  7.8
UK      2.3 14.0

> pam(euro, 2)

Call:
pam(x = euro, k = 2)
Medoids:
  landbouw  bbp
D      3.5 18.7
P     17.4  7.8
Clustering vector:
 B DK D GR E F IRL I L NL P UK
 1  1 1  2 2 1   2 1 1  1 2  1
Objective function:
    build     swap
 3.429317 3.36061

Available arguments:
[1] "medoids"    "clustering" "objective"  "isolation"
[5] "clusinfo"   "silinfo"    "diss"       "data"
[9] "call"
```

We can visualize the cluster with the command below. The plot is displayed in Figure 23.2.

```
> plot(pam(euro, 2))
```

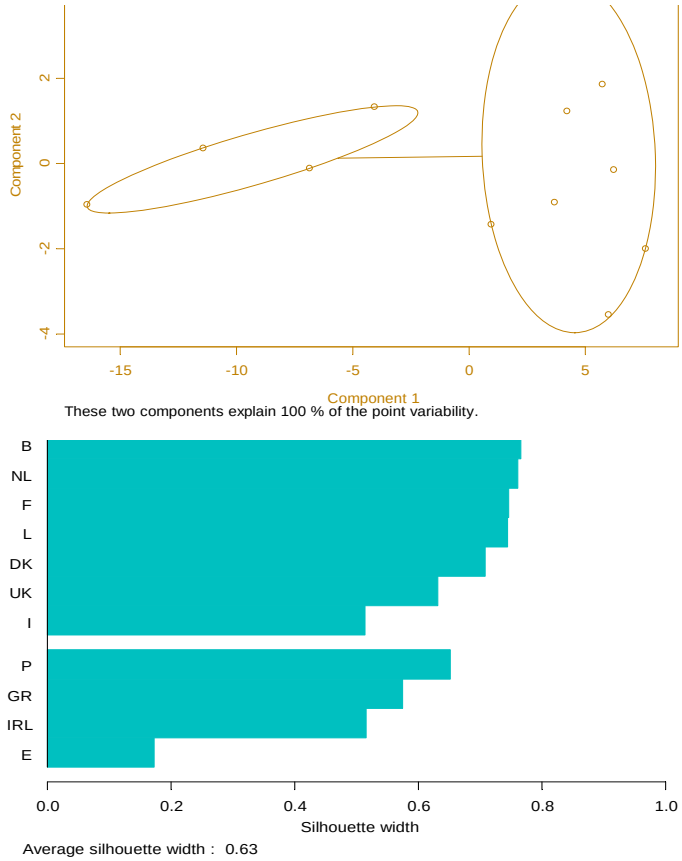

Figure 23.2: *Clusplot and silhouette plot of* `pam(euro, 2)`.

Example: Ruspini Data

We will compare the silhouette plots for two different partitions of the Ruspini data. We first use `pam` to partition the data into four clusters. After that, a partition into five clusters is constructed. The four medoids resulting from the first call are points in the centers of the four clusters. The second call to `pam` produces the same four medoids, and takes an intermediate object as the fifth medoid. The minimal value reached for the objective function is a little smaller when five clusters are formed. However, that does not necessarily imply that the second clustering is better. From the clustering vector, and the numerical output per cluster, it can be seen that both clusterings are similar. The second partition places the three most outlying points of the third cluster in a separate cluster. This new cluster is an isolated one.

On the other hand, the clusters resulting from the second call are not as well-separated as those from the first call. Looking at the silhouette plots in Figure 23.3, the conclusion is similar. With the first clustering, all $s(i)$ are above 0.4. The second clustering yields very large silhouette widths for the new cluster with three objects. But some of the silhouette widths of the second and third cluster have decreased. That is, those objects lie somewhere between two clusters. According to the overall average silhouette width both clustering structures are approximately of the same quality, $k = 4$ slightly preferable over $k = 5$.

```
> plot(pam(ruspini, 4), which = 2)
> plot(pam(ruspini, 5), which = 2)
```

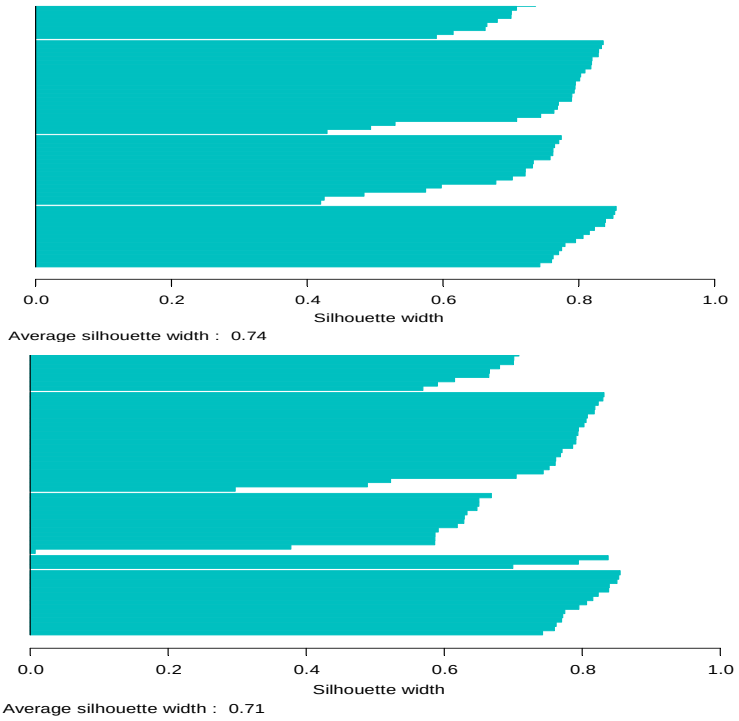

Figure 23.3: *Silhouette plots generated by* `pam(ruspini,4)` *and* `pam(ruspini,5)`.

Clustering Large Applications

As the k-means and partitioning around medoids techniques construct dissimilarities between all pairs of observations, their memory requirements are quadratic in the number of observations. This can be prohibitive when the number of observations is large. The Clustering Large Applications technique uses a less memory-intensive algorithm.

The method `clara` is fully described in Chapter 3 of Kaufman and Rousseeuw (1990). Compared to other partitioning methods such as `pam`, the `clara` function can deal with much larger data sets. Internally, this is achieved by considering data subsets of fixed size, so that the overall time and storage requirements become linear in the total number of objects, rather than quadratic.

The function `pam` needs to store the dissimilarity matrix of the entire data set (which has $O(n^2)$ entries) in central memory, while its computation time goes up accordingly. For larger data sets (say, with more than 250 objects) this becomes less convenient. To avoid this problem, the function `clara` does not compute the entire dissimilarity matrix at once. Therefore, this function only accepts input of an $n \times p$ data matrix.

Algorithm

The algorithm takes a data subset, and then applies the `pam` algorithm to it. This divides the data subset into k clusters. The remaining objects of the original data set are then assigned to the nearest medoid. In this way, all n objects are assigned. The objective function is then computed for the entire data set, namely by summing all n terms $d(i, m_{v_i})$.

This procedure is repeated for several data subsets, and the clustering with the lowest overall objective function is retained. In this way, we only need to compute and store the dissimilarity matrix of one data subset at any one time, which makes the overall order of complexity linear in n.

The first data subset is drawn randomly. Each of the following data subsets is forced to contain the currently best medoids, supplanted with randomly drawn objects.

Graphical Display

The clustering obtained by `clara` can also be represented by means of clusplots and silhouette plot, described in the previous section on `pam`. Due to the potential sizes of the data sets, the silhouette plot is given only for the best data subset.

Example: A Large Data Set

This data set, consisting of 500 two-dimensional points, is generated in Spotfire S+ using the following command:

```
> x <- rbind(cbind(rnorm(200, 0, 8),rnorm(200, 0, 8)),
+ cbind(rnorm(300, 50, 8), rnorm(300, 50, 8)))
> plot(x[,1], x[,2])
```

A plot of the points is shown in Figure 23.4.

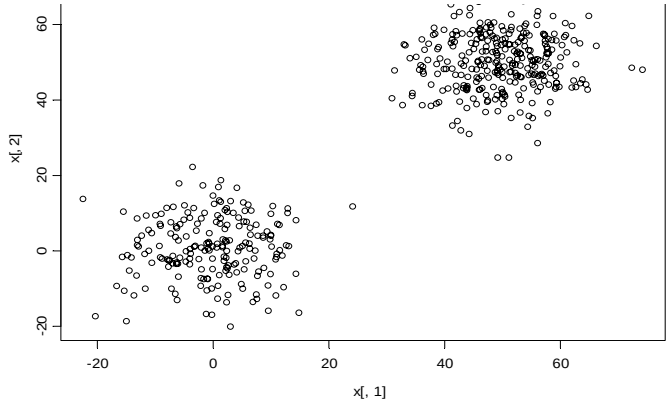

Figure 23.4: *A large data set of 500 points.*

The objects in the data set are clearly divided into two clusters. If `pam` had been used with this data set, `124750=500*499/2` dissimilarities would have been considered. The function `clara` uses only `946=44*43/2` dissimilarities, since the default sample size is `40+2*k = 40+2*2 = 44`. The `clara` function still finds the correct clustering. The average silhouette width, `0.82`, indicates a good clustering structure.

```
> a<- clara(x, 2)
> names(a)

[1] "sample"     "medoids"    "clustering" "objective"
[5] "clusinfo"   "silinfo"    "diss"       "data"
[9] "call"
```

```
> a$medoids

          [,1]         [,2]
[1,]  2.374335  -0.05215445
[2,] 49.636427  48.02134564

> plot(a)
```

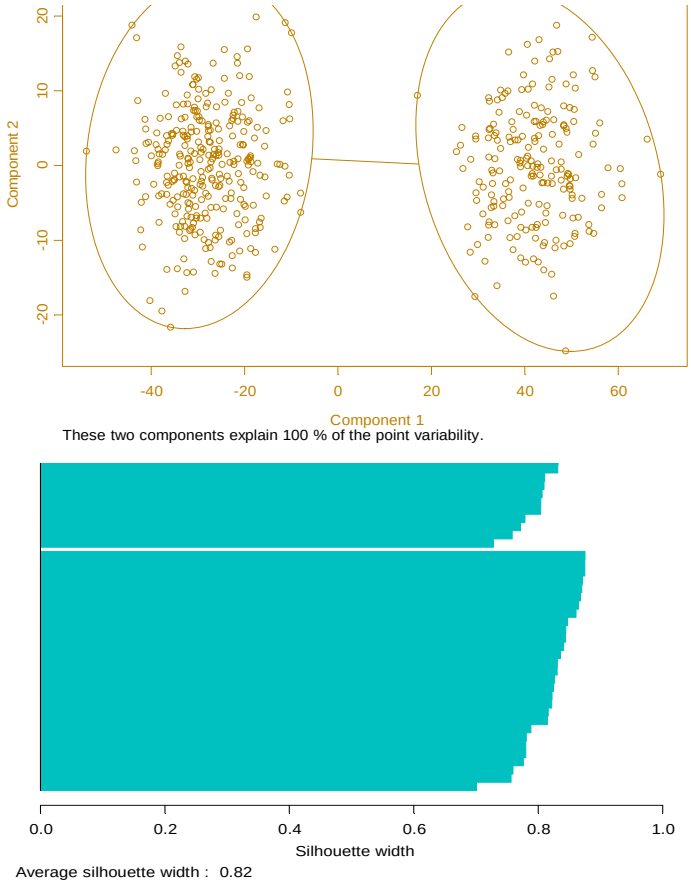

Figure 23.5: *Clusplot and silhouette plot of* `clara(x, 2)`*, where* `x` *is the large data set.*

Fuzzy Analysis

The functions `kmeans`, `pam`, and `clara` are *crisp* clustering methods. This means that each object of the data set is assigned to exactly one cluster. For instance, an object lying between two clusters must be assigned to one of them. In *fuzzy* clustering, each observation is given fractional membership in multiple clusters.

The method `fanny` is fully described in Chapter 4 of Kaufman and Rousseeuw (1990). Compared to other fuzzy clustering methods, `fanny` has the following features: (a) it accepts a dissimilarity matrix; (b) it is more robust to the "spherical cluster" assumption (see Kaufman and Rousseeuw); (c) its graphical display is in the form of a clusplot or silhouette plot.

For each object i and each cluster v there will be a *membership* u_{iv} which indicates how strongly object i belongs to cluster v.

Memberships have to satisfy the following conditions:

1. $u_{iv} \geq 0$ for all $i = 1, \ldots, n$ and all $v = 1, \ldots, k$.
2. $\sum_{v=1}^{k} u_{iv} = 1 = 100\%$ for all $i = 1, \ldots, n$.

Algorithm

The memberships are defined through minimization of:

$$\text{objective function} = \sum_{v=1}^{k} \frac{\sum_{i,j=1}^{n} u_{iv}^2 u_{jv}^2 d(i,j)}{2 \sum_{j=1}^{n} u_{jv}^2} \tag{23.7}$$

In this expression, the dissimilarities $d(i, j)$ are known and the memberships u_{iv} are unknown. The minimization is carried out numerically by means of an iterative algorithm, taking into account the above conditions that memberships need to obey. To have an idea of "how fuzzy" the resulting clustering is, *Dunn's partition coefficient* is computed:

$$F_k = \sum_{i=1}^{n} \sum_{v=1}^{k} \frac{u_{iv}^2}{n} \tag{23.8}$$

F_k always lies in the range $\left[\frac{1}{k}, 1\right]$.

Dunn's partition coefficient attains its extreme values in the following situations:

1. *entirely fuzzy clustering;* all $u_{iv} = \frac{1}{k} \Rightarrow F_k = nk\frac{1}{nk^2} = \frac{1}{k}$

2. *crisp clustering;* all $u_{iv} = 0$ or $1 \Rightarrow F_k = \frac{n}{n} = 1$

The normalized version of this coefficient is

$$F_k' = \frac{F_k - \frac{1}{k}}{1 - \frac{1}{k}} = \frac{kF_k - 1}{k - 1} \tag{23.9}$$

which always lies in the range [0, 1].

Graphical Display

For any fuzzy clustering, such as the one produced by `fanny`, the *nearest crisp clustering* method should be considered for graphical output. It assigns each object i to the cluster v in which it has the highest membership u_{iv}. This crisp clustering is then represented graphically by means of a clusplot or silhouette plot.

Example: Ruspini Data

When we call `fanny` with the `ruspini` data and $k = 4$, nearly all objects have a large membership to one of the clusters. The three objects that were placed in a separate cluster when calling `pam` for $k = 5$ now are classified in a fuzzy way, since none of their memberships is much higher than the other memberships. We conclude that the majority of the data can be divided into four clusters, but some objects are situated between the clusters. The nearest crisp clustering is the same as that from `pam` with $k = 4$. Hence, the silhouette plots are identical. But this is not always the case. When we call `fanny` for $k = 5$, the nearest crisp clustering is different from that produced by `pam`. The second cluster has been split instead of the third one. Because the average silhouette width is smaller than before, the clustering structure is less clear (Figure 23.6).

```
> plot(fanny(ruspini, 4), which = 2)
> plot(fanny(ruspini, 5), which = 2)
```

```
> fanny(ruspini, 4)

Call:
fanny(x = ruspini, k = 4)
 iterations objective
         12  422.8389
Membership coefficients:
         [,1]       [,2]       [,3]       [,4]
 1 0.65700251 0.10241150 0.09105386 0.14953212
 2 0.71377401 0.09277800 0.07872431 0.11472369
 3 0.76033966 0.07322710 0.06478832 0.10164492
  …
73 0.09673152 0.04828669 0.06629964 0.78868216
74 0.11367653 0.05369059 0.07298550 0.75964738
75 0.11731903 0.04977991 0.06446637 0.76843470
Coefficients:
 dunn_coeff normalized
  0.6237448  0.4983264
Closest hard clustering:
 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
 1 1 1 1 1 1 1 1 1  1  1  1  1  1  1  1  1  1  1  2  2  2
 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41
  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2
 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60
  2  2  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3
 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75
  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4

Available arguments:
[1] "membership" "coeff"      "clustering" "objective"
[5] "silinfo"    "diss"       "data"       "call"
```

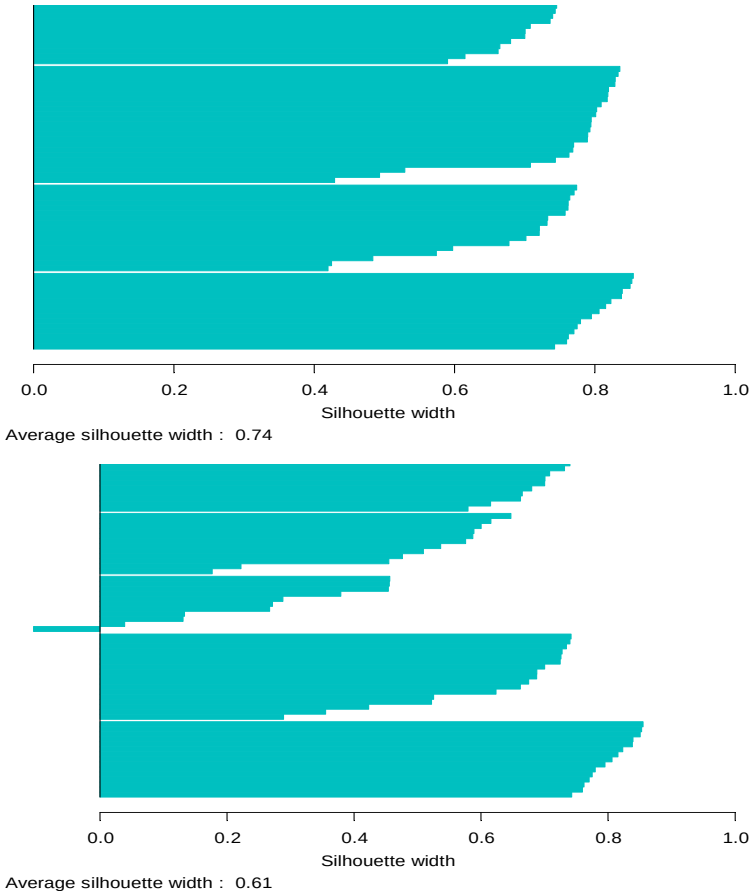

Figure 23.6: *Silhouette plots generated by* `fanny(ruspini,4)` *and* `fanny(ruspini,5)`.

HIERARCHICAL METHODS

The partitioning algorithms discussed previously are based on specifying an initial number of groups, and iteratively reallocating observations between groups until some equilibrium is attained. In contrast, hierarchical algorithms proceed by combining or dividing existing groups, producing a hierarchical structure displaying the order in which groups are merged or divided.

Agglomerative methods start with each observation in a separate group, and proceed until all observations are in a single group. *Divisive* methods start with all observations in a single group and proceed until each observation is in a separate group.

Agglomerative Nesting

The two most widespread clustering techniques are k-means and agglomerative hierarchical clustering. Spotfire S+ has three functions for agglomerative hierarchical clustering: `hclust`, `mclust`, and `agnes`. The oldest is `hclust`, and its capabilities have largely been subsumed by `mclust` and `agnes`. The `agnes` function provides more sophisticated plots than `mclust`, and has an interface consistent with the other functions in the cluster library. However, `mclust` does offer some computational methods not available in agnes, and is thus of interest in its own right. (The `mclust` function is discussed in a later section.)

The method `agnes` is fully described in Chapter 5 of Kaufman and Rousseeuw (1990). Compared to other agglomerative clustering methods such as `hclust`, `agnes` has the following features: (a) it yields the *agglomerative coefficient* which measures the amount of clustering structure found; (b) apart from the usual clustering tree, it also utilizes the *banner* plot.

As the function `agnes` is an agglomerative hierarchical clustering method, it yields a sequence of clusterings. In the first clustering each of the n objects forms its own separate cluster. In subsequent steps clusters are merged, until (after $n - 1$ steps) only one large cluster remains, consisting of all the objects.

Algorithm

The algorithm is based on dissimilarities only. If a data matrix is input, the function starts by computing the dissimilarity matrix.

Initially (at step 0), each object is considered as a separate cluster. The rest of the computation consists of iteration of the following steps:

1. Merge the two clusters with smallest between-cluster dissimilarity;
2. Compute the dissimilarity between the new cluster and all remaining clusters.

The between-cluster dissimilarity can be defined in various ways, notably:

1. Group average method

$$l(R,\ Q) = \frac{1}{|R||Q|} \sum_{i \in R,\ j \in Q} d(i,\ j)$$

2. Nearest neighbor method, or single linkage method

$$l(R,\ Q) = \min_{i \in R,\ j \in Q} d(i,\ j)$$

3. Furthest neighbor method, or complete linkage method

$$l(R,\ Q) = \max_{i \in R,\ j \in Q} d(i,\ j)$$

The *group average method* is taken as the default, based on arguments of robustness and consistency.

The function `agnes` also provides the *agglomerative coefficient* (Rousseeuw 1986), which measures the clustering structure of the data set. For each object i, $d(i)$ denotes its dissimilarity to the first cluster it is merged with, divided by the dissimilarity of the merger in the last step of the algorithm. The agglomerative coefficient (AC) is defined as the average of all $1 - d(i)$. Because the AC grows with the number of objects, this measure should not be used to compare data sets of very different sizes.

Graphical Display: The Clustering Tree and Banner

The hierarchy obtained from `agnes` can be graphically displayed in two ways, by means of a *clustering tree* or by a *banner.*

1. *Clustering tree*. This is a tree in which the leaves represent objects. The vertical coordinate of the place where two branches join equals the dissimilarity between the corresponding clusters.
2. *Banner.* The banner shows the successive mergers from left to right. Imagine the ragged flag parts at the left, and the flagstaff at the right; the objects are listed from top to bottom. The mergers, which commence at the between-cluster dissimilarity, are represented by horizontal bars of the correct length. The banner thus contains the same information as the clustering tree.

Note that the agglomerative coefficient defined above can also be defined as the average width (or the percentage filled) of the banner plot.

Example: Republican Votes Data

The `votes.repub` data set is standard in Spotfire S+. This matrix contains the percentage of people in the 50 states of the USA that voted Republican in the 31 presidential elections between 1856 and 1956. If a state did not yet belong to the USA in the year in question, an `NA` value is given.

When `agnes` is applied to this data set, the clustering tree indicates a division of the data into two well-separated clusters. A cluster containing eight of the Southern states is merged with the other states in the last step. The dissimilarity between the two clusters is large in comparison with the dissimilarities of the mergers at the other stages. When the complete linkage method is used, the same clustering structure is found. The clustering tree obtained by the single linkage method looks very different. Upon closer scrutiny, one sees that the states which are merged in the final steps are exactly those states that the other methods considered as a separate cluster. The single linkage method has a tendency towards chains of clusters, which causes the differences between the trees in this example. The `diana` function discussed in the next section finds the same main clustering structure: the eight Southern states are already split off at the first stage.

Since all of these hierarchical methods seem to agree on the division of the data set into two clusters, the conclusion might be that the voting behavior in the Southern states of the USA is rather different from that in the other states. The further division of the clusters is not so clear-cut: different methods yield more or less different structures.

```
> plot(agnes(votes.repub), which = 2)
```

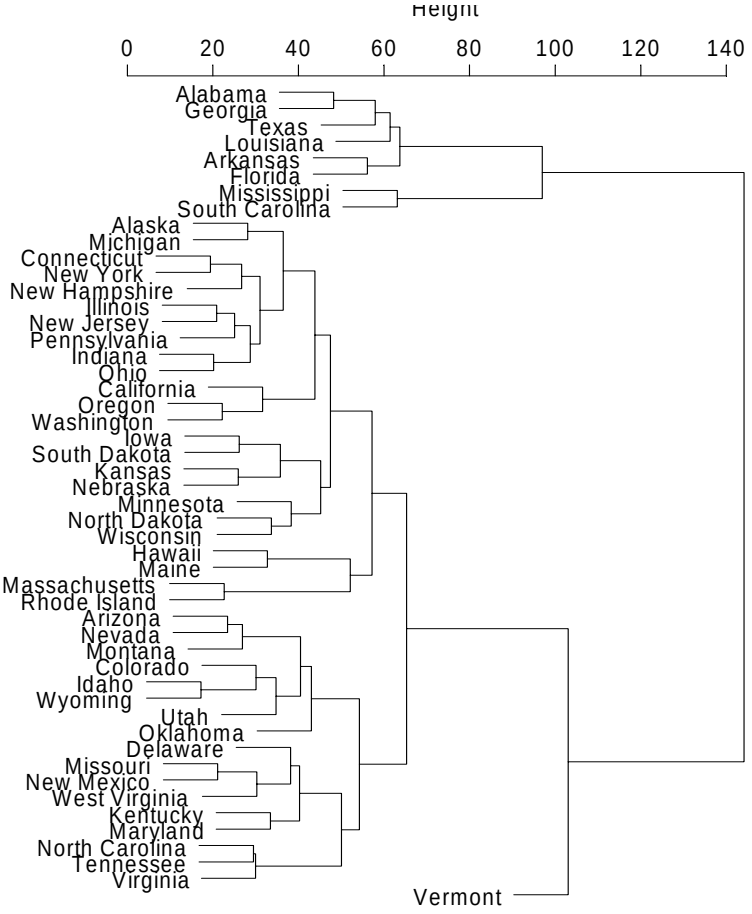

Figure 23.7: *Clustering tree of* `agnes(votes.repub)`.

Note that Figure 23.7 and Figure 23.8 have been rotated in this manual. When you run the code above, the resulting plots have trees with branches downward and labels on the bottom. Spotfire S+ does not provide an easy method for rotating plots to match what you see in these figures.

Divisive Analysis

While agglomerative clustering starts with many groups and combines them to form one group, divisive analysis starts with one group and repeatedly divides groups to form many groups.

The method `diana` is fully described in Chapter 6 of Kaufman and Rousseeuw (1990). It is probably unique in computing a divisive hierarchy, because most other software for hierarchical clustering is agglomerative. Moreover, `diana` provides (a) the divisive coefficient which measures the amount of clustering structure found; and (b) the banner plot.

The function `diana` is a divisive hierarchical method. The initial clustering (at step 0) consists of one large cluster containing all n objects. In each subsequent step, the largest available cluster is split into two smaller clusters, until finally all clusters contain but a single object.

In the first step of an agglomerative method, there are $\binom{n}{2} = \frac{n(n-1)}{2}$ possible ways to merge two clusters. But in the first step of a divisive method, we are faced with $2n-1-1$ possibilities to split up the data set into two clusters. The latter number is much larger than the first, and in practice it is not feasible to try all possible splits.

Algorithm

To avoid considering all possible splits, `diana` divides the data set in the following way (based on dissimilarities only):

1. Find the most disparate object, which is the one with the highest average dissimilarity to the other objects. This object initiates the *splinter group*, analogous to a dissenting fraction of a political party.
2. For each object i outside the splinter group, compute

   $$V_i = \text{average}_{j \notin \text{splinter group}} d(i, j) - \text{average}_{j \in \text{splinter group}} d(i, j)$$

   to find the object h for which this difference is largest. If $V_h > 0$, then h is on average closer to the splinter group than to the remainder, so add object h to the splinter group.
3. Repeat step 2 until all differences V_h are negative. The data set is then split into two clusters.

4. Select the cluster with the largest diameter; the diameter of a cluster is the largest dissimilarity between any two of its objects. Then divide this cluster as in steps 1 to 3.
5. Repeat step 4 until all clusters contain only a single object.

The function `diana` also provides the *divisive coefficient* (Rousseeuw, 1986), which measures the clustering structure of the data set. For each object i, $d(i)$ denotes the diameter of the last cluster to which it belongs (before being split off as a single object), divided by the diameter of the whole data set. The divisive coefficient (DC) is then defined as the average of all $d(i)$. Like the AC in the previous section on `agnes`, the DC also grows with the number of objects. Therefore, the DC should not be used to compare data sets of very different sizes.

Graphical Display

The hierarchy obtained from `diana` can again be graphically displayed either as a clustering tree or as a banner. Note that the divisive coefficient (DC) defined above can also be defined as the average width (or the percentage filled) of the banner plot.

Examples

We mentioned in the section Agglomerative Nesting that `diana` gives a clustering tree quite similar to that from `agnes` on the Republican voting data. The following command shows this:

```
> plot(diana(votes.repub), which = 2)
```

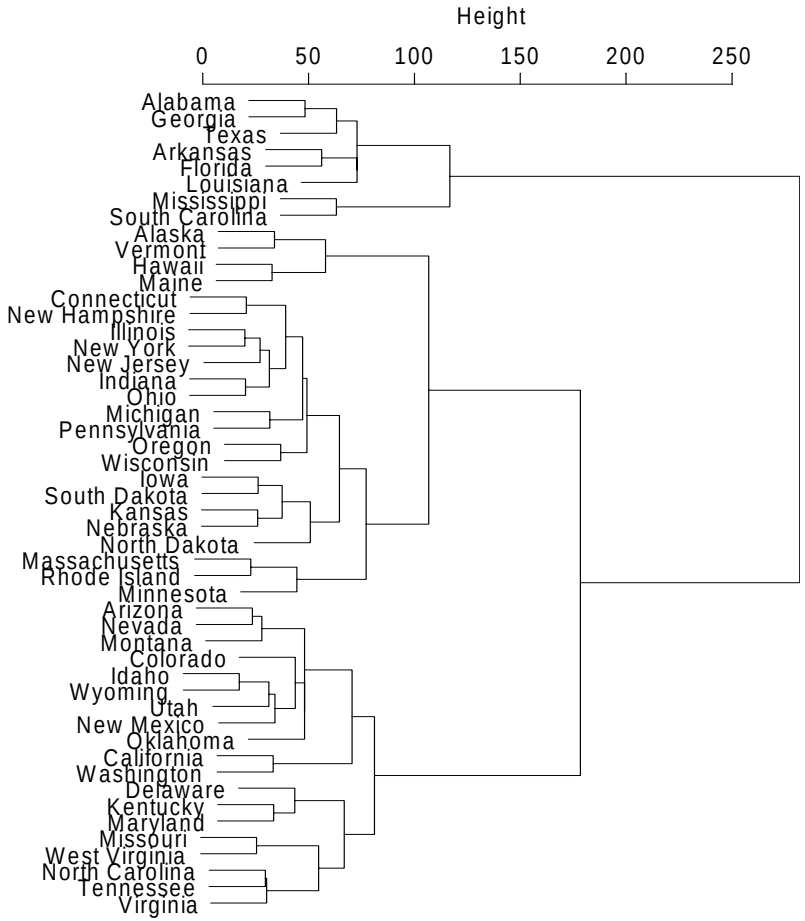

Figure 23.8: *Clustering tree of* `diana(votes.repub)`.

Monothetic Analysis

When all of the variables in a data set are binary, a natural way to divide the observations is by splitting the data into two groups based on the two values of a particular binary variable. Monothetic analysis produces a hierarchy of clusters in which at each step a group is split in two based on the value of one of the binary variables.

The method `mona` is fully described in Chapter 7 of Kaufman and Rousseeuw (1990). It is a different type of divisive hierarchical method. Contrary to `diana`, which can process a dissimilarity matrix as well as a data matrix with interval-scaled variables, `mona` operates on a data matrix with binary variables. For each split `mona` uses a

single (well-chosen) variable, which is why it is called a *monothetic* method. Most other hierarchical methods (including `agnes` and `diana`) are *polythetic*; that is, they use all variables simultaneously.

Algorithm

First all missing values in the binary data matrix (all those values *not* = 0 or 1) are replaced by estimated values, obtained as follows. Suppose that x_{if} is missing. Then we consider any other variable g, and construct the contingency table

f\g	1	0
1	a_{fg}	b_{fg}
0	c_{fg}	d_{fg}

The association between f and g is then defined as

$$A_{fg} = |a_{fg}d_{fg} - b_{fg}c_{fg}|$$

The variable t for which $A_{ft} = \max_g A_{fg}$ is the most correlated with variable f. The missing values of f are then estimated by means of variable t in the following way:

$$\text{put } x_{if} = x_{it} \text{ when } a_{ft}d_{ft} - b_{ft}c_{ft} > 0$$

$$\text{put } x_{if} = 1 - x_{it} \text{ when } a_{ft}d_{ft} - b_{ft}c_{ft} < 0$$

When all missing values have been replaced, the actual splitting can begin. If the data matrix cannot be filled in completely, due to too many missing values in the original data, the method stops with an error message.

The `mona` algorithm constructs a hierarchy of clusterings, starting with one large cluster. In each step, each available cluster is divided according to one variable. The cluster is divided into two: one cluster with all objects having value 1 for that variable, and another cluster with all objects having value 0 for that variable.

The variable used for splitting a cluster is the variable with the largest total association to the other variables. The association between variables f and g is given by the expression A_{fg} above, but now the contingency table uses only the objects of the cluster to be split. The total association of a variable f is then defined as:

$$A_f = \sum_{g \neq f} A_{fg} \tag{23.10}$$

The variable t which satisfies $A_t = \max_f A_f$ is selected for splitting the cluster. We continue to divide clusters in this way, until each cluster consists of objects having identical values for all variables. Such clusters cannot be split any more. A final cluster is thus a singleton or an indivisible cluster.

Graphical Display

The clustering hierarchy constructed by `mona` can be represented by means of a banner. This is again a divisive banner; however, the length of a bar is now given by the number of divisive steps needed to make that split. Inside the bar, the variable responsible for the split is listed.

Example: Animals Data

Six binary attributes are considered for twenty animals.

Table 23.2: *Animal attributes.*

Abbreviation	Attribute
`war`	Warm or cold blooded
`fly`	Flying or nonflying
`var`	Vertebrate or invertebrate
`end`	Endangered or not

Table 23.2: *Animal attributes. (Continued)*

Abbreviation	Attribute
`gro`	Lives in social groups, or not
`hai`	Hairy or not hairy

This example illustrates the use of `mona`. The following call produces the banner plot shown in Figure 23.9:

```
> plot(mona(animals))
```

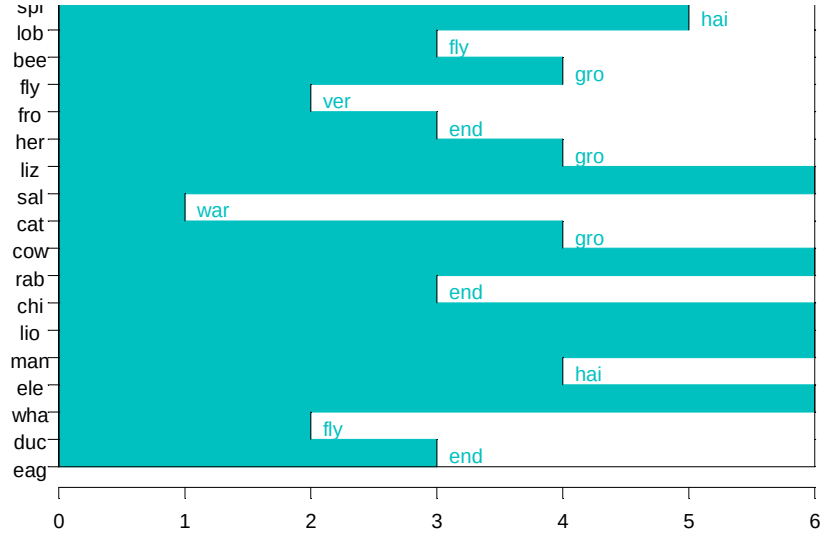

Figure 23.9: *Banner of* `mona(animals)`.

Figure 23.9 shows that `mona` classifies the animals according to the six attributes. In the first step, cold- and warm-blooded animals are put in separate clusters. The first cluster is then split into vertebrate and invertebrate animals, and the second cluster into flying and nonflying animals. Finally, after the fifth step, animals belonging to the same group have the same value for all six variables; on the banner, no bar is drawn between these animals.

If we wished to apply `agnes` or `diana` to this data set, we would have to compute the dissimilarities with `daisy`, because the variables are not numeric. The instruction is: `agnes(daisy(animals),diss=T)`. When we consider variable two (flying or not flying), and six (hairy or not hairy) as asymmetric binary, the call becomes:

```
agnes(daisy(animals,type=list(asymm=c(2,6))),diss=T)
```

The resulting clusterings will differ from the previous clustering. Since `agnes` and `diana` operate on the dissimilarities only, they do not use the individual variables. The function `mona` is probably more suitable for this example, where the animals have been classified nicely according to their attributes.

Table 23.3: *The animals and the three-letter abbreviations used in the data.*

ant	caterpillar	frog	man
bee	duck	hermit crab	rabbit
cat	eagle	lion	salamander
chimpanzee	elephant	lizard	spider
cow	fly	lobster	whale

```
> animals

    war fly ver end gro hai
ant   1   1   1   1   2   1
bee   1   2   1   1   2   2
cat   2   1   2   1   1   2
cpl   1   1   1   1   1   2
chi   2   1   2   2   2   2
cow   2   1   2   1   2   2
duc   2   2   2   1   2   1
eag   2   2   2   2   1   1
ele   2   1   2   2   2   1
fly   1   2   1   1   1   1
fro   1   1   2   2  NA   1
her   1   1   2   1   2   1
lio   2   1   2  NA   2   2
```

```
liz   1   1   2   1   1   1
lob   1   1   1   1  NA   1
man   2   1   2   2   2   2
rab   2   1   2   1   2   2
sal   1   1   2   1  NA   1
spi   1   1   1  NA   1   2
wha   2   1   2   2   2   1
```

Model-Based Hierarchical Clustering

Another approach to hierarchical clustering is *model-based clustering*, which is based on the assumption that the data are generated by a mixture of underlying probability distributions. The `mclust` function fits model-based clustering models. It also fits models based on heuristic criteria similar to those used by `pam`. The `mclust` function is separate from the cluster library, and has somewhat different semantics than the methods discussed previously.

Heuristic Criteria

The basic hierarchical agglomeration algorithm starts with each object in a group of its own. At each iteration it merges two groups to form a new group; the merger chosen is the one that leads to the smallest increase in the sum of within-group sums of squares. The number of iterations is equal to the number of objects minus one, and at the end all the objects are together in a single group. This is known variously as Ward's method, the sum of squares method, or the trace method.

The hierarchical agglomeration algorithm can be used with criteria other than the sum of squares criterion. For example, in the single link (or nearest neighbor) method, the distance between two groups is defined to be the smallest distance between any two members from different groups, and at each iteration the two closest groups are merged. The complete link method, also known as the compact or farthest neighbor method, is similar except that the distance between any two groups is defined to be the largest distance between any two members from different groups, while the centroid method defines the distance between two groups to be the distance between their centroids. The average weighted link method uses the average of the distances between the objects in one group and the objects in the other group. These are all heuristic criteria.

Model-Based Criteria

Model-based clustering is based on the assumption that the data are generated by a mixture of underlying probability distributions. Specifically, it is assumed that the population of interest consists of G

different subpopulations, and that the density of an observation x from the kth subpopulation is $f_k(x;\theta)$ for some unknown vector of parameters θ. Given data $D = (x_1, \ldots, x_n)$, we let $\gamma = (\gamma_1, \ldots, \gamma_n)$ denote the identifying labels, where $\gamma_i = k$ if x^i comes from the kth subpopulation. In the classification maximum likelihood procedure, θ and γ are chosen so as to maximize the likelihood.

$$L(D; \theta, \gamma) = \prod_{i=1}^{n} f_{\gamma_i}(x_i;\theta) \tag{23.11}$$

We consider mainly the situation where $f_k(x;\theta)$ is a multivariate normal density with mean μ_k and variance matrix Σ_k. If $\Sigma_k = \sigma^2 I$ for each k, where I is the identity matrix, then maximizing the likelihood (Equation (23.11)) is the same as minimizing the sum of within-group sums of squares that underlies Ward's method. Thus, Ward's method corresponds to the situation where clusters are hyperspherical with the same variance. If clusters are not of this kind (for example, if they are thin and elongated), Ward's method tends to break them up into hyperspherical blobs.

Other forms of Σ_k yield clustering methods that are appropriate in different situations; see Banfield and Raftery (1992). The key to specifying this is the eigenvalue decomposition of Σ_k. The eigenvectors of Σ_k specify the orientation of the kth cluster, the biggest eigenvalue specifies its variance or size, and the ratios of the other eigenvalues to the largest one specify its shape. We can constrain some but not all of these features (orientation, size, and shape) to be the same across clusters. For example, if we let $\Sigma_k = \sigma_k^2 I$, the criterion corresponds to hyperspherical clusters of different sizes; this is the *Spherical* criterion.

A criterion that appears to work well in a variety of situations results from constraining only the shape to be the same across clusters; this is denoted by *S**. Here you must specify the shape, represented by the eigenvalue ratios $\alpha_j = \lambda_j / \lambda_1$ $(j = 2, \ldots p)$, where $\{\lambda_1, \ldots, \lambda_p\}$ are

the eigenvalues ordered from largest to smallest. Specifying each $\alpha_j = 0.2$ leads to elliptical clusters that are moderately concentrated about a line in p-space, while choosing each $\alpha_j = 0.01$ yields very concentrated and linear clusters. Setting each $\alpha_j = 1$ gives the Spherical criterion as a special case. Your choice will be determined by the kind of data that you are working with, but we have found setting each $\alpha_j = 0.2$ often to be a good first guess.

Table 23.4 shows the different model-based clustering criteria and the assumptions that they embody.

Table 23.4: *Model-based clustering criteria with corresponding assumptions.*

Criterion	Reference	Distribution	Orientation	Size	Shape
Sum of Squares	Ward (1963)	Spherical	None	Same	Same
Spherical	Banfield and Raftery (1992)	Spherical	None	Different	Same
Determinant	Friedman and Rubin (1967)	Ellipsoidal	Same	Same	Same
S	Murtagh and Raftery (1984)	Ellipsoidal	Different	Same	Same
S*	Banfield and Raftery (1992)	Ellipsoidal	Different	Different	Same
Unconstrained	Scott and Symons (1971)	Ellipsoidal	Different	Different	Different

Choosing the Number of Clusters

In model-based clustering, choosing the number of clusters is the same as choosing a model for the data. A standard approach to this is to calculate the Bayes factor, B_k, for the model defined by k clusters against the model defined by a single cluster (that is, all the objects belong to the same group). The Bayes factor is the odds for one

model against another given the data (provided that one has no initial preference for either model). Thus the larger B_k, the more evidence there is for the existence of k clusters.

The approximate weight of evidence for k clusters (AWE_k) is an approximation to $2 \log B_k$; see Banfield and Raftery (1992). This is calculated by `mclust`. The larger AWE_k, the more evidence there is for the existence of k clusters. By definition, $\mathrm{AWE}_1 = 0$, so if all the AWE_k ($k = 2, \ldots, n$) are negative, there is no evidence for any clustering.

The value of k which maximizes AWE_k is the number of clusters for which there is the most evidence. However, we do not recommend using the AWE criterion to choose a single number of clusters unless the evidence is overwhelming. Rather, we suggest that the plot of AWE_k be inspected with a view to picking several plausible possibilities to be further investigated. The change in the approximate weight of evidence, $\mathrm{AWE}_k - \mathrm{AWE}_{k-1}$, is often large and positive for the first few values of k, $k = 2, \ldots, K$, say, and small or negative thereafter. If that is the case, ideas of parsimony suggest considering the classification into K groups, as well as the value of k which maximizes AWE_k, and any intervening values.

Robust Clustering

So far, it has been assumed that each object belongs to a cluster. However, even when a data set is made up mainly of clusters of the prescribed type, there may be other data points that do not follow this pattern. This possibility can be allowed for by extending the model given by Equation (23.11) to include such isolated observations, or outliers, assumed to occur according to a Poisson process with an intensity which is constant over the region from which the data have been drawn. The likelihood (Equation (23.11)) is modified accordingly. This yields a class of clustering algorithms designed to be robust to outliers; see Banfield and Raftery (1992).

Performing Model-Based Clustering

The function `mclust` performs the analyses described in this section. It carries out hierarchical agglomerative clustering using the six model-based criteria shown in Table 23.4, and also the five heuristic criteria discussed at the start of this section. For the model-based

criteria, it returns the AWE statistic for each number of clusters k; this is used to determine the number of clusters. Functions related to model-based clustering are listed in Table 23.5.

If `noise=T` is specified in `mclust`, it will do robust clustering (available for the model-based criteria only). If the existence of outliers is suspected, it may be a good idea to run `mclust` with `noise=F` and `noise=T` and to compare the results. Important differences between the resulting classifications would suggest that there are outliers that are contaminating the results, in which case either these outliers could be removed from the data sets and studied separately, or the robust clustering results (with `noise=T`) could be used. Note that the *number* of clusters indicated by the AWE in the nonrobust case (`noise=F`) will tend to be larger than in the robust case (`noise=T`), because in the nonrobust case some of the outliers may be classified as single-member groups.

Iterative relocation for any of the eleven criteria listed can be done using the function `mreloc`. The function `mclass` takes the output of `mclust` or `mreloc` and produces a classification of the data objects.

The output of `mclust` and `mreloc` can be used to plot and manipulate classification trees. The function `plclust` plots the tree, `subtree` extracts part of the tree, `clorder` reorders the leaves of the tree, `labclust` labels the leaves of the tree, and `cutree` creates groups using the tree.

The function `hclust` also does hierarchical agglomerative clustering, but only for three of the heuristic criteria included in `mclust`. `mclust` is much more general and is to be preferred for many purposes. However, `hclust` has two features which can be advantages in certain situations. It takes as argument a distance matrix rather than a data matrix, and it is applicable even when the data cannot be represented by points in Euclidean space; it accepts a dissimilarity matrix which need not be a distance matrix in the strict sense. A distance matrix can be calculated from a data matrix using the function `dist`. Also, unlike `mclust`, `hclust` returns the height at which each merger was made; this can yield more informative plots of the classification tree.

Table 23.5: *Functions for model-based clustering.*

Function	Use
`clorder`	Re-order leaves of a classification tree
`cutree`	Create groups from hierarchical agglomerative clustering
`labclust`	Label the leaves of a classification tree
`mclass`	Classify objects (uses output of `mclust`)
`mclust`	Model-based and heuristic hierarchical agglomerative clustering Determination of the number of clusters Robust Clustering
`mreloc`	Model-based iterative relocation
`plclust`	Plot a classification tree
`subtree`	Extract part of a classification tree

Example of Simple Use

We can use model-based clustering to explore the percent of votes given to the Republican candidate in presidential elections from 1856 to 1976. In the `votes.repub` data, rows represent the 50 states and columns the 31 elections.

```
> elect.years <- c("1960", "1964", "1968", "1972", "1976")
> votes.S <- mclust(votes.repub[,elect.years],
+ method = "S", noise = T)
> plclust(votes.S$tree, label = state.abb)
> plot( x = 1:length(votes.S$awe), y = votes.S$awe)
> # 9-cluster classification
> votes.9 <- mclass(votes.S, 9)
> # 3-cluster classification
> votes.3 <- mclass(votes.S, 3, votes.9)
> votes.3 <- mreloc(votes.3, votes.repub[,elect.years],
+ method = "S", noise = T)
```

CLUSTER LIBRARY ARCHITECTURE

Object-Oriented Structure

The algorithms of Kaufman and Rousseeuw (1990), summarized above, have been implemented in Spotfire S+ as a library of functions, which generate objects of seven different classes. For each class of objects, methods for textual or graphical output are available. Most of the objects are named after the function that generates them. In this way, classes `"pam"`, `"clara"`, `"fanny"`, `"agnes"`, `"diana"`, and `"mona"` exist. The seventh class, `"dissimilarity"`, is generated by the function `daisy` but is also be part of the objects of classes `"pam"`, `"clara"`, and `"fanny"`.

Some of these classes are grouped together and inherit from the same superclass. The created hierarchy of classes is as follows:

1. Class `"dissimilarity"`
2. Class `"partition"`
   - Class `"pam"`
   - Class `"clara"`
   - Class `"fanny"`
3. Class `"hierarchical"`
   - Class `"agnes"`
   - Class `"diana"`
4. Class `"mona"`

These classes have methods for the following functions:

1. `print`, for classes `"dissimilarity"`, `"pam"`, `"clara"`, `"fanny"`, `"agnes"`, `"diana"`, and `"mona"`.
2. `summary`, for classes `"pam"`, `"clara"`, `"fanny"`, `"agnes"`, `"diana"`, and `"mona"`. These `summary` methods return new objects of class `summary.`*`oldclass`*. For each of those new summary classes, a `print` method is available.
3. `plot`, for classes `"partition"`, `"agnes"`, `"diana"`, and `"mona"`.
4. `clusplot`, for class `"partition"`.
5. `pltree`, for class `"hierarchical"`.

The partition class has a method for the generic plot function that is common to all its subclasses.

Calling the Functions

The daisy function, for calculating dissimilarities, is similar to the older function dist. One advantage of daisy it that it accepts data sets with different types of variables. The function's header is

```
daisy(x, metric = "euclidean", stand = F, type = list())
```

When all variables are interval scaled, this specifies the metric to be used for calculating dissimilarities, and whether or not to standardize first. When other variable types occur, a list of types can be given. The output of daisy can be used as input for several of the new clustering functions.

The input arguments of the six clustering functions are similar. The calls to the six functions are given in Table 23.6.

Table 23.6: *Summary of clustering functions.*

Function	Description and example function call
daisy	Computes a dissimilarity matrix from a data matrix. `daisy(x, metric = "euclidean", stand = F, type = list())`
pam	A crisp partitioning method for smaller data sets. `pam(x, k, diss = F, metric = "euclidean", stand = F, save.x = T, save.diss = T)`
clara	A method for larger data sets (more than 250 objects) using the same basic algorithm as pam. `clara(x, k, metric = "euclidean", stand = F, samples = 5, sampsize = 40 + 2 * k, save.x = T, save.diss = T)`
fanny	A fuzzy partitioning method, employing the concept of memberships. `fanny(x, k, diss = F, metric = "euclidean", stand = F, save.x = T, save.diss = T)`

Table 23.6: *Summary of clustering functions.*

Function	Description and example function call
`agnes`	An agglomerative hierarchical method, computes a measure of the clustering found. `agnes(x, diss = F, metric = "euclidean", stand = F, method = "average", save.x = T, save.diss = T)`
`diana`	A divisive hierarchical method, computing a measure of the divisive clustering found. `diana(x, diss = F, metric = "euclidean", stand = F, save.x = T, save.diss = T)`
`mona`	A divisive hierarchical method that works on binary data. `mona(x)`

All functions, except for `clara` and `mona`, accept two possible input structures: a dissimilarity matrix or a data matrix. The logical argument `diss` tells the algorithm how `x` should be interpreted, the default being a data matrix of observations by variables. When a dissimilarity matrix is given as input, it is preferably an object of class `"dissimilarity"`. However, the functions will also accept dissimilarities produced with `dist`, or a vector that can be interpreted as a dissimilarity matrix.

The algorithms of `clara` and `mona` don't accept dissimilarities as input, but only accept the second input form: a matrix of observations by variables.

If a function has to compute dissimilarities from a given data matrix, the function needs to know which metric to use and whether or not to standardize first. These arguments are similar to the corresponding arguments of `daisy`. Since `mona` doesn't compute dissimilarities, it does not have the arguments `metric` and `stand`.

The function `clara` has two additional arguments, specifying the number of samples and the size of each sample. Also `agnes` has a special argument defining the method to be used for calculating dissimilarities between clusters.

By default, all functions store a copy of the data (if specified) and the dissimilarities as part of the returned model object. This information is needed to produce clusplots, but otherwise is provided solely for reference. The size of the returned model object may be reduced by setting `save.x` and/or `save.diss` to `FALSE`, in which case the data and/or dissimilarities are not returned.

Sometimes the output of the functions is rather extensive, especially when the `summary` method is invoked for an object of one of the `partition` classes. In those cases, the output scrolls off the screen. Therefore, all available components of the output are listed on the last output lines. Those components can be extracted from the result like a component from a list: *`object`*`$`*`component`*.

Objects resulting from the clustering functions can be given as input to high level graphics functions.

- The `plot` method for partition objects (`pam`, `clara`, and `fanny`) produces clusplots and silhouette plots.
- The `plot` methods for `agnes` and `diana` produce clustering trees and banner plots.
- The `plot` method for `mona` produces a banner plot.

More information and details about the input arguments and the structure of the output can be found in the help files.

REFERENCES

Banfield, J.D. & Raftery, A.E. (1992). Model-based Gaussian and non-Gaussian clustering. *Biometrics* **49**:803-822.

Eurostat(1994). *Cijfers en feiten: Een statisch portret van de Europese Unie.*

Friedman, H.P. and Rubin, J. (1967). On some invariant criteria for grouping data. *Journal of the American Statistical Association* **62**:1159-1178.

Gordon, A.E. (1981). *Classification: Methods for the Exploratory Analysis of Multivariate Data.* New York: Chapman and Hall.

Hartigan, J.A. (1975). *Clustering Algorithms.* New York: John Wiley & Sons, Inc.

Kaufman, L. and Rousseeuw, P.J. (1990). *Finding Groups in Data: An Introduction to Cluster Analysis.* New York: John Wiley & Sons, Inc.

Murtagh, F. and Raftery, A.E. (1984). Fitting straight lines to point patterns. *Pattern Recognition* **17**:479-483.

Murtagh, F. (1985). *Multidimensional Clustering Algorithms.* CompStat Lectures, 4. Heidelberg: Physica-Verlag.

Rousseeuw, P.J. (1986). A Visual display for hierarchical classification. In E. Diday, Y. Escoufier, L. Lebart, J. Pages, Y. Schektman, R. Tomassone (Eds.). *Data Analysis and Informatics*, vol. 4, Amsterdam: North-Holland, 743-48.

Rousseeuw, P.J. (1987). Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *J. Comput. Appl. Math.* **20**:53-65.

Ruspini, E.H. (1970). Numerical methods for fuzzy clustering. *Inform. Sci.* **2**:319-350.

Scott, A.J. and Symons, M.J. (1971). Clustering methods based on likelihood ratio criteria. *Biometrics* **27**:387-397.

Struyf, A., Hubert, M., and Rousseeuw, P.J. (1997). Integrating robust clustering techniques in Spotfire S+. *Computational Statistics and Data Analysis* **26**:17-37.

Ward, J.H. (1963). Hierarchical groupings to optimize an objective function. *Journal of the American Statistical Association* **58**:236-244.

HEXAGONAL BINNING

24

INTRODUCTION

This chapter describes the use of the `hexbin` function to graphically display spatial data. The S+SPATIALSTATS module, available for both UNIX and Windows, provides a more extensive set of tools for analyzing spatial data in the form of geostatistical data, lattice data, and spatial point patterns.

THE APPEAL OF HEXAGONAL BINNING

Hexagonal binning is a data grouping or reduction method typically employed on large data sets to clarify spatial structure. It can be thought of as partitioning a scatter plot into larger units to reduce dimensionality, while maintaining a measure of data density. The groups or bins are used to make hexagon mosaic maps colored or sized according to density. Rectangular or square grids are often used in this context for image-processing applications, for example, in grayscale, contour, and perspective maps. However, hexagons are preferable for visual appeal and representational accuracy (Carr, Olsen, and White, 1992). Hexagonal binning can also be used to group geostatistical data into a lattice for use in spatial regression modeling.

The data frame `quakes.bay` contains the locations of earthquakes in the San Francisco Bay Area for 1962-1981. Hexagonal bins are maintained in an object of class `"hexbin"`. Use the function `hexbin` to create the `hexbin` object for the earthquake data as follows.

```
> quakes.bin <- hexbin(quakes.bay$longitude,
+ quakes.bay$latitude)
> summary(quakes.bin)

Call:
hexbin(x = quakes.bay$longitude, y = quakes.bay$latitude)
Total Grid Extent: 36 by 31
      cell            count            xcenter
Min.   :  17.0   Min.   :  1.000   Min.   :-123.3
1st Qu.: 239.0   1st Qu.:  1.000   1st Qu.:-122.0
Median : 419.0   Median :  3.000   Median :-121.6
Mean   : 467.9   Mean   :  7.505   Mean   :-121.5
3rd Qu.: 696.0   3rd Qu.:  5.000   3rd Qu.:-121.0
Max.   :1091.0   Max.   :144.000   Max.   :-119.8
     ycenter
Min.   :36.01
1st Qu.:36.51
Median :36.94
Mean   :37.06
3rd Qu.:37.59
Max.   :38.50
```

The summary function shows the four components of the hexbin object and their distributions. The hexagon identified by cell contains count observations, and has center of mass at (xcenter, ycenter). The default settings for hexbin partition the range of x values into approximately 30 equal-sided hexagonal bins. The most useful bin size depends on the number of observations, and is best chosen iteratively. Plot the hexagonal bins as follows.

```
> trellis.device(color = F)
> at.quakes <- c(0, 10, 20, 30, 40, 50, 150)
> plot(quakes.bin, border = T, col.regions = 80:15,
+ at = at.quakes)
```

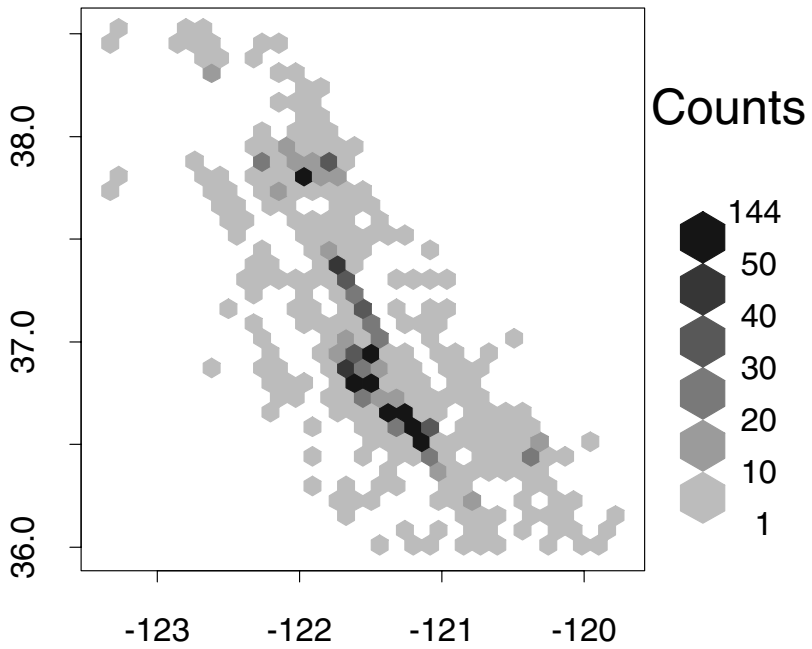

Figure 24.1: *The San Andreas Fault has a clear ridge of frequent earthquakes.*

The Trellis graphics device produces the best color and grayscale images for hexagonal binning. The default settings for plot.hexbin plot the hexagonal bins as a full tessellation, containing equally sized hexagons with color corresponding to grouped bin counts. By default, the groups are equal in range. Since the distribution of quakes.bin$count (shown by the summary output above) is skewed,

we have chosen the groups formed by `at.quakes`. The plot in Figure 24.1 shows the ridge of frequent earthquakes along the San Andreas Fault.

Hexagonal Bin Plot Styles

Besides the default grayscale style used here, there are four other plot styles available which plot the hexagons in varying sizes depending on cell density. Plot the earthquake `hexbin` object with differing sizes of hexagons as follows:

```
> plot(quakes.bin, style = "centroids", cuts = 6)
```

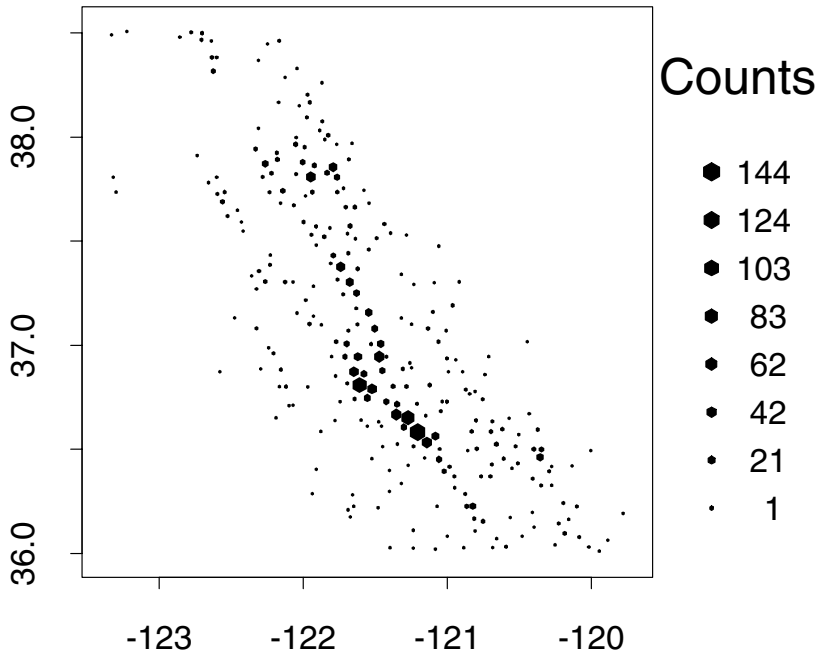

Figure 24.2: *As an alternative to using different grayscales in a hex plot, the hexagons can be drawn to a range of sizes. The range is determined by the `cuts` argument.*

The `"centroids"` style shown in the figure scales the hexagon sizes by cell count, and plots them at the center of mass determined by `xcenter` and `ycenter`. The `cuts=6` argument yields six different hexagon sizes. There are two nested plot styles (`nested.lattice` and `nested.centroids`, not shown) which provide depth when plotted on a color screen.

Examining Individual Bins

There are several large bins in the plot which we may want to examine more closely. The generic `identify` function can be used to interactively identify points on a hexagonal bin plot. The two largest bins can be identified as follows.

```
> quake.par <- plot(quakes.bin, style = "centroids",
+ cuts = 6)
> oldpar <- par(quake.par)
> identify(quakes.bin, use.pars = quake.par, offset = 1)

[1] 114 79

> par(oldpar)
```

First it is necessary to save the graphical parameters used to plot the hexagonal bin. After entering the `identify` command, use the cross-hairs to locate the point of interest on the graphics screen, and click the left mouse button. The count in the closest cell will appear on the graphics screen. We have used the optional argument `offset` to make the count easier to read. When you have identified both points, click the center or right mouse button, while keeping your pointer within the graphics window. The index of the points you have identified will appear on your command line, as above. Then use the `par` function to reset the graphics parameters.

Directional Rays

The `rayplot` function can be used to display the magnitudes of a variable of interest at spatial locations using directional rays. For smaller data sets, these rays or other types of symbols can be plotted at each data location. However, when the number of sites is large, the magnitudes and trends are easier to visualize if the locations are first binned using `hexbin`. The following example uses the `ozone` data set:

1. Create a `hexbin` object for the `ozone` data, using eight bins in the x direction.

   ```
   > ozone.bin <- hexbin(ozone.xy$x, ozone.xy$y,
   + xbins = 8)
   ```

2. Map each (x, y) pair in the original data to a hexagonal cell using the function `xy2cell`.

   ```
   > ozone.cells <- xy2cell(ozone.xy$x, ozone.xy$y,
   + xbins = 8)
   ```

3. Use the function `tapply` to calculate the median for each cell, and use these values as angles for the rayplot.

```
> ozone.angle <- tapply(ozone.median, ozone.cells,
+ median)
> library(maps)

Warning messages:
The functions and datasets in library section maps
are not supported by TIBCO Software Inc.

> map(region = c("new york", "new jersey", "conn",
+ "mass"), lty = 2)
> rayplot(ozone.bin$xcenter, ozone.bin$ycenter,
+ ozone.angle)
```

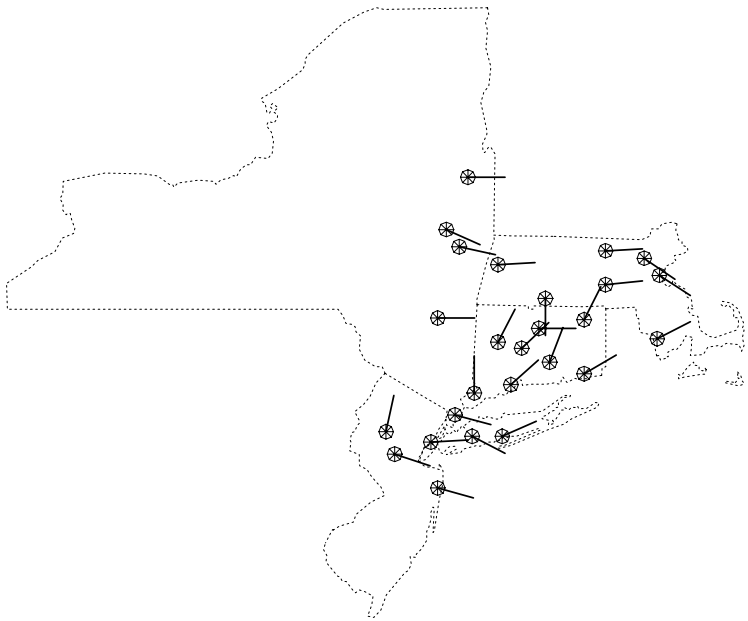

Figure 24.3: *Rayplots add direction as well as density. This plot shows median ozone emissions.*

The plot shows the median ozone emissions for the group of sites within each hexagonal bin. The ray is plotted at the center of each bin, and the medians are scaled so the rays follow an arc from -π/ 2 (lowest median) to π/ 2 (highest median). It appears that the highest

emissions for the time period covered are in Connecticut. Additional attributes can be used with `rayplot` to add confidence intervals and a second variable to the plot. Also, the lengths and widths of the rays and the size of the base octagon can be changed. See the online help file for more information on `rayplot`.

REFERENCES

Carr, D.B., Olsen, A.T., & White, D. (1992). Hexagon mosaic maps for display of univariate and bivariate geographical data. *Cartography and Geographical Information Systems* **19**:228-236.

ANALYZING TIME SERIES AND SIGNALS

25

INTRODUCTION

There are two general approaches to analyzing time series and signals. One is to use time domain methods in which the values of the process are used directly; the other is to use frequency domain methods. Frequency methods investigate the periodic properties of the process. The books by Chatfield (1984) and Shumway (1988) provide readable introductions to time series analysis, which covers both time domain and frequency domain methods.

Fields of study tend to focus on analyzing data in one domain or the other. For example, economists use the time domain extensively while electrical engineers often use the frequency domain. To a large extent, this division arises from the types of questions that are being asked of the data. However, combining the approaches can at times give a more thorough understanding of the data.

Robust methods are necessary for both domains because the failure of model assumptions (such as Gaussian errors) can cause misleading results when classical techniques are applied.

AUTOCORRELATION IN SERIES DATA

If data are collected over time, there may be correlation between successive observations; this is known as *autocorrelation* or *serial correlation*. In this section, we show how to use visual analysis (simple plots, lagged plots, and autocorrelation plots) and numerical analysis to look into autocorrelation.

You can visually explore autocorrelation in your data using Spotfire S+ functions to make three kinds of plots:

- *Simple time series and signal plots*, which you can read about in the chapter Time Series and Signal Basics in the *Programmer's Guide*.
- *Lagged scatter plots*, which are scatter plots of pairs of values (y_t, y_{t+m}) of a time series separated by a *lag* of one or more time units.
- *Autocorrelation function plots*, which provide an estimate of the correlation between observations separated by a lag of zero, one, or more time units.

To illustrate the use of these functions, we use the function `rnorm` to create *uncorrelated* normal random numbers. From these numbers, we create a *correlated* series `x.cor`:

```
> r.norm <- rnorm(100)
> x.cor <- signalSeries(data = r.norm[1:98] +
+ r.norm[2:99] + r.norm[3:100])
```

The series `x.cor` is serially correlated at lags 1 and 2; that is, `x.cor[i]` is correlated with `x.cor[i+1]` and `x.cor[i+2]`. But `x.cor` is serially uncorrelated at lags greater than 2; that is, `x.cor[i]` and `x.cor[i+k]` are uncorrelated for $k > 2$.

Basic Time Series Plots

The basic time series plot shows each observation plotted against its observation time. For example, our series `x.cor` can be plotted as follows, using both lines and plotting symbols:

```
> plot(x.cor, plot.args = list(type = "b", pch = 16))
```

This expression yields the plot of Figure 25.1.

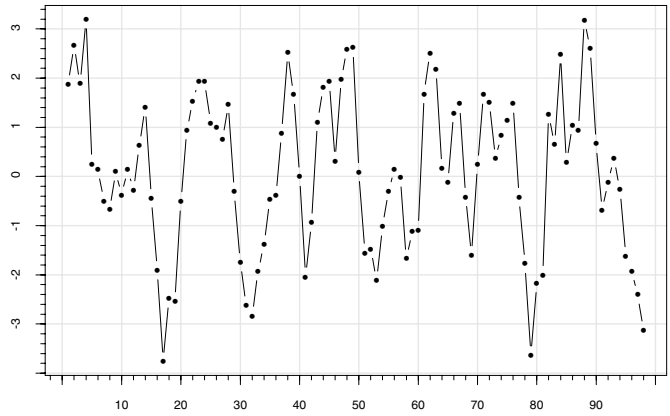

Figure 25.1: *Time series plot for a correlated series.*

The values of successive observations tend to be close together, so you suspect some serial correlation. You can see this more clearly with `lag.plot` and `acf`, as described in the following sections.

Lagged Scatter Plots

The lagged scatter plots in Figure 25.2 consist of scatter plots of pairs of values (y_t, y_{t+m}) of a time series separated by m time units for $m = 1, 2, \ldots, M$. The figure is generated with the following expression:

```
> lag.plot(x.cor, lags = 4,layout = c(2,2))
```

The maximum lag M is specified by the `lags` argument to `lag.plot`. For the above example, the choice `lags=4` results in $M = 4$, and so there are four plots. The `layout` argument specifies the way the M scatter plots are arranged in a single figure, just as you use the function `par` to specify multiple figure layout.

A circular shape for a lagged scatter plot at a specific lag m indicates that there is little correlation at that lag. On the other hand, an elliptical shape for a lag m scatter plot in the 45 degree direction indicates positive correlation at lag m. An elliptical shape in the 135 degree direction indicates negative correlation. In the above example

using `x.cor`, the lag 1 plot shows clear evidence of positive correlation, and the lag 2 plot shows some indication of positive correlation.

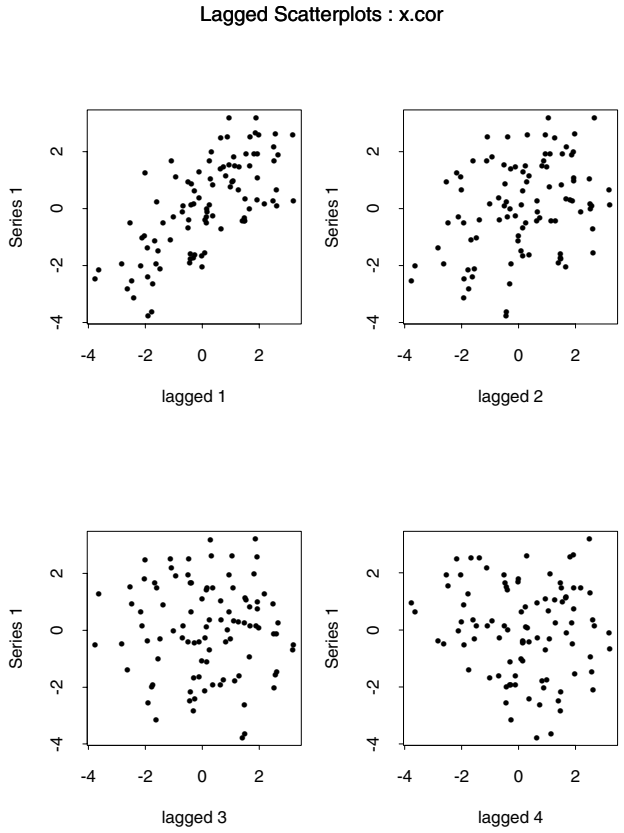

Figure 25.2: *Lagged scatter plots for a correlated series.*

Autocorrelation Function in Univariate Series

The autocovariance and autocorrelation functions are important tools for describing the serial (or temporal) dependence structure of a univariate time series. Let x_t be a stationary time series with mean μ

and variance σ_x^2, and assume for ease of notation that t takes on integer values $t = 0, \pm 1, \pm 2, \ldots$ The autocovariance function of x_t at lag k is defined as

$$\gamma(k) = E(x_t - \mu)(x_{t+k} - \mu). \tag{25.1}$$

Since x_t is stationary, this does not depend on t. The autocorrelation function at lag k is defined as

$$\rho(k) = \frac{\gamma(k)}{\gamma(0)} = \frac{\gamma(k)}{\sigma_x^2}, \tag{25.2}$$

and is simply a standardized version of the autocovariance function. Both the autocovariance function and the autocorrelation function are even functions; that is, $\gamma(k) = \gamma(-k)$ and $\rho(k) = \rho(-k)$. In addition, the autocorrelation function satisfies

$$|\rho(k)| \leq 1 \qquad \text{for all } k = 0, \pm 1, \pm 2, \ldots \tag{25.3}$$

Example: white noise.

A stationary time series for which x_t and x_{t+k} are uncorrelated is called *white noise*, and satisfies $\gamma(k) = E(x_t - \mu)(x_{t+k} - \mu) = 0$ for all integers $k \neq 0$. Such a process is sometimes loosely termed a "purely random process." Since $\gamma(0) = \sigma_x^2$, a white noise process has the autocovariance function

$$\gamma(k) = \begin{cases} \sigma_x^2 & k = 0 \\ 0 & k \neq 0 \end{cases} \tag{25.4}$$

The autocorrelation function is

$$\rho(k) = \begin{cases} 1 & k = 0 \\ 0 & k \neq 0 \end{cases} \tag{25.5}$$

Example: moving average process.

A moving average process of order q, denoted MA(q), is defined by the equation

$$x_t = \mu + \beta_0 \varepsilon_t + \beta_1 \varepsilon_{t-1} + \cdots + \beta_q \varepsilon_{t-q}, \tag{25.6}$$

where ε_t is a white noise process. It is easy to show that the autocovariance function for this process is given by

$$\gamma(k) = \begin{cases} \sum_{t=0}^{q-|k|} \beta_t \beta_{t+|k|} & |k| \leq q \\ 0 & |k| > q \end{cases} \tag{25.7}$$

The autocorrelation function is given by

$$\rho(\tau) = \begin{cases} \sum_{t=0}^{q-|\tau|} \beta_t \beta_{t+|\tau|} & |\tau| \leq q \\ 0 & |\tau| > q \end{cases} . \tag{25.8}$$

The *autocovariance* function estimate at lag k is:

$$\hat{\gamma}(k) = \frac{1}{n}\sum_{t=1}^{n-k}(x_t - \bar{x})(x_{t+k} - \bar{x}), \tag{25.9}$$

where

$$\bar{x} = \frac{1}{n}\sum_{t=1}^{n} x_t$$

is the mean of the series and n is the length of the observed series. Notice that the divisor n is used, even though there are only $n-k$ terms. As a result, $\hat{\gamma}(k)$ is a biased estimate, even if $\bar{x}$ is replaced by the true mean μ. However, $\hat{\gamma}(k)$ has a few properties that make up for a small amount of bias. In particular, use of the divisor n ensures positive semi-definiteness of the function $\hat{\gamma}(k)$, and the mean squared error of this estimate is often smaller than that obtained when n^{-1} is replaced by $(n-k)^{-1}$. See Priestley (1981) for details.

The *autocorrelation* function estimate at lag k is

$$\hat{\rho}(k) = \frac{\hat{\gamma}(k)}{\hat{\gamma}(0)}. \tag{25.10}$$

Autocorrelation Function in Multivariate Series

The autocovariance and autocorrelation functions for multivariate series are defined analogously to those of univariate series. In addition, one is interested in *crosscovariance* and *crosscorrelation* functions. Suppose that x_t is an m-variate stationary time series and

$x_{it} = (x_t)_i$ is the ith time series, for $i = 1, \ldots, m$. In addition, suppose that the ith series has a mean value of $\mu_i = Ex_{it}$. The covariance function matrix for $x_t = (x_{1t}, \ldots, x_{mt})$ at lag k is defined as

$$\Gamma(k) = E(\boldsymbol{x}_t - \mu)(\boldsymbol{x}_{t+k} - \mu)^T, \tag{25.11}$$

where T denotes the transpose and $\mu = (\mu_1, \mu_2, \ldots, \mu_m)$. The covariance matrix $\Gamma(k)$ is $m \times m$ and has the property that $\Gamma^T(k) = \Gamma(-k)$. The ith main diagonal element of $\Gamma(k)$ is the *autocovariance* function

$$\gamma_{ii}(k) = E(x_{it} - \mu_i)(x_{i(t+k)} - \mu_i) \tag{25.12}$$

for the ith time series x_{it}. The ijth off-diagonal element of $\Gamma(k)$ is the *crosscovariance function*

$$\gamma_{ij}(k) = E(x_{it} - \mu_i)(x_{j(t+k)} - \mu_j) \tag{25.13}$$

for the ith and jth series x_{it} and x_{jt}, where $i, j = 1, \ldots, m$ and $i \neq j$. Note that a crosscovariance function $\gamma_{ij}(k)$, $i \neq j$ is not generally symmetric in k; that is, $\gamma_{ij}(k) \neq \gamma_{ij}(-k)$ in general. The estimate of either an autocovariance or crosscovariance at lag k is given by

$$\hat{\gamma}_{ij}(k) = \frac{1}{n} \sum_{t=1}^{n-k} (x_{it} - \bar{x}_i)(x_{j(t+k)} - \bar{x}_j), \tag{25.14}$$

where

$$\bar{x}_i = \frac{1}{n}\sum_{t=1}^{n} x_{it}.$$

Note that for $i = j$, the autocovariance estimate $\hat{\gamma}_{ij}(k)$ in Equation (25.14) has the same form as Equation (25.9). The autocorrelation and crosscorrelation estimates at lag k are

$$\rho_{ij}(k) = \frac{\hat{\gamma}_{ij}(k)}{\sqrt{\hat{\gamma}_{ii}(0)\hat{\gamma}_{jj}(0)}} \tag{25.15}$$

Partial Autocorrelation

Another useful diagnostic tool for the analysis of the serial dependence is the partial autocorrelation function. Background on this function is deferred to the next section, after introducing autoregressive processes.

Simple Use of Autocorrelation Function

The function `acf` can be used to compute the sample autocovariance, autocorrelation, or partial correlation functions for a specified number k of lags.

To compute an estimate of the autocorrelation function $\gamma(k)$ for lags $k = 0, 1, \ldots, M$ of the `x.cor` series, we can use the command:

```
> x.acr <- acf(x.cor, plot=F)
```

To generate a plot of $\gamma(k)$ along with a plot of the time series, we can use the following commands:

```
> par(mfrow=c(2,1))
> plot(x.cor)
> acf.plot(x.acr)
```

The result is shown in Figure 25.3. The autocorrelation estimate at each lag is given by the height of the vertical lines in the `acf` plot. You can specify the number of lags M for which you want autocorrelations by using the optional argument `lag.max` to `acf`.

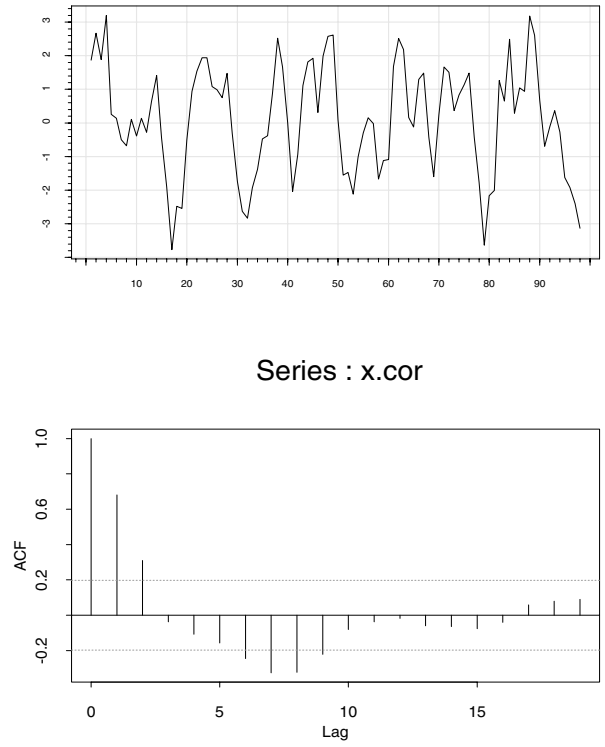

Figure 25.3: *Time series plot and ACF plot for a correlated series.*

The value of the autocorrelation function at lag 0 is always 1. The horizontal band about zero represents the approximate 95% confidence limits for H_0:ρ = 0. If no autocorrelation estimate falls outside the strip defined by the two dotted lines, and the data contain no outliers, you may safely assume that there is no serial correlation. Otherwise, you should be concerned about the presence of serial correlation. In our example, the `acf` plot indicates serial correlation at lags 1 and 2.

The function `acf.plot` can be used to plot the results from `acf`. This function takes the list returned by `acf` and uses its components to calculate approximate limits and decide appropriate labeling for the plot. For more details, see the help file for `acf.plot`.

AUTOREGRESSION METHODS

Univariate Autoregression

Consider a time series x_t that satisfies the difference equation (recursion)

$$x_t = \alpha_1 x_{t-1} + \alpha_2 x_{t-2} + \cdots + \alpha_p x_{t-p} + \varepsilon_t \ , \tag{25.16}$$

where ε_t is a white noise process with zero mean and finite variance σ_ε^2. The time series x_t is called an autoregressive process of order p and is denoted AR(p). The x_t in Equation (25.16) has zero mean, a fact which can be easily verified. An AR(p) process with nonzero mean μ is generated by the equation

$$x_t - \mu = \alpha_1 (x_{t-1} - \mu) + \cdots + \alpha_p (x_{t-p} - \mu) + \varepsilon_t \ . \tag{25.17}$$

It is worth noting that an AR(p) process is a pth-order Markov process.

Not all values of the autoregression coefficients $\alpha_1, \ldots, \alpha_p$ result in a stationary process. In particular, in an AR(1) process

$$x_t = \alpha x_{t-1} + \varepsilon_t \tag{25.18}$$

it is fairly easy to show that $|\alpha| < 1$ is the condition for stationarity. For $\alpha = 1$, the AR(1) process becomes a discrete time random walk, which is known to be *nonstationary*.

For an AR(p) process, the condition for stationarity is that the (complex) roots of

$$\phi(z) = 1 - \alpha_1 z - \alpha_2 z^2 - \cdots - \alpha_p z^p \tag{25.19}$$

lie outside the unit circle. An interpretation of AR(2) models from a physical point of view is given by Priestley (1981).

Autoregressive models have a wide range of uses in statistics, including forecasting and autoregression-type spectral density function estimation. Autoregressive modeling is also widely applicable in engineering, where it is referred to as linear prediction modeling. For example, speech analysis and recognition systems rely on autoregressive models. For many applications, autoregression provides good linear approximations, which have the virtue of extreme simplicity. In particular, the equations used to estimate the unknown coefficients $\alpha_1, \ldots, \alpha_p$ are linear, as we point out below. Of course one should be careful not to insist on using an autoregression model where, for example, a moving average component is needed, nonstationarity must be dealt with, or a nonlinear model is required. When in doubt, consult an experienced statistician with a time series background.

The Yule-Walker Equations

Let $\gamma(k)$ be the autocovariance function for the AR(p) process x_t. It can be shown that the AR(p) coefficients $\alpha_1, \ldots, \alpha_p$ satisfy the Yule-Walker equations

$$\sum_{k=1}^{p} \gamma(k-i)\,\alpha_k = \gamma(i) \qquad i = 1, 2, \ldots, p \tag{25.20}$$

In addition, one can show that

$$\sigma_x^2 = \sum_{k=1}^{p} \gamma(k)\,\alpha_k + \sigma_\varepsilon^2 . \tag{25.21}$$

Given that the AR(p) coefficients satisfy the Yule-Walker equations in (25.20), there is a very natural way to obtain estimates $\alpha_1, \alpha_2, \ldots, \alpha_p$ based on a finite sample $x_1, x_2, \ldots, x_n$ of the time series. Namely, replace the $\gamma(k)$ in Equation (25.20) by the estimates

$$\hat{\gamma}(k) = \frac{1}{n} \sum_{t=1}^{n-|k|} (x_t - \bar{x})(x_{t+|k|} - \bar{x}), \tag{25.22}$$

where

$$\bar{x} = \sum_{t=1}^{n} x_t \tag{25.23}$$

and solve the resulting equations for $\hat{\alpha}_1, \ldots, \hat{\alpha}_p$. Since $\hat{\gamma}(-k) = \hat{\gamma}(k)$, we can write the equations as

$$\begin{aligned}
\hat{\gamma}(1) &= \alpha_1\hat{\gamma}(0) + \alpha_2\hat{\gamma}(1) + \alpha_3\hat{\gamma}(2) + \cdots + \alpha_p\hat{\gamma}(p-1) \\
\hat{\gamma}(2) &= \alpha_1\hat{\gamma}(1) + \alpha_2\hat{\gamma}(0) + \alpha_3\hat{\gamma}(1) + \cdots + \alpha_p\hat{\gamma}(p-2) \\
\hat{\gamma}(3) &= \alpha_1\hat{\gamma}(2) + \alpha_2\hat{\gamma}(1) + \alpha_3\hat{\gamma}(0) + \cdots + \alpha_p\hat{\gamma}(p-3) \\
&\cdots \\
\hat{\gamma}(p) &= \alpha_1\hat{\gamma}(p-1) + \alpha_2\hat{\gamma}(p-2) + \alpha_3\hat{\gamma}(p-3) + \cdots + \alpha_p\hat{\gamma}(0)
\end{aligned} \tag{25.24}$$

We call these equations the sample-based Yule-Walker equations. Once the $\hat{\alpha}_j$'s are obtained by solving Equation (25.24), we can use them along with the $\gamma(k)$ in Equation (25.21),

$$\hat{\gamma}(0) = \alpha_1\hat{\gamma}(1) + \alpha_2\hat{\gamma}(2) + \cdots + \alpha_p\hat{\gamma}(p) + \hat{\sigma}_\varepsilon^2 \qquad (25.25)$$

to solve for σ_ε^2.

In practice, the order of the autoregression is not known, and it is often desirable to compare solutions of various orders. Hence, we wish to solve Equation (25.24) for a variety of values of p from 1 through p_{max}, where p_{max} is sometimes 10, 15, or even larger.

The Levinson-Durbin Recursion

The matrix of coefficients in Equation (25.24) is a Toeplitz matrix; that is, the elements on each diagonal are all the same. Because of this property, there is a recursive method that allows you to obtain estimates for a kth-order model from the estimates of the $k-1$ model in a fast and accurate manner. The method is referred to as the Levinson or Levinson-Durbin algorithm. Let $a_{i,\,k}$ denote the estimate of the ith autoregression coefficient α_i in an AR(k) model. If we have the estimates $a_{1,\,k}, \ldots, a_{k-1,\,k-1}$, and the estimated error variance σ_{k-1}^2 assuming an AR($k-1$) model, then estimates for an AR(k) model are

$$a_{k,\,k} = \frac{\hat{\gamma}(k) - \sum_{j=1}^{k-1} a_{j,\,k-1}\hat{\gamma}(j-k)}{\sigma_{k-1}^2}, \qquad (25.26)$$

where

$$a_{j,\,k} = a_{j,\,k-1} - a_{k,\,k}a_{k-j,\,k-1} \quad \text{for } 1 \le j \le k-1 \qquad (25.27)$$

and

$$\sigma_k^2 = \sigma_{k-1}^2 (1 - a_{k,k}^2) \tag{25.28}$$

From Equation (25.28), we see that the squares of the $a_{k,k}$ can be interpreted as a measure of the usefulness of increasing the order of the AR process from $k-1$ to k. The $a_{k,k}$ sequence is called the *partial autocorrelation function* or "reflection coefficients," depending on the field of study. This sequence is useful in diagnosing whether the series is in fact an AR process. If the process is an AR(p), then all $a_{k,k}$ should be close to zero for $k > p$. A common approximation for the standard error of the $a_{k,k}$ for $k > p$ is $1/\sqrt{n}$. See Box and Jenkins (1976).

AIC Order Selection

A way of selecting the order of the AR process is to find an order that balances the reduction of estimated error variance with the number of parameters being fit. One such measure is Akaike's Information Criterion (AIC). For an order k model, this criterion can be written as

$$\mathrm{AIC}(k) = n \log(\sigma_{\varepsilon,k}^2) + 2k \tag{25.29}$$

If the series is an AR process, then the value of k that minimizes $\mathrm{AIC}(k)$ is an estimate of the order of the autoregression.

Multivariate Autoregression

If the scalar quantities x_t, ε_t, and μ in Equation (25.17) are replaced by m-dimensional vectors $\boldsymbol{x}_t$, ε_t, and μ, and the scalars α_t are replaced by $m \times m$ matrices A_t, we obtain the multivariate pth-order autoregression

$$x_t - \mu = \boldsymbol{A}_1(\boldsymbol{x}_{t-1} - \mu) + \cdots + \boldsymbol{A}_p(\boldsymbol{x}_{t-p} - \mu) + \varepsilon_t \ . \tag{25.30}$$

Here, ε_t is an m-dimensional white noise series with a mean of zero and a covariance matrix Q. This covariance matrix is sometimes loosely referred to as the "prediction variance."

The vector autoregression $\boldsymbol{x}_t$ satisfies a vector analogue of the Yule-Walker equations in (25.20). Namely, with $\Gamma(i) = \text{cov}\{\boldsymbol{x}_t, \boldsymbol{x}_{t+i}\}$ we have

$$\sum_{k=1}^{p} \Gamma(k-i)\boldsymbol{A}_k = \Gamma(i), \qquad i = 1, 2, \ldots, p \tag{25.31}$$

We also have the vector autoregression analogue of Equation (25.21):

$$\Gamma(0) = \sum_{k=1}^{p} \Gamma(k)\boldsymbol{A}_k + \boldsymbol{Q} \tag{25.32}$$

Sample Yule-Walker equations for this vector case are obtained by replacing $\gamma(k)$ in Equation (25.22) by

$$\hat{\Gamma}(k) = \frac{1}{n} \sum_{t=1}^{n-|k|} (\boldsymbol{x}_t - \bar{\boldsymbol{x}})(\boldsymbol{x}_{t+|k|} - \bar{\boldsymbol{x}})^T, \tag{25.33}$$

where

$$\bar{\boldsymbol{x}} = \frac{1}{n} \sum_{t=1}^{n} \boldsymbol{x}_t. \tag{25.34}$$

The equations

$$\sum_{i=1}^{p} \hat{\Gamma}(k-i)\hat{\boldsymbol{A}}_k = \hat{\Gamma}(i), \quad i = 1, 2, \ldots, p \tag{25.35}$$

are then solved for the estimates $\hat{A}_k$, for $k = 1, \ldots, p$. The multivariate version of Equation (25.25) is therefore

$$\hat{\Gamma}(0) = \sum_{k=1}^{p} \hat{\Gamma}(k)\hat{\boldsymbol{A}}_k + \hat{\boldsymbol{Q}} \tag{25.36}$$

which may be solved for $\hat{Q}$.

There is also an analogue of the Levinson-Durbin algorithm (Equations (25.26) to (25.28)), which may be used to obtain estimates $\hat{\boldsymbol{A}}_{i,\,k}$, $i = 1, \ldots, k$, and $\hat{\boldsymbol{Q}}_k$ for a kth-order vector autoregression, given estimates $\hat{\boldsymbol{A}}_{i,\,k}$, $i = 1, \ldots, k-1$, and $\hat{\boldsymbol{Q}}_{k-1}$ for an order $k-1$ vector autoregression. This method is referred to as *Whittle's recursion.*

Autoregression Estimation via Yule-Walker Equations

The Spotfire S+ function `ar.yw` fits autoregressive models to multivariate time series using Whittle's extension to the Levinson-Durbin recursion. We can use it to fit an autoregression model to a short piece of the `say.wavelet` speech data set as follows:

```
> sp <- say.wavelet[2501:2600,]
> sp.ar <- ar.yw(sp)
> sp.ar$order.max

  [1] 20

> sp.ar$order

  [1] 6

> acf.plot(sp.ar)
```

```
> plot(sp.ar$aic, type = "l",
+ main = "Akaike Information Criteria for sp")
```

The result of the `acf.plot` command is shown in Figure 25.4; the output from the `plot` command is shown in Figure 25.5.

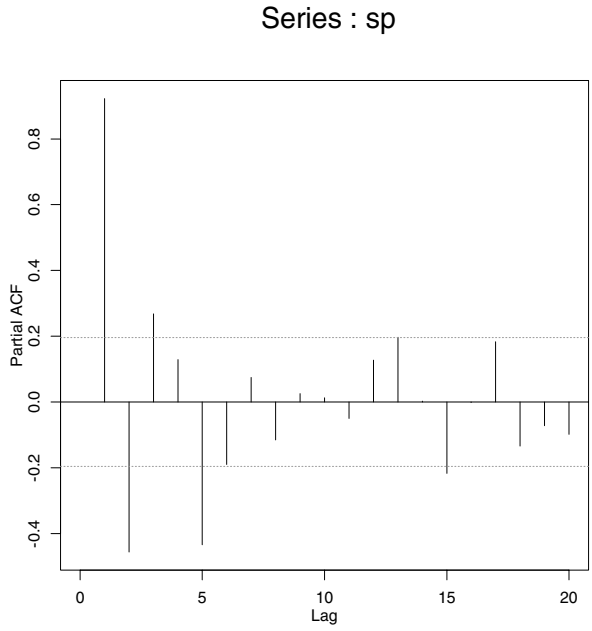

Figure 25.4: *Autocorrelation plot for speech data.*

The maximum order fit defaults to 20 in this case, and the AIC picks a model of order 6. Figure 25.4 shows the minimum AIC at 7; this plot starts indexing at 1, but the first element of the `sp.ar$aic` component is for order 0.

We can plot the residuals of the `sp.ar` model with the following command:

```
> plot(sp.ar$resid,
+ main = "Residuals after fitting an AR(6) to sp")
```

The result is shown in Figure 25.6.

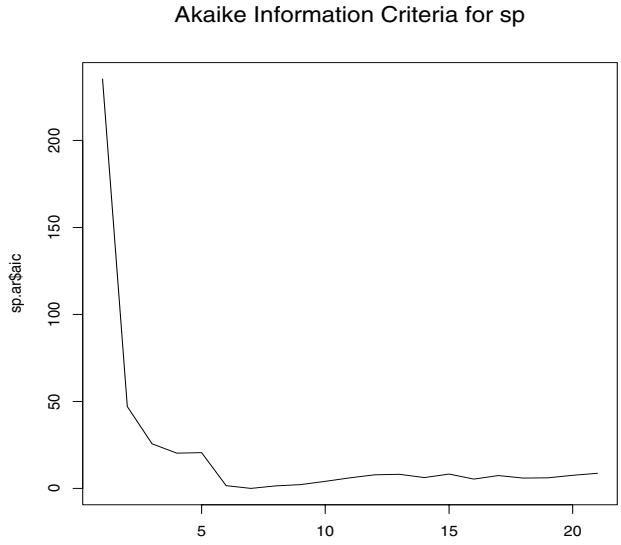

Figure 25.5: *AIC for the speech data.*

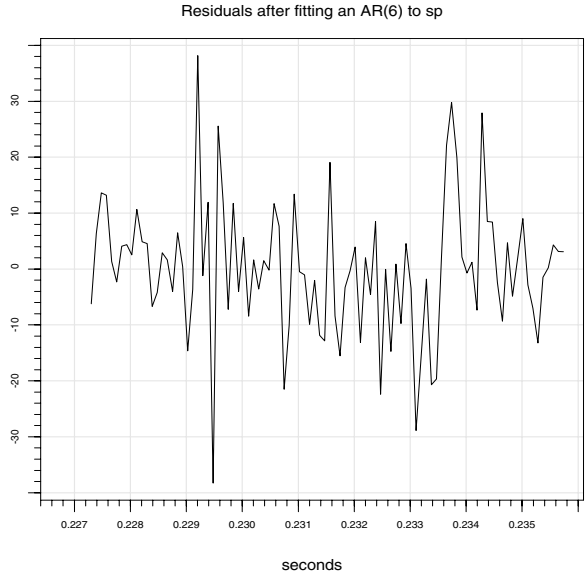

Figure 25.6: *Residuals for the speech data.*

Autoregression Estimation with Burg's Algorithm

This section presents Burg's algorithm, an alternative to using Yule-Walker equations for fitting autoregressive models. Burg's approach is based on estimating the kth partial correlation coefficient by minimizing the sum of forward and backward prediction errors:

$$SS(a_{k,\,k}) = \sum_{t=k+1}^{n} \{[x_t - a_{1,\,k}x_{t-1} - \cdots - a_{k,\,k}x_{t-k}]^2 + [x_{t-k} - a_{1,\,k}x_{t-k+1} - \cdots - a_{k,\,k}x_t]^2\} \tag{25.37}$$

Given all of the coefficients for the order $k-1$ model, Equation (25.37) is a function only of $a_{k,\,k}$. The function essentially measures how well the order k model predicts forwards and backwards. The algorithm is optimal in the sense of maximizing a measure of entropy. See Burg (1967). The following Spotfire S+ commands fit an AR(2) model using Burg's algorithm on the same piece of the `say.wavelet` data set that we used in the previous section.

```
> sp.arb <- ar.burg(sp, F, 2)
> sp.ar <- ar(sp, aic = F, order.max = 2)
> sp.arb$ar

, , 1
           [,1]
[1,]  1.3642945
[2,] -0.4733473

> sp.ar$ar

, , 1
           [,1]
[1,]  1.3430803
[2,] -0.4559053
```

Finding the Roots of a Polynomial Equation

The function `polyroot` finds the zeroes of the complex-valued polynomial equation:

$$a_k z^k + \cdots + a_1 z + a_0 = 0$$

Use this function to find the roots of an autoregression or moving average operator with user-specified coefficients. For example, if you have estimated pth-order autoregressive coefficients $\hat{\phi}_1, \hat{\phi}_2, \ldots, \hat{\phi}_p$, then the autoregression polynomial is $1 - \hat{\phi}_1 z - \hat{\phi}_2 z^2 - \ldots - \hat{\phi}_p z^p$. In this case, you would choose $a = (a_0, a_1, \ldots, a_k)$ with $k = p$, $a_0 = 1$, and $a_i = -\hat{\phi}_i$ for $i = 1, \ldots, p$.

To solve the equation $z^2 + 5z + 6 = 0$ in Spotfire S+, we use the following command:

```
> polyroot(c(6,5,1))

  [1] -2+0i -3+0i
```

UNIVARIATE ARIMA MODELING

Spotfire S+ provides several functions for fitting autoregressive integrated moving-average (ARIMA) models to univariate time series data. ARIMA models are useful for a wide variety of problems including forecasting, quality control, seasonal adjustment, spectral estimation, and general summarization of data. Box and Jenkins (1976) give a comprehensive account of ARIMA modeling, and discussions of ARIMA models can be found in many recent standard textbooks for time series.

ARMA Models

A stationary autoregressive moving-average process is obtained by combining Equation (25.6) for an MA process and Equation (25.16) for an AR process. A zero mean ARMA(p, q) process x_t can be written in the form

$$x_t - \phi_1 x_{t-1} - \cdots - \phi_p x_{t-p} = \varepsilon_t - \theta_1 \varepsilon_{t-1} - \cdots - \theta_q \varepsilon_{t-q} \ . \qquad (25.38)$$

Here, ε_t is a white noise process; that is, the ε_t are uncorrelated with zero mean and variance σ^2. The process ε_t is sometimes called the *innovations* process. The parameters $\phi_1, \ldots, \phi_p$ are the autoregressive coefficients, and the parameters $\theta_1, \ldots, \theta_q$ are the moving-average coefficients.

If the innovations ε_t are Gaussian and uncorrelated, then they are also independent. This is a frequently used assumption.

The ARMA model of Equation (25.38) is often written in the form $\phi(B)x_t = \theta(B)\varepsilon_t$, where B is a *backshift* operator. That is, $B(x_t) = x_{t-1}$ and

$$\begin{aligned} \phi(B) &= 1 - \phi_1 B - \cdots - \phi_p B^p \\ \theta(B) &= 1 - \theta_1 B - \cdots - \theta_q B^q. \end{aligned} \qquad (25.39)$$

ARIMA Models

Many time series encountered in practice are *nonstationary*. For these series, simple ARMA models are typically inadequate. However, the *differenced* series may be stationary. Box and Jenkins (1976) developed a methodology for fitting ARMA models to differenced data. These are known as autoregressive integrated moving-average (ARIMA) models. An ARIMA$(p,\ d,\ q)$ process x_t is

$$\phi(B)\nabla^d x_t = \theta(B)\varepsilon_t \tag{25.40}$$

where ε_t, $\phi(B)$, and $\theta(B)$ are as defined in Equation (25.38), $\nabla = 1 - B$ is the first-difference operator and $\nabla^d = (1-B)^d$ is the d-fold differencing operator. When $d = 1$, the differenced series $w_t = \nabla x_t = x_t - x_{t-1}$ follows an ARMA$(p,\ q)$ process: $\phi(B)w_t = \theta(B)\varepsilon_t$. When $d = 2$, the twice differenced series w_t is an ARMA$(p,\ q)$ process:

$$w_t = \nabla^2 x_t = \nabla(x_t - x_{t-1}) = x_t - 2x_{t-1} + x_{t-2}$$

Seasonal Models

Time series data frequently exhibit seasonal cycles or periodicities. For example, data collected on a monthly basis may have a period of length $s = 12$ months, reflecting the *seasonal* behavior of the process. The framework for ARIMA models can be extended to handle periodicities as well (see Box and Jenkins (1976), Chapter 9). Seasonal behavior is modeled by using seasonal autoregressive moving-average processes and differencing operators. For a period of length s, these operators are of the form

$$\begin{aligned} \Phi(B^s) &= 1 - \Phi_1 B^s - \cdots - \Phi_P B^{sP} \\ \Theta(B^s) &= 1 - \Theta_1 B^s - \cdots - \Theta_Q B^{sQ} \\ \nabla_s^D &= (1 - B^s)^D \end{aligned} \tag{25.41}$$

The parameters $\Phi_1, \ldots, \Phi_p$ are the seasonal autogressive coefficients and the parameters $\Theta_1, \ldots, \Theta_Q$ are the seasonal moving average coefficients. The operator ∇_s^D is the seasonal d-fold differencing operator.

Typically, $\Phi(B^s)$, $\Theta(B^s)$, and ∇_s^D are combined with the ordinary operators $\phi(B)$, $\theta(B)$, and ∇^d in a multiplicative fashion. The multiplicative seasonal ARIMA$(p, d, q) \times (P, D, Q)_s$ process can be represented by

$$\Phi(B^s)\phi(B)\nabla_s^D \nabla^d x_t = \Theta(B^s)\theta(B)\varepsilon_t . \qquad (25.42)$$

In general, Spotfire S+ allows for any number of multiplicative operators with arbitrary periods. However, Equation (25.42) should be sufficiently general for most problems.

ARIMA Models with Regression Variables

In addition to using past values to model a series, it is often desirable to use explanatory or regression variables. The regression variables may simply be a constant (intercept) term, a deterministic function of time, dummy variables to model outliers, or lagged values of another time series.

Let z_t be a vector of m elements. An ARIMA process y_t with known regression variables is defined by

$$y_t = z_t'\beta + x_t, \qquad (25.43)$$

where β is an unknown parameter vector and x_t is an ARIMA process. For example, setting $z_t' = (1,t)$ results in a straight line regression model component $z_t'\beta = \beta_1 + \beta_2 t$ with slope β_2 and intercept β_1.

Identifying and Fitting ARIMA Models

Box and Jenkins (1976) give the following paradigm for fitting ARIMA models.

1. *Model identification:* Determination of the ARIMA model orders (p, d, q) and (P, D, Q).
2. *Estimation of model parameters:* The unknown parameters in Equations (25.42) and (25.43) are estimated.
3. *Diagnostics and model criticism:* The residuals are used to validate the model and suggest potential alternative models which may be better.

These steps are repeated until a satisfactory model is found.

Model Identification

Initial model identification is done using the autocorrelation and partial autocorrelation functions. These can be computed using the Spotfire S+ function `acf`. See Chapter 6 of Box and Jenkins (1976) for a complete discussion on the identification of ARIMA models.

An alternative procedure for selecting the model order is use of a penalized log-likelihood measure. One such measure is Akaike's Information Criterion (AIC). For autoregressive models, AIC is given by Equation (25.29). For general ARIMA models, AIC is defined below in Equation (25.46).

Estimation of Model Parameters

ARMA models

The log-likelihood for an ARMA model, Equation (25.40), can be computed using the *prediction error decomposition* (see Harvey (1981)). Consider an ARMA process x_t as in Equation (25.38), and assume the innovations ε_t are independent Gaussian random variables. Let

$$\hat{x}_t^{t-1} = E(x_t | x_1, \ldots, x_{t-1}, \phi_1, \ldots, \phi_p, \theta_1, \ldots, \theta_q)$$

denote the conditional mean one-step-ahead prediction of x_t based on the data $x_1, x_2, \ldots, x_{t-1}$, and let

$$r^2 f_t = \mathrm{var}(x_1, \ldots, x_{t-1}, \phi_1, \ldots, \phi_p, \theta_1, \ldots, \theta_q) \qquad (25.44)$$

denote the conditional variance of $\hat{x}_t^{t-1}$. The parameter σ^2 is the variance of the innovations process ε_t. Defining the *prediction errors* by $e_t = x_t - \hat{x}_t$ and letting $L = L(x_1, \ldots, x_n)$ denote the likelihood, one can show that

$$-2\log L(x_1, \ldots, x_n) = n\log(2\pi\sigma^2) + \sum_{t=1}^{n} f_t + \frac{1}{\sigma^2} + \sum_{t=1}^{n} e_t^2 / f_t \qquad (25.45)$$

Fitting an ARMA(p, q) model by Gaussian maximum likelihood involves finding the estimates $\hat{\phi}_1, \ldots, \hat{\phi}_p$ and $\hat{\theta}_1, \ldots, \hat{\theta}_{1q}$ that yield a minimum in Equation (25.45). The parameters $\phi_1, \ldots, \phi_p$ and $\theta_1, \ldots, \theta_q$ enter into Equation (25.45) through Equation (25.44). The estimate of σ^2 is $\sum_{t=1}^{n} e_t^2 / f_t$, which can be concentrated out of the likelihood. The likelihood is, in general, nonlinear in $\phi_1, \ldots, \phi_p$ and $\theta_1, \ldots, \theta_q$ and so a nonlinear optimizer must be used.

The likelihood for an ARMA model (Equation (25.43)) with regression variables can be computed in a similar fashion. In this case, replace x_t's by y_t's in Equation (25.45). The regression coefficients can be concentrated out of the likelihood (see Kohn and Ansley, (1985)).

A so-called conditional log-likelihood approximation to Equation (25.45) can be obtained by conditioning on the first p values of the series, where p is the order of the autoregressive operator. This conditional log-likelihood function is given by

$$\begin{aligned} -2\log L(x_{p+1}, \ldots, x_n | x_1, \ldots, x_p) &= (n-p)\log(2\pi\sigma^2) \\ &+ \sum_{t=p+1}^{n} \log f_t + \frac{1}{\sigma^2} \sum_{t=p+1}^{n} e_t^2 / f_t \end{aligned} \qquad (25.46)$$

Bell and Hillmer (1987) give several arguments in favor of Equation (25.46). The main advantage of using the conditional log-likelihood approximation is that the AR parameters $\phi_1, \ldots, \phi_p$ can be concentrated out of the likelihood, reducing the computational complexity of the nonlinear optimization. Usually, little information is lost in using Equation (25.46) instead of Equation (25.45).

The prediction errors e_t and their variances f_t can be computed in a number of ways. Ansley (1979) gives an efficient algorithm based on the Choleski decomposition of the covariance of the process x_t. However, if missing values are present, this algorithm no longer applies. Alternative algorithms are based on applying the Kalman filter to a state space representation of an ARMA process. See Jones (1980), Harvey (1981), and Kohn and Ansley (1986) for various methods based on the Kalman filter approach. All of these methods handle missing values, although the Kohn and Ansley approach is the most general.

Multiplicative ARIMA models

Estimating multiplicative ARIMA models by Gaussian maximum likelihood is a straightforward extension to estimating ARMA models. With no missing values present, the likelihood for a nonstationary series is obtained by differencing the data and computing the likelihood for the differenced process.

With missing values present, the likelihood can be computed using the Kalman filter: see Kohn and Ansley (1986) and Bell and Hillmer (1987). The simplest approach is to condition on the first $p^* + d^*$ observations, where p^* and d^* are the orders of the expanded autoregressive and differencing operators obtained by multiplying the regular and seasonal AR and the regular and seasonal difference operators in Equation (25.42). Specifically, $p^* = p + sP$ is the order of the polynomials $\Phi(B^s)\phi(B)$, and $d^* = d + sD$ is the order of $\nabla_s^D \nabla^d$. This gives the general ARIMA analog to the ARMA log-likelihood in Equation (25.46), and is equivalent to the differencing approach in the case of no missing values.

Missing values in the beginning of the series

If a missing value occurs in the first $p^* + d^*$ observations, then conditioning on the first $p^* + d^*$ observations is not possible. In this case, the series can be reversed and the likelihood function is computed for the reversed series. The likelihood is invariant to reversing the order of the data. If there are missing values at both the beginning and the end of the series, then the exact likelihood must be computed using a modification of the Kalman filter, derived by Kohn and Ansley (1986). However, an approximate likelihood can be obtained by including a dummy regression variable for each missing value, and then replacing the missing value by an arbitrary number (see Bruce and Martin (1989)). The dummy regression variable is zero at all time points except for the time of the missing value.

Starting values for the optimizer

The likelihood is maximized using a general quasi-Newton optimizer (see the `nlmin` help file for a discussion of the optimizer). It is necessary to provide starting values for the ARIMA parameters. Poor starting values can lead to slow convergence to the maximum, or even worse, convergence to a local maximum. To avoid this, it is advisable to use a stepwise fitting procedure, starting with relatively simple ARIMA models and adding one coefficient at a time. Several tuning constants can be adjusted to provide better performance, but the default values in `nlmin` are usually sufficient.

Transformation to ensure stationarity and invertibility

The ARIMA coefficients can be transformed to ensure stationarity and invertibility of the model (see Jones, (1980)). If the solution lies on the boundary of stationarity or invertibility, then the optimizer may take many steps to converge. For this reason it may be desirable *not* to constrain the model to be invertible.

Warning

If printed output from the optimizer is requested, the printed coefficients are the transformed coefficients and not the original ARIMA coefficients.

AIC and model selection

One method of model selection is based on Akaike's information criterion (AIC). The best model is given by the model with the lowest AIC value. AIC is a penalized version of the log-likelihood function in Equation (25.46), and is defined by

$$AIC = -2\log L(x_{m+1}, \ldots, x_n | x_1, \ldots, x_m) + 2r, \qquad (25.47)$$

where r is the total number of parameters estimated. Specifically, r is the number of AR, MA, and regression coefficients. For example, $r = 2$ in an ARIMA$(1, 1, 1)$ model.

When comparing the AIC values for different models, it is important to *condition* the likelihood on the same number of observations. In other words, m should be the same in Equation (25.47) for all models. This allows one to use AIC to compare models with different numbers of AR or differencing coefficients.

Computational notes

The Spotfire S+ function `arima.mle` fits ARIMA models to univariate time series data through Gaussian maximum likelihood. The conditional form of the likelihood (Equation (25.46)) is used.

The regression parameters are concentrated out of the likelihood, as in Kohn and Ansley (1985). With no missing data, an algorithm similar to that of Ansley (1979) is used to compute the likelihood. With missing data, the Kalman filter is used with the state space representation of Kohn and Ansley (1986). However, missing values are not permitted in the beginning of the series; see the above discussion on missing values.

By default, the moving average parameters are transformed to ensure invertibility. However, if the solution lies on the boundary of invertibility, better performance by the optimizer can be obtained by *not* transforming the parameters. In certain circumstances, it might be useful to fit models in which lower order AR or MA parameters are constrained to be zero. In this case, the coefficients cannot be transformed to ensure stationarity or invertibility.

Examples of simple use

Simulate an MA(2) series and fit it using a Gaussian maximum likelihood.

```
> ma <- arima.sim(100, model = list(ma = c(-0.5,-0.25)))
> ma.fit <- arima.mle(ma, model = list(ma = c(-0.5,-0.25)))
```

Fit a Box-Jenkins (0,1,1) x (0,1,1) Airline model to the `ship` data. Use zeroes as the starting values for the optimizer.

```
> model <- list(list(order = c(0,1,1)), list(order =
+ c(0,1,1), period = 12))
> fit <- arima.mle(ship, model = model)
```

Diagnostics and Model Criticism

The third stage in fitting ARIMA models consists of validating the model through examination of the one-step prediction residuals e_t. See Chapter 8 of Box and Jenkins (1976) for a more complete discussion of ARIMA model diagnostics. The single most important diagnostic is a plot of the *standardized residuals* $\tilde{e}_t \equiv e_t / \sqrt{f_t}$ over time. If the correct ARIMA model is fit and the data are Gaussian, then $\tilde{e}_t$ should behave approximately like a Gaussian white noise process with zero mean and a variance of 1. Problems to look for in the plot of $\tilde{e}_t$ include outliers, nonhomogeneity of variance, and obvious structure in time.

Another basic diagnostic technique is to examine the autocorrelation function of the residuals e_t. Let $\hat{\gamma}_k$ be the autocorrelations of the residuals e_t. If the model is adequate, then the $\hat{\gamma}_k$ should be uncorrelated Gaussian random variables with zero mean and a variance of n^{-1}. Hence, the presence of large autocorrelations indicates that a model may be inadequate, and the nature of the $\hat{\gamma}_k$ may suggest how to improve it. However, some caution should be exercised in the use of $\hat{\gamma}_k$ to evaluate the model. For example, the variance times n^{-1} can be a serious overestimate of the true variance for small lags, which underestimates the significance for lack of fit.

In addition to examining the $\hat{\gamma}_k$ individually, it is useful to base a diagnostic on the autocorrelations as a whole. Define the *portmanteau* test statistic Q by

$$Q = n \sum_{k=1}^{K} \hat{\gamma}_k^2,$$

where K is a fixed maximum number of lags and n is the number of observations used to compute the likelihood. Typically, K should be between 10 and 20. If the correct ARIMA model is fit and the data are Gaussian, then Q is approximately distributed as a chi-square random variable on $K - r$ degrees of freedom, where r is the number of parameters in the model.

The Spotfire S+ function `arima.diag` computes the three diagnostics discussed in this section for ARIMA models fit to univariate time series.

Examples of simple use

Compute diagnostics for simulated AR(1) series.

```
> x <- arima.sim(model = list(ar = 0.9))
> fit <- arima.mle(x, model = list(ar = 0.9))
> diag <- arima.diag(fit)
```

By default, the argument `plot = T` in `arima.diag`, and the diagnostics are plotted using the function `arima.diag.plot`.

Forecasting Using ARIMA Models

An important application of ARIMA models is to forecast beyond the end of a series. Assuming that the model order and parameters are known, the forecast means and confidence intervals are easily produced using the Kalman filter (see Harvey (1981)). Typically, one would first fit an ARIMA model using the techniques described in the section Model Identification. The resulting model can then be used to produce forecasts for the series.

The Spotfire S+ function `arima.forecast` produces forecasts given an ARIMA model for a univariate time series.

Predicted and Filtered Values for ARIMA Models

The Spotfire S+ function `arima.filt` produces one-step predicted values and their variances f_t, as defined in Equation (25.44). The primary application of `arima.filt` is for use in other Spotfire S+ functions; it computes the residuals in `arima.diag`, and it computes the forecasts in `arima.forecast`.

If autoregressive or differencing operators are present in the model, then predicted values are not produced for the first $p^* + d^*$ time points, where p^* and d^* are the orders of the expanded autoregressive and differencing polynomials.

Computational Note

The function `arima.filt` also returns filtered values and their variances. Let y_t be a process which behaves according to a signal plus noise model:

$$y_t = x_t + v_t,$$

where x_t is the signal and v_t is the noise. A common problem is to extract the signal by filtering the observed process y_t. The filtered values and their variances are $E(x_t|y_1, \ldots, y_t)$ and $var(x_t|y_1, \ldots, y_t)$.

For a pure signal, v_t is 0 for all t and the filtered values are simply the observations themselves. The current version of Spotfire S+ does not support signal plus noise models. Hence, the filtered values are the same as the input series. However, the filtered values are returned for compatibility with future releases.

Simulating ARIMA Processes

The Spotfire S+ function `arima.sim` generates a simulated ARIMA process of the form in Equations (25.42) or (25.43), given an ARIMA model structure, regression variables and a vector of innovations or a random generator. The innovations vector corresponds to ε_t in Equation (25.40), and can be input directly. Alternatively, a random generator may be supplied, and the innovations are generated accordingly.

For stationary ARMA processes, the series can be initialized by generating an initial random state vector according to a state space form of the model. The initial state vector is computed by transforming a white noise vector using the Choleski decomposition of the unconditional covariance matrix of the state vector.

For nonstationary ARIMA processes, the unconditional covariance matrix of the state vector doesn't exist. Hence, the simulated series is initialized by assuming that the initial state vector is zero. This is equivalent to assuming past innovations and simulated values are zero. To avoid the effects of the initialization, a series longer than the one needed is generated, and the simulated series is taken from the end of the generated series.

Examples of simple use

Simulate an ARMA(1,1):

```
> x <- arima.sim(model = list(ar = 0.5, ma = -0.6), n = 100)
```

Simulate an ARIMA(0,1,1) with contaminated innovations:

```
> rand.gen <- function(n) ifelse(runif(n) > 0.9, rnorm(n),
+ rcauchy(n))
> x.wild <- arima.sim(100, model = list(ndiff = 1,
+ ma = 0.6), start.innov = 50, rand.gen = rand.gen)
```

Modeling Effects of Trading Days

In many monthly economic time series, the data are affected by the number of trading days in that month. For example, if a given month has more weekdays and fewer weekends than other months, then one might expect a higher level of economic activity during that month. One approach to handling the trading day effect is to include regression variables reflecting the number of Mondays, Tuesdays, etc., in each month or quarter.

The function `arima.td` returns a multivariate time series that is suitable for use as a regression variable. The first column gives the number of days in the month or quarter. The following six columns give the number of Saturdays, Sundays, Mondays, Tuesdays, Wednesdays, and Thursdays minus the number of Fridays in the month or quarter. See Hillmer, Bell, and Tiao (1983) for use of trading day variables in ARIMA modeling of time series data.

Using the holiday functions in Spotfire S+, you can modify `arima.td` to take into account the number of holidays. See the section Calculating Holiday Dates in the chapter Dates, Times, Time Intervals, and Sequences in the *Programmer's Guide*.

Examples of simple use

```
> td.ship <- arima.td(ship)
> mle.td <- arima.mle(ship, model = list(order = c(0,1,1)),
+ xreg = td.ship)
```

LONG MEMORY TIME SERIES MODELING

Long memory is a common feature of time series in a wide variety of areas. It is hard to detect, but has enormous effects on statistical quantities such as standard errors and tests, and hence on the conclusions drawn. One major application has been to time series of wind speeds (Haslett and Raftery, (1989)), where long memory means that there is a tendency to observe not just windy weeks and months, but windy years and decades, and presumably also windy centuries and millennia; we often say that there is variation at all temporal scales.

Long memory time series have autocorrelations that decay slowly as lag increases. Typically, the autocorrelations tend to zero hyperbolically; that is, $\rho(k) \sim k^{-\alpha}$, with $\alpha > 0$ so that the sum of the autocorrelations is infinite, $\sum_{k=0}^{\infty} \rho(k) = \infty$. Thus, the autocorrelations between observations far away from one another in time are small, but not negligible. The spectrum of a long memory time series goes to infinity as the frequency goes to zero at the rate $f(\omega) \sim \omega^{-(1-\alpha)}$.

One important property is that the variance of the sample mean declines at a slower rate than the usual $O(n^{-1})$. If $\rho(k) \sim k^{-\alpha}$, then $var(\bar{X}) = O(n^{-\alpha})$. (Note that a long memory time series is stationary only if $0 < \alpha \leq 1$.) This can have huge consequences. In the wind speed example, α is estimated to be 0.34. This implies that, for estimating the mean wind speed at a given location, twenty years of actual data are worth only about the same as one month of independent daily observations.

The ARMA models (with no differencing) discussed in the section Autoregression Methods and the section Univariate ARIMA Modeling are, by contrast, short memory models. For ARMA models, the autocorrelations decay exponentially, the sum of the autocorrelations is finite, the spectrum is finite at zero, and the variance of the sample mean is the usual $O(n^{-1})$. Fitting a short memory ARMA model to data can give very misleading results if the long memory property holds, even if the fitted model matches the

lower-lag autocorrelations well. In the wind speed example, a short memory ARMA model underestimates the variance of the sample mean by a factor of more than ten in many cases.

The long memory property in time series is discussed by Mandelbrot (1977), who calls it the *Joseph effect* because of the sequence of seven years of plenty followed by seven lean years, as recounted in the Book of Genesis story of Joseph. Mandelbrot pointed out that long memory time series tend to be asymptotically approximately self-similar and hence to be, at least approximately, equivalent to fractals.

Fractionally Differenced ARIMA Modeling

Fractionally differenced ARIMA models

The fractionally differenced ARIMA$(p,\ d,\ q)$ model has been found to represent long memory time series quite well. It is defined by Equation (25.40), namely

$$\phi(B)\nabla^d x_t = \theta(B)\varepsilon_t.$$

However, d may take any value in the interval $[0,\ 1]$ instead of being restricted to either 0 or 1, and $\nabla^d = (1-B)^d$ is defined by the binomial expansion $(1-B)^d = \sum_{j=0}^{\infty} C(d,\ j)(-1)^j B^j$, where $C(d,\ j)$ are the binomial coefficients. When the series has a nonzero mean μ, the model is better written as

$$x_t = \mu + \nabla^{-d}\phi(B)^{-1}\theta(B)\varepsilon_t \qquad (25.48)$$

For the model given in Equation (25.48), $\rho(k) = k^{-(1-2d)}$, so that $\alpha = 1-2d$, where α is defined in the section Long Memory Time Series Modeling. This model is stationary only for $0 < d \le 1/2$ and reduces to the usual short memory ARMA$(p,\ q)$ model when $d = 0$.

Estimation of model parameters

The log-likelihood for the fractionally differenced ARIMA$(p,\ d,\ q)$ model of Equation (25.48) can be computed exactly using the prediction error decomposition given by Equation (25.45). In this

calculation, $\hat{x}_t^{t-1}$ and f_t are given by Equations 4.3 and 4.4 of Haslett and Raftery (1989). A major practical problem with maximum likelihood estimation based on this likelihood is that the required CPU time is $O(n^2)$. This can be enormous for the long series that are typical of application areas in which long memory is known to arise often. For example, in the wind speed data set, $n = 6574$.

We therefore use an approximation described in section 4.3 of Haslett and Raftery (1989) that essentially uses asymptotic values to approximate the dependence of x_t on x_{t-j} for $j > M$. This reduces the order of the required CPU time from $O(n^2)$ to $O(n)$. In practice, the approximation is extremely accurate, and for the wind speed data, it reduces the actual computer time by a factor of 70. We have found $M = 100$ to be a good choice; the exact maximum likelihood estimator can be recovered by setting $M = n$.

Computational notes

The Spotfire S+ function `arima.fracdiff` estimates the parameters of the fractionally differenced ARIMA(p, d, q) model. It returns exact or approximate maximum likelihood values, standard errors, covariance and correlation matrices of the parameter estimates, and the log-likelihood. The degree of approximation is determined by M; we recommend $M = 100$. The exact maximum likelihood estimator can be found by setting $M = n$, but if the series is long it can require significant CPU time. The log-likelihood is useful for comparing models and choosing the number of AR and MA parameters. An approximate test of the long memory property can be carried out by dividing the estimate of d by its standard error and comparing the result with a standard normal distribution.

Simulating Fractionally Differenced ARIMA Processes

The Spotfire S+ function `arima.fracdiff.sim` generates a simulated fractionally differenced ARIMA(p, d, q) series of the form in Equation (25.48), given the values of d, the AR and MA parameters, and the mean μ. This function uses the prediction error decomposition to generate x_t from its conditional distribution, given all of the aforementioned values.

Examples of simple use

Simulate a fractionally differenced ARIMA(2,.33,0).

```
> x.sim <- arima.fracdiff.sim(model = list(d=0.33,
+ ar=c(0.01, -0.06), mu=3.1))
> arima.fracdiff(x.sim, model = list(ar=rep(NA,2)))

$model:
$model$ar:
[1]  0.01145420 -0.06152254

$model$ma:
[1] 0

$model$d:
[1] 0.3189504


$var.coef:
                 d            ar1            ar2
  d   1.959256e-04 -1.914622e-04 -9.319428e-05
ar1 -1.914622e-04  2.867117e-04  8.999233e-05
ar2 -9.319428e-05  8.999233e-05  1.439553e-04

$loglik:
[1] -14162.39

$h:
[1] 0.0001492355

$d.tol:
[1] 0.0001220703

$M:
[1] 100

$hess:
             d          ar1          ar2
  d -17064.33  -9863.2099  -4881.2697
ar1  -9863.21 -10040.2167   -108.7296
ar2  -4881.27   -108.7296 -10038.6852

$call:
arima.fracdiff(x = x.sim, model = list(ar = rep(NA, 2)))
```

SPECTRAL ANALYSIS

Let x_t be a stationary time series with sampling interval Δt. A major theorem for time series states that any series with zero mean $\mu = Ex_t = 0$ and finite variance $\sigma = varx_t$ may be well approximated by a truncated Fourier series:

$$x_t \approx \sum_{i=1}^{J} A_j \cos(2\pi f_j t) + B_j \sin(2\pi f_j t), \tag{25.49}$$

where A_j and B_j are random Fourier (series) coefficients, the f_j are well-chosen frequencies, and J is sufficiently large. This approximation of x_t as a Fourier series may be re-expressed in complex exponential form

$$x_t \approx \sum_{j=-J}^{J} C_j e^{i2\pi f_j t}, \tag{25.50}$$

where the C_j are *complex random* Fourier coefficients that have zero mean, $EC_j = 0$. The C_j are also uncorrelated:

$$cov(C_j, C_k) = EC_j\bar{C}_k = 0 \quad \text{for } j \neq k \tag{25.51}$$

The notation $\bar{a}$ denotes the complex conjugate of a.

Sometimes the set of real coefficients, A_j and B_j, or the set of complex coefficients C_j, are referred to as the (discrete time) Fourier transform of x_t.

Time series with a nonzero mean may be approximated by adding the mean μ to the right hand side of Equation (25.50):

$$x_t \approx \mu + \sum_{j=1}^{J} C_j e^{i2\pi f_j t} \tag{25.52}$$

The exact version of the approximation in Equation (25.50) is an integral known as the *spectral representation* of x_t. The *spectrum* or *spectral density* $S(f)$ for the series x_t can be described in terms of the coefficients C_j defined in Equation (25.50):

$$S(f_j) = E|C_j|^2 \tag{25.53}$$

Thus, the value of the spectrum at frequency f_j is the second moment of the random amplitude at frequency f_j. The spectrum $S(f)$ at an arbitrary frequency f can also be expressed exactly in terms of the autocovariance sequence

$$R(l) = EX_t X_{t+l}, \qquad l = 0, \pm 1, \pm 2, \ldots \tag{25.54}$$

Namely, $S(f)$ has the exact Fourier series representation

$$S(f) = \sum_{l=-\infty}^{\infty} R(l) e^{-il2\pi f}. \tag{25.55}$$

The autocovariances are the Fourier coefficients of $S(f)$:

$$R(l) = \int_{-\frac{1}{2}}^{\frac{1}{2}} S(f) e^{il2\pi f} df \tag{25.56}$$

Again, we often refer to $S(f)$ as the (discrete time) Fourier transform of $R(l)$, and refer to $R(l)$ as the inverse Fourier transform of $S(f)$.

Estimating the Spectrum from the Periodogram

Suppose that we have a time series $x_1, \ldots, x_n$ observed at a sampling interval Δ. The spectrum of this series may be estimated from the periodogram by using the function `spec.pgram`. The steps involved in this computation are described below.

1. Detrending and de-meaning

The first step in estimating the spectrum is to ensure that the mean is zero for the time series. If it is thought that the original series may contain a linear trend, this is accomplished by subtracting a least squares regression line from the series; that is, replace x_t with $x_t - \hat{\gamma} - \hat{\beta}t$, where $\hat{\gamma} + \hat{\beta}t$ is the least squares linear fit to the data. If it is thought that there is no trend in the data, subtract the mean from the series; i.e., replace x_t with $x_t - \bar{x}$, where $\bar{x}$ is the sample mean of $x_1, \ldots, x_N$. By default, the `spec.pgram` function removes the least squares line.

2. Tapering

A data taper is often applied to a detrended or de-meaned series. A taper sequence w_t multiplies each value in a series by a number between 0 and 1. Tapering reduces leakage of power. See Bloomfield (1976) and Priestley (1981) for discussions of tapering. The `spec.pgram` function includes a default split cosine taper of ten percent on each end of the series. See the section Tapering for further details.

3. Padding

Padding consists of increasing the length of the series x_t from n to n' by adding $n' - n$ zero values $x_{n+1} = \ldots = x_{n'} = 0$. Padding may generally be ignored for the spectrum function. See discussions on the fast Fourier transform (FFT) in the references for explanation.

4. The periodogram

To avoid extra notation, let n be the length of the series with or without padding. Let $\Delta = 1/\ freq$ be the sampling interval, where $freq$ is the frequency sampling rate component of the `tspar` attribute. Following the above operations, an estimate of the power spectrum at discrete Fourier frequencies $f_k = k/\ \Delta n$ is found by forming the periodogram

$$\begin{aligned} '(f_k) &= \frac{\Delta}{n}\left|\sum_{t=1}^{n} \tilde{x}_t \exp(-2\pi i f_k t)\right|^2 \\ &= \frac{n}{4}(A_k^2 + B_k^2), \quad k = 0,\ 1,\ \ldots,\ n/\ 2 \end{aligned} \tag{25.57}$$

where $\tilde{x}_t = w_t(x_t - \hat{\gamma} - \hat{\beta}t)$ is the tapered, detrended series. Note that $\hat{\beta} = 0$ and $\hat{\gamma} = \bar{x}$ if only a mean was removed from the series. The discrete Fourier transform (DFT) sum in Equation (25.57) is computed using a mixed radix fast Fourier transform (FFT) algorithm.

5. Smoothing

The periodogram is smoothed to reduce variability in the spectrum estimate (the estimates in Equation (25.57) do not become less variable as the length of the series increases). However, smoothing also introduces bias in the estimates, and there is a trade-off between the variability of the estimates and the bias. A thorough analysis might include inspecting the periodogram with several levels of smoothing. The smoothing that is performed on the periodogram is a sequence of running averages. The user can specify lengths of

modified Daniell windows to be run sequentially over the periodogram in `spec.pgram`. The `spec.pgram` function yields the smoothed estimate $\tilde{S}(f_k)$ expressed in decibels $(10 \times \log 10\tilde{S}(f_k))$.

6. Degrees of freedom and bandwidth

The `spec.pgram` function also computes the degrees of freedom for a chi-square approximation of the spectral density estimate at each Fourier frequency. When there is no smoothing, tapering or padding, there are $n = 2$ degrees of freedom. The degrees of freedom n increases with the amount of smoothing.

Bandwidth is a measure of the amount of smoothing. The formula for bandwidth used by `spec.pgram` is

$$b_w = \frac{I}{\Delta n}\sum_{j=0}^{2k}\left[\left(\frac{1}{12} + (j-k)^2\right)a_j\right]^{1/2}, \qquad (25.58)$$

where a_j are the values of the smoothing filter for $j = 0, \ldots, 2k$, and $1/\Delta n$ is the interval between discrete Fourier frequencies. The `spec.pgram` function returns the smoothing filter in the `filter` component, which has an index starting at zero. See Bloomfield (1976) for details.

Readers with no interest in multivariate time series may skip to the section Example of simple use.

Cross-Spectra Coherency and Phase

The *cross-spectrum* $S_{xy}(f_j)$ between two time series x_t and y_t at frequency f_j is approximately $EC_{xj}\overline{C}_{yj}$, where the C_{xj} and C_{yj} are given by Equation (25.50) (the extra subscript, x or y, distinguishes coefficients for the two series). One can think of this quantity as the complex covariance between C_{xj} and C_{yj}. The *phase* of x_t and y_t at frequency f_j is the angle of the cross-spectrum $S_{xy}(f_j)$.

The *squared-coherency* $K(f_j)$ between x_t and y_t at frequency f_j is the squared modulus of the cross-spectrum at f_j, normalized by the product of the two spectral densities $S_x(f_j)$ and $S_y(f_j)$:

$$K(f_j) = \frac{|S_{xy}(f_j)|^2}{S_x(f_j)S_y(f_j)}.$$

In view of Equation (25.53), we have

$$K(f_j) \approx \frac{|EC_{xj}\overline{C}_{yj}|^2}{E|C_{xj}|^2 E|C_{yj}|^2} = |\mathrm{corr}(C_{xj}, C_{yj})|^2.$$

This provides the most natural interpretation of squared-coherency: it is the square of the correlation between the random coefficients C_{xj} and C_{yj} of the series x_t and y_t at frequency f_j.

Smoothing of the spectral estimates is mandatory for the estimation of coherency. If no smoothing is performed, the estimate is identically 1. See Priestley (1981). Similarly, the estimation of phase is basically meaningless unless smoothing is performed.

The `spec.pgram` function computes estimates of the squared-coherency and the phase for multivariate series. The output is in the form of matrices, where each column is identified with a particular pair of univariate components. If j is less than k, then the column associated with the pair (j, k) is $\frac{j + (k-1)(k-2)}{2}$.

Example of simple use

A spectral estimate of the square root of the `sunspots` data may be obtained with:

```
> srsun.sp <- spec.pgram(sqrt(sunspots),
+ spans = c(3, 5, 7, 9), detrend = F, demean = T)
> spec.plot(srsun.sp)
```

The `spec.pgram` command subtracts the mean from the series, but assumes that there is no trend. The spectrum is smoothed with a series of 4 running averages. By default, ten percent on each end of the series has been tapered with a split cosine bell. The length of the

series is automatically padded from 2739 to 2744. A plot of the spectrum is produced by the `spec.plot` function. The result is shown in Figure 25.7.

Another simple example of the `spec.pgram` function is:

```
> llynx <- log(lynx)
> ll.sp <- spec.pgram(llynx, taper = 0)
> spec.plot(ll.sp)
```

The result is shown in Figure 25.8.

The spectral estimate of the `lynx` series uses no tapering, and since it also uses no smoothing, it is the raw periodogram estimate. The data are detrended, allowing for the possibility of a linear trend in the data. Note that this is probably a poor spectral estimate for the dataset.

Below, we analyze monthly CO_2 concentrations at Mauna Loa, Hawaii from January 1958 to December 1975. A `ts.plot` of the data reveals a strong linear trend and obvious cyclic behavior. Not surprisingly, the cycles appear to be yearly. The analysis is shown in Figure 25.9.

```
> par(mfrow = c(3, 1)) # put three plots in the figure
> co.sp1 <- spec.pgram(co2, plot=T)
> co.sp2 <- spec.pgram(co2, spans=c(9,9), plot=T)
> co.sp3 <- spec.pgram(co2, spans=c(3,3,3), plot=T)
```

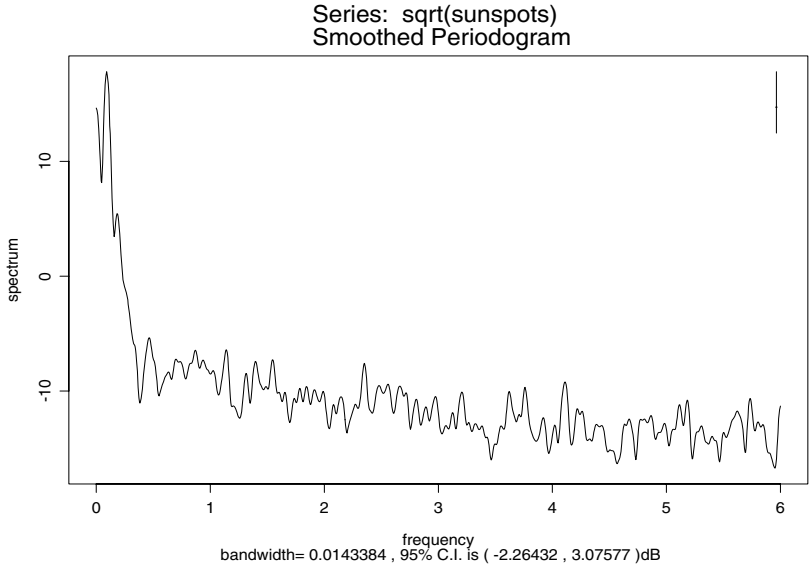

Figure 25.7: *Smoothed periodogram of the sunspot data.*

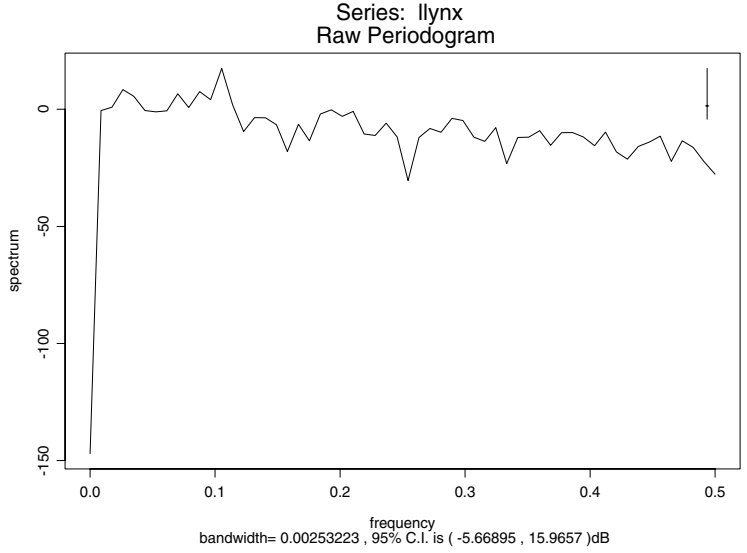

Figure 25.8: *Periodogram of the lynx data.*

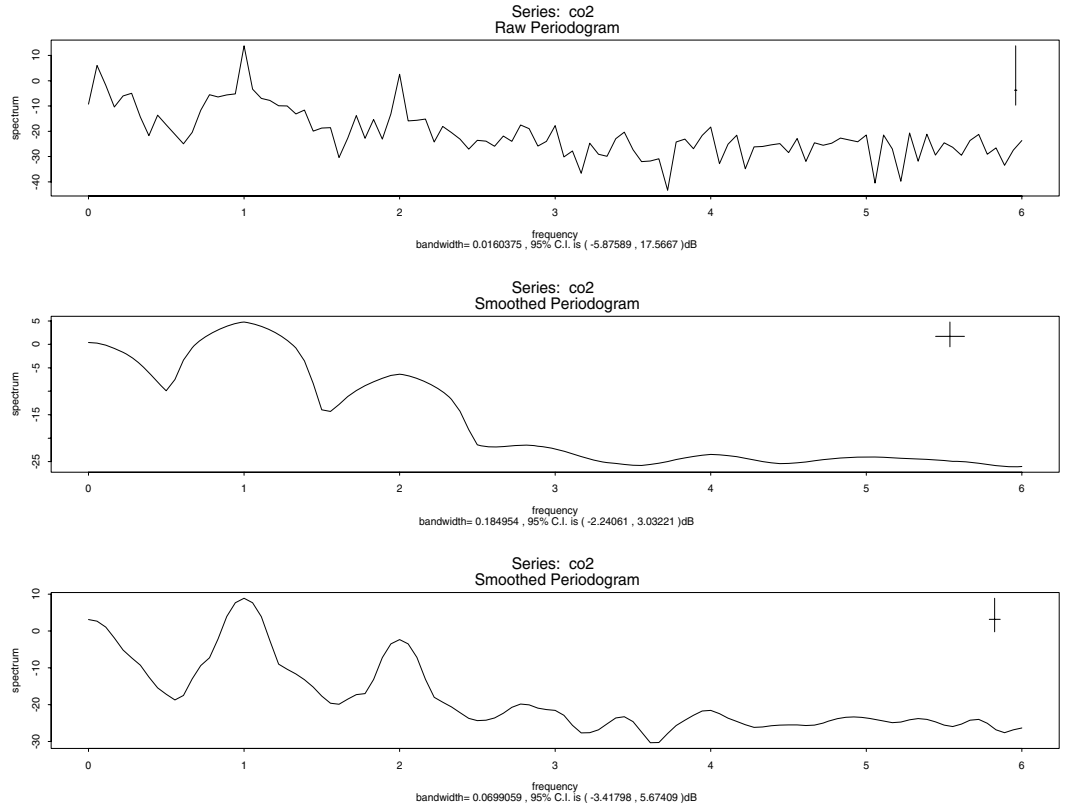

Figure 25.9: *Spectral estimates for the CO_2 data.*

Autoregressive Spectrum Estimation

An alternative spectral estimate to the smoothing of the periodogram is to compute an autoregressive (or some other) model. The spectrum of the estimated model can then be used as the spectral estimate of the smoothed periodogram.

The spectrum $S(f)$ of an autoregressive process with coefficients $\alpha_1, \ldots, \alpha_p$ is

$$S(f) = \frac{\sigma_\varepsilon^2}{\left|1 - \alpha_1 \exp(-2\pi i f) - \cdots - \alpha_p \exp(-2\pi i p f)\right|^2}, \qquad (25.59)$$

where f is the frequency in cycles per unit time, and σ_ε^2 is the variance of the innovation process ε_t.

Phase and coherency may also be estimated for multivariate series. The Spotfire S+ function `spec.ar` computes the autoregressive spectrum of a time series.

Examples of simple use

```
> lynx.ar <- ar(log(lynx))
> lynx.spar <- spec.ar(lynx.ar, plot = T)
```

The resulting plot is displayed in Figure 25.10. The `spectrum` function can be used in the same way as `spec.pgram` to allows for different types of spectrum estimates. The function `spec.plot` can be used to plot the output from any spectrum estimation function.

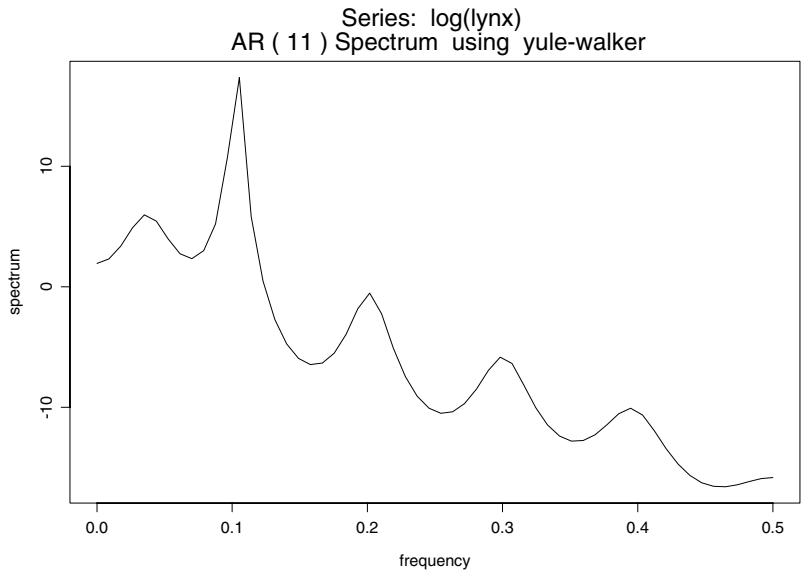

Figure 25.10: *Autoregression spectral estimate for the lynx data.*

Tapering

Tapering is a technique applied to time series to reduce the *leakage* phenomenon in spectral estimates. Leakage occurs when there is a large amplitude peak at a particular frequency f. The spectral estimates at frequencies near f can be higher than expected, and can easily obscure nearby lower amplitude peaks.

A *data taper* w_t, $0 \le w_t \le 1$, applied to a time series x_t produces a new tapered series.

$$\tilde{x}_t = w_t x_t \qquad t = 1, \ldots, n \tag{25.60}$$

Typically the values of w_t are close to zero at the ends, and close to one in the central part of the data.

The function `spec.taper` implements a split cosine bell taper. Let p be the portion to be tapered at each end of the series, and let n be the length of the series. For $m = np$ the split cosine bell taper is

$$w_t = \begin{cases} \frac{1}{2}[1 - \cos(\pi(t - 0.5)/m)] & t = 1, \ldots, m \\ 1 & t = m+1, \ldots, n-m \\ \frac{1}{2}[1 - \cos(\pi(n - t + 0.5)/m)] & t = n-m+1, \ldots, n \end{cases} \tag{25.61}$$

Examples of simple use

```
> lynx.taper <- spec.taper(lynx)
> lynx.taper.5 <- spec.taper(lynx, .05)
```

All the values in `lynx.taper` are smaller than the corresponding value in `lynx`. In `lynx.taper.5`, five percent of the values on each end are tapered.

LINEAR FILTERS

The most important and widely used type of *filter* is a linear time-invariant filter. With this kind of filter, the relationship between the input series x_t and the filtered output series is described by a constant coefficient linear difference equation. Linear time-invariant filters are often referred to as a *digital filters* by engineers. This class of filters has two primary types: convolution filters and recursive filters.

1. *Convolution filters* are usually called *finite-impulse response* (FIR) filters in the engineering literature, and *moving average* (MA) filters in the statistical literature.
2. *Recursive filters* are usually referred to as *infinite-impulse response* (IIR) filters in the engineering literature, and *autoregressive* (AR) filters in the statistical literature.

Convolution Filters

If x_t is the original series and $\mathbf{a} = (a_0, a_1, \ldots, a_q)$ is the set of filter coefficients, then the filtered series y_t is related to x_t by the convolution equation

$$y_t = \sum_{j=0}^{q} a_j x_{t-j} \qquad t = 0, \pm 1, \ldots \tag{25.62}$$

We note that the filter is "causal", in that each y_t is formed as a linear combination of present and past x_t, namely $x_t, x_{t-1}, \ldots, x_{t-q}$. If one is dealing with a spatial series rather than a time series, then the *noncausal* symmetric form of the convolution filter can be used:

$$y_t = \sum_{j=-q/2}^{q/2} a_j x_{t-j}, \tag{25.63}$$

In Equation (25.63), the filter coefficients are $a_{-q/2}, a_{-q/2+1}, \ldots, a_0, a_1, \ldots, a_{q/2}$ and q is an even integer. In this case, the a_j are usually symmetric; that is, $a_{-j} = a_j$ for $= 1, \ldots, q/2$. The noncausal symmetric form of the convolution filter can also be used when one is dealing with a time series in an "off-line" mode, as opposed to a real-time application, as is usually the case for Spotfire S+ users.

Recursive Filters

A recursive filter uses an autoregressive-type recursion to transform the series. If x_t is the original series and $a = (a_0, a_1, \ldots, a_q)$ are the coefficients, then the filtered series y_t is obtained by the recursion

$$y_t = \sum_{j=0}^{p} a_j y_{t-j-1} + x_t \qquad (25.64)$$

Examples of simple use

Here are two examples using convolution filters:

```
> flynx <- filter(log(lynx), rep(0.2,5))
> ts.plot(log(lynx), flynx)
> gaussfilt <- exp(-((-15:15)^2/7))
> gaussfilt <- gaussfilt/sum(gaussfilt)
> gflynx <- filter(log(lynx), gaussfilt)
> ts.plot(log(lynx), gflynx)
```

The resulting plots are shown in Figure 25.11 and Figure 25.12.

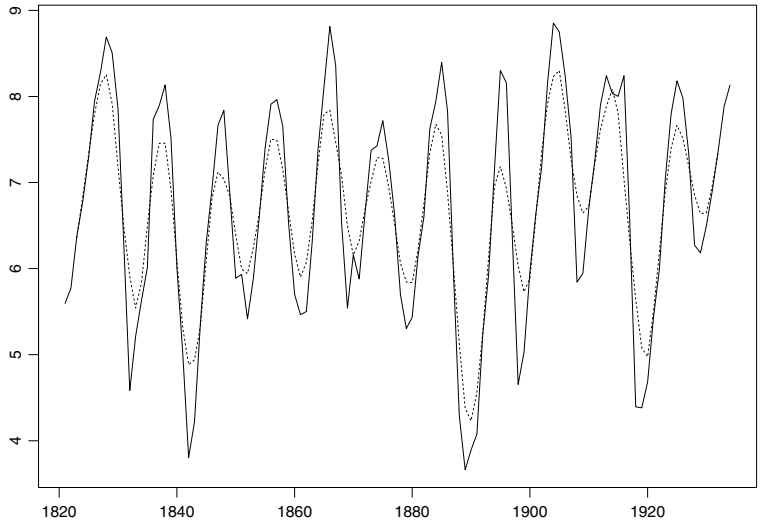

Figure 25.11: *Moving average of the lynx data.*

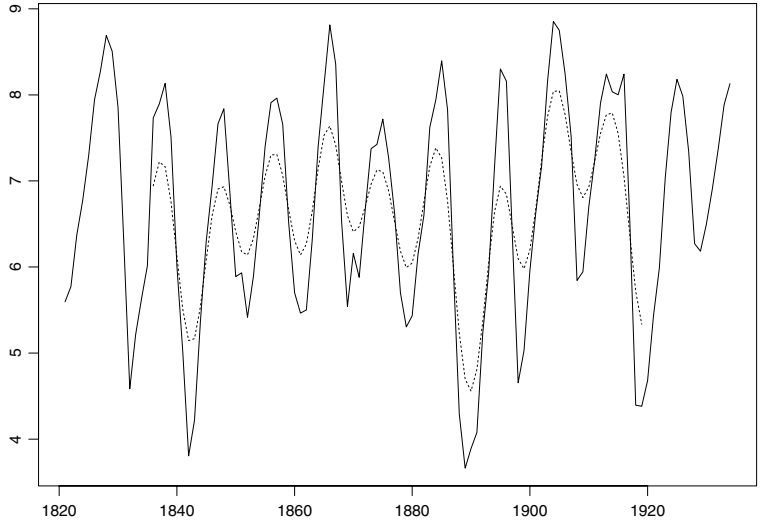

Figure 25.12: *Gaussian filtering of the lynx data.*

The `flynx` structure is a simple, equal-weight moving average of the logarithm of the `lynx` data, while `gflynx` is filtered with a Gaussian filter.

Here is an an example using a recursive filter:

```
> set.seed(14) # set the seed to reproduce this example
> ar.sim <- filter(rnorm(500), c(0.5, -0.3, 0.35), "r",
+ init = rnorm(3))
> ar.sim <- ar.sim[101:500]
> ts.plot(ar.sim, main = "AR(3) simulation")
```

This example is a simulation of an AR(3) process. The first part of the simulation is removed to more closely approximate a stationary process. The resulting plot is shown in Figure 25.13.

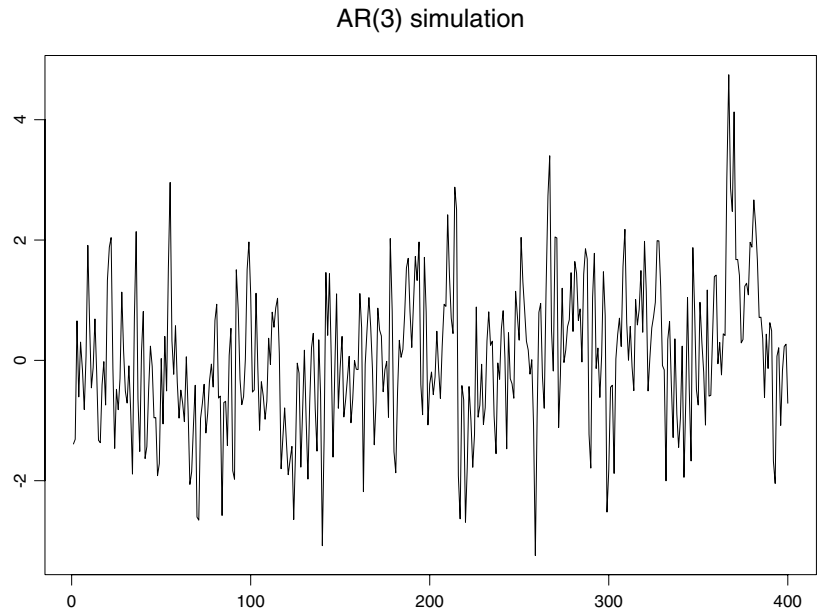

Figure 25.13: *Simulated autoregression.*

Complex Demodulation and Least Squares Low-Pass Filtering

Complex demodulation is a technique for analyzing a time series that does not assume stationarity. Inherent in the technique is the use of a low-pass filter. Hence, these two topics are presented together. The function `demod` can be used not only to perform complex demodulation of a time series, but also to generate a least squares low-pass filter with specific qualities.

Complex Demodulation

Suppose that a time series x_t satisfies

$$x_t = R_t \cos(\lambda t + \phi_t) + z_t, \tag{25.65}$$

where R_t and ϕ_t are smooth processes that vary slowly over time, and z_t is a process without a component at frequency λ. The R_t multiplier is the amplitude at time t of the periodic component with frequency λ, and ϕ_t is the phase of this periodic component at time t. Hence the model fits a series with an oscillation at some given frequency λ that changes slowly over time.

Equation (25.65) may be rewritten using complex numbers as:

$$x_t = \frac{1}{2} R_t [e^{i(\lambda t + \phi_t)} + e^{-i(\lambda t + \phi_t)}] + z_t \tag{25.66}$$

The series is then transformed into

$$y_t = x_t e^{-i\lambda t} = \frac{1}{2} R_t e^{i\phi_t} + \frac{1}{2} R_t e^{-i(2\lambda t + \phi_t)} + z_t e^{-i\lambda t} \tag{25.67}$$

A smooth component of y_t yields estimates of R_t and ϕ_t. The problem is to extract this component.

Least Squares Low-Pass Filtering

An ideal low-pass filter with cutoff frequency f_c has a transfer function

$$H(f) = \begin{cases} 1 & \text{if } f \leq f_c \\ 0 & \text{if } f > f_c \end{cases} \tag{25.68}$$

That is, all frequencies less than f_c are left unchanged, while no frequencies higher than f_c are allowed to pass through. Such an ideal filter does not exist, but it can be approximated arbitrarily well by using a sufficiently complex filter. A common approach is to design a fixed length filter using the least squares approximation method; the approximation improves as the filter length increases. See Bloomfield (1976) for details.

Examples of simple use

In the commands below, the `lynx` data are demodulated at the peak frequency of the raw periodogram. The phase and amplitude of the demodulation are plotted separately.

```
> lynx.sp <- spectrum(log(lynx))
> lynx.pk <- lynx.sp$freq[lynx.sp$spec ==
+ max(lynx.sp$spec)]
> lynx.dem <- demod(log(lynx), lynx.pk, .05, .10)
> ts.plot(lynx.dem$phase, xlab = "Time", ylab = "Phase")
> ts.plot(lynx.dem$amp, xlab = "Time", ylab = "Amplitude")
```

Figure 25.14 shows the phase estimate of demodulation of the `lynx` data, while Figure 25.15 shows the amplitude estimate of demodulation.

Figure 25.14: *Phase estimate in the demodulation of the lynx data.*

Figure 25.15: *Amplitude estimate in the demodulation of the lynx data.*

A method for obtaining a low-pass filter of length 50 with cutoff frequency 0.08 when the data are sampled at intervals of one time unit is shown below.

```
> filt50 <- demod(rnorm(200), 0.1, 0.08-1/49,
+ 0.08+1/49)$filter
```

ROBUST METHODS

Outliers in time series typically cause bias and an increase in the variability of conventional Gaussian maximum likelihood or least squares estimates. Unacceptably large biases can occur even in large sample size situations, when the fraction of outliers is not negligibly small. In particular, this problem occurs for both the Yule-Walker and Burg methods of fitting autoregressions.

As a simple example, consider the Yule-Walker estimate of the first-order autoregression parameter $\hat{\phi}$, which is also the lag 1 autocorrelation:

$$\hat{\phi} = \frac{\sum_{t=1}^{n-1} y_t y_{t-1}}{\sum_{t=1}^{n} y_t^2} .$$

Suppose that $y_{t_0} = \xi$ is an outlier for some given time t_0, where $|\xi|$ is large. Then $\hat{\phi}$ is small, and in fact, $\hat{\phi} \to 0$ as $|\xi| \to \infty$.

The robust procedures described in this section are designed to minimize the increased bias and variability due to outliers, whether they appear in isolation or in patches. We describe four functions for dealing with outliers: `ar.gm`, `acm.filt`, `acm.ave`, and `acm.smo`.

Typically, `ar.gm` and `acm.ave` are used in conjunction. The `ar.gm` function provides initial robust autoregression parameter estimates, which are then used by the robust "smoother" algorithm `acm.ave`. The function `acm.smo` is an alternative robust smoother, and both `acm.ave` and `acm.smo` use the robust filter `acm.filt` as a basic building block.

We elaborate on our setup and terminology. Consider the general replacement type outliers (or RO) model:

$$y_t = (1 - z_t)x_t + z_t w_t . \qquad (25.69)$$

In Equation (25.69), x_t is a pth-order autoregression, z_t is a 0-1 process with probability $1-\gamma$ of being 1, and w_t is a *contamination* process. Here, γ is the fraction of contamination. This general replacement model contains the so-called additive outliers (AO) model

$$y_t = x_t + v_t \tag{25.70}$$

as the special case where $w_t = x_t + \tilde{v}_t$ with $v_t = 0$ when $z_t = 0$, and $v_t = \tilde{v}_t$ when $z_t = 1$. Although the methods described in this section work for the general RO model, it is sometimes convenient for purposes of discussion to use the AO model. In doing so, we think in terms of the v_t having a contaminated normal distribution

$$F_v = (1-\gamma)N(0, \sigma_0^2) + \gamma H$$

where H is an arbitrary outlier-generating distribution. The $N(0, \sigma_0^2)$ term is the "nominal" Gaussian distribution of the additive noise v_t. In the context of Equations (25.69) or (25.70), a *filter* $\hat{x}_t = \hat{x}_t(y_1, \ldots, y_t)$ is an estimate of the unobservable "signal" x_t that depends on the present and past observations $y_1, \ldots, y_t$ at time t. A *smoother* $\hat{x}_t = \hat{x}_t(y_1, \ldots, y_n)$ is an estimate of x_t that depends on all the observations $y_1, \ldots, y_n$ for each time $t = 1, \ldots, n$. This is common terminology in the engineering literature. Both filters and smoothers often perform a *smoothing* operation, in the sense that they are weighted linear combinations of $y_1, \ldots, y_t$ and $y_1, \ldots, y_n$, respectively, and act approximately like local weighted means of the observations.

Robust filters and smoothers are nonlinear functions of the data that are designed to give good estimates of x_t in the presence of outliers generated by Equation (25.69) or (25.70). Although `acm.filt`, `acm.ave`, and `acm.smo` are capable of robust filtering and smoothing when σ_0^2 is known and positive, none of these functions are capable of estimating σ_0^2 from the data. Estimation of σ_0^2 along with the

autoregression parameters for x_t is a more difficult problem that we will hopefully address in future releases of Spotfire S+. Thus, we assume for the most part that $\sigma_0^2 = 0$. This corresponds to the frequently occurring situation in which the autoregression x_t is observed perfectly a large fraction $(1 - \gamma)$ of the time, and observed with additive outliers a small fraction (γ) of the time.

When $\sigma_0^2 = 0$, the values of x_t are observed perfectly a fraction $1 - \gamma$ of the time, and this corresponds to $z_t = 0$ in Equation (25.69) and $v_t = 0$ in Equation (25.70). For the fraction of time γ when the x_t are unobservable, we replace the terms robust filter and robust smoother by *robust filter-cleaner* and *robust smoother-cleaner*, respectively. We often shorten these terms to simply *filter-cleaner* and *smoother-cleaner.*

A well-designed filter-cleaner has the following intuitively desirable property: for times at which $y_t = x_t$ by virtue of $v_t = 0$ or $z_t = 0$, we have $\hat{x}_t = y_t$. This occurs a large fraction $1 - \gamma$ of the time. For times at which y_t is a gross outlier by virtue of v_t having a large magnitude, $\hat{x}_t$ is a pure prediction based on the previous filter-cleaned values $\hat{x}_1, \ldots, \hat{x}_{t-1}$. A well-designed smoother-cleaner behaves similarly, except at the times when y_t is a gross outlier; at these times, $\hat{x}_t$ is a pure interpolation based on all the other smoother-cleaned data $\hat{x}_1, \ldots, \hat{x}_{t-1}, \hat{x}_{t+1}, \ldots, \hat{x}_r$.

In order to use a robust filter-cleaner or smoother-cleaner for autoregression models

$$x_t = \phi_1 x_{t-1} + \cdots + \phi_p x_{t-p} + \varepsilon_t \ , \qquad (25.71)$$

we must specify the unknown parameters $\phi_1, \phi_2, \ldots, \phi_p$ and s_ε, where s_ε is the scale parameter for the distribution F_ε of the *innovations* ε_t. In the case where $F_\varepsilon = N(0, s_\varepsilon^2)$, we have $s_\varepsilon^2 = \sigma_\varepsilon^2$.

Since we seldom know the parameters $\phi_1, \phi_2, \ldots, \phi_p$ or s_ε, we must estimate them robustly from the data. This may be done using `ar.gm`, which computes a so-called generalized M-estimate, or GM estimate. This kind of robust estimate is also called a bounded influence autoregression estimate, and is described in the section Generalized M-Estimates for Autoregression. The GM estimate produces robust parameter estimates $\ddot{\phi}_1, \ddot{\phi}_2, \ldots, \ddot{\phi}_p$ and $\hat{s}_\varepsilon$, which may be used in any of the robust filter or smoother functions: `acm.ave`, `acm.filt`, and `acm.smo`.

Typically, one uses least squares autoregression model fitting via `ar.yw` or `ar.burg` to produce improved parameter estimates $\ddot{\phi}_1, \ddot{\phi}_2, \ldots, \ddot{\phi}_p$ and $\hat{s}_\varepsilon$. These can, in turn, be used to run `acm.ave` again and obtain improved smoother-cleaned values and least squares estimates of these autoregression parameters. Although one could iterate this procedure several times, we recommend using just one complete iteration of this form, which produces a second set of improved values $\hat{x}_1, \hat{x}_2, \ldots, \hat{x}_p, \ddot{\phi}_1, \ddot{\phi}_2, \ldots, \ddot{\phi}_p$, and $\hat{s}_\varepsilon$. Because of the strongly nonlinear nature of `acm.ave`, further iteration can lead to poor solutions.

Generalized M-Estimates for Autoregression

Generalized M-estimates (or GM estimates) $\ddot{\phi}$ and $\hat{s}_\varepsilon$ of autoregression parameters $\phi^T = (\phi_1, \phi_2, \ldots, \phi_p)$ and the innovations scale parameter s_ε are obtained by solving the equations

$$\sum_{t=p}^{n-1} W(\boldsymbol{y}_t)\boldsymbol{y}_t w_t \cdot (y_{t+1} - \boldsymbol{y}_t^T\hat{\phi}) = 0 \tag{25.72}$$

$$\sum_{t=p}^{n-1} \chi\left(\frac{y_{t+1} - \boldsymbol{y}_t^T\hat{\phi}}{\hat{s}_\varepsilon}\right) = 0\,, \tag{25.73}$$

where the observed time series is $y_1, y_2, \ldots, y_n$, $\mathbf{y}_t^T = y_t, y_{t-1}, \ldots, y_{t-p+1}$, χ is a bounded and continuous function, and both $W(\mathbf{y}_t)$ and w_t are nonnegative, data-dependent weight functions. As we see below, w_t depends on $\hat{\phi}$ as well. We focus our description on Equation (25.72), and refer the reader to Martin (1980) for details concerning Equation (25.73).

Equation (25.72) provides a linear weighted least squares estimate. The estimate is linear in the case where the "big" weights $W(\mathbf{y}_t)$ and the "little" weights w_t are replaced by fixed weights; that is, the weights are independent of both the data $\mathbf{y}_t$ and the estimate $\hat{\phi}$. Because the w_t depend upon $\hat{\phi}$, the equations in (25.72) are nonlinear. They are solved by an iterative weighted least squares method:

$$\sum_{t=p}^{n-1} W(\mathbf{y}_t)\mathbf{y}_t w_t^j \cdot (y_{t+1} - \mathbf{y}_t^T \hat{\phi}^{j+1}) = 0 \qquad j = 0, 1, \ldots, \textit{iter}, \tag{25.74}$$

where ***iter*** is the desired number of iterations, and the first iteration starts with the least squares estimate $\hat{\phi}^0 = \hat{\phi}_{LS}$. Equation (25.73) is also iterated, yielding an estimate $\hat{s}_\varepsilon^j$ at iteration j.

The big weights $W(\mathbf{y}_t)$ are constructed so that $W(\mathbf{y}_t)\mathbf{y}_t$ is bounded and continuous, and the little weights w_t are constructed so that $w_t \cdot (y_{t+1} - \mathbf{y}_t^T \hat{\phi})$ is bounded and continuous. This achieves the basic requirement for robustness: the summands of the estimating equation in (25.72) must be bounded and continuous.

Specifically, the weights w_t^j are obtained from a psi-function ψ_c:

$$w_t^j = \frac{\hat{s}_\varepsilon^j \psi_c((y_{t+1} - \mathbf{y}_t^T \hat{\phi}^j) / \hat{s}_\varepsilon^j)}{y_{t+1} - \mathbf{y}_t^T \hat{\phi}^j}$$

where c is a tuning constant. Two kinds of psi-functions are typically used. Huber's favorite psi-function is defined as (Huber (1964)):

$$\psi_H(r) = \begin{cases} r & |r| < c \\ c \cdot \operatorname{sgn}(r) & |r| \geq c \end{cases}$$

and Tukey's bisquare psi-function is (Mosteller and Tukey (1977)):

$$\psi_B(r) = \begin{cases} r(1-r^2)^2 & |r| \leq c \\ 0 & |r| > c \end{cases}$$

The separate tuning constants c for the psi-functions are adjusted to obtain a compromise between high efficiency when the data are actually Gaussian, and robustness towards outliers.

The "big" weights $W(\boldsymbol{y}_t)$ are also derived from a psi-function of either the Huber or Tukey type. As a default, `ar.gm` uses Tukey's bisquare psi-function. Details concerning the formation of the weights $W(\boldsymbol{y}_t)$ may be found in Martin (1980).

The main idea behind the choice of big weights and little weights is as follows. Basing the big weights $W(\boldsymbol{y}_t)$ on the Tukey bisquare function, the $W(\boldsymbol{y}_t)$ are close to one when $\boldsymbol{y}_t$ is not too large, and $\boldsymbol{y}_t$ therefore has little effect. However, when $\boldsymbol{y}_t$ is "very large" (that is, when it is a gross outlier in the vector sense), the $W(\boldsymbol{y}_t)$ are zero and $\boldsymbol{y}_t$ has no influence on the estimate $\hat{\phi}$. Similar comments apply when w_t is based on the Tukey bisquare function. When the residual $r_t = y_{t+1} - \boldsymbol{y}_t^T\hat{\phi}$ is not too large, w_t is close to one, and when $|r_t|$ is "very large" by virtue of y_{t+1} being a gross outlier, w_t is zero.

Despite its attractive properties, a difficulty arises when w_t is based on the Tukey bisquare function ψ_B. The equations in (25.72) have multiple solutions, and starting the iteration in Equation (25.74) with a least squares estimate might lead to a poor solution. This difficulty is avoided when w_t is based on the Huber psi-function ψ_H, since Equation (25.72) has an essentially unique solution. However, basing

w_t on ψ_H does not result in as much robustness toward large outliers as when w_t is based on ψ_B. Thus, the strategy adopted is to iterate Equation (25.74) a number of times using w_t based on the Huber psi-function, followed by a number of iterations using w_t based on the Tukey psi-function.

Example of simple use

```
> robar <- ar.gm(bicoal.tons, 2)
```

Robust Filtering

Consider the special case where x_t in Equation (25.70) is an AR(1) process with known parameter ϕ. In this case, the robust filtering algorithm is given by

$$\hat{x}_t = \phi\hat{x}_{t-1} + \frac{m_t}{s_t}\psi\left(\frac{y_t - \phi\hat{x}_{t-1}}{s_t}\right),$$

where s_t is a measure of scale for the observation prediction residuals $r_t = y_t - \phi\hat{x}_{t-1}$. The quantity s_t is computed using an auxiliary data-dependent recursion. See Martin (1981) for details. The psi-function we use to compute $\hat{x}_t$ is the Hampel two-part redescending type:

$$\psi_{HA}(r) = \begin{cases} r & |r| \le a \\ \operatorname{sgn}(r)\dfrac{a}{b-a}(b - |r|) & a < |r| \le b \\ 0 & |r| > b \end{cases}$$

The robust filter has the property that, if y_t is a gross outlier large enough that the scaled residual $(y_t - \phi\hat{x}_{t-1})/R_t$ is larger in absolute value than b, then $\hat{x}_t$ is a pure prediction based on the previous robust filter value, $\hat{x}_t = \phi\hat{x}_{t-1}$.

Now consider the case where x_t is a pth-order autoregression. In this case, x_t may be represented in state space form

$$\boldsymbol{x}_t = \Phi \boldsymbol{x}_{t-1} + \varepsilon_t,$$

where $\varepsilon_t^T = \varepsilon_t, 0, \ldots, 0$ and $\boldsymbol{x}_t^T = x_t, x_{t-1}, \ldots, x_{t-p+1}$ are p-dimensional vectors, and

$$\Phi = \begin{bmatrix} \phi_1 & \phi_2 & \cdots & \phi_p \\ 1 & 0 & \ldots & 0 \\ 0 & 1 & 0 & 0 \\ \vdots & & & \vdots \\ 0 & \ldots & 0 & 1 \end{bmatrix}$$

is the so-called state transition matrix. In this case, the robust filter value of time t is

$$\hat{x}_t = (\hat{\boldsymbol{x}}_t)_1 .$$

Namely, x_t is the first component of the *vector* filtered value $\hat{\boldsymbol{x}}_t$ obtained from the recursion

$$\hat{\boldsymbol{x}}_t = \Phi \hat{\boldsymbol{x}}_{t-1} + \frac{\tilde{\boldsymbol{m}}_t}{\boldsymbol{s}_t} \Psi\left(\frac{y_t - \hat{y}_t^{t-1}}{s_t}\right).$$

Here $\tilde{\boldsymbol{m}}_t$ is obtained from an auxiliary, data-dependent recursion. See Martin and Thomson (1982) for details. In the recursion for $\hat{\boldsymbol{x}}_t$,

$$\hat{y}_t^{t-1} = (\Phi \hat{\boldsymbol{x}}_{t-1})_1$$

is the first component of the vector one-step-ahead prediction $\Phi \hat{\boldsymbol{x}}_{t-1}$.

In the usual case where we can use `acm.filt` as a filter-cleaner by setting $\sigma_0 = s_0$ equal to zero, it turns out that

$$\tilde{\boldsymbol{m}}_t^T = s_t, 0, \ldots, 0$$

It is easy to check that when the Hampel two-part psi-function Ψ_{HA} is used and y_t is a "good" data point by virtue of $(y_t - \hat{y}_t^{t-1})/\ s_t$ being less than a in magnitude, then $\hat{x}_t = y_t$. In this case, y_t is not altered if it is "good." This usually occurs for most of the data points when `acm.filt` is used in the filter-cleaner mode.

Examples of simple use

```
> gm <- ar.gm(bicoal.tons, 3)
> bicoal.filt <- acm.filt(bicoal.tons, gm)
```

Two-Filter Robust Smoother

The robust smoother `acm.ave` is constructed using two `acm.filt` robust filters: one "forward" filter $\hat{x}_t^+$ going forward in time over the data, and one "backward" filter $\hat{x}_t^-$ going backward in time over the data.

Let $\hat{x}_{t+1,\ t}^-$ denote the backward one-step-ahead predictor of x_t, given the data $y_{t+1}, \ldots, y_n$. Let p_t^+ denote conditional mean squared error, conditioned on $y_1, \ldots, y_t$, for filtering for the forward filter (this is computed in `acm.filt`). Let m_t^- be the conditional mean-squared error, conditioned on $y_{t+1}, \ldots, y_n$, for predicting x_t for the backward filter (this is also computed in `acm.filt`). Then the robust smoother $\hat{x}_t^n$ is obtained by confining $\hat{x}_t^+$ and $\hat{x}_{t+1,\ t}^-$ in the natural Bayesian way:

$$\hat{x}_t^n = \frac{m_t^- x_t^+ + p_t^+ x_{t+1,\ t}^-}{p_t^+ + m_t^-}$$

This smoother has the following characteristics when used as a smoother-cleaner by setting $\sigma_0 = s_0 = 0$: "good" data points y_t are left unaltered, while gross outliers are replaced by interpolates based on the cleaner data $\hat{x}_1, \ldots, \hat{x}_{t-1}, \hat{x}_{t+1}, \ldots, \hat{x}_r$.

Examples of simple use

```
> gm <- ar.gm(bicoal.tons, 3)
```

```
> bicoal.smo <- acm.ave(bicoal.tons, gm)
```

Alternative Robust Smoother

The alternative robust smoother `acm.smo` is an approximate conditional mean type robust smoother. For details, see Martin (1979).

Examples of simple use

```
> gm <- ar.gm(bicoal.tons, 3)
> bicoal.smo <- acm.smo(bicoal.tons, gm)
```

REFERENCES

Ansley, C.F. (1979). An algorithm for the exact likelihood of a mixed autoregressive-moving average process. *Biometrika* **66**: 59-65.

Bell, W. and Hillmer, S. (1987). Initializing the Kalman filter in the non-stationary case. Research Report CENSUS/SRC/RR-87/33. Washington, DC: Statistical Research Division, Bureau of the Census.

Bloomfield, P. (1976). *Fourier Analysis of Time Series: An Introduction.* New York: Wiley.

Box, G.E.P. and Jenkins, G.M. (1976). *Time Series Analysis: Forecasting and Control.* Oakland, CA: Holden-Day.

Bruce, A. and Martin, R.D. (1989). Leave-k-out diagnostics for time series. *Journal of the Royal Statistical Society, Series B* **51**: 363-401.

Burg, J.P. (1967). Maximum Entropy Spectral Analysis. Paper presented at the 37th Annual International S.E.G. Meeting, Oklahoma City, OK.

Chatfield, C. (1984). *The Analysis of Time Series: An Introduction* (3rd ed.). London: Chapman and Hall.

Dennis, J.E., Gay, D.M., and Welsch, R.E. (1980). An adaptive nonlinear least-squares algorithm. *ACM Transaction Mathematical Software* **7**:348-383.

Harvey A.C. (1981). *Time series models.* New York: Wiley.

Harvey, A.C. and Pierse, A.G. (1984). Estimating missing observations in economic time series. *Journal of the American Statistical Association* **79**: 125-131.

Haslett, J. and Raftery, A.E. (1989). Space-time modelling with long-memory dependence: Assessing Ireland's wind power resource (with discussion). *Journal of the Royal Statistical Society, Series C–Applied Statistics* **38**: 1-50.

Hillmer S.C., Bell W.R., & Tiao G.C. (1983). Modeling considerations in the seasonal adjustments of economic time series. In A. Zellner (ed.) *Applied Time Series Analysis of Economic Data* (pp. 74-100). Washington, D.C.: U.S. Bureau of the Census.

Huber, P.J. (1964). Robust estimation of a location parameter. *Annals of Mathematical Statistics* **35**: 73-101.

James, D.A., and Pregibon, D. (1992). *Chronological Objects in S.* AT&T Technical Report. Muray Hill, NJ: AT&T Bell Laboratories.

Jones, R.H. (1980). Maximum likelihood fitting of ARIMA models to time series with missing observations. *Technometrics* **22**: 389-395.

Kohn, R. and Ansley, C.F. (1985). Efficient estimation and prediction in time series regression models. *Biometrika* **72**: 694-697.

Kohn, R. and Ansley, C.F. (1986). Estimation, prediction, and interpolation for ARIMA models with missing data. *Journal of the American Statistical Association* **81**: 751-761.

Mandelbrot, B.B. (1977). *Fractals: Form, Chance and Dimension.* San Francisco: Freeman.

Martin, R D. (1979). Approximate conditional mean type smoothers and interpolators. In *Smoothing Techniques for Curve Estimation*, pp. 117-143. T. Gasser and M. Rosenblatt (Eds.). Berlin: Springer Verlag.

Martin, R.D. (1980). Robust estimation of autoregressive models. In *Directions in Time Series*, pp. 228-254. D.R. Brillinger and G.C. Tiao (Eds.). Hayward, CA: Institute of Mathematical Statistics.

Martin, R.D. (1981). Robust methods for time series, pp. 683–759. In *Applied Time Series Analysis.* D.F. Findley (Ed.). New York: Academic Press.

Martin, R.D. and Thomson, D.J. (1982). Robust resistant spectrum estimates. *Proceedings of the IEEE* **70**:1097-1115.

Mosteller, F. and Tukey, J.W. (1977). *Data Analysis and Regression.* Reading, MA: Addison-Wesley.

Priestley, M.B. (1981). *Spectral Analysis and Time Series.* London: Academic Press.

Shumway, R.H. (1988). *Applied Statistical Time Series Analysis.* Englewood Cliffs, NJ: Prentice Hall.

Singleton, R.C. (1969). An algorithm for computing the mixed radix fast Fourier transform. *IEEE Transactions on Audio and Electronics* **17**: 93–103.

Whittle, P. (1983) *Prediction and Regulation by Linear Least-Square Methods* (2nd ed.). Minneapolis: University of Minnesota Press.

OVERVIEW OF SURVIVAL ANALYSIS

26

INTRODUCTION

The term *survival analysis* originated in the study and analysis of times to death (that is, survival times) for medical patients diagnosed with some fatal disease. Survival analysis is now a well-developed field of statistical research and methodology that pertains to modeling and testing hypotheses of *failure time data*. These data can be for humans as well as for animals, machines, electronic equipment, automobile components, etc. Hence, the methodology is far more general than the analysis of survival times. In fact, fields of study other than medicine have given other names to the identical methodology discussed here. This chapter might just a well have been called any one of the following:

- Analysis of Failure Time Data
- Reliability Analysis
- Event History Analysis

However, because of the focus of most of the examples, and because of the history of the development of this material, we call it *Survival Analysis*. This helps to simplify the presentation. In examples, we will simply refer to *patients* (or people or subjects) and their *survival times*. You can substitute the appropriate terminology for your field of study as you read if you wish.

Modeling of survival times is based on two distinct approaches: *parametric* and *nonparametric*. The material in this and the following chapters covers both approaches. The addition of parametric survival models extends the functionality of earlier versions of Spotfire S+. The parametric survival functions include methods that predate the nonparametric methods but are still widely used in industrial and manufacturing settings, where estimation of component and system reliability may require extrapolation from accelerated tests. The nonparametric methods are widely used in clinical trials, and include Kaplan-Meier estimates of survival, Cox proportional hazards regression models and extensions due to Andersen and Gill (1982). Miller (1981) and Kalbfleisch and Prentice (1980) are excellent references.

OVERVIEW OF SPOTFIRE S+ FUNCTIONS

Nonparametric survival analysis in Spotfire S+ is based on the **survival5**[1] StatLib entry produced by Terry Therneau of the Mayo Clinic. It differs only slightly from the version 5 code found in StatLib. The expected survival routines have been modified to use `dates` objects for dates, and there have been some minor bug fixes and enhancements. Major enhancements include penalized and frailty models. Terry Therneau has been an important contributor to the documentation for survival analysis in Spotfire S+.

Spotfire S+ 4.5 introduced a new set of functions for life testing analysis based on estimation code originally developed by Meeker and Duke (1981), and subsequently refined by W.Q. Meeker. Additional parametric survival analysis code (`survReg`) was added to Spotfire S+ 2000; this code was taken from the **survival5** library, with a name change from `survreg` to `survReg` for backward compatibility.

In this section we present a brief overview of the functions used for doing survival analysis in Spotfire S+. This section provides an overview of the type of computations, model fitting, and graphical displays available for doing survival analysis in Spotfire S+. More in depth information is contained in the chapters that follow.

Survival Curve Estimates

The function `survfit` fits a Kaplan-Meier or a Fleming-Harrington survival curve, or computes the predicted survival curve for a Cox proportional hazards model.

Examples

- A simple Kaplan-Meier estimate:

```
> survfit(Surv(time, status), data = leukemia)
```

- Print the survival curve estimate, standard errors, and confidence intervals:

```
> summary(survfit(Surv(time, status),
+ data = leukemia))
```

1.

- A Fleming-Harrington estimate:

```
> survfit(Surv(time, status), data = leukemia,
+ type = "fleming-harrington")
```

- A Kaplan-Meier estimate with two groups:

```
> survfit(Surv(time, status) ~ group,
+ data = leukemia)
```

- Predict survival at the average predictor for a Cox model:

```
> survfit(coxph(Surv(futime, fustat) ~ age,
+ data = ovarian))
```

- Predict survival at other than the average predictor for a Cox model:

```
> survfit(coxph(Surv(futime, fustat) ~ age, data =
+ ovarian), newdata = data.frame(age = 70))
```

Important Options

1. Kaplan-Meier or Fleming-Harrington estimate of survival.
2. Greenwood or Tsiatis variance estimate.

Comparing Kaplan-Meier Survival Curves

The function `survdiff` computes one- and k-sample versions of the Fleming-Harrington G^{ρ} family of tests. This includes the log-rank and Gehan-Wilcoxon tests as special cases.

Examples

- Test for the presence of a separate baseline survival for each sex:

```
> survdiff(Surv(time, status) ~ sex, data = lung)
```

- A one-sample test:

```
> pred <- survexp(time ~ ratetable(sex = sex,
+ year = 1970, age = age * 365.25), data = lung,
+ cohort = F)

> survdiff(Surv(time, status) ~ offset(pred),
+ data = lung)
```

Cox Proportional Hazards Models

The function `coxph` fits a Cox proportional hazards model. Available options include stratified and penalized models, as well as models with fixed coefficients. Penalized Cox models include ridge regression, smoothing splines, and frailty terms (random effects) as special cases. User-written penalty functions are also supported.

The `cox.zph` function computes a test of proportional hazards for a fitted Cox model, and also estimates time-dependent coefficients suitable for graphing. This function has an option to compute a global test of the proportional hazards assumption, in addition to tests for each covariate.

Examples

- A standard Cox model:

```
> coxph(Surv(time, status) ~ group,
+ data = leukemia)
```

- A model with time dependent data:

```
> coxph(Surv(start, stop, event) ~ (age + surgery) *
+ transplant, data = heart)
```

- A stratified model containing a separate baseline hazard function for each institution, with patient covariates sex and Karnofsky score:

```
> coxph(Surv(time, status) ~ sex + pat.karno
+ strata(inst), data = lung, na.action=na.exclude)
```

- A simplified ridge regression:

```
> coxph(Surv(futime, fustat) ~ rx + ridge(age,
+ ecog.ps, theta=1), data = ovarian)
```

- A model with a penalized p-spline fit for the `age` variable:

```
> coxph(Surv(futime, fustat) ~ rx + pspline(age),
+ data = ovarian)
```

- A model with a Gaussian random effect for `litter`:

```
> coxph(Surv(time, status) ~ rx + frailty(litter,
+ distribution="gaussian"), data = rats)
```

- Force in a known term, `age`, without estimating a coefficient for it:

```
> coxph(Surv(time, status) ~ offset(age) + sex,
+ data = lung)
```

- Compute proportional hazards test for fitted model:

```
> cox.zph(coxph(Surv(time, status) ~ age + sex +
+ ph.ecog, data = lung, na.action = na.exclude))
```

- Display the estimated coefficients as a function of time:

```
> plot(cox.zph(coxph(Surv(time, status) ~ age + sex +
+ ph.ecog, data = lung, na.action = na.exclude)))
```

Important Options

Breslow, Efron, or exact partial likelihood methods for handling ties.

Parametric Survival Models

The functions `survReg` and `censorReg` fit parametric survival models. The `survReg` function supersedes `survreg`, and includes options for frailty models, nonparametric smooth terms, penalized models, and user-defined distributions; the syntax for penalized models is analogous to that for penalized Cox models. In contrast to other survival functions that use `Surv` to specify the censored response, `censorReg` uses `censor`. The `censor` function in similar in structure to `Surv`, but has flexible options for custom definitions of censor codes.

The function `kaplanMeier`, which extends `survfit` to allow for left and interval censoring, fits Kaplan-Meier models using the same syntax as `censorReg`.

Examples

- A stratified model, with separate baseline hazards for males and females:

```
> survReg(Surv(time, status) ~ sex + age + ph.karno +
+ strata(sex), data = lung, na.action = na.exclude)
```

- Fit a log-gaussian model:

```
> survReg(Surv(days, event) ~ voltage,
+ data = capacitor, dist = "loggaussian")
```

- Fit a Weibull distribution:

```
> censorReg(censor(days, event) ~ voltage,
+ data = capacitor2, weights = weights)
```

- Predict life times from a model for default failure rates:

```
> predict(censorReg(censor(days, event) ~ voltage,
+ data = capacitor2, weights = weights))
```

- Predict failure rates from a model for given life times:

```
> predict(censorReg(censor(days, event) ~ voltage,
+ data = capacitor2, weights = weights), q = c(100,
+ 200, 300),  type = "prob")
```

- Fit an extreme value distribution:

```
> censorReg(censor(days, event) ~ voltage,
+ data = capacitor2, weights = weights,
+ dist = "extreme")
```

- Fit a Weibull distribution stratified by the unique values of voltage:

```
> censorReg(censor(days, event) ~ strata(voltage),
+ data = capacitor2, weights = weights)
```

- Fit a Kaplan-Meier model stratified by the unique values of voltage:

```
> kaplanMeier(censor(days, event) ~ voltage,
+ data = capacitor2, weights = weights)
```

Important Options

1. Distributions include Weibull, smallest extreme value, logistic, log-logistic, normal, log-normal, exponential, log-exponential, Rayleigh, and log-Rayleigh.
2. It is possible to fix the scale parameter, or have it estimated as part of the regression.

Predicted Survival

The function `survexp` predicts survival for an age and sex matched cohort of subjects given a baseline matrix of known hazard rates for the population. Most often, the hazard rates are entries in U.S. mortality tables. Also, a prior Cox model can act as the rate table.

Examples

- Average conditional cohort survival, which defaults to U.S. white:

```
> survexp(time ~ ratetable(sex = sex, year = 1970,
+ age = age * 365.25), conditional = T, data = lung)
```

- Data to enter into a one sample test for comparing the given group to a known population:

```
> pred <- survexp(time ~ ratetable(sex = sex,
+ year = 1970, age = age * 365.25), data = lung,
+ cohort = F)
```

Important Options

1. Matrix of known hazards include U.S., Arizona, Florida, and Minnesota.
2. It is possible to compute estimates of `individual` or `cohort` expected survival.

Utility Functions

The `Surv` and `censor` commands are *packaging* functions; like `I` and `C`, they don't transform their arguments. The `Surv` function is used for the left-hand side of all formulas in the nonparametric survival model fitting functions. The `censor` function, which supports user-defined censor codes, is used for the left-hand side of `censorReg` formulas. For details on specifying censor codes, see the chapter Life Testing.

A `strata` term in a model formula marks a variable or group of variables as strata. If there are multiple variables, each unique combination forms a stratum.

A `frailty` term in a model formula marks a variable as a penalized term or random effect. The distribution of the random effect can be `"gamma"`, `"gaussian"`, or `"t"`.

A `pspline` term in a model formula fits a smoothing spline to the variable using the p-spline basis.

A `ridge` term in a model formula fits a group of variables as a simplified ridge regression.

A `cluster` term in a model formula identifies correlated groups of observations.

An `offset` term in a model formula includes a variable in the model with a fixed coefficient of 1.

Examples

- Right censored data with `status=1` for death and `status=0` for censored:

```
Surv(time, status)
```

- Right censored data, where a value of 3 corresponds to a death:

```
Surv(time, status = 3)
```

- Counting process data:

```
Surv(start, stop, event)
```

- Left censored data:

```
Surv(time, status, type = "left")
```

- Specify a vector of censoring codes explicitly:

```
censor(failure, upper, censor.codes = cens)
```

- Specify `rx` as a stratification variable:

```
strata(rx)
```

- Specify `rx` and `residual.dz` as stratification variables:

```
strata(rx, residual.dz)
```

- Make `NA` a separate group rather than omitting `NA`:

```
strata(rx, na.group = T)
```

- A model with a Gaussian random effect for `litter`:

```
> coxph(Surv(time, status) ~ rx + frailty(litter,
+ distribution="gaussian"), data = rats)
```

- A model with a penalized p-spline fit for the `age` variable:

```
> coxph(Surv(futime, fustat) ~ rx + pspline(age),
+ data = ovarian)
```

- A simplified ridge regression:

```
> coxph(Surv(futime, fustat) ~ rx + ridge(age,
+ ecog.ps, theta=1), data=ovarian)
```

- Mark the observations in the `group` variable as correlated:

```
cluster(group)
```

- Force in a known term, `age`, without estimating a coefficient for it:

```
> coxph(Surv(time, status) ~ offset(age) + sex,
+ data = lung)
```

MISSING VALUES

The handling of missing values (NA) for the survival analysis functions has been enriched in recent releases of Spotfire S+. In particular, the functions `naresid` and `naprint` provide new methods for handling missing values. The main improvements follow.

1. You can specify a global default function for handling missing values. This frees you from having to do it in the call to the model fitting function. For example, to set the global missing value action to delete missing values row-wise, type:

   ```
   > options(na.action = "na.exclude")
   ```

2. A brief report of the action taken is included when printing a fitted model. For example, if `na.exclude` is the action, a message similar to the following is included when the fit object is printed:

   ```
   "14 observations deleted due to missing values".
   ```

3. When residuals and predictions are computed, NAs are appropriately inserted so that the resulting vectors are the same length as the original variables. This allows you to, for example, plot the residuals versus the predictors without worrying about whether the vectors are different lengths. Because of this feature, you can do the following:

   ```
   > fit <- coxph(Surv(time, status) ~ age + sex +
   + ph.ecog + ph.karno, data = lung,
   + na.action = na.exclude)
   > plot(lung$age, residuals(fit))
   ```

Warning

Specifying a global default for handling NAs through the `options` list affects all of the model fitting functions that call `model.frame.default`. The `tree` function does not rely on `model.frame.default`, so it is immune to the global setting. However, virtually all of the remaining model fitting functions call `model.frame.default`, and the global setting effects them. Because of this, it is recommended that you provide the NA action function (for example, `na.exclude`) as the `na.action` argument to the fitting function, rather than rely on the global action.

Additionally, if you fit a survival model relying on a global `NA` action, and you use the fitted model in later computations, errors and/or incorrect values can result if the global `NA` action is different than at the time of fitting the model. If you expect to change the global `NA` action, it is safer to provide the `NA` action function as the `na.action` argument to the fitting function, rather than as a global option.

REFERENCES

Andersen, P.K. & Gill, R.D. (1982). Cox's regression model for counting processes: A large sample study. *Annals of Statistics* **10**: 1100-1120.

Kalbfleisch, J. & Prentice, R.L. (1980). *The Statistical Analysis of Failure Time Data*. New York: John Wiley & Sons, Inc.

Meeker W.Q. & Duke S.D. (1981). CENSOR: A user-oriented computer program for life data analysis. *The American Statistician*, 35(2): 112.

Miller, Rupert G. (1981). *Survival Analysis*. New York: John Wiley & Sons, Inc. (pp. 49-50).

ESTIMATING SURVIVAL 27

INTRODUCTION

A *survival function* defined over time t is, by definition, the probability that a person survives at least to time t. More formally, let T be a positive random variable with distribution function $F(t)$ and density $f(t)$. The survival function $S(t)$ is

$$S(t) = 1 - F(t) = P\{T > t\}$$

and the *hazard rate* or *hazard function* $\lambda(t)$ is

$$\lambda(t) = \frac{f(t)}{S(t)}.$$

The hazard rate has the interpretation $\lambda(t) = P\{$patient dies in the next small unit of time, $\Delta(t)$, given they have survived to time $t\}$. A constant hazard indicates that, over each interval, a constant proportion of surviving subjects is expected to die. A familiar example is radioactive decay, where the "death" of an atom corresponds to its decay. Constant hazard may also be associated with some fatal diseases, such as metastatic cancer.

The *cumulative hazard* $\Lambda(t)$ is defined as

$$\Lambda(t) = \int_0^t \lambda(t)dt = -\log S(t).$$

What distinguishes survival analysis from most other statistical methods is the presence of *censoring*. In a study of survival following two different treatment regimens, for example, analysis of the trial typically occurs well before all of the patients have died. For those still alive at the time of analysis, the true survival time is known only to be greater than the time observed to date. Such an observation is said to be *censored*. Survival data are presented to the computer program as a pair (t_i, δ_i), where t_i is the observed survival time and $\delta_i = 0$ if the observation is censored, $\delta_i = 1$ if a death is observed. Survival data is often presented using a + for the censored observation, so that a set of times might be 8, 11+, 14, 22, 36+, etc.

Let $t_1^* < t_2^* < \cdots < t_m^*$ denote the m distinct death times. Let $Y_i(s)$ be an indicator function, which is 1 if person i is still at risk at time s and 0 otherwise; that is, $Y_i(s) = 1$ if $s \leq t_i^*$. Then the number at risk at time s is $r(s) = \sum_1^n Y_i(s)$. We can similarly define $d(s)$ as the number of deaths occurring at time s .

In order to discuss some of the more recent methods in survival analysis, it is helpful to recast the problem as a counting process, a notation found in Andersen and Gill (1982) and others. A good reference is Fleming and Harrington (1981). Let $N_i(t)$ be a counting process associated with the ith subject, so N_i increases by 1 at each observed event (for example, heart attack). In this notation a subject can have multiple events. $Y_i(t)$ is an indicator function as before, but now can have multiple transitions from 0 (zero) to 1 (one), with a subject entering and leaving the risk set.

KAPLAN-MEIER ESTIMATOR

The most common estimate of the survival distribution, the Kaplan-Meier (KM) estimate, is a product of survival probabilities

$$\hat{S}_{KM}(t) = \prod_{t_i < t} \frac{r(t_i) - d(t_i)}{r(t_i)},$$

where r and d are the number at risk and the number of deaths, respectively, as defined above. Graphically, the Kaplan-Meier survival curve appears as a step function with a drop at each death. Censoring times are often marked on the plot as "+" symbols.

Example: AML Study

The data presented in Table 27.1 are preliminary results from a clinical trial to evaluate the efficacy of maintenance chemotherapy for acute myelogenous leukemia (AML). The study was conducted by Embury, *et al.* (1977) at Stanford University. After reaching a status of remission through treatment by chemotherapy, the patients who entered the study were assigned randomly to two groups. The first group received maintenance chemotherapy; the second, or control, group did not. The objective of the trial was to see if maintenance chemotherapy prolonged the time until relapse.

Table 27.1: *Data for AML maintenance study. A+ indicates a censored value.*

Group	Length of Complete Remission (in weeks)
Maintained	9, 13, 13+, 18, 23, 28+, 31, 34, 45+, 48, 161+
Nonmaintained	5, 5, 8, 8, 12, 16+, 23, 27, 30, 33, 43, 45

The Kaplan-Meier estimator of survival for the maintained group is computed by hand as follows:

$$
\begin{aligned}
S(0) &= 1, \\
S(9) &= S(0) \times \frac{10}{11} = 0.91, \\
S(13) &= S(9) \times \frac{9}{10} = 0.82, \\
S(18) &= S(13) \times \frac{7}{8} = 0.72, \\
S(23) &= S(18) \times \frac{6}{7} = 0.61, \\
S(28) &= S(23) \times \frac{6}{6} = 0.61, \\
S(31) &= S(23) \times \frac{4}{5} = 0.49, \\
S(34) &= S(31) \times \frac{3}{4} = 0.37, \\
S(48) &= S(34) \times \frac{1}{2} = 0.18
\end{aligned}
$$

In Spotfire S+, the `survfit` function produces Kaplan-Meier survival curve estimates by default. The data displayed in Table 27.1 is in a data frame named `leukemia`, with the variables listed below.

- `time`: Time to relapse
- `status`: Indicator whether the observed time was a relapse (1) or censored (0).
- `group`: Treatment group indicator taking values `Maintained` and `Nonmaintained`.

You compute the KM estimate as follows:

```
> leukemia.surv <- survfit(Surv(time, status) ~ group,
+ data = leukemia)

> summary(leukemia.surv)
```

```
Call: survfit(formula = Surv(time, status) ~ group, data =
leukemia)

                group=Maintained
time n.risk n.event survival std.err lower 95% CI upper 95%
CI
   9     11       1    0.909  0.0867       0.7541        1.000
  13     10       1    0.818  0.1163       0.6192        1.000
  18      8       1    0.716  0.1397       0.4884        1.000
  23      7       1    0.614  0.1526       0.3769        0.999
  31      5       1    0.491  0.1642       0.2549        0.946
  34      4       1    0.368  0.1627       0.1549        0.875
  48      2       1    0.184  0.1535       0.0359        0.944

             group=Nonmaintained
time n.risk n.event survival std.err lower 95% CI upper 95%
CI
   5     12       2   0.8333  0.1076       0.6470        1.000
   8     10       2   0.6667  0.1361       0.4468        0.995
  12      8       1   0.5833  0.1423       0.3616        0.941
  23      6       1   0.4861  0.1481       0.2675        0.883
  27      5       1   0.3889  0.1470       0.1854        0.816
  30      4       1   0.2917  0.1387       0.1148        0.741
  33      3       1   0.1944  0.1219       0.0569        0.664
  43      2       1   0.0972  0.0919       0.0153        0.620
  45      1       1   0.0000      NA           NA           NA
```

The `survfit` function returns an object of class `"survfit"`. The function produces the tabled output including columns for the survival estimates, the standard errors of the estimates, and confidence bounds for the estimates. The `NA`s on the last line result from not being able to estimate a standard error and, consequently, a confidence interval for zero survival on a log survival scale.

NELSON AND FLEMING-HARRINGTON ESTIMATORS

Another approach is to estimate Λ, the cumulative hazard, using Nelson's estimate,

$$\hat{\Lambda}_N(t) = \sum_{t_i < t} \frac{d(t_i)}{r(t_i)},$$

or, using counting process notation,

$$\hat{\Lambda}_N(t) = \sum_{i=1}^{n} \int_0^t \frac{dN_i(s)}{r(s)}.$$

The Nelson estimate is also a step function. It starts at zero and has a step of size $\frac{d(t)}{r(t)}$ at each death.

One problem with the Nelson estimate is that it is susceptible to ties in the data. For example, assume that 3 subjects die at 3 nearby times t_1, t_2, t_3, with 7 other subjects also at risk. Then the total increment in the Nelson estimate is 1/10 + 1/9 + 1/8. However, if time data were grouped such that the distinction between t_1, t_2, and t_3 was lost, the increment would be the smaller step 3/10. If there are a large number of ties this can introduce significant bias. One solution is to employ a modified Nelson estimate that always uses the larger increment, as suggested by Nelson (1969) and Fleming and Harrington (1984). This is not an issue with the Kaplan-Meier estimate, however; with or without ties, the multiplicative step would be 7/10.

The relationship $\Lambda(t) = -\log S(t)$, which holds for any continuous distribution, leads to the Fleming-Harrington (FH) (Fleming and Harrington (1984)) estimate of survival:

$$\hat{S}_{FH}(t_j) = e^{-\hat{\Lambda}_N(t_j)} \tag{27.1}$$

This estimate has natural connections to survival curves for a Cox model. For sufficiently large sample sizes the FH and KM estimates are arbitrarily close to one another, but keep in mind that unless there is heavy censoring, the number at risk $r(t)$ is always small in the right-hand tail of the estimated curve.

Example: AML Study (cont.)

You produce the Fleming-Harrington estimate of survival for the data in Table 27.1 by specifying the `type` argument in the call to `survfit`.

```
> summary(survfit(Surv(time, status) ~ group,
+ data = leukemia, type = "fleming-harrington"))

Call: survfit(formula = Surv(time, status) ~ group, data =
leukemia, type = "fleming-harrington")

                group=Maintained
time n.risk n.event survival std.err lower 95% CI upper 95%
CI
   9     11       1    0.913  0.0871       0.7575        1.000
  13     10       1    0.826  0.1174       0.6253        1.000
  18      8       1    0.729  0.1422       0.4974        1.000
  23      7       1    0.632  0.1572       0.3882        1.000
  31      5       1    0.517  0.1731       0.2687        0.997
  34      4       1    0.403  0.1781       0.1695        0.958
  48      2       1    0.244  0.2038       0.0477        1.000

                group=Nonmaintained
time n.risk n.event survival std.err lower 95% CI upper 95%
CI
   5     12       2   0.8465   0.109       0.6572        1.000
   8     10       2   0.6930   0.141       0.4645        1.000
  12      8       1   0.6116   0.149       0.3791        0.987
  23      6       1   0.5177   0.158       0.2849        0.941
  27      5       1   0.4239   0.160       0.2021        0.889
  30      4       1   0.3301   0.157       0.1300        0.838
  33      3       1   0.2365   0.148       0.0692        0.808
  43      2       1   0.1435   0.136       0.0225        0.914
  45      1       1   0.0528     Inf       0.0000        1.000
```

You produce the modified Nelson estimate similarly by specifying `type = "fh2"`. Note that you can abbreviate the character string passed to the `type` argument:

```
# Fleming-Harrington estimate
```

```
> survfit(Surv(time, status) ~ group, data = leukemia,
+ type = "flem")

# Nelson estimate
> survfit(Surv(time, status) ~ group, data = leukemia,
+ type = "fh")
```

VARIANCE ESTIMATION

Several estimates of the varaiance of $\hat{\Lambda}_N$ are possible. Since $\hat{\Lambda}_N$ can be treated as a sum of independent increments, the variance is a cumulative sum with terms of

$$\frac{d(t)}{r(t)[r(t)-d(t)]} \quad \text{Greenwood}$$

$$\frac{d(t)}{r^2(t)} \quad \text{Tsiatis}$$

$$\frac{d(t)[r(t)-d(t)]}{r^3(t)} \quad \text{Klein}$$

See Klein (1991) for details. Using Equation (27.1) and the simple Taylor series approximation $\text{var} \log f \approx \text{var} f / f$, the variance of the KM or FH estimators is

$$\text{var}(\hat{S}(t)) = \hat{S}^2(t)\text{var}(\hat{\Lambda}_N(t)) \tag{27.2}$$

Klein also considers two other forms for the variance of S, but concludes

- For computing the variance of $\hat{\Lambda}_N$ the Tsiatis formula is preferred.
- For computing the variance of $\hat{S}$, the Greenwood formula along with Equation (27.2) is preferred.

Confidence intervals for $S(t)$ can be computed on the plain (identity) scale,

$$S \pm 1.96\text{se}(S) \tag{27.3}$$

on the cumulative hazard or log-survival scale,

$$\exp(\log S \pm 1.96\ \mathrm{se}(\Lambda)) \tag{27.4}$$

or on the log-hazard or log-log survival scale,

$$\exp(-\exp(\log(-\log S) \pm 1.96\ \mathrm{se}(\log \Lambda))) \tag{27.5}$$

where "se" refers to the standard error.

Confidence intervals based on Equation (27.3) may give survival probabilities that are greater than 1 or less than zero. Those based on Equation (27.4) may sometimes be greater than 1, but those based on Equation (27.5) are always between 0 and 1. For this reason many users prefer the log-hazard formulation. Link (1984), (1986), however, suggests that confidence intervals based on the cumulative-hazard scale have the best performance. All three methods have been implemented in the `survfit` function and are referred to as the `"plain"`, `"log"`, and `"log-log"` confidence types. By default, the `summary.survfit` confidence intervals are based on the log-survival (or cumulative hazard) scale. Intervals on the two other scales may be specified through the `conf.type` argument to `survfit`. Intervals on the other scales are computed based on the following relationships:

$$\mathrm{se}(S) \cong S\,\mathrm{se}(\Lambda)$$

$$\mathrm{se}(\log \Lambda) \cong \frac{1}{\Lambda}\,\mathrm{se}(\Lambda$$

A further refinement to the confidence intervals is suggested by Dorey and Korn (1987). When the tail of the survival curve contains much censoring and few deaths, there will be one or more long flat segments. Confidence intervals based strictly on Equation (27.3), Equation (27.4), or Equation (27.5) are constant across these intervals. Dorey and Korn point out that, as censored subjects are removed from the sample, the effective sample size decreases, so the actual reliability of the curve should also decrease. Their correction retains the original upper confidence limit and a modified lower limit which agrees with the standard limits at each death time but is based on the *effective number at risk* between death times.

Three lower confidence limit methods (the `conf.lower` argument) are implemented in `survfit`. The usual method (`conf.lower="usual"`) uses, optionally, either the Greenwood or the Tsiatis formulation unaltered.

Peto's method (`conf.lower="peto"`) assumes that

$$\text{var}(\hat{\Lambda}_N(t)) = \frac{c}{r(t)},$$

where $r(t)$ is the number at risk and $c \equiv 1 - \hat{S}(t)$. The Peto limit is known to be conservative. The *modified* Peto limit (`conf.lower="modified"`) chooses c such that the variance at each death time is equal to the usual estimate but between death times the usual variance estimate is multiplied by $\frac{r^*(t)}{r(t)}$, where $r(t)$ is the number at risk and $r^*(t)$ is the number at risk at the last jump in the curve (last death time). This is almost identical to Dorey and Korn's estimator and is the recommended procedure.

Example: AML Study (cont.)

Applying the methods of this section to the `leukemia` data, you can compute the conservative lower confidence intervals of Peto for survival based on the log-hazard scale as follows:

```
> summary(survfit(Surv(time, status) ~ group,
+ data = leukemia, conf.type = "log-log",
+ conf.lower = "peto"))

Call: survfit(formula = Surv(time, status) ~ group, data =
leukemia, conf.type = "log-log", conf.lower = "peto")

                group=Maintained
time n.risk n.event survival std.err lower 95% CI upper 95%
CI
   9     11       1    0.909  0.0867       0.5390        0.987
  13     10       1    0.818  0.1163       0.4729        0.951
  18      8       1    0.716  0.1397       0.3645        0.899
  23      7       1    0.614  0.1526       0.2854        0.835
  31      5       1    0.491  0.1642       0.1802        0.753
  34      4       1    0.368  0.1627       0.1132        0.657
  48      2       1    0.184  0.1535       0.0288        0.525
```

```
                group=Nonmaintained
time n.risk n.event survival std.err lower 95% CI upper 95%
CI
   5     12       2   0.8333  0.1076       0.5235        0.956
   8     10       2   0.6667  0.1361       0.3753        0.860
  12      8       1   0.5833  0.1423       0.2906        0.801
  23      6       1   0.4861  0.1481       0.2024        0.730
  27      5       1   0.3889  0.1470       0.1421        0.650
  30      4       1   0.2917  0.1387       0.0901        0.561
  33      3       1   0.1944  0.1219       0.0476        0.461
  43      2       1   0.0972  0.0919       0.0166        0.349
  45      1       1   0.0000      NA           NA           NA
```

MEAN AND MEDIAN SURVIVAL

For the Kaplan-Meier estimate, the estimated mean survival is undefined if the last observation is censored. The procedure used by Spotfire S+ is to redefine the estimate to be zero beyond the last observation. This gives an estimated mean that is biased towards zero, but there are no compelling alternatives that do better. With this definition, the mean is estimated as

$$\hat{\mu} = \int_0^T \hat{S}(t)dt,$$

where $\hat{S}$ is the Kaplan-Meier estimate and T is the maximum observed follow-up time in the study. The variance of the mean is

$$\mathrm{var}(\hat{\mu}) = \int_0^T \left(\int_t^T \hat{S}(u)du \right)^2 \frac{dN(t)}{r(t)(r(t)-N(t)},$$

where $N(t) = \sum N_i(t) = d(t)$ is the total number of deaths up to time t, and $r(t) = \sum Y_i(t)$ is the number at risk at time t.

The sample median is defined as the first time at which $\hat{S}(t) \leq 0.5$. Upper and lower confidence intervals for the median are defined in terms of the confidence intervals for S: the upper confidence interval is the first time at which the upper confidence interval for $\hat{S}$ is ≤ 0.5. This corresponds to drawing a horizontal line at 0.5 on the graph of the survival curve, and using intersections of this line with the curve and its upper and lower confidence bands. In the event that the survival curve has a horizontal portion at exactly 0.5 (for example, an even number of subjects and no censoring before the median) then the average time of that horizontal segment is used. This agrees with the usual definition of the median for uncensored data when the sample size is an even number. If neither confidence band for $S(t)$ reaches 0.5, as in the example which follows, then the corresponding confidence limit for the median is unknown and is reported as an `NA`.

Example: AML Study (cont.)

The mean, median, and confidence intervals for the median survival time are part of the table produced by printing a `"survfit"` object. For the `leukemia` data set these statistics are produced as follows:

```
> leukemia.surv <- survfit(Surv(time, status) ~ group,
+ data = leukemia)
> leukemia.surv

Call: survfit(formula = Surv(time, status) ~ group, data =
  leukemia)

                     n events mean se(mean) median 0.95LCL
   group=Maintained 11      7 52.6    19.83     31      18
group=Nonmaintained 12     11 22.7     4.18     23       8

                    0.95UCL
   group=Maintained      NA
group=Nonmaintained      NA
```

Printing the object returned by `survfit` produces a brief report of the resulting fits. For each fit, the `print` method prints the number of subjects in the cohort (`n`), the total number of events (`events`), as well as the mean, its standard error (`se(mean)`), the median, and confidence intervals for the median survival time (the last two columns).

COMPARISON OF SURVIVAL CURVES

Assume that we wish to compare p different groups with respect to their survival distributions. One method is to form the $p \times 2$ table at each death time.

Groups	**1**	**2**	**...**	***p***	
Deaths	d_1	d_2		d_p	d
Alive and at risk	a_1	a_2		a_p	a
Totals	n_1	n_2		n_p	N

If there are no tied deaths, then $d = 1$ for each table. Treating this table as a simple multinomial experiment with d events in N trials, the expected number of deaths in each group is dn_i / N with a standard multinomial variance matrix V.

Treating each of the k unique death time tables as independent, we can sum over the tables to obtain an observed and an expected number of deaths for each group. This "O-E" vector has variance matrix $\sum V_k$. The argument may be generalized by the inclusion of weights w_k for each death time. The overall weighted vector is then $\sum w_k(O_k - E_k)$, where O_k is the top row of table k, E_k is the expected, and the variance is $\sum w_k^2 V_k$. When $w_k = 1$ this is the Mantel-Haenszel or log-rank test, for $w_k = n_k$ it is the Gehan-Wilcoxon test, and for $w_k = S_{KM}(t_k)$ it is the Peto-Peto modification of the Wilcoxon test.

The `survdiff` function implements a family of tests suggested by Fleming and Harrington (1981) for comparing two or more survival curves. A single parameter ρ controls the weights given to different

survival times; $\rho = 0$ yields the log-rank test and $\rho = 1$ the Peto-Wilcoxon. Other values give a test that is intermediate to these two. The default value is $\rho = 0$.

The log rank test is most powerful for a proportional hazards alternative, that is, when $\lambda_i(t) / \lambda_j(t) = c_{i_j}$ for any two groups i and j, and some constant c which is independent of time. This assumption is found to hold, at least approximately, in many clinical trials. Other values for ρ produce tests more sensitive to early differences in S $(\rho > 0)$ or to later differences $(\rho < 0)$.

Example: AML Study (cont.)

Returning to the `leukemia` data frame, compare the two treatment groups using `survdiff`. The `survdiff` function takes a formula and a data frame as its first two arguments. Recalling that $\rho = 0$ by default, the log-rank test for difference between the maintained and nonmaintained groups is produced as follows:

```
> survdiff(Surv(time, status) ~ group, data = leukemia)

                    N Observed Expected (O-E)^2/E
   group=Maintained 11        7   10.689     1.273
group=Nonmaintained 12       11    7.311     1.862

 Chisq= 3.4 on 1 degrees of freedom, p= 0.06534
```

Thus, there is mild evidence to suggest that the maintained group has better survival than the nonmaintained group.

MORE ON SURVFIT

The `survfit` function fits Kaplan-Meier or, optionally, Fleming-Harrington survival curves. For example,

```
> sf <- survfit(Surv(futime, fustat) ~ rx + residual.dz,
+ data = ovarian)
> sf

Call: survfit(formula = Surv(futime, fustat) ~ rx +
      residual.dz, data = ovarian)

                     n events mean se(mean) median 0.95LCL
rx=1, residual.dz=1 5      1  989      101     NA     638
rx=1, residual.dz=2 8      6  430      131    299     156
rx=2, residual.dz=1 6      2  943      161     NA     563
rx=2, residual.dz=2 7      3  833      156     NA     464

                     0.95UCL
rx=1, residual.dz=1      NA
rx=1, residual.dz=2      NA
rx=2, residual.dz=1      NA
rx=2, residual.dz=2      NA
```

This command results in four Kaplan-Meier survival curves, indexed by the two levels of treatment (`rx`) and the two levels of residual disease (`residual.dz`). The right hand side of the formula is interpreted differently than it would be for an ordinary linear or Cox model. The `survfit` function uses the + operator to specify an interaction.

A summary of important options to `survfit` are listed below.

- `weights`: Case weights.
- `type`: Type of fit. The choices are `"kaplan-meier"`, `"fleming-harrington"` or `"fh2"`.
- `error`: Type of variance estimate. The choices are `"greenwood"` or `"tsiatis"`.
- `conf.int`: Level for the two-sided confidence interval of median survival. The default is 0.95.

- `conf.type`: One of `"none"`, `"plain"`, `"log"`, or `"log-log"`. The default is `"log"`.
- `conf.lower`: One of `"usual"`, `"peto"`, or `"modified"`.

The `plot.survfit` function plots survival curves returned by `survfit`. For the AML data, you can plot survival curves, and add a title and legend as follows:

```
> plot(leukemia.surv, xlab = "Survival Time in Weeks",
+ ylab = "Proportion Surviving", cex = 2, lty = 2:3)
> title("AML Maintenance Study")
> legend(c(75, 130), c(0.95, 0.85),
+ c("Maintenance", "No Maintenance"), lty = 2:3)
```

Figure 27.1 displays the results of plotting the Kaplan-Meier estimates of survival stratified by the maintenance grouping variable `group`. Some important optional arguments to `plot.survfit` are as follows:

- `conf.int`: Plot confidence intervals for the curves. The default is `TRUE` for a single curve and `FALSE` for multiple curves.
- `mark.time`: If logical, indicates whether to mark the curves at censoring times. If a numeric vector, the curve is marked at each time indicated.
- `mark`: Vector of characters or integers specifying special symbols used to mark the curve. The default value of 3 produces a + at the censored values.
- `cex`: Character size of the censor marks.

By default, confidence intervals are suppressed if there are multiple curves. Marks are normally placed on the curve(s) at each censoring time. If there are a large number of censored observations, this can make the plot too "busy." In this case, the `mark.time` option can be used to specify the time values at which curves are labeled.

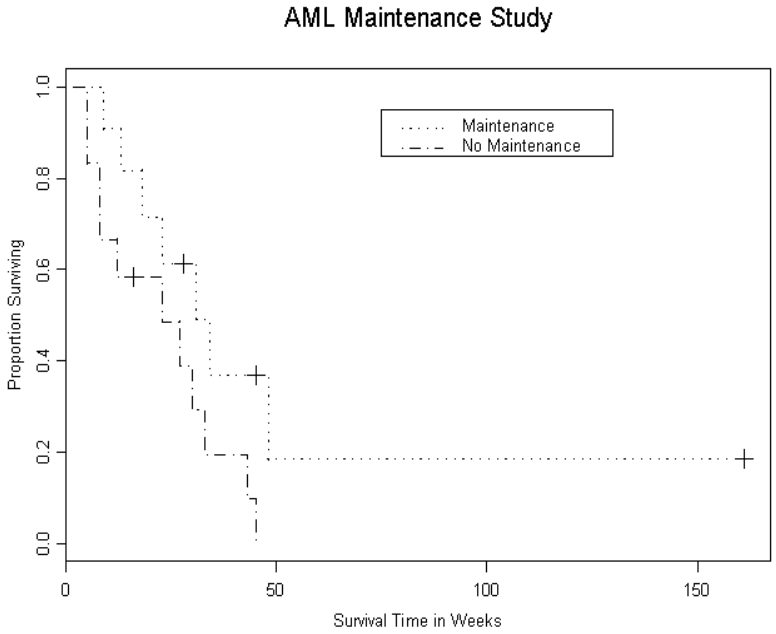

Figure 27.1: *Kaplan-Meier estimates of survival for the maintained and nonmaintained groups of the AML study.*

REFERENCES

Andersen, P.K. & Gill, R.D. (1982). Cox's regression model for counting processes: A large sample study. *Annals of Statistics* **10**:1100-1120.

Dorey, F.J. & Korn, E.L. (1987). Effective sample sizes for confidence intervals for survival probabilities. *Statistics in Medicine* **6**:679-687.

Embury, S.H., Elias, L., Heller, P.H., Hood, C.E., Greenberg, P.L., & Schrier, S.L. (1977). Remission maintenance therapy in acute myelogenous leukemia. *Western Journal of Medicine* **126**:267-272.

Fleming, T.R. & Harrington, D.P. (1981). A class of hypothesis tests for one and two sample censored survival data. *Communications in Statistics*, A10(8):763-794.

Fleming, T.R. & Harrington, D.P. (1984). Nonparametric estimation of the survival distribution in censored data. *Communications in Statistics*, 13(20):2469-2486.

Fleming, T. & Harrington, D. (1991). *Counting Processes and Survival Analysis*. New York: John Wiley & Sons, Inc.

Klein, J.P. (1991). Small sample moments of the estimators of the variance of the Kaplan-Meier and Nelson-Aalen estimators. *Scandinavian Journal of Statistics* **18**:333-340.

Link, C.L. (1984). Confidence intervals for the survival function using Cox's Proportional-Hazard model with covariates. *Biometrics* **40**:601-610.

Link, C.L. (1986). Confidence intervals for the survival function in the presence of covariates. *Biometrics* **42**:219-220.

Nelson W.B. (1969). Hazard plotting for incomplete failure data. *Journal of Quality Technology* **1**:27-52.

THE COX PROPORTIONAL HAZARDS MODEL

28

INTRODUCTION

The Cox proportional hazards model is the most commonly used regression model for survival data. If $Z_i(t)$ is the vector of covariates for the ith individual at time t, the model assumes that the hazard for a subject is of the form

$$\lambda(t;Z_i) = \lambda_0(t) r_i(t),$$

where

$$r_i(t) = e^{\beta' Z_i(t)}$$

is referred to as the *risk score* for the ith subject, β is a vector of regression parameters, and $\lambda_0(t)$ is an arbitrary and unspecified baseline hazard function. The vector of coefficients β does not include an intercept term; it is absorbed into λ_0. The exponential function guarantees that λ is positive for any β. Assume that a death has occurred at time t^*. Then conditional on this death occurring, the likelihood that it would be subject i rather than some other subject is

$$L_i(\beta) = \frac{\lambda_0(t^*)\, r_i(t^*)}{\sum_j Y_j(t^*)\, \lambda_0(t^*)\, r_j(t^*)} = \frac{r_i(t^*)}{\sum_j Y_j(t^*)\, r_j(t^*)} . \qquad (28.1)$$

The product of the terms (Equation (28.1)) over all death times, $L(\beta) = \prod L_i(\beta)$, was termed a partial likelihood by Cox (1972). Maximization of $\log(L(\beta))$ gives an estimate for β without the need to estimate the nuisance parameter $\lambda_0(t)$. An estimator of the covariance matrix is given by the inverse of the second derivative matrix. The proportional hazards model is nonparametric in the

sense that it depends only on the ranks of the survival times. It remains sensitive, however, to skewed covariates. The first derivative of $\log(L(\beta))$ is the p by 1 vector

$$\begin{aligned} J(\beta) &= \sum_{i=1}^{n} \int_0^{\infty} [Z_i(t) - \bar{Z}(\beta,\ t)]\, dN_i(t) \\ &= \sum_{i=1}^{n} \int_0^{\infty} [Z_i(t) - \bar{Z}(\beta,\ t)]\, dM_i(\beta,\ t \end{aligned} \tag{28.2}$$

$$\begin{aligned} J(\beta) &= \sum_{i=1}^{n} \int_0^{\infty} [Z_i(t) - \bar{Z}(\beta,\ t)]\, dN_i(t) \\ &= \sum_{i=1}^{n} \int_0^{\infty} [Z_i(t) - \bar{Z}(\beta,\ t)]\, dM_i(\beta,\ t \end{aligned} \tag{28.3}$$

and the p by p information matrix is

$$I(\beta) = \sum_{i=1}^{n} \int_0^{\infty} \frac{\sum_j Y_j(t)\, r_j(t) [Z_i(t) - \bar{Z}(t)][Z_i(t) - \bar{Z}(\beta,\ t)]'}{\sum_j Y_j(t)\, r_j(t)} dN_i(t) \tag{28.4}$$

where $\bar{Z}$ is the weighted covariate mean for those still at risk at time t

$$\bar{Z}(\beta,\ t) = \frac{\sum_j Y_j(t)\, r_j(t) Z_i(t)}{\sum_i Y_i(t)\, r_i(t)}.$$

Cox proposed, and it was later shown by Efron (1977) and Oakes (1977), that the partial likelihood contains nearly all of the information about β. That is, the calendar times when deaths occur give information about the overall hazard rate λ_0 but little about the

relative rates for different values of Z. The Cox model thus gives very efficient estimates as compared to a parametric proportional hazards model, such as the Weibull, even when the data actually come from the parametric model. The notation for L_i in Equation (28.1) is derived from the *counting process* representation found in Fleming and Harrington (1991). It allows for several extensions to the original Cox model formulation, including:

- Multiple events per subject;
- Time-dependent covariates including cation variables;
- Discontinuous intervals of risk, where Y_i may change states from 1 to 0 and back again multiple times;
- Left truncation, where subjects need not enter the risk set at time 0.

This extension is known as the *multiplicative hazards model.*

Example: Ovarian Cancer

This example uses data from a study of ovarian cancer [EFD+79]. The variables are listed below.

- `futime`: The number of days from enrollment until death or censoring, whichever comes first;
- `fustat`: An indicator of death (1) or censoring (0);
- `age`: The patient age in years (actually, the age in days divided by 365.25);
- `residual.dz`: An indicator of the extent of residual disease;
- `rx`: An indicator of the treatment given;
- `ecog.ps`: A measure of performance score or functional status, using the Eastern Cooperative Oncology Group's scale. It ranges from 0 (fully functional) to 4 (completely disabled). Level 4 subjects are usually considered too ill to enter a randomized trial such as this.

The data are stored in a data frame named `ovarian`. A summary produces the following:

```
> summary(ovarian)
```

```
     futime          fustat           age
 Min.   :  59.0  Min.   :0.0000  Min.   :38.89
 1st Qu.: 368.0  1st Qu.:0.0000  1st Qu.:50.17
 Median : 476.0  Median :0.0000  Median :56.85
 Mean   : 599.5  Mean   :0.4615  Mean   :56.17
 3rd Qu.: 794.8  3rd Qu.:1.0000  3rd Qu.:62.38
 Max.   :1227.0  Max.   :1.0000  Max.   :74.50
  residual.dz          rx            ecog.ps
 Min.   :1.000   Min.   :1.0     Min.   :1.000
 1st Qu.:1.000   1st Qu.:1.0     1st Qu.:1.000
 Median :2.000   Median :1.5     Median :1.000
 Mean   :1.577   Mean   :1.5     Mean   :1.462
 3rd Qu.:2.000   3rd Qu.:2.0     3rd Qu.:2.000
 Max.   :2.000   Max.   :2.0     Max.   :2.000
```

Start by modeling survival as a function of `age` only:

```
> ov.fit1 <- coxph(Surv(futime, fustat) ~ age,
+ data = ovarian)
> ov.fit1

Call: coxph(formula = Surv(futime,fustat) ~ age, data =
ovarian)
      coef exp(coef) se(coef)    z      p
age 0.162      1.18   0.0497 3.25 0.0012
Likelihood ratio test=14.3  on 1 df, p=0.000156  n=26
```

Printing the resulting fit produces the estimated coefficient $(\hat{\beta})$, the estimated relative risk for a one unit change in the variable $e^{(\hat{\beta})}$, the standard error of the estimated coefficient, a z-test $(\hat{\beta})/\ se(\hat{\beta})$ along with its p-value for the significance of the estimated coefficient, and a likelihood ratio test for goodness of fit. The z-test is sometimes referred to as Wald's test. An estimate of the relative risk of dying of ovarian cancer for two patients in the study differing in age by one year is 1.18 which is significantly larger than one $(p = 0.000156)$. The older patient has an estimated 1.18 times higher risk of dying of ovarian cancer than the younger patient. You produce a summary of the survival curve with a combination of the `summary` function and the `survfit` function. For example,

```
> summary(survfit(ov.fit1))
```

```
Call: survfit(formula = ov.fit1)

 time n.risk n.event survival std.err
   59     26       1    0.988  0.0142
  115     25       1    0.974  0.0244
  156     24       1    0.955  0.0364
  268     23       1    0.933  0.0482
  329     22       1    0.897  0.0621
  353     21       1    0.862  0.0724
  365     20       1    0.824  0.0819
  431     17       1    0.775  0.0934
  464     15       1    0.724  0.1032
  475     14       1    0.673  0.1112
  563     12       1    0.596  0.1226
  638     11       1    0.520  0.1287

 lower 95% CI upper 95% CI
        0.961        1.000
        0.927        1.000
        0.886        1.000
        0.844        1.000
        0.783        1.000
        0.732        1.000
        0.678        1.000
        0.612        0.982
        0.548        0.958
        0.487        0.931
        0.398        0.892
        0.321        0.845
```

The Fleming-Harrington estimate of survival for a patient with age equal to the *average* is produced in this case because the model was fit using `coxph` and survival for a particular age was not specified with the `newdata` argument.

You can produce a plot of the survival curve, shown in Figure 28.1, at the average age as follows:

```
> plot(survfit(ov.fit1), xlab = "Survival in Days",
+ ylab = "Proportion Surviving")
> title("Suvival for Ovarian Cancer Study")
```

The default, when you plot only one curve, is to add confidence limits.

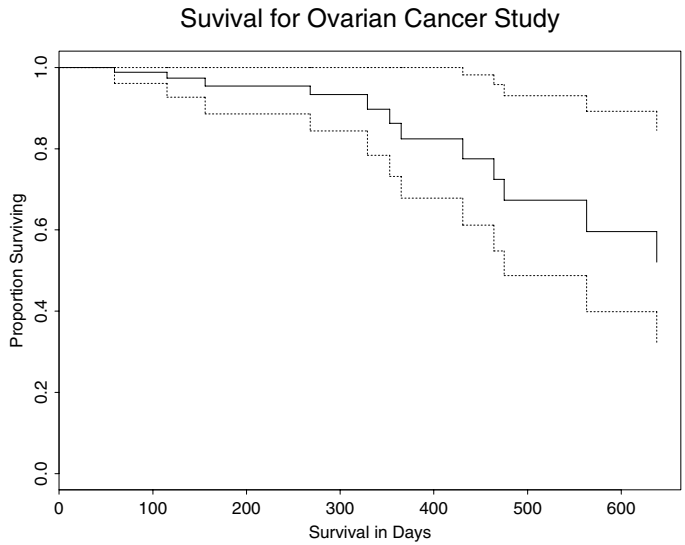

Figure 28.1: *Cox regression estimate of survival for a subject of average age (56.17 years), from the ovarian cancer study.*

HYPOTHESIS TESTS

Once you fit a Cox model, three tests of hypothesis are produced that are asymptotically equivalent, but are not always equivalent in practice. Let β_0 be the initial value of the coefficients and $\hat{\beta}$ the solution after fitting the model. The *likelihood ratio test* is defined as

$$2\{\log(L(\beta_0)) - \log(L(\hat{\beta}))\},$$

and is the most reliable. The *Wald statistic* is defined as

$$(\hat{\beta} - \beta_0)' \Sigma_{\hat{\beta}}^{-1} (\hat{\beta} - \beta_0),$$

where $\Sigma_{\hat{\beta}}^{-1}$. It is the estimated variance-covariance matrix, and is perhaps the most natural because it provides a per-variable test rather than an overall measure of significance. The *score test* is defined as

$$U^t I U,$$

where U is the vector of derivatives given by Equation (28.3) and I is the information matrix given by Equation (28.4), both evaluated at β_0. The score test does not require iteration and, consequently, is more computationally efficient if a large number of models are to be tested.

Example: Ovarian Cancer (cont.)

For the ovarian cancer example, you can compute all three tests by computing a summary of the resulting fit.

```
> summary(ov.fit1)

Call: coxph(formula = Surv(futime, fustat) ~ age, data =
ovarian)
  n= 26
      coef exp(coef) se(coef)  z        p
age 0.162      1.18   0.0497 3.25 0.0012
    exp(coef) exp(-coef) lower .95 upper .95
age      1.18      0.851      1.07       1.3

Rsquare= 0.423   (max possible= 0.932 )
```

```
Likelihood ratio test= 14.3 on 1 df,    p=0.000156
Wald test            = 10.6 on 1 df,    p=0.00116
Efficient score test = 12.3 on 1 df,    p=0.000463
```

The summary of a fit returns the *efficient score test*, in addition to the likelihood ratio test and Wald's test. Additionally, a confidence interval is estimated for the relative risk estimated by $e^{\hat{\beta}}$. To produce confidence limits with a different confidence level use the `conf.int` argument in the call to `summary`. For example, specifying `conf.int=0.99` produces 99% confidence intervals for the relative risk. It is clear that age is an important predictor of survival. Let's add the other predictors to the model:

```
> ov.fit2 <- coxph(Surv(futime, fustat) ~ age +
+ residual.dz + rx + ecog.ps, ovarian)
> ov.fit2

Call:
coxph(formula = Surv(futime, fustat) ~ age + residual.dz +
rx + ecog.ps, data = ovarian)

               coef exp(coef) se(coef)      z      p
age           0.125     1.133   0.0469  2.662 0.0078
residual.dz   0.826     2.285   0.7896  1.046 0.3000
rx           -0.914     0.401   0.6533 -1.400 0.1600
ecog.ps       0.336     1.400   0.6439  0.522 0.6000

Likelihood ratio test=17 on 4 df, p=0.0019 n= 26
```

To check for an overall improved fit over the `age`-only model, compute the likelihood ratio test between the models as follows:

```
> -2*(ov.fit1$loglik[2] - ov.fit2$loglik[2])

[1] 2.749708
```

The `loglik` component of the fit is a vector of the log-likelihoods for two fits. The null model (intercept only) is the first value, and the current model is the second value. Noting that there is a difference of three degrees of freedom between the models, the *p*-value for the likelihood ratio test is computed as follows:

```
> pchisq(2.75, df = 3)

[1] 0.5682029
```

There is no significant difference between the two models, indicating that `residual.dz`, `rx`, and `ecog.ps` don't improve the fit. Note that this approach will not work if there are missing values in the data.

STRATIFICATION

A simple extension of the Cox model is to allow multiple strata. The hazard for a subject contained in stratum j is then

$$\lambda(t,\ Z)\ =\ \lambda_j(t)\, e^{\beta Z(t)}.$$

When a variable is entered into the model as a stratification factor rather than as a covariate, it allows for nonproportional hazards to exist between levels of the variable. However, the disadvantage is that no β is available to estimate the effect of that variable. For instance, in a multi-center drug study the enrolling center is often entered into the model as a stratum variable. Because of different patient populations (for example, a higher proportion of acute cases), the centers may well have different shapes for their baseline survival curves. If modeled as a covariate, this nonproportionality could bias the estimate of the treatment effect.

Example: Ovarian Cancer (cont.)

You can stratify the ovarian cancer fit with respect to treatment, `rx`, still fitting `age` as a covariate, as follows:

```
> ov.fit3 <- coxph(Surv(futime, fustat) ~ age +
+ strata(rx), data = ovarian)
> survfit(ov.fit3)

Call: survfit(formula = ov.fit3)

      n events mean se(mean) median 0.95LCL 0.95UCL
rx=1 13      7  512     72.8    638     329      NA
rx=2 13      5  522     22.5     NA     475      NA
```

Printing the resulting fit displays the usual summary statistics for the survival curve for each stratum. Applying the `summary` function to the fit produces a more detailed table which includes the survival curve, standard errors and confidence intervals for each stratum.

```
> summary(survfit(ov.fit3))

Call: survfit(formula = ov.fit3)

                rx=1
 time n.risk n.event survival std.err
   59     13       1    0.978  0.0269
  115     12       1    0.950  0.0481
  156     11       1    0.910  0.0758
  268     10       1    0.862  0.1050
  329      9       1    0.736  0.1525
  431      8       1    0.625  0.1698
  638      5       1    0.341  0.2225

 lower 95% CI upper 95% CI
       0.9264            1
       0.8607            1
       0.7725            1
       0.6793            1
       0.4902            1
       0.3671            1
       0.0947            1

                rx=2
 time n.risk n.event survival std.err
  353     13       1    0.943  0.0558
  365     12       1    0.880  0.0812
  464      9       1    0.791  0.1125
  475      8       1    0.701  0.1318
  563      7       1    0.602  0.1460

 lower 95% CI upper 95% CI
        0.840        1.000
        0.735        1.000
        0.599        1.000
        0.485        1.000
        0.374        0.968
```

You can produce a plot of the stratified fit as follows:

```
> plot(survfit(ov.fit3), lty = 2:3)
> legend(100, 0.6, c("Treatment 1","Treatment 2"), lty=2:3)
> title("Ovarian Cancer Stratified by Treatment")
```

The result is displayed in Figure 28.2. The plot is one method of viewing a nonparametric estimate of treatment effect, after adjusting for possible differences in age distributions.

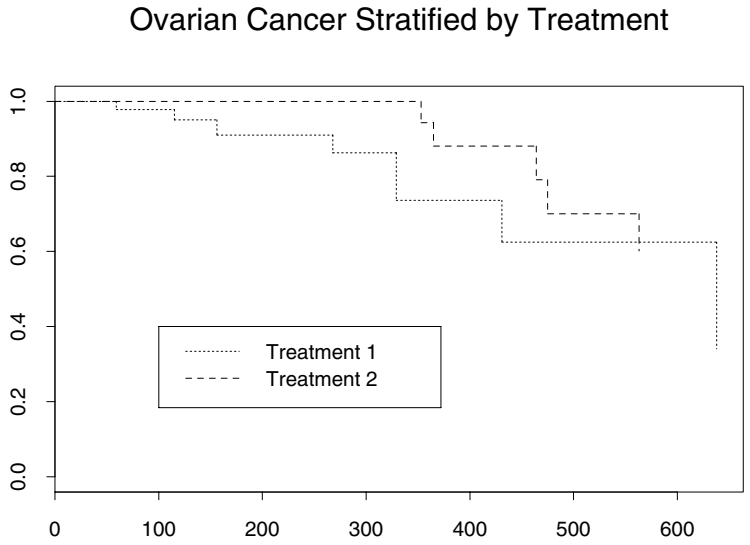

Figure 28.2: *A plot of the stratified fit of the ovarian cancer data adjusted for average age.*

RESIDUALS

The Breslow (or Tsiatis, Link, or Nelson-Aalen) estimate of the baseline hazard is

$$\hat{\Lambda}_0(\beta, t) = \int_0^t \frac{\sum_{i=1}^n dN_i(s)}{\sum_{i=1}^n Y_i(s)\, r_i(\beta, s}.$$

The martingale residual at time t is

$$M_i(t) = N_i(t) - \int_0^t r_i(\beta, s) Y_i(s) d\hat{\Lambda}_0(\beta, s. \qquad (28.5)$$

The residual is computed at $t = \infty$ and $\beta = \hat{\beta}$. If there are no time-dependent covariates, then $r_i(t)$ can be factored out of the integral, giving $\hat{M}_i = N_i - \hat{r}_i \hat{\Lambda}_0(\hat{\beta}, t_i)$. The deviance residual is a normalizing transform of the martingale residual

$$D_i = sign(\hat{M}_i) \cdot \sqrt{-\hat{M}_i - N_i \log((N_i - \hat{M}_i)/ N_i}.$$

The other two residuals are based on the score process $U_{ij}(b, t)$ for the ith subject and the jth variable:

$$R_{ij}(\beta, t) = \int_0^t (Z_{ij}(s) - Z_j(\beta, s)) d\hat{M}_i(\beta, s.$$

The score residual is defined as $U_{ij}(\hat{\beta}, \infty)$ for each subject and each variable (an n by p matrix). It is the sum of the score process over time. The usual score vector $U(\beta)$ (Equation (28.2)) is the column sum of the matrix of score residuals. The martingale and score residuals are integrals over time for a given subject. Specifically, in setting up a multiplicative hazards model, a single subject is represented by multiple lines of the input data, as though the subject was a set of different individuals observed over disjoint times. The residual for that person is the sum of the residuals for these "pseudo" subjects.

The Schoenfeld residuals (Schoenfeld, 1982) are defined as a matrix

$$s_{ij}(\beta) = Z_{ij}(t_i) - \bar{Z}_j(\beta, t_i) \tag{28.6}$$

with one row per death and one column per covariate, where i and t_i are the subject and the time that the event occurred. Note that the Schoenfeld residuals are related to the score process $U_{ij}(\beta, t)$. To see this, sum the score process over individuals to get a total score process $\sum_i U_{ij}(\beta, t) = U(\beta, t)$. This is just the score vector at time t, so that at $\hat{\beta}$ we must have $U(\hat{\beta}, 0) = U(\hat{\beta}, \infty) = 0$. Because $\hat{\Lambda}$ is discrete, our estimated score process is also discrete, having jumps at each of the unique death times. There are two simplifying identities for these residuals:

$$\begin{aligned} U(\beta, t) &= \sum_i \int_0^t Z_{ij}(s)\, dM_i(\beta, s) \\ &= \sum_i \int_0^t (Z_{ij}(s) - \bar{Z}_j(\beta, s))\, dN_i(s) \end{aligned} \tag{28.7}$$

Note that $d\hat{M}_i(t)$ is zero when subject i is not in the risk set at time t. Since the sums are the same for all t, each increment of the processes must be the same as well. Comparing the second of these to Equation (28.6), we see that the Schoenfeld residuals are the increments or jumps in the total score process.

There is a small nuisance with tied death times: under the integral formulation the *O-E* process has a single jump at each death time, leading to one residual for each unique event time, while under the Schoenfeld representation there is one residual for each event. In practice, the latter formulation has been found to work better for both plots and diagnostics, as it leads to residuals that are approximately equivariant. For the alternative of one residual per *unique* death time, both the size and variance of the residual is proportional to the number of events.

The last and most general residual is the entire score process R_{ijk}, where i indexes subjects, j indexes the covariates, and k indexes the event times:

$$R_{ijk} = [Z_{ij}(t_k) - \bar{Z}_j(t_k)][d(N_i(t_k) - r_i(t_k)d\hat{\Lambda}_0(t_k))]$$

The score and Schoenfeld residuals are the marginal sums of this array. Lin, Wei and Ying (1992) suggest a global test of the proportional hazards model based on the maximum of the array.

Uses for the Residuals

Four possible uses of residuals are addressed in this section.

1. Discovering the correct functional form for a predictor.
2. Identifying subjects who are poorly predicted by the model.
3. Identifying influential points (points with high leverage).
4. Assessing the proportional hazards assumption.

Discovering the Functional Form for a Predictor

The martingale residual, M_i, is given by Equation (28.5) evaluated at $t = \infty$. Assume that the true functional form for a covariate in the exponent is $h(Z)$. Then Therneau, Grambsch, and Fleming (1990) show that the martingale residuals, after regression on the other variables, satisfy

$$E(M_i) \equiv (h(t) - \bar{h})E(N_i).$$

A smoothed plot of the M_i versus x gives an approximate image of the true functional form, with the y-axis scaled by a constant that depends on the proportion of censoring. If there are several covariates, the martingale residuals from a model with all covariates except Z_1, for example, can be plotted against the residuals of a regression of Z_1 on the others. This is similar to the *adjusted variable plots* for the linear model in Chambers, Cleveland, Kleiner, and Tukey (1983).

Another use is to plot the residuals from a null model, that is, with `iter.max=0`, against each predictor. This is roughly equivalent to the standard scatter plots of y against each Z that is used for uncensored

data, before a model is fit. Addition of a local regression smooth curve using `loess` gives, in both cases, a first approximation to what transformations, if any, might be appropriate for each Z.

Identifying Poorly Predicted Subjects

The martingale residuals can be highly skewed. The deviance residual, d_i, is a normalized transform of M_i, and can be used to identify individuals who are poorly predicted by a model. However, you should exercise extreme caution when using deviance residuals in analysis. Recent experience has shown that deviance residuals cannot be recommended in all cases. For more details, see Therneau, *et al.* (1990).

Identifying Influential Points

In a linear model, the influence of a point on the fit depends on both its residual and its distance from the center of the predictor space, roughly $\text{resid}_i \cdot (Z_i - \bar{Z})$. In a Cox model, the mean of the covariates changes over time as subjects leave the risk set, which suggests an average of some sort. The score residuals are a decomposition of the first derivative or score vector; large values indicate a point with high leverage. In particular, $-I^{-1}L_i$, where I^{-1} is the Cox model variance matrix, is approximately the change that would occur in β if observation i were dropped from the model. These changes in β are returned when you specify `type="dfbeta"` or `type="dfbetas"` to the `residuals` function.

Assessing the Proportional Hazards Assumption

The Schoenfeld residuals are increments in time for the total score process; see Equation (28.6). If the proportional hazards assumption holds, the Schoenfeld residuals should be a random walk. Conversely, assume that some variable, such as treatment, has a large positive effect early but that the effect trails off. The treatment might influence how many patients survive to some point t, but once they are "cured" it has no influence on survival beyond t. In this case, proportional hazards does not hold and the fitted models underestimate the true treatment effect for small t, and overestimate it for large t. If treatment has a beneficial effect ($\beta < 0$), the Schoenfeld residuals have an early negative trend followed by a late positive trend. Harrell (1986) suggests using the correlation of `rank(time)` with this residual as a test for nonproportional hazards. Therneau, *et al.* (1990) use the maximum of the absolute cumulative summed Schoenfeld residual, a

Kolmogorov-type test. Grambsch and Therneau (1994) further show that a rescaled Schoenfeld residual can correct for correlation among the covariates and be more interpretable. This result is the basis for the `cox.zph` function.

Example: Lung Cancer

This example examines data from a study of lung cancer patients conducted by the North Central Cancer Treatment Group. The `lung` data frame includes the usual survival times (`time`) and indicator variable of death or censoring (`status`) plus the following additional variables on each patient.

- `inst`: A numeric code for the institution at which the patient was hospitalized.
- `age`: Patient's age.
- `sex`: Sex of the patient. Possible values are 1 for males and 2 for females.
- `ph.ecog`: Physician's estimate of the ECOG performance score (0-4).
- `ph.karno`: Physician's estimate of the Karnofsky score, a competitor to the ECOG performance score.
- `pat.karno`: Patient's assessment of his/her Karnofsky score.
- `meal.cal`: Calories consumed at meals excluding beverages and snacks.
- `wt.loss`: Weight loss in the last 6 months.

A summary of the `lung` data frame follows:

```
> summary(lung)

      inst            time            status
 Min.: 1.00      Min.:   5.0      Min.:1.000
1st Qu.: 3.00   1st Qu.: 166.8   1st Qu.:1.000
 Median:11.00    Median: 255.5    Median:2.000
   Mean:11.09      Mean: 305.2      Mean:1.724
3rd Qu.:16.00   3rd Qu.: 396.5   3rd Qu.:2.000
   Max.:33.00      Max.:1022.0      Max.:2.000
   NA's: 1.00

      age             sex          ph.ecog
```

```
   Min.:39.00      Min.:1.000      Min.:0.0000
1st Qu.:56.00   1st Qu.:1.000   1st Qu.:0.0000
 Median:63.00    Median:1.000    Median:1.0000
   Mean:62.45      Mean:1.395      Mean:0.9515
3rd Qu.:69.00   3rd Qu.:2.000   3rd Qu.:1.0000
   Max.:82.00      Max.:2.000      Max.:3.0000
                                   NA's:1.0000

    ph.karno        pat.karno         meal.cal
   Min.: 50.00      Min.: 30.00      Min.:  96.0
1st Qu.: 75.00   1st Qu.: 70.00   1st Qu.: 635.0
 Median: 80.00    Median: 80.00    Median: 975.0
   Mean: 81.94      Mean: 79.96      Mean: 928.8
3rd Qu.: 90.00   3rd Qu.: 90.00   3rd Qu.:1150.0
   Max.:100.00      Max.:100.00      Max.:2600.0
   NA's:  1.00      NA's:  3.00      NA's:  47.0

    wt.loss
   Min.:-24.000
1st Qu.:  0.000
 Median:  7.000
   Mean:  9.832
3rd Qu.: 15.750
   Max.: 68.000
   NA's: 14.000
```

Note that the `status` variable takes values 1 (censoring) and 2 (event), as does the `sex` variable. The `coxph` function recognizes either a 0/1 or a 1/2 binary variable as an indicator of censored/event status, so you needn't transform the `status` variable in this case. Let's start the example by fitting a model on all the variables stratified by `sex`.

```
> lung.fit1 <- coxph(Surv(time, status) ~ strata(sex) +
+ age + ph.ecog + ph.karno + pat.karno + meal.cal +
+ wt.loss, data = lung, na.action = na.exclude)
```

```
> lung.fit1

Call: coxph(formula = Surv(time, status) ~ strata(sex) +
              age + ph.ecog + ph.karno + pat.karno +
              meal.cal + wt.loss, data = lung,
              na.action = na.exclude)
               coef exp(coef) se(coef)     z      p
age         9.05e-03     1.009 0.011601  0.78 0.4400
ph.ecog     7.07e-01     2.029 0.222773  3.17 0.0015
ph.karno    2.07e-02     1.021 0.011282  1.84 0.0660
pat.karno -1.33e-02     0.987 0.008050 -1.65 0.0980
meal.cal  -5.27e-06     1.000 0.000263 -0.02 0.9800
wt.loss   -1.52e-02     0.985 0.007890 -1.93 0.0540

Likelihood ratio test=21.6 on 6 df, p=0.00145 n=168
   (60 observations deleted due to missing values)
```

The resulting fit indicates that `age` and `meal.cal` are not important predictors of survival, so we drop them from the model:

```
> lung.fit2 <- coxph(Surv(time, status) ~ strata(sex) +
+ ph.ecog + ph.karno + pat.karno + wt.loss, data = lung,
+ na.action = na.exclude)
> lung.fit2

Call:
coxph(formula = Surv(time, status) ~ strata(sex) +
        ph.ecog + ph.karno + pat.karno + wt.loss,
        data = lung, na.action = na.exclude)
             coef exp(coef) se(coef)     z p
ph.ecog    0.6495     1.915  0.20070  3.24 0.0012
ph.karno   0.0173     1.017  0.01031  1.68 0.0930
pat.karno -0.0167     0.983  0.00726 -2.30 0.0220
wt.loss   -0.0137     0.986  0.00691 -1.99 0.0470
Likelihood ratio test=25.7 on 4 df, p=3.61e-05 n=210
   (18 observations deleted due to missing values)
```

Because of the different number of missing values for these two models, you cannot compare them directly using a likelihood ratio, as we did for the `ovarian` data.

Assessing Functional Form

We now look at the functional form of the relationship with respect to each of the important predictors in the model. We do this by plotting the martingale residuals from a model with the variable of interest

removed, versus the variable of interest. We then add a `loess` smooth line to estimate the relationship. You can accomplish both the plot and the smooth using the `scatter.smooth` function. To make the handling of `NA`s a bit easier, begin by creating a new data frame with just the variables in the model and with the `NA`s removed.

```
> nlung <- na.exclude(lung[, c("time", "status", "sex",
+ "ph.ecog", "ph.karno", "pat.karno", "wt.loss")])
```

Note that the 18-row difference between the two data frames is confirmed by the number of `NA`s that were deleted in fitting `lung.fit2`:

```
> dim(nlung)

[1] 210 7

> dim(lung)

[1] 228 10
```

The four plots displayed in Figure 28.3 show the estimated relationships for each predictor.

```
> par(mfrow = c(2,2))
> attach(nlung)
> fit1 <- coxph(Surv(time, status) ~ strata(sex) +
+ ph.karno + pat.karno + wt.loss, data = nlung)
> scatter.smooth(ph.ecog, resid(fit1))

> fit2 <- coxph(Surv(time, status) ~ strata(sex) +
+ ph.ecog + pat.karno + wt.loss, data = nlung)
> scatter.smooth(ph.karno, resid(fit2))

> fit3 <- coxph(Surv(time, status) ~ strata(sex) +
+ ph.ecog + ph.karno + wt.loss, data = nlung)
> scatter.smooth(pat.karno, resid(fit3))

> fit4 <- coxph(Surv(time, status) ~ strata(sex) +
+ ph.ecog + ph.karno + pat.karno, data = nlung)
> scatter.smooth(wt.loss, resid(fit4))
```

All of the relationships look reasonably linear.

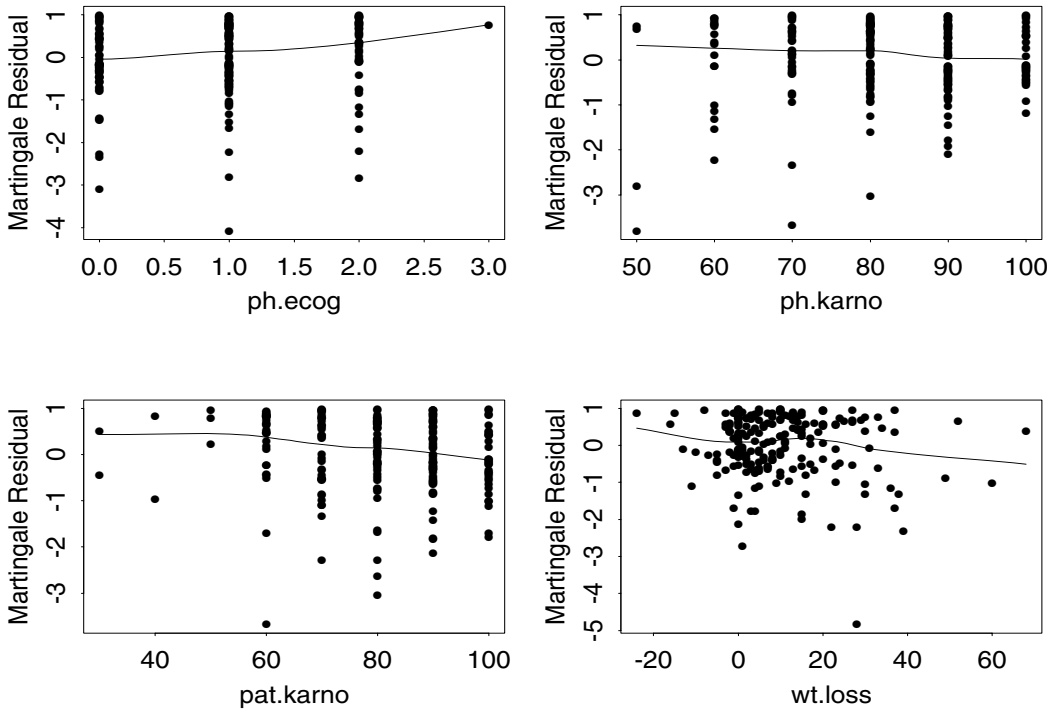

Figure 28.3: *Plots of the martingale residuals for four models with each variable in turn left out of the model for the lung cancer study.*

Poorly Predicted Subjects

Subjects with large deviance residuals are poorly predicted by the model. You produce the deviance residual plot for the second lung cancer model as follows:

```
> plot(resid(lung.fit2, type = "deviance"))
```

Figure 28.4 displays the resulting plot. There are no wildly deviant observations.

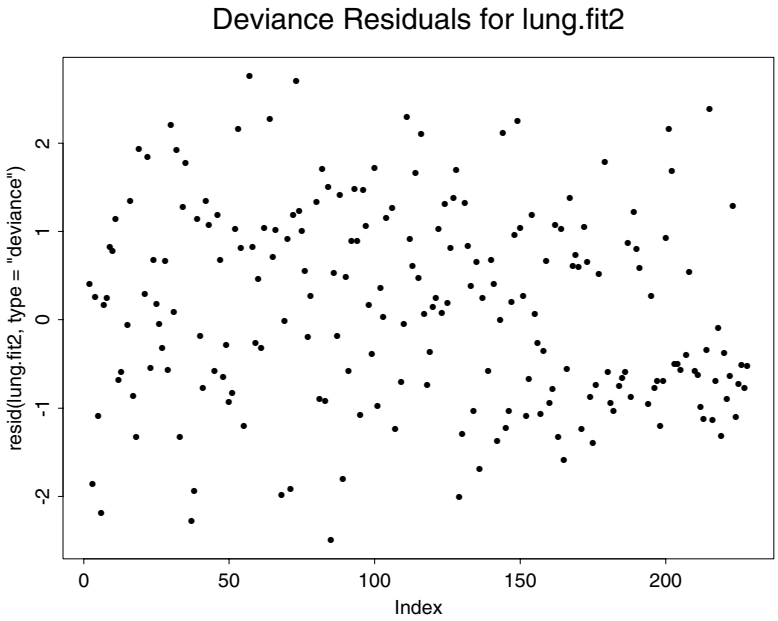

Figure 28.4: *Plots of the deviance residuals for model* `lung.fit2` *of the lung cancer study.*

Influence

Another set of plots examines the influence of individual observations on the parameter estimates. Use the changes in the estimated scaled coefficient due to dropping each observation from the fit (`type="dfbetas"`) as a measure of influence. The first of the four plots is created as follows:

```
> bresid <- resid(lung.fit2, type = "dfbetas")
> plot(1:228, bresid[,1], type = "h",
+ ylab = "Scaled change in coef",
+ xlab = "Observation")
> title("ph.ecog")
```

The remaining plots are created by selecting the appropriate columns of `bresid` and changing labels on the plots. The resulting plots are displayed in Figure 28.5. Note the use of `1:228` to generate the indices for the observations even though the fit had only 210 observations after deleting missing values. The dimension of `bresid` is 228 x 4. The number of rows matches that of `lung` because the `naresid` method for omitting missing values (`na.exclude`) inserts `NA`s in the residual matrix returned.

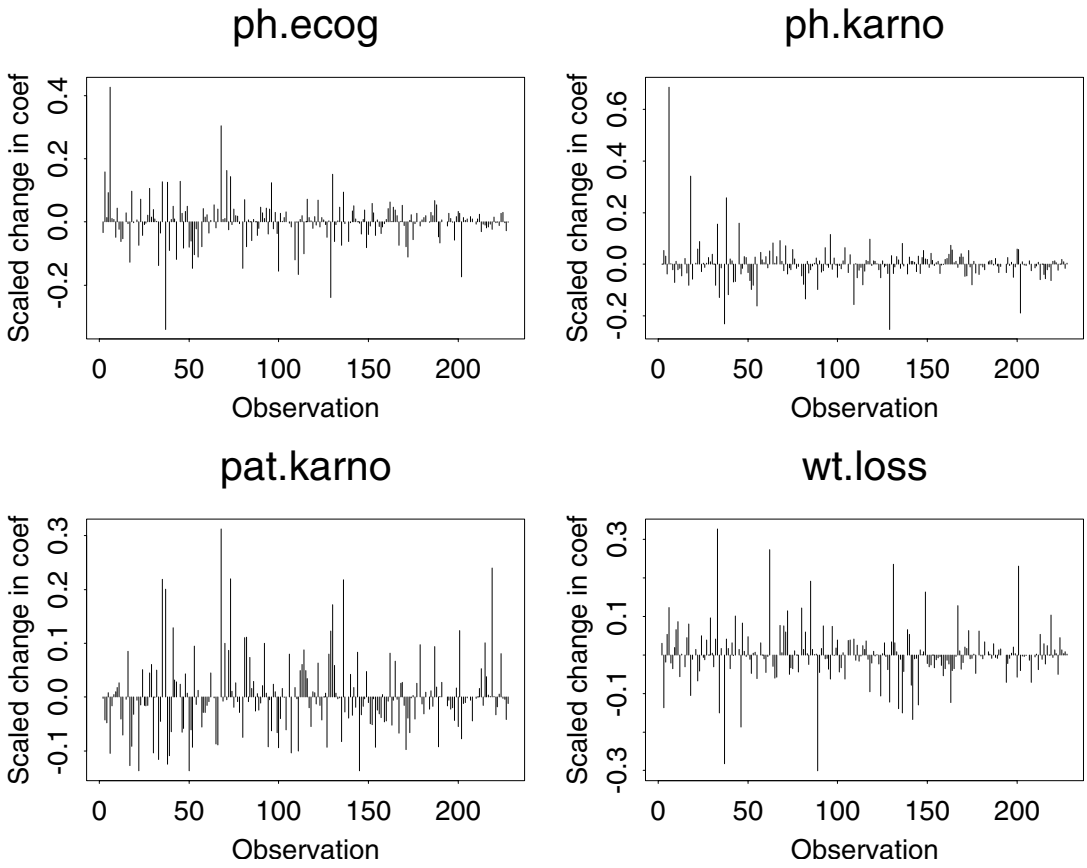

Figure 28.5: *A plot of influence by observation number for the four important predictors for the lung cancer study.*

The largest change in a regression coefficient is 0.6 standard errors of the coefficient for `ph.karno` (upper right corner plot). Since the coefficient for `ph.karno` is marginally significant at best, you need not worry much about this observation. The other plots are reasonable.

Assessing Proportional Hazards

You can examine the assumption of proportional hazards both graphically and statistically for the `lung.fit2` model. The plot in Figure 28.6 is produced as follows:

```
> plot(cox.zph(lung.fit2))
```

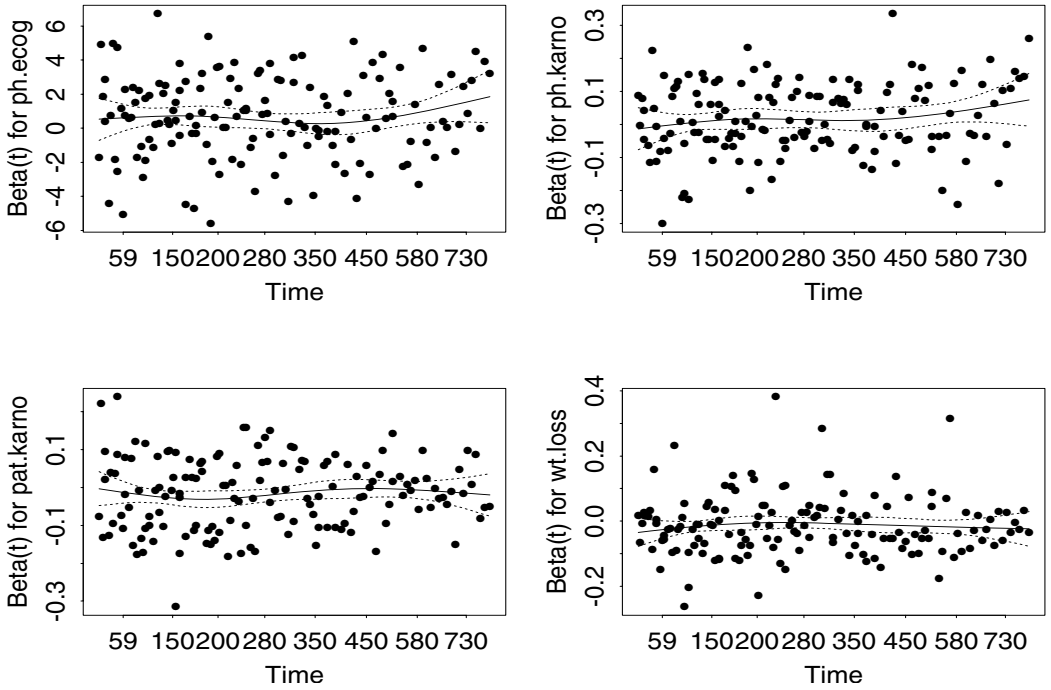

Figure 28.6: *A plot of the rescaled Schoenfeld residuals to assess the proportional hazards assumption for four covariates in lung cancer study.*

All of the smooth curves are flat, indicating proportional hazards is a reasonable assumption. Statistical tests for significant slope in the scatter plots of Figure 28.6 support the interpretation of the graphical displays.

```
> cox.zph(lung.fit2)

              rho  chisq     p
  ph.ecog 0.05189 0.3905 0.532
 ph.karno 0.14216 2.2081 0.137
pat.karno 0.04773 0.3812 0.537
  wt.loss 0.00857 0.0131 0.909
   GLOBAL      NA 4.4476 0.349
```

Plotting the Resulting Fit

Finally, you can plot estimated survival curves for the `lung.fit2` model as follows:

```
> plot(survfit(lung.fit2), lty = 2:3)
> legend(500, .9, c("Male", "Female"), lty = 2:3)
> title("Survival for Male and Female Patients
Continue string: \nwith Average Covariates")
```

The fitted Cox models are presented in Figure 28.7.

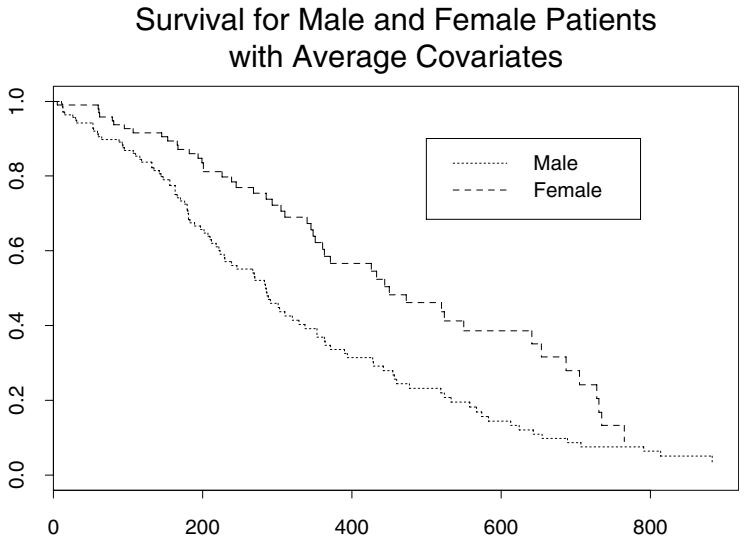

Figure 28.7: *Cox regression estimation of baseline survival curves for a sample of lung cancer patients.*

Recall that the model was stratified on `sex`. The resulting survival curves are for two pseudo patients (a male and a female) with average values for each of `ph.ecog`, `ph.karno`, `pat.karno`, and `wt.loss`.

USING THE COUNTING PROCESS NOTATION

The Anderson-Gill formulation of the proportional hazards model as a counting process is useful not only theoretically but also in the practice of fitting models. From a data analysis point of view, each subject is treated as an observation of a (very slow) Poisson process. A censored subject is thought of not as incomplete data, but as one whose event count is still zero. Time-dependent covariates effect the rate for upcoming events, and can depend in any way on past observation of the subject. Furthermore, intervals of observation need not be contiguous.

Organizing data in this framework has advantages. Each subject is represented by a set of observations: $_{ij}$, t_{ij}, δ_{ij}, x_{ij}, k, $j = 1, ..., n$, where $(s_{ij}, t_{ij}]$ is an interval of risk, open on the left and closed on the right. The term δ_{ij} is equal to 1 if the subject had an event at time t_{ij}, x_{ij} is the covariate vector over the interval, and k_{ij} is the stratum the subject belongs to during the interval. Data sets like this are easy to construct in Spotfire S+. Following are a few specific examples to aid in constructing the analysis data frame.

Multiple Events

This example comes from a study of myocardial infarction (heart attack) patients where one of the events of interest is fatal or nonfatal re-infarction. Several patients had multiple events. The maximum number of events was three. Analysis was done using the counting process formulation by breaking any patient with multiple events into multiple intervals of risk. For example, one patient had infarctions on days 100 and 185 and was followed until day 250. This patient had three rows of data with time intervals (0, 100], (100, 185], and (185, 250] and corresponding event status codes of 1, 1, and 0.

Time-Dependent Covariates

The most common type of time-dependent covariates are repeated measurements on a subject or a change in the subject's treatment. Both of these situations are easily handled by the counting process formulation. As an example consider the Stanford heart transplant study, where treatment is a time-dependent covariate. Suppose there are two patients whose time from enrollment to death is 102 and 343

days, respectively, and that the second patient had a heart transplant 21 days after enrollment. The data for these two patients displayed are in Table 28.1.

Table 28.1: *Data for two hypothetical patients in the Stanford heart transplant study.*

Interval	Status	Transplant	Age	Prior Surgery
(0, 102]	1	0	41	0
(0, 21]	0	0	48	1
(21, 343]	1	1	48	0

The static covariates such as `age` and `surgery` are repeated over the multiple rows for a given patient. A minor modification is needed when there is a tie between the event or censoring time and the time at which a time-dependent covariate changes value. In this case, decrease the time for the time-dependent covariate slightly so it precedes the event or censoring time. For a patient who is transplanted and dies on day 5, the transplant time is set to 4.9 and the death is recorded at 5. Multiple test results are easily coded as well. For a patient with tests on days 0, 60, and 120, and follow-up to day 140, the data are coded as three time intervals, 0-60, 60-120, and 120-140. This implicitly assumes that the time-dependent covariate is a step function with jumps at the measurement points. Alternatively, you can break at the midpoints between the measurement times or interpolate the test measurements over smaller intervals of time. If test results vary markedly from visit to visit, interpolation of the measurements or redesign of the study may be required.

Discontinuous Intervals of Risk

In a study of tumor progression and its relationship to a particular blood marker, the key time-dependent variable is the monthly measurement of the marker. A few patients, however, had gaps in their visit record. One choice for analysis is to interpolate these patients' values over the missing time periods. An alternate, more conservative course is to treat the values on the marker as missing. This strategy effectively removes these subjects from the risk set for the missing visit times, but they are not removed entirely from the study.

Another application of discontinuous risk intervals results when multiple events are possible, but the treatment for an event temporarily protects the patient from another event. In the study of hip-fracture in the elderly, hospitalization following a fracture protects the patient from further fractures. For studies with low event rates, discontinuous risk intervals probably have little impact on the analysis.

Multiple Time Scales

The usual Cox model forms risk groups based on time since entry. For some studies a more logical grouping might be based on another alignment, such as age or time since diagnosis. An example is with Parkinson's disease patients. Natural history of the disease suggests that risk groups be based on the time since diagnosis. The Mayo Clinic is a referral center and frequently receives such patients sometime after diagnosis. Using the counting process formulation, the interval for a referred patient who is enrolled one year after diagnosis and has an event in the second year is (1, 2]. This patient is not in the risk set for an early enrollee with an event at six months. The risk set for the event at two years is all subjects. This is known as *left truncation.*

Time-Dependent Strata

Another case where alignment is a potential issue concerns time-dependent strata. The example is a study of Dutch patients with primary biliary cirrhosis of the liver (PBC), which is a rare but fatal chronic liver disease of unknown cause. The hazard rate for patients with the disease grows over time, as does the rate of degeneration in their hepatic function, tracked by various blood tests. A portion of the patients receive a liver transplant at some point during the follow-up. One objective of the study was to assess the value of covariates such as age and bilirubin in predicting patient outcome, both before and after transplantation. The transplant was treated as a time-dependent stratification variable. In the post-transplant strata, the most natural hazard function is based on time since transplant. Surgical death is a major risk for such an extensive procedure, and this time scale properly aligns the patient's clock with the dominating hazard.

Proper alignment for time-dependent strata is not always clear. One appealing method of analysis for the myocardial infarction study is to place patients into new strata after each cardiac event. The baseline hazard for a patient with multiple events may be quite different than the group as a whole. It is not obvious, however, whether time since enrollment or time since last event is the better index of an appropriate risk group.

MORE DETAILED EXAMPLES

Complex Cox models usually involve time-dependent data, which is handled by using the counting process notation developed by Andersen and Gill (1982). For a technical reference see Fleming and Harrington (1991). The examples in this section involve time-dependent variables in some way. In the Stanford heart transplant example, the time dependency is on a binary covariate indicating whether a patient has had a heart transplant. For patients who received heart transplants during the study, the `transplant` variable changes. The second example involves a bladder cancer study for patients with multiple occurrences of bladder tumors. The multiple events are modeled using the counting process notation and an additional notion of correlated responses.

Stanford Heart Transplant Study

The example below reproduces an analysis of the Stanford Heart Transplant Study found in Kalbfleisch and Prentice (1980), section 5.5.3. The data itself are taken from Crowley and Hu (1977), because the values listed in the appendix of Kalbfleisch and Prentice are rounded and do not reproduce the results of their section 5.5. The covariates in the study, contained in the `heart` data frame, are described as follows.

- `transplant`: Binary variable indicating whether the patient received a heart transplant (1) or not (0)
- `age`: (Age at acceptance in days)/365.25 - 48
- `year`: (Date of acceptance in days since 1 Oct 1967)/365.25
- `surgery`: Binary variable indicating whether the patient had prior surgery (1 = yes, 0 = no)

The `transplant` variable is the only time-dependent variable. From the time of admission into the study until the time of death, a patient was eligible for a heart transplant. The time to transplant depends on the next available donor heart with an appropriate tissue-type match.

In the `heart` data frame, a transplanted patient is represented by two rows of data. The first row is over the time period from enrollment (time 0) until the transplant, and has `transplant=0`. The second row is over the period from transplant to death or last follow-up, and has `transplant=1`. All other covariates are the same on the two lines.

Subjects without a transplant are represented by a single row of data. Each row of data contains two variables `start` and `stop` that mark the time interval `(start, stop]` for the data. Each row also has an indicator variable `event` that is 1 if there was a death at time `stop` and 0 otherwise. For example, a subject who was transplanted at day 10 and followed up until day 31 has a first row of data corresponding to the time interval (0, 10] and a second row corresponding to the interval (10, 31].

Below is the Spotfire S+ code used to fit the six models found in Kalbfleisch and Prentice. Note the use of the `options` call, which forces the factors to be coded as dummy variables; see the help file on `contr.treatment` for more details. Since the data set contains tied death times, you must use the Breslow approximation to match the coefficients that Kalbfleisch and Prentice produce. See the section Computations for Tied Deaths for more details on methods for handling ties.

```
> options(contrasts = c("contr.treatment", "contr.poly"))
> heart.fit1 <- coxph(Surv(start, stop, event) ~
+ (age + surgery)*transplant,
+ data = heart, method = "breslow")

> heart.fit2 <- coxph(Surv(start, stop, event) ~
+ year * transplant,
+ data = heart, method = "breslow")

> heart.fit3 <- coxph(Surv(start, stop, event) ~
+ (age + year) * transplant,
+ data = heart, method = "breslow")

> heart.fit4 <- coxph(Surv(start, stop, event) ~
+ (year + surgery) * transplant,
+ data = heart, method = "breslow")

> heart.fit5 <- coxph(Surv(start, stop, event) ~
+ (age + surgery)*transplant + year,
+ data = heart, method = "breslow")

> heart.fit6 <- coxph(Surv(start, stop, event) ~
+ age * transplant + surgery + year,
+ data = heart, method = "breslow")
```

A summary of the first fit produces the following:

```
> summary(heart.fit1)

Call:
coxph(formula = Surv(start, stop, event) ~ (age + surgery)
* transplant, data = heart, method = "breslow")

n= 172

                     coef   exp(coef) se(coef)      z
               age  0.0138     1.014   0.0181  0.763
           surgery -0.5457     0.579   0.6109 -0.893
       transplant  0.1181     1.125   0.3277  0.360
    age:transplant  0.0348     1.035   0.0273  1.276
surgery:transplant -0.2916     0.747   0.7582 -0.385

                       p
               age 0.45
           surgery 0.37
       transplant 0.72
    age:transplant 0.20
surgery:transplant 0.70

                   exp(coef) exp(-coef) lower .95
               age     1.014      0.986     0.979
           surgery     0.579      1.726     0.175
       transplant     1.125      0.889     0.592
    age:transplant     1.035      0.966     0.982
surgery:transplant     0.747      1.339     0.169
                   upper .95
               age      1.05
           surgery      1.92
       transplant      2.14
    age:transplant      1.09
surgery:transplant      3.30

Rsquare= 0.07   (max possible= 0.969 )
Likelihood ratio test= 12.4 on 5 df,   p=0.0291
Wald test            = 11.6 on 5 df,   p=0.0402
Efficient score test = 12 on 5 df,   p=0.0345
```

The summary indicates that `n=172`. This is the number of observations in the study, not the number of subjects. There are actually 103 patients, of which 69 had a transplant and are thus represented with 2 rows of data.

You can create a table of coefficients similar to Kalbfleisch and Prentice's table 5.2 as follows:

```
> var.names <- c("age", "year", "surgery", "transplant",
+ "age:transplant", "year:transplant",
+ "surgery:transplant")

> round(rbind(heart.fit1$coef[var.names],
+ heart.fit2$coef[var.names], heart.fit3$coef[var.names],
+ heart.fit4$coef[var.names], heart.fit5$coef[var.names],
+ heart.fit6$coef[var.names]), digits = 4)

       age   year surgery transplant age:transplant
[1,] 0.014     NA  -0.546      0.118          0.035
[2,]    NA -0.265      NA     -0.282             NA
[3,] 0.016 -0.274      NA     -0.588          0.034
[4,]    NA -0.254  -0.236     -0.292             NA
[5,] 0.015 -0.136  -0.419      0.077          0.027
[6,] 0.015 -0.136  -0.621      0.047          0.027
     year:transplant surgery:transplant
[1,]              NA             -0.292
[2,]           0.136                 NA
[3,]           0.201                 NA
[4,]           0.164             -0.550
[5,]              NA             -0.298
[6,]              NA                 NA
```

When there are time-dependent covariates, the predicted survival curve can present something of a dilemma. The usual call to `survfit` is for a *pseudo cohort* whose covariates do not change:

```
> heart.surv1 <- survfit(heart.fit2,
+ data.frame(year = 2, transplant = 0) )

> heart.surv2 <- survfit(heart.fit2,
+ data.frame(year = 2, transplant = 1) )
```

The second curve, `heart.surv2`, represents a cohort of patients whose `transplant` variable is always 1, even on day 0 (that is, patients who had no waiting time for a transplant). There were none of these in the study, so just what does it represent? Time-dependent covariates that represent repeated measurements on a patient, such as a blood enzyme level, are particularly problematic. With time-dependent covariates, it is easy to create predicted survival curves for "patients" that never would or perhaps never could exist.

Because the model depends on the time-dependent covariate, `transplant`, a proper predicted survival requires specification of a *future covariate history* for the patient in question. See the discussion of *internal* and *external* covariates in section 5.3 of Kalbfleisch and Prentice for a more complete exposition on these issues.

It is possible to obtain the projected survival for some particular pattern of change in the covariates by supplying a multiple-line data frame that reflects that pattern, and then setting `individual=T`. The example below produces the survival curve for a cohort aged 50 with prior surgery and a transplant at 6 months. That is, over the time interval (0, .5] the covariate set is (50, 1, 0), and over the time interval (.5, 3] it is (50, 1, 1). Note that `start` and `stop` times are in days rather than years. In order to specify the time points, the failure time variables `start`, `stop`, and `event`, must be specified in the data frame as well as the covariates, though the value for `event` will be ignored.

```
> newdata <- data.frame(start=c(0,183), stop=c(183,3*365),
+ event=c(1,1), age=c(50,50), surgery=c(1,1),
+ transplant=c(0,1))
> survfit(heart.fit1, newdata, individual=T)
```

Bladder Cancer Study

This example is taken from the paper by Wei, Lin, and Weissfeld (1989). The study is of time to recurrence of bladder cancer, and the data are contained in the `bladder` data frame. The data fram `bladder` has either 4 or 5 rows for each subject. Many subjects had recurrences, sometimes as many as four, and were followed beyond the fourth recurrence. The variables in `bladder` are defined as follows.

- `id`: Patient ID
- `rx`: Treatment group (1 = placebo, 2 = thiopeta)
- `size`: Size of the largest initial tumor

- `number`: The number of initial tumors
- `start`: Entry into the study or the time of last recurrence
- `stop`: Time to event (months)
- `event`: Indicator of cancer recurrence (1) or censoring (0)
- `enum`: Number of recurrences of bladder cancer

A summary of `bladder` follows:

```
> summary(bladder)
```

```
       id              rx            number
 Min.: 1.00      Min.:1.000      Min.:1.000
1st Qu.:22.75   1st Qu.:1.000   1st Qu.:1.000
 Median:43.00    Median:1.000    Median:1.000
   Mean:43.18      Mean:1.443      Mean:2.145
3rd Qu.:64.00   3rd Qu.:2.000   3rd Qu.:3.000
   Max.:85.00      Max.:2.000      Max.:8.000

      size           start            stop
   Min.:1.000      Min.: 0.00      Min.: 1.00
1st Qu.:1.000   1st Qu.: 1.00   1st Qu.:13.00
 Median:1.000    Median:15.00    Median:25.00
   Mean:1.997      Mean:18.03      Mean:25.73
3rd Qu.:3.000   3rd Qu.:29.00   3rd Qu.:38.00
   Max.:7.000      Max.:59.00      Max.:64.00

     event            enum
   Min.:0.0000      Min.:1.000
1st Qu.:0.0000   1st Qu.:2.000
 Median:0.0000    Median:3.000
   Mean:0.3182      Mean:2.585
3rd Qu.:1.0000   3rd Qu.:4.000
   Max.:1.0000      Max.:5.000
```

We create two data frames for analysis. The first one has only the first four rows for each subject and has `start` removed.

```
> bladder1 <- bladder[bladder$enum < 5, ]
> bladder1$start <- NULL
```

The second one has removed all rows for which `start` and `stop` are equal.

```
> bladder2 <- bladder[bladder$start < bladder$stop, ]
```

Wei, *et al.* fit four separate models, one for each recurrence, and then combined the results. The first of the individual fits is based on time from the start of the study until the first event for all patients. The second fit is based on time from the start of the study until the second event again for all patients, and likewise for the third and fourth fits. The model estimated by Wei, *et al*. is fit by the following commands. The key to the model is the `cluster(id)` term, which asserts that subjects with the same value of the variable `id` may be correlated. To compare the results directly to Wei, *et al.*, we first set the factor contrasts to `"contr.treatment"`.

```
> options(contrasts = c("contr.treatment","contr.poly"))
```

We can now fit the model as follows:

```
> wfit <- coxph(Surv(stop, event) ~ (rx + size + number) *
+ strata(enum) + cluster(id), data = bladder1,
+ method = "breslow")

# Coefficients for the treatment effect
> rx <- c(1,4,5,6)
# Contrast matrix
> cmat <- diag(4); cmat[,1] <- 1
# Coefs in WLW (table 5)
> cmat %*% wfit$coef[rx]

           [,1]
[1,] -0.5175702
[2,] -0.6194396
[3,] -0.6998691
[4,] -0.6504161

> wvar <- cmat %*% wfit$var[rx,rx] %*% t(cmat)
# Var matrix from WLW (eqn 3.2)
> sqrt(diag(wvar))

[1] 0.3075006 0.3639071 0.4151602 0.4896743
```

The same coefficients can also be obtained, as Wei, *et al.* do, by performing four separate fits, but it takes more work. A major advantage to fitting the model as above is that it allows us to fit submodels that are not available using separate fits for each stratum. In particular, the model

```
> coxph(Surv(stop, event) ~ rx + (size + number) *
+ strata(enum) + cluster(id), data = bladder1,
+ method = "breslow")
```

differs only in that there is no treatment by strata interaction. It gives an average treatment coefficient of -0.60, which is near to the weighted average of the marginal fits (based on the diagonal of `wvar`) suggested by Wei, *et al.*.

Wei, *et al.* also give the results for two suggestions proposed by Prentice, Williams, and Peterson (1981). For time to first event, these are the same as above. For the second event they use only patients who experienced at least one event, and use either the time from start of study (method a) or the time since the occurrence of the last event (method b). The Spotfire S+ commands for these are as follows:

```
> fit2pa <- coxph(Surv(stop, event) ~ rx + size + number,
+ data = bladder2, subset = (enum == 2))
> fit2pb <- coxph(Surv(stop - start, event) ~ rx + size +
+ number, data = bladder2, subset = (enum == 2))
```

Lastly, the authors also make use of an Andersen-Gill model for comparison. This model has the advantage that it uses all of the data directly, but because of correlation it may underestimate the variance of the relevant coefficients. A method to address this is given in a paper by Lee, Wei, and Amato (1992); it is essentially the same method found in the Wei, *et al*. paper. This method for variance estimation is invoked by specifying the `cluster(id)` term.

```
> afit <- coxph(Surv(start, stop, event) ~ rx + size +
+ number + cluster(id), data = bladder2)
```

```
> afit

Call:
coxph(formula = Surv(start, stop, event) ~ rx + size +
number + cluster(id), data = bladder2)

           coef exp(coef) se(coef) robust se      z     p
    rx -0.4116     0.663   0.1999    0.2415 -1.704 0.088
  size -0.0411     0.960   0.0703    0.0723 -0.568 0.570
number  0.1637     1.178   0.0478    0.0569  2.876 0.004


Likelihood ratio test=14.7  on 3 df, p=0.00213 n= 190

> sqrt(diag(afit$var))

[1] 0.24876453 0.07421445 0.05842243

> sqrt(diag(afit$naive.var))

[1] 0.19989234 0.07029462 0.04776578
```

The naive estimate of standard error is 0.20, the correct estimate of 0.24 is intermediate between the naive estimate and the linear combination estimate. Further discussion on these estimators can be found in the section Robust Variance Estimation.

PENALIZED COX MODELS

Consider a Cox model with both constrained and unconstrained effects

$$\lambda_i(t) = \lambda_0(t) e^{X_i\beta + Z_i\omega},$$

where X and Z are the covariates, and β and ω are the unconstrained and constrained coefficients, respectively. The problem is solved by maximizing a *penalized partial likelihood*

$$PPL = PL(\beta, \omega; \text{data}) - f(\omega, \theta)$$

over both β and ω. Here PL is the usual Cox partial likelihood, treating ω as "just another parameter," and f is some constraint function which gives large values to "bad" values of ω. For the moment assume that θ, a vector of tuning parameters, is known and constant.

Following Gray (1992), let I be the usual Cox model information matrix, and let

$$H = I - \begin{pmatrix} 0 & 0 \\ 0 & f'' \end{pmatrix}$$

be the second derivative matrix for the penalized likelihood PPL. Gray's suggested estimate of the variance is

$$V = H^{-1} I H^{-1}. \tag{28.8}$$

Let c be a column vector of constants and (β', ω') be the combined vector of $p+q$ parameters. Then for a general test of the hypothesis $z = (\beta', \omega')c = 0$, Gray recommends the Wald test $z'(c' H^{-1} c)^{-1} z$. Because of the shrinkage, this is not necessarily a chi-square statistic. Let e be the eigenvalues of the matrix $(c' H^{-1} c)^{-1}(c' V c)$; under H_0 the Wald test is distributed as $\sum e_i X_i^2$, where the X_i are independent identically distributed (iid) Gaussian random variables.

Let $k = \sum e_i$. When the e_i are all 0 or 1, the case for non-penalized models, the mean and variance of the test statistic are k and $2k$, respectively, and the distribution is chi-square on k degrees of freedom. In penalized models, $e_i \leq 1$ and the variance is $\sum e_i^2 < 2k$, so the distribution of the statistic is more compact than a standard chi-square based on k degrees of freedom. Therefore, the test is conservative.

The generalized degrees of freedom for the test statistic can be written as

$$df = \text{trace}[(c' H^{-1} c)^{-1}(c' Vc)].$$

Thus, the computation of eigenvalues is not strictly necessary. For a particular term in the model, this becomes $\text{trace}((H^{-1}[i, i])^{-1} V[i, i])$, where [ ] indicates Spotfire S+-style subscripts and i indexes the columns corresponding to the term.

An alternate variance estimator is to use H^{-1} directly, the inverse of the second derivative of the full log-likelihood, which is the variance used in the Wald statistic. It has an interpretation as a posterior variance in a Bayes setting, and tends to be larger than V and thus more conservative. Spotfire S+ returns both `var2`$=V$ and `var`$=H^{-1}$. The chi-square tests are based on `var`, as simulation experiments suggest that this is the more reliable choice for tests.

Fitting Penalized Models

Spotfire S+ provides two functions for including penalized terms in the Cox model. The `ridge` function implements a simplified pseudo-ridge regression, while the `pspline` function implements a penalized B-spline fit. Both functions are packaging functions that provide a convenient interface to the functions that actually do the fitting: a control function is used to estimate θ and a penalty function computes f and its first and second derivatives.

Fitting a Ridge Model

For `ridge`, let $f(\omega, \theta) = (\theta/2)\sum\omega_j^2$ be a penalty function which tends to shrink the coefficients ω_j towards zero. The penalty function inside `ridge` is then

```
function(coef, theta)
{
    list(penalty = sum(coef^2) * theta/2,
         first = theta * coef,
         second = rep(theta, length(coef)),
         flag = F)
}
```

The control function is even simpler:

```
function(parms, ...) list(theta = parms$theta, done = T)
```

As an example of using `ridge`, consider again the `ovarian` data set. Recall that these data give the survival time of 26 women with advanced ovarian carcinoma, with major covariates `age` and `ecog.ps`. The `ecog.ps` variable is a performance score that measures physical debilitation, with 0 corresponding to normal and 4 corresponding to bedridden. In the example below, `fit0` is the standard Cox model and `fit1` is the penalized model. The shrinkage parameter $\theta = 1$ is chosen arbitrarily.

```
> fit0 <- coxph(Surv(futime,fustat) ~ rx + age + ecog.ps,
+ data = ovarian)
> fit0

Call:
coxph(formula = Surv(futime, fustat) ~ rx + age +
  ecog.ps, data = ovarian)

           coef exp(coef) se(coef)     z      p
     rx -0.815     0.443   0.6342 -1.28 0.2000
    age  0.147     1.158   0.0463  3.17 0.0015
ecog.ps  0.103     1.109   0.6064  0.17 0.8600

Likelihood ratio test=15.9  on 3 df, p=0.00118  n= 26

> fit1 <- coxph(Surv(futime, fustat) ~ rx + ridge(age,
+ ecog.ps, theta = 1), data = ovarian)
> fit1
```

```
Call:
coxph(formula = Surv(futime, fustat) ~ rx + ridge(age,
  ecog.ps, theta = 1), data = ovarian)

                 coef se(coef)    se2 Chisq DF      p
           rx -0.856 0.6161   0.6156  1.93 1  0.1600
    ridge(age)  0.123 0.0385   0.0354 10.21 1  0.0014
ridge(ecog.ps)  0.109 0.5734   0.5484  0.04 1  0.8500

Iterations: 1 outer, 4 Newton-Raphson
Degrees of freedom for terms= 1.0 1.8
Likelihood ratio test=15.6  on 2.76 df, p=0.00104  n= 26
```

The likelihood ratio test that is printed is twice the difference in the PL between the null model ($\beta = \omega = 0$) and the final fitted model. The p-value is based on comparison to a chi-square distribution with 2.73 degrees of freedom. As mentioned earlier, this comparison is somewhat conservative (p is too large). The eigenvalues for the problem, `eigen(solve(fit1$var, fit1$var2))`, are 1, 0.9156, and 0.8486. The respective quantiles of the weighted sum of squared normals and the chi-square distribution `qchisq(q, 2.73)` are:

	80%	90%	95%	99%
Actual sum	4.183	5.580	7.027	10.248
$\chi^2_{2.73}$	4.264	5.818	7.337	10.789

From this table, we see that the actual distribution is somewhat more compact than the chi-square approximation.

The shrinkage has a smaller effect on age than on the performance score. Although the unpenalized coefficients for the two covariates are of about the same magnitude, as shown by `fit0`, the standard error for `ecog.ps` is much larger. The impact on overall fit (Cox PL) of shrinking the age coefficient is thus larger than that for the performance score; the age coefficient is "harder to change."

Fitting Spline Models

The `pspline` function is used to fit a general spline term within the Cox model. The method used is p-splines, described in Eilers and Marx, 1996. The p-spline approach has several useful properties:

- For moderate degrees of freedom, a smaller number of basis functions give a fit that is nearly identical to the standard smoothing spline.
- The p-spline basis has basis functions that are evenly spaced and identical in shape. Because of the symmetry of the basis functions, the usual spline penalty $\int [f''(x)]^2 dx$ is very close to the sum of second differences of the coefficients, `θ*sum((diff(diff(coef)))^2)`, which is very easy to program.
- The penalty does not depend on the values of the data, other than for establishing the range of the spline basis.
- If the coefficients are a linear series, the fitted function is a line. Thus a linear trend test on the coefficients is a test for the significance of a linear model. This makes it relatively easy to test for the significance of nonlinearity.
- Since there are a small number of terms, ordinary methods of estimation can be used. That is, the program can compute and return the variance matrix of $\hat{\beta}$. Contrast this to the classical smoothing spline basis, which has a term (knot) for each unique data value; for a large sample size, storage of the n by n matrix H becomes infeasible.

The penalty function for the p-spline is $(\omega, \theta) = ([\theta/(1-\theta)](\omega' P\omega))/2$, where $P = T'T$, and T is the matrix of second differences. The case $\theta = 1$ corresponds exactly to the straight line model (an infinite penalty for curvature).

As an example, consider again the `ovarian` data and fit three models:

```
> fit1 <- coxph(Surv(futime, fustat) ~ rx + age,
+ data = ovarian)

> fit2 <- coxph(Surv(futime, fustat) ~ rx + pspline(age,
+ df = 2), data = ovarian)

> fit4 <- coxph(Surv(futime, fustat) ~ rx + pspline(age,
+ df = 4), data = ovarian)

> fit1
```

```
Call:
coxph(formula = Surv(futime, fustat) ~ rx + age, data
   = ovarian)

       coef exp(coef) se(coef)     z      p
 rx -0.804     0.448   0.6320 -1.27 0.2000
age  0.147     1.159   0.0461  3.19 0.0014

Likelihood ratio test=15.9  on 2 df, p=0.000355  n= 26
```

```
> fit2
```

```
Call:
coxph(formula = Surv(futime, fustat) ~ rx + pspline(
  age, df = 2), data = ovarian)

                            coef se(coef)    se2
                       rx -0.589 0.6990   0.6786
pspline(age, df = 2), lin  0.144 0.0433   0.0433
pspline(age, df = 2), non
                          Chisq   DF       p
                       rx  0.71 1.00 0.40000
pspline(age, df = 2), lin 11.09 1.00 0.00087
pspline(age, df = 2), non  0.84 0.93 0.33000

Iterations: 2 outer, 7 Newton-Raphson
     Theta= 0.447
Degrees of freedom for terms= 0.9 1.9
Likelihood ratio test=17  on 2.87 df, p=0.0006  n= 26
```

```
> fit4
```

```
Call:
coxph(formula = Surv(futime, fustat) ~ rx + pspline(
  age, df = 4), data = ovarian)
```

```
                              coef se(coef)   se2 Chisq
                        rx -0.373 0.761    0.749 0.24
pspline(age, df = 4), lin  0.139 0.044    0.044 9.98
pspline(age, df = 4), non                       2.59
                              DF      p
                        rx 1.00 0.6200
pspline(age, df = 4), lin 1.00 0.0016
pspline(age, df = 4), non 2.93 0.4500

Iterations: 3 outer, 13 Newton-Raphson
     Theta= 0.242
Degrees of freedom for terms= 1.0 3.9
Likelihood ratio test=19.4  on 4.9 df, p=0.00149  n= 26
```

The printout for the simple Cox model `fit1` shows an increase in the log-hazard for death of 0.147 per year of age, with an overall chi-square for the model of 15.9. The p-spline basis functions sum to a constant, so the first one is deleted to remove the singularity.

There are seven coefficients associated with the fit with two degrees of freedom, which are summarized in the printout as a linear and nonlinear effect. Similarly, the thirteen coefficients associated with the four degrees of freedom fit are summarized as simply a linear and nonlinear effect. Because of the symmetry of the basis functions, the chi-square test for linearity is a test for zero slope in a regression of the spline coefficients on the centers of the basis functions, using `var` as the known variance matrix of the coefficients. The linear "coefficient" that is printed is the slope of this regression. This computation of coefficient and *p*-value is equivalent to the approximate backwards elimination method of Lawless and Singhal (1978), here removing all the nonlinear terms for age. If the terms being dropped are important (that is, there is a significant nonlinearity), the approximation for the linear coefficient is not as accurate.

As a more interesting example, consider the data from the Multicenter Post-Infarction Project (MPIP) contained in the data set `mpip`. This data set contains data on 866 patients, gathered after hospital admission for myocardial infarction. The main goal of the study was to determine which factors, if any, were predictive of the future clinical course of the patients.

Our model of survival time uses four variables:

- `ved`, ventricular ectopic polarizations per hour obtained from analysis of a 24 hour Holter monitor. A large number of these irregular heartbeats is indicative of a high risk for fatal arrhythmia.
- `nhya`, New York Heart Association class. This is a measure of the amount of activity that a subject is able to undertake without angina, ranging from 1 to 4.
- `rales`, presence of pulmonary rales on initial examination.
- `ef`, ejection fraction. This is the proportion of blood cleared from the heart on each contraction.

The `ved` variable is very skewed; it has a mean value of 19.1, a median of 0.45, a maximum value of 733, and 14% of the subjects have a value of 0. The minimum nonzero value is 0.042, so we use the derived covariate `lved = log(ved+0.02)` instead. This is still a skewed variable, but not unmanageably so. A simple linear fit of the four variables shows all to be highly significant:

```
> fit1 <- coxph(Surv(futime, status) ~ lved + nyha + rales +
+ ef, data = mpip, na.action = na.exclude)
> fit1

Call:
coxph(formula = Surv(futime, status) ~ lved + nyha +
  rales + ef, data = mpip, na.action =
  na.exclude)

         coef exp(coef) se(coef)     z        p
 lved  0.1007     1.106  0.04266  2.36  1.8e-02
 nyha  0.3707     1.449  0.09379  3.95  7.7e-05
rales  0.4535     1.574  0.10527  4.31  1.7e-05
   ef -0.0265     0.974  0.00833 -3.18  1.5e-03

Likelihood ratio test=79.4  on 4 df, p=2.22e-016  n=764
 (102 observations deleted due to missing values)
```

Now we explore more complicated forms for the effect of the covariates. Since `rales` is a binary covariate, it allows no further transformation; `nyha`, with four levels, is entered as a factor variable. That leaves the two continuous variables, `lved` and `ef`, to be modeled as p-splines with the default four degrees of freedom:

```
> fit2 <- coxph(Surv(futime, status) ~ pspline(lved) +
+ factor(nyha) + rales + pspline(ef), data = mpip,
+ na.action = na.exclude)
> fit2

Call:
coxph(formula = Surv(futime, status) ~ pspline(lved) +
  factor(nyha) + rales + pspline(ef), data =
  mpip, na.action = na.exclude)

                          coef se(coef)     se2 Chisq
pspline(lved), linear   0.0982 0.04384 0.04359  5.02
pspline(lved), nonlin                          2.59
        factor(nyha)1 -0.0308 0.15917 0.15890  0.04
        factor(nyha)2  0.2426 0.10380 0.10337  5.46
        factor(nyha)3  0.2008 0.06745 0.06725  8.86
                rales  0.4204 0.10816 0.10761 15.11
  pspline(ef), linear -0.0256 0.00738 0.00737 12.03
  pspline(ef), nonlin                          8.06
                         DF       p
pspline(lved), linear  1.00 0.02500
pspline(lved), nonlin  3.06 0.47000
        factor(nyha)1  1.00 0.85000
        factor(nyha)2  1.00 0.01900
        factor(nyha)3  1.00 0.00290
                rales  1.00 0.00010
  pspline(ef), linear  1.00 0.00052
  pspline(ef), nonlin  3.01 0.04500

Iterations: 4 outer, 11 Newton-Raphson
     Theta= 0.776
     Theta= 0.66
Degrees of freedom for terms= 4.1 3.0 1.0 4.0
Likelihood ratio test=92.5  on 12.04 df, p=1.69e-014
  n=764 (102 observations deleted due to missing values)
```

From this, we conclude that the first two classes of `nyha` can be combined, the nonlinear effect for VED is not significant, and the nonlinear effect from ejection fraction is important.

Plots of the two spline terms can be produced as follows:

```
> temp <- predict(fit2, type = "terms", se.fit = T)
> tmat <- cbind(temp$fit[,1],
+ temp$fit[,1] - 1.96 * temp$se.fit[,1],
+ temp$fit[,1] + 1.96 * temp$se.fit[,1])

> jj <- match(sort(unique(mpip$lved)), mpip$lved)
> matplot(mpip$lved[jj], tmat[jj,], type = "l",
+ lty = c(1,2,2), xaxt = "n")

> xx <- c(0, 1, 50, 100, 500)
> axis(1, log(xx + .2), as.character(xx))
> title(xlab = "VED", ylab = "log hazard")

> tmat2 <- cbind(temp$fit[,4],
+ temp$fit[,4] - 1.96 * temp$se.fit[,4],
+ temp$fit[,4] + 1.96 * temp$se.fit[,4])

> jj2 <- match(sort(unique(mpip$ef)), mpip$ef)
> matplot(mpip$ef[jj2], tmat2[jj2,],type = "l",
+ lty = c(1, 2, 2), xlab = "Ejection Fraction",
+ ylab = "log hazard")
```

The resulting plot is shown in Figure 28.8. Some extra work was required to label the first graph in the original `ved` units; this is done with the `axis` command. The `match` function and the `jj` subscripts sort the plot from left to right; otherwise, the line becomes a scribble. The graph shows an increase in risk with ejection fractions below 60%, sharply so below 20%. The rise after 70% is not significant, given the wide confidence intervals. This agrees with the conventional wisdom of the physicians that the instrumentation is not able to reliably distinguish values above this level.

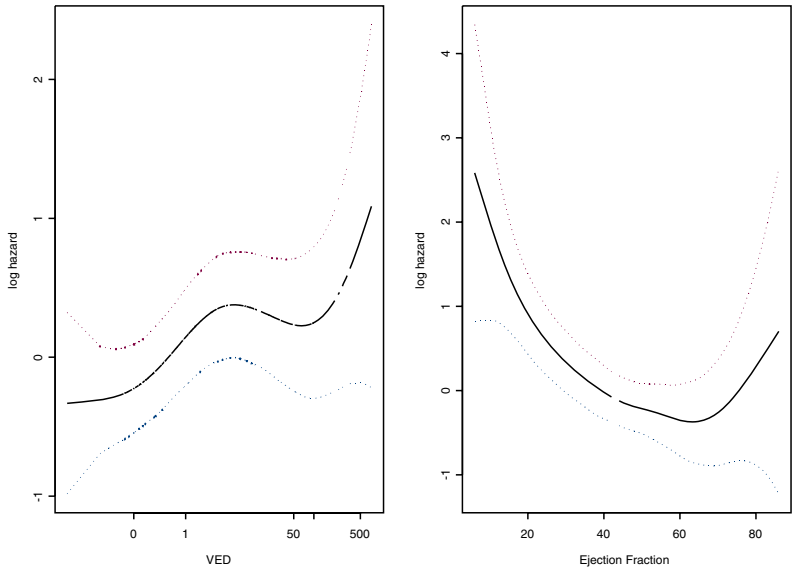

Figure 28.8: *Plots of spline fit terms in* `mpip` *data model.*

FRAILTY MODELS

In this section, we consider survival models to which a random effect is added. The random effect is usually viewed as a categorical variable that describes excess risk, or *frailty* for an individual or family. The idea is that individuals have different frailties, and those who are most frail die earlier than the others.

Computationally, the frailty is usually viewed as an unobserved covariate. This has led naturally to the use of the EM algorithm as an estimation tool. Assume a proportional hazards model with random effects, or *frailties*, with the hazard function

$$\lambda_i(t) = \lambda_0(t)e^{X_i\beta + Z_i\omega}$$

Here β is a vector of p fixed effects and ω is a vector of q random effects, where the individual elements ω_j are iid realizations from some distribution $W(\theta)$. The matrix X normally contains measured covariate values, and Z is a design matrix that describes how the random effects apply to individual subjects. Both X and Z might contain time-dependent effects, but we ignore this complication for the moment. The baseline hazard may contain other parameters ξ, but these are also ignored. If X contains an intercept term (implicit in the proportional hazards model), we can constrain ω to have mean 0.

We can treat the random effects as unobserved data and apply the EM algorithm. The x of the formal EM argument is the entire observed data (time, status, covariates) plus the frailties, and y is the data without the frailties. The full log-likelihood had we observed ω is

$$L_f = \sum_{j=1}^{q} \log(W(\omega_j;\theta)) + \sum_{n} \delta_i[\log(\lambda_0(t_i)) + X_i\beta + Z_i\omega] - \Lambda_0(t_i)e^{X_i\beta + Z_i\omega}$$

Here $\delta_i = 0$ for censored observations and 1 for events. The X term is an n by p matrix, and Z is an n by q matrix.

This model setup is similar in notation to random effects models in linear regression. Another notation, more common in the survival literature, is to define $\omega = \exp(Z_i\omega)$ as the frailty parameter for each subject. Then

$$\lambda_i(t) = \bar{\omega}_j\lambda_0(t)e^{X_i\beta}$$

where subject i is a member of the jth family.The imposed constraint is usually $E(\omega) = 1$ rather than $E(\omega) = 0$.

The most popular choice for the random distribution is the gamma frailty model, where ω is from a gamma distribution with mean 1 and variance $\theta = \sigma^2$. Then the marginal likelihood L_g, after integrating out the frailty, is

$$L_g = PL + \sum_j \nu \log\left(\frac{\nu}{\nu + E_j^*}\right) + \log\left(\frac{\Gamma(\nu + D_j)}{\Gamma(\nu)}\right) - D_j \log(\nu + D_j)$$

where PL is the numerical value returned as the partial likelihood by a standard Cox model for the given values of β and ω, ω having been entered as an offset term. This result applies only to the simple frailty problem where each subject i is a member of exactly one family j, with one random effect per family. Then D_j gives the number of events in the family and $E_j^* = E_j / (\exp(\omega_j))$, where E_j is the expected number of events for the family, using the final model.

There is an interesting connection between frailty models and penalized likelihoods. In particular, let the penalty function for a constrained solution be the log-gamma density

$$-f(\omega,\nu) = \nu(\omega - e^{\omega}) + \nu\log(\nu) - \log\Gamma(\nu)$$

with $\vartheta = 1/\nu$ as the variance of the random effect, and with Z defined as in the frailty model. The first and second derivatives are $\nu(1 - \exp(\omega))$ and $-\nu\exp(\omega)$, respectively. Surprisingly, for any fixed value of ν, the EM algorithm and this constrained minimization have the same solution. This connection between the two methods has several interesting consequences, as listed below.

- Since penalized likelihood methods are well understood numerically, this leads to more stable computational methods; the EM algorithm is slow, and the proper variance estimate is uncertain. In particular, the penalized likelihood methods fit nicely into the new `coxph` function.
- There is a connection to the "degrees of freedom" for a fit.
- It suggests a heuristic approach for other frailty distributions and/or frailty terms such as nested models, for which the EM algorithm is not tenable.

Fitting a Cox Model with Frailty

To add a frailty term to the Cox model, use the `frailty` function within the call to `coxph`. For example, consider the `rats` data set, which contains information on the effect of treatment for survival of 150 female rats from 50 different litters. The data set has three rats per litter, one of which received a potentially tumorigenic treatment. Forty rats developed a tumor during follow-up. We use the Breslow approximation for tied times to match other analyses of this same data in the literature:

```
> rfit <- coxph(Surv(time, status) ~ rx + frailty(litter),
+ data = rats, method = "breslow")
> rfit

Call:
coxph(formula = Surv(time, status) ~ rx + frailty(
  litter), data = rats, method = "breslow")

                 coef se(coef)   se2 Chisq   DF     p
            rx 0.906 0.323    0.319  7.88  1.0 0.005
frailty(litter)                     16.89 13.8 0.250

Iterations: 6 outer, 20 Newton-Raphson
     Variance of random effect = 0.474   EM likelihood =
 -181.1
Degrees of freedom for terms = 1.0 13.9
Likelihood ratio test = 36.3  on 14.83 df, p = 0.00145
    n = 150

> rfit0 <- coxph(Surv(time, status) ~ rx, data = rats,
+ method = "breslow")
> rfit0
```

```
Call:
coxph(formula = Surv(time, status) ~ rx, data = rats,
  method = "breslow")


    coef exp(coef) se(coef)    z      p
rx 0.898      2.46    0.317 2.83 0.0047

Likelihood ratio test=7.87  on 1 df, p=0.00503  n= 150
```

```
> rfit1 <- coxph(Surv(time, status) ~ rx + frailty(litter,
+ theta = 1), data = rats, method = "breslow")
> rfit1

Call:
coxph(formula = Surv(time, status) ~ rx + frailty(litter,
  theta = 1), data = rats, method = "breslow")

                           coef se(coef)   se2 Chisq
                       rx 0.918 0.327    0.321  7.85
frailty(litter, theta = 1                      27.25
                            DF      p
                       rx  1.0 0.0051
frailty(litter, theta = 1 22.7 0.2300

Iterations: 1 outer, 5 Newton-Raphson
     Variance of random effect= 1    EM likelihood = -181.5
Degrees of freedom for terms=  1.0 22.7
Likelihood ratio test=50.7  on 23.67 df, p=0.001  n= 150
```

The main thing to notice about these results is how little the treatment coefficient is changed by the inclusion of a random effect term. This is likely a consequence of the balanced model; each litter received both active and control treatments.

For a fixed value of the frailty, the iteration is nearly as efficient as for a normal Cox model, which usually requires 3–4 iterations. The generalized fit required six guesses to maximize the profile likelihood, and about three internal iterations per ν value.

The "likelihood ratio test" is always the difference in partial likelihood between the initial and final fit, ignoring penalty terms and corrections. The default for the initial fit is $(\beta, \omega) = 0$, which is a fit with no covariates or random effect.

The solution using a Gaussian frailty is not much different:

```
> rfit2 <- coxph(Surv(time, status) ~ rx + frailty(litter,
+ dist = "gauss"), data = rats, method="breslow")
> rfit2

Call:
coxph(formula = Surv(time, status) ~ rx + frailty(litter,
  dist = "gauss"), data = rats, method="breslow")

                           coef se(coef)   se2 Chisq
                       rx 0.905 0.322    0.318  7.89
frailty(litter, dist = "g                       14.94
                            DF      p
                       rx  1.0  0.005
frailty(litter, dist = "g 11.5  0.220

Iterations: 5 outer, 14 Newton-Raphson
     Variance of random effect= 0.396
Degrees of freedom for terms=  1.0 11.5
Likelihood ratio test=34.2  on 12.51 df, p=0.000846310
  n= 150
```

ADDITIONAL TECHNICAL DETAILS

The remaining subsections provide additional details on computations and options available for fitting proportional hazards models, including:

- The handling of ties
- The effect of ties on the definitions of residuals
- Tests for proportional hazards
- Robust variance estimation
- The handling of case weights
- Details about the computations of `coxph`

Computations for Tied Deaths

For untied data, the terms in the partial likelihood (Equation (28.1)) look like

$$\left(\frac{r_1}{\sum_i r_i}\right)\left(\frac{r_2}{\sum_{i>1} r_i}\right)\cdots,$$

where $r_1, r_2, \ldots, r_n$ are the subject risk scores. Assume the real data are continuous, but the recorded data have tied death times. For example, several subjects might die on the first day of their hospital stay but they do not all perish at the same moment. For a simple example, assume 5 subjects (ordered by time of death or censoring) are in a study and the first two die at the same recorded time. If the time data had been more precise, the first two terms in the likelihood would be either

$$\left(\frac{r_1}{r_1 + r_2 + r_3 + r_4 + r_5}\right)\left(\frac{r_2}{r_2 + r_3 + r_4 + r_5}\right)$$

or

$$\left(\frac{r_2}{r_1 + r_2 + r_3 + r_4 + r_5}\right)\left(\frac{r_1}{r_1 + r_3 + r_4 + r_5}\right).$$

Notice that the numerators remain constant, but the denominators do not. The question is, how do you approximate the correct term for the likelihood?

The Breslow approximation is the most commonly used because it is the easiest to program. It uses the complete sum, $r_1 + r_2 + r_3 + r_4 + r_5$, for both denominators. Clearly, if the proportion of ties is large, this deflates the partial likelihood.

The Efron approximation uses $0.5r_1 + 0.5r_2 + r_3 + r_4 + r_5$ as the second denominator, based on the idea that r_1 and r_2 each have a 50% chance of appearing in the "true" second term. If there were 4 tied deaths, the ratios for r_1 to r_4 would be 1, 3/4, 1/2, and 1/4 in each of the four denominator terms, respectively. Though it is not widely used, the Efron approximation is only slightly more difficult to program than the Breslow version. In particular, since the down-weighting is independent of any case weights, the form of the derivatives of the likelihood is unchanged.

An alternate approach attempts an "exact" computation. The exact partial likelihood comes from viewing the data as genuinely discrete. The denominator in this case is $\sum_{i \neq j} r_i r_j$ if two subjects are tied, $\sum_{i \neq j \neq k} r_i r_j r_k$ if three subjects are tied, etc.

When using the `coxph` function to fit proportional hazards models, you can specify any of the above three methods for handling ties. The default is the Efron approximation, `method="efron"`. The other two may be specified by setting `method="breslow"` or `method="exact"`. Note that when there are no ties, all three methods produce the same likelihood function.

Effect of Ties on Residual Definitions

The Efron approximation induces changes in the residuals' definitions. In particular, the Cox score statistic is still

$$U = \sum_{i=1}^{n} \int_0^{\infty} (Z_i(s) - \bar{Z}(s)) dN_i(s). \tag{28.9}$$

However, the definition of $\bar{Z}(s)$ changes if there are tied deaths at time s. If there are d deaths at s, there are d different values of $\bar{Z}$ used at the time point. The Schoenfeld residuals use $\bar{\bar{Z}}$, the average of these d values, in the computation. The martingale and score residuals require a new definition of $\hat{\Lambda}$. If there are d tied deaths at time t, we again assume that in the exact (but unknown) untied data there are events and corresponding jumps in the cumulative hazard at $t \pm \varepsilon_1 < \ldots < t \pm \varepsilon_d$. Then each of the tied subjects will in expectation experience all of the first hazard increment, but only $(d-1)/d$ of the second, $(d-2)/d$ of the next, and so on. If we equate observed to expected hazard at each of the d deaths, the total increment in hazard at the time point is the sum of the denominators of the weighted means.

Recall our earlier example of 5 subjects of which 1 and 2 have tied deaths:

$$\hat{\Lambda}(t) = \frac{1}{r_1 + r_2 + r_3 + r_4 + r_5} + \frac{1}{r_1/2 + r_2/2 + r_3 + r_4 + r}.$$

For the null model where $r_i = 1$ for all i, the new definition above agrees with the suggestion of Nelson (1969) to use 1/5+1/4 rather than 2/5 as the increment to the cumulative hazard. The formula for the score residuals is demonstrated using our previous example with five subjects, the first two being tied. For subject 1, the residual at time 1 is the sum $a + b$ where

$$a = \left(Z_1 - \frac{r_1 Z_1 + r_2 Z_2 + \ldots + r_5 Z_5}{r_1 + r_2 + \ldots + r_5}\right)\left(\frac{dN_1}{2} - \frac{r_1}{r_1 + r_2 + \ldots + r_5}\right)$$

$$= \left(Z_1 - \frac{r_1 Z_1/2 + r_2 Z_2/2 + \ldots + r_5 Z_5}{r_1/2 + r_2/2 + \ldots + r_5}\right)\left(\frac{dN_1}{2} - \frac{r_1/2}{r_1/2 + r_2/2 + \ldots + r_5}\right)$$

The products defining a and b do not neatly collapse into $(Z_1 - \bar{\bar{Z}})d\hat{M}$, but they are easy to compute. The connection between residuals and the exact partial likelihood is not as precise and are thus not implemented. If residuals are requested after a Cox fit with `method="exact"` the Breslow formulae are used.

Tests for Proportional Hazards

The key ideas of this section are taken from Grambsch and Therneau (1994). Most of the common alternatives to the hypothesis test of proportional hazards can be cast in terms of a *time-varying coefficient* model. That is, we assume that

$$\lambda(t;Z) = \lambda_0(t)e^{\beta_1(t)Z_1 + \beta_2(t)Z_2 + \cdots}.$$

If Z_j is a 0/1 covariate such as treatment, this formulation is completely general in that it encompasses all alternatives to proportional hazards. The proportional hazards assumption is then a test for $\beta(t) = \beta$, which is a test for zero slope in the appropriate plot of $\hat{\beta}(t)$ on t. Let i index subjects, j index variables, and k index the death times. Then let s_k be the Schoenfeld residual and V_k be the contribution to the information matrix (Equation (28.4)) at time t_k. Define the rescaled Schoenfeld residual as

$$s_k^* = \hat{\beta} + s_k V_k^{-1}.$$

The main results are:

- $E(s_k^*) = \beta(t_k)$, so that a smoothed plot of s^* versus time gives a direct estimate of $\hat{\beta}(t)$.
- Many of the common tests for proportional hazards are linear tests for zero slope, applied to the plot of s^* versus $g(t)$ for some function g. In particular, the Z:PH test popularized in the SAS PHGLM procedure corresponds to $g(t) =$ rank of the death time. The test of Lin (1991) corresponds to $g(t) = K(t)$, where K is the Kaplan-Meier.

- Confidence bands, tests for individual variables, and a global test are available, and all have the fairly standard *linear models* form.
- The estimates and tests are affected very little if the individual variance estimates V_k are replaced by their global average $\bar{V} = \sum V_k / d = I / d$. Calculations then require only the Schoenfeld residuals and the standard Cox variance estimate I^{-1}.

For the global test, let $g(t)$ be the desired transformation of time, and let $g_k = g(t_k)$ be the value of g at the kth death time. Then

$$T = (\sum g_k s_k)' D^{-1} (\sum g_k s_k)$$

is asymptotically χ^2 on p degrees of freedom, where

$$D = \sum g_k^2 V_k - (\sum g_k V_k)(\sum V_k)^{-1} (\sum g_k V_k)' \quad .$$

Because the s_k sum to zero, a little algebra shows that the above expression is invariant if g_k is replaced by $g_k - c$ for any constant c. Subtraction of a mean will, however, result in less computer round-off error. A further simplification occurs by using $\bar{V}$, leading to

$$T = [\sum (g_k - \bar{g}) s_k]' \left[\frac{dI^{-1}}{\sum (g_k - \bar{g})^2} \right] [\sum (g_k - \bar{g}) s_k] \quad . \qquad (28.10)$$

For a given covariate j, the diagnostic plot has s^*_{kj} on the vertical axis and g_k on the horizontal. The variance matrix of s^*_{kj} is $\Sigma_j = (A - cJ) + cI$, where A is a $d \times d$ diagonal matrix whose kth diagonal element is $V^{-1}_{k,jj}$. Here $c = I^{-1}_{jj}$, J is a $d \times d$ matrix of ones and I is the identity matrix. The constant cI reflects the uncertainty in s^* due to the $\hat{\beta}$ term. If only the shape of $\beta(t)$ is of interest (for

example, is it linear or sigmoid) the c can be dropped. If absolute values are important (for example, $\beta(t) = 0$ for $t > 2$ years), it should be retained.

For smooths that are linear operators, such as splines or the loess function, the final smooth is $\hat{s}^* = Hs^*$ for some matrix H. Then $\hat{s}^*$ is asymptotically normal with mean 0 and variance $H\Sigma_j H'$. Standard errors are computed using ordinary linear model methods. If V_k is replaced with $\bar{V}$, then S_j simplifies to $I_{jj}^{-1}((d+1)I - J)$. With the same substitution, the component-wise test for linear association is

$$t_j = \frac{\sum(g_k - \bar{g})y_k}{\sum dI_{jj}^{-1}(g_k - \bar{g})^2} . \tag{28.11}$$

The `cox.zph` function uses Equation (28.10) as a global test of proportional hazards, and Equation (28.11) to test individual covariates. The plot method for `cox.zph` uses a natural spline smoother. Confidence bands for the smooth are based on the full covariance matrix, with $\bar{V}$ replacing V_k.

Though the simulations in Grambsch and Therneau (1994) did not uncover any situations where the simpler formulae based on $\bar{V}$ are less reliable, such cases could arise. The substitution trades a possible increase in bias for a substantial reduction in the variance of the individual V_k. It is likely to be unwise in those cases where the variance of the covariates within the risk sets differs substantially from the variance between different risk sets. Two examples come to mind. The first is a stratified Cox model, where the strata represent different populations. In a multi-center clinical trial (for instance, inner city, Veterans Administration, and suburban hospitals), disparate populations are often serviced. In this case, a separate average $\bar{V}$ should be formed for each strata. A second example is where the covariate mix changes markedly over time, perhaps because of aggressive censoring of certain patient types. These cases have not

been addressed directly in the software. However, `coxph.detail` returns all of the V_k matrices, which can then be used to construct specialized tests for such situations.

Clearly, no one scaling function $g(t)$ is optimal for all situations. The `cox.zph` function directly supports four common choices: identity, log, rank, and 1 – Kaplan-Meier. By default, it uses the last of these, based on the following rationale. Since the test for proportional hazards is essentially a test for significant regression of the scaled residual modeled linearly in the g_k, we expect this test to be adversely effected if there are outliers in the g_k. We would also like the test to be at most mildly affected by the censoring pattern of the data. The Kaplan-Meier transform appears to satisfy both of these criteria.

Robust Variance Estimation

The following technical discussion of robust variance estimation for Cox models leads to a rather simple implementation conceptually. The basic idea is to compute an approximate matrix of changes in estimated coefficients, L, resulting from leaving out each observation one at a time. The robust estimate of variance is then $L' L$, which relates to other variance estimators as follows:

- $L' L$ is equivalent to the *working independence estimate* in generalized estimating equations models.
- $L' L$ is an approximate jackknife estimate of variance.
- $L' L$ is equivalent to the Wei, Lin, and Weissfeld (1989) variance estimate for a Cox model.
- $L' L$ is a robust *sandwich estimate* as discussed in Huber (1967).

If the observations are grouped and correlated within groups, this idea works if entire groups (rather than individual observations) are left out for computing the approximate jackknife variance estimate. This case corresponds to Cox models with a counting process formulation and multiple observations per subject. The resulting estimator of variance is called the *grouped jackknife* estimator.

The Sandwich Estimator

The following discussion describes the general sandwich estimator, a modification of the sandwich estimator for grouped data, and its implementation for Cox models. Robust variance calculations are based on the *sandwich estimate*

$$V = ABA',$$

where $A^{-1} = I$ is the usual information matrix and B is a *correction term*. The genesis of this formula can be found in Huber (1967), who discusses the behavior of any solution to an estimating equation

$$\sum_{i=1}^{n} \phi(x_i, \hat{\beta}) = 0.$$

Of particular interest is the case of a maximum likelihood estimate based on distribution f, so that $\phi = \partial \log(f) / \partial \beta$, when in fact the data are observations from distribution g. Then under appropriate conditions, $\hat{\beta}$ is asymptotically normal with mean β and covariance $V = ABA'$, where

$$A = \left(\frac{\partial E\Phi(\beta)}{\partial \beta} \right)^{-1}$$

and B is the covariance matrix for $\Phi = \sum \phi(x_i, \beta)$. Under most situations the derivative can be moved inside the expectation, and A is the inverse of the usual information matrix. This formula was rediscovered by White (1980, 1982) and is also known in the econometric literature as *White's method*. Under the common case of maximum likelihood estimation we have

$$\sum_{i=1}^{\cdot\cdot} \phi(x_i, \hat{\beta}) = \sum_{i=1}^{\cdot\cdot} \frac{\partial \log f(x_i)}{\partial \beta}$$

$$= \sum_{i=1}^{n} u_i(\beta)$$

By interchanging the order of the expectation and the derivative, A^{-1} is the expected value of the information matrix, which is estimated by the observed information I. Since $E[u_i(\beta)] = 0$,

$$\begin{aligned} B &= \text{var}(\Phi) = E(\Phi^2) \\ &= \sum_{i=1}^{n} E[u'_i(\beta)u_i(\beta)] + \sum_{i \neq j}^{n} E[u'_i(\beta)u_i(\beta)] \end{aligned} \tag{28.12}$$

where $u_i(\beta)$ is assumed to be a row vector. If the observations are independent, the u_i are also independent and the cross terms in Equation (28.12) are zero. A natural estimator of B is

$$\begin{aligned} \hat{B} &= \sum_{i=1}^{n} u'_i(\hat{\beta})u_i(\hat{\beta}) \\ &= U'U, \end{aligned}$$

where U is the matrix of *score residuals* (the ith row of U equals $u_i(\hat{\beta})$). The column sums of U are the efficient score vector Φ.

As a simple example, consider generalized linear models. McCullagh and Nelder (1989) maintain that overdispersion "is the norm in practice and nominal dispersion the exception." To account for overdispersion they recommend inflating the nominal covariance matrix of the regression coefficients $A = (X'WX)^{-1}$ by a factor

$$c = \sum_{i=1}^{n} \frac{(y_i - \mu_i)^2}{V_i} / (n-p),$$

where V_i is the nominal variance. Smith and Heitjan (1993) show that AB may be regarded as a multivariate version of this variance adjustment factor, and that c and AB may be interpreted as the average ratio of actual variance $(y_i - \mu_i)^2$ to nominal variance V_i. By

premultiplying by AB, each element of the nominal variance-covariance matrix A is adjusted differentially for departures from nominal dispersion.

Modified Sandwich Estimator

When the observations are not independent, the estimator B must be adjusted accordingly. The natural choice $(\sum u_i)^2$ is not available of course, since $\Phi(\hat{\beta}) = 0$ by definition. However, a reasonable estimate is available when the correlation is confined to subgroups. In particular, assume that the data come from clustered sampling with $i = 1, 2, \ldots, k$ clusters, where there may be correlation within clusters but observations from different clusters are independent. Using Equation (28.12), the cross-product terms between clusters can be eliminated and the resulting equation rearranged as

$$var(\Phi) = \sum_{i=1}^{k} \tilde{u}(\beta)' \, \tilde{u}(\beta),$$

where $\tilde{u}_j$ is the sum of u_i over all subjects in the jth cluster. This leads to the *modified sandwich estimator*

$$V = A(\tilde{U}' \, \tilde{U})A,$$

where the collapsed score matrix $\tilde{U}$ is obtained by replacement of each cluster of rows in U by the sum of those rows. If the total number of clusters is small, this estimate is sharply biased towards zero and some other estimate must be considered. In fact, $rank(V) < k$, where k is the number of clusters. Asymptotic results for the modified sandwich estimator require that the number of clusters tend to infinity.

Implementation for Cox Models

Application of these results to the Cox model proceeds by defining a weighted Cox partial likelihood and letting

$$u_i(\beta) = \left(\frac{\partial U}{\partial w_i}\right)_{w=1},$$

where w is the vector of weights. This approach is used by Cain and Lange to define a leverage or influence measure for Cox regression. In particular, they derive the leverage matrix

$$L = UI^{-1},$$

where L_{ij} is the approximate change in $\hat{\beta}_j$ when observation i is removed from the data set. Their estimate can be recognized as a form of the *infinitesimal jackknife*; see, for example, the discussion in Efron (1982) for the linear models case.

The connection to the jackknife is quite general. For any model stated as an estimating equation, the Newton-Raphson iteration has step

$$\Delta\beta = 1'(UI^{-1}),$$

where the column sums of the matrix $L = UI^{-1}$. At the solution $\hat{\beta}$ the iteration's step size is zero by definition. Consider the following approximation to the jackknife:

1. Treat the information matrix I as fixed.
2. Remove observation i.
3. Beginning at the full data solution $\hat{\beta}$ and do one Newton-Raphson iteration.

This is equivalent to removing one row from L and using the new column sum as the increment. Since the column sums of $L(\hat{\beta}) = 0$ are zero, the increment must be $\Delta\beta = -L_{i.}$. That is, the rows of L are an approximation to the jackknife and the sandwich estimate of variance $L'L$ is an approximation to the jackknife estimate of variance.

Lin and Wei (1989) show the applicability of Huber's work to the partial likelihood, and derive the ordinary Huber sandwich estimate $V = I^{-1}(U'U)I^{-1} = L'L$, the approximate jackknife. When the data are correlated, the appropriate form of the jackknife is to leave out an entire subject at a time, rather than just one observation; this is the grouped jackknife. To approximate this, we leave out groups of rows from L, leading to $\tilde{L}'\tilde{L}$ as the approximation to the jackknife.

Examples

Lee, Wei, and Amato (1992) consider highly stratified data sets which arise from inter-observation correlation. As an example, they use paired eye data on visual loss due to diabetic retinopathy, where photocoagulation was randomly assigned to one eye of each patient. There are $n/2 = 1742$ clusters (patients) with 2 observations per cluster. Treating each pair of eyes as a cluster, they derive the modified sandwich estimate $V = \tilde{L}' \tilde{L}$, where $\tilde{L}$ is derived from L in the following way. The L term has one row, or observation, per eye. Because of possible correlation, we want to reduce this to a leverage matrix $\tilde{L}$ with one row per individual. The leverage (or row) for an individual is simply the sum of the rows for each of their eyes. A subject, if any, with only one eye retains the single row of unchanged leverage data. The resulting estimator is shown to be much more efficient than analysis stratified by cluster. A second example given in Lee, Wei, and Amato concerns a litter-matched experiment; in this case, the number of rats per litter may vary.

Wei, Lin, and Weissfeld (1989) consider multivariate survival times. An example is the measurement of both time to progression of disease and time to death for a group of cancer patients. The data set again contains $2n$ observations with time and status variables, subject id, and covariates. It also contains an indicator variable `etype` to distinguish the event type, progression vs. survival. The suggested model is stratified on event type, and includes all strata x covariate interaction terms. One way to do this with `coxph` is

```
> fit2 <- coxph(Surv(time,status) ~ (rx + size + number)*
+ strata(etype))
> Ltilde <- residuals(fit2, type = "dfbeta",
+ collapse = subject.id)
> newvar <- t(Ltilde)
```

The per-subject leverage matrix $\tilde{L}'\tilde{L}$ is `newvar`. An alternate way to do this is

```
> fit2a <- coxph(Surv(time,status) ~ (rx + size + number)*
+ strata(etype) + cluster(id))
```

The `cluster` argument asserts that subjects with the same value of `id` may be correlated.

The data for fitting the above two models is not built into Spotfire S+. However, similar computations can be performed using the `bladder` data frame for comparison. Two ways of producing the robust variance estimate follow.

```
> bladder2 <- bladder[bladder$start< bladder$stop, ]
> afit <- coxph(Surv(start, stop, event) ~ rx + size +
+ number + cluster(id), data = bladder2)
> sqrt(diag(afit$var))

[1] 0.24876453 0.07421445 0.05842243
```

Performing the computation in an alternate way, we get:

```
> bfit <- coxph(Surv(start, stop, event) ~ rx + size +
+ number, data = bladder2)
> db <- resid(bfit, type = "dfbeta", collapse =
+ bladder2$id)
> sqrt(diag(t(db) %*% db))

[1] 0.24876453 0.07421445 0.05842243
```

Using the grouped jackknife approach as suggested here, rather than separate fits for each event type, has some practical advantages:

- It is easier to program, particularly when the number of events per subject is large.
- Other models can be encompassed. In particular, one need not include all of the strata x covariate interaction terms.
- There need not be the same number of events for each subject. The method for building up a joint variance matrix requires that all of the score residual matrices be of the same dimension, which is not the case if information on one of the failure types was not collected for some subjects.

Weighted Cox Models

A Cox model that includes case weights has been suggested by Binder (1992) in the context of survey data. If w_i are the weights, the modified score statistic is

$$U(\beta) = \sum_{i=1}^{n} w_i u_i(\beta). \tag{28.13}$$

The individual terms u_i are still $Z_i(t) - \bar{Z}(t)$ but the weighted mean $\bar{Z}$ is changed in the obvious way to include both the risk weights r and the external weights w. The information matrix can be written as $I = \sum \delta_i w_i v_i$, where δ_i is the censoring variable and v_i is a weighted covariance matrix. The definition of v_i changes in the obvious way from Equation (28.4). If all of the weights are integers, then for the Breslow approximation this reduces to ordinary case weights; that is, the solution is identical to what you obtain by replicating each observation w_i times. With the Efron approximation or the exact partial likelihood approximation, replication of a subject results in a correction for ties.

The `coxph` function allows general case weights. Residuals from the fit are such that the sum of weighted residuals is zero, and the returned values from the `coxph.detail` function are the individual terms u_i and v_i, so that U and I are weighted sums. The sandwich estimator of variance has $L'WL$ as its central term, where W is the diagonal matrix of weights. The estimate of $\hat{\beta}$ and the sandwich estimate of its variance are unchanged if each w_i is replace by cw_i for any $c > 0$. Multiplying weights by c does not change the `robust se` reported by printing a `coxph` fit, but it does decrease the `se(coef)` reported by a factor of `sqrt(c)`.

For either of the Breslow or the Efron approximations, the extra programming to handle weights is modest. For the Breslow method, the logic behind the addition is straightforward and corresponds to

the derivation given above. For tied data and the Efron approximation, the formula is based on extending the basic idea of the approximation

$$E(f(r_1,\ r_2,\ ..)) \approx\ f(E(r_1),\ E(r_2),\ ..)$$

to include the weights as necessary. Returning to the simple example of the section Computations for Tied Deaths, the second term of the partial likelihood is either

$$\left(\frac{w_1 r_1}{w_1 r_1 + w_3 r_3 + w_4 r_4 + w_5 r_5}\right)$$

or

$$\left(\frac{w_2 r_2}{w_2 r_2 + w_3 r_3 + w_4 r_4 + w_5 r_5}\right).$$

To compute the Efron approximation, separately replace the numerator with $0.5(w_1 r_1 + w_2 r_2)$ and the denominator with $0.5 w_1 r_1 + 0.5 w_2 r_2 + w_3 r_3 + w_4 r_4 + w_5 r_5$.

An exciting use of weights is presented in Pugh, Robins, Lipsitz, and Harrington (1993), for inference with missing covariate data. Let π_i be the probability that none of the covariates for subject i is missing, and let p_i be an indicator function which is 0 if any of the covariates actually is `NA`, so that $E(p_i) = \pi_i$. The usual strategy is to compute the Cox model fit over only the complete cases (those with $p_i = 1$). If information is not missing at random, this can lead to serious bias in the estimate of $\hat{\beta}$. A weighted analysis with weights of p_i / π_i corrects for this imbalance. There is an obvious connection between this idea and survey sampling; both reweight cases from underrepresented groups.

In practice, π_i is unknown and the authors suggest estimating it using a logistic regression with p_i as the dependent variable. The covariates for the logistic regression may be some subset of the Cox model covariates (those without missing information), as well as others. In an

example, the authors use a logistic model with follow-up time and status as the predictors. Let T be the matrix of score residuals from the logistic model, that is,

$$T_{ij} = \frac{\partial}{\partial \alpha_j}[p_i \log \pi_i(\alpha) + (1 - p_i)\log(1 - \pi_i(\alpha))] ,$$

where α are the coefficients of the fitted logistic regression. Then the estimated variance matrix for $\hat{\beta}$ is the sandwich estimator $I^{-1}BI^{-1}$, where

$$\text{¦} = U' U - [U' T][T' T]^{-1}[T' U.$$

This is equivalent to first replacing each row of U with the residuals from a regression of U on T, and then forming the product $U' U$. Note that if the logistic regression is completely uninformative (π_i = constant), this reduces to the ordinary sandwich estimate.

Computations

The `coxph` function is used to fit Cox proportional hazards models. The input data are assumed to consist of observations or rows of data, each of which contains the covariate values Z, a status indicator variable (1=event, 0=censored), an optional stratum indicator variable (referenced by the `strata` function), along with the time interval (*start*, *stop*] over which this information applies. This means that each row is treated as a separate subject whose Y_i variable is 1 on the interval (*start*, *stop*] and 0 otherwise. The risk set at time t only uses the applicable rows of the data.

The code for `coxph` does not specifically accommodate time-dependent covariates, time-dependent strata, multiple events, or any of the other special features mentioned. Consequently, *it is your responsibility to construct an appropriate data set.* This strategy leads to a fitting program that is simpler, shorter, easier to debug, and more computationally efficient than one with multiple specific options. A significantly more important benefit is that the flexibility inherent in building the proper data set allows analyses not originally considered; left truncation is a case in point.

The more common way to deal with time-dependent Cox models is to do a computation for each death time. For example, BMDP and SAS PHREG do this. One advantage of this over the algorithm implemented in `coxph` is the ability to code continuously varying time-dependent covariates. The `coxph` function only accommodates step functions. However, this does not appear to be a deficiency in practice. For the common case of repeated measurements on each subject, the data for `coxph` are quite easy to set up since they correspond to the original measurements of one line of data per visit.

The `coxph` function typically runs much faster when there are stratification variables in the model. When strata are introduced, `coxph` spends less time locating the current risk set because it only looks within the stratum it is estimating. If the start time is omitted, it is assumed to be zero for all cases. In this case the algorithm is equivalent to the standard Cox model.

REFERENCES

Andersen, P.K. & Gill, R.D. (1982). Cox's regression model for counting processes: A large sample study. *Annals of Statistics* **10**:1100-1120.

Binder, D.A. (1992). Fitting Cox's proportional hazards models from survey data. *Biometrika* **79**:139-147.

Chambers, J.M., Cleveland, W.S., Kleiner, B., & Tukey, P.A. (1983). *Graphical Methods for Data Analysis*. Belmont, CA: Wadsworth.

Cox, D.R. (1972). Regression models and life-tables. *Journal of the Royal Statistical Society, Series B* **34**:187-202.

Crowley, J. and Hu, M. (1977). Covariance analysis of heart transplant data. *Journal of the American Statistical Association* **72**:27-36.

Efron, B. (1977). The efficiency of Cox's likelihood function for censored data. *Journal of the American Statistical Association* **72**:557-565.

Efron, B. (1982). *The jackknife, the bootstrap, and other resampling plans*. Volume 38 of the CBMS-NSF Regional Conference Series in Applied Mathematics. SIAM.

Eilers, P.H. & Marx, B.D. (1996). Flexible smoothing with B-splines and penalties. *Statistical Science* **11**:89-121.

Fleming, T. & Harrington, D. (1991). *Counting Processes and Survival Analysis*. New York: John Wiley & Sons, Inc.

Grambsch, P. & Therneau, T.M. (1994). Proportional hazards tests and diagnostics based on weighted residuals. *Biometrika* **81**:515-526.

Gray R.J. (1992). Flexible methods for analyzing survival data using splines, with application to breast cancer prognosis. *Journal of the American Statistical Association* **87**: 942-951.

Harrell, F. (1986). *The PHGLM procedure*. SAS Supplemental Library User's Guide, Version 5. Cary, NC: SAS Institute, Inc.

Huber, P.J. (1967). The behavior of maximum likelihood estimates under non-standard conditions. *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability* **1**:221-233.

Kalbfleisch, J. & Prentice, R.L. (1980). *The Statistical Analysis of Failure Time Data*. New York: John Wiley & Sons, Inc.

Lawless J.F. & Singhal K. (1978). Efficient screening of nonnormal regression models. *Biometrics* **34**: 318-327.

Lee, E.W., Wei, L.J., and Amato, D. (1992). Cox-type regression analysis for large number of small groups of correlated failure time observations. In J.P Klein and P.K. Goel (Eds.), *Survival Analysis, State of the Art* (pp. 237-247). Netherlands: Kluwer Academic Publishers.

Lin, D.Y. and Wei, L.J. (1989). The robust inference for the Cox proportional hazards model. *Journal of the American Statistical Association* **84**:1074-1079.

Lin, D.Y. (1991). Goodness-of-fit analysis for the Cox regression model based on a class of parameter estimators. *Journal of the American Statistical Association* **86**:725-728.

Lin, D.Y., Wei, L.J., & Ying, Z. (1992). *Checking the Cox model with cumulative sums of martingale-based residuals.* Technical Report #111, Seattle: Dept. of Biostatistics, University of Washington.

McCullagh, P. & Nelder, J.A. (1989). *Generalized Linear Models* (2nd ed.) London: Chapman and Hall.

Nelson W.B. (1969). Hazard plotting for incomplete failure data. *Journal of Quality Technology* **1**:27-52.

Oakes, D. (1977). The asympotic information in censored survival data. *Biometrika* **64**:441-448.

Prentice, R.L., Williams, B.J., and Peterson, A.V. (1981). On the regression analysis of multivariate failure time data. *Biometrika* **68**:373-89.

Pugh, M., Robins, J., Lipsitz, S., and Harrington, D. (1993). *Inference in the Cox proportional hazards model with missing covariate data*, in press.

Schonfeld, D. (1982). Partial residuals for the proportional hazards regression mode. *Biometrika* **69**:239-241.

Smith, P.J. & Heitjan, D.F. (1993). Testing and adjusting for departures from nominal dispersion in generalized linear models. *Applied Statistics* **42**:31-41.

Therneau, T.M., Grambsch, P.M., & Fleming, T.R. (1990). Martingale-based residuals for survival models. *Biometrika* **77**:147-160.

Wei, L.J., Lin, D.Y., & Weissfeld, L. (1989). Regression analysis of multivariate incomplete failure time data by modeling marginal distributions. *Journal of the American Statistical Association* **84**: 1065-1073.

White, H. (1980). A heteroskedasticity-consistent covariance matrix estimator and a direct test for heteroskedasticity. *Econometrica* **48**:817-830.

White, H. (1982). Maximum likelihood estimation of misspecified models. *Econometrica* **50**:1-25.

PARAMETRIC REGRESSION IN SURVIVAL MODELS

29

INTRODUCTION

In contrast to the non-parametric (and semi-parametric) survival curve estimates of Kaplan-Meier, Fleming-Harrington, and Cox, among others, this chapter presents a parametric formulation to the estimation problem. Assume the survival time y satisfies $t(y) = X\beta + \sigma W$, where W follows some given distribution and t is a given transformation. For example, if t is the identity function and W is Gaussian, this corresponds to ordinary linear regression. The usual choice for t is $\log(y)$, which corresponds to an *accelerated failure time* (AFT) model. Using the log transform, if $\Lambda_w(t)$ is the cumulative hazard function for W, the cumulative hazard function for subject i is $\Lambda_w[\exp(-\eta/(X\beta))t]$. That is, the time scale for the subject is accelerated by a constant factor.

The development and use of parametric survival models actually predates that of the non-parametric methods. Although non-parametric methods now dominate in fields of study where the primary concern is to assess the risk of failure and its relation to covariates (for example, the effect of treatment arm on breast cancer recurrence), parametric methods are still vitally important in situations where extrapolation of results is necessary to predict failure rates under different conditions than those in the original study. A typical question addressed by non-parametric methodology is "How much does the risk of dying decrease if a new treatment is given to a lung cancer patient?" A typical question addressed by the parametric methodology in an accelerated testing setting is "What proportion of heaters will fail when run at 1100° F for 2 years, even though the original study ran heaters at temperatures ranging from 1520° to 1710° for only four months?"

In a manufacturing setting, studies of failure rates for new products cannot typically be done under normal operating conditions because they take too long to complete. Consequently, accelerated tests are conducted, exposing the product to more severe stresses than normal so that failures occur. Extrapolation is then used to estimate failure rates under normal operating conditions. In contrast, the Kaplan-Meier and Cox models do not extrapolate past the last observation. If the data are reasonably well modeled by one of the parametric

distributions, parametric models provide information for assessing properties of the baseline hazard function which the non-parametric models don't.

To perform parametric regression in Spotfire S+, you use the `survReg` function. The `survReg` function is similar to the `survreg` function available in earlier versions of SPOTFIRE S+, but has some new and modified arguments. The `survreg` function is still available, but is now deprecated.

As a simple example, consider the lung cancer data set included in Spotfire S+. We can fit a Weibull model to this data using `survReg` as follows:

```
> options(na.action = na.exclude)
> lung.survReg <- survReg(Surv(time, status) ~ age + sex +
+ ph.karno, data = lung, dist = "weibull")
> lung.survReg

Call:
survReg(formula = Surv(time, status) ~ age + sex +
     ph.karno, data = lung, dist = "weibull")

Coefficients:
 (Intercept)          age       sex    ph.karno
    5.326344 -0.008910282 0.3701786 0.009263843

Scale= 0.7551354

Loglik(model)= -1138.7   Loglik(intercept only)= -1147.5
  Chisq= 17.59 on 3 degrees of freedom, p= 0.00053
n=227 (1 observations deleted due to missing values)
```

STRATA

In a Cox model, the `strata` statement is used to allow separate baseline hazards for subgroups of the data, while retaining common coefficients for the other covariates across groups. For parametric models, the statement allows for a separate scale parameter for each subgroup, but again keeping the other coefficients common across groups. For instance, assume that separate baseline hazards are desired for males and females in the lung cancer data set. If we think of the intercept and scale as the baseline shape, an appropriate model can be fit as follows:

```
> lung.sfit <- survReg(Surv(time, status) ~ sex + age +
+ ph.karno + strata(sex), data = lung,
+ na.action = na.exclude)
> lung.sfit

Call:
survReg(formula = Surv(time, status) ~ sex + age + ph.karno
+ strata(sex), data = lung,
  na.action = na.exclude)

Coefficients:
 (Intercept)       sex          age   ph.karno
    5.059089 0.3566277 -0.006808082 0.01094966

Scale:
     sex=1     sex=2
 0.8165161 0.6222807

Loglik(model)= -1136.7   Loglik(intercept only)= -1146.2
  Chisq= 18.95 on 3 degrees of freedom, p= 0.00028
n=227 (1 observations deleted due to missing values)
```

The intercept-only model used for the likelihood ratio test has 3 degrees of freedom, corresponding to the intercept and two scales, as compared to the 6 degrees of freedom for the full model.

This is quite different from the effect of `strata` in `censorReg`, where it acts as a "by" statement and causes a completely separate model to be fit to each gender. The same fit (but not as nice a printout) can be obtained from `survReg` by adding an explicit interaction to the formula:

```
> survReg(Surv(time, status) ~ sex + (age +
+ ph.karno) * strata(sex), data = lung)
```

SPECIFYING A DISTRIBUTION

The `survReg` fitting routine is quite general, and can accept any distribution that spans the real line for W and any monotone transformation of y. The following distributions are included by default:

- exponential
- extreme
- Gaussian
- logistic
- Rayleigh
- t
- Weibull
- log Gaussian
- log logistic

RESIDUALS

The `residuals` method for parametric survival objects can return any of several types of residuals. This section describes the available types along with their strengths and weaknesses.

Response

Response residuals for other models such as `lm` or `glm` are defined as $y - \hat{y}$, where y is the observed data value. For censored data, some modifications must be made. If the observation is exact, y is the observed value; if the observation is left- or right-censored, the censoring value is used for y. One could argue that the returned residuals in this case should be marked as left- or right-censored, but this has not been done. For an interval-censored observation, y is chosen as the MLE from a fit with $n = 1$. That is, it is chosen so that the observed interval has the largest possible probability. For a symmetric distribution such as Gaussian or logistic, this is the center of the interval. However, it is somewhat more complicated for non-symmetric distributions such as the extreme value.

Response residuals are the default type:

```
> resid(lung.survReg)

         1         2         3         4         5
-48.57054 80.95766 593.7474 -202.5602 442.3329
         6         7         8         9        10
777.2215 -140.0104 -38.37526 -137.2232 -164.7835
        11        12        13        14        15
-206.0581 203.9896 186.3892 -233.2154 190.9419
        16        17        18        19        20
-199.9997 245.5643 437.0149 -395.4849 -324.5602
...
```

Deviance

Deviance residuals are response residuals transformed to the log-likelihood scale:

$$d_i = \text{sign}(r_i)\sqrt{LL(y_i, \hat{y}_0;\sigma) - LL(y_i, \eta;\sigma)} \ .$$

Here $\hat{y}_0$ is the unconstrained MLE for a fit with $n = 1$ (only the observation in question), but with σ fixed at its value from the overall fit. This leads to $\hat{y}_0 = -\infty$ and $+\infty$ for right- and left-censored observations, respectively. The first term under the square root is zero.

The advantages of the deviance residuals for plotting and outlier detection are nicely detailed in McCullagh and Nelder (1990). However, unlike GLMs, deviance residuals for interval-censored data are not free of the scale parameter. This means that if there are interval-censored data values and you fit two models A and B, the sum of the squared deviance residuals for model A minus the sum for model B does not equal the difference in log-likelihoods. This is one reason that the current `survReg` function does not inherit from class `"glm"`: the models created by `glm` use the deviance as the main summary statistic.

Deviance residuals are obtained by specifying `type="deviance"` in the call to `resid`:

```
> resid(lung.survReg, type = "deviance")

          1          2         3          4         5
-0.1889512 0.2711838 2.543388 -0.7786656 1.086197
          6          7          8          9         10
3.643226 -0.4560836 -0.1308644 -0.5838006 -0.7929039
         11         12         13        14         15
-0.895371 0.5395109 0.4189815 -1.46457 0.5978532
         16         17        18         19         20
-0.9683285 0.7638346 1.614463 -1.862741 -1.533163
...
```

Dfbeta

The `dfbeta` residuals are a matrix with one row per subject and one column per parameter. The *i*th row gives the approximate change in the parameter vector resulting from observation *i*; that is, it is the change in $\hat{\beta}$ when observation *i* is added to a fit based on all observations but the *i*th. The `dfbetas` residuals scale each column of the `dfbeta` matrix by the standard error of the respective parameter.

To obtain the `dfbeta` residuals, use `type="dfbeta"` in the call to `resid`. To obtain the `dfbetas` residuals, use `type="dfbetas"`:

```
> resid(lung.survReg, type = "dfbeta")

     (Intercept)            age            sex
1  0.01511630872 -1.133792e-004  0.00002623577
2 -0.00696784585  1.451185e-004 -0.00325004547
3  0.06865167740 -1.420568e-003 -0.01938509704
4 -0.01038268752  2.135163e-004  0.00380158171
5 -0.03436155488 -7.371198e-005 -0.01080367964
6  0.24197961727  2.394794e-003 -0.02658673104
...
        ph.karno     Log(scale)
1 -1.065334e-004 -0.00355606713
2  4.546980e-005 -0.00364612597
3  7.566163e-004  0.01453486059
4 -1.338031e-004 -0.00178635419
5  7.467109e-004  0.00219801001
6 -4.089797e-003  0.02732249126
...

> resid(lung.survReg, type = "dfbetas")

             [,1]          [,2]          [,3]
1  0.02280083554 -0.0159498488  0.0002050366
2 -0.01051002003  0.0204148405 -0.0253996072
3  0.10355144468 -0.1998413454 -0.1514975268
4 -0.01566083063  0.0300368545  0.0297099481
5 -0.05182959519 -0.0103695863 -0.0844324247
...
           [,4]          [,5]
1 -0.0238667512 -0.0576293685
2  0.0101866266 -0.0590888556
3  0.1695052040  0.2355509068
4 -0.0299759836 -0.0289495277
5  0.1672860906  0.0356207923
...
```

Working

The Newton-Raphson iteration used to solve the model can be viewed as an iteratively reweighted least-squares problem with a dependent variable of *current prediction-correction*. The working residual is the correction term. You can obtain the working residuals by specifying `type="working"` in the call to `resid`.

Likelihood Displacement

Escobar and Meeker (1982) define a matrix of likelihood displacement residuals for the accelerated failure time model. The full residual information is a square matrix A with dimension equal to the number of perturbations considered. Three examples are developed in detail, all with dimension n, the number of observations: the likelihood displacement residuals for a perturbation in the case weight for observation i (`ldcase`), a perturbation in the response value (`ldresp`), or a perturbation in the shape (`ldshape`).

Case weight perturbations measure the overall effect on the parameter vector of dropping a case. Let V be the variance matrix of the model, and let L the n by p matrix with elements $(\partial L_i) / (\partial \beta_j)$, where L_i is the likelihood contribution of the ith observation. Then $A = LVL'$. The `residuals` function with `type="ldcase"` returns the diagonal values of the matrix and LV equals the `dfbeta` residuals.

Response perturbations correspond to a change of one σ unit in one of the response values. For a Gaussian linear model, the equivalent computation yields the diagonal elements of the hat matrix. Shape perturbations measure the effect of a change in the log of the scale parameter by 1 unit. The `matrix` residual returns the raw values that can be used to compute these and other LD influence measures. The result is an $n \times 6$ matrix, containing columns for the following quantities:

$$L_i \qquad \frac{\partial L_i}{\partial \eta} \qquad \frac{\partial^2 L_i}{\partial \eta^2} \qquad \frac{\partial L_i}{\partial \log(\sigma)} \qquad \frac{\partial^2 L_i}{\partial \log(\sigma)^2} \qquad \frac{\partial^2 L_i}{\partial \eta \partial \log(\sigma)}$$

PREDICTED VALUES

The `predict` method for `survReg` objects allows several types of predictions. They fall into three groups: the linear predictor and predicted response, terms, and predicted quantiles.

Linear Predictor and Predicted Response

The linear predictor is $\eta = x'_i\hat{\beta}$, where x_i is the covariate vector for subject i and $\hat{\beta}$ is the final parameter estimate. The standard error of the linear predictor is $x'_i V x_i$, where V is the variance matrix for $\hat{\beta}$. You obtain the linear predictions by using `predict` with the argument `type="lp"`:

```
> predict(lung.survReg, type = "lp")

  [1] 5.870907 5.924369 6.031292 6.022382 6.088290
  [6] 5.500354 6.109271 5.989901 5.872746 5.801464
 [11] 5.929744 6.109271 6.294548 5.717736 5.929744
 [16] 5.840641 5.906548 5.598367 6.123556 6.022382
 . . .
```

The predicted response is identical to the linear predictor for fits to the untransformed distributions (the extreme-value, logistic, and Gaussian). For transformed distributions such as the Weibull, in which $\log(y)$ is from the extreme-value distribution, the linear predictor is on the transformed scale and the response is on the original scale of the data; this is $\exp(\eta)$ for the Weibull. The standard error of the transformed response is the standard error of η times the first derivative of the inverse transform.

The predicted response is the default prediction. You can ask for it explicitly by specifying `type="response"`.

```
> predict(lung.survReg)

  [1] 354.5705 374.0423 416.2526 412.5602 440.6671
  [6] 244.7785 450.0104 399.3753 355.2232 330.7835
 [11] 376.0581 450.0104 541.6108 304.2154 376.0581
 [16] 343.9997 367.4357 269.9851 456.4849 412.5602
 . . .
```

Terms

Predictions of type `terms` are useful for examination of terms in the model that expand into multiple dummy variables, such as factors and p-splines. The result is a matrix with one column for each of the terms in the model, along with an optional matrix of standard errors. Here is an example using p-splines on the `stanford2` data set:

```
> fit <- survReg(Surv(time,status) ~ pspline(age, df=3) +
+ t5, data = stanford2, dist = "lognormal",
+ na.action = na.exclude)

> tt <- predict(fit, type = "terms", se.fit = T)
> yy <- cbind(tt$fit[,1],
+ tt$fit[,1] - 1.96*tt$se.fit[,1],
+ tt$fit[,1] + 1.96*tt$se.fit[,1])

> matplot(stanford2$age, yy, type = "l", lty = c(1, 2, 2))
> plot(stanford2$age, stanford2$time, log = "y",
+ xlab = "Age", ylab = "Days", ylim = c(0.1,10000))
> matlines(stanford2$age, exp(yy+attr(tt$fit, "constant")),
+ lty = c(1, 2, 2))
```

The second plot, shown in Figure 29.2, puts the fit onto the scale of the data and thus is similar to Figure 1 in Escobar and Meeker (1982). Their plot is for a quadratic fit to age, without the T5 mismatch score in the model. For more details on p-splines, see the section Fitting Spline Models on page 314.

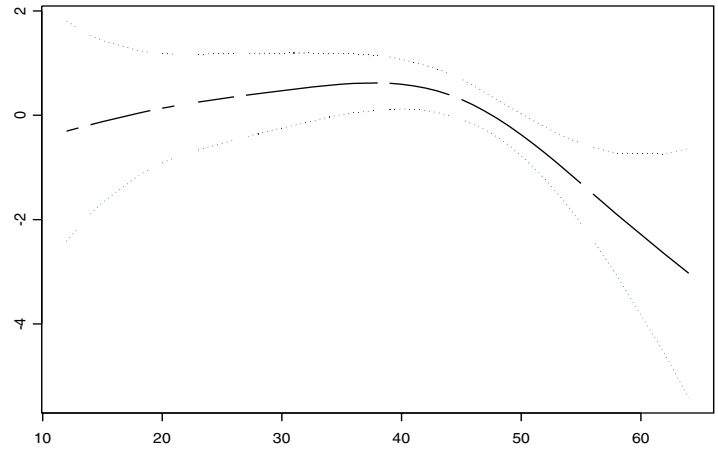

Figure 29.1: *Plot of p-spline fit with error bands.*

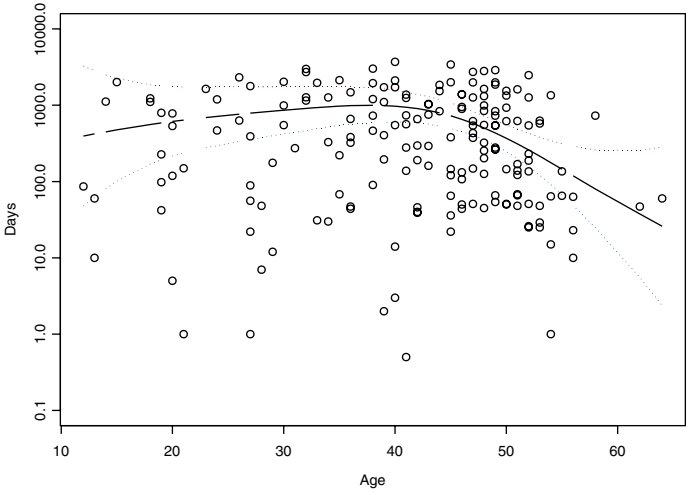

Figure 29.2: *Plot of p-spline fit with scale of the data.*

Quantiles

If predicted quantiles are desired, the set of probability of values p must also be given to the `predict` function. A matrix of n rows by p columns is returned, whose ijth element is the p_jth quantile of the predicted survival distribution, based on the covariates of subject i. This can be written as $X\beta + z_q\sigma$ where z_q is the qth quantile of the parent distribution. The variance of the quantile estimate is then cVc', where V is the variance matrix of (β, σ) and $c = (X, z_q)$.

In computing confidence bands for the quantiles, it may be preferable to add standard errors on the untransformed scale. You can do this using the `"uquantile"` prediction type. For example, consider the motor reliability data of Nelson and Hahn (1972, as cited in Kalbfleisch and Prentice, 1980). We first fit the standard quantile confidence intervals:

```
> fit <- survReg(Surv(time, status) ~ temp, data = motor)
> q1 <- predict(fit, data.frame(temp = 130),
+ type = "quantile", p = c(0.1, 0.5, 0.9), se.fit = T)

> ci1 <- cbind(q1$fit,
+ q1$fit - 1.96*q1$se.fit,
+ q1$fit + 1.96*q1$se.fit)

> dimnames(ci1) <- list(c(0.1, 0.5, 0.9), c("Estimate",
+ "Lower ci", "Upper ci"))

> round(ci1)

    Estimate Lower ci Upper ci
0.1    15935     9057    22812
0.5    29914    17395    42433
0.9    44687    22731    66643
```

Next we fit the standard errors on the untransformed scale:

```
> q2 <- predict(fit,data.frame(temp = 130),
+ type = "uquantile", p = c(0.1, 0.5, 0.9), se.fit = T)

> ci2 <- cbind(q2$fit,
+ q2$fit - 1.96 * q2$se.fit,
+ q2$fit + 1.96 * q2$se.fit)

> ci2 <- exp(ci2)
```

```
> dimnames(ci2) <- list(c(0.1, 0.5, 0.9), c("Estimate",
+ "Lower ci", "Upper ci"))

> round(ci2)

    Estimate Lower ci Upper ci
0.1    15935    10349    24535
0.5    29914    19684    45459
0.9    44687    27340    73041
```

Using the default Weibull method, the data are fit on the $\log(y)$ scale. The confidence bands obtained by the second method are asymmetric and may be more reasonable. They are also guaranteed to be positive.

The following example reproduces Figure 1 of Escobar and Meeker:

```
> plot(stanford2$age, stanford2$time, log = "y",
+ xlab = "Age", ylab = "Days", ylim = c(0.01, 10^6),
+ xlim = c(1, 65))

> fit <- survReg(Surv(time, status) ~ age + age^2,
+ data = stanford2, dist = "lognormal")

> qq <- predict(fit, newdata = list(age = 1:65),
+ type = "quantile", p = c(0.1, 0.5, 0.9))

> matlines(1:65, qq, lty = c(1, 2, 2))
```

The plot is shown in Figure 29.3. Note that the percentile bands on this figure are really quite a different object than the confidence bands on the spline fit. The latter reflect the uncertainty of the fitted estimate and are related to the standard error. The quantile bands reflect the predicted distribution of a subject at each given age, assuming no error in the quadratic estimate of the mean, and are related to the standard deviation of the population.

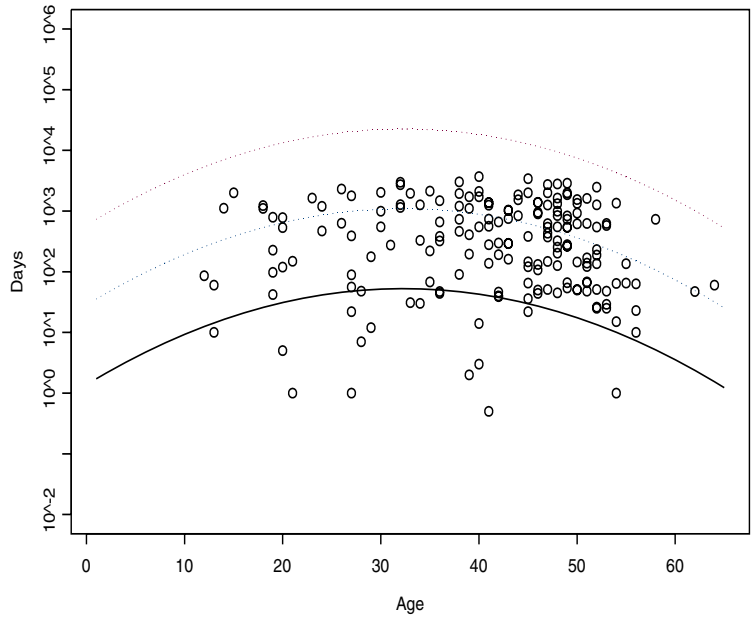

Figure 29.3: *Predicted 10th, 50th, and 90th survival quantiles for subjects at given age.*

FITTING THE MODEL

With some care, parametric survival can be formulated as an iteratively reweighted least squares (IRLS) problem used in Generalized Linear Models (GLM) of McCullagh and Nelder (1990). A detailed description of this setup for general maximum likelihood computation is found in Green (1984).

Let y be the response vector and x_i be the vector of covariates for the ith observation. Assume that

$$z_i \equiv \frac{t(y_i) - x'_i\beta}{\sigma} \sim f \tag{29.1}$$

for some distribution f, where y may be censored and t is a differentiable transformation function. The likelihood for $t(y)$ is

$$= \left(\prod_{\text{exact}} \frac{(f(z_i))}{\sigma} \right) \left(\prod_{\text{right}} \int_{z_i}^{\infty} f(u)du \right) \left(\prod_{\text{left}} \int_{-\infty}^{z_i} f(u)du \right) \left(\prod_{\text{interval}} \int_{z_i^*}^{z_i} f(u)du \right),$$

where *exact*, *right*, *left*, and *interval* refer to uncensored, right-censored, left-censored, and interval-censored observations, respectively. The z_i^* term is the lower endpoint of a censoring interval. The log-likelihood is defined as

$$\log(l) = \sum_{\text{exact}} g_1(z_i) - \log(\sigma) + \sum_{\text{right}} g_2(z_i) + \sum_{\text{left}} g_3(z_i) + \sum_{\text{interval}} g_4(z_i) \tag{29.2}$$

Derivatives of the log-likelihood with respect to the regression parameters are

$$\frac{\partial \log(l)}{\partial \beta_j} = \sum_{i=1}^{n} x_{ij} \frac{\partial g}{\partial n_i} \tag{29.3}$$

$$\frac{\partial^2 \log(l)}{\partial \beta_j \beta_k} = \sum x_{ij} x_{ik} \frac{\partial^2 g}{\partial \eta^2} \tag{29.4}$$

where $\eta = X'\beta$ is the vector of linear predictors.

Thus, if we treat σ as fixed, iteration is equivalent to IRLS with weights of $-g''$ and adjusted dependent variable of $\eta - g'/g''$. The Newton-Raphson step defines an update δ by

$$(X^T D X)\delta = X^T U \tag{29.5}$$

where D is the diagonal matrix formed from $-g''$ and U is the vector g'. The current estimate β satisfies $X\beta = \eta$, so that the new estimate $\beta + \delta$ will have

$$(X^T D X)(\beta + \delta) = X^T D\eta + X^T U = (X^T D)(\eta + D^{-1}U) \tag{29.6}$$

At the solution to the iteration, we might expect that $\hat{\eta} \approx y$. In fact, weighted regression with y replacing η gives, in general, good starting estimates for the iteration; for an interval-censored observation, we use the center of the interval as y. If all the observations are uncensored, this reduces to using the linear regression of y on X as a starting estimate: $y = \eta$ so $z = 0$, thus $g' = 0$ and g'' = a constant (all of the supported densities have a mode at 0).

This clever starting estimate is introduced in McCullagh and Nelder, and works extremely well in that context: convergence often occurs in 3–4 iterations. It does not work quite so well here, since a "good" fit to a right-censored observation might have $\eta \ll y$. Secondly, the other coefficients are not independent of σ, and σ often appears to be the most touchy variable in the iteration.

Most often, the parametric survival functions are used with $\log(y)$, which corresponds to the set of accelerated failure time models. The transform can be applied implicitly or explicitly. For example, the following two fits give identical coefficients:

```
> fit1 <- survReg(Surv(futime, fustat) ~ age + rx,
+ data = ovarian, dist = "weibull")

> fit2 <- survReg(Surv(log(futime), fustat) ~ age + rx,
+ data = ovarian, dist = "extreme")
```

The log-likelihoods for the two fits differ by a constant,the sum of $(d\log(y))/(dy)$ for the uncensored observations. In addition, certain predicted values and residuals will be on the y versus $\log(y)$ scale.

Derivatives of the Log-Likelihood

This section is very similar to the appendix of Escobar and Meeker, differing only in the use of $\log(\sigma)$ rather than σ as the natural parameter. Let f and F denote the density and cumulative distribution functions, respectively, of one of the parametric survival distributions. Using Equation (29.2) for defining $g_1, \ldots, g_4$, we have the equations listed below.

$$\frac{\partial g_1}{\partial \eta} = -\frac{1}{\sigma}\left[\frac{f'(z)}{f(z)}\right]$$

$$\frac{\partial g_4}{\partial \eta} = -\frac{1}{\sigma}\left[\frac{f(z^u) - f(z^l)}{F(z^u) - F(z^l)}\right]$$

$$\frac{\partial^2 g_1}{\partial \eta^2} = \frac{1}{\sigma^2}\left[\frac{f'(z)}{f(z)}\right] - \left(\frac{\partial g_1}{\partial \eta}\right)^2$$

$$\frac{\partial^2 g_4}{\partial \eta^2} = \frac{1}{\sigma^2}\left[\frac{f'(z^u) - f'(z^l)}{F(z^u) - F(z^l)}\right] - \left(\frac{\partial g_4}{\partial \eta}\right)^2$$

$$\frac{\partial g_1}{\partial \log\sigma} = -\left[\frac{z f'(z)}{f(z)}\right]$$

$$\frac{\partial g_4}{\partial \log\sigma} = -\left[\frac{z^u f(z^u) - z^l f(z^l)}{F(z^u) - F(z^l)}\right]$$

$$\frac{\partial^2 g_1}{\partial (\log\sigma)^2} = \left[\frac{z^2 f''(z) + z f'(z)}{f(z)}\right] - \left(\frac{\partial g_1}{\partial \log\sigma}\right)^2$$

$$\frac{\partial^2 g_4}{\partial (\log\sigma)^2} = \left[\frac{(z^u)^2 f'(z^u) - (z^l)^2 f'(z^l) + [z^u f(z^u) - z^l f(z^l)]}{F(z^u) - F(z^l)}\right] - \frac{\partial g_1}{\partial \log\sigma}\left(1 + \frac{\partial g_1}{\partial \log\sigma}\right)$$

$$\frac{\partial^2 g_1}{\partial \eta \partial \log\sigma} = \frac{z f''(z)}{\sigma f(z)} - \frac{\partial g_1}{\partial \eta}\left(1 + \frac{\partial g_1}{\partial \log\sigma}\right)$$

$$\frac{\partial^2 g_4}{\partial \eta \partial \log\sigma} = \frac{z^u f'(z^u) - z^l f'(z^l)}{\sigma[F(z^u) - F(z^l)]} - \frac{\partial g_4}{\partial \eta}\left(1 + \frac{\partial g_4}{\partial \log\sigma}\right)$$

To obtain the derivatives for g_2, set the upper endpoint z^u to ∞ in the equations for g_4. To obtain the equations for g_3, left-censored data, set the lower endpoint to $-\infty$. The internal iteration is done in terms of $\log(\sigma)$, which avoids the boundary condition at zero and helps the iteration speed considerably for some test cases.

By the chain rule:

$$\frac{\partial LL}{\partial \log\sigma} = \sigma\frac{\partial LL}{\partial \sigma}$$

$$\frac{\partial^2 LL}{\partial (\log\sigma)^2} = \sigma^2\frac{\partial^2 LL}{\partial \sigma^2} + \sigma\frac{\partial LL}{\partial \sigma}$$

$$\frac{\partial^2 LL}{\partial \eta \partial \log\sigma} = \sigma\frac{\partial^2 LL}{\partial \eta \partial \sigma}$$

At the solution, $\partial LL / \partial\sigma = 0$ so the variance matrix for σ is a simple scale change of the returned matrix for $\log(\sigma)$.

DISTRIBUTIONS

The presentation of the distributions contained in this section is similar to that in Nelson (1982). Derivatives of the terms in the log-likelihood, Equation (29.2), are presented following the details for each distribution.

For each distribution, the *standardized variable* z is defined by Equation (29.1), where $\eta = x'_i\beta$ is the linear predictor and σ is the scale parameter. The details for each distribution are written in terms of the standardized variable, z.

Gaussian

This is perhaps the most frequently used distribution in applied statistics. It is more commonly known as the *normal* distribution. The continual calls to Φ may make it slow on censored data, however. The standardized variable z has mean 0 and variance 1. The standard normal distribution is then defined by

$$\begin{aligned} \Phi(z) &= \Phi(z) \\ f(z) &= \exp(-z^2/2)/(\sqrt{2\pi}) \\ f'(z) &= -zf(z) \\ f''(z) &= (z^2-1)f(z) \end{aligned}$$

The derivatives of the terms in the log-likelihood are given by

$$\begin{aligned} g_1 &= -z^2/2 - \log(\sqrt{2\pi}) \\ g'_1 &= -z \\ g''_1 &= -1 \\ g_2 &= \log(1-\Phi(z)) \\ g'_2 &= -(f(z))/(1-\Phi(z)) \\ g''_2 &= (-f'(z))/((1-\Phi(z))-(g'_2)^2) \end{aligned}$$

For uncensored data, the "standard" GLM results are obtained by substituting g_1 into Equations (29.2) through (29.6). The first derivative vector is equal to $X'r$ where $r = -z/\sigma$ is a scaled residual,

the update step $D^{-1}U$ is independent of the estimate of σ, and the maximum likelihood estimate of $n\sigma$ is the sum of squared residuals. None of these hold so neatly for right-censored data.

Least Extreme Value

If y has a Weibull distribution, $\log(y)$ is distributed according to the least extreme value distribution. Fits on the latter scale are numerically preferable because they remove the range restriction on y. A Weibull distribution with the scale constrained to be 1 gives an exponential model.

The standardized variable z defined by Equation (29.1) has mean 0.5722 and variance $\pi^2/6$. Let $w = e^z$. Then the standard least extreme value distribution is defined as

$$\begin{aligned} F(z) &= 1 - e^{-w} \\ f(z) &= we^{-w} \\ f'(z) &= (1-w)f(z) \\ f''(z) &= (w^2 - 3w + 1)f(z) \end{aligned}$$

The derivatives for the terms in the log-likelihood, Equation (29.2), are given by:

$$\begin{aligned} g_1 &= z - w & g_2 &= -w & g_3 &= \log(1 - e^{-w}) \\ g'_1 &= w - 1 & g'_2 &= w & g'_3 &= -(we^{-w})/(1 - e^{-w}) \\ g''_1 &= -w & g''_2 &= -w & g''_3 &= -\frac{(we^{-w}(1-w))}{(1-e^{-w})} - (g'_3)^2 \end{aligned}$$

The mode of the distribution is at $z = 0$ with $f(0) = 1/e$. For an exact observation, the deviance term has $\hat{y} = y$. For interval-censored data where the interval is of length $b = z^u - z^l$, most mass is covered if the interval has a lower endpoint of

$$a = \log(b/(e^{b-1})),$$

so that the resulting log-likelihood is

$$\log(e^{-e^a} - e^{-e^{a+b}}).$$

Logistic

This distribution is very close to the Gaussian except in the extreme tails, but it is far easier to work with. All the computations are closed form. However, some data sets may contain survival times close to zero, leading to differences in fit between the lognormal and log-logistic choices. In such cases, the rationality of a Gaussian fit may also be in question.

The standardized variable z defined by Equation (29.1), has mean 0 (zero) and variance $\pi^2/3$. Again, let $w = e^z$. The standard logistic distribution is defined by

$$\begin{aligned} F(z) &= w/(1+w) \\ f(z) &= w/(1+w)^2 \\ f'(z) &= f(z)((1-w)/(1+w)) \\ f''(z) &= f(z)((w^2-4w+1)/(1+w)^2) \end{aligned}$$

The derivatives for the terms in the log-likelihood, Equation (29.2), are given by:

$$g_1 = z - 2\log(1+w) \qquad g_2 = -\log(1+w) \qquad g_3 = z - \log(1+w)$$

$$g'_1 = \frac{(w-1)}{(w+1)} \qquad g'_2 = \frac{w}{(1+w)} \qquad g'_3 = -\frac{1}{1+w}$$

$$g''_1 = -\frac{2w}{(1+w)^2} \qquad g''_2 = -\frac{w}{(1+w)^2} \qquad g''_3 = -\frac{w}{(1+w)^2}$$

The distribution is symmetric about 0, so for an exact observation the contribution to the deviance term is $-\log(4)$. For an interval-censored observation with span $2b$ the contribution is

$$\log(F(b) - F(-b)) = \log\left(\frac{e^b - 1}{e^b + 1}\right).$$

Other Distributions

Some other population hazards can be fit into this location-scale framework, while others cannot.

Distribution	Hazard
Weibull	$p\lambda(\lambda t)^{p-1}$
Extreme value	$(1/\sigma)e^{(t-\eta)/\sigma}$
Rayleigh	$a + bt$
Gompertz	bc^t
Makeham	$a + bc^t$

We can see that an extreme value distribution on $t' = \log(t))$ is equivalent to a Weibull hazard on t, with $p = 1/\sigma$.

The Makeham hazard $a + bc^t$ seems to fit human mortality experience beyond infancy quite well. Here a is a constant mortality that is independent of the health of the subject (accidents, homicide, etc.), and the second term models the Gompertz assumption that "the average exhaustion of a man's power to avoid death is such that, at the end of equal infinitely small intervals of time, he has lost equal portions of his remaining power to oppose destruction which he had at the commencement of these intervals." For older ages, a is a negligible portion of the death rate and the Gompertz model holds.

The next two statements follow from the form of the hazards in the table:

- The Weibull distribution with $p = 2$ $(\sigma = 0.5)$ is the same as a Rayleigh distribution with $a = 0$. It is not, however, the most general form of a Rayleigh.
- The extreme value and Gompertz distributions have the same hazard function, with $\sigma = 1/(\log(c))$ and $\exp(-\eta/\sigma) = b$.

On first glance, it appears that the Gompertz can be fit with an identity link function combined with the extreme value distribution, but this ignores a boundary restriction. If $f(x;\eta\ \sigma)$ is the extreme value distribution with parameters η and σ, the definition of the Gompertz density is

$$\begin{aligned} g(x;\eta\ \sigma) &= 0 & x < 0 \\ g(x;\eta\ \sigma) &= cf(x;\eta\ \sigma) & x \geq 0 \end{aligned}$$

where $c = \exp(\exp(-\eta/\sigma))$ is the constant necessary so that g integrates to 1. If η/σ is far from 1, the correction term is minimal and `survReg` should give a reasonable fit to Gompertz data. If not, the distribution cannot be made to easily conform to the general fitting scheme of the function. The `censorReg` function, however, can fit the data using the `truncation` argument to specify that each observation is restricted to $(0, \infty)$.

The Makeham distribution falls into the gamma family (equation 2.3 of Kalbfleisch and Prentice) but with the same range restriction problem.

A FINAL EXAMPLE

The `capacitor` data frame contains data from a simulated life testing of capacitors from Meeker and Duke (1982). The `capacitor` data frame is close enough to the data modeled in Nelson (1990), page 302, that it works as a verification data set. The variables in `capacitor` are:

- `days`, time to failure
- `event`, indicator of failure (1) or censoring (0)
- `voltage`, voltage at which the test was run

A summary of this data frame follows:

```
> summary(capacitor)

      days            event          voltage
 Min.   :  0.68   Min.   :0.000   Min.   :20.00
 1st Qu.: 73.87   1st Qu.:0.000   1st Qu.:26.00
 Median :300.00   Median :0.000   Median :26.00
 Mean   :205.20   Mean   :0.432   Mean   :26.72
 3rd Qu.:300.00   3rd Qu.:1.000   3rd Qu.:29.00
 Max.   :300.00   Max.   :1.000   Max.   :32.00
```

You fit a Weibull model to the `capacitor` data as follows:

```
> capac.fit1 <- survReg(Surv(days, event) ~ voltage,
+ data = capacitor)
```

You don't have to specify the distribution in this case because `survReg` defaults to `dist="weibull"`.

Printing the resulting fit produces the following display:

```
> capac.fit1

Call:
survReg(formula = Surv(days, event) ~ voltage,
     data = capacitor)

Coefficients:
 (Intercept)     voltage
    24.13993 -0.6403297
```

```
Scale= 1.203916

Loglik(model)= -316.5   Loglik(intercept only)= -372.8
  Chisq= 112.61 on 1 degrees of freedom, p= 0
n= 125
```

The summary of the fit object is shown below:

```
> summary(capac.fit1)

Call:
survReg(formula = Surv(days, event) ~ voltage, data =
capacitor)
              Value Std. Error     z         p
(Intercept)  24.140     2.4493  9.86 6.48e-023
    voltage  -0.640     0.0811 -7.89 2.93e-015
 Log(scale)   0.186     0.1113  1.67 9.54e-002

Scale= 1.2

Weibull distribution
Loglik(model)= -316.5   Loglik(intercept only)= -372.8
  Chisq= 112.61 on 1 degrees of freedom, p= 0
Number of Newton-Raphson Iterations: 5
n= 125

Correlation of Coefficients:
            (Intercept) voltage
   voltage -0.998
Log(scale)  0.560      -0.559
```

Voltage is clearly quite significant in the model. McCullagh and Nelder discuss the utility of deviance residual plots in assessing the fit of a model. The following code constructs the plot of deviance residuals versus the logged fitted values displayed in Figure 29.4.

```
> plot(log(fitted(capac.fit1)), resid(capac.fit1,
+ type = "deviance"))
```

The example in Nelson (1990), page 302, displays a Weibull model with the logged scale parameter, $\log_e(\alpha)$, modeled as a linear function of $\log_e(\text{voltage})$. We fit and display a partial summary of this second model as follows.

```
> capac.fit2 <- survReg(Surv(days, event) ~ log(voltage),
+ data = capacitor)

> summary(capac.fit2)

Call:
survReg(formula = Surv(days, event) ~ log(voltage), data =
capacitor)
                Value Std. Error     z         p
 (Intercept)  67.945      8.151  8.34 7.71e-017
log(voltage) -18.546      2.396 -7.74 9.81e-015
  Log(scale)   0.191      0.111  1.71 8.67e-002

Scale= 1.21

Weibull distribution
Loglik(model)= -316.4   Loglik(intercept only)= -372.8
  Chisq= 112.71 on 1 degrees of freedom, p= 0
Number of Newton-Raphson Iterations: 6
n= 125

Correlation of Coefficients:
             (Intercept) log(voltage)
log(voltage) -1.000
  Log(scale)  0.543      -0.542
```

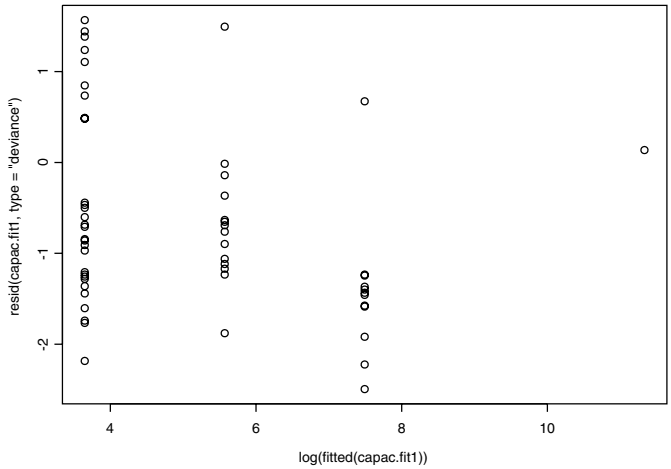

Figure 29.4: *Deviance residuals versus fitted values for a model of capacitor failure times versus voltage.*

REFERENCES

Escobar, L.A. & Meeker, Jr., W.Q. (1982). Assessing influence in regression analysis with censored data. *Biometrics* **48**:507-528.

Green, P.J. (1984). Iteratively reweighted least squares for maximum likelihood estimation, and some robust and resistant alternatives (with discussion). *Journal of the Royal Statistical Society, Series B* **46**:149-192.

Kalbfleisch, J. & Prentice, R.L. (1980). *The Statistical Analysis of Failure Time Data*. New York: John Wiley & Sons, Inc.

McCullagh, P. & Nelder, J.A. (1990). *Generalized Linear Models* (2nd ed.). London: Chapman and Hall.

Meeker, Jr., W.Q., & Duke, S.D. (1982). *User's Manual for CENSOR–A User-Oriented Computer Program for Life Data Analysis*. Statistical Laboratory, Ames, Iowa: Iowa State University.

Nelson, W. (1982). *Applied Life Data Analysis*. New York: John Wiley & Sons, Inc.

Nelson, W. (1990). *Accelerated Life Testing*. New York: John Wiley & Sons, Inc.

Nelson, W.B. & Hahn, G.J. (1972). Linear estimation of a regression relationship from censored data, part 1: Simple methods and their applications (with discussion). *Technometrics* **14**: 247-276.

LIFE TESTING

30

INTRODUCTION

Parametric regression models for censored data are used in a variety of contexts, ranging from manufacturing to studies of environmental contaminants. Because of their frequent use for modeling failure time or survival data, they are often referred to as *parametric survival* models. In this context, they are used throughout engineering to discover reasons why engineered products fail. They are called *accelerated failure time* models or *accelerated testing* models when the product is tested under more extreme conditions than normal to accelerate its failure time.

Most product engineering can't wait long enough to observe ample failures for fitting models under normal operating conditions. The results obtained under extreme conditions are related to the results that would be obtained when the product is subject to normal wear. Thus, for example, capacitors may be operated under higher temperatures and voltages than normal to increase their likelihood of failure. The resulting fitted model is used to extrapolate failure rates back to normal operating conditions. Similar use is made of these failure time distributions in the context of survival analysis, where living organisms rather than engineered products are the primary interest.

In the context of environmental studies, the measures of interest may be chemical contaminant levels rather than failure times, but these data are frequently censored or obtained from truncated distributions. Censored and/or truncated data regression methodology applies equally well in these cases but, of course, the values of interest have nothing to do with survival.

Model selection is a major concern when using censored regression models. As in other model fitting activities, the distributional assumptions that are made must be appropriate for the data collected, and the model must also reasonably account for variation in the independent variables. Consequently, visual comparisons of the predicted distribution of the response with nonparametric estimates of the distribution is an important activity when fitting models. To obtain the most appropriate model, usually a number of models with different failure distributions and/or dependence relationships with

the independent variables are fitted and compared. Visual comparison and statistical tests are then used to determine the most appropriate model.

Given that a model has been obtained, the results may be extrapolated to new values for the independent variables, and inference procedures may be used to obtain interval estimates for failure probabilities or quantiles of the response. In doing this, the usual precautions apply: one should not try to extrapolate model information too far beyond the values collected in the data. Moreover, because the interval estimate procedures are asymptotic, the confidence levels should be treated as approximate, especially in small samples.

In this chapter we discuss a set of functions for the analysis of censored and/or truncated data or, more specifically, for the analysis of accelerated failure time and survival data. These functions are based upon estimation code originally developed by Meeker and Duke (1981) and refined subsequently by W.Q. Meeker (personal communication). This estimation code has been modified slightly for inclusion in the Spotfire S+ product. The Spotfire S+ code that calls the underlying estimation routines borrows from work done by both W.Q. Meeker and Terry Therneau. For further reading on analyzing accelerated test data see Nelson (1990) or Meeker and Escobar (1998).

Taken as a whole, the Spotfire S+ functions we discuss in this chapter allow you to easily specify and fit censored data models. They allow you to graph and compare the fitted models with appropriate nonparametric estimates of these models. You can also make inferences regarding the model parameters, predicted failure probabilities, and quantiles. We begin by briefly discussing the nonparametric estimates and how they may be computed. This brief introduction is followed by a complete discussion of the model fitting software for censored data with emphasis on accelerated failure time models. We then discuss the ANOVA function, which can be used to compare one or more fitted models, and we describe the various visualizations that can be performed once a model has been fit. In the final sections of this chapter, we discuss the estimation of quantiles and failure probabilities at various points for selected values of the independent variables.

The censorReg function discussed in this chapter supersedes the survreg function available in previous versions of Spotfire S+, as it provides more extensive parametric survival capabilities. The censor function is a new function for use in formulas, and specifies censoring codes in a more general way than does the Surv function. The kaplanMeier function is a companion function to survfit, providing Kaplan-Meier estimates for survival models specified by the censor function in censorReg.

THE GENERALIZED KAPLAN-MEIER ESTIMATE

The Kaplan-Meier estimator produces nonparametric estimates of failure probability distributions for a single sample of data that either contains the exact time of failure, or is *right-censored*. A right-censored observation is one in which the failure time is only known to be greater than the time it was removed, or *censored*, from the study. Because we consider data that may be left-censored, observed in an interval, and/or grouped as well, we use a generalization of the Kaplan-Meier estimate originally developed by Turnbull (1974, 1976).

Specifying Interval Censored Data

Consider the following (artificial) table of failure times:

Table 30.1: *Failure time format.*

Unit	Failure	Upper	Censor	Censor Codes
1	7	NA	right	0
2	4	NA	exact	1
3	5	NA	exact	1
4	9	NA	right	0
5	3	NA	left	2
6	2	9	interval	3
7	7	12	interval	3
8	4	NA	exact	1
9	11	NA	right	0

First we define what we mean by the censoring types. Let $C = (L,U)$ be a random censoring interval, let T be the failure time, and suppose that C and T are independent. Note that less strict assumptions are possible; see, for example, Andersen, *et al.*, 1993. An observation is an *exact failure* if the failure time T is observed so that $T < L$. The observation is *right-censored* if the censoring time L is observed so that $T > L$. The observation is *interval-censored* if all that is known is that $L \leq T < U$. Finally, the observation is *left-censored* if all that is known is that $0 \leq T < U$; that is, the observation is interval-censored with a lower censoring time of zero.

In Spotfire S+, a *censoring code* indicates the type of censoring. Censoring codes are handled quite generally, allowing you to specify a set of values for each type of censoring. The default codes are 0 for right-censored observations, 1 for exact failures, 2 for left-censored observations, and 3 for interval-censored observations. To specify a censored distribution dependent variable, you must give both the time of failure or censoring, and except in exact failure (or complete) data, the censoring code. The Spotfire S+ function `censor` is used to specify the dependent variable. For the data in Table 30.1, the correct specification is:

```
> unit <- c(1:9)
> failure <- c(7, 4, 5, 9, 3, 2, 7, 4, 11)
> upper <- c(rep(NA,5), 9, 12, NA, NA)
> Censor <- c("right", "exact", "exact", "right",
+ "left", "interval", "interval", "exact", "right")

> censor.codes <- c(0, 1, 1, 0, 2, 3, 3, 1, 0)
> censor(failure, upper, censor.codes)

[1]  7+    4     5     9+    3-    [ 2,  9] [ 7, 12]
[8]  4       11+
```

When three arguments are specified to `censor`, the default censoring type is "interval." To show the generality of the `censor` function, an alternate way of specifying the censor codes is by using the `Censor` column and stating explicitly what the codes are:

```
> cens <- censor(failure, upper, Censor, event = "exact",
+ right = "right", left = "left", interval = "interval")
> cens
```

```
[1]  7+    4     5     9+    3-   [ 2,  9] [ 7, 12]
[8]  4       11+
```

While this is lengthier command, it is far more general and allows you to specify a vector of codes for each of the four censoring types.

It is always a good idea to display the output from the `censor` function to verify that you have correctly specified the censoring information. This is especially important because it is common practice to reverse the censoring codes for exact failures and right-censored observations; these values must be correctly specified if the analysis is to be meaningful. An additional check you can do is to examine the censor codes map as follows:

```
> censorCodesMap(cens)

   event: exact ==> 1
   right: right ==> 2
    left: left ==> 3
interval: interval ==> 4
```

The internal codes 1, 2, 3, and 4 are used by the estimation routine.

The `outCodes` argument to `censor` allows you to use it with the `coxph`, `survreg`, and `survfit` routines, which require internal codes of 1 (event), 0 (right), 2 (left), and 3 (interval). Setting the `outCodes` argument to `"0-3"` results in the internal codes that the three survival functions require:

```
> cens <- censor(failure, upper, Censor, right = "right",
+ left = "left", event = "exact", interval = "interval",
+ outCodes = "0-3")

> censorCodesMap(cens)

   event: exact ==> 1
   right: right ==> 0
    left: left ==> 2
interval: interval ==> 3
```

The outCodes argument allows you to generate output that is equivalent to the output from Surv. This allows you to pass a censor object to those functions that require an object of class "Surv". A simple example shows the idea. You can fit a model using coxph with the following call to censor:

```
> coxph(censor(time, status, outCodes = "0-3") ~ age + sex,
+ data = lung)
```

When you specify outCodes="0-3", not only are the output codes set accordingly, but the return value of censor inherits from Surv, which is required by coxph. You can also go the other way, from Surv to censor, by selecting each column of a Surv object to pass to censor.

Computing Kaplan-Meier Estimates

The kaplanMeier function is used to compute Kaplan-Meier estimates and Turnbull's generalization of the Kaplan-Meier estimates. It generalizes survfit by allowing left- and interval-censored data, and it uses the same formula specification as the censorReg function discussed later in this chapter. For the data in Table 30.1, use the following Spotfire S+ statements to create a data frame called int.data and compute Kaplan-Meier estimates with standard confidence intervals:

```
> int.data <- data.frame(unit, failure, upper, Censor,
+ censor.codes)

> int.data

  unit failure upper   Censor censor.codes
1    1       7    NA    right            0
2    2       4    NA    exact            1
3    3       5    NA    exact            1
4    4       9    NA    right            0
5    5       3    NA     left            2
6    6       2     9 interval            3
7    7       7    12 interval            3
8    8       4    NA    exact            1
9    9      11    NA    right            0

> kaplanMeier(censor(failure, upper, censor.codes) ~ 1,
+ data = int.data, conf.interval="identity")

Number Observed: 9
Number Censored: 6
```

```
Confidence Type: identity
            Survival Std.Err 95% LCL 95% UCL
(-Inf,   2]    1.000   0.000   1.000   1.000
(   3,   4]    0.861   0.127   0.612   1.000
(   4,   5]    0.583   0.173   0.244   0.922
(   5,   7]    0.444   0.166   0.120   0.769
(   9,  11]    0.444   0.166   0.120   0.769
(  12, Inf)    0.000   0.000      NA      NA
```

In the output, each row begins with a label indicating the observation interval. The time interval is followed by the survival estimate, the standard error for the estimate, and approximate confidence intervals for the estimate.

The `kaplanMeier` model computed above estimates the survival curve for a single sample. If independent variables were available in the sample, the values of all the independent variables would have to be identical to obtain meaningful results from `kaplanMeier`. If an independent variable is used on the right side of a formula, it is treated as a stratification variable and separate survival curves are estimated for each value.

Consider the `capacitor2` data set distributed with Spotfire S+. This data set contains four variables:

- `days`, the time of failure or censoring.
- `event`, the censoring code. A value of 1 is a failure at time `days` while 0 is right-censoring at time `days`.
- `weights`, the number of observations represented by that row.
- `voltage`, the voltage at which the capacitor was tested. There are four distinct voltages in the data set.

To analyze the failure date without regard to the test voltage, the following statement is used:

```
> kaplanMeier(censor(days, event) ~ 1, weights = weights,
+ data = capacitor2)
```

However, this model ignores the different test voltages. An alternate analysis computes a nonparametric estimate of the failure time for each voltage. This is done with the statements:

```
> km.cap <- kaplanMeier(censor(days, event) ~ voltage,
+ weights = weights, data = capacitor2,
```

```
+ conf.interval="identity")

> km.cap

voltage=20
Number Observed: 25
Number Censored: 25
[1] Not enough failures available to fit a nonparametric
censored data model

voltage=26
Number Observed: 50
Number Censored: 39
Confidence Type: identity
                 Survival Std.Err 95% LCL 95% UCL
(  -Inf,  12.95]     1.00   0.000   1.000   1.000
( 12.95,  28.41]     0.98   0.020   0.941   1.000
( 28.41,  63.10]     0.96   0.028   0.906   1.000
( 63.10, 136.33]     0.94   0.034   0.874   1.000
(136.33, 139.37]     0.92   0.038   0.845   0.995
(139.37, 179.02]     0.90   0.042   0.817   0.983
(179.02, 187.80]     0.88   0.046   0.790   0.970
(187.80, 201.28]     0.86   0.049   0.764   0.956
(201.28, 214.28]     0.84   0.052   0.738   0.942
(214.28, 271.73]     0.82   0.054   0.714   0.926
(271.73, 277.33]     0.80   0.057   0.689   0.911
(277.33, 300.00]     0.78   0.059   0.665   0.895

voltage=29
Number Observed: 20
Number Censored: 7
Confidence Type: identity
                 Survival Std.Err 95% LCL 95% UCL
(  -Inf,  10.21]     1.00   0.000   1.000   1.000
( 10.21,  40.69]     0.95   0.049   0.854   1.000
( 40.69,  45.85]     0.90   0.067   0.769   1.000
. . .
```

For `voltage=20`, there are not enough observations in the sample to compute estimates. For `voltage=26`, `voltage=29`, and `voltage=32`, estimates are computed and displayed in separate tables.

The Kaplan-Meier estimates of failure probabilities can also be used to compute nonparametric estimates of the quantiles. For example, the statement

```
> qkaplanMeier(km.cap, p = seq(from=0.1, to=0.9, by=0.1))
```

produces the result below.

```
$"voltage=20":
[1] NA

$"voltage=26":
    0.1    0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9
 139.37 271.73 Inf Inf Inf Inf Inf Inf Inf

$"voltage=29":
   0.1   0.2   0.3    0.4   0.5    0.6 0.7 0.8 0.9
 45.85 55.73 91.81 108.62 164.2 257.88 Inf Inf Inf

$"voltage=32":
  0.1  0.2  0.3   0.4   0.5   0.6  0.7   0.8   0.9
 2.81 5.45 6.26 11.51 15.16 20.86 65.9 94.08 149.2
```

Notice that because no failures were observed beyond 300 days, survival drops to 0.0 in the final intervals for 26 and 29 volts, resulting in quantile estimates that are infinite. The true value is, of course, finite, but is not estimable from these data.

Plotting Kaplan-Meier Survival Curves

The plot method for the `kaplanMeier` function produces a plot of the estimated survival curves with optional confidence bands. For example, you can plot the fit `km.cap` from the previous section with the command:

```
> plot(km.cap)
```

To add confidence intervals to the curves, specify a logical vector to the `conf.int` argument as follows:

```
> plot(km.cap, conf.int = c(T, T, T))
```

Figure 30.1 displays the resulting plot.

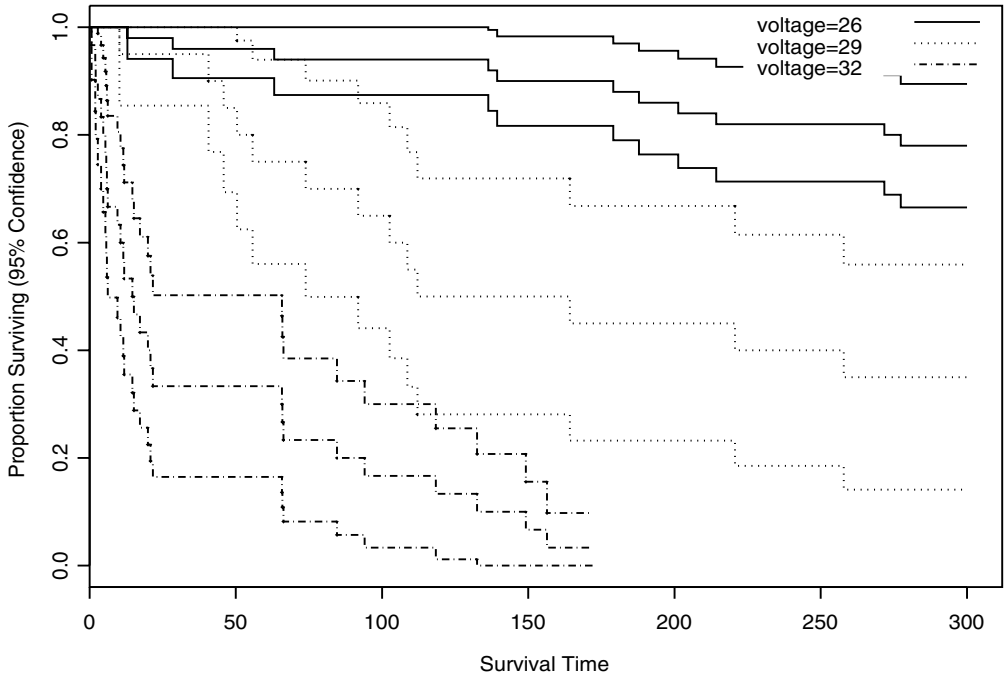

Figure 30.1: *Plot of* `km.cap`.

The `conf.int` argument allows you to specify confidence intervals for each curve independently, so you can turn some intervals on and leave others off. Confidence intervals are automatically added when only one survival curve is plotted, as for a nonstratified fit. When more than one curve is plotted with confidence intervals, the line type for the confidence interval automatically matches that of the survival curve.

Additional arguments to `plot.kaplanMeier` allow you to specify the color of the survival curves with color-matching confidence intervals. In addition, the line type and line width of the curves and confidence intervals can be specified along with *x*- and *y*-axis labels. In general, any argument to the generic `plot` function can be given to `plot.kaplanMeier`, including `xlim` for specifying *x*-axis limits and `main` for specifying a main title for the plot.

You can also use `plot.kaplanMeier` as a low-level graphics function for adding survival curves to an existing plot. This requires, of course, that the axis limits be set appropriately so that warning messages are not generated when the survival curves or their confidence intervals extend beyond the range of the plot region. An example uses the built-in data set `lung`. To do a stratified fit the `inst` column, plot two curves from the fit and then overlay a third plot using the following commands:

```
> kap.lung <- kaplanMeier(censor(time, status) ~ inst,
+ data = lung, na.action = na.exclude)

> plot.kaplanMeier(kap.lung$fits[c(1, 5)])
> plot.kaplanMeier(kap.lung$fits[9], conf.int = T, lty = 4,
+ add = T)
```

Note that the `plot.kaplanMeier` function is called explicitly here, because fits that are subscripted out of a `kaplanMeier` object lose their class designation. Also note that the above example is for pedagogy only. It could more easily be accomplished by doing the plots in a single call to `plot.kaplanMeier`, as follows:

```
> plot.kaplanMeier(kap.lung$fits[c(1, 5, 9)],
+ conf.int = c(F, F, T))
```

Figure 30.2 shows the result of this latter call.

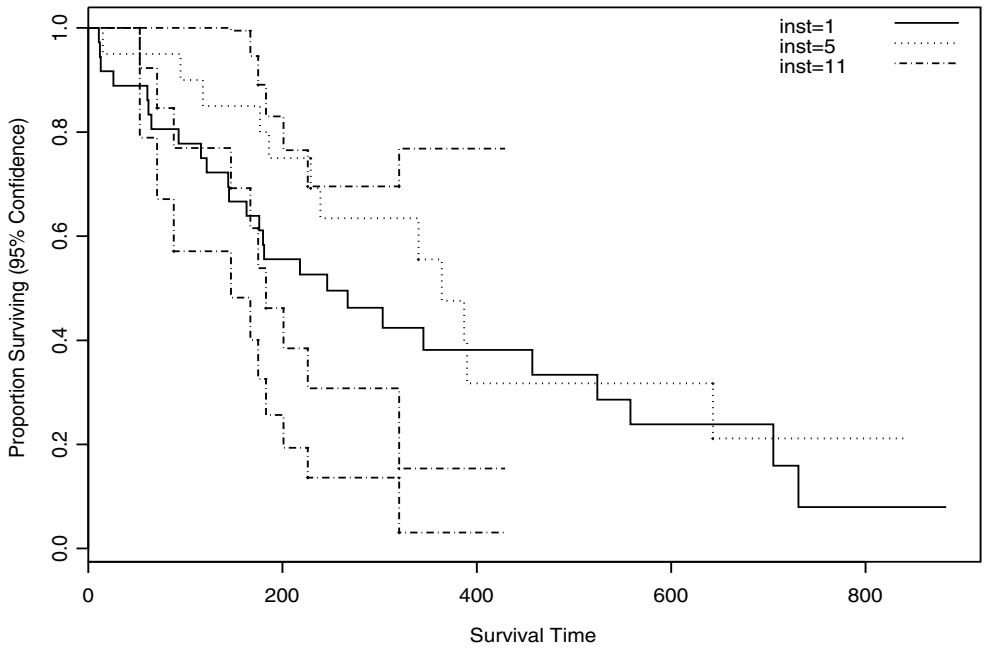

Figure 30.2: *Plot of* `kap.lung`.

PARAMETRIC SURVIVAL MODELS

Parametric (rather than nonparametric) estimates of failure distributions can be easily computed with the `censorReg` function. Like `kaplanMeier`, the `censorReg` function can handle interval-, right-, and left-censoring. In addition, `censorReg` handles three general families of failure distributions with logged and unlogged versions, truncated data, offsets, a threshold parameter, fixed coefficients, and much more.

An Example Model

As the simplest possible example, use the defaults for most arguments in a `censorReg` model with no covariates. For the `capacitor2` data set, a possible Spotfire S+ statement is:

```
> para.fit <- censorReg(censor(days,event) ~ 1,
+ weights = weights, data = capacitor2)

> para.fit

Call:
censorReg(formula = censor(days, event) ~ 1, data =
capacitor2, weights = weights)

Distribution: Weibull

Coefficients:
 (Intercept)
    6.704817

Dispersion (scale) = 1.821207
Log-likelihood: -372.7664

Observations: 125 Total; 71 Censored
Parameters Estimated: 2
```

As with the `kaplanMeier` function, the response is specified by the `censor` function. Because the model formula contains no covariates, a parametric model is fit for a single sample of observations. In this case, the parametric family defaults to the Weibull distribution. In the output, the location parameter for the Weibull distribution is estimated as 6.704 and the scale parameter is estimated as 1.82.

As with other Spotfire S+ fitting functions, `summary` can be used to obtain a more detailed summary of the fit. Following is the result of calling `summary` on the fit object:

```
> summary(para.fit)

Call:
censorReg(formula = censor(days, event) ~ 1, data =
capacitor2, weights = weights)

Distribution:  Weibull

Standardized Residuals:
             Min   Max
Uncensored 0.020 0.553
  Censored 0.577 0.577

Coefficients:
            Est. Std.Err. 95% LCL 95% UCL z-value   p-value
(Intercept)  6.7    0.296    6.12    7.29    22.6 3.01e-113
Extreme value distribution: Dispersion (scale) = 1.821207
Observations: 125 Total; 71 Censored
-2*Log-Likelihood: 746
```

Specifying the Parametric Family

The `censorReg` function supports 10 parametric distribution families. A particular family can be specified by setting the `distribution` argument in `censorReg` equal to the quoted string in the first column of Table 30.2.

The discussion following the table describes the internal specification of the parametric distribution families as they are viewed by the estimation routines. The general user need not be concerned with this aspect of the family specification, and can safely skip the rest of this section. The discussion is included here for the user who wants or needs access to the internal routines.

Table 30.2: *Distributions supported by* `censorReg`.

Argument	Distribution
`"weibull"`	Weibull
`"extreme"`	Smallest extreme value
`"lognormal"`	Log-normal or log-Gaussian
`"normal"`	Normal or Gaussian
`"loglogistic"`	Log-logistic
`"logistic"`	Logistic
`"logexponential"`	Log-exponential
`"exponential"`	Exponential
`"lograyleigh"`	Log-Rayleigh

Internally, the distributions are defined by two quantities, following the development of standard textbooks on parametric survival analysis: the distribution of the random variable, and the link function. Let $g(\bullet)$ denote the link function, and let

$$z = \frac{g(y) - x\beta}{\sigma}$$

be the random variable for failure time y. Here σ is the scale factor, x is a vector of covariates, and β is a vector of coefficients. In the simplest model, $x = 1$ for the intercept term. The term $x\beta$ specifies the location of the estimates. Two link functions $g(\bullet)$ are possible: the *identity* link

$$g(x) = x,$$

and the *log* link

$$g(x) = \log x .$$

Three distributions for z are available. These are the *logistic*

$$f(z) = \frac{\exp(-z)}{(1+\exp(-z))^2},$$

the *normal* or *Gaussian* distribution

$$f(z) = \frac{1}{\sqrt{2\pi}}\exp -\frac{1}{2}z^2,$$

and the *smallest extreme value* distribution

$$f(z) = \exp(z - \exp(z)) .$$

When the log link is used with a fixed value of $\sigma = 1$, the smallest extreme value distribution becomes an exponential distribution; if $\sigma = 0.5$, this becomes the Rayleigh distribution. When the smallest extreme value distribution is used with the log link, the distribution can be made equivalent to the two-parameter Weibull distribution:

$$f(z) = \frac{1}{\sigma\exp(x\beta)}\left(\frac{z}{\exp(x\beta)}\right)^{\frac{1}{\sigma}-1}\exp\left(-\left(\frac{x}{\exp(x\beta)}\right)^{1/\sigma}\right).$$

Here, $\theta = \frac{1}{\sigma}$ is the shape parameter.

Typically, failure times are positive since failure at a negative time is not usually meaningful. However, it is possible to give `censorReg` negative values for survival times when the identity link function is used. This might be useful with a Gaussian distribution, for example, which takes values over the entire real line.

To fit a Gaussian model to the `capacitor2` data, type:

```
> censorReg(censor(days, event) ~ 1, data = capacitor2,
+ distribution = "gaussian")
```

The hazard rate is the instantaneous rate of failure. This is computed as the first derivative of the failure density with respect to time. Different distributions result in different hazard rates, and thus in

different models. Much time in model building can be spent in deciding upon the correct model to use. The plotting functions discussed below can help in making this decision.

Accounting for Covariates

In the `censorReg` models above, we consider only a single sample of observations from the same distribution. Typically, a survival model also includes covariates to describe the distribution. For example, accelerated failure time models describe designed experiments in which a covariate is held fixed at a specified value for some observations, and the time to failure for the observations is recorded. The `capacitor2` data set is an example of such an experiment. Four values of the covariate `voltage` were observed: `voltage=20`, `voltage=26`, `voltage=29`, and `voltage=32`. Suppose we assume that the location parameter varies linearly with the `voltage` covariate:

$$z = \frac{g(y) - \alpha_0 - \alpha_1 x}{\sigma},$$

for intercept α_0. This model can be specified with the Spotfire S+ statement:

```
> censorReg(censor(days, event) ~ voltage,
+ weights = weights, data = capacitor2)

Call:
censorReg(formula = censor(days, event) ~ voltage, data =
capacitor2, weights = weights)

Distribution: Weibull

Coefficients:
 (Intercept)    voltage
    24.14083 -0.6403586

Dispersion (scale) = 1.203945
Log-likelihood: -316.4589

Observations: 125 Total; 71 Censored
Parameters Estimated: 3
```

In this model, the location parameter is obtained by regression on voltage. This requires a *linear* relationship of the hazard rate on voltage. Assuming that the relationship is not linear, a more general model fits

$$z = \frac{g(y) - \alpha_0 - \alpha_i}{\sigma}.$$

In this model, i indexes the different voltages, and the location parameter is allowed to vary in an arbitrary manner with voltage. This model is accomplished with the following command:

```
> censorReg(censor(days, event) ~ factor(voltage),
+ weights = weights, data = capacitor2)
```

Alternatively, suppose that the scale parameters are different for different values of the covariate. Then the model

$$z = \frac{g(y) - \alpha_0 - \alpha_i}{\sigma_i}$$

can be specified using the Spotfire S+ statement:

```
> censorReg(censor(days, event) ~ strata(voltage),
+ weights = weights, data = capacitor2)
```

In all but the last case, an object of class `"censorReg"` is produced. When the `strata` function is used to create a stratified fit (as in the last example), an object of class `"censorRegList"` is produced. This object contains a list of `censorReg` objects.

The `anova` function is used to compare the models described above. This is discussed in more detail in the section Comparing Parametric Survival Models.

Truncation Distributions

Aside from the distributions above, it is also possible to specify a different truncation distribution for each observation. In truncated data, the item being tested is not observed over the entire positive axis. Instead, observation of the item is made over a known interval that is a subset of the time period in which the observation could fail. Thus, if there is left truncation, the items under test may be

manufactured, used for a time, and then placed on test. Although the time to failure is scored as the time since manufacture, items that fail prior to being placed on test are not scored.

Let $t = 0$ be the time of manufacture, and suppose that testing begins at $t = \theta$. If $F(\theta)$ is the cumulative distribution of the failure time when observation starts at time zero, the distribution of the left-truncated failure times is given by

$$F(t|\theta) = \frac{F(t)}{1 - F(\theta)}.$$

Similarly, in right truncation, observation of failure or censoring is only made until $t = \theta$. Observations that either fail or are censored after time θ cannot be observed or are thrown out. Finally, in interval truncation, observation is made over a fixed interval (θ_1, θ_2), and observations that fail or are censored outside of the interval are not considered.

Truncation distributions can be fit easily with the `censorReg` function. For example, to obtain a Gaussian fit to the data in Table 30.3, use the following set of commands to first build the data set:

```
> unit <- c(1:9)
> failure <- c(7, 4, 5, 9, 4, 5, 7, 4, 11)
> upper <- c(rep(NA,5), 9, 12, NA, NA)
> Censor <- c("right", "exact", "exact", "right",
+ "left", "interval", "interval", "exact", "right")
> censor.codes <- c(0, 1, 1, 0, 2, 3, 3, 1, 0)
> tlower <- c(3, 0, 0, 3, 9, 3, 3, 0, 3)
> tupper <- c(rep(NA,5), 20, 20, NA, NA)
> trunc.codes <- c(2, 1, 1, 2, 0, 3, 3, 1, 2)
> table4 <- data.frame(unit, failure, upper,
+ Censor, censor.codes, tlower, tupper, trunc.codes)
```

Table 30.3: *Truncated data.*

Unit	Failure	Upper	Censor	Censor Codes	Lower Truncation	Upper Truncation	Truncation Codes
1	7	NA	right	0	3	NA	2
2	4	NA	exact	1	0	NA	1
3	5	NA	exact	1	0	NA	1
4	9	NA	right	0	3	NA	2
5	4	NA	left	2	9	NA	0
6	5	9	interval	3	3	20	3
7	7	12	interval	3	3	20	3
8	4	NA	exact	1	0	NA	1
9	11	NA	right	0	3	NA	2

The following call to `censorReg` computes the Gaussian fit:

```
> trunc.fit <- censorReg(
+ censor(failure, upper, censor.codes) ~ 1, data = table4,
+ truncation = censor(tlower, tupper, trunc.codes),
+ distribution = "lognormal")

> trunc.fit

Call:
censorReg(censor(failure, upper, cens) ~ 1, data = table4,
truncation = censor(tlower, tupper, trunc.codes),
distribution = "lognormal")
Distribution: Lognormal

Coefficients:
 (Intercept)
```

```
    1.920974

Dispersion (scale) = 0.9211897
Log-likelihood: -12.49965

Observations: 9 Total; 6 Censored
Parameters Estimated: 2
```

Because the log-likelihood is numerically complex when truncation distributions are used, it is important to verify convergence. Here, convergence is verified by the near-zero values of the first derivatives of the log-likelihood. We can extract the derivatives from the `trunc.fit` model as follows:

```
> trunc.fit$first.deriv

    (Intercept)          scale
-6.594777e-010 -4.993228e-009
```

Threshold Parameter

Truncation distributions modify the fitted distribution by considering failure in a subset of the positive real line. A distribution with a threshold parameter also modifies the failure distribution, but in a slightly different way. The idea of the threshold parameter is that test items cannot fail for a period of time after testing begins. Although testing begins at time zero, no tested item fails for some fixed period γ after time zero. Thus, the failure distribution is given by $F(t|\gamma) = F(t-\gamma)$. The net effect of the threshold parameter is to shift the failure distribution to the right by a fixed amount.

Maximum likelihood estimation of γ is not easily accomplished, though there is some discussion of this in Meeker and Escobar (1998, pp. 224-231). You can either compute the value of γ yourself and enter it as input to the `censorReg` function, or `censorReg` can estimate γ in two different ways. The first is to simply decrease the smallest failure time by 10%. The second works only for log distributions, and computes a value for γ which optimally linearizes a qqplot of the Kaplan-Meier survival estimate. By default, $\gamma = 0$. Once computed, γ is carried along with the `censorReg` object for further computations and information.

For the example in Table 30.3, we can set the threshold parameter equal to two as follows:

```
> censorReg(censor(failure, upper, censor.codes) ~ 1,
+ data = table4, truncation = censor(tlower, tupper,
+ trunc.codes), distribution = "lognormal", threshold = 2)

Call:
censorReg(formula = censor(failure, upper, censor.codes) ~
1, data = table4, truncation = censor(tlower, tupper,
trunc.codes), distribution = "lognormal", threshold = 2)

Distribution: Lognormal

Coefficients:
 (Intercept)
    1.664897

Dispersion (scale) = 1.38711
Log-likelihood: -12.23809

Observations: 9 Total; 6 Censored
Parameters Estimated: 2
Threshold Parameter: 2
```

Notice that the coefficient estimates have dramatically changed.

Offsets

Like threshold parameters, offsets are also used to change the distribution of the failure time variable. Let ω denote a known, fixed offset, and let y denote the failure time. When offsets are used, the transformed failure time becomes

$$z = \frac{g(y) - \omega - x\beta}{\sigma}.$$

A typical use of offsets is in likelihood ratio tests. Suppose that $x_1\hat{\beta}_1 + x_2\hat{\beta}_2$ optimizes the likelihood when covariates x_1 and x_2 are included in the model. Then a likelihood ratio test of $H_0: \hat{\beta}_1 = \kappa$ is

obtained by setting $\bar{\omega} = x_1\kappa$, and then comparing the optimized value of the likelihood of a model $\bar{\omega} + x_2\hat{\beta}_2$ with the optimized likelihood for model $x_1\hat{\beta}_1 + x_2\hat{\beta}_2$.

We illustrate this idea using the built-in `capacitor2` failure data. When the `voltage` covariate is included in the model, the output is:

```
> censorReg(censor(days,event) ~ voltage,
+ weights = weights, data = capacitor2)

Call:
censorReg(formula = censor(days, event) ~ voltage, data =
capacitor2, weights = weights)

Distribution: Weibull

Coefficients:
 (Intercept)     voltage
    24.14083 -0.6403586

Dispersion (scale) = 1.203945
Log-likelihood: -316.4589

Observations: 125 Total; 71 Censored
Parameters Estimated: 3
```

A likelihood ratio test that the voltage coefficient is fixed at -0.5 is obtained by fitting a second model that fixes the parameter estimate of `voltage`. This is accomplished with an `offset` term:

```
> censorReg(censor(days, event) ~ offset(-0.5 * voltage),
+ weights = weights, data = capacitor2)

Call:
censorReg(formula = censor(days, event) ~ offset(-0.5 *
voltage), data = capacitor2, weights = weights)

Distribution: Weibull

Coefficients:
 (Intercept)
    19.94567
```

```
Dispersion (scale) = 1.090527
Log-likelihood: -1129.826

Observations: 125 Total; 71 Censored
Parameters Estimated: 2
Offset has been specified
```

Computing the likelihood ratio test from the above two fits by hand we get:

$$\text{LRT} = -2*(-1129.8 + 316.5) = 1626.6$$

which is compared with a chi-squared distribution with one degree of freedom. Clearly, this is a significant result.

Fixing Parameters

It is also possible to simply fix parameters in the model. Most often the scale parameter is fixed, but it is possible to fix any parameter. For example, in the `capacitor2` example we may fix the voltage coefficient to be -0.5 using the command:

```
> censorReg(censor(days, event) ~ voltage, data =
+ capacitor2, weights = weights, fixed = list(
+ voltage = -0.5))

Distribution: Weibull

Coefficients:
 (Intercept)
    19.94567

Dispersion (scale) = 1.090527
Log-likelihood: -1129.826

Observations: 125 Total; 71 Censored
Parameters Estimated: 2
```

Comparing this with the results in which `offset` is used, we see that the effect of fixing voltage to be -0.5 is the same as specifying the offset to be `-0.5*voltage`.

COMPARING PARAMETRIC SURVIVAL MODELS

The anova function is used to compare models. If a single object is input to anova, then one term at a time is added to the model. The anova comparisons start from the smallest possible model (usually the intercept-only model) and continue until the model object is obtained. As an example, consider the following model:

```
> fit <- censorReg(censor(days, event) ~ voltage +
+ voltage^2, weights = weights, data = capacitor2)
```

Apply the anova function to the fit as follows:

```
> anova(fit, test = "Chisq")

Likelihood Ratio Test Table

Weibull model

Response: censor(days, event)

Terms added sequentially (first to last)
             N.Params -2*LogLik Df      LRT   Pr(Chi)
        NULL        2  745.5327
     voltage        3  632.9178  1 112.6149 0.0000000
I(voltage^2)        4  632.8494  1   0.0684 0.7937407
```

The results suggest that the location parameter of the distribution depends on voltage linearly, and that the quadratic term is unimportant. We verify this hypothesis below.

When two or more censorReg or censorRegList objects are input to the anova function, the models are compared with likelihood ratio tests. Suppose we are interested in testing whether the model for the capacitor2 data should be

$$z = \frac{g(y) - x\beta}{\sigma},$$

where x is voltage.

More general models (in the sense of having more parameters) are

$$z = \frac{g(y) - \alpha_i}{\sigma},$$

and

$$z = \frac{g(y) - \alpha_i}{\sigma_i},$$

where i indexes the different voltages. These three models plus an intercept-only model can be generated in Spotfire S+ using the following statements:

```
> fit0 <- censorReg(censor(days,event) ~ 1,
+ weights = weights, data = capacitor2)

> fit1 <- censorReg(censor(days,event) ~ voltage,
+ weights = weights, data = capacitor2)

> fit2 <- censorReg(censor(days, event) ~ factor(voltage),
+ weights = weights, data = capacitor2)

> fit3 <- censorReg(censor(days, event) ~ strata(voltage),
+ weights = weights, data = capacitor2)
```

The models are then compared using the `anova` function as follows:

```
> anova(fit0, fit1, fit2, fit3, test = "Chisq")

Likelihood Ratio Test(s)

Response: censor(days, event)

      Terms N.Params -2*LogLik      Test Df     LRT  Pr(Chi)
1         1       2   745.53
2   voltage       3   632.92  + voltage  1 112.615  0.0000
3 factor(voltage) 5   632.37    2 vs. 3  2   0.547  0.7605
4 strata(voltage) 6   630.40    3 vs. 4  1   1.973  0.1601
```

The evidence is now quite strong that we can't do any better than the model that linearly relates the location parameter of the distribution and voltage. We can verify this by looking at graphics.

PLOTS FOR PARAMETRIC SURVIVAL MODELS

The plot method for objects of class "censorReg" generates four to six plots depending on the type of fit. You can generate all possible plots for a censorReg fit object by simply using the plot function as follows:

```
> plot(fit1)
```

The first three graphs that result from plot are equivalent to those produced for fits of class "lm" or "glm", so they are not discussed further here. The last four graphs are presented in Figure 30.3 through Figure 30.6.

Figure 30.3 displays a probability plot of the standardized residuals. Standardized residuals are described in Meeker and Escobar (1998), where they are referred to as "censored Cox-Snell" residuals. For diagnostic purposes, a maximum likelihood estimate of a null model (intercept only) is displayed in Figure 30.3, along with the residuals.

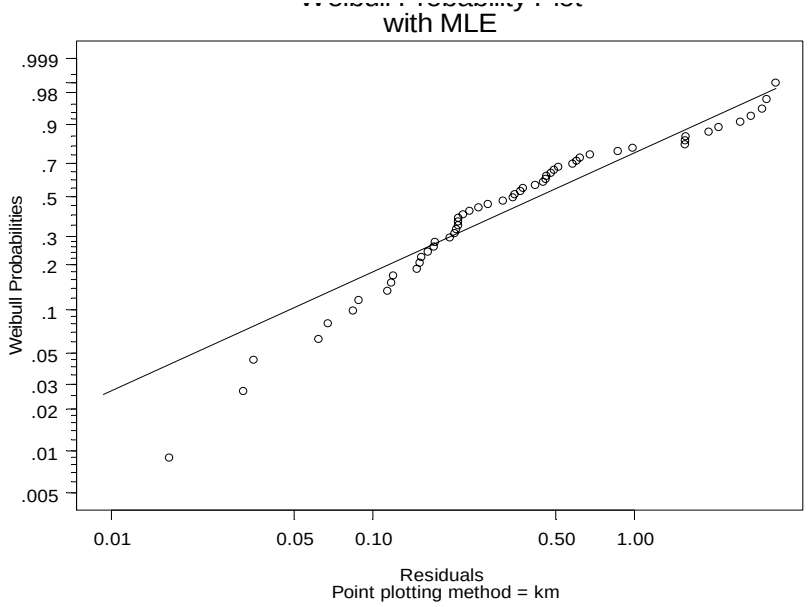

Figure 30.3: *Probability plot of standardized residuals with maximum likelihood estimate.*

Figure 30.4 displays a probability plot of the fitted model along with the noncensored observations. Each line corresponds to the fit for a different value of the covariate, and each set of points corresponds to the noncensored observations. Although the `censorReg` function is not constrained to single covariates, probability plots are currently available for single covariate models only. For more details, see the help file for `probplot.censorReg`, which is the function that produces these kinds of plots.

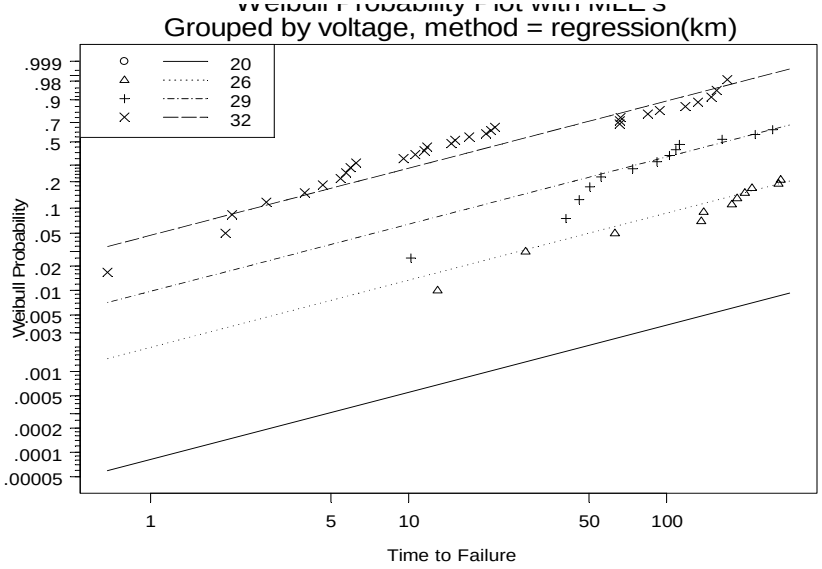

Figure 30.4: *Probability plot of the fit with maximum likelihood estimates.*

Figure 30.5 displays what engineers refer to as a *stress* plot. It plots the noncensored observations and equi-probability lines for the predictor variable (the *stressor*) versus failure times. It is quite clear from the graph that as voltage (or stress) decreases, failure times increase. Like probability plots, stress plots are constrained to single covariate regression models. For more details, see the help file for `stressplot.censorReg`.

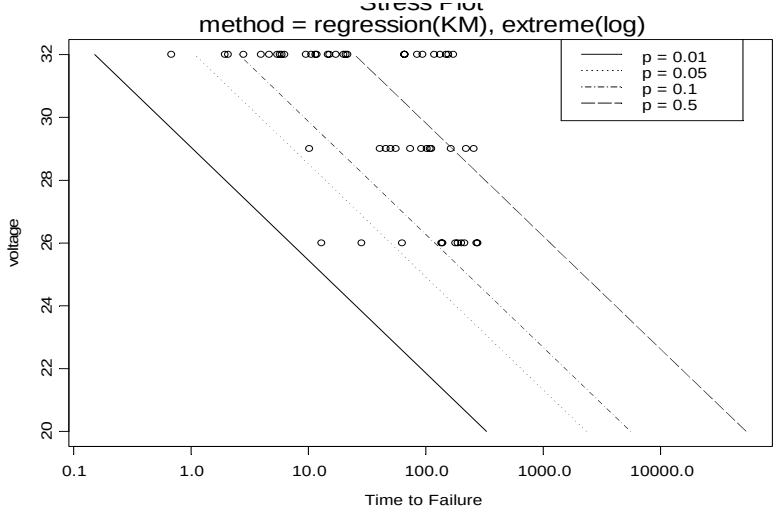

Figure 30.5: *Stress plot of the fit.*

The final diagnostic plot for a `censorReg` object is displayed in Figure 30.6. This is the same plot as Figure 30.4, but repeated for six distributions. The distributions are the Weibull, the lognormal, and the loglogistic, coupled with their nonlogged counterparts. This plot is provided primarily for distribution assessment. It's quite clear from Figure 30.6 that a nonlogged distribution does not fit the data well. Exactly which logged distribution fits the data best is not so clear. For more information on this kind of plot, see the help file for `probplot6.censorReg`.

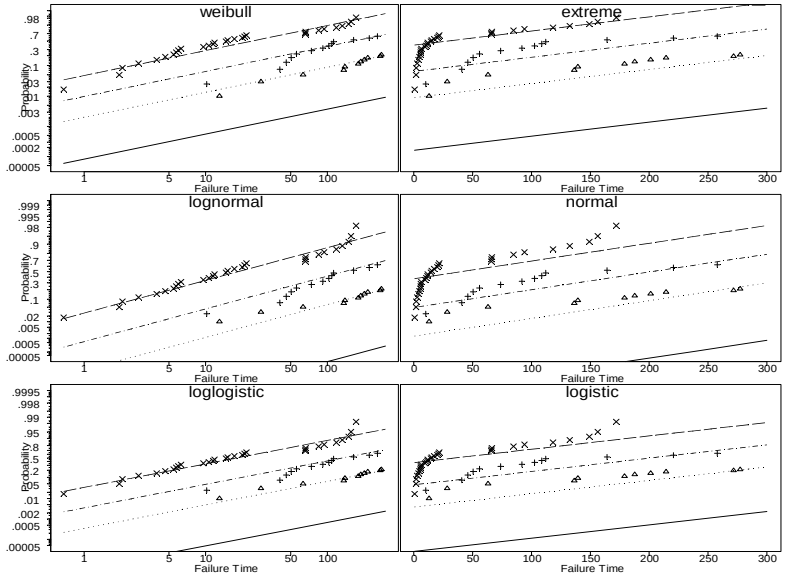

Figure 30.6: *Six-distribution plot of the fit.*

Although the three plotting functions `probplot.censorReg`, `stressplot.censorReg`, and `probplot6.censorReg` are called by the `plot` method for a `censorReg` object, they were designed to be called directly. This provides more capabilities than those available through the general `plot` command alone. For example, the `method` argument in each of the plotting functions allows the plotted points to be computed based upon some alternative model. This argument defaults to the `"KM"`, or Kaplan-Meier estimates, but four other methods are possible. They are:

1. The `"one"` or null (intercept only) model, where

$$z = \frac{g(y) - \mu}{\sigma},$$

for location parameter μ.

2. The `"regression"` model, in which

$$z = \frac{g(y) - x\beta}{\sigma},$$

for covariate x.

3. The "factor" model, which uses

$$z = \frac{g(y) - \alpha_i}{\sigma}$$

to compute separate locations for each value of the covariate.

4. The "separate" model,

$$z = \frac{g(y) - \alpha_i}{\sigma_i}.$$

This is the most general single-variable parametric model, and allows separate location and scale parameter estimates for each value of the covariate.

To compare the regression fit stored in fit1 with the more general "separate" fit, use the statement:

```
> probplot(fit1, method = "separate", add.legend = T,
+ legend.loc = "auto")
```

This results in the plot shown in Figure 30.7. The plotted points in Figure 30.7 are obtained from the "separate" model and show some deviation from the "regression" model. However, the deviation is not statistically significant, as we saw when we compared the models with a likelihood ratio test.

It is also possible to add confidence intervals for each maximum likelihood estimate to get a feel for the variability of the estimated distribution(s). To do this, set the argument mle.interval=T in the probplot.censorReg and probplot6.censorReg functions.

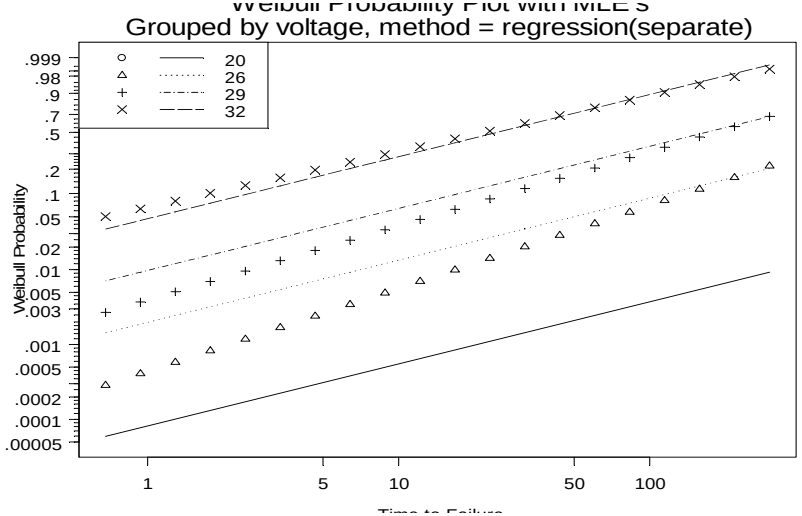

Figure 30.7: *Probability plot for comparing models.*

COMPUTING PROBABILITIES AND QUANTILES

The `predict` method for `censorReg` objects computes predictions for specified covariate values in a fitted model on either probability or response scales, at designated quantiles or probabilities, respectively. For example, suppose you want to estimate the time to 10%, 50%, and 90% failure from our `fit1` regression model for voltages of 16, 20, and 24. The call to the `predict` function is:

```
> predict(fit1, newdata = data.frame(voltage =
+ c(16, 20, 24)))

$"voltage=16":
      Estimate  Std.Err   95% LCL    95% UCL
0.1   72097.22 1.028782   9598.82   541525.8
0.5  696503.03 1.133190  75570.05  6419427.9
0.9 2955616.38 1.217862 271644.96 32158403.5

$"voltage=20":
      Estimate   Std.Err   95% LCL    95% UCL
0.1   5565.468 0.7211182  1354.206   22872.76
0.5  53765.809 0.8136006 10913.602  264877.00
0.9 228155.656 0.8986737 39199.343 1327956.02

$"voltage=24":
      Estimate   Std.Err   95% LCL   95% UCL
0.1   429.6203 0.4384364  181.9228  1014.571
0.5  4150.3943 0.5003670 1556.5971 11066.302
0.9 17612.2327 0.5853507 5591.9462 55470.980
```

Operating the capacitor at 16 volts increases its life span by about 170 times compared to operating at 24 volts. The probability values (proportion failed) are 0.1, 0.5, and 0.9 by default when calling the `predict` function. These values can be modified with the `p` argument in `predict`. For example, to compute the 10%, 20%, and 30% failure times, enter:

```
> predict(fit1, p = c(0.1, 0.2, 0.3),
+ newdata = data.frame(voltage = c(16, 20, 24)))
```

Alternatively, to predict the failure rates for given quantiles of the failure time distribution (i.e., the proportion failed), specify the `type="probability"` argument to `predict`. For example, to compute the failure rates for 16, 20 and 24 volts at 1000, 2000, and 3000 days, type:

```
> predict(fit1, q = c(1000, 2000, 3000),
+ type = "probability",
+ newdata = data.frame(voltage = c(16, 20, 24)))

$"voltage=16":
        Estimate   Std.Err      95% LCL    95% UCL
1000 0.003011831 0.7981976 0.0006315958 0.01423447
2000 0.005350046 0.7977207 0.0011250641 0.02504342
3000 0.007484339 0.7997419 0.0015703341 0.03489259

$"voltage=20":
       Estimate   Std.Err     95% LCL    95% UCL
1000 0.02500204 0.5845648 0.008088394 0.07462304
2000 0.04403085 0.5949867 0.014147239 0.12879193
3000 0.06111373 0.6058481 0.019466567 0.17587955

$"voltage=24":
      Estimate   Std.Err    95% LCL   95% UCL
1000 0.1914712 0.4138868 0.09520448 0.3476748
2000 0.3147595 0.4679518 0.15510243 0.5347458
3000 0.4110080 0.5191918 0.20142736 0.6587655
```

The difference is again dramatic when comparing 16 and 24 volts. After 1000 days, you expect only about 3 of 1000 capacitors to fail when operated at 16 volts, compared to 19 out of 1000 that are operated at 24 volts.

Additional arguments to `predict` allow you to specify the level of the confidence intervals, and whether you want to print the standard errors and confidence intervals. For more details, see the descriptions of the `coverage`, `se.fit`, and `conf.interval` arguments in the `predict.censorReg` help file.

REFERENCES

Andersen, P.K., Borgan, O., Gill, R.D., & Keiding N. (1993). *Statistical Models Based on Counting Processes*. New York: Springer-Verlag.

Meeker, W.Q. & Duke S.D. (1981). CENSOR: A user-oriented computer program for life data analysis. *The American Statistician*, 35(2): 112.

Meeker, W.Q. & Escobar, (1998). *Statistical Methods for Reliability Data*. New York: John Wiley & Sons, Inc.

Nelson (1990). *Accelerated Life Testing*. New York: John Wiley & Sons, Inc.

Turnbull, B. (1974). Nonparametric estimation of a survivorship function with doubly censored data. *Journal of the American Statistical Association* **69**:169-173.

Turnbull, B. (1976). The empirical distribution function with arbitrarily grouped, censored, and truncated data. *Journal of the Royal Statistical Society* (Series B) **37**:290-295.

EXPECTED SURVIVAL 31

INTRODUCTION

This chapter describes several methods for estimating expected survival curves. Typically expected curves are used for comparison with another study. Sometimes the results of an earlier study are compared with a later one to assess, for example, improvement in treatment. Expected survival curves can be computed from tables of hazards rates or from a previously computed Cox model.

The methodology described in this chapter includes the computation of *individual* and *cohort* expected survival curves. Individual expected curves are typically used to compute tests that compare the observed survival with that expected (for example, the one-sample log-rank test) for a matched (for example, on age, sex, and year of entry) control population. Cohort expected curves are useful for graphical comparisons, sample size computations, and forecasting.

Three methods are available for computing cohort expected survival curves: the Ederer or *exact* method, Hakulinen's method, and the conditional estimate. In the Cox model literature, these have been called the *direct-adjusted*, *Bonsel*, and *expected survival* curves. Each method generates a matched control for each subject in the study and then computes the expected survival for the matched controls. The difference between the methods lies in the assumptions made when computing the expected survival. The basic assumptions of each and a brief description of its utility follows.

- *Ederer*: Assumes complete follow-up (that is, no censoring). Each control is followed until death. This is most appropriate when doing forecasting, sample size calculations, or other predictions of the "future" where censoring is not an issue.
- *Hakulinen*: Assumes maximal potential follow-up. Each control is followed until death or censoring of its matched case. This is useful for graphical comparison with the study population.
- *Conditional*: Has the same assumptions and is asymptotically equivalent to Hakulinen's method.

The implementation of expected survival curve estimation allows you to add your own table of hazard rates or compute expected survival based on a previous Cox model. Additionally, the notion of *person years* of follow-up time is discussed as an example.

INDIVIDUAL EXPECTED SURVIVAL

Let $\lambda_i(t)$ and $\Lambda_i(t)$ be the derived hazard and cumulative hazard functions, respectively, for subject i, starting at their time of entry to the study. Then $S_i(t) = \exp(-\Lambda_i(t))$ is the subject's expected survival function. Some authors use the product form $S = 1 - \Pi(1 - q_k)$, where the q are yearly probabilities of death; yet others use an equation similar to actuarial survival estimates. Numerically, it makes little difference which form is chosen. The Spotfire S+ functions use the hazard based formulation for its convenience.

The survival tables published by the Department of the Census contain one year survival probabilities by age and sex, optionally subgrouped by race and geographic region. The entry for age 21 in 1950 is the probability that a subject who turns 21 during 1950 will live to his or her 22nd birthday. For convenience, the tables stored in Spotfire S+ contain the daily hazard rate λ rather than the probability of survival p

$$p = \exp(-365.25 \times \lambda).$$

If a, s, and y are subscripts into the age by sex by calendar year table of rates, then the cumulative hazard for a given subject is simply the sequential sum of $\lambda_{asy} \times$ number of days in state(a, s, y). That is, the patient progresses through the rate table on a diagonal line whose starting point is (date of entry, age at entry, sex). See Berry (1983) for a nice graphical illustration.

COHORT EXPECTED SURVIVAL

The expected survival curve for a cohort of n subjects is an average of the n individual survival curves for the subjects. There are 3 main methods for combining these; for some data sets they can give substantially different results. Let S_e be the expected survival for the cohort as a whole, and let S_i and λ_i be the individual survival and hazard functions, respectively. All three methods can be written as

$$S_e(t) = \exp\left(-\int_0^t \frac{\sum \lambda_i(s) w_i(s)}{\sum w_i(s)} ds\right). \tag{31.1}$$

The methods differ only in the weight function w_i.

The cohort curve should be distinguished from the individual curve for an average subject. For example, assume we had a cohort of grandfathers and their grandsons, the grandfathers average 70 years of age and the grandsons average 10 years. The cohort curve, which is an estimate of the curve we would expect from long term follow-up of these subjects, is considerably different than the curve for the average subject with mean age of 40 years.

The Exact Method

A weight function of $w_i(t) = S_i(t)$ corresponds to the *exact* method. This is the oldest and most commonly used technique, and is described in Ederer, Axtel and Cutler (1961). An equivalent expression for the estimate is

$$S_e(t) = (1/n)\sum S_i(t). \tag{31.2}$$

The exact method corresponds to selecting a population-matched control for each subject in the study, and then computing the expected survival of this cohort assuming complete follow-up. The

exact method is most appropriate when doing forecasting, sample size calculations, or other predictions of the future where censoring is not an issue.

A common use of the expected survival curve is to plot it along with the Kaplan-Meier estimate of the sample in order to assess the relative survival of the study group. When used in this way, several authors have shown that the exact method can be misleading if censoring is not independent of age and sex (or whatever the matching factors are for the referent population). Indeed, independence is often not the case. For example, in a long study it is not uncommon to allow older patients to enroll only after the initial phase. A severe example of this is demonstrated in Verheul, *et al.* (1993), concerning aortic valve replacement over a 20 year period. The proportion of patients over 70 years of age was 1% in the first ten years, and 27% in the second ten years. Assume that analysis of the data took place immediately at the end of the study period. The Kaplan-Meier curve for the later years of follow-up time will be too flat, since it is computed only over the early enrollees, who are younger on the average. The Ederer or exact curve does not reflect this bias, and makes the treatment look better than it is. The exact expected survival curve forms a reference line, in reality, for what the Kaplan-Meier will be when follow-up is complete, rather than for what the Kaplan-Meier is now.

Hakulinen's Method

In Hakulinen's method (Hakulinen (1982), Hakulinen and Abeywickrama (1985)), each study subject is again paired with a fictional referent from the cohort population, but the referents are now treated as though they were followed in the same way as the study patients. The referents thus have a maximum potential follow-up; that is, they will become censored at the analysis date. Let $c_i(t)$ be a censoring indicator which is 1 during the period of potential follow-up and 0 thereafter. The weight function for the Hakulinen or *cohort* method is $w_i(t) = S_i(t)c_i(t)$.

If the study subject is censored, the referent is presumably censored at the same time. However, if the study subject dies, the censoring time for the matched referent is the time at which the study subject would have been censored. For observational studies or clinical trials where censoring is induced by the analysis date, this should be straightforward, but determination of the potential follow-up could be a problem if there are large numbers lost to follow-up. As pointed out

long ago by Berkeson (1950), if a large number of subjects are lost to follow-up then any conclusion is subject to doubt. Did patients stop responding to follow-up letters at random because they were cured or because they were at death's door?

In practice, the program is invoked using the actual follow-up time for those patients who are censored, and the *maximum potential follow-up* for those who have died. By the maximum potential follow-up, we mean the difference between enrollment date and the average last contact date. For example, if patients are contacted every 3 months on average and the study was closed six months ago, the maximum potential follow-up date would be 7.5 months ago. It may be true that the (hypothetical) matched control for a case who died 30 years ago has little actual chance of such long follow-up, but this is not really important. Almost all of the numerical differences between the Ederer and Hakulinen estimates result from censoring those patients who most recently entered the study. For these recent patients, presumably enough is known about the operation of the study to give a rational estimate of potential follow-up.

The Hakulinen formula can be expressed in a product form

$$S_e(t+s) = S_e(t) \times \frac{\sum p_i(t,\ s) S_i(t) c_i(t)}{\sum S_i(t) c_i(t)} \tag{31.3}$$

where $p_i(t,\ s)$ is the conditional probability $\exp(\Lambda_i(t) - \Lambda_i(t+s))$ of surviving from time t to time $t+s$. The formula is technically correct only over time intervals $(t,\ t+s)$ for which c_i is constant for all i (censoring only at the ends of the interval).

The Conditional Method

The conditional estimate is advocated by Verheul, *et al*. (1993), and was also suggested as a computation simplification of the exact method by Ederer and Heise (1977). For this estimate, the weight function $w_i(t)$ is defined to be 1 while the subject is alive and at risk, and 0 otherwise. It is clearly related to Hakulinen's method, since $E(w_i(t)) = S_i(t)c_i(t)$. Most authors present the estimator in the product-limit form $\Pi[1 - d(t)/\ n(t)]$, where d and n are the

numerator and denominator terms within the integral of Equation (31.1). One disadvantage of the product-limit form is that the value of the estimate at time t depends on the number of intervals into which the time axis has been divided. For this reason we use the integral form (Equation (31.1)) directly.

One advantage of the conditional estimate, shared with Hakulinen's method, is that it remains consistent when the censoring pattern differs between age-sex strata. A problem with the conditional estimator is that it has a much larger variance than either the exact or Hakulinen estimate. In fact, the variance of these latter two can usually be assumed to be zero, at least in comparison to the variance of the Kaplan-Meier of the sample. Rate tables are normally based on a very large sample size, so the individual λ_i are very precise and the censoring indicators c_i are based on the study design rather than on patient outcomes. The conditional estimate $S_c(t)$, however, depends on the actual death times and w_i is a random variable.

The main use of the conditional estimate is when making conditional statements about survival. For example, in studies of surgical intervention such as hip replacement, the observed and expected survival curves often diverge initially due to surgical mortality, and then appear to become parallel. It is tempting to say that survival beyond hospital discharge is equivalent to expected. This is a conditional probability statement, and it should not be made unless a conditional estimate is used.

A hypothetical example may make this clearer. For simplicity, assume no censoring. Suppose we have studies of two diseases which have identical age distributions at entry. Disease A kills 10% of the subjects in the first month, independent of age or sex, and thereafter has no effect. Disease B also kills 10% of its subjects in the first month, but predominately affects the old. After the first month it exerts a continuing though much smaller force of mortality, still biased toward the older ages. With proper choice of the age effect, studies A and B will have almost identical survival curves, as the patients in B are always younger, on average, than those in A.

In our hypothetical example, two different questions can be asked under the guise of expected survival:

1. What is the overall effect of the disease? In this sense both A and B have the same effect, in that the 5 year survival probability for a diseased group is *x*% below that of a matched population cohort. The Hakulinen estimate is preferred because of its lower variance. It estimates the curve we would have gotten if the study had included a control group.
2. What is the ongoing effect of the disease? Detection of the differential effects of A and B after the first month requires the conditional estimator. We can look at the slopes of the curves to judge if they have become parallel.

The actual curve generated by the conditional estimator remains difficult to interpret, however. The difficulty lies in the fact that the control subject is removed from the calculation whenever the matching case dies. In general, Hakulinen's cohort estimate is probably best. If there is a question about delayed effects, as in the above example, there would be an apparent flattening of the Kaplan-Meier curves after the first month. Then one can plot a new curve using only those patients who survived at least one month.

APPROXIMATIONS

The Hakulinen cohort estimate (Equation (31.3)) is "Kaplan-Meier like," in that it is a product of conditional probabilities and the time axis is partitioned according to the observed death and censoring times. Both the exact and conditional estimators can be written in this way as well. They are unlike a Kaplan-Meier calculation, however, in that the ingredients of each conditional estimate are the n distinct individual survival probabilities at that time point, rather than just a count of the number at risk.

For a large data set, this requirement for $O(n)$ temporary variables can be a problem. An approximation is to use longer intervals and allow subjects to contribute partial information to each interval. For instance, in Equation (31.3) replace the 0/1 weight $c_i(t)$ by $\int_t^{t+s} c_i(u)du/$, which is the proportion of time that subject i was uncensored during the interval $(t,\ t+s)$. If those with fractional weights form a minority of those at risk during the interval, the approximation should be reliable. More formally, if the sum of their weights is a minority of the total sum of weights, the approximation is reliable. By Jensen's inequality, the approximation is always biased upwards, but it is very small. For the Stanford heart transplant data used in the examples, an exact 5 year estimate using the cohort method is 0.94728, an approximate cohort computation using only the half year intervals yields 0.94841. The exact estimate is unchanged under repartitioning of the time axis.

TESTING

All of the above discussion has been geared towards a plot of $S_e(t) = \exp(-\Lambda_e(t))$ which attempts to capture the proportion of patients who will have died by t. When comparing observed to expected survival, an appropriate test is the one-sample log-rank test (Harrington and Fleming (1982)). This is defined as $(O-E)^2/E$, where O is the observed number of deaths and

$$\begin{aligned} E &= \sum_{i=1}^{n} e_i \\ &= \sum_{i=1}^{n} \int \lambda_i(s) Y_i(s) \end{aligned} \tag{31.4}$$

Here E is the expected number of deaths, given the observation time of each subject. This follows Mantel's concept of *exposure to death* (Mantel (1966)), and is the expected number of deaths during this exposure. Notice how this differs from the expected number of deaths $nS_e(t)$ in the matched cohort at time t. In particular, E can be greater than n. Equation (31.4) is referred to as the *person-years estimate* of the expected number of deaths. The log-rank test is usually more powerful than one based on comparing the observed survival at time t to $S_e(t)$; the former is a comparison of the entire observed curve to the expected, and the latter is a test for difference at one point in time.

Tests at a particular time point, though less powerful, are appropriate if some fixed time is of particular interest, such as 5 year survival. In this case, the test should be based on the cohort estimate. The H_0 of the test is, "Is survival different than what a control-group's survival would have been?" A pointwise test based on the exact estimate may well be invalid if there is censoring. A pointwise test based on the conditional estimate has two problems: the first is that an appropriate variance is difficult to construct and the second, more serious one, is that it is unclear exactly what alternative is being tested against.

Hartz, Giefer, and Hoffman (1983) argue strongly for the pointwise tests based on an expected survival estimate equivalent to Equation (31.3), and claim that such a test is both more powerful and more logical than the person-years approach. Subsequent letters to the editor (Hartz, Giefer, and Hoffmann (1984, 1985)) challenged these views, and it appears that the person-years method is preferred.

Berry (1983) provides an excellent overview of the person-years method. Let the e_i be the expected number of events for each subject, treating them as an $n = 1$ Poisson process. We have

$$\begin{aligned} e_i &= \int_0^\infty Y_i(s)\lambda_i(s)ds \\ &= \Lambda_i(t_i) \end{aligned}$$

where t_i is the observed survival or censoring time for a subject. This quantity e_i is the total amount of hazard that would have been experienced by the population-matched referent subject, over the time interval that subject i was actually under observation. If we treat e_i as though it were the follow-up time, this corrects for the background mortality by mapping each subject onto a time scale where the baseline hazard is 1.

Tests can now be based on a Poisson model, using δ_i as the response variable (1 = dead, 0 = censored) and e_i as the time of observation (an offset of log e). The intercept term of the model estimates the overall difference in hazard between the study subjects and the expected population. An intercept-only model is equivalent to the one sample log-rank test. Covariates in the model estimate the effect of a predictor on excess mortality, whereas an ordinary Poisson or Cox model estimates its effect on total mortality.

Andersen and Væth (1989) consider both multiplicative and additive models for excess risk. Let λ_i^* be the actual hazard function for the individual at risk and let λ_i be that for the matched control from the population. The multiplicative hazard model is

$$\lambda_i^*(t) = \beta(t)\lambda_i(t).$$

If $\beta(t)$ is constant, then

$$\hat{\beta}_0 \equiv \frac{\sum N_i}{\sum e_i}$$

is an estimate of the *standard mortality ratio* or SMR, which is identical to `exp(`*`intercept`*`)` in the Poisson model used by Berry (assuming a log link). Their estimate over time is based on a modified Nelson hazard estimate

$$\hat{B}'(t) = \int_0^t \frac{\sum dN_i(s)}{\sum Y_i(s)\lambda_i(s)} ds,$$

which estimates the integral of $\beta(t)$. If the SMR is constant, a plot of $\hat{B}'(t)$ versus t should be a straight line through the origin.

For the additive hazard model

$$\lambda_i^*(t) = \alpha(t) + \lambda_i(t),$$

the integral $A(t)$ of α is estimated as

$$\log[S_{KM}(t) / S_c(t)].$$

This is the difference between the Kaplan-Meier and the conditional estimator when plotted on log scale. Under the hypothesis of a constant additive risk, a plot of $\hat{A}(t)$ versus t should approximate a line through the origin.

COMPUTING EXPECTED SURVIVAL CURVES

The Spotfire S+ function that computes expected survival curves is `survexp`. Besides taking the typical arguments of a model fitting function, `survexp` also takes the arguments listed below.

- `times`: Vector of follow-up times at which the resulting survival curve is evaluated. If absent, the result is reported for each unique value of the vector of follow-up times supplied in the formula.
- `cohort`: Logical value. If `cohort=FALSE`, each subject is treated as a subgroup of size 1. The default value is `TRUE`.
- `conditional`: Logical value. If `conditional=TRUE`, the follow-up times supplied in the formula are death times and conditional expected survival is computed. If `conditional=FALSE`, the follow-up times are potential censoring times. If follow-up times are missing in the formula, this argument is ignored.
- `ratetable`: Table of event rates, such as `survexp.uswhite` or a fitted Cox model.

Table 31.1 summarizes the argument settings used to compute expected survival curves by the various methods. The real-life examples of the following section show the use of the various argument settings to obtain the different estimates of expected survival.

Table 31.1: *Summary of arguments settings for invoking the various methods of estimating expected survival.*

Method	Conditional Argument	Cohort Argument	Follow-up Times
Individual survival	Not used	`cohort=F`	Yes
Cohort survival:			
`Ederer`	`conditional=F`	`cohort=T`	No
`Hakulinen`	`conditional=F`	`cohort=T`	Yes
`Conditional`	`conditional=T`	`cohort=T`	Yes

EXAMPLES

The examples in this section show how the methods discussed earlier in this chapter are implemented in Spotfire S+. In addition to computing various expected survival curves, an example of a closely related topic, person years of follow-up, is provided. The person-years example uses a function called `pyears`, and the expected survival examples use the `survexp` function.

All of the examples use a data frame, `hearta`, computed from `heart` as follows:

```
> hearta <- by(heart, heart$id,
+ function(x) x[x$stop == max(x$stop), ])
> hearta <- do.call("rbind", hearta)
```

Because the transplanted patients are represented by two rows in the `heart` data frame, you first need to extract only those rows that correspond to death or censoring. Do this by selecting all rows for which `stop` is a maximum for each patient, and then use `rbind` to put them back together into the data frame called `hearta`. Once this is done, `stop` contains only the total follow-up times for each patient. Note that this depends on each patient having a start time of zero.

Computing Expected Survival from National Hazard Rate Tables

The computation of expected survival curves requires either a table of hazard rates or a fitted Cox model to act as a hazard rate table. Several rate tables are built into Spotfire S+, including tables for the U.S. population, Minnesota, Florida, and Arizona. The U.S. and state rate tables contain the expected hazard rate for a subject, stratified by age, sex, calendar year, and optionally by race.

You can add new rate tables for other areas if you wish. Created rate tables have no restrictions on the number or names of the stratification variables. See the help file for `survexp.us` for details.

Warning

When using a rate table, it is important that all time variables be in the same units as were used for the table. For the U.S. tables, this is hazard/day, so time must be in days. All time variables must also have the same start date. Year is an exception; see the examples below.

The following example computes the conditional expected survival curves for the two surgery groups in the heart transplant study. No `ratetable` argument is supplied, so the default table `survexp.us` is used.

```
> expsurv <- survexp(stop ~ surgery + ratetable(
+ age = (age + 48) * 365.25, sex = "male",
+ year = year + 1967.75), data = hearta, conditional = T)
```

In addition to follow-up times, the formula contains `stop`, a grouping variable, `surgery`, which causes the output to contain two curves, and a special function, `ratetable`. The `ratetable` function matches the data frame's variables to the corresponding dimensions of the rate table. The order of the arguments to the `ratetable` function is not important. The necessary key words `age`, `sex`, and `year` are contained in the `"dimid"` attribute of the rate table providing the hazard rates. The `hearta` data frame does not contain a sex variable so `sex` is set conservatively to `"male"`. Setting values such as this must be done by providing an integer subscript or a match to one of the `"dimnames"`.

This example produces a cohort survival curve which is almost always plotted along with the observed (Kaplan-Meier) survival of the data for visual comparison. For this example, you can plot the survival curves together as follows:

```
> plot(survfit(Surv(stop, event) ~ surgery, data = hearta),
+ lty = 2:3)
> lines(expsurv, lty = 2:3)
> legend(750, 0.9, c("No Prior Surgery", "Prior Surgery"),
+ lty = 2:3)
```

Figure 31.1 displays the resulting plot. In general, there are three different methods for calculating the cohort curve, which are discussed in detail in the section Cohort Expected Survival. They are:

1. The conditional method illustrated in the above example, which uses the actual death or censoring time;
2. The method of Hakulinen, which uses the potential follow-up time of each subject;
3. The uncensored population method of Ederer, Axtel, and Cutler, which requires no response variable.

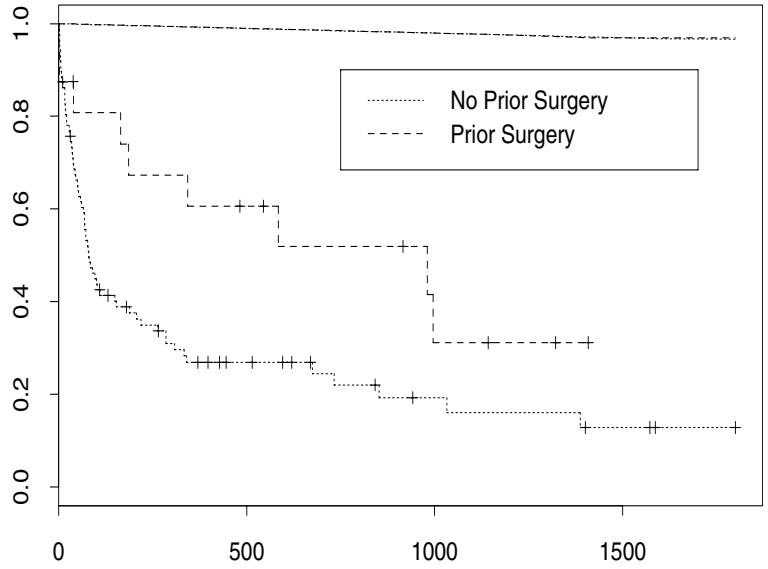

Figure 31.1: *Comparison of the heart transplant study population stratified according to prior surgery to a matched cohort from a national survival rate table.*

Individual Expected Survival Probabilities

Formal tests of observed versus expected survival are not usually based on the cohort curve directly; instead, they are based on the individual expected survival probabilities for each subject. These probabilities are always based on the actual death/censoring time:

```
> surv.prob <- survexp(stop ~ ratetable(
+ age = (age + 48) * 365.25, sex = "male",
+ year = year * 365.25), data = hearta, cohort = F)

> # convert from survival to hazard
> newtime <- -log(surv.prob)

> summary(glm(stop ~ offset(log(newtime)),
+ family = poisson, data = hearta))

Call: glm(formula = stop ~ offset(log(newtime)), family =
poisson, data = hearta)
Deviance Residuals:
       Min         1Q     Median         3Q        Max
 -34.04402  -3.601203  -0.5733439  4.343454  39.95069
```

```
Coefficients:
               Value  Std. Error  t value
(Intercept) 10.77882 0.005593555 1927.006

(Dispersion Parameter for Poisson family taken to be 1 )

    Null Deviance: 13699.27 on 102 degrees of freedom

Residual Deviance: 13699.27 on 102 degrees of freedom

Number of Fisher Scoring Iterations: 4
```

When `cohort=F`, the `survexp` function returns a vector of survival probabilities, one per subject. For the purposes of modeling, the negative log of the survival probability can be treated as an "adjusted time" for the subject. The one-sample log-rank test for equivalence of the observed survival to the expected survival is the test for intercept equal to zero in the Poisson regression model shown. A test for treatment difference, adjusted for any age-sex differences between the two arms, is obtained by adding a treatment variable to the model.

Computing Person Years

Expected survival is closely related to a standard method in epidemiology called *person years*, which consists of counting the total amount of follow-up time contributed by the subjects within any of several strata. Person-years analysis is accomplished in Spotfire S+ with the `pyears` function. The main complication in computing person years is that a subject may contribute to several different cells of the output array during his/her follow-up. For example, if the desired output table is treatment group by age in years, subjects with 4 years of observation each contribute to five different cells of the table (four cells if they entered the study exactly on their birthdates).

This example counts up years of observation for the Stanford heart patients by age group and surgical status. Using the `hearta` data frame computed above, the person-years table is produced as follows:

```
> pyears(stop/365.25 ~ tcut(age + 48, c(0, 50, 60,
+ 70, 100)) + surgery, data = hearta, scale = 1)

$call:
pyears(formula = stop/365.25 ~ tcut(age + 48,
c(0,50,60,70,100)) + surgery, data = hearta, scale = 1)
```

```
$pyears:
                        0         1
 0+ thru  50 44.9253936 18.960986
50+ thru  60 16.7501711  6.093087
60+ thru  70  0.7556468  0.000000
70+ thru 100  0.0000000  0.000000

$n:
              0  1
 0+ thru  50 56 13
50+ thru  60 33  6
60+ thru  70  3  0
70+ thru 100  0  0

$offtable:
[1] 0
```

The `scale` argument is provided because `pyears` defaults to input times in days and output times in years (`scale=365.25`). A 48 is added to `age` to relocate it back to its original scale. For `surgery`, a 0 corresponds to no prior surgery and a 1 corresponds to prior surgery. See the help file for `heart` for more detail.

The `tcut` function has the same arguments as `cut`, but also indicates that the category is time based. If you use `cut` in the formula above, the final table is based only on each subject's age at entry. With `tcut`, a subject who entered at age 58.5 and had 4 years of follow-up contributed 1.5 years to the 50-60 category and 2.5 years to the 60-70 category. A consequence of this is that the `age` and `stop` variables must be in the same units for the calculation to proceed correctly. In this case both should be in years, given the cutpoints that were chosen. The `surgery` variable is treated as a factor, exactly as it is treated by `survfit`.

The output of `pyears` is a list of arrays containing the total amount of time contributed to each cell, and the number of subjects who contributed some fraction of time to each cell. The `offtable` component that is returned is the number of person years of exposure in the cohort that is not part of any cell in the `pyears` component. This is often useful as an error check. If there is a mismatch of units between two variables, nearly all the person years may be in `offtable`.

If the response variable is a Surv object, the output also contains an array with the observed number of events for each cell. If a rate table is supplied, the output contains an array with the expected number of events in each cell. These can be used to compute observed and expected rates, along with confidence intervals.

Using a Cox Model as a Rate Table

Many times, a study group is compared to a historical control. If the comparison is to be adjusted for differences in certain covariates, it is usually based on a Cox model fit to the historical data. The methods used in this example are parallel to the previous examples using national rate tables (for example, survexp.us). However, in this example, a prior Cox model acts as the "rate table" for survexp.

Individual survival curves can be obtained using survfit, as described in Chapter 28, The Cox Proportional Hazards Model. For convenience, we reproduce an example from that chapter here:

```
> ov.fit1 <- coxph(Surv(futime, fustat) ~ age,
+ data = ovarian)
```

Extending the example, the command

```
> s1 <- survfit(ov.fit1, newdata = data.frame(age = 35))
```

gives the expected curve for a 35 year old subject. In addition,

```
> s2 <- survfit(ov.fit1, newdat = ovarian)
```

gives a matrix of 26 survival curves, one for each subject in the ovarian data set.

The Ederer estimate is the average of the 26 survival curves in s2 and can be obtained as follows:

```
> s3 <- survexp(~ ratetable(age = age), data = ovarian,
+ ratetable = ov.fit1)
```

In the Cox model literature, the Ederer estimate has been called the *direct adjusted survival curve*. Thomsen, Keiding, and Altman (1991) point out the importance of the Ederer estimate and the difference between the Ederer estimate, average survival, and the individual survival of a subject with the average age.

The equivalent of Hakulinen's estimate has been labeled the *Bonsel* estimator. For studies with a short accrual, it does usually not differ from the Ederer method. Thomsen, *et al.* (1991) also discuss the conditional estimator, but conclude that the final curve is "hard to interpret."

CREATING RATE TABLES

You can create your own rate tables to use in place of those provided in Spotfire S+. Table 31.2 through Table 31.5 show yearly death rates per 100,000 subjects based on their smoking status.

Table 31.2: *Death rates for former male light smokers (1–20 cigarettes per day).*

			Duration of Abstinence (years)					
Age	**Never Smoked**	**Current Smokers**	**< 1**	**1–2**	**3–5**	**6–10**	**11–15**	**≥ 16**
45–49	186.0	439.2	234.4	365.8	159.6	216.9	167.4	159.5
50–54	255.6	702.7	544.7	431.0	454.8	349.7	214.0	250.4
55–59	448.9	1132.4	945.2	728.8	729.4	590.2	447.3	436.6
60–64	733.7	1981.1	1177.7	1589.2	1316.5	1266.9	875.6	703.0
65–69	1119.4	3003.0	2244.9	3380.3	2374.9	1820.2	1669.1	1159.2
70–74	2070.5	4697.5	4255.3	5083.0	4485.0	3888.7	3184.3	2194.9
75–79	3675.3	7340.6	5882.4	6597.2	7707.5	4945.1	5618.0	4128.9

Assume the eight data columns are stored in a file named **data.smoke**. A rate table is created using the following Spotfire S+ code:

```
> temp <- matrix(scan("data.smoke"), ncol = 8,
+ byrow = T)/100000

> smoke.rate <- c(rep(temp[,1], 6), rep(temp[,2], 6),
+ temp[,3:8])

> attributes(smoke.rate) <- list(
+ dim = c(7, 2, 2, 6, 3),
+ dimnames = list(c("45-49", "50-54", "55-59", "60-64",
+ "65-69", "70-74", "75-79"),
```

```
+ c("1-20", "21+"), c("Male", "Female"),
+ c("<1", "1-2", "3-5", "6-10", "11-15", ">=16"),
+ c("Never", "Current", "Former")),
+ dimid = c("age", "amount", "sex", "duration", "status"),
+ factor = c(0,1,1,0,1),
+ cutpoints = list(c(45, 50, 55, 60, 65, 70, 75),
+ NULL, NULL, c(0, 1, 3, 6, 11, 16), NULL),
+ class = "ratetable")

> is.ratetable(smoke.rate)
```

The smoking data cross-classifies subjects by five characteristics: age group, sex, status (`Never`, `Current`, or `Former` smoker), the number of cigarettes consumed per day, and for the prior smokers, the duration of abstinence.

Table 31.3: *Death rates for former male heavy smokers (more than 21 cigarettes per day).*

			Duration of Abstinence (years)					
Age	**Never Smoked**	**Current Smokers**	**< 1**	**1–2**	**3–5**	**6–10**	**11–15**	**≥ 16**
45–49	186.0	610.0	497.5	251.7	417.5	122.6	198.3	193.4
50–54	255.6	915.6	482.8	500.7	488.9	402.9	393.9	354.3
55–59	448.9	1391.0	1757.1	953.5	1025.8	744.0	668.5	537.8
60–64	733.7	2393.4	1578.4	1847.2	1790.1	1220.7	1100.0	993.3
65–69	1119.4	3497.9	2301.8	3776.6	2081.0	2766.4	2268.1	1230.7
70–74	2070.5	5861.3	3174.6	2974.0	3712.9	3988.8	3268.6	2468.9
75–79	3675.3	6250.0	4000.0	4424.8	7329.8	6383.0	7666.1	5048.1

Table 31.4: *Death rates for former female light smokers (1–20 cigarettes per day).*

			Duration of Abstinence (years)					
Age	**Never Smoked**	**Current Smokers**	**< 1**	**1–2**	**3–5**	**6–10**	**11–15**	**≥ 16**
45–49	125.7	225.6		433.9	212.0	107.2	135.9	91.0
50–54	177.3	353.8	116.8	92.1	289.5	200.9	121.3	172.1
55–59	244.8	542.8	287.4	259.5	375.9	165.8	202.2	247.2
60–64	397.7	858.0	1016.3	365.0	650.9	470.8	570.6	319.7
65–69	692.1	1496.2	1108.0	1348.5	1263.2	864.8	586.6	618.0
70–74	1160.0	2084.8	645.2	1483.1	1250.0	1126.3	1070.5	1272.1
75–79	2070.8	3319.5		2580.6	2590.7	3960.4	1666.7	1861.5

Table 31.5: *Death rates for former female heavy smokers (more than 21 cigarettes per day).*

			Duration of Abstinence (years)					
Age	**Never Smoked**	**Current Smokers**	**< 1**	**1–2**	**3–5**	**6–10**	**11–15**	**≥ 16**
45–49	125.7	277.9	266.7	102.7	178.6	224.7	142.1	138.8
50–54	177.3	517.9	138.7	466.8	270.1	190.2	116.8	83.0
55–59	244.8	823.5	473.6	602.0	361.0	454.5	412.2	182.1
60–64	397.7	1302.9	1114.8	862.1	699.6	541.7	373.1	356.4
65–69	692.1	1934.9	2319.6	1250.0	1688.0	828.7	797.9	581.5

Table 31.5: *Death rates for former female heavy smokers (more than 21 cigarettes per day). (Continued)*

			Duration of Abstinence (years)					
Age	**Never Smoked**	**Current Smokers**	**< 1**	**1–2**	**3–5**	**6–10**	**11–15**	**≥ 16**
70–74	1160.0	2827.0	4635.8	2517.2	1687.3	2848.7	1621.2	1363.4
75–79	2070.8	4273.1	2409.6	5769.2	3125.0	2987.7	2803.7	2195.4

In the Spotfire S+ implementation, a rate table is an array with added attributes, and thus must be rectangular. In order to cast the above data into a single array, the rates for the `Never` and `Current` smokers need to be replicated across all six levels of the duration. We do this by first creating the `smoke.rate` vector. The array of rates is then saddled with a list of descriptive attributes. The `dim` and `dimnames` attributes are as they would be for an array, and give its shape and printing labels, respectively. The `dimid` attribute is the list of keywords to be is recognized by the `ratetable` function when the table is used with the `survexp` or `pyears` function. For the U.S. total table, for instance, the keywords are `"age"`, `"sex"`, and `"year"`. These keywords must be in the same order as the array dimensions (as found in the `dimid` attribute).

The `factor` attribute of a rate table identifies each dimension as fixed or varying with time. For a subject with fifteen years of follow-up, for example, the sex category remains fixed but the age and duration of abstinence continue to change; more than one of the age groups must be referenced to compute the subject's total hazard. For each dimension that is not a factor, the starting value for each of the rows of the array must be specified so that the routine can change rows at the appropriate time. This information is specified in the `cutpoints` attribute. The cutpoints are null for a factor dimension. Because the cutpoints must be self-consistent, you should check them for any rate tables you create. The function `is.ratetable` does this for you automatically.

As an example, we apply our smoke.rate rate table to the hearta data, assuming that all of the subjects were current heavy smokers:

```
> ptime <- hearta$stop/365.24
> exp4 <- survexp(ptime ~ ratetable(age = (age/365.24),
+ status = "Current", amount = "21+", duration = "<1",
+ sex = "Male"), data = hearta, ratetable = smoke.rate,
+ conditional = F, scale = 1)
```

This example illustrates some important points. First, since we are using the current smoker category, duration is unimportant, so any value can be specified. Second, note that we must rescale age. The smoke.rate table contains rates per year, while the U.S. tables contain rates per day. It is crucial that all of the time variables (age, duration, etc.) be scaled to the same units, or the results may not be correct. The U.S. rate tables were created using days as the basic unit, since year of entry is normally a Julian date; for the smoking data, years seems more natural.

An optional portion of a rate table, not illustrated in the example above, is a summary attribute. This is a user-written function which is given a matrix and returns a character string. The matrix must have one column per dimension of the rate table, in the order of the dimid attribute, and must be preprocessed to remove illegal values. To see an example of a summary function, use the following command:

```
> attr(survexp.us, "summary")
```

In this summary function, the returned character string lists the range of ages and calendar years in the output of survexp. This printout is the only good way to catch errors in the time units. For example, if the string contained "age ranges from 0.13 to 0.26 years," it is a reasonable guess that age was given in years when it should have been stated in days.

REFERENCES

Andersen, P.K. & Væth, M. (1989). Simple parametric and non-parametric models for excess and relative mortality. *Biometrics* **45**:523-535.

Berkeson, J. & Gage, R.P. (1950). Calculation of survival rates for cancer. *Proc. Staff Meeting Mayo Clinic* **25**: 270-286.

Berry, G. (1983). The analysis of mortality by the subject years method. *Biometrics* **39**:173-184.

Ederer, F., Axtell, L.M., & Cutler, S.J. (1961). The relative survival rate: A statistical methodology. *National Cancer Institute Monographs* **6**:101-121.

Ederer, F. and Heise, H. (1977). *Instructions to IBM 650 programmers in processing survival computations.* Methodological Note No. 10, End Results Evaluation Section, National Cancer Institute.

Hakulinen, T. (1982). Cancer survival corrected for heterogeneity in patient withdrawal. *Biometrics* **38**:933.

Hakulinen, T. & Abeywickrama, K.H. (1985). A computer program package for relative survival analysis. *Computer Programs in Biomedicine* **19**:197-207.

Harrington, D.P. and Fleming, T.R. (1982). A class of rank test procedures for censored survival data. *Biometrika* **69**:553-566.

Hartz, A.J., Giefer, E.E., and Hoffmann, G.G. (1983). A comparison of two methods for calculating expected mortality. *Statistics in Medicine* **2**:381-386.

Hartz, A.J., Giefer, E.E., & Hoffmann, G.G. (1984). Letter and rejoinder on "A comparison of two method for calculating expected mortality." *Statistics in Medicine* **3**:301-302.

Hartz, A.J., Giefer, E.E., & Hoffmann, G.G. (1985). Letters and rejoinder on "A comparison of two method for calculating expected mortality." *Statistics in Medicine* **4**:105-109.

Mantel, N. (1966). Evaluation of survival data and two new rank order statistics arising in its consideration. *Cancer Chemotherapy Reports* **50**:163-166.

Thomsen B.L., Keiding N., & Altman D.G. (1991). A note on the calculation of expected survival, illustrated by the survival of liver transplant patients. *Statistics in Medicine* **10**:733-738.

Verheul, H.A., Dekker, E., Bossuyt, P., Moulijn, A.C., & Dunning, A.J. (1993). Background mortality in clinical survival studies. *Lancet* **341**:872-875.

QUALITY CONTROL CHARTS 32

INTRODUCTION

Spotfire S+ provides several functions for doing quality control charts. Table 32.1 lists the type of basic charts available. Both Shewhart and cusum charts are available for each basic chart type except the R chart, for which a cusum chart has not been implemented. Ryan (1989) provides a good discussion of the use and utility of both Shewhart and cusum charts

Table 32.1: *Types of basic quality control charts available in Spotfire S+.*

Type	Statistic Charted	Chart Description
`xbar`	Mean	Means of a continuous process variable
`s`	Standard Deviation	Standard deviations of a continuous process variable
`R`	Range	Ranges of a continuous process variable
`np`	Count	Number of nonconforming units
`p`	Proportion	Proportion of nonconforming units
`c`	Count	Number of nonconforming units
`u`	Count	Number of nonconforming units for variable unit sizes

In addition to the basic chart types listed in Table 32.1, several extensions to Shewhart charts allow charting non-grouped, one-at-a-time data. These extensions typically use standard deviation estimates based on moving or sliding intervals of data values to improve the power of the resulting chart. The extensions are listed in Table 32.2.

Table 32.2: *Types of extended Shewhart control charts available in Spotfire S+.*

Type	Statistic Charted	Chart Description
`ma`	Moving Average	Moving means of a continuous process variable
`ms`	Moving Standard Deviation	Moving standard deviations of a continuous process variable
`mR`	Moving Range	Moving ranges of a continuous process variable
`ewma`	Moving Average	Exponentially weighted moving average of a continuous process variable

CONTROL CHART OBJECTS

Spotfire S+ quality control charts are produced in two steps:

1. Create a `qcc` object from process data known to be gathered when the process was in a state of control.
2. Create a chart of new data using the `qcc` object as the reference data.

You can think of the `qcc` object as containing the data necessary to calibrate the control chart. It contains information on the type of chart being plotted as well as the process center and variability, which are necessary to compute the control limits.

The `qcc` function produces an object of class `"qcc"`. Its only required arguments are `data` and `type`, which specify the process data (grouped appropriately) and the chart type, respectively. A simple example is given below.

```
# Set the seed for reproducibility.
> set.seed(15)
> qcdata <- matrix(10 + rnorm(100), ncol = 5)
> qccobj <- qcc(qcdata, type = "xbar")
```

A `print` method summarizes the `qcc` object:

```
> qccobj

xbar based on qcdata

Summary of Group Statistics:
  Min. 1st Qu. Median  Mean 3rd Qu.  Max.
 9.163   9.655  10.14 10.09   10.51 11.31

 Group Sample Size:  5
 Number of Groups:  20
 Center of Group Statistics:  10.09016
 Standard Deviation:  1.022341
```

In this example, the `data` argument is given as a matrix. In general, if `data` is a matrix or data frame, each row represents a group and the number of columns corresponds to the group size. If you have unequal group sizes, your process data must be a list that has one component for each group.

The arguments to `qcc` are:

- `data`, the control data in the form of a vector, matrix, data frame, or list.
- `type`, a character string or function specifying the group statistics to compute.
- `std.dev`, a numeric vector or function for specifying the within-group standard deviation(s).
- `sizes`, a numeric vector specifying the sample sizes associated with each group.
- `labels`, a character vector of labels for each group.

You can pass functions to the `type` and `std.dev` arguments to extend the built-in capabilities of `qcc`. The function that is used by default to compute the group summary statistics and the center of the group summary statistics is named `stats.`*`type`*, where *`type`* corresponds to the value of the `type` argument. For example, the default summary statistics and center for an xbar chart are computed by `stats.xbar`. Similarly, the default function that computes the standard deviation for an xbar chart is `sd.xbar`. When `type` is given as a function, `std.dev` must also be given (usually as a function as well, though not necessarily).

You can use the `stats.xbar` and `sd.xbar` functions as templates for additional functions accepted by the `type` and `std.dev` arguments. For example, the function below is similar to `stats.xbar` but computes the summary statistics and the center as medians instead of means.

```
> stats.med

function(data, sizes)
{
        if(is.list(data)) {
                statistics <- sapply(data, median)
                center <- median(unlist(data))
        }
        else {
                statistics <- apply(data, 1, median)
                center <- median(data)
        }
        list(statistics = statistics, center = center)
}
```

The `stats.med` function depends on `data` being given as a matrix or list. The `qcc` function insures this by coercing vectors to matrices.

The following function is derived from `sd.xbar`, and computes a standard deviation based on the median absolute deviation:

```
> sd.med

function(data, sizes)
{
  if(is.list(data))
    std.dev.within <- sapply(data, mad)
  else {
    std.dev.within <- apply(data, 1, mad)
    if(dim(data)[2] == 1)
      warning("MAD computation based on group sizes of 1")
  }
  if(length(sizes) == 1)
    sizes <- rep(sizes, length = length(std.dev.within))
  sum(sizes * std.dev.within)/sum(sizes)
}
```

You can now compute a `qcc` object with the `stats.med` and `sd.med` functions as follows:

```
> qccobj.med <- qcc(qcdata, type = "med")
```

```
> qccobj.med

med based on qcdata

Summary of Group Statistics:
   Min. 1st Qu. Median   Mean 3rd Qu.   Max.
  8.782  9.599  10.060  9.989 10.520  11.160

 Group Size:  5
 Number of Groups:  20
 Center of Group Statistics:  10.14026
 Standard Deviation:  0.8418576
```

If the functions are not named with the proper prefixes (`stats` and `sd`, respectively), you must pass the function names to the `type` and `std.dev` arguments explicitly. For example, if your functions are named `st.med` and `sd.mad`, you would need to type:

```
> qccobj.med <- qcc(qcdata, type = st.med,
+ std.dev = sd.mad)
```

To chart the control data and any ongoing process data, you can produce Shewhart or cusum charts with the Spotfire S+ functions `shewhart` and `cusum`. Typically, Shewhart charts are used for detecting large shifts in a process (two to three sigma shifts), whereas cusum charts are used to detect smaller shifts in a process (one-half to one sigma shifts).

SHEWHART CHARTS

Overview

You can produce a Shewhart chart of the data in `qcdata` by using the `shewhart` function. For example:

```
> shewhart(qccobj)
```

Note that `qcdata` is preserved in `qccobj` as a `qcc` object, which is the type of object that `shewhart` requires. Figure 32.1 displays the resulting chart.

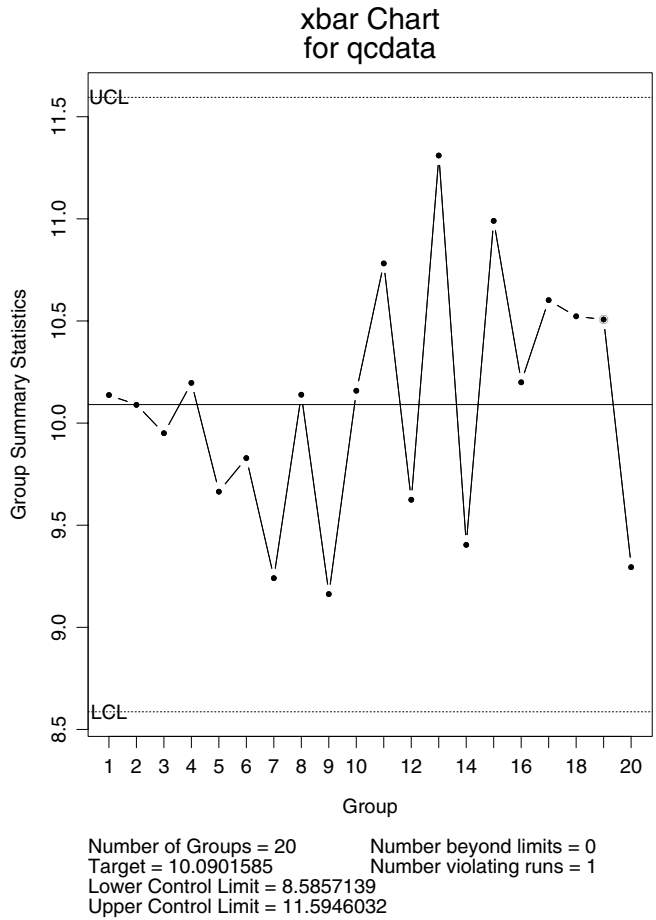

Figure 32.1: *Shewhart chart of the data in* `qccobj`.

The text at the bottom of the chart displays pertinent statistics. The `target` value is the center of the group summary statistics unless it is given as a separate argument to `shewhart`. The `Number beyond limits` indicates the number of points beyond the control limits. The `Number violating runs` indicates how many points violate the runs criterion; by default, this criterion is five or more consecutive points on one side of the center. You can change the run length by specifying the `run.length` argument to `shewhart`:

```
> shewhart(qccobj, run.length = 8)
```

By default, the `shewhart` function computes control limits based on the `center` and `std.dev` components of the `qcc` object. As with the runs criterion, however, both of these can be overridden by providing additional arguments in the call to `shewhart`.

The default control limits produced by `shewhart` are *probability limits* for all charts except the u chart. Probability limits are centered in probability about the estimate of the center of the summary statistics' distribution; if the `target` argument to `shewhart` is specified, probability limits are centered about the target value instead. In contrast, you can specify *sigma limits* through the `nsigmas` argument to `shewhart`. In this case, the control limits are placed at the center, plus or minus `nsigmas` times the standard errors of the group summary statistics. For u charts, only sigma limits are implemented. If the sample sizes vary, the standard errors also vary and a step function is plotted for each control limit.

Arguments and Return Values

The arguments to `shewhart` are as follows.

- `object`: An object of class `"qcc"`, which provides information on the type of group summary statistics to plot and the within-group standard deviation necessary for computing the control limits.
- `newdata`: A vector, matrix, data frame, or list to be charted. By default, Spotfire S+ charts the data used to create `object`.
- `type`: A character string or function specifying the group summary statistics to compute.
- `limits`: A numeric vector or matrix, or a function specifying the control limits.

- `target`: A numeric value specifying the center of the process. By default, this is the `center` component of `object`.
- `std.dev`: A numeric value specifying the overall within-group standard deviation.
- `sizes`: A vector of the number of observations (or the number of units examined) in each group of `newdata`. By specifying sample sizes, you can supply a vector of group summary statistics to `newdata` instead of the entire data matrix. In this case however, you must also specify the within-group standard deviations.
- `labels`: A character vector of labels for each group in `newdata`.
- `label.limits`: A character vector of length two with labels for the control limits.
- `confidence.level`: A numeric value between 0 and 1 specifying the confidence level of the computed probability limits. By default, a confidence level of 0.999 is used. If `confidence.level` is given, `nsigmas` is ignored.
- `nsigmas`: A numeric value specifying one-half the width of the control limits in the number of standard errors of the group summary statistics. If `nsigmas` is given, `confidence.level` is ignored.
- `add.stats`: A logical value indicating whether statistics should be listed at the bottom of the chart.
- `chart.all`: A logical value indicating whether the `statistics` component of `object` should be plotted along with the `new.statistics` component and the summary statistics of `newdata` (if given).
- `ylim.min`: A numeric vector of values to be included in the computation of the approximate *y*-axis limits for the control chart.
- `rules`: A function of rules to apply to the chart. By default, this is the `shewhart.rules` function.
- `highlight`: A list of plotting parameters to be used for highlighting the points violating `rules`.
- `...`: Additional arguments to the `rules` function.

See the `shewhart` help file for more detailed descriptions of the arguments listed above.

The `shewhart` function returns an object that contains all the components of the reference `object`, plus the following additional components.

- `new.statistics`: A vector of group summary statistics for `newdata`.
- `new.sizes`: A vector of group sample sizes for `newdata`.
- `target`: The `target` argument if specified.
- `cntrl.limits`: The control limits.
- `newdata.name`: A character string containing the name of the input data passed to the `newdata` argument.

Specifying New Data

The `newdata` argument to `shewhart` allows you to chart new data with a reference `qcc` object provided as the `object` argument. As an illustration, we add 1/ 2 to the last six rows of `qcdata` and call it `newdata`:

```
> newdata <- qcdata
> newdata[15:20,] <- newdata[15:20,] + 1/2
```

The Shewhart chart of `newdata` is as follows:

```
> qccobj.shew <- shewhart(qccobj, newdata = newdata,
+ labels = paste("Lot", 1:20, sep = ""))
```

We use the optional `labels` argument to print descriptive labels on the chart, which provide clarity in later examples of this section. Figure 32.2 displays the resulting chart. In addition to viewing the chart, we can also print the invisible return value of `shewhart` to see a summary of both `qccobj` and `newdata`. The command below displays the `qccobj.shew` object.

```
> qccobj.shew

xbar based on qcdata

Summary of Statistics in qcdata.
  Min. 1st Qu. Median  Mean 3rd Qu.  Max.
 9.163   9.655  10.14 10.09   10.51 11.31
```

```
 Group Sample Size:  5
 Number of Groups:  20
 Center of Statistics:  10.09016
 Standard Deviation:  1.022341

Summary of New Data Statistics in newdata.
  Min. 1st Qu. Median  Mean 3rd Qu.  Max.
 9.163   9.762  10.14 10.24   10.84 11.49

 Group Sample Size:  5
 Number of Groups:  20

 Target Value: 10.09016

 Control Limits:
      LCL     UCL
 8.585714 11.5946
```

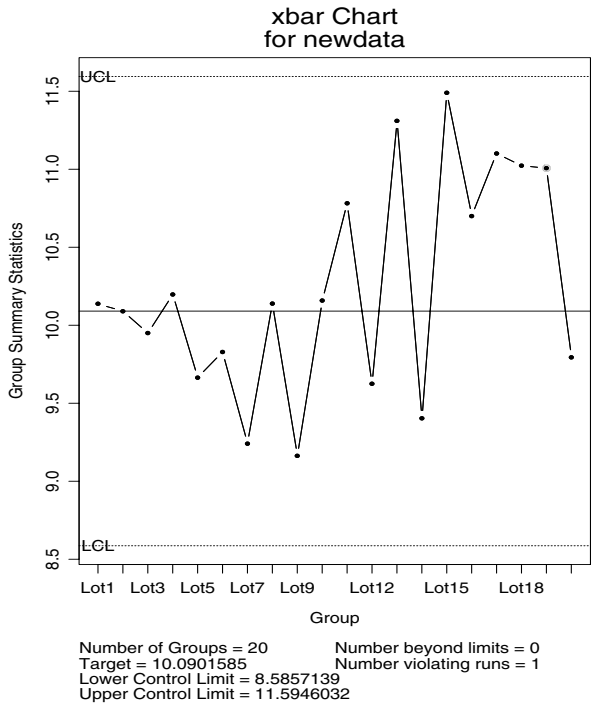

Figure 32.2: *Shewhart chart of* `newdata` *using* `qccobj` *as the reference data.*

If you want to see `newdata` displayed alongside the original calibration data, use the `chart.all` argument to `shewhart`:

```
> shewhart(qccobj.shew, chart.all = T)
```

Figure 32.3 shows the Shewhart chart that displays both the old and new data. The vertical dashed line separates the in-control calibration data from the ongoing process data.

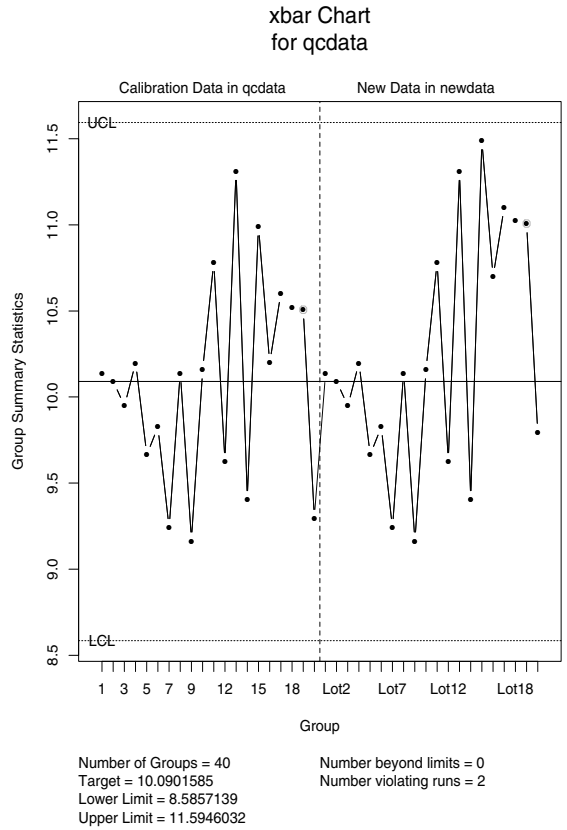

Figure 32.3: *Shewhart chart of `newdata` using `qccobj` as the reference data. Both the new and the old data are displayed in the plot.*

To create an s chart of `newdata` using `qcdata` for calibration, type:

```
> shewhart(qcc(qcdata, type = "s"), newdata = newdata)
```

The type argument to shewhart allows you to specify a summary statistic for newdata that is different than the one used to compute the reference object. For example, the object qccobj.med from the section Control Chart Objects on page 446 contains robust estimates of location and scale for the reference data qcdata. Typically, you do not want to estimate the location of the ongoing process robustly, since extreme values are of key interest. In this case, you can compute control limits based on the robust estimates, and then compute the group summary statistics of the ongoing process by specifying the usual type for your data. For example:

```
> shewhart(qccobj.med, newdata = newdata, type = "xbar")
```

If you want to compute the summary statistics for newdata in the same way you did for the reference data, the type argument is not needed. Thus, the following command estimates the group summary statistics of the ongoing process robustly with the stats.med function:

```
> shewhart(qccobj.med, newdata = newdata,
+ limits = limits.xbar)
```

The limits argument is required when using a summary statistics function that is not built into Spotfire S+, such as stats.med. Otherwise, a function with the name limits.*type* must exist. In the above example, the limits argument is required unless a function named limits.med exists, since we use type="med" to create the qccobj.med object. The limits.xbar function uses the center and std.dev components of object to compute control limits based on normally distributed data, so it is reasonable (though not exactly correct) to use in this example. Ideally, you would write a custom limits.med function. For more information on the way the control limits are computed by shewhart, see the help file for limits.xbar. Any of the functions in the help file can be used as a template when writing a custom limits function.

When tracking a process, it is possible to repeatedly capture the return value from shewhart and pass it as the new object argument in a subsequent shewhart call. In addition, you can provide even newer data as the newdata argument, which allows you to continually track a process without recomputing the reference qcc object. The shewhart function incorporates the newest data into the new.statistics

component of `object`, and then charts all the new data accordingly. In this situation, the function calls might look something like the following:

```
> qccobj.shew.1 <- shewhart(qccobj, newdata = newdata.1)
> qccobj.shew.2 <- shewhart(qccobj.shew.1,
+ newdata = newdata.2)
```

Customizing Shewhart Charts

A *rules function* for `shewhart` refers to a method of examining the plotted summary statistics for patterns that suggest a shift in the process. For example, five or more successive points on one side of the center may indicate a shift in the process. The function `runs.target` checks for runs in a process, and the function `beyond.limits` locates points beyond the control limits. For details about either of these functions, see their help files.

By default, `shewhart` applies both `runs.target` and `beyond.limits` through the wrapper function `shewhart.rules`, which highlights points in the same way regardless of which rule is violated. If you want to display the points violating the two rules differently, provide the appropriate graphical parameters to the `highlight` argument of `shewhart`. For example:

```
> shewhart(qccobj.shew, highlight = list(
+ list(pch=1, col=2), list(pch=2, col=3)))
```

In addition to graphical displays, you can view a list of violating points by calling `runs.target`, `beyond.limits`, or `shewhart.rules` directly. Each of these functions accept objects returned by `shewhart`. For example:

```
> shewhart.rules(qccobj.shew)

[[1]]:
numeric(0)
attr([[1]], "names"):
character(0)
attr([[1]], "label"):
[1] "beyond limits"
```

```
[[2]]:
 Lot19
    19
attr([[2]], "label"):
[1] "violating runs"
```

To add labeling information to a Shewhart chart, you can use the `identify` function. To do this, chart a `shewhart` object with no statistics and then apply `identify`:

```
> shewhart(qccobj.shew, add.stats = F)
> identify(qccobj.shew)

[1] 19
```

Left-click on the highlighted point in the chart, which is `Lot 19` in this example. To exit `identify`, press either the middle or right mouse button, and `19` is returned at the Spotfire S+ prompt. Figure 32.4 displays the resulting chart with the 19th observation labeled.

It is possible to use `runs.target` to make a Shewhart chart more sensitive to small shifts off the center. However, such rules are typically *ad hoc*. A better way to detect small shifts is through the use of *cusum charts*, which we discuss in the next section.

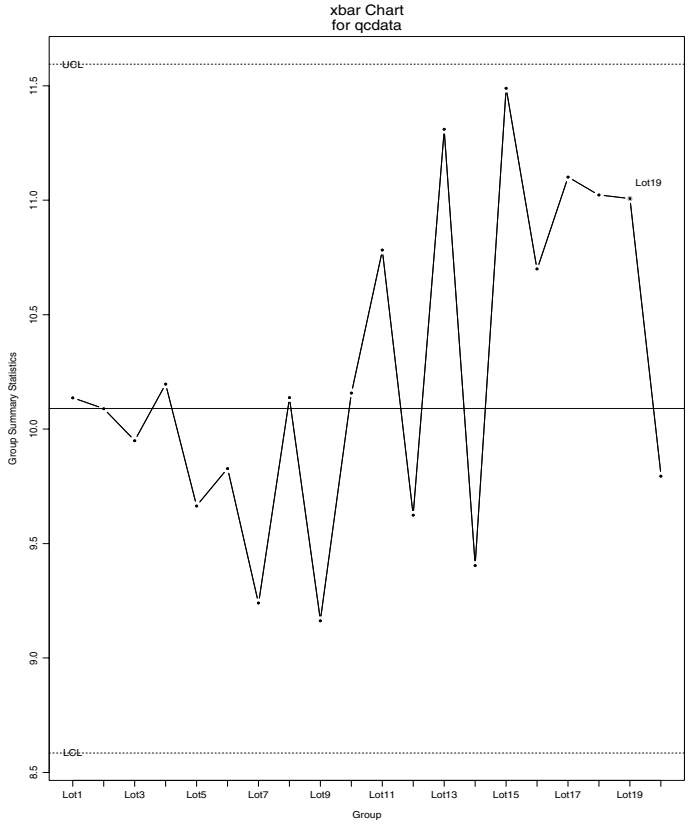

Figure 32.4: *Shewhart chart of the new data in* `qccobj.shew` *with the 19th observation labeled.*

CUSUM CHARTS

Overview

Cusum charts display how the group summary statistics deviate above or below the process center or target value, relative to the standard errors of the summary statistics. In essence, a cusum chart accumulates *z*-scores of deviations above (or below) the center and then charts them. Consequently, the points plotted are not the original data, but are cumulative sums of deviations in standard errors from the center. Cusum charting in Spotfire S+ follows a decision interval scheme discussed in detail by Ryan (1989) and Wetherill and Brown (1991).

For the *i*th group in an xbar chart, the upper cumulative sum S_{Ui} and lower cumulative sum S_{Li} are defined as follows:

$$S_{Ui} = max\{0,\ (z_i - k) + S_{Ui-1}\} \tag{32.1}$$

$$S_{Li} = max\{0,\ (-z_i - k) + S_{Li-1}\} \tag{32.2}$$

In these equations, z_i is the *z*-score for the *i*th group centered about $\bar{\bar{x}}$, the center of the group summary statistics :

$$z_i = \frac{\bar{x}_i - \bar{\bar{x}}}{\sigma_{\bar{x}_i}}$$

The lower cumulative sums are charted as $-S_{Li}$. In both Equation (32.1) and Equation (32.2), k is called the *reference value* and corresponds to the amount by which the absolute *z*-score must exceed the target before either cumulative sum increases.

A cusum chart in Spotfire S+ is really a composite of two charts: one of the upper cumulative sums and one of the negative lower cumulative sums. The upper and lower sums, typically charted separately in standard quality control text books, are plotted on the same graph by the `cusum` function in Spotfire S+.

In the section Control Chart Objects on page 446, we simulated some process data in the matrix `qcdata`, and created a corresponding `qcc` object named `qccobj`. You can plot a cusum chart of the data with the following command:

```
> cusum(qccobj)
```

Note that `qcdata` is preserved in `qccobj` as a `qcc` object, which is the type of object that `cusum` requires. In the section Specifying New Data on page 453, we also created a matrix called `newdata`. To see the new data charted, request it in addition to the reference data in the call to `cusum`. The following command uses the `chart.all=T` argument to plot both the new and old data on the same graph:

```
> cusum(qccobj, newdata = newdata, chart.all = T)
```

Figure 32.5 displays the resulting chart. Compare this figure with the Shewhart chart displayed in Figure 32.2 to see how dramatically cusum charts signal a detectable shift in the process. In `newdata`, the last six observations are shifted up from `qcdata` by one standard deviation of the population, which is about two standard errors of the summary statistics. This shift is not highlighted in the Shewhart chart, but it is clearly detected in the cusum chart.

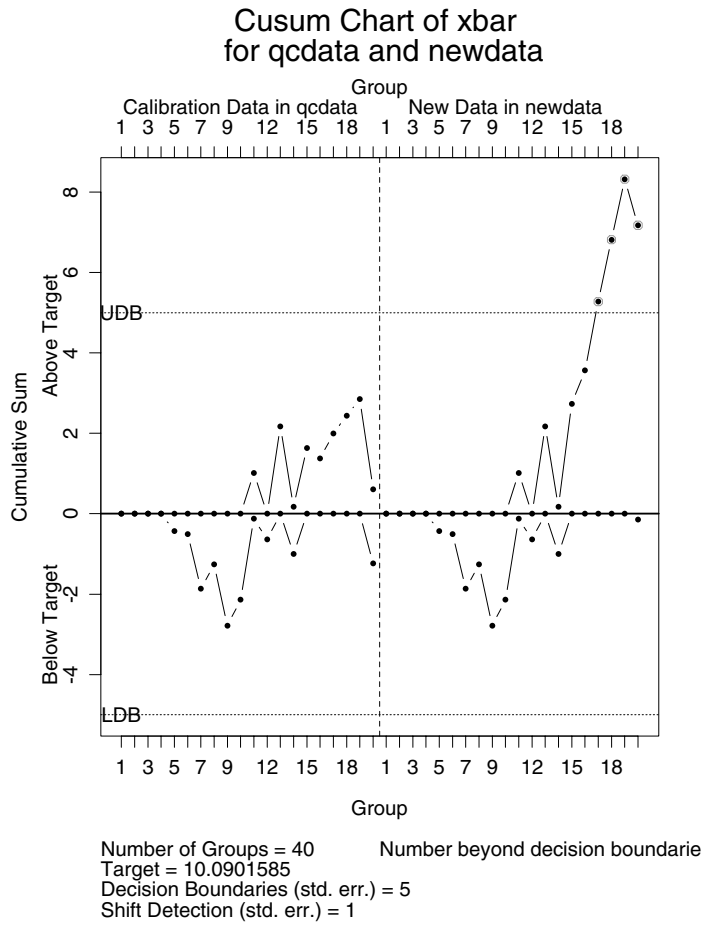

Figure 32.5: *Cusum chart of* `newdata` *using* `qccobj` *as the reference data. Both the new and old data are included in the chart.*

Arguments and Return Values

A summary of the arguments to `cusum` are as follows.

- `object`: An object of class `"qcc"`, which provides information on the type of group summary statistics to compute and the within group standard deviation necessary for computing the z-scores.
- `newdata`: A vector, matrix, data frame, or list to be charted.
- `type`: A character string or function specifying the group statistics to compute.

- `z.scores`: An optional function to compute the *z*-scores. This argument is required if `type` is not one of `"xbar"`, `"s"`, `"R"`, `"p"`, `"np"`, `"u"`, or `"c"`, or if there does not exist a function with the name `zs.`*type*.
- `decision.int`: A numeric value specifying the number of standard errors of the summary statistics at which the cumulative sum is out of control.
- `se.shift`: The amount of shift to detect in the process, measured in standard errors of the summary statistics.
- `target`: A numeric value specifying the center of the process. By default, this is the `center` component of `object`.
- `std.dev`: A numeric value specifying the overall within-group standard deviation.
- `sizes`: A numeric vector specifying the sample sizes associated with each group of `newdata`.
- `labels`: A character vector of labels to associate with each group of `newdata`.
- `label.bounds`: A character vector of length two containing labels for the decision interval boundaries.
- `headstart`: A numeric value specifying the number of standard errors of the group summary statistics at which the cumulative sums should be reset. This argument is ignored if `reset=FALSE`.
- `reset`: A logical value indicating whether the cumulative sums should be reset after an out-of-control signal.
- `add.stats`: A logical value indicating whether statistics should be listed at the bottom of the chart.
- `chart.all`: A logical value indicating whether the cumulative sums of the `object$statistics` component should be charted along with the `new.statistics` component and the cumulative sums of the summary statistics for `newdata` (if given).
- `ylim.min`: A numeric vector of values to be included in the computation of the approximate *y*-axis limits for the control chart.

- `check.cl`: A logical value indicating whether the summary statistics beyond the control limits of the Shewhart chart should be highlighted on the chart, in addition to the decision boundary violations of the cumulative sums of the summary statistics.
- `highlight`: A list of graphical parameters for highlighting the points outside the decision boundaries, or beyond the Shewhart control limits.

The `type` argument is the same as that specified for the `shewhart` function; see the section Shewhart Charts for more details. If `type` is one of `"xbar"`, `"s"`, `"R"`, `"p"`, `"np"`, `"u"`, or `"c"`, there are built in functions for computing the group summary statistics and *z*-scores. If `type` is not one of these options, you must either create two functions with names `stats.`*type* and `zs.`*type*, or pass functions to the `type` and `z.scores` arguments explicitly.

The `type` and `z.scores` arguments to `cusum` are useful when charting is based on nonstandard summary statistics. For example, recall the `qccobj.med` object we create in the section Control Chart Objects, in which the estimate of the center of the process is based on the median and the standard deviation is based on the MAD. If the `type` component of `qccobj.med` is equal to `"med"` and you have defined the external functions `stats.med` and `zs.med`, simply type the following command to see the cusum chart:

```
> cusum(qccobj.med, newdata = newdata)
```

If you haven't defined appropriate `stats` and `zs` functions, or if you require functions other than those used to create `qccobj.med`, you must specify the names explicitly in the call to `cusum`. For example, to do a cusum chart of the group means of `newdata` with the center and standard deviation based on the median and mad, respectively, use `type="xbar"`:

```
> cusum(qccobj.med, newdata = newdata, type = "xbar")
```

With this argument, the `stats.xbar` function is used to compute the summary statistics, and the *z*-score function associated with xbar charts, `zs.xbar`, is used as well.

The `se.shift` argument is twice the reference value k in Equations (32.1) and (32.2). This corresponds roughly to the sensitivity of the cusum chart, in terms of detecting shifts in standard errors of the

summary statistics. By default, `se.shift=1`, which corresponds to a cusum chart being sensitive to one standard error shift and is equivalent to setting $k = 1/2$ in (32.1) and (32.2).

Usually, when an out-of-control signal is generated by a large (in absolute value) cumulative sum, a search is conducted and a cause is assigned and removed to correct the process. In this case, the cumulative sums are reset and the monitoring continues. The act of resetting the sums to something other than zero is called a *headstart*. With headstarts, you can produce a fast initial response (FIR) cusum chart. This is useful for quickly detecting a process that has not been fully corrected. When `reset=TRUE` the cumulative sums are reset to `headstart` each time one exceeds a decision boundary.

A group summary statistic greater than three standard errors from the target in a cusum chart is equivalent to that summary statistic being outside three-sigma Shewhart control limits. When `check.cl=TRUE`, summary statistics violating Shewhart control limits are flagged as well as large cumulative sums. If `object` is of class `"shewhart"`, it has a `cntrl.limits` component that is used to check for violating summary statistics. Otherwise, three-sigma Shewhart control limits (centered about `target`) are computed to check for violating summary statistics.

The `cusum` function returns an object that contains all the components of the reference `object`, plus the following additional components.

- `new.statistics`: A vector of group summary statistics for `newdata`.
- `new.sizes`: A vector of group sample sizes for `newdata`.
- `target`: The `target` argument if specified.
- `newdata.name`: A character string containing the name of the input data passed to the `newdata` argument.
- `cusum.upper`: A numeric vector of the upper cumulative sums.
- `cusum.lower`: A numeric vector of the lower cumulative sums.

EXTENSIONS TO SHEWHART CHARTS

To create "moving" charts, Spotfire S+ bases standard deviation estimates on a moving interval within which the standard deviation is computed. These estimates may be calculated from the range of values in a given interval, or from the standard deviation of values within the interval. You can produce moving Shewhart charts based on one-at-a-time data in the same way you produce basic Shewhart charts, but with the addition of two optional arguments to `qcc`.

- `sigma.span`: The number of data values used in each computation of sigma. This can be any integer larger than 1, but cannot be larger than the length of the data. The default value is 2.
- `moving.sigma`: A character string specifying the method used to compute the standard deviation in each interval. This must be one of `"range"` or `"s"`.

In addition to these arguments, the `type` argument may be one of the four options listed in Table 32.2: `"ma"`, `"ms"`, `"mR"`, and `"ewma"`. As an example, the following commands convert the `qcdata` matrix to a vector, and then generate a moving average chart with a moving window of three observations:

```
> mqcdata <- as.vector(qcdata)
> shewhart(qcc(mqcdata, type = "ma", sigma.span = 3))
```

For exponentially weighted moving average charts, specify the `type` argument as `"ewma"`, and provide a value from the closed interval [0.1,0.5] for the weight argument `wt`. The default for `wt` is 0.25. The sequence

$$k_i = w\bar{X}_i + (1-w)k_{i-1}$$

is plotted, where w is the `wt` value and $\bar{X}_i$ is the group mean or one-at-a-time data values associated with the ith group. For more details, see the help files for `stats.`*`type`* and `sd.`*`type`*, where *`type`* is one of the chart types.

PROCESS CAPABILITY

Process capability computations quantify the ability of a process to maintain its end product within the specification limits required by engineering. That is, capability compares the requirements of product engineering with the reality of the process. You can compute process capability with the `capability` function using an optional `qcc` object to define the process. The two values computed by `capability` are defined as follows:

$$C_p = \frac{USL - LSL}{6\sigma}$$

$$C_{pk} = \min\left\{\frac{USL - \mu}{6\sigma}, \frac{\mu - LSL}{6\sigma}\right\}$$

where USL is the upper specification limit, LSL is the lower specification limit, μ is the process center, and σ is the process standard deviation. The quantity $USL - LSL$ is referred to in some texts as the *allowable range.*

The arguments to `capability` are as follows.

- `qccobj`: An object returned by a call to the `qcc` function.
- `allowable.range`: The range between the upper and lower specification limits.
- `limits`: A vector of length two providing upper and lower specification limits.
- `center`: The process center.
- `std.dev`: The process standard deviation.
- `nsigmas`: The number of sigmas used to compute control limits. By default, `nsigmas=3`.

To compute C_p, provide a `qcc` object and the allowable range in the call to `capability`. If `limits` is not specified, then C_{pk} is set equal to C_p. If the `center` and `std.dev` arguments are not specified, the corresponding values from `qccobj` are used.

For example, to compute process capability for the `mqcdata` vector defined in the previous section, do the following:

```
> capability(qcc(mqcdata, type = "ma", sigma.span = 3),
+ allowable.range = 6, limits = c(8,14))

       cp       cpk
 0.927223 0.6443803
```

If you know the process parameters but do not have a `qcc` object, you can still compute process capability by providing the parameters directly as follows:

```
> capability(allowable.range = 6, limits = c(8,14),
+ std.dev = 1.09, center = 10.08)

        cp       cpk
 0.9174312 0.6360856
```

PROCESS MONITORING

In many manufacturing situations, processes are monitored in real time by production engineers and product managers. You can use Spotfire S+ for real-time monitoring with a few simple functions. We present examples below of two functions, `monitor` and `get.process`, which you can use to monitor a process data file and update a control chart as data comes in.

The basic idea is the following:

1. Create a file for accumulating the process data. In our example, this file is called **Process**.
2. Track the growth of **Process** with `get.process` and `monitor`, updating the control chart only when new data have been added to the file.

Suppose a typical line of the **Process** data file looks like the following:

Lot1 9.496215 8.718396 11.470395 9.671888 11.328800

Also, suppose you want to accumulate the data in a matrix. Then you could write the data-reading function, `get.process`, as follows:

```
> get.process <- function(file, skip = 0) {
+ data <- scan(file, what = list(names = "",0,0,0,0,0),
+               skip = skip)
+ nm <- data$names
+ data <- cbind(data[[2]],data[[3]],data[[4]],data[[5]],
+               data[[6]])
+ dimnames(data) <- list(nm, NULL)
+ return(data)
+ }
```

The configuration of the data fields are built into the `get.process` function. The first field in **Process** is a character label, and the remaining five fields are numeric data. The `skip` argument is added so that previously read data can be skipped when it is time to update the chart.

The `monitor` function keeps track of which data have already been read, and updates the control chart with any new data. An example of what `monitor` might look like is the following:

```
> monitor

function(file, qc.object, sleep.time = 5)
{
# define a subfunction to obtain the length of a file
   file.length <- function(file)
      length(count.fields(file, sep="\n"))
#
#
   old.length <- file.length(file)
   new.data <- get.process(file)
#
# put up initial chart
#
   qcc.shew <- shewhart(qc.object, new.data, add.stats=F)
   cat("to quit type CNTRL-CESC\n")
   repeat {
      new.length <- file.length(file)
      if(new.length > old.length) {
#
# new data have come in, we need to update the plot
#
         new.data <- get.process(file, skip = old.length)
         old.length <- new.length
         qcc.shew <- shewhart(qcc.shew, new.data,
            add.stats = F)
      }
      sleep(sleep.time)
   }
}
```

The statistics on the bottom of the chart have been turned off so that a number of charts can be efficiently placed within a single figure. The `monitor` function makes use of the fact that `shewhart` updates its return object; thus, the data that have just been added to **Process** are all you need to scan in each iteration.

Suppose now that `qcdata`, defined in the section Control Chart Objects, is coming in one row (or one lot) at a time. Place the first lot in the file **Process** to start the monitoring, and then run `monitor` as follows:

```
> monitor("Process", qccobj)

to quit type CNTRL-CESC
```

Spotfire S+ now monitors **Process** for a change in size. When one is detected, the new data are read in and the chart is updated. Figure 32.6 displays the results of 19 updates.

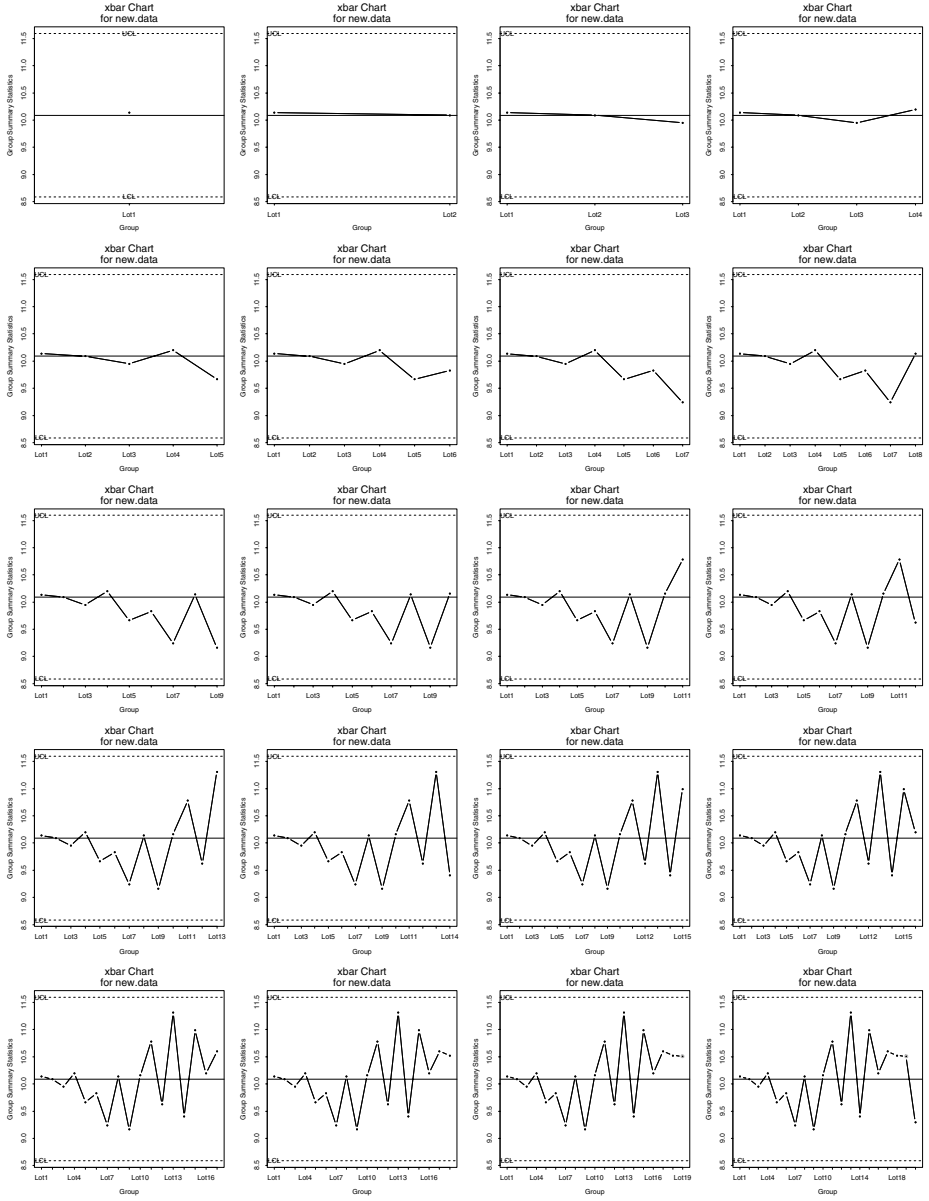

Figure 32.6: *A series of Shewhart charts of the data resulting from running* `monitor` *on a growing process data file.*

REFERENCES

Ryan, T.P. (1989). *Statistical Methods for Quality Improvement.* New York: John Wiley & Sons, Inc.

Wetherill, G.B. & D.W. Brown (1991). *Statistical Process Control.* New York: Chapman and Hall.

RESAMPLING TECHNIQUES: BOOTSTRAP AND JACKKNIFE 33

INTRODUCTION

In statistical analysis, the researcher is usually interested in obtaining not only a point estimate of a statistic but also an estimate of the variation in this point estimate, and a confidence interval for the true value of the parameter. For example, a researcher may calculate not only a sample mean, but also the standard error of the mean and a confidence interval for the mean.

Traditionally, researchers have relied on the central limit theorem and normal approximations to obtain standard errors and confidence intervals. These techniques are valid only if the statistic, or some known transformation of it, is asymptotically normally distributed. Hence, if the normality assumption does not hold, then the traditional methods should not be used to obtain confidence intervals.

A major motivation for the traditional reliance on normal-theory methods has been computational tractability. Now, with the availability of modern computing power, researchers need no longer rely on asymptotic theory to estimate the distribution of a statistic. Instead, they may use resampling methods which return inferential results for either normal or nonnormal distributions.

Resampling techniques such as the bootstrap and jackknife provide estimates of the standard error, confidence intervals, and distributions for any statistic. In the bootstrap, for example, B new samples, each of the same size as the observed data, are drawn with replacement from the observed data. The statistic is calculated for each new set of data, yielding a bootstrap distribution for the statistic. The fundamental assumption of bootstrapping is that the observed data are representative of the underlying population. By resampling observations from the observed data, the process of sampling observations from the population is mimicked. For more detailed descriptions of bootstrapping, see Efron and Tibshirani (1993) and Shao and Tu (1995).

Spotfire S+ includes a suite of functions for bootstrapping and jackknifing with the basic capabilities listed below.

- Given a vector, matrix, or data frame, create bootstrap or jackknife resamples of observations and use these to calculate resampling replicates of a specified statistic. The statistic may be a scalar, vector, or matrix and may be specified as an Spotfire S+ function or call.
- Produce informative summaries and plots for a resample object (`resamp`) produced by bootstrapping or jackknifing.
- Calculate empirical percentile and BCa confidence limits for a bootstrap object, and empirical percentiles for a jackknife object.
- Use jackknife after bootstrap to examine the influence of observations, and to estimate the standard error of a functional of the bootstrap distribution for a statistic.

A list of the bootstrapping and jackknifing functions is presented in Table 33.1.

Table 33.1: *Spotfire S+ bootstrapping and jackknifing functions.*

Function	Description
`bootstrap`	Main bootstrap function
`jackknife`	Main jackknife function
`summary.bootstrap`	Summary method for `bootstrap`

`print.resamp`
`plot.resamp`
`qqnorm.resamp`
`summary.resamp`
	Methods for `resamp` objects

`limits.emp` `limits.bca`	Calculate empirical and BCa percentiles
`jack.after.bootstrap`	Perform jackknife after bootstrap
`print.jack.after.bootstrap` `plot.jack.after.bootstrap`	Methods for jackknife after bootstrap object

Table 33.1: *Spotfire S+ bootstrapping and jackknifing functions. (Continued)*

Function	Description
`update.bootstrap`	Add more replicates to a `boot` object
`bootstats` `jackstats`	Called by `bootstrap` and `jackknife` to calculate resampling statistics

`samp.boot.mc`
`samp.boot.bal`
`samp.permute`
	Functions to generate resampling indices

CREATING A RESAMPLE OBJECT

There are two types of resample objects: bootstrap objects and jackknife objects. The main functions for generating these objects are `bootstrap` and `jackknife`. These functions call the more primitive functions `bootstats` and `jackstats`, which use the replicated parameter values and other information to calculate the bootstrap or jackknife statistics, and return an object of the appropriate class.

The Bootstrap

In bootstrap resampling, *B* new samples, each of the same size as the observed data, are drawn with replacement from the observed data. The statistic is first calculated using the observed data and then recalculated using each of the new samples, yielding a bootstrap distribution. The resulting replicates are used to calculate the bootstrap estimates of bias, mean, and standard error for the statistic.

Main Arguments

The main arguments in bootstrapping are the `data` (a vector, matrix, or data frame) and a `statistic`, which returns a scalar, vector, or matrix. This statistic may be a Spotfire S+ function or an unevaluated call (that is, any expression that one might type at the command line). Additional arguments to `statistic` may be passed as a list through `args.stat`.

You may specify the number *B* of resamples to draw. The default is 1000, which is the recommended minimum for estimating percentiles. Although a smaller *B* may be specified, 250 is recommended as a minimum for estimating standard errors.

Optional Arguments

- `seed`: Sets the random number seed. It may be a legal random number seed, or an integer between 0 and 1000.
- `group`: Specifies a stratifying variable. If supplied, then resampling is performed independently within each stratum. This argument can be used to bootstrap a two-sample or multiple-sample statistic. Note that the bootstrap estimates are not adjusted based on stratifying.
- `sampler`: Generates resampling indices. The default function `samp.boot.mc` performs standard Monte Carlo bootstrapping of observations. The `samp.boot.bal` function performs

balanced bootstrapping. In some cases, the `bootstrap` function may be used to perform a permutation test by using `samp.permute` with an appropriately defined statistic.

- `block.size`: Controls computational details of the bootstrapping. By default, this is set to `min(B,100)` and the bootstrapping is performed using one large `lapply`. If the sample size *n* and number *B* of resamples are large, then this default may be slower than the alternative of performing a `for` loop over smaller blocks of observations. The `block.size` argument specifies the size of each block over which a `for` is applied. For example, if `n=1000` and `B=1000`, then it may be preferable to do 10 loops with `block.size=100` rather than a single `lapply`.

Note

Pressing ESC during the looping interrupts the process and saves the replicates computed so far.

- `block.size`: Controls computational details of the bootstrapping. For efficiency, the samples are drawn in blocks of size `block.size` and `lapply` is used over each block to evaluate the statistic. The drawing of blocks is embedded within a `for` loop to draw a total of `B` samples. When `n` is small it is most efficient to perform a single `lapply` so that `block.size=B`. When `n` is large, it is more efficient to use a smaller `block.size`. For example, if `n=1000` and `B=1000`, then it may be preferable to do 10 loops with `block.size=100` rather than a single `lapply`. By default the `block.size` is set to `min(B,100)`.
- `assign.frame1`: Logical flag indicating whether the resampled data should be assigned to frame 1 before evaluating the statistic. This may be necessary if the statistic is reevaluating the call of a model object. If all bootstrap estimates are identical, try setting `assign.frame1=T`. Note that this option slows down the algorithm.
- `trace`: Logical flag indicating whether to print a message indicating which set of replicates is currently being drawn.

- `save.indices`: Logical flag indicating whether to save the matrix of resampling indices. By default, the value of the random number seed used is saved, and the sampler used is specified in the call, which is enough information to reproduce the resampling indices in later analyses. The matrix of resampling indices may be saved as part of the object by setting `save.indices=T`. This matrix has dimension `n` x `B`.

Additional arguments are described in the help file for the `bootstrap` function.

Other Functions

The `bootstrap` function calls `bootstats` to calculate bootstrap statistics. If you specify the required information, then `bootstats` may be called directly to produce a bootstrap object. The main caveat is that `limits.bca` and `jack.after.bootstrap` look at the `call` component of the object, so the function calling `bootstats` should pass along an appropriate call if these functions are to be used on the resulting object.

Components of the Object

A `bootstrap` object has components `call`, `observed`, `replicates`, `estimate`, `B`, `n`, `dim.obs`, `group`, `seed.start`, and `seed.end`. The `observed` component contains the observed parameter values calculated using the original data. The `estimate` data frame contains bootstrap estimates of bias, mean, and standard error. The `replicates` are the bootstrap replicates of the parameters. The `call` component, starting random number seed `seed.start`, ending random number seed `seed.end`, and `group` are stored for future reference, as are the number `B` of replicates and the sample size `n`. If `statistic` returns a matrix, then its dimension is stored as `dim.obs` for use in the layout of plots. In many cases, `dim.obs` and `group` are `NULL`.

The Jackknife

In jackknife resampling, a statistic is calculated for the `n` possible samples of size `n-1`, each with one observation left out. The default sample size is `n-1`, but more than one observation may be removed using the `group.size` argument (see below). Jackknife estimates of bias, mean, and standard error are available and are calculated differently than the equivalent bootstrap statistics.

Arguments

The `jackknife` function takes the arguments `data`, `statistic`, `args.stat`, and `assign.frame1`, which have the same meanings as for `bootstrap`.

The `seed` argument may be used to specify a seed for randomization done by the statistic, and for random assignment of observations to groups if `group.size` is not equal to one. It may be a legal random number seed, or an integer between 0 and 1000.

The `group.size` argument may be used to specify the removal of more than one point in each sample. This argument is useful in partial jackknifing for calculating the acceleration when forming BCa percentiles. It forms `floor(n/group.size)` replicates, each missing `group.size` observations. These replicates are treated as a jackknife sample of size `floor(n/group.size)`.

Other Functions

The `jackstats` function calculates the jackknife statistics.

METHODS FOR RESAMPLE OBJECTS

The print Method

The `print` method for a resample object, `print.resamp`, prints out the call, the number of resamples used, and a table giving the values of the statistic for the original data and resampling estimates of bias, mean, and standard error for the statistic.

The summary Method

The `summary` method for a resample object prints out the same information as `print.resamp`, followed by the empirical percentiles of the replicates. The `summary` of a `bootstrap` object also calculates BCa percentiles. If the statistic is vector-valued, a correlation matrix for the components of the vector is also printed. The optional `probs` argument specifies probabilities at which the empirical quantiles are calculated.

Additional arguments useful in `limits.bca` may be specified with `summary.bootstrap`. These arguments include `z0`, `acceleration`, and `group.size`. By default, a `group.size` of `floor(n/20)` is used in `limits.bca` for reasons of speed. To do a full jackknifing when estimating `acceleration`, specify `group.size=1`.

The plot Method

The `plot` method for a resample object produces plots of the distributions of the statistics. For each statistic, a histogram of the replicates is displayed with an overlaid smooth density estimate. A solid vertical line is plotted at the observed parameter value, and a dashed vertical line is plotted at the mean of the replicates. The distance between the dotted line and the solid line is the estimated bias. The shape of the distribution may be examined to assess issues such as skewness of the distribution of the statistic.

You may specify `plot` with a `bandwidth.func` argument to calculate the bandwidth of the density estimate. By default, the normal reference density estimate is used. In addition, you may specify `plot` with an `nclass.func` argument to calculate the number of classes in the histogram. By default, the Freedman and Diaconis rule is used. Arguments may also be passed to `histogram` through the ellipsis (`...`).

Plots are displayed in a grid (`grid=T`) by default. Use `nrow` to specify the number of rows in the grid. If the statistic is a matrix, then by default the plots are arranged in the same order as the terms appear in the matrix.

Normal Quantile-Quantile Plots

The `qqnorm` method for a resample object produces a plot with the same layout as in `plot.resamp`, but with each plot containing a normal quantile-quantile plot for the relevant statistic. If the argument `lines=T`, as is the default, then a `qqline` is also added to each plot.

This plot is used to assess the normality of the distribution of each statistic. If the points fall on a straight line, the empirical distribution of the replicates is similar to that of a normal random variate.

PERCENTILE ESTIMATES

Two types of percentile estimates are supported: empirical percentiles, and bias-corrected and adjusted (BCa) percentiles. These are calculated by `limits.emp` and `limits.bca`, respectively. The empirical percentiles are available for bootstrap and jackknife objects, while BCa percentiles are available only for bootstrap objects. The empirical percentiles are easy to calculate, but may not be accurate unless the sample size is very large. The BCa percentiles require more computation but are more accurate. For either type of percentile, using at least 1000 replications is recommended for accurate estimation. The `probs` argument to the `limits.emp` and `limits.bca` functions specifies which percentiles are computed.

Empirical Percentiles

The empirical percentiles are simply the percentiles of the empirical distribution of the replicates. Linear interpolation is used if necessary to obtain the specified percentiles.

BCa Percentiles

The BCa method transforms the specified `prob` values to determine which percentiles of the empirical distribution most accurately estimate the percentiles of interest. The percentiles of the empirical distribution corresponding to these values are then returned.

To estimate the BCa percentiles, the bias correction (denoted z_0) and the acceleration must be calculated. If these values are not specified (and they usually are not), the bias correction is obtained from the replicates and the acceleration is obtained using jackknifing. Note that rather than doing a complete delete-1 jackknife, the data are broken into groups of size `group.size` and the groups are jackknifed. If `group.size` is not specified, it is calculated as `floor(n/20)`, which yields roughly 20 jackknife replicates depending on the magnitude of `n`.

To return the values of `z0`, `acceleration`, and the empirical percentile level for each BCa percentile, set `detail=T`.

JACKKNIFE AFTER BOOTSTRAP

Jackknife after bootstrap is a technique for obtaining estimates of the variation in functionals of a bootstrap distribution, such as the bias or standard error of a statistic, without performing a second level of bootstrapping. It also provides information on the influence of each observation on the functionals. See Efron and Tibshirani (pp. 275-280) for details on this procedure.

Simulation studies have shown that, in general, jackknife after bootstrap standard error estimates tend to be too large. A technique called *weighted jackknife after bootstrap* may resolve some of these difficulties. This technique is currently under investigation and has not yet been implemented.

The Jackknife After Bootstrap Object

The jackknife after bootstrap object has components `call`, `functional`, `rel.influence`, `large.rel.influence`, `values.functional`, `dim.obs`, and `threshold`. The value of the functional for the bootstrapped parameter replicates, and for the jackknife after bootstrap estimates of standard errors, is given as the `functional` data frame. The value of the functional over the samples with each point removed is given in `values.functional`. Normalized versions of these values are given in `rel.influence`. The list `large.rel.influence` gives the relative influence values for points with absolute relative influences in excess of `tolerance`. The `call` is the call to `jack.after.bootstrap`. The `dim.obs` is the corresponding component of the `bootstrap` object. The jackknife after bootstrap object is of class `"jack.after.bootstrap"`.

The print Method

The `print` method for a `jack.after.bootstrap` object displays the `call`, the description of the functional under consideration, the data frame of functional values and standard errors, and the list of large relative influences.

The plot Method

The `plot` method for a `jack.after.bootstrap` object produces a plot for each parameter, indicating the relative influence of each observation. Values greater than a specified tolerance (default = 2) are flagged as being particularly influential.

EXAMPLES

This section describes three examples. The first is a bootstrap of a variance and discusses the output and basic plots associated with the bootstrap object. The second example resamples a correlation coefficient, and details the application of bootstrap, jackknife after bootstrap, and jackknife tools. The third example shows how to test linear regression coefficients using the bootstrap and jackknife after bootstrap.

Resampling the Variance

This example uses data from the `swiss.x` matrix, which contains socioeconomic indicators for the provinces of Switzerland in 1888. More particularly, this example resamples the variance of the `Education` variable, the percent of the population whose education is beyond primary school.

First, `Education` is separated from the `swiss.x` matrix.

```
> Education <- swiss.x[,3]
> Education

 [1] 12  9  5  7 15  7  7  8  7 13  6 12  7 12  5  2  8 28 20
[20]  9 10  3 12  6  1  8  3 10 19  8  2  6  2  6  3  9  3 13
[39] 12 11 13 32  7  7 53 29 29
```

The `bootstrap` function is used to draw resamples and construct a bootstrap object.

```
> boot.obj1 <- bootstrap(Education, var, B = 1000, seed = 0)

Forming replications  1  to  100
Forming replications  101  to  200
Forming replications  201  to  300
Forming replications  301  to  400
Forming replications  401  to  500
Forming replications  501  to  600
Forming replications  601  to  700
Forming replications  701  to  800
Forming replications  801  to  900
Forming replications  901  to  1000
```

To prevent the preceding messages from being displayed, set `trace=F`.

Note

All examples in this section use `B=1000`, the number of resamples recommended for accurate estimation of percentiles. Users who want to replicate the examples might use a lower number of resamples (say, `B=250`) to speed up estimation. Note, however, that results will differ slightly from those shown here.

Printing the object displays the call used to construct it, the number of replications used, and summary statistics for the parameter. The summary statistics are the observed value of the parameter, the mean of the parameter estimate replicates, and bootstrap estimates of bias and standard error.

```
> boot.obj1

Call:
bootstrap(data = Education, statistic = var, B = 1000,
seed = 0)

Number of Replications: 1000

Summary Statistics:
    Observed   Bias  Mean    SE
var    92.46 -3.362 89.09 38.67
```

A more complete summary of the bootstrap object, obtained via the `summary` function, includes empirical and BCa percentiles for the statistic. The BCa percentiles, for example, show that the 95% confidence interval for the `Education` variance has endpoints 45.34 and 221.2.

```
> summary(boot.obj1)

Call:
bootstrap(data = Education, statistic = var, B = 1000,
seed = 0)

Number of Replications: 1000
```

```
Summary Statistics:
    Observed   Bias  Mean    SE
var    92.46 -3.362 89.09 38.67

Empirical Percentiles:
    2.5%    5%   95% 97.5%
var 32.9 36.17 163.9 177.1

BCa Percentiles:
     2.5%    5%   95% 97.5%
var 45.34 51.44 211.6 221.2
```

Empirical and BCa percentiles may also be obtained separately using the `limits.emp` and `limits.bca` functions, respectively.

```
> limits.emp(boot.obj1)

        2.5%       5%      95%    97.5%
var 32.89544 36.16716 163.8941 177.1408

> limits.bca(boot.obj1)

        2.5%      5%      95%    97.5%
var 45.33665 51.4373 211.6284 221.1731
```

Plotting the bootstrap object provides a histogram of the replicated variances along with a smooth density estimate (Figure 33.1). The solid line indicates the observed parameter value, and the dotted line indicates the mean of the replicates. The difference between these two values is the bootstrap estimate of bias.

```
> plot(boot.obj1)
```

The histogram in Figure 33.1 shows that the distribution of replicated variances is highly skewed. A normal quantile-quantile plot can be used to further assess deviation from the normal distribution. Figure 33.2 suggests that both tails of the distribution of replicated variances deviate from the normal distribution. Thus there is evidence that bootstrapping is a better approach than normal-based methods.

```
> qqnorm(boot.obj1)
```

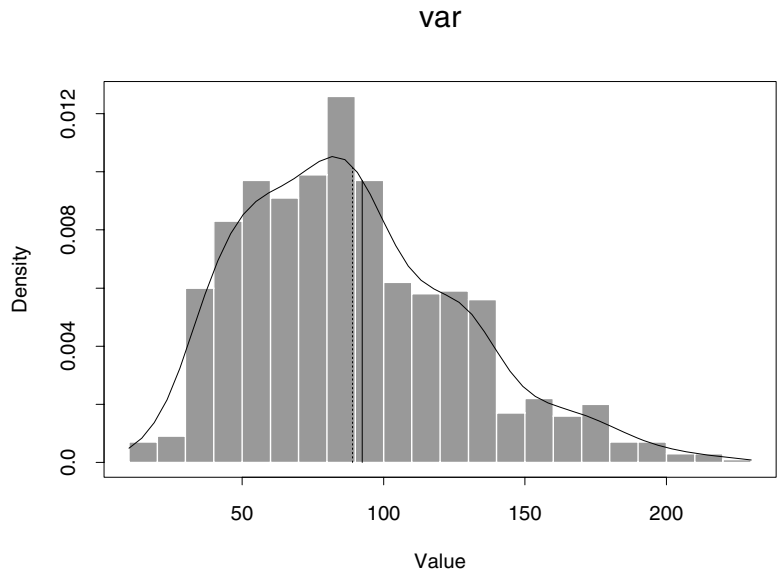

Figure 33.1: *Histogram of replicated variances.*

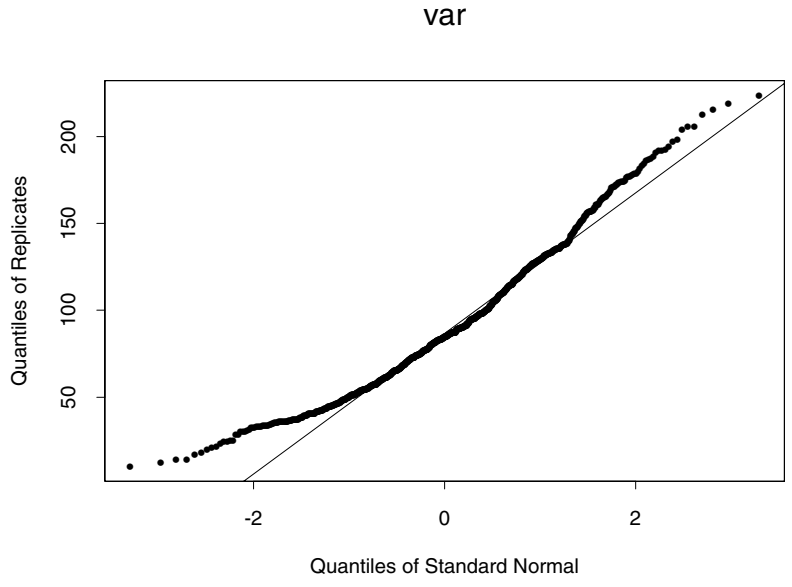

Figure 33.2: *Normal qq-plot of replicated variances.*

Resampling the Correlation Coefficient

This example uses the law school data from Efron and Tibshirani (p. 9). Starting with 82 American law schools participating in a study of admission practices, they constructed a random sample of 15 schools. Efron and Tibshirani then examined the correlation between LSAT score and GPA for the 1973 entering classes at these schools (p. 49).

Traditionally, Fisher's transformation would be used to transform the correlation coefficient into a normally distributed variable on which normal-based inference would be used. This example uses resampling to obtain inferential quantities instead of employing Fisher's transformation.

First, the data are entered into Spotfire S+ and stored as a data frame.

```
> school <- 1:15
> lsat <- c(576, 635, 558, 578, 666, 580, 555, 661, 651,
+ 605, 653, 575, 545, 572, 594)

> gpa <- c(3.39, 3.30, 2.81, 3.03, 3.44, 3.07, 3.00, 3.43,
+ 3.36, 3.13, 3.12, 2.74, 2.76, 2.88, 2.96)

> law.data <- data.frame(School = school, LSAT = lsat,
+ GPA = gpa)
```

Next, the `bootstrap` function is used, and the summary of the resulting object displayed.

```
> boot.obj2 <- bootstrap(law.data, cor(LSAT, GPA),
+ B = 1000, seed = 0, trace = F)

> summary(boot.obj2)

Call:
bootstrap(data = law.data, statistic = cor(LSAT, GPA),
B = 1000, seed = 0, trace = F)

Number of Replications: 1000

Summary Statistics:
      Observed      Bias   Mean     SE
Param   0.7764 -0.008768 0.7676 0.1322
```

```
Empirical Percentiles:
        2.5%    5%    95%  97.5%
Param 0.4673 0.523 0.9432 0.9593

BCa Percentiles:
        2.5%    5%    95%  97.5%
Param 0.3443 0.453 0.9255 0.9384
```

The bootstrap object is plotted to obtain a histogram of the replicated correlation values along with a smooth density estimate (Figure 33.3). The distribution is clearly skewed.

```
> plot(boot.obj2)
```

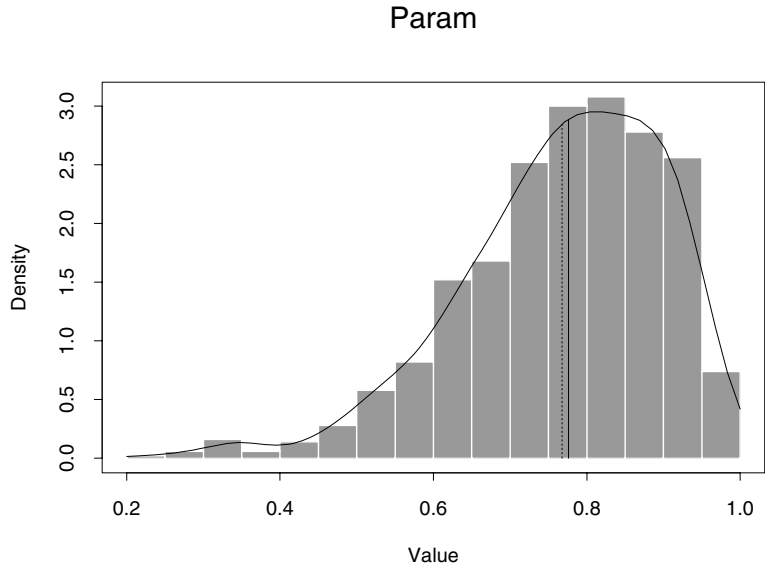

Figure 33.3: *Histogram of replicated correlations.*

Another tool available for exploring the bootstrap object is the jackknife after bootstrap (Efron and Tibshirani, p. 275). This technique provides standard error estimates for functionals of the bootstrap distribution, and influence measures for each observation. By default, the functional is the mean of the distribution. In this case, the standard error of the functional is the standard error of the mean, and the influence indicates the influence of each observation on the mean. Jackknife after bootstrap is commonly used to get standard error estimates for the bootstrap estimate of standard error.

```
> jab.obj2 <- jack.after.bootstrap(boot.obj2)
> jab.obj2

Call:
jack.after.bootstrap(boot.obj = boot.obj2, functional =
mean)

Functional Under Consideration:
mean

Functional of Bootstrap Distribution of Parameters:
        Func SE.Func
Param 0.7676  0.1432

Observations with Large Influence on Functional:
$Param:
   Param
1 -3.025
```

Plotting the `jack.after.bootstrap` object provides an influence plot similar to a Cook's distance plot (Figure 33.4). Observations with absolute relative influence greater than 2 are considered particularly influential.

```
> plot(jab.obj2)
```

The jackknife after bootstrap identifies observation 1 as being particularly influential. A plot of GPA versus LSAT with this observation plotted as a triangle shows that this point is indeed an outlying observation (Figure 33.5).

```
> plot(lsat[-1], gpa[-1], xlab = "LSAT", ylab = "GPA")
> points(lsat[1], gpa[1], pch = 2)
```

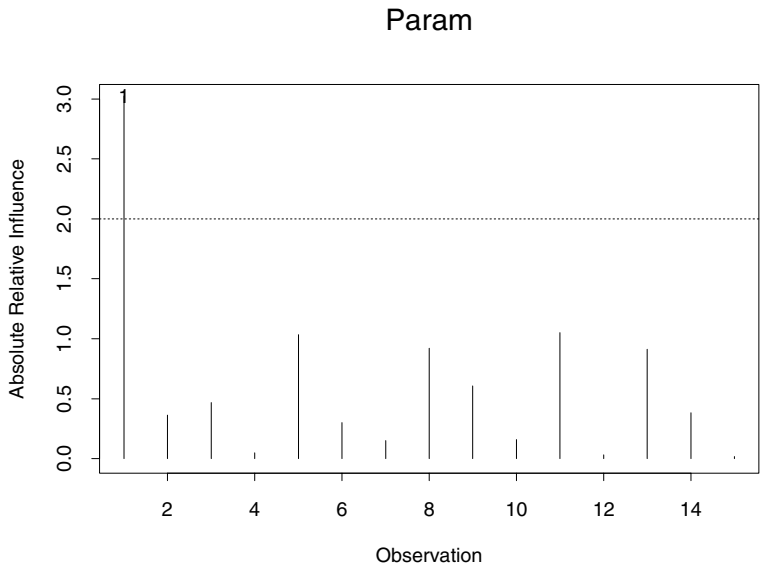

Figure 33.4: *Influence plot for correlation.*

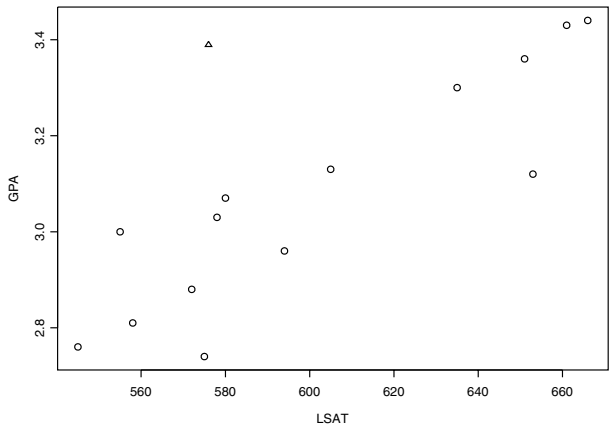

Figure 33.5: *GPA versus LSAT.*

Jackknife summary statistics for the correlation may be obtained also.

```
> jackknife(law.data, cor(LSAT, GPA))
```

```
Call:
jackknife(data = law.data, statistic = cor(LSAT, GPA))

Number of Replications: 15

Summary Statistics:
      Observed      Bias   Mean     SE
Param   0.7764 -0.006473 0.7759 0.1425
```

Resampling Regression Coefficients

The last example shows how to test linear regression coefficients, and uses the bootstrap to obtain standard error estimates and confidence intervals. The data are from operation of a plant for the oxidation of ammonia to nitric acid, measured on 21 consecutive days. See the Spotfire S+ help file for `stack` for details.

First, the `stack.loss` vector and `stack.x` matrix are combined into a data frame.

```
> stack <- data.frame(stack.loss, stack.x)
> names(stack)

[1] "stack.loss" "Air.Flow"   "Water.Temp" "Acid.Conc."
```

The `bootstrap` function resamples the vector of linear regression coefficients from the model of `stack.loss` regressed on `Air.Flow`, `Water.Temp`, and `Acid.Conc`.

```
> boot.obj3 <- bootstrap(stack,
+ coef(lm(stack.loss ~ Air.Flow + Water.Temp + Acid.Conc.,
+ stack)), B = 1000, seed = 0, trace = F)

> boot.obj3

Call:
bootstrap(data = stack, statistic = coef(lm(stack.loss ~
Air.Flow + Water.Temp + Acid.Conc., stack)), B = 1000,
seed = 0, trace = F)

Number of Replications: 1000

Summary Statistics:
            Observed      Bias     Mean     SE
(Intercept) -39.9197  0.829215 -39.0905 8.8239
   Air.Flow   0.7156  0.004886   0.7205 0.1749
```

```
 Water.Temp   1.2953 -0.031415   1.2639 0.4753
 Acid.Conc.  -0.1521 -0.005164  -0.1573 0.1180
```

The summary for a vector statistic includes the correlation matrix for the replicate values. Based on the 95% confidence limits, for either the empirical or the BCa percentiles, all coefficients except the `Acid.Conc.` coefficient are significantly different from zero.

```
> summary(boot.obj3)

Call:
bootstrap(data = stack, statistic = coef(lm(stack.loss ~
Air.Flow + Water.Temp + Acid.Conc., stack)), B = 1000,
seed = 0, trace = F)
Number of Replications: 1000

Summary Statistics:
            Observed      Bias     Mean     SE
(Intercept) -39.9197  0.829215 -39.0905 8.8239
   Air.Flow   0.7156  0.004886   0.7205 0.1749
 Water.Temp   1.2953 -0.031415   1.2639 0.4753
 Acid.Conc.  -0.1521 -0.005164  -0.1573 0.1180

Empirical Percentiles:
                2.5%       5%      95%     97.5%
(Intercept) -55.4846 -52.7583 -23.4913 -17.84522
   Air.Flow   0.3844   0.4454   1.0136   1.05255
 Water.Temp   0.3913   0.4768   2.0544   2.23920
 Acid.Conc.  -0.4181  -0.3604   0.0209   0.06103

BCa Percentiles:
                2.5%       5%        95%     97.5%
(Intercept) -58.8427 -54.3320 -25.385390 -21.48317
   Air.Flow   0.3197   0.3897   0.987308   1.01691
 Water.Temp   0.4977   0.5811   2.278439   2.46017
 Acid.Conc.  -0.4250  -0.3743   0.008729   0.04447
```

```
Correlation of Replicates:
            (Intercept) Air.Flow Water.Temp Acid.Conc.
(Intercept)     1.00000  -0.1376    0.03551    -0.7848
   Air.Flow    -0.13760   1.0000   -0.79387    -0.1096
 Water.Temp     0.03551  -0.7939    1.00000    -0.2007
 Acid.Conc.    -0.78483  -0.1096   -0.20067     1.0000
```

The plot function provides histograms of the replicated regression coefficients (Figure 33.6). Skewness is particularly evident in the `Acid.Conc.` coefficients.

```
> plot(boot.obj3)
```

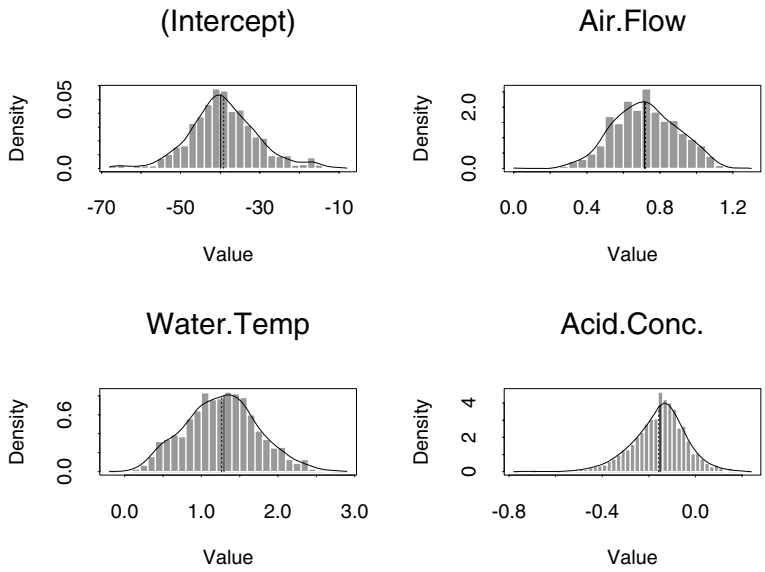

Figure 33.6: *Histograms of replicated regression coefficients.*

Next, the jackknife after bootstrap is used to assess the accuracy of the standard error estimates, and the influence of each observation on these estimates.

```
> jab.obj3 <- jack.after.bootstrap(boot.obj3, "SE")
```

```
> jab.obj3

Call:
jack.after.bootstrap(boot.obj = boot.obj3, functional =
"SE")

Functional Under Consideration:
[1] "SE"

Functional of Bootstrap Distribution of Parameters:
              Func SE.Func
(Intercept) 8.8239 3.67775
   Air.Flow 0.1749 0.06149
 Water.Temp 0.4753 0.17850
 Acid.Conc. 0.1180 0.05395

Observations with Large Influence on Functional:

$"(Intercept)":
   (Intercept)
21       2.863

$Air.Flow:
   Air.Flow
21    3.672

$Water.Temp:
   Water.Temp
21      3.214

$Acid.Conc.:
   Acid.Conc.
14     -2.184
21      2.589
```

The jackknife after bootstrap and the corresponding influence plot (Figure 33.7) suggest that points 14 and 21 are particularly influential.

```
> plot(jab.obj3)
```

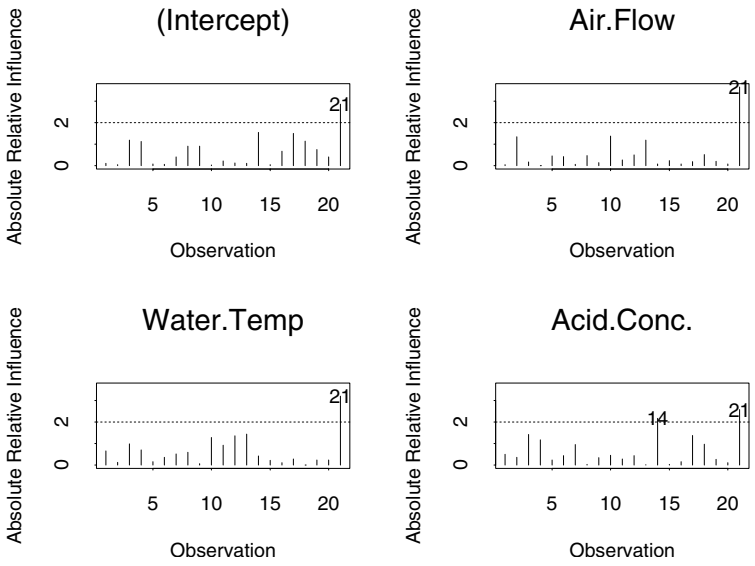

Figure 33.7: *Influence plots for regression coefficients.*

REFERENCES

Efron, B. & Tibshirani, R.J. (1993). *An Introduction to the Bootstrap*. San Francisco: Chapman & Hall.

Shao, J. & Tu, D. (1995). *The Jackknife and Bootstrap*. New York: Springer-Verlag.

MATHEMATICAL COMPUTING IN SPOTFIRE S+ 34

INTRODUCTION

Spotfire S+ was designed for data analysis, so it is rich in quantitative methods. Many of these methods, while designed for particular data analysis tasks, have been implemented as general mathematical tools. These tools can be applied to a wide variety of numerical applications. This chapter is a brief survey of mathematical computing in Spotfire S+.

In this chapter, we assume a basic familiarity with the operation of the command line. For the most part, however, this chapter is self-contained and can be read independently of the other chapters in this manual.

ARITHMETIC OPERATIONS

You perform basic arithmetic in Spotfire S+ as you would with a calculator, using the operators +, -, *, and /:

```
> 2 + 2
[1] 4

> 9 - 3
[1] 6

> 3 * 8
[1] 24

> 17 / 4
[1] 4.25
```

Use the operator ^ for exponentiation, including root extraction:

```
> 3 ^ 2
[1] 9

> 7 ^ (1 / 3)
[1] 1.912931
```

Operators have their usual precedence (powers, multiplication/division, addition/subtraction), and parentheses can be used (as in the previous example) to group calculations. Two other operators provide integer quotients and remainders. The integer divide operator, %/%, returns the integer quotient q and the modulo operator, %%, returns the remainder r of two numbers y and x, so that $y = qx + r$:

```
> 24.5 %/% 3.2
[1] 7

> 24.5 %% 3.2
[1] 2.1

> 7 * 3.2 + 2.1
[1] 24.5
```

The `abs` function returns the absolute value of a number:

```
> abs(-4.5)
[1] 4.5
```

The greatest-integer function $\lfloor x \rfloor$ is obtained using `floor`:

```
> floor(2.3)
[1] 2
```

Similarly, the "next integer" $\lceil x \rceil$ is obtained using `ceiling`:

```
> ceiling(2.3)
[1] 3
```

A *vector* in Spotfire S+ is an ordered set of values. Simple numeric vectors can be created with the `c` function or the sequence operator (`:`):

```
> x <- c(3, 1, 7)
> x

[1] 3 1 7

> w <- 1:6
> w

[1] 1 2 3 4 5 6
```

A *matrix*, in Spotfire S+, is simply a vector with a specified number of rows and columns, that is, an ordered set of data in a rectangular array. You can create matrices with the `matrix` command:

```
> A <- matrix(c(19, 8, 11, 2, 18, 17, 15, 19, 10), nrow = 3)

     [,1] [,2] [,3]
[1,]   19    2   15
[2,]    8   18   19
[3,]   11   17   10
```

You can also build matrices from existing vectors using `rbind` (which assigns vectors to the *rows* of the matrix) or `cbind` (which assigns vectors to the *columns* of the matrix):

```
> m <- c(14, 13, 10)
> n <- c(10, 11, 15)
> o <- c(19, 3, 15)
```

```
> B <- cbind(m, n, o)
> B

      m  n  o
[1,] 14 10 19
[2,] 13 11  3
[3,] 10 15 15
```

Most calculations on vectors or matrices are carried out *element by element*, so, for example, if $X = \{x_{ij}\}$ and $Y = \{y_{ij}\}$, we have $X*Y = \{x_{ij}y_{ij}\}$. Multiplying `A` times `B` with the standard `*` operator yields the following:

```
> A * B

      m    n    o
[1,] 266   20  285
[2,] 104  198   57
[3,] 110  255  150
```

For matrices, these element by element operations require that the matrices have the same dimension; that is, the same number of rows and the same number of columns, so that the matrices are *conformable for addition*. For vectors, if one vector is shorter than the other, the shorter vector is repeated cyclically to match the length of the longer vector:

```
> x + w
[1]  4  3 10  7  6 13
```

Mathematical operations on combinations of vectors and matrices are permitted, but may have unexpected results. For example, suppose you define the matrix `E` as follows:

```
> E <- matrix(1:4, nrow = 2)
```

Dividing by the previously defined vectors `x` and `w` yields the following results:

```
> E/w

[1] 1.0000000 1.0000000 1.0000000 1.0000000 0.2000000
[6] 0.3333333
Warning messages:
  Length of longer object is not a multiple of the
```

```
 length of the shorter object in: E/w

> E/x

          [,1]      [,2]
[1,] 0.3333333 0.4285714
[2,] 2.0000000 1.3333333
Warning messages:
  Length of longer object is not a multiple of the
  length of the shorter object in: E/x
```

Spotfire S+ returns an object with the attributes of the longer object in the calculation. Since `length(E) < length(w)`, `E/w` returned an object matching the attributes of `w`, namely a vector of length 6. On the other hand, since `length(E) > length(x)`, `E/x` returned an object matching the attributes of `E`, namely, a matrix of length 4 with `dim = c(2,2)`.

To perform matrix multiplication, use the *matrix multiplication operator* `%*%`

```
> A %*% B

       m    n    o
[1,]  442  437  592
[2,]  536  563  491
[3,]  475  447  410
```

The two matrices must be *conformable for multiplication*, that is, the number of columns of `A` must be the same as the number of rows of `B`.

Using the matrix multiplication operator on two equal length vectors yields the *vector dot product*:

```
> z <- c(1, 0, 3, 4, 8)
> y <- c(2, 9, 3, 2, 7)
> z %*% y

     [,1]
[1,]   75
```

COMPLEX ARITHMETIC

In addition to the ordinary operators described in the section Arithmetic Operations, five special operators are provided for manipulating complex numbers.

`Re` and `Im` are used to extract the real and imaginary parts, respectively, from a complex number. `Mod` and `Arg` return the *modulus* and *argument* for the polar representation of the complex number. `Conj` returns the complex conjugate of the complex number.

When you graph a vector of complex numbers with `plot`, the real parts are graphed along the *x*-axis and the imaginary parts are graphed along the *y*-axis.

ELEMENTARY FUNCTIONS

The elementary functions included in Spotfire S+ are listed in Table 34.1.

Table 34.1: *Elementary Functions in Spotfire S+.*

Name	Operation
`sqrt`	Square root
`abs`	Absolute value
`sin, cos, tan`	Trigonometric functions (radians)
`asin, acos, atan`	Inverse trigonometric functions (radians)
`sinh, cosh, tanh`	Hyperbolic trigonometric functions (radians)
`asinh, acosh, atanh`	Inverse hyperbolic trigonometric functions (radians)
`exp, log`	Exponential and natural logarithm (base e)
`log10`	Common logarithm (base 10)
`logb`	Logarithm for bases other than e and 10
`gamma, lgamma`	Gamma function and its natural logarithm

Each function acts *element-by-element* on its argument:

```
> J

     [,1] [,2] [,3] [,4]
[1,]   12   15    6   10
[2,]    2    9    2    7
[3,]   19   14   11   19
```

```
> sqrt(J)

         [,1]     [,2]     [,3]     [,4]
[1,] 3.464102 3.872983 2.449490 3.162278
[2,] 1.414214 3.000000 1.414214 2.645751
[3,] 4.358899 3.741657 3.316625 4.358899

> tan(J)

           [,1]       [,2]         [,3]      [,4]
[1,] -0.6358599 -0.8559934   -0.2910062 0.6483608
[2,] -2.1850399 -0.4523157   -2.1850399 0.8714480
[3,]  0.1515895  7.2446066 -225.9508465 0.1515895
```

You can use `logb` to compute logarithms of any base with the optional argument `base`. For example, to compute $\log_2 7$:

```
> logb(7, base = 2)

[1] 2.807355
```

VECTOR AND MATRIX COMPUTATIONS

The p-norm of a vector $\boldsymbol{x}$ of length n is defined as:

$$[\boldsymbol{x}_1^p + \boldsymbol{x}_2^p + \cdots + \boldsymbol{x}_n^p]^{1/p}$$

for $p \geq 1$. To obtain the p-norm of a vector in Spotfire S+, use the `vecnorm` function (by default, $p = 2$):

```
> vecnorm(1:2)
[1] 2.236068

> ( sum( (1:2) ^ 2) ) ^ (1/2)
[1] 2.236068
```

The `vecnorm` function works with both real and complex vectors:

```
> vecnorm(1+2i)
[1] 2.236068
```

You can specify the type of norm desired with the `p` argument. Possible values include real numbers greater than or equal to 1, `Inf`, and the character strings `"euclidean"` or `"maximum"`:

```
> vecnorm(1:2, p = 1)
[1] 3

> vecnorm(1:2, p = "maximum")
[1] 2

> vecnorm(1:2, p = Inf)
[1] 2
```

To obtain the transpose of a matrix, use the `t` function:

```
> J

     [,1] [,2] [,3] [,4]
[1,]   12   15    6   10
[2,]    2    9    2    7
[3,]   19   14   11   19
```

```
> t(J)

     [,1] [,2] [,3]
[1,]   12    2   19
[2,]   15    9   14
[3,]    6    2   11
[4,]   10    7   19
```

You can obtain the diagonal of a matrix with the `diag` function:

```
> diag(J)
[1] 12 9 11
```

You can also use `diag` to construct diagonal matrices:

```
> x <- c(3, 1, 7)
> diag(x)

     [,1] [,2] [,3]
[1,]    3    0    0
[2,]    0    1    0
[3,]    0    0    7
```

To obtain the *trace* of a square matrix, use `sum` with `diag`, as follows:

```
> sum(diag(A))
[1] 47
```

Identity Matrices

To generate identity matrices in Spotfire S+, use `diag` with an integer argument representing the rank *n* as follows:

```
> diag(n)
```

For example, the rank 4 identity matrix is created as follows:

```
> diag(4)

     [,1] [,2] [,3] [,4]
[1,]    1    0    0    0
[2,]    0    1    0    0
[3,]    0    0    1    0
[4,]    0    0    0    1
```

Determinants

There is no built-in Spotfire S+ function to calculate determinants. However, the following one-line function can be used to calculate determinants for *real*-valued matrices:

```
> det <- function(x) prod(eigen(x)$values)
```

The `eigen` function is discussed in the section Eigenvalues and Eigenvectors.

Kronecker Products

A *Kronecker product* of two matrices $\boldsymbol{A}_{p \times q}$ and $\boldsymbol{B}_{m \times n}$ is the matrix

$$\begin{bmatrix} a_{11}\boldsymbol{B} & \ldots a_{1q}\boldsymbol{B} \\ \vdots & \vdots \\ a_{p1}\boldsymbol{B} & \ldots a_{pq}\boldsymbol{B} \end{bmatrix}$$

To calculate a Kronecker product in Spotfire S+, use the `kronecker` function:

```
> N <- matrix(5:8, nrow = 2)
> O <- matrix(4:1, nrow = 2)
> kronecker(N, O)

     [,1] [,2] [,3] [,4]
[1,]   20   10   28   14
[2,]   15    5   21    7
[3,]   24   12   32   16
[4,]   18    6   24    8
```

You can generalize `kronecker` to other operations besides multiplication by changing the operator with the `fun` argument:

```
> kronecker(N, O, fun = "+")

     [,1] [,2] [,3] [,4]
[1,]    9    7   11    9
[2,]    8    6   10    8
[3,]   10    8   12   10
[4,]    9    7   11    9
```

SOLVING SYSTEMS OF LINEAR EQUATIONS

Spotfire S+ provides several methods for solving systems of linear equations such as the following:

$$\begin{aligned} 19a + 2b + 15c &= 9 \\ 8a + 18b + 19c &= 5 \\ 11a + 17b + 10c &= 14 \end{aligned}$$

This system of equations can be expressed as the matrix equation $Ax = y$, where A is the matrix of coefficients, x is the (column) vector of unknowns (a, b, c), and y is the column vector of known values $(9, 5, 14)$. To define the coefficient matrix, type:

```
> A <- matrix(c(19, 8, 11, 2, 18, 17, 15, 19, 10), nrow=3)
> A

     [,1] [,2] [,3]
[1,]   19    2   15
[2,]    8   18   19
[3,]   11   17   10
```

The `solve` function takes the square matrix of coefficients and the vector of known values as arguments, and it returns the solution vector:

```
> solve(A, c(9, 5, 14))

[1]  0.9914429  0.6161109 -0.7379758
```

You can also use `solve` to obtain the inverse of a matrix:

```
> solve(A)

            [,1]         [,2]        [,3]
[1,]  0.04219534 -0.069341989  0.06845677
[2,] -0.03806433 -0.007376807  0.07111242
[3,]  0.01829448  0.088816760 -0.09619357
```

If the matrix is singular, `solve` returns an error message:

```
> S <- matrix(c(9, 3, 3, 3, 1, 1, 2, 4, 7), ncol = 3,
+ byrow = T)
> S

     [,1] [,2] [,3]
[1,]    9    3    3
[2,]    3    1    1
[3,]    2    4    7

> solve(S)

Error in solve.qr(a): apparently singular matrix
```

If the matrix of coefficients is upper triangular, you can use `backsolve` to solve the system of equations:

```
> U <- matrix(c(3, 0, 0, 1, 1, 0, 4, 5, 9), ncol=3)
> U

     [,1] [,2] [,3]
[1,]    3    1    4
[2,]    0    1    5
[3,]    0    0    9

> backsolve(U, c(9, 5, 14))

[1]  1.851852 -2.777778  1.555556
```

Choleski Decomposition

For symmetric, positive-definite matrices, the *Choleski decomposition* factors the matrix X uniquely in the form $X = R^T R$, where R is upper triangular. You can use the Choleski decomposition to generate upper triangular matrices for use with the `backsolve` function. Spotfire S+ has two functions for performing the Choleski decomposition: `chol` and `choleski`. The `chol` function is most useful for obtaining new matrices, since it returns only the upper triangular matrix R. The `choleski` function returns a list with the R matrix as one of its components.

For more information on the Choleski decomposition, see the `chol` help file and Chapter 8 of the *LINPACK User's Guide* by Dongarra, *et al.*

QR Decomposition

The *QR decomposition* expresses an $n \times p$ matrix X as the product of an $n \times n$ orthogonal matrix Q and an $n \times p$ upper triangular matrix R. The *QR* decomposition is the foundation for `solve` and `lsfit`, the (nonrobust) least-squares fit function.

To compute a *QR* decomposition, use the `qr` function. The value returned by `qr` is a list representing the *QR* numerical decomposition. The first component of the list is an $n \times p$ matrix in which the upper triangle, including the diagonal, is the R matrix. The entries under the diagonal contain most of a compact representation of Q. To obtain R and Q explicitly, use the functions `qr.R` and `qr.Q`, respectively. Another function, `qr.X`, reconstructs the original $n \times p$ matrix X from the numerical decomposition. In the following example, we use all four *QR* functions on the matrix A defined at the beginning of this section:

```
> qr(A)

$qr:
           [,1]        [,2]       [,3]
[1,] -23.3666429 -15.7917422 -23.409439
[2,]   0.3423684 -19.1734420  -8.987649
[3,]   0.4707565   0.6457152  -7.564412

$qraux:
[1] 1.813125 1.763578 0.000000

$rank:
[1] 3

$pivot:
[1] 1 2 3


> qr.Q(qr(A))

          [,1]       [,2]       [,3]
[1,] -0.8131249  0.5653998 -0.1383870
[2,] -0.3423684 -0.6568151 -0.6718465
[3,] -0.4707565 -0.4989158  0.7276478
```

```
> qr.R(qr(A))

          [,1]      [,2]       [,3]
[1,] -23.36664 -15.79174 -23.409439
[2,]   0.00000 -19.17344  -8.987649
[3,]   0.00000   0.00000  -7.564412

> qr.X(qr(A))

     [,1] [,2] [,3]
[1,]   19    2   15
[2,]    8   18   19
[3,]   11   17   10
```

The following functions use the return value from `qr` to perform additional calculations.

- `qr.coef`: Returns the coefficients obtained by a least-squares fit of response data y to the X matrix on which `qr` was used.
- `qr.fitted`: Returns the fitted values obtained by a least-squares fit of response data y to the X matrix on which `qr` was used.
- `qr.resid`: Returns the residuals obtained by a least-squares fit of response data y to the X matrix on which `qr` was used.
- `qr.qy`: Returns the results of the matrix multiplication Qy, where Q is the orthogonal transformation represented by `qr` and y is the response data.
- `qr.qty`: Returns the results of the matrix multiplication $Q^T y$, where Q is the orthogonal transformation represented by `qr` and y is the response data.

For more details on the *QR* decomposition, see the help files for `qr`, `qr.coef`, and `qr.Q` and Chapter 9 of the *LINPACK User's Guide* by Dongarra, *et al.*

The Singular Value Decomposition

The *singular value decomposition* takes an $n \times p$ matrix X and decomposes it into UDV^T, where U and V are orthogonal and D is a diagonal matrix. The elements of D are the *singular values* of X. The squares of the singular values of X are the eigenvalues of the matrix X^TX.

To obtain a singular value decomposition in Spotfire S+, use the `svd` function. This function returns a list in which the first component is a vector of singular values, the second component is the orthogonal matrix V, and the third component is the orthogonal matrix U. In the following example, we compute the singular value decomposition for the matrix A defined at the beginning of this section:

```
> svd(A)

$d:
[1] 40.000114 14.687207  5.768609

$v:
           [,1]       [,2]       [,3]
[1,] -0.5280363  0.6449356  0.5524814
[2,] -0.5533835 -0.7547957  0.3522074
[3,] -0.6441618  0.1197558 -0.7554563

$u:
           [,1]       [,2]        [,3]
[1,] -0.5200456  0.8538399 -0.02258456
[2,] -0.6606048 -0.4188323 -0.62304157
[3,] -0.5414369 -0.3090905  0.78186261
```

The singular value decomposition can be used as a numerically stable way to perform many operations that are used in multivariate statistics. One such operation is estimating the *rank* of a matrix ***X***.

For more information on the singular value decomposition, see the `svd` help file and Chapter 10 of the *LINPACK User's Guide* by Dongarra, *et al.*

EIGENVALUES AND EIGENVECTORS

If A is a square matrix and $Ax = \lambda x$ for a scalar λ and a vector x, then λ is an *eigenvalue* of A and x is an *eigenvector* of A. The Spotfire S+ function `eigen` returns both the eigenvalues and the eigenvectors associated with them. In the following example, we compute the eigenvalues and eigenvectors for the matrix A defined in the section Solving Systems of Linear Equations:

```
> eigen(A)

$values:
[1]  39.581985  13.677784  -6.259769

$vectors:
          [,1]       [,2]       [,3]
[1,] 0.6224278  0.8664541  0.3124109
[2,] 0.8793762 -0.6095730  0.3450415
[3,] 0.7368032 -0.2261540 -0.5721007
```

For more information on the `eigen` function, see the `eigen` help file.

INTEGRALS, DIFFERENCES, AND DERIVATIVES

Use the `integrate` function to compute the integral of a real-valued function over a given interval. The `integrate` function returns a list, of which the first two components are the integral and the absolute error:

```
> integrate(sin, 0, pi)[1:2]

$integral:
[1] 2


$abs.error:
[1] 2.220446e-14
```

For this simple example, we know that the integral of $\sin(x)$ over the interval $[0,\ \pi]$ is equal to $-\cos\pi - (-\cos 0)$. We can therefore check the result that Spotfire S+ returns with the following command:

```
>  (-cos(pi)) -  -cos(0)
[1] 2
```

Like many of the Spotfire S+ mathematical functions, `integrate` is most commonly used inside other function definitions. The following "wrapper" function provides a convenient command-line interface, and returns a single numeric value:

```
> integral <- function(f, lower, upper, ...) {
+        results <- integrate(f, lower, upper, ...)
+        if(results$message != "normal termination")
+                results$message
+        else results$integral
+ }
```

Use the `diff` function to obtain the *n*th difference of lag k for a set of data x. The default for both k and n is 1. The data may be in the form of a vector, time series, or matrix:

```
> y <- (1:10)^2
> diff(y)

[1]  3  5  7  9 11 13 15 17 19
```

With the following command, we compute differences for the built-in `corn.rain` time series:

```
> diff(corn.rain)

1891:  3.3 -3.0 -1.2 -1.9  5.7  0.5 -2.9  0.0  0.0  0.7
1901: -3.0  8.4 -2.1 -3.5 -0.6  1.5  2.1 -1.5 -0.1 -2.7
1911: -1.6  3.3 -4.1  2.6  7.0 -7.2  0.1 -0.7  0.8  2.1
1921:  0.5 -4.1  2.7  3.2 -2.6  0.3 -1.2
```

Differences on matrices are performed on each column separately:

```
> K <- matrix(c(12, 2, 13, 5, 10, 16, 7, 1), nrow=4)
> K

     [,1] [,2]
[1,]   12   10
[2,]    2   16
[3,]   13    7
[4,]    5    1

> diff(K)

     [,1] [,2]
[1,]  -10    6
[2,]   11   -9
[3,]   -8   -6
```

You can use `diff` to write a function for approximating the derivative of a data set:

```
> numdiff <- function(y, x=seq(along=y)) diff(y)/diff(x)
```

To perform symbolic differentiation, use the `D` function. AT&T suggests the `deriv` function, but `deriv` is most useful for providing derivatives to other Spotfire S+ functions. The `D` function is more useful for obtaining an isolated derivative:

```
> D(expression(3*x^2), "x")
3 * (2 * x)

> D(expression(exp(x^2)), "x")
exp(x^2) * (2 * x)

> D(expression(log(y)), "y")
1/y
```

INTERPOLATION AND APPROXIMATION

Spotfire S+ has a variety of functions for interpolation and approximation, most of them developed to aid in fitting curves and lines to data. However, they are sufficiently general to have wide application in mathematical settings.

Linear Interpolation

To find interpolated values in Spotfire S+, use the `approx` function. You provide a vector of *x* values, a vector of associated *y* values, and (optionally) a vector of *x* values at which you want interpolated values. Spotfire S+ returns a list of *x* values and the associated *y* values:

```
> approx(1:10, (1:10)^2, xout = c(2.5, 3.5))

$x:
[1]  2.5 3.5

$y:
[1]  6.5 12.5
```

A more specialized interpolation function, `interp`, can be used to generate input for the three-dimensional plotting functions `image`, `contour`, and `persp`. The `interp` function interpolates the value of the *z* variable onto an evenly spaced grid of the *x* and *y* variables:

```
> x <- cos(seq(-pi, pi, len = 9))
> y <- sin(seq(-pi, pi, len = 9))
> z <- x + y
> slanted.disk <- interp(x, y, z)
> persp(slanted.disk)
```

The resulting plot is shown in Figure 34.1.

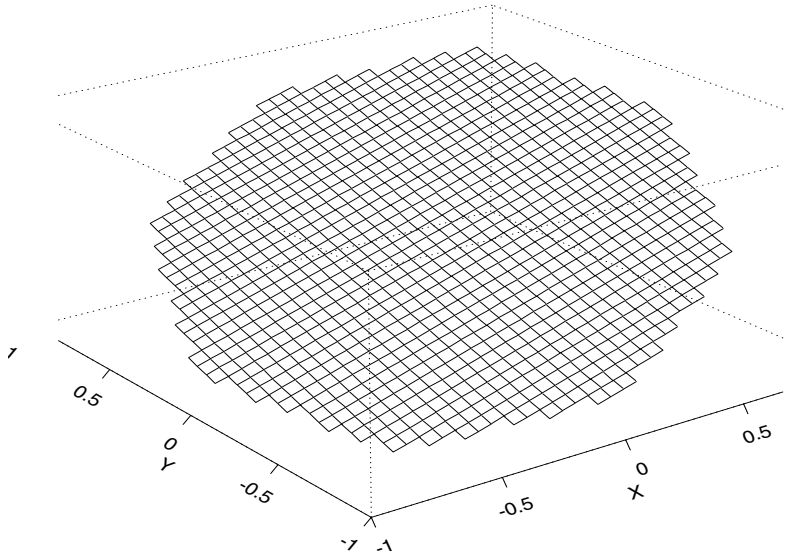

Figure 34.1: *A perspective plot created using* `interp`.

Convex Hull

To obtain the convex hull of a planar set of points, use the `chull` function, which returns the indices of the points belonging to the hull:

```
> chull(corn.rain)

 [1]  1  2 13 26 35 37 38 33 24  5
```

The `peel` option allows you to peel off the convex hull, take the convex hull of the remaining points, peel off *that* hull, and so on, until either all points are assigned to a hull or a user-specified limit is reached:

```
> chull(corn.rain, peel = T)

$depth:
 [1] 1 1 2 2 1 2 2 3 4 5 4 2 1 2 6 5 5 3 4 4 3 2 5 1 4 1 3
[28] 4 2 3 3 2 1 3 1 2 1 1
```

```
$hull:
 [1]  1  2 13 26 35 37 38 33 24  5  4  3  6  7 14 32 36 29
[19] 22 12 21  8 18 31 34 30 27  9 11 19 20 28 25 10 17 23
[37] 16 15

$count:
[1] 10 10  7  6  4  1
```

The `depth` component specifies which hull each point belongs to; `1` is the outermost hull. The `hull` component gives the indices of the points belonging to each hull. The first `count[1]` points belong to the outermost hull, the next `count[2]` points belong to the next hull, and so on.

Cubic Spline Approximation

Splines approximate a function with a set of polynomials defined on subintervals. A cubic spline is a collection of polynomials of degree less than or equal to 3 such that the second derivatives agree at the "knots." That is, the spline has a continuous second derivative.

When interpolating a number of points, a spline can be a much better solution than a polynomial interpolation, since the polynomial can oscillate wildly in order to hit all of the points. Polynomials fit the data *globally* while splines fit the data *locally*.

Use the `spline` function to obtain a cubic spline approximation:

```
> x <- 1:5
> y <- c(5, -5, 0, -5, 5)
> spline(x, y)

$x:
 [1] 1.000000 1.333333 1.666667 2.000000 2.333333 2.666667
 [7] 3.000000 3.333333 3.666667 4.000000 4.333333 4.666667
[13] 5.000000

$y:
 [1]  5.0000000  0.1851852 -3.5185184 -5.0000000 -3.7037036
 [6] -1.2962964  0.0000000 -1.2962964 -3.7037036 -5.0000000
[11] -3.5185184  0.1851852  5.0000000
```

The `spline` function is primarily used for graphing, and so it returns approximately three times as many output points as input points. For more details, see the `spline` help file.

Step Functions

The Spotfire S+ function `stepfun` creates a step function from either two vectors or a list with components named `x` and `y`. You can specify whether the step function is left- or right-continuous with the `type` argument. If `type="left"` (the default), the given points are at the left end of the level steps of the function; this gives a right-continuous function. If `type="right"`, the given points are at the right end of the level steps of the function; this gives a left-continuous function.

```
> x <- seq(1, 15, length=5)
> y <- x^2
> stepfun(x, y)

$x:
[1]  1.0  4.5  4.5  8.0  8.0 11.5 11.5 15.0 15.0

$y:
[1]   1.00   1.00  20.25  20.25  64.00  64.00 132.25 132.25
[9] 225.00

> plot(stepfun(x, y), type = "l")
```

The resulting plot is shown in Figure 34.2.

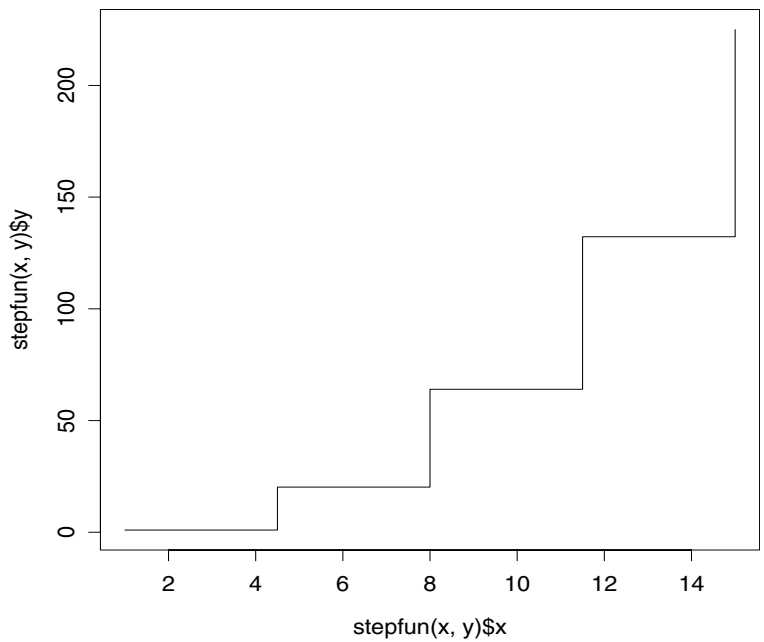

Figure 34.2: *A step function.*

INITIAL VALUE PROBLEMS

Initial Value Problems for Ordinary Differential Equations

An *initial value problem* is a system of first order differential equations together with a complete set of initial conditions, one for each equation in the system:

$$y_1'(x) = f_1(x, y_1(x), \ldots, y_n(x)) \qquad y_1(x_0) = a_1$$

$$\vdots$$

$$y_n'(x) = f_n(x, y_1(x), \ldots, y_n(x)) \qquad y_n(x_0) = a_n$$

Since second and higher order differential equations can always be expressed as systems of first order equations, this definition of the initial value problem is completely general. The solution of an initial value problem with n equations is, for any specified point x, a vector y of function values $(y_1(x), \ldots, y_n(x))$. The initial conditions $a_1, \ldots, a_n$ are simply the solution at a given initial point x_0.

To solve initial value problems in Spotfire S+, use the `ivp.ab` function. You set up the problem as follows:

1. Specify a system of differential equations as in the above definition. For example, consider the following initial value problem:

$$y_1' = -y_2 \qquad y_1(0) = 2$$
$$y_2' = y_1 + x \qquad y_2(0) = -1$$

2. Define a Spotfire S+ function that returns a vector representing the derivatives y_i' expressed as functions of x and y_i. For the system given above, a Spotfire S+ function `ivp.ex1` is defined as follows:

```
> ivp.ex1 <- function(x, y) c(-y[2], y[1] + x)
```

3. Define the initial condition as a vector of the form $(x_0,(y_1(x_0), \ldots, y_n(x_0)))$. The first element of this vector specifies the initial point and the remaining elements are the

initial conditions for the equations in the system. In our example, this vector can be defined with the Spotfire S+ command `init.vals <- c(0, c(2,-1))`.

4. Choose the point at which you want Spotfire S+ to calculate the solution; this point is called the *final point*.

Once you've worked out the form of the derivatives and the initial condition list, the call to `ivp.ab` is straightforward. For example, here we solve our initial value problem at the final point $x = \pi$:

```
> ivp.ex1.sol <- ivp.ab(final.point=pi, initial=init.vals,
+ derivatives=ivp.ex1)
```

The `ivp.ab` function returns a list with several components, many of which are most useful as input for further iterations of the function. The solution itself is stored in the `values` component:

```
> ivp.ex1.sol$values

     point      val 1     val 2
 3.141593 -5.141677 2.999694
```

This particular example has the known analytic solution

$$\begin{aligned} y_1(x) &= -x + A\cos x + (1-B)\sin x \\ y_2(x) &= 1 - (1-B)\cos x + A\sin x \end{aligned},$$

where $A = y_1(0)$ and $B = y_2(0)$. You can therefore check the solution that `ivp.ab` computes by substituting $A = 2$, $B = -1$, and the final point $x = \pi$ into the analytic solution:

```
> c(-pi + 2*cos(pi) + (1 - (-1))*sin(pi),
+ 1 - (1 - (-1))*cos(pi) + 2*sin(pi))

[1] -5.141593  3.000000
```

Example: Projectile Motion

One familiar source of initial value problems is projectile motion. Consider a body of constant mass m that is launched vertically upward from the surface of the earth at an initial velocity of $v_0 = 500$ meters per second. Near sea level, the gravitational acceleration is assumed to be constant at $g = 9.8$ meters per second squared. What is the height of the projectile at time $t = 4$?

We have the following one-dimensional initial value problem:

$$y'(t) = v_0 - gt \qquad y(0) = 0.$$

We can set up our derivative function to accept v_0 and g as parameters with the following Spotfire S+ command:

```
> ivp.ex2 <- function(t,y,v0,g) return(v0 - g*t)
```

We then specify the values of v_0 and g as a list to the `aux` argument in `ivp.ab`. The following syntax solves our initial value problem at the point $t = 4$:

```
> ivp.ex2.sol <- ivp.ab(final.point = 4, initial = c(0,0),
+ derivatives = ivp.ex2, aux = list(v0=500, g=9.8))

> ivp.ex2.sol$values

 point  val 1
     4 1921.6
```

The analytic solution to this equation is easily obtained by integrating:

$$y(t) = \int_0^t (v_0 - gx)\,dx + y(0)$$

$$= v_0 t - \frac{1}{2} g t^2$$

We can therefore verify the solution that `ivp.ab` returns by substituting the appropriate values into the analytic solution:

```
> 500*4 - 0.5*9.8*4^2

[1] 1921.6
```

You can also use the Spotfire S+ function `integrate` to solve one-dimensional problems directly:

```
> integrate(function(t, v0 = 500, g = 9.8) {v0-g*t},
+ lower = 0, upper = 4)$integral

[1] 1921.6
```

For more details on the `integrate` function, see the section Integrals, Differences, and Derivatives.

Example: Simple Harmonic Motion

Simple harmonic motion is governed by the second-order differential equation

$$mu'' = -ku,$$

where u is displacement as a function of time, u'' is acceleration, m is mass, and k is the *spring constant*. For example, consider a spring that has a natural length of 5 centimeters. Suppose the spring stretches an additional centimeter when a 30 gram weight is attached to one end of it. In this case, $m = 30$ grams, $\iota = 1/100$ meters, and u'' is gravitational acceleration (9.8 meters per second squared). The spring constant k can be calculated by substituting these values into the above equation:

$$k = \frac{30 \times 9.8}{1/100} = 29,400.$$

The differential equation that describes the motion of the spring is therefore

$$30u'' = -29,400u.$$

If the spring is stretched 2 centimeters past its natural length and then released so that it oscillates, what is the displacement u at time $t = 1.75$?

To solve second-order differential equations in Spotfire S+, we must rewrite them as systems of first-order equations. Let $y_1 = u$ and $y_2 = u'$. This change of variables gives the following initial value problem for the spring example:

$$\begin{aligned} y_1' &= y_2 & y_1(0) &= 2 \\ y_2' &= -\frac{29,400}{30} y_1 & y_2(0) &= 0 \end{aligned}.$$

We define our derivative function `ivp.ex3` with the following Spotfire S+ command:

```
> ivp.ex3 <- function(t,y) c(y[2], -(29400/30)*y[1])
```

To solve the initial value problem at time $t = 1.75$, we call `ivp.ab` and extract the resulting `values` component:

```
> ivp.ex3.sol <- ivp.ab(final.point=1.75,
+ initial=c(0, c(2,0)), derivatives=ivp.ex3)

> ivp.ex3.sol$values

  point      val 1    val 2
   1.75 -0.3854707 61.40628
```

The displacement at $t = 1.75$ is therefore $u = y_1 = -0.3854707$. This solution indicates that the spring is approximately 0.385 centimeters shorter than its natural length after 1.75 seconds.

Computational Notes

You should be aware that you won't be able to solve all initial value problems with the `ivp.ab` function. First of all, not every initial value problem has a solution. Secondly, the problem may be *unstable*, so that errors are magnified by the numerical method as the solution proceeds. Finally, the method has difficulty with some *stiff* problems, in which solutions have several components that exhibit widely varying behaviors over the solution interval. For further discussion of these and other topics, see Shampine and Gordon (1975).

THE FAST FOURIER TRANSFORM

Spotfire S+ has several functions useful for signal processing, including the fast Fourier transform and three kinds of filters: convolution, recursive, and low-pass. For a complete description of the filters implemented in Spotfire S+, see the section Linear Filters in the chapter Analyzing Time Series and Signals.

The function `fft` calculates the unnormalized discrete Fourier transform of the input data, which can be any numeric or complex vector, array, or time series. For a vector z of length n, the definition of the transform `x<-fft(z)` is:

$$x_j = \sum_{t=1}^{n} z_t \exp(-i\omega_{j-1}(t-1)),$$

where $1 \le j \le n$ and ω_j is the jth *Fourier frequency* $2\pi j / n$. Because of the imaginary exponent, the output from `fft` is of mode `"complex"`. This can be seen in the following example:

```
> fft(1:10)

 [1]  55+ 0.000000i  -5+15.388418i  -5+ 6.881910i
 [4]  -5+ 3.632713i  -5+ 1.624598i  -5+ 0.000000i
 [7]  -5- 1.624598i  -5- 3.632713i  -5- 6.881910i
[10]  -5-15.388418i
```

If the input data is an array (for example, a matrix), `fft` returns the multi-dimensional unnormalized discrete Fourier transform of the array. For an $m \times n$ array Z, the definition of the transform `X<-fft(Z)` is:

$$X_{jk} = \sum_{s=1}^{m} \sum_{t=1}^{n} Z_{st} e^{-2\pi i(j-1)(s-1)} e^{-2\pi i(k-1)(t-1)},$$

where $1 \le j \le m$, $1 \le k \le n$, and X_{jk} denotes the $[j,k]$ entry of X. The result is a complex array with the same shape as the input data Z. Therefore, using `fft` on a multivariate time series does not compute the time transform. With the following command, we compute the discrete Fourier transform of the matrix A that we defined in the section Solving Systems of Linear Equations.

```
> A

     [,1] [,2] [,3]
[1,]   19    2   15
[2,]    8   18   19
[3,]   11   17   10

> fft(A)

                  [,1]             [,2]             [,3]
[1,]  119.0+0.000000i  -2.5+ 6.062178i  -2.5- 6.062178i
[2,]   -5.5-6.062178i  23.0+20.784610i  11.0- 6.928203i
[3,]   -5.5+6.062178i  11.0+ 6.928203i  23.0-20.784610i
```

To compute the inverse transform, use `fft` with the argument `inverse=TRUE`. For a vector x of length n, the definition of the transform `z<-fft(x, inverse=T)` is:

$$z_t = \sum_{j=1}^{n} x_j \exp(i\omega_{t-1}(j-1)),$$

where $1 \le t \le n$ and $\omega_t = 2\pi t/n$. Likewise, for an $m \times n$ matrix X, the definition of the transform `Z<-fft(X, inverse=T)` is:

$$Z_{st} = \sum_{j=1}^{m} \sum_{k=1}^{n} X_{jk} e^{2\pi i(s-1)(j-1)} e^{2\pi i(t-1)(k-1)},$$

where $1 \le s \le m$, $1 \le t \le n$, and Z_{st} denotes the $[s,t]$ entry of Z. In the following example, we compute the inverse Fourier transform of the vector, $(-1 + i\sqrt{3})/2$, $(-1 - i\sqrt{3})/2$.

```
> cuberoot.1 <- (cos(2*pi/3) + sin(2*pi/3)*1i)^(0:2)
> cuberoot.1

[1]  1.0+0.0000000i -0.5+0.8660254i -0.5-0.8660254i

> fft(cuberoot.1, inverse = T)

[1] 0.000000e+00+3.330669e-16i 2.220446e-16+3.142072e-16i
[3] 3.000000e+00-6.472741e-16i
```

Note that the *unnormalized* transforms implemented in `fft` involve sums instead of means. For an n dimensional vector x, this causes the commands

```
> fft(fft(x), inverse=T)
> fft(fft(x, inverse=T))
```

to both return nx (approximately, depending on roundoff error). In contrast, the *normalized* discrete Fourier transform divides the unnormalized result by the length of the input; the normalized output contains the *Fourier coefficients*. If you require Fourier coefficients for your analysis, you should divide the value that `fft` returns by the length of your input vector.

The discrete Fourier transform is used to compute an approximation to the continuous Fourier transform of a periodic function f. In the usual definition, n points are sampled from f symmetrically around 0; that is, the domain of the sampled points is $-N/2, N/2$, where N is the period of f. However, Spotfire S+ assumes the n points are sampled from the interval $[0, N]$. When this convention is followed, the resulting frequencies are shifted. For example, let j be the index of the sampled points and suppose n is even. In Spotfire S+, the zero frequency corresponds to $j = 1$, the positive frequencies correspond to $2 \le j \le n/2$, the negative frequencies correspond to $n/2 + 2 \le j \le n$, and the Nyquist critical frequency corresponds to $j = n/2 + 1$. The definitions are analogous if n is odd: the zero frequency corresponds to $j = 1$, the positive frequencies correspond to $2 \le j \le \frac{n+1}{2}$, the negative frequencies correspond to $\frac{n+1}{2} + 1 \le j \le n$, and there is no Nyquist critical frequency. For more details, see Press *et al.* (1996).

PROBABILITY AND RANDOM NUMBERS

Spotfire S+ has many functions that perform probability calculations for the most common distributions. Each of these functions has a name that begins with a one-letter code indicating the type of function: `r`*`dist`*, `p`*`dist`*, `d`*`dist`*, and `q`*`dist`*, respectively, where *`dist`* is the Spotfire S+ distribution function. The one-letter codes are as follows.

- `r`: Random number generator. Requires an argument specifying the sample size, plus any required distribution parameters.
- `p`: Probability function. Requires a vector of quantiles, plus any required distribution parameters.
- `d`: Density function. Requires a vector of quantiles, plus any required distribution parameters.
- `q`: Quantile function. Requires a vector of probabilities, plus any required distribution parameters.

For a detailed description of the distribution functions implemented in Spotfire S+, see the chapter Probability in the *Guide to Statistics, Volume 1*. The probability chapter includes a table of distributions currently supported by Spotfire S+, along with the codes used to identify them.

For those users interested in understanding the Spotfire S+ *pseudo-random number generator* (PNG), we present the internals of the algorithm here. All Spotfire S+ functions that generate random numbers rely on the underlying PNG, which computes uniform random numbers in the interval $(0,1)$. We discuss the algorithm briefly and at a relatively high level; for general background knowledge on random number generators, see Ripley (1987) or Kennedy and Gentle (1980).

The Spotfire S+ Pseudo-Random Number Generator

The pseudo-random number generator implemented in Spotfire S+ is based on George Marsaglia's original "Super-Duper" package from 1973. It produces a 32-bit integer whose top 31 bits are divided by $2^{31} = 2,147,483,648$. The result is a real number in the half-open interval $[0, 1)$. The 32-bit integer is computed by a bitwise exclusive-or of two additional 32-bit integers: one produced by a *congruential generator*, and one produced by a *Tausworthe generator.*

The congruential generator, also known as the *linear* or *mixed congruential method*, is one of the most commonly-used random number generators. It produces a sequence of numbers X_i via the recursive relationship

$$X_{i+1} = (aX_i + c) \bmod m,$$

where a is a multiplicative constant and c is an additive constant. The *modulus* m is chosen to be as large as possible, since the period of the generator cannot be larger than m. In Spotfire S+, $m = 2^{32}$ and all overflowing bits are discarded. The initial value X_0 is called the *seed*. Various combinations have been proposed for a and c, and Spotfire S+ uses the values $a = 69069$ and $c = 0$:

$$X_{i+1} = (69069X_i) \bmod 2^{32}.$$

This recursion results in a strictly multiplicative generator that has a period of $n/4 = 2^{30}$.

The Tausworthe generator produces a sequence of numbers Y_i via the exclusive-or operation

$$Y_i = Y_{i-p} \text{ xor } Y_{i-(p-q)}.$$

In Spotfire S+, $p = 32$ and $q = 15$. For most starting seeds Y_0, this generator has a period of $4,292,868,097$. Combining this with the congruential part gives a PNG that has a period of $2^{30} \times 4,292,868,097 \approx 4.6 \times 10^{18}$. The Spotfire S+ generator skips cases in which the result is exactly 0, producing random numbers in the open interval $(0,1)$; this reduces the period by a small amount.

The .Random.seed Object

The Spotfire S+ vector `.Random.seed` stores the starting values X_0 and Y_0 for the congruential and Tausworthe generators, respectively. The first time random numbers are computed in a Spotfire S+ session, `.Random.seed` is modified and copied to the local working database. In general, `.Random.seed` is updated with the current congruential and Tausworthe values whenever Spotfire S+ computes a random sample. The following example illustrates this. In the code below, the function `set.seed` defines the starting seeds for a particular sequence of random numbers.

```
> set.seed(1)
> .Random.seed

 [1] 21 14 49 48 24  1 32 22 36 23 28  3

> x <- rnorm(100)
> .Random.seed

 [1] 13  8 57 53 18  3 33 33 11 41 53  3
```

This mechanism maintains the long-term properties of the generator, and also allows for reproducibility of results. For more details, see the help files for `.Random.seed` and `set.seed`.

When a function containing a call to the random number generator aborts before finishing, `.Random.seed` is not modified. The following contrived example illustrates this. In the code below, we create a function `test.func` that is guaranteed to abort before it completes.

```
> .Random.seed

 [1] 13  8 57 53 18  3 33 33 11 41 53  3

> test.func <- function() {
+ x <- rnorm(100)
+ x[1] <- NA
+ if(any(is.na(x))) stop("NAs are not allowed in x")
+ return(x)
+ }

> test.func()

Error in test.func(): NAs are not allowed in x

> .Random.seed
```

```
 [1] 13  8 57 53 18  3 33 33 11 41 53  3
```

The `.Random.seed` vector encodes the base 64 representations of the current congruential and Tausworthe values. It stores twelve integers from the interval $[0,63]$, where the sixth and twelfth entries are from $[0,3]$. The congruential value X is encoded in the first six integers of `.Random.seed`, and the Tausworthe part Y is encoded in the last six. If the entries of `.Random.seed` are denoted by r_i for $i = 1, 2, \ldots, 12$, then

$$X = \sum_{i=1}^{6} r_i 2^{6(i-1)},$$

$$Y = \sum_{i=1}^{6} r_{i+6} 2^{6(i-1)}.$$

It is possible to shorten the period of this generator dramatically by changing specific bits in `.Random.seed`. Because of this, set the starting seed only with the methods outlined in the `set.seed` and `.Random.seed` help files.

Computational Note

Note that 64 bits are required by the Spotfire S+ pseudo-random number generator (32 for X and 32 for Y) to produce the final value returned. This means that it is possible to generate repeats in a long sequence of random numbers, even though the full period has not been reached. For example, consider the following command:

```
> set.seed(15)
> table(table(runif(500000)*2147483648))

     1  2
499866 67
```

We multiply the results from `runif` by $2^{31} = 2,147,483,648$ to recover the original 31 bits produced by the exclusive-or of X and Y. The output says that 499,866 of the random numbers show up once in the sequence and 67 show up twice. However, the period of the Spotfire S+ random number generator is much larger than 500,000, the length of the sequence in this example.

PRIMES AND FACTORS

Spotfire S+ can be useful in many number-theoretic computations, as we have already seen with the %% and %/% operators. You can define simple functions to list prime numbers and perform factorization; although they will not set computational records, you may find them useful.

The primes function returns all prime numbers less than or equal to a given n, where by default $n = 100$:

```
> primes <- function(n = 100) {
+         n <- as.integer(abs(n))
+         if(n < 2)
+                 return(integer(0))
+         p <- 2:n
+         smallp <- integer(0)    # the sieve
+         repeat {
+                 i <- p[1]
+                 smallp <- c(smallp, i)
+                 p <- p[p %% i != 0]
+                 if(i > sqrt(n))
+                         break
+         }
+         c(smallp, p)
+ }

> primes(75)

 [1]  2  3  5  7 11 13 17 19 23 29 31 37 41 43 47 53 59 61
[19] 67 71 73
```

The factors function returns the prime factors of an integer n:

```
> factors <- function(n) {
+         n <- as.integer(abs(n))
+         if(!exists(".Primes") || max(.Primes) < sqrt(n))
+                 assign(".Primes", primes(as.integer(1.3 *
+                         sqrt(n))), where = 1)
+         pfactors <- integer(0)
+         while(n > 1) {
+                 new.factors <- .Primes[n %% .Primes == 0]
```

```
+                 if(length(new.factors) == 0)
+                         new.factors <- n
+                 n <- as.integer(n/(prod(new.factors)))
+                 pfactors <- c(pfactors, new.factors)
+         }
+         sort(pfactors)
+ }
> factors(3012)

[1]   2   2   3 251
```

A NOTE ON COMPUTATIONAL ACCURACY

Spotfire S+ performs its computations in double precision, unless specifically written as integer or single precision. Computed values are accurate to approximately 14 decimal places. However, computed values can provide no more significant digits than the data they are computed from.

The exact limits on computations in Spotfire S+ are determined by the parameters of machine arithmetic stored in the Spotfire S+ object `.Machine`. The object `.Machine` is a list with various numeric components whose names are made up of the characters `single.` or `double.` followed by the name of a particular parameter of machine arithmetic. For example, `single.digits` is the number of base `single.base` digits in the floating point representation of a single-precision number. In addition, the component `integer.max` is the largest integer.

See the `.Machine` help file for more information.

REFERENCES

Dongarra, J.J., Bunch, J.R., Moler, C.B. & Stewart, G.W. (1979). *LINPACK User's Guide*. Philadelphia: SIAM.

Kennedy, W. J. & Gentle, J. E. (1980). *Statistical Computing*. New York: Marcel Dekker.

Marsaglia, G. *et al.* (1973). *Random Number Package: "Super-Duper"*. School of Computer Science, McGill University.

Press, W.H., Teukolsky, S.A., Vetterling, W.T., & Flannery, B.P. (1996). *Numerical Recipes in Fortran 77: The Art of Scientific Computing* (2nd ed.). New York: Cambridge University Press.

Rice, J.A. (1995). *Mathematical Statistics and Data Analysis* (2nd ed.). Belmont, CA: Duxbury Press.

Ripley, B.D. (1987). *Stochastic Simulation*. New York: John Wiley and Sons.

Shampine, L.F. & Gordon, M. (1975). *Computer Solution of Ordinary Differential Equations: The Initial Value Problem*. San Francisco: Freeman.

Venables, W.N. & Ripley B.D. (1997). *Modern Applied Statistics with* Spotfire S+ (2nd ed.). New York: Springer-Verlag.

INDEX

Symbols

Numerics

A

B

C

D

G

H

M

Q

R

S

T

X

Y