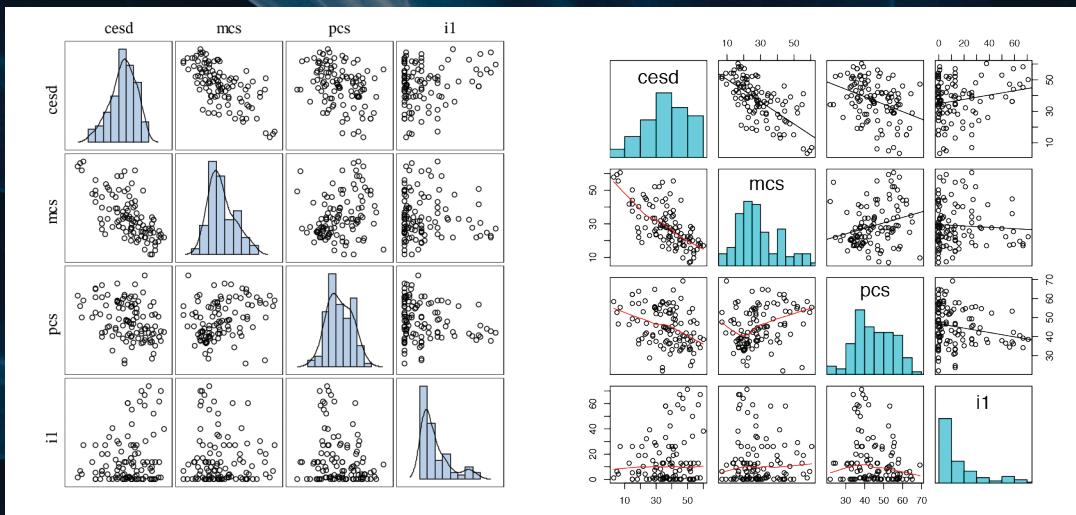


# SAS and R

## Data Management, Statistical Analysis, and Graphics

SECOND EDITION



Ken Kleinman and Nicholas J. Horton



CRC Press  
Taylor & Francis Group

A CHAPMAN & HALL BOOK



# SAS and R

Data Management,  
Statistical Analysis,  
and Graphics

SECOND EDITION



# SAS and R

Data Management,  
Statistical Analysis,  
and Graphics

SECOND EDITION

Ken Kleinman

Department of Population Medicine  
Harvard Medical School and  
Harvard Pilgrim Health Care Institute  
Boston, Massachusetts, U.S.A.

Nicholas J. Horton

Department of Mathematics and Statistics  
Amherst College  
Amherst, Massachusetts, U.S.A.



CRC Press  
Taylor & Francis Group  
Boca Raton London New York

CRC Press is an imprint of the  
Taylor & Francis Group an **informa** business  
A CHAPMAN & HALL BOOK

CRC Press  
Taylor & Francis Group  
6000 Broken Sound Parkway NW, Suite 300  
Boca Raton, FL 33487-2742

© 2014 by Taylor & Francis Group, LLC  
CRC Press is an imprint of Taylor & Francis Group, an Informa business

No claim to original U.S. Government works  
Version Date: 20140415

International Standard Book Number-13: 978-1-4665-8450-1 (eBook - PDF)

This book contains information obtained from authentic and highly regarded sources. Reasonable efforts have been made to publish reliable data and information, but the author and publisher cannot assume responsibility for the validity of all materials or the consequences of their use. The authors and publishers have attempted to trace the copyright holders of all material reproduced in this publication and apologize to copyright holders if permission to publish in this form has not been obtained. If any copyright material has not been acknowledged please write and let us know so we may rectify in any future reprint.

Except as permitted under U.S. Copyright Law, no part of this book may be reprinted, reproduced, transmitted, or utilized in any form by any electronic, mechanical, or other means, now known or hereafter invented, including photocopying, microfilming, and recording, or in any information storage or retrieval system, without written permission from the publishers.

For permission to photocopy or use material electronically from this work, please access [www.copyright.com](http://www.copyright.com) (<http://www.copyright.com/>) or contact the Copyright Clearance Center, Inc. (CCC), 222 Rosewood Drive, Danvers, MA 01923, 978-750-8400. CCC is a not-for-profit organization that provides licenses and registration for a variety of users. For organizations that have been granted a photocopy license by the CCC, a separate system of payment has been arranged.

**Trademark Notice:** Product or corporate names may be trademarks or registered trademarks, and are used only for identification and explanation without intent to infringe.

Visit the Taylor & Francis Web site at  
<http://www.taylorandfrancis.com>

and the CRC Press Web site at  
<http://www.crcpress.com>

# Contents

<b>List of figures</b>	<b>xvii</b>
<b>List of tables</b>	<b>xix</b>
<b>Preface to the second edition</b>	<b>xxi</b>
<b>Preface to the first edition</b>	<b>xxiii</b>
<b>1 Data input and output</b>	<b>1</b>
1.1 Input . . . . .	1
1.1.1 Native dataset . . . . .	1
1.1.2 Fixed format text files . . . . .	2
1.1.3 Other fixed files . . . . .	3
1.1.4 Reading more complex text files . . . . .	3
1.1.5 Comma separated value (CSV) files . . . . .	4
1.1.6 Read sheets from an Excel file . . . . .	5
1.1.7 Read data from R into SAS . . . . .	5
1.1.8 Read data from SAS into R . . . . .	6
1.1.9 Reading datasets in other formats . . . . .	6
1.1.10 Reading data with a variable number of words in a field . . . . .	7
1.1.11 Read a file byte by byte . . . . .	8
1.1.12 Access data from a URL . . . . .	9
1.1.13 Read an XML-formatted file . . . . .	9
1.1.14 Manual data entry . . . . .	10
1.2 Output . . . . .	11
1.2.1 Displaying data . . . . .	11
1.2.2 Number of digits to display . . . . .	11
1.2.3 Save a native dataset . . . . .	12
1.2.4 Creating datasets in text format . . . . .	12
1.2.5 Creating Excel spreadsheets . . . . .	12
1.2.6 Creating files for use by other packages . . . . .	13
1.2.7 Creating HTML formatted output . . . . .	14
1.2.8 Creating XML datasets and output . . . . .	14
1.3 Further resources . . . . .	15
<b>2 Data management</b>	<b>17</b>
2.1 Structure and meta-data . . . . .	17
2.1.1 Access variables from a dataset . . . . .	17
2.1.2 Names of variables and their types . . . . .	17
2.1.3 Values of variables in a dataset . . . . .	18

2.1.4	Label variables . . . . .	18
2.1.5	Add comment to a dataset or variable . . . . .	19
2.2	Derived variables and data manipulation . . . . .	19
2.2.1	Add derived variable to a dataset . . . . .	19
2.2.2	Rename variables in a dataset . . . . .	19
2.2.3	Create string variables from numeric variables . . . . .	20
2.2.4	Create categorical variables from continuous variables . . . . .	20
2.2.5	Recode a categorical variable . . . . .	21
2.2.6	Create a categorical variable using logic . . . . .	21
2.2.7	Create numeric variables from string variables . . . . .	22
2.2.8	Extract characters from string variables . . . . .	23
2.2.9	Length of string variables . . . . .	23
2.2.10	Concatenate string variables . . . . .	24
2.2.11	Set operations . . . . .	24
2.2.12	Find strings within string variables . . . . .	25
2.2.13	Find approximate strings . . . . .	25
2.2.14	Replace strings within string variables . . . . .	26
2.2.15	Split strings into multiple strings . . . . .	26
2.2.16	Remove spaces around string variables . . . . .	27
2.2.17	Upper to lower case . . . . .	27
2.2.18	Lagged variable . . . . .	28
2.2.19	Formatting values of variables . . . . .	28
2.2.20	Perl interface . . . . .	29
2.2.21	Accessing databases using SQL (structured query language) . . . . .	29
2.3	Merging, combining, and subsetting datasets . . . . .	29
2.3.1	Subsetting observations . . . . .	30
2.3.2	Drop or keep variables in a dataset . . . . .	30
2.3.3	Random sample of a dataset . . . . .	31
2.3.4	Observation number . . . . .	32
2.3.5	Keep unique values . . . . .	32
2.3.6	Identify duplicated values . . . . .	32
2.3.7	Convert from wide to long (tall) format . . . . .	33
2.3.8	Convert from long (tall) to wide format . . . . .	34
2.3.9	Concatenate and stack datasets . . . . .	35
2.3.10	Sort datasets . . . . .	35
2.3.11	Merge datasets . . . . .	35
2.4	Date and time variables . . . . .	37
2.4.1	Create date variable . . . . .	37
2.4.2	Extract weekday . . . . .	38
2.4.3	Extract month . . . . .	38
2.4.4	Extract year . . . . .	38
2.4.5	Extract quarter . . . . .	38
2.4.6	Create time variable . . . . .	39
2.5	Further resources . . . . .	39
2.6	Examples . . . . .	39
2.6.1	Data input and output . . . . .	39
2.6.2	Data display . . . . .	43
2.6.3	Derived variables and data manipulation . . . . .	44
2.6.4	Sorting and subsetting datasets . . . . .	51

<b>3 Statistical and mathematical functions</b>	<b>53</b>
3.1 Probability distributions and random number generation . . . . .	53
3.1.1 Probability density function . . . . .	53
3.1.2 Quantiles of a probability density function . . . . .	54
3.1.3 Setting the random number seed . . . . .	55
3.1.4 Uniform random variables . . . . .	55
3.1.5 Multinomial random variables . . . . .	56
3.1.6 Normal random variables . . . . .	56
3.1.7 Multivariate normal random variables . . . . .	56
3.1.8 Truncated multivariate normal random variables . . . . .	58
3.1.9 Exponential random variables . . . . .	58
3.1.10 Other random variables . . . . .	58
3.2 Mathematical functions . . . . .	59
3.2.1 Basic functions . . . . .	59
3.2.2 Trigonometric functions . . . . .	60
3.2.3 Special functions . . . . .	60
3.2.4 Integer functions . . . . .	60
3.2.5 Comparisons of floating point variables . . . . .	61
3.2.6 Complex numbers . . . . .	61
3.2.7 Derivatives . . . . .	62
3.2.8 Integration . . . . .	62
3.2.9 Optimization problems . . . . .	62
3.3 Matrix operations . . . . .	63
3.3.1 Create matrix from vector . . . . .	63
3.3.2 Combine vectors or matrices . . . . .	63
3.3.3 Matrix addition . . . . .	64
3.3.4 Transpose matrix . . . . .	64
3.3.5 Find the dimension of a matrix or dataset . . . . .	64
3.3.6 Matrix multiplication . . . . .	65
3.3.7 Invert matrix . . . . .	65
3.3.8 Component-wise multiplication . . . . .	66
3.3.9 Create submatrix . . . . .	66
3.3.10 Create a diagonal matrix . . . . .	66
3.3.11 Create a vector of diagonal elements . . . . .	67
3.3.12 Create a vector from a matrix . . . . .	67
3.3.13 Calculate the determinant . . . . .	67
3.3.14 Find eigenvalues and eigenvectors . . . . .	67
3.3.15 Find the singular value decomposition . . . . .	68
3.4 Examples . . . . .	68
3.4.1 Probability distributions . . . . .	68
<b>4 Programming and operating system interface</b>	<b>71</b>
4.1 Control flow, programming, and data generation . . . . .	71
4.1.1 Looping . . . . .	71
4.1.2 Conditional execution . . . . .	72
4.1.3 Sequence of values or patterns . . . . .	73
4.1.4 Referring to a range of variables . . . . .	74
4.1.5 Perform an action repeatedly over a set of variables . . . . .	74
4.1.6 Grid of values . . . . .	75
4.1.7 Debugging . . . . .	76
4.1.8 Error recovery . . . . .	76

4.2 Functions and macros . . . . .	77
4.2.1 SAS macros . . . . .	77
4.2.2 R functions . . . . .	78
4.3 Interactions with the operating system . . . . .	78
4.3.1 Timing commands . . . . .	78
4.3.2 Suspend execution for a time interval . . . . .	79
4.3.3 Execute a command in the operating system . . . . .	79
4.3.4 Command history . . . . .	80
4.3.5 Find working directory . . . . .	80
4.3.6 Change working directory . . . . .	80
4.3.7 List and access files . . . . .	81
<b>5 Common statistical procedures</b>	<b>83</b>
5.1 Summary statistics . . . . .	83
5.1.1 Means and other summary statistics . . . . .	83
5.1.2 Other moments . . . . .	84
5.1.3 Trimmed mean . . . . .	84
5.1.4 Quantiles . . . . .	85
5.1.5 Centering, normalizing, and scaling . . . . .	85
5.1.6 Mean and 95% confidence interval . . . . .	86
5.1.7 Proportion and 95% confidence interval . . . . .	86
5.1.8 Maximum likelihood estimation of parameters . . . . .	86
5.2 Bivariate statistics . . . . .	87
5.2.1 Epidemiologic statistics . . . . .	87
5.2.2 Test characteristics . . . . .	87
5.2.3 Correlation . . . . .	89
5.2.4 Kappa (agreement) . . . . .	89
5.3 Contingency tables . . . . .	90
5.3.1 Display cross-classification table . . . . .	90
5.3.2 Displaying missing value categories in a table . . . . .	90
5.3.3 Pearson chi-square statistic . . . . .	91
5.3.4 Cochran–Mantel–Haenszel test . . . . .	91
5.3.5 Cramér’s V . . . . .	91
5.3.6 Fisher’s exact test . . . . .	92
5.3.7 McNemar’s test . . . . .	92
5.4 Tests for continuous variables . . . . .	92
5.4.1 Tests for normality . . . . .	92
5.4.2 Student’s <i>t</i> test . . . . .	93
5.4.3 Test for equal variances . . . . .	93
5.4.4 Nonparametric tests . . . . .	94
5.4.5 Permutation test . . . . .	94
5.4.6 Logrank test . . . . .	95
5.5 Analytic power and sample size calculations . . . . .	95
5.6 Further resources . . . . .	97
5.7 Examples . . . . .	97
5.7.1 Summary statistics and exploratory data analysis . . . . .	97
5.7.2 Bivariate relationships . . . . .	101
5.7.3 Contingency tables . . . . .	103
5.7.4 Two sample tests of continuous variables . . . . .	107
5.7.5 Survival analysis: logrank test . . . . .	111

<b>6 Linear regression and ANOVA</b>	<b>113</b>
6.1 Model fitting . . . . .	113
6.1.1 Linear regression . . . . .	113
6.1.2 Linear regression with categorical covariates . . . . .	114
6.1.3 Changing the reference category . . . . .	114
6.1.4 Parameterization of categorical covariates . . . . .	115
6.1.5 Linear regression with no intercept . . . . .	116
6.1.6 Linear regression with interactions . . . . .	117
6.1.7 One-way analysis of variance . . . . .	117
6.1.8 Analysis of variance with two or more factors . . . . .	117
6.2 Tests, contrasts, and linear functions of parameters . . . . .	118
6.2.1 Joint null hypotheses: several parameters equal 0 . . . . .	118
6.2.2 Joint null hypotheses: sum of parameters . . . . .	118
6.2.3 Tests of equality of parameters . . . . .	119
6.2.4 Multiple comparisons . . . . .	119
6.2.5 Linear combinations of parameters . . . . .	120
6.3 Model diagnostics . . . . .	120
6.3.1 Predicted values . . . . .	120
6.3.2 Residuals . . . . .	121
6.3.3 Standardized and Studentized residuals . . . . .	121
6.3.4 Leverage . . . . .	122
6.3.5 Cook's D . . . . .	122
6.3.6 DFFITS . . . . .	123
6.3.7 Diagnostic plots . . . . .	123
6.3.8 Heteroscedasticity tests . . . . .	124
6.4 Model parameters and results . . . . .	124
6.4.1 Parameter estimates . . . . .	124
6.4.2 Standardized regression coefficients . . . . .	124
6.4.3 Standard errors of parameter estimates . . . . .	125
6.4.4 Confidence interval for parameter estimates . . . . .	125
6.4.5 Confidence limits for the mean . . . . .	125
6.4.6 Prediction limits . . . . .	126
6.4.7 R-squared . . . . .	127
6.4.8 Design and information matrix . . . . .	127
6.4.9 Covariance matrix of parameter estimates . . . . .	127
6.4.10 Correlation matrix of parameter estimates . . . . .	128
6.5 Further resources . . . . .	128
6.6 Examples . . . . .	128
6.6.1 Scatterplot with smooth fit . . . . .	129
6.6.2 Linear regression with interaction . . . . .	130
6.6.3 Regression diagnostics . . . . .	135
6.6.4 Fitting the regression model separately for each value of another variable . . . . .	138
6.6.5 Two-way ANOVA . . . . .	139
6.6.6 Multiple comparisons . . . . .	144
6.6.7 Contrasts . . . . .	146

<b>7 Regression generalizations and modeling</b>	<b>149</b>
7.1 Generalized linear models . . . . .	149
7.1.1 Logistic regression model . . . . .	149
7.1.2 Conditional logistic regression model . . . . .	151
7.1.3 Exact logistic regression . . . . .	152
7.1.4 Ordered logistic model . . . . .	152
7.1.5 Generalized logistic model . . . . .	152
7.1.6 Poisson model . . . . .	153
7.1.7 Negative binomial model . . . . .	153
7.1.8 Log-linear model . . . . .	153
7.2 Further generalizations . . . . .	154
7.2.1 Zero-inflated Poisson model . . . . .	154
7.2.2 Zero-inflated negative binomial model . . . . .	154
7.2.3 Generalized additive model . . . . .	155
7.2.4 Nonlinear least squares model . . . . .	155
7.3 Robust methods . . . . .	156
7.3.1 Quantile regression model . . . . .	156
7.3.2 Robust regression model . . . . .	156
7.3.3 Ridge regression model . . . . .	156
7.4 Models for correlated data . . . . .	157
7.4.1 Linear models with correlated outcomes . . . . .	157
7.4.2 Linear mixed models with random intercepts . . . . .	158
7.4.3 Linear mixed models with random slopes . . . . .	158
7.4.4 More complex random coefficient models . . . . .	159
7.4.5 Multilevel models . . . . .	160
7.4.6 Generalized linear models with correlated outcomes . . . . .	160
7.4.7 Generalized linear mixed models . . . . .	161
7.4.8 Generalized estimating equations . . . . .	161
7.4.9 MANOVA . . . . .	162
7.4.10 Time series model . . . . .	162
7.5 Survival analysis . . . . .	163
7.5.1 Proportional hazards (Cox) regression model . . . . .	163
7.5.2 Proportional hazards (Cox) model with frailty . . . . .	163
7.5.3 Nelson–Aalen estimate of cumulative hazard . . . . .	164
7.5.4 Testing the proportionality of the Cox model . . . . .	164
7.5.5 Cox model with time-varying predictors . . . . .	165
7.6 Multivariate statistics and discriminant procedures . . . . .	166
7.6.1 Cronbach’s $\alpha$ . . . . .	166
7.6.2 Factor analysis . . . . .	166
7.6.3 Recursive partitioning . . . . .	166
7.6.4 Linear discriminant analysis . . . . .	167
7.6.5 Latent class analysis . . . . .	167
7.6.6 Hierarchical clustering . . . . .	168
7.7 Complex survey design . . . . .	168
7.8 Model selection and assessment . . . . .	169
7.8.1 Compare two models . . . . .	169
7.8.2 Log-likelihood . . . . .	170
7.8.3 Akaike Information Criterion (AIC) . . . . .	170
7.8.4 Bayesian Information Criterion (BIC) . . . . .	170
7.8.5 LASSO model . . . . .	171
7.8.6 Hosmer–Lemeshow goodness of fit . . . . .	171

7.8.7	Goodness of fit for count models . . . . .	171
7.9	Further resources . . . . .	172
7.10	Examples . . . . .	172
7.10.1	Logistic regression . . . . .	172
7.10.2	Poisson regression . . . . .	176
7.10.3	Zero-inflated Poisson regression . . . . .	178
7.10.4	Negative binomial regression . . . . .	180
7.10.5	Quantile regression . . . . .	181
7.10.6	Ordered logistic . . . . .	182
7.10.7	Generalized logistic model . . . . .	183
7.10.8	Generalized additive model . . . . .	185
7.10.9	Reshaping a dataset for longitudinal regression . . . . .	187
7.10.10	Linear model for correlated data . . . . .	190
7.10.11	Linear mixed (random slope) model . . . . .	193
7.10.12	Generalized estimating equations . . . . .	197
7.10.13	Generalized linear mixed model . . . . .	199
7.10.14	Cox proportional hazards model . . . . .	200
7.10.15	Cronbach's $\alpha$ . . . . .	201
7.10.16	Factor analysis . . . . .	202
7.10.17	Recursive partitioning . . . . .	205
7.10.18	Linear discriminant analysis . . . . .	206
7.10.19	Hierarchical clustering . . . . .	208
8	A graphical compendium	211
8.1	Univariate plots . . . . .	211
8.1.1	Barplot . . . . .	211
8.1.2	Stem-and-leaf plot . . . . .	212
8.1.3	Dotplot . . . . .	212
8.1.4	Histogram . . . . .	213
8.1.5	Density plots . . . . .	213
8.1.6	Empirical cumulative probability density plot . . . . .	214
8.1.7	Boxplot . . . . .	214
8.1.8	Violin plots . . . . .	215
8.2	Univariate plots by grouping variable . . . . .	215
8.2.1	Side-by-side histograms . . . . .	215
8.2.2	Side-by-side boxplots . . . . .	215
8.2.3	Overlaid density plots . . . . .	216
8.2.4	Bar chart with error bars . . . . .	216
8.3	Bivariate plots . . . . .	217
8.3.1	Scatterplot . . . . .	217
8.3.2	Scatterplot with multiple y values . . . . .	218
8.3.3	Scatterplot with binning . . . . .	219
8.3.4	Transparent overplotting scatterplot . . . . .	219
8.3.5	Bivariate density plot . . . . .	220
8.3.6	Scatterplot with marginal histograms . . . . .	220
8.4	Multivariate plots . . . . .	221
8.4.1	Matrix of scatterplots . . . . .	221
8.4.2	Conditioning plot . . . . .	221
8.4.3	Contour plots . . . . .	222
8.4.4	3-D plots . . . . .	222
8.5	Special purpose plots . . . . .	223

8.5.1	Choropleth maps . . . . .	223
8.5.2	Interaction plots . . . . .	223
8.5.3	Plots for categorical data . . . . .	224
8.5.4	Circular plot . . . . .	224
8.5.5	Plot an arbitrary function . . . . .	224
8.5.6	Normal quantile-quantile plot . . . . .	225
8.5.7	Receiver operating characteristic (ROC) curve . . . . .	225
8.5.8	Plot confidence intervals for the mean . . . . .	226
8.5.9	Plot prediction limits from a simple linear regression . . . . .	226
8.5.10	Plot predicted lines for each value of a variable . . . . .	226
8.5.11	Kaplan–Meier plot . . . . .	227
8.5.12	Hazard function plotting . . . . .	228
8.5.13	Mean-difference plots . . . . .	228
8.6	Further resources . . . . .	230
8.7	Examples . . . . .	230
8.7.1	Scatterplot with multiple axes . . . . .	230
8.7.2	Conditioning plot . . . . .	232
8.7.3	Scatterplot with marginal histograms . . . . .	232
8.7.4	Kaplan–Meier plot . . . . .	234
8.7.5	ROC curve . . . . .	235
8.7.6	Pairs plot . . . . .	236
8.7.7	Visualize correlation matrix . . . . .	238
<b>9</b>	<b>Graphical options and configuration</b>	<b>241</b>
9.1	Adding elements . . . . .	241
9.1.1	Arbitrary straight line . . . . .	242
9.1.2	Plot symbols . . . . .	242
9.1.3	Add points to an existing graphic . . . . .	243
9.1.4	Jitter points . . . . .	243
9.1.5	Regression line fit to points . . . . .	244
9.1.6	Smoothed line . . . . .	244
9.1.7	Normal density . . . . .	245
9.1.8	Marginal rug plot . . . . .	245
9.1.9	Titles . . . . .	246
9.1.10	Footnotes . . . . .	246
9.1.11	Text . . . . .	246
9.1.12	Mathematical symbols . . . . .	247
9.1.13	Arrows and shapes . . . . .	247
9.1.14	Add grid . . . . .	248
9.1.15	Legend . . . . .	248
9.1.16	Identifying and locating points . . . . .	249
9.2	Options and parameters . . . . .	250
9.2.1	Graph size . . . . .	250
9.2.2	Grid of plots per page . . . . .	250
9.2.3	More general page layouts . . . . .	251
9.2.4	Fonts . . . . .	252
9.2.5	Point and text size . . . . .	252
9.2.6	Box around plots . . . . .	252
9.2.7	Size of margins . . . . .	253
9.2.8	Graphical settings . . . . .	253
9.2.9	Axis range and style . . . . .	253

9.2.10	Axis labels, values, and tick marks . . . . .	254
9.2.11	Line styles . . . . .	254
9.2.12	Line widths . . . . .	255
9.2.13	Colors . . . . .	255
9.2.14	Log scale . . . . .	255
9.2.15	Omit axes . . . . .	256
9.3	Saving graphs . . . . .	256
9.3.1	PDF . . . . .	256
9.3.2	Postscript . . . . .	256
9.3.3	RTF . . . . .	257
9.3.4	JPEG . . . . .	258
9.3.5	Windows Metafile (WMF) . . . . .	258
9.3.6	Bitmap image file (BMP) . . . . .	258
9.3.7	Tagged image file format (TIFF) . . . . .	259
9.3.8	Portable Network Graphics (PNG) . . . . .	259
9.3.9	Closing a graphic device . . . . .	260
<b>10</b>	<b>Simulation</b>	<b>261</b>
10.1	Generating data . . . . .	261
10.1.1	Generate categorical data . . . . .	261
10.1.2	Generate data from a logistic regression . . . . .	263
10.1.3	Generate data from a generalized linear mixed model . . . . .	264
10.1.4	Generate correlated binary data . . . . .	267
10.1.5	Generate data from a Cox model . . . . .	269
10.1.6	Sampling from a challenging distribution . . . . .	271
10.2	Simulation applications . . . . .	274
10.2.1	Simulation study of Student's <i>t</i> test . . . . .	274
10.2.2	Diploma (or hat-check) problem . . . . .	276
10.2.3	Monty Hall problem . . . . .	278
10.3	Further resources . . . . .	280
<b>11</b>	<b>Special topics</b>	<b>281</b>
11.1	Processing by group . . . . .	281
11.2	Simulation-based power calculations . . . . .	284
11.3	Reproducible analysis and output . . . . .	287
11.4	Advanced statistical methods . . . . .	290
11.4.1	Bayesian methods . . . . .	290
11.4.2	Propensity scores . . . . .	296
11.4.3	Bootstrapping . . . . .	303
11.4.4	Missing data . . . . .	304
11.4.5	Finite mixture models with concomitant variables . . . . .	311
11.5	Further resources . . . . .	313
<b>12</b>	<b>Case studies</b>	<b>315</b>
12.1	Data management and related tasks . . . . .	315
12.1.1	Finding two closest values in a vector . . . . .	315
12.1.2	Tabulate binomial probabilities . . . . .	317
12.1.3	Calculate and plot a running average . . . . .	318
12.1.4	Create a Fibonacci sequence . . . . .	320
12.2	Read variable format files . . . . .	321
12.3	Plotting maps . . . . .	324

12.3.1	Massachusetts counties, continued . . . . .	324
12.3.2	Bike ride plot . . . . .	325
12.3.3	Choropleth maps . . . . .	327
12.4	Data scraping and visualization . . . . .	329
12.4.1	Scraping data from HTML files . . . . .	330
12.4.2	Reading data with two lines per observation . . . . .	331
12.4.3	Plotting time series data . . . . .	333
12.4.4	URL APIs and truly random numbers . . . . .	334
12.5	Manipulating bigger datasets . . . . .	336
12.6	Constrained optimization: the knapsack problem . . . . .	337
<b>A</b>	<b>Introduction to SAS</b>	<b>341</b>
A.1	Installation . . . . .	341
A.2	Running SAS and a sample session . . . . .	341
A.3	Learning SAS and getting help . . . . .	346
A.4	Fundamental elements of SAS syntax . . . . .	347
A.5	Work process: The cognitive style of SAS . . . . .	349
A.6	Useful SAS background . . . . .	349
A.6.1	Dataset options . . . . .	349
A.6.2	Subsetting . . . . .	350
A.6.3	Formats and informats . . . . .	350
A.7	Output Delivery System . . . . .	351
A.7.1	Saving output as datasets and controlling output . . . . .	351
A.7.2	Output file types and ODS destinations . . . . .	355
A.8	SAS macro variables . . . . .	355
A.9	Miscellanea . . . . .	356
<b>B</b>	<b>Introduction to R and RStudio</b>	<b>357</b>
B.1	Installation . . . . .	358
B.1.1	Installation under Windows . . . . .	358
B.1.2	Installation under Mac OS X . . . . .	359
B.1.3	RStudio . . . . .	359
B.1.4	Other graphical interfaces . . . . .	359
B.2	Running R and sample session . . . . .	360
B.2.1	Replicating examples from the book and sourcing commands . . . . .	361
B.2.2	Batch mode . . . . .	362
B.3	Learning R and getting help . . . . .	362
B.4	Fundamental structures and objects . . . . .	365
B.4.1	Objects and vectors . . . . .	365
B.4.2	Indexing . . . . .	365
B.4.3	Operators . . . . .	366
B.4.4	Lists . . . . .	366
B.4.5	Matrices . . . . .	367
B.4.6	Dataframes . . . . .	367
B.4.7	Attributes and classes . . . . .	369
B.4.8	Options . . . . .	369
B.5	Functions . . . . .	369
B.5.1	Calling functions . . . . .	369
B.5.2	The <code>apply</code> family of functions . . . . .	370
B.6	Add-ons: packages . . . . .	371
B.6.1	Introduction to packages . . . . .	371

B.6.2	CRAN task views . . . . .	372
B.6.3	Installed libraries and packages . . . . .	373
B.6.4	Packages referenced in this book . . . . .	374
B.6.5	Datasets available with R . . . . .	377
B.7	Support and bugs . . . . .	377
<b>C</b>	<b>The HELP study dataset</b>	<b>379</b>
C.1	Background on the HELP study . . . . .	379
C.2	Roadmap to analyses of the HELP dataset . . . . .	379
C.3	Detailed description of the dataset . . . . .	381
<b>R</b>	<b>References</b>	<b>385</b>



# List of Figures

3.1	Comparison of standard normal and $t$ distribution with 1 degree of freedom (df) . . . . .	69
3.2	Descriptive plot of the normal distribution . . . . .	70
5.1	Density plot of depressive symptom scores (CESD) plus superimposed histogram and normal distribution . . . . .	100
5.2	Scatterplot of CESD and MCS for women, with primary substance shown as the plot symbol . . . . .	102
5.3	Graphical display of the table of substance by race/ethnicity . . . . .	106
5.4	Density plot of age by gender . . . . .	111
6.1	Scatterplot of observed values for age and I1 (plus smoothers by substance) . . . . .	130
6.2	SAS table produced with <code>latext</code> destination in <code>ODS</code> . . . . .	134
6.3	Q-Q plot from SAS, default diagnostics from R . . . . .	137
6.4	Empirical density of residuals, with superimposed normal density . . . . .	137
6.5	Interaction plot of CESD as a function of substance group and gender . . . . .	140
6.6	Boxplot of CESD as a function of substance group and gender . . . . .	140
6.7	Pairwise comparisons . . . . .	146
7.1	Scatterplots of smoothed association of PCS with CESD . . . . .	186
7.2	Side-by-side box plots of CESD by treatment and time . . . . .	193
7.3	Recursive partitioning tree from R . . . . .	206
7.4	Graphical display of assignment probabilities or score functions from linear discriminant analysis by actual homeless status . . . . .	209
7.5	Results from hierarchical clustering . . . . .	210
8.1	Plot of InDUC and MCS vs. CESD for female alcohol-involved subjects . . . . .	231
8.2	Association of MCS and CESD, stratified by substance and report of suicidal thoughts . . . . .	233
8.3	Association of MCS and CESD with marginal histograms . . . . .	234
8.4	Kaplan–Meier estimate of time to linkage to primary care by randomization group . . . . .	236
8.5	Receiver operating characteristic curve for the logistical regression model predicting suicidal thoughts using the CESD as a measure of depressive symptoms (sensitivity = true positive rate; 1-specificity = false positive rate) . . . . .	237
8.6	Pairsplot of variables from the HELP dataset . . . . .	238
8.7	Visual display of correlations and associations . . . . .	240
10.1	Plot of true and simulated distributions . . . . .	274

11.1 Sample Markdown input file . . . . .	288
11.2 Formatted output from R Markdown example . . . . .	289
12.1 Running average for Cauchy and $t$ distributions . . . . .	320
12.2 Massachusetts counties . . . . .	324
12.3 Bike plot with map background . . . . .	326
12.4 Choropleth map . . . . .	329
12.5 Sales plot . . . . .	334
12.6 Number of flights departing Bradley airport on Mondays over time . . . . .	338
A.1 SAS Windows interface . . . . .	342
A.2 Running a SAS program . . . . .	343
A.3 Results from <code>proc print</code> . . . . .	344
A.4 Results from <code>proc univariate</code> . . . . .	345
A.5 The SAS window after running the sample session code . . . . .	346
A.6 The SAS Explorer window . . . . .	347
A.7 Opening the on-line help . . . . .	348
A.8 The SAS Help and Documentation window . . . . .	348
B.1 R Windows graphical user interface . . . . .	358
B.2 R Mac OS X graphical user interface . . . . .	359
B.3 RStudio graphical user interface . . . . .	360
B.4 Sample session in R . . . . .	361
B.5 Documentation on the <code>mean()</code> function . . . . .	363
B.6 Display after running <code>RSiteSearch("eta squared anova")</code> . . . . .	364

# List of Tables

3.1 Quantiles, probabilities, and pseudo-random number generation: distributions available in SAS and R . . . . .	54
6.1 Formatted results using the <code>xtable</code> package . . . . .	134
7.1 Generalized linear model distributions supported by SAS and R . . . . .	150
11.1 Bayesian modeling functions available within the <code>MCMCpack</code> package . . . . .	292
12.1 Weights, volume, and values for the knapsack problem . . . . .	337
B.1 CRAN task views . . . . .	373
C.1 Analyses undertaken using the HELP dataset . . . . .	379
C.2 Annotated description of variables in the HELP dataset . . . . .	381



# Preface to the second edition

Software systems evolve, and so do the approaches and expertise of statistical analysts.

After the publication of the first edition of *SAS and R: Data Management, Statistical Analysis, and Graphics*, we began a blog in which we explored many new case studies and applications, ranging from generating a Fibonacci series to fitting finite mixture models with concomitant variables. We also discussed some additions to SAS and new or improved R packages. The blog now has hundreds of entries and (according to Google Analytics) has received hundreds of thousands of visits.

The volume you are holding is nearly 50% longer than the first edition, and much of the new material is adapted from these blog entries, while it also includes other improvements and additions which have emerged in the last few years.

We have extensively reorganized the material in the book and created three new chapters. The first, *Simulation*, includes examples where data are generated from complex models such as mixed effects models and survival models, and from distributions using the Metropolis–Hastings algorithm. We also explore three interesting statistics and probability examples via simulation. The second is *Special topics*, where we describe some key features, such as processing by group, and detail several important areas of statistics, including Bayesian methods, propensity scores, and bootstrapping. The last is *Case studies*, where we demonstrate examples of some data management tasks, read complex files, make and annotate maps, and show how to “scrape” data from web pages.

We also cover some important new tools, including the use of RStudio, a powerful and easy-to-use front end for R that adds innumerable features to R. In our experience, it at least doubles the productivity of R users, and our SAS-using students find it an extremely comfortable interface that bears some similarity to the SAS GUI.

We have added a separate section and examples that describe “reproducible analysis.” This is the notion that code, results, and interpretation should live together in a single place. We used two reproducible analysis systems (SASweave and Sweave) to generate the example code and output in the book. Code extracted from these files is provided on the book web site. In this edition, we provide a detailed discussion of the philosophy and use of these systems. In particular, we feel that the `knitr` and `markdown` packages for R, which are tightly integrated with RStudio, should become a part of every R user’s toolbox. We can’t imagine working on a project without them.

Finally, we’ve reorganized much of the material from the first edition into smaller, more focused chapters. Users will now find separate (and enhanced) chapters on data input and output, data management, statistical and mathematical functions, and programming, rather than a single chapter on “data management.” Graphics are now discussed in two chapters: one on high-level types of plots, such as scatterplots and histograms, and another on customizing the fine details of the plots, such as the number of tick marks and the color of plot symbols.

We’re immensely gratified by the positive response the first edition elicited, and hope the current volume will be as useful to you.

**On the web**

The book website at <http://www.amherst.edu/~nhorton/sasr2> includes the table of contents, the indices, the HELP dataset, example code in SAS and R, a pointer to the blog, and a list of errata.

**Acknowledgments**

In addition to those acknowledged in the first edition, we would like to thank Kathryn Aloisio, Gregory Call, J.J. Allaire and the RStudio developers, plus the many individuals who have created and shared R packages or SAS macros. Their contributions to SAS, R, or L<sup>A</sup>T<sub>E</sub>X programming efforts, comments, guidance, and/or helpful suggestions on drafts of the revision have been extremely helpful. Above all we greatly appreciate Sara and Julia as well as Abby, Alana, Kinari, and Sam, for their patience and support.

*Amherst, MA  
March 16, 2014*

# Preface to the first edition

SAS<sup>TM</sup> (SAS Institute [153]) and R (R development core team [135]) are two statistical software packages used in many fields of research. SAS is commercial software developed by SAS Institute; it includes well-validated statistical algorithms. It can be licensed but not purchased. Paying for a license entitles the licensee to professional customer support. However, licensing is expensive and SAS sometimes incorporates new statistical methods only after a significant lag. In contrast, R is free, open-source software, developed by a large group of people, many of whom are volunteers. It has a large and growing user and developer base. Methodologists often release applications for general use in R shortly after they have been introduced into the literature. Professional customer support is not provided, though there are many resources for users. There are settings in which one of these useful tools is needed, and users who have spent many hours gaining expertise in the other often find it frustrating to make the transition.

We have written this book as a reference text for users of SAS and R. Our primary goal is to provide users with an easy way to learn how to perform an analytic task in both systems, without having to navigate through the extensive, idiosyncratic, and sometimes (often?) unwieldy documentation each provides. We expect the book to function in the same way that an English–French dictionary informs users of both the equivalent nouns and verbs in the two languages as well as the differences in grammar. We include many common tasks, including data management, descriptive summaries, inferential procedures, regression analysis, multivariate methods, and the creation of graphics. We also show some more complex applications. In toto, we hope that the text will allow easier mobility between systems for users of any statistical system.

We do not attempt to exhaustively detail all possible ways available to accomplish a given task in each system. Neither do we claim to provide the most elegant solution. We have tried to provide a simple approach that is easy to understand for a new user, and have supplied several solutions when they seem likely to be helpful. Carrying forward the analogy to an English–French dictionary, we suggest language that will communicate the point effectively, without listing every synonym or providing guidance on native idiom or eloquence.

## Who should use this book

Those with an understanding of statistics at the level of multiple-regression analysis will find this book helpful. This group includes professional analysts who use statistical packages almost every day as well as statisticians, epidemiologists, economists, engineers, physicians, sociologists, and others engaged in research or data analysis. We anticipate that this tool will be particularly useful for sophisticated users, those with years of experience in only one system, who need or want to use the other system. However, intermediate-level analysts should reap the same benefit. In addition, the book will bolster the analytic abilities of a relatively new user of either system, by providing a concise reference manual and annotated examples executed in both packages.

## **Using the book**

The book has three indices, in addition to the comprehensive Table of Contents. These include: 1) a detailed topic (subject) index in English; 2) a SAS index, organized by SAS syntax; and 3) an R index, describing R syntax. SAS users can use the SAS index to look up a task for which they know the SAS code and turn to a page with that code as well as the associated R code to carry out that task. R users can use the dictionary in an analogous fashion using the R index.

Extensive example analyses are presented; see Table C.1 for a comprehensive list. These employ a single dataset (from the HELP study), described in Appendix C. Readers are encouraged to download the dataset and code from the book website. The examples demonstrate the code in action and facilitate exploration by the reader.

## **Differences between SAS and R**

SAS and R are so fundamentally distinct that an enumeration of their differences would be counterproductive. However, some differences are important for new users to bear in mind.

SAS includes data management tools that are primarily intended to prepare data for analysis. After preparation, analysis is performed in a distinct step, the implementation of which effectively cannot be changed by the user, though often extensive options are available. R is a programming environment tailored for data analysis. Data management and analysis are integrated. This means, for example, that calculating body mass index (BMI) from weight and height can be treated as a function of the data, and as such is as likely to appear within a data analysis as in making a “new” piece of data to keep.

SAS Institute makes decisions about how to change the software or expand the scope of included analyses. These decisions are based on the needs of the user community and on corporate goals for profitability. For example, when changes are made, backwards compatibility is almost always maintained, and documentation of exceptions is extensive. SAS Institute’s corporate conservatism means that techniques are sometimes not included in SAS until they have been discussed in the peer-reviewed literature for many years. While the R core team controls base functionality, a very large number of users have developed functions for R. Methodologists often release R functions to implement their work concurrently with publication. While this provides great flexibility, it comes at some cost. A user-contributed function may implement a desired methodology, but code quality may be unknown, documentation scarce, and paid support nonexistent. Sometimes a function which once worked may become defunct due to a lack of backwards compatibility and/or the author’s inability to, or lack of interest in, updating it.

Other differences between SAS and R are worth noting. Data management in SAS is undertaken using row by row (observation-level) operations. R is inherently a vector-based language, where columns (variables) are manipulated. R is case sensitive, while SAS is generally not.

## **Where to begin**

We do not anticipate that the book will be read cover to cover. Instead, we hope that the extensive indexing, cross-referencing, and worked examples will make it possible for readers to directly find and then implement what they need. A user new to either SAS or R should begin by reading the appropriate appendix for that software package, which includes a sample session and overview.

**On the web**

The book website includes the Table of Contents, the indices, the HELP dataset, example code in SAS and R, and a list of errata.

**Acknowledgments**

We would like to thank Rob Calver, Shashi Kumar, and Sarah Morris for their support and guidance at Informa CRC/Chapman and Hall, the Department of Statistics at the University of Auckland for graciously hosting NH during a sabbatical leave, and the Office of the Provost at Smith College. We also thank Allyson Abrams, Tanya Hakim, Ross Ihaka, Albyn Jones, Russell Lenth, Brian McArdle, Paul Murrell, Alastair Scott, David Schoenfeld, Duncan Temple Lang, Kristin Tyler, Chris Wild, and Alan Zaslavsky for contributions to SAS, R, or L<sup>A</sup>T<sub>E</sub>X programming efforts, comments, guidance, and/or helpful suggestions on drafts of the manuscript.

Above all we greatly appreciate Sara and Julia as well as Abby, Alana, Kinari, and Sam, for their patience and support.

*Amherst, MA and Northampton, MA*

*March 2009*



# Chapter 1

## Data input and output

This chapter reviews data input and output, including reading and writing files in spreadsheet, ASCII file, native, and foreign formats.

### 1.1 Input

Both SAS and R provide comprehensive support for data input and output. In this section we address aspects of these tasks.

SAS native datasets are rectangular files with data stored in a special format. They have the form `filename.sas7bdat` or something similar, depending on version. In the following, we assume that files are stored in directories and that the locations of the directories in the operating system can be labeled using Windows syntax (though SAS allows UNIX/Linux/Mac OS X-style forward slash as a directory delimiter on Windows). Other operating systems will use local idioms in describing locations.

R organizes data in dataframes (B.4.6), or connected series of rectangular arrays, which can be saved as platform independent objects. R also allows UNIX-style directory delimiters (forward slash) on Windows.

#### 1.1.1 Native dataset

*Example: 7.10*

SAS

```
libname libref "dir_location";
data ds;
  set libref.sasfilename;    /* Note: no file extension */
  ...
run;
or
data ds;
  set "dir_location\sasfilename.sas7bdat"; /* Windows only */
  set "dir_location/sasfilename.sas7bdat";
  /* works on all OS including Windows */
  ...
run;
```

*Note:* The file `sasfilename.sas7bdat` is created by using a `libref` in a `data` statement; see 1.2.3.

**R**

```
load(file="dir_location/savedfile")    # works on all OS including Windows
load(file="dir_location\\savedfile")   # Windows only
```

*Note:* Forward slash is supported as a directory delimiter on all operating systems; a double backslash is supported under Windows. The file `savedfile` is created by `save()` (see 1.2.3). Running the command `print(load(file="dir_location/savedfile"))` will display the objects that are added to the workspace.

### 1.1.2 Fixed format text files

See 1.1.4 (read more complex fixed files) and 12.2 (read variable format files).

**SAS**

```
data ds;
  infile 'C:\file_location\filename.ext';
  input varname1 ... varnamek;
run;
or
filename filehandle 'file_location/filename.ext';

proc import datafile=filehandle
  out=ds dbms=dlm;
  getnames=yes;
run;
```

*Note:* The `infile` approach allows the user to limit the number of rows read from the data file using the `obs` option. Character variables are noted with a trailing ‘\$’, e.g., use a statement such as `input varname1 varname2 $ varname3` if the second position contains a character variable (see 1.1.4 for examples). The `input` statement allows many options and can be used to read files with variable format (12.2).

In `proc import`, the `getnames=yes` statement is used if the first row of the input file contains variable names (the variable types are detected from the data). If the first row does not contain variable names then the `getnames=no` option should be specified. The `guessingrows` option (not shown) will base the variable formats on other than the default 20 rows. The `proc import` statement will accept an explicit file location rather than a file associated by the `filename` statement as in 7.10.

Note that in Windows installations, SAS accepts either slashes or backslashes to denote directory structures. For Linux, only forward slashes are allowed. Behavior in other operating systems may vary.

In addition to these methods, files can be read by selecting the `Import Data` option on the `file` menu in the GUI.

**R**

```
ds = read.table("dir_location\\file.txt", header=TRUE) # Windows only
or
ds = read.table("dir_location/file.txt", header=TRUE)  # all OS (including
                                                       # Windows)
```

*Note:* Forward slash is supported as a directory delimiter on all operating systems; a double backslash is supported under Windows. If the first row of the file includes the name of the variables, these entries will be used to create appropriate names (reserved characters such as ‘\$’ or '[' are changed to '.') for each of the columns in the dataset. If the first row doesn't include the names, the `header` option can be left off (or set to `FALSE`), and the variables will be called `V1`, `V2`, ... `Vn`. A limit on the number of lines to be read can be specified

through the `nrows` option. The `read.table()` function can support reading from a URL as a filename (see 1.1.12) or browse files interactively using `file.choose()` (see 4.3.7).

### 1.1.3 Other fixed files

See 1.1.4 (read more complex fixed files) and 12.2 (read variable format files).

Sometimes data arrives in files that are very irregular in shape. For example, there may be a variable number of fields per line, or some data in the line may describe the remainder of the line. In such cases, a useful generic approach is to read each line into a single character variable, then use character variable functions (see 2.2) to extract the contents.

#### SAS

```
data ds;
data filenames;
  infile "file_location/file.txt";
  input var1 $32767. ;
run;
```

*Note:* The `$32767.` allows input lines as long as 32,767 characters, the maximum length of SAS character variables.

#### R

```
ds = readLines("file.txt")
or
ds = scan("file.txt")
```

*Note:* The `readLines()` function returns a character vector with length equal to the number of lines read (see `file()`). A limit on the number of lines to be read can be specified through the `nrows` option. The `scan()` function returns a vector, with entries separated by white space by default. These functions read by default from standard input (see `stdin()` and `?connections`), but can also read from a file or URL (see 1.1.12). The `read.fwf()` function may also be useful for reading fixed width files. The `capture.output()` function can be used to send output to a character string or file (see also `sink()`).

### 1.1.4 Reading more complex text files

See 1.1.2 (read fixed files) and 12.2 (read variable format files).

Text data files often contain data in special formats. One common example is date variables. Special values can be read in using informats (A.6.3). As an example below we consider the following data.

```
1 AGKE 08/03/1999 $10.49
2 SBKE 12/18/2002 $11.00
3 SEKK 10/23/1995 $5.00
```

#### SAS

```
data ds;
  infile 'C:\file_location\filename.dat';
  input id initials $ datevar mmddyy10. cost dollar7.4;
run;
```

*Note:* The SAS informats (A.6.3) denoted by the `mmddyy10.` and `dollar7.4` inform the `input` statement that the third and fourth variables have special forms and should not be

treated as numbers or letters, but read and interpreted according to the rules specified. In the case of `datevar`, SAS reads the date appropriately and stores a SAS date value (A.6.3). For `cost`, SAS ignores the ‘\$’ in the data and would also ignore commas, if they were present. The `input` statement allows many options for additional data formats and can be used to read files with variable format (12.2).

Other common features of text data files include very long lines and missing data. These are addressed through the `infile` or `filename` statements. Missing data may require the `missover` option to the `infile` statement as well as listing the columns in which variables appear in the dataset in the `input` statement. Long lines (many columns in the data file) may require the `lrecl` option to the `infile` or `filename` statement. For a thorough discussion, see the on-line help: Contents; SAS Products; Base SAS; SAS Language Reference: Concepts; DATA Step Concepts; Reading Raw Data; Reading Raw Data with the INPUT statement.

## R

```
tmpds = read.table("file_location/filename.dat")
id = tmpds$V1
initials = tmpds$V2
datevar = as.Date(as.character(tmpds$V3), "%m/%d/%Y")
cost = as.numeric(substr(tmpds$V4, 2, 100))
ds = data.frame(id, initials, datevar, cost)
rm(tmpds, id, initials, datevar, cost)
```

*Note:* In R, this task is accomplished by first reading the dataset (with default names from `read.table()` denoted V1 through V4). These objects can be manipulated using `as.character()` to undo the default coding as factor variables, and coerced to the appropriate data types. For the `cost` variable, the dollar signs are removed using the `substr()` function. Finally, the individual variables are gathered together as a dataframe.

### 1.1.5 Comma separated value (CSV) files

*Example: 2.6.1*

#### SAS

```
data ds;
  infile 'dir_location\filename.csv' delimiter=',';
  input varname1 ... varnamek;
run;
or
proc import datafile='dir_location\full_filename'
  out=ds dbms=csv;
  delimiter=',';
  getnames=yes;
run;
```

*Note:* Character variables are noted with a trailing ‘\$’, e.g., use a statement such as `input varname1 varname2 $ varname3` if the second column contains characters. The `proc import` syntax allows for the first row of the input file to contain variable names, with variable types detected from the data. If the first row does not contain variable names then use `getnames=no`.

In addition to these methods, files can be read by selecting the `Import Data` option on the `file` menu in the GUI.

**R**

```
ds = read.csv("dir_location/file.csv")
```

*Note:* The `stringsAsFactors` option can be set to prevent automatic creation of factors for categorical variables. A limit on the number of lines to be read can be specified through the `nrows` option. The command `read.csv(file.choose())` can be used to browse files interactively (see 4.3.7). The comma-separated file can be given as a URL (see 1.1.12). The `colClasses` option can be used to speed up reading large files.

**1.1.6 Read sheets from an Excel file****SAS**

```
proc import out=ds
  datafile="dir_location\full_filename" dbms=excel replace;
  range="sheetname$a1:zz4000";
  getnames=yes; mixed=no; usedate=yes; scantext=yes;
run;
```

*Note:* The `range` option specifies the sheet name and cells to select within the Excel workbook. The \$ after the sheet name indicates that a range of cells follows; without it the entire sheet is read. The `a1:zz4000` gives the upper left and lower right cells of the region to be read, separated by a colon. The `getnames` option indicates whether the names are included in the first row. If `mixed=yes` (default is `no`) then numeric values are converted to character if any values are character. If `usedate=yes` then Excel date values are converted to SAS date values. If `scantext=yes` then SAS checks for the longest character value in the Excel data and sets the SAS character value length accordingly. Note that the `dbms` option also accepts the values `excelcs` and `xls`, either of which may be helpful in some settings. Documentation is found in SAS Products; SAS/ACCESS; SAS/ACCESS Interface to PC files: Reference; Import and Export Wizards and Procedures; File Format-Specific Reference for the IMPORT and EXPORT Procedures.

**R**

```
library(gdata)
ds = read.xls("http://www.amherst.edu/~nhorton/sasr2/datasets/help.xlsx",
  sheet=1)
```

*Note:* In this implementation, the sheet number is provided, rather than name.

**1.1.7 Read data from R into SAS**

The R package `foreign` includes the `write.dbf()` function; we recommend this as a reliable format for extracting data from R into a SAS-ready file, though other options are possible. Then SAS proc import can easily read the DBF file. Because we describe moving from R to SAS, we begin with the R entry.

**R**

```
tosas = data.frame(ds)
library(foreign)
write.dbf(tosas, "dir_location/tosas.dbf")
```

**SAS**

```
proc import datafile="dir_location\tosas.dbf"
  out=fromr dbms=dbf;
run;
```

### 1.1.8 Read data from SAS into R

#### SAS

```
proc export data=ds
  outfile = "dir_location\to_r.dbf" dbms=dbf;
run;
```

#### R

```
library(foreign)
ds = read.dbf("dir_location/to_r.dbf")
or
library(sas7bdat)
helpfromSAS = read.sas7bdat("dir_location/help.sas7bdat")
```

*Note:* The first set of code (obviously) requires a working version of SAS. The second can be used with any SAS formatted data set; it is based on reverse-engineering of the SAS data set format, which SAS has not made public.

### 1.1.9 Reading datasets in other formats

*Example: 6.6.1*

#### SAS

```
libname ref spss 'filename.sav'; /* SPSS */
libname ref bmdp 'filename.dat'; /* BMDP */
libname ref v6 'filename.ssd01'; /* SAS vers. 6 */
libname ref xport 'filename.xpt'; /* SAS export */
libname ref xml 'filename.xml'; /* XML */
```

```
data ds;
set ref.filename;
run;
or
```

```
proc import datafile="filename.ext" out=ds
  dbms=excel; /* Excel */
run;
```

```
... dbms=access; ... /* Access */
... dbms=dta; ... /* Stata */
```

*Note:* The `libname` statements above refer to files, rather than directories. The extensions shown above are those typically used for these file types, but in any event the full name of the file, including the extension, is needed in the `libname` statement. In contrast, only the file name (without the extension) is used in the `set` statement. The data type options specified above in the `libname` statement and `dbms` option are available in most operating systems. To see what data types are available, check the on-line help. For Windows: Contents, Using SAS Software in Your Operating Environment, SAS Companion for Windows, Features of the SAS language for Windows, SAS Statements under Windows, LIBNAME statement.

In addition to these methods, files can be read by selecting the `Import Data` option on the `file` menu in the GUI.

**R**

```
library(foreign)
ds = read.dbf("filename.dbf")           # DBase
ds = read.epiinfo("filename.epiinfo")    # Epi Info
ds = read.mtp("filename.mtp")           # Minitab portable worksheet
ds = read.octave("filename.octave")      # Octave
ds = read.ssd("filename.ssd")           # SAS version 6
ds = read.xport("filename.xport")        # SAS XPORT file
ds = read.spss("filename.sav")           # SPSS
ds = read.dta("filename.dta")            # Stata
ds = read.systat("filename.sys")         # Systat
```

*Note:* The `foreign` package can read Stata, Epi Info, Minitab, Octave, SPSS, and Systat files (with the caveat that SAS files may be platform dependent). The `read.ssd()` function will only work if SAS is installed on the local machine.

### 1.1.10 Reading data with a variable number of words in a field

Reading data in a complex data format will generally require a tailored approach. Here we give a relatively simple example and outline the key tools useful for reading in data in complex formats. Suppose we have data as follows:

```
1 Las Vegas, NV --- 53.3 --- --- 1
2 Sacramento, CA --- 42.3 --- --- 2
3 Miami, FL --- 41.8 --- --- 3
4 Tucson, AZ --- 41.7 --- --- 4
5 Cleveland, OH --- 38.3 --- --- 5
6 Cincinnati, OH 15 36.4 --- --- 6
7 Colorado Springs, CO --- 36.1 --- --- 7
8 Memphis, TN --- 35.3 --- --- 8
8 New Orleans, LA --- 35.3 --- --- 8
10 Mesa, AZ --- 34.7 --- --- 10
11 Baltimore, MD --- 33.2 --- --- 11
12 Philadelphia, PA --- 31.7 --- --- 12
13 Salt Lake City, UT --- 31.9 17 --- 13
```

The `---` means that the value is missing. Note two complexities here. First, fields are delimited by both spaces and commas, where the latter separates the city from the state. Second, cities may have names consisting of more than one word.

**SAS**

```
data ds;
  infile "dir_location/cities.txt" dlm=", ";
  input id city & $20. state $2. v1 - v5;
run;
```

*Note:* The `infile` and `input` statements in the data step can accommodate many features of text files. The `dlm=", "` tells SAS that both commas and spaces are delimiters in this file. In the `input` statement, the instruction `city & $20.` is parsed as: read up to 20 characters, and within that distance, spaces should not be considered delimiters. In this example, the `---` are interpreted by SAS as “invalid data” but are recorded in the `ds` data set as missing values.

Full details on these two key data step statements can be found in the on-line help: SAS Products; Base SAS; SAS Statements: Reference; Dictionary of SAS Statements.

**R**

```

readcities = function(thisline) {
  thislen = length(thisline)
  id = as.numeric(thisline[1])
  v1 = as.numeric(thisline[thislen-4])
  v2 = as.numeric(thisline[thislen-3])
  v3 = as.numeric(thisline[thislen-2])
  v4 = as.numeric(thisline[thislen-1])
  v5 = as.numeric(thisline[thislen])
  city = paste(thisline[2:(thislen-5)], collapse=" ")
  return(list(id=id,city=city,v1=v1,v2=v2,v3=v3,v4=v4,v5=v5))
}
file
  = readLines("http://www.amherst.edu/~nhorton/sasr2/datasets/cities.txt")
split = strsplit(file, " ")  # split up fields for each line
as.data.frame(t(sapply(split, readcities)))

```

*Note:* In R, we first write a function that processes a line and converts each field other than the city name into a numeric variable. The function works backwards from the end of the line to find the appropriate elements, then calculates what is left over to store in the city variable. We need each line to be converted into a character vector containing each “word” (character strings divided by spaces) as a separate element. We’ll do this by first reading each line, then splitting it into words. This results in a list object, where the items in the list are the vectors of words. Then we can call the `readcities()` function for each vector using an invocation of `sapply()` (B.5.2), which avoids use of a `for` loop. The resulting object is transposed then coerced into a dataframe (see also `count.fields()`).

### 1.1.11 Read a file byte by byte

It may be necessary to read data that is not stored in ASCII (or other text) format. At such times, it may be useful to read the raw bytes stored in the file.

**SAS**

```

data test;
  infile "dir_location/full_filename" recfm=n;
  input byte ib1. @@;
run

```

*Note:* The `recfm=n` option tells SAS to read the file in binary; note that this may differ by OS. The `ib1.` informat tells SAS to read one byte. The `@@` tells SAS to hold this line of input, rather than skipping to a new line, when data is read. A new line will be begun only when the current line is finished. SAS will read bytes until there are no more to read. Other tools can then be used to assemble the bytes into usable data.

**R**

```

finfo = file.info("full_filename")
toread = file("full_filename", "rb")
alldata = readBin(toread, integer(), size=1, n=finfo$size, endian="little")

```

*Note:* In R, the `readBin()` function is used to read the file, after some initial prep work. The function requires we input the number of data elements to read. An overestimate is OK, but we can easily find the exact length of the file using the `file.info()` function; the resulting object has a `size` constituent with the number of bytes. We’ll also need a *connection* to the file, which is established in a call to the `file()` function. The `size` option gives the length

of the elements, in bytes, and the  `endian` option helps describe how the bytes should be read.

### 1.1.12 Access data from a URL

SAS

```
filename myurl url "https://example.com/file.txt";
proc import datafile=myurl out=ds dbms=filetype;
run;
```

*Note:* If the URL requires a username and password, the `filename` statement accepts `user=` and `pass=` options. The url “handle”, here `myurl`, can be no longer than 8 characters. The url handle can be used in an `import` procedure as shown, or with an `infile` statement in a `data` step (see 12.2). The `import` procedure supports many `filetypes`, as shown in 1.1.2, 1.1.5, 1.1.6, 1.1.7, and 1.1.9.

R

```
library(RCurl)
myurl = getURL("https://example.com/file.txt")
ds = readLines(textConnection(myurl))
```

*Note:* The `readLines()` function reads arbitrary text, while `read.table()` can be used to read a file with cases corresponding to lines and variables to fields in the file (the `header` option sets variable names to entries in the first line). To read Hypertext Transfer Protocol Secure (https) URLs, the `getURL()` function from the `RCurl` package is needed in conjunction with the `textConnection()` function (see also `url()`). Access through proxy servers as well as specification of username and passwords is provided by the function `download.file()`. A limit on the number of lines to be read can be specified through the `nrows` option.

### 1.1.13 Read an XML-formatted file

A sample (flat) XML form of the HELP dataset can be found at <http://www.amherst.edu/~nhorton/sasr2/datasets/help.xml>. The first ten lines of the file consist of:

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<TABLE>
  <HELP>
    <id> 1 </id>
    <e2b1 Missing=". " />
    <g1b1> 0 </g1b1>
    <i11 Missing=". " />
    <pcs1> 54.2258263 </pcs1>
    <mcs1> 52.2347984 </mcs1>
    <cesd1> 7 </cesd1>
```

Here we consider reading simple files of this form. While support is available for reading more complex types of XML files, these typically require considerable additional sophistication.

**SAS**

```
libname ref xml 'dir_location\filename.xml';

data ds;
  set ref.filename_without_extension;
run;
```

*Note:* The `libname` statement above refers to a file name, rather than a directory name. The “xml” extension is typically used for this file type, but in any event the full name of the file, including the extension, is needed.

**R**

```
library(XML)
urlstring = "http://www.amherst.edu/~nhorton/sasr2/datasets/help.xml"
doc = xmlRoot(xmlTreeParse(urlstring))
tmp = xmlSApply(doc, function(x) xmlSApply(x, xmlValue))
ds = t(tmp)[,-1]
```

*Note:* The `XML` package provides support for reading XML files. The `xmlRoot()` function opens a connection to the file, while `xmlSApply()` and `xmlValue()` are called recursively to process the file. The returned object is a character matrix with columns corresponding to observations and rows corresponding to variables, which in this example are then transposed (see also `readHTMLTable()`).

### 1.1.14 Manual data entry

**SAS**

```
data ds;
input x1 x2;
cards;
1 2
1 3
1.4 2
123 4.5
;
run;
```

*Note:* The above code demonstrates reading data into a SAS dataset within a SAS program. The semicolon following the data terminates the `data` step, meaning that a `run` statement is not actually required. The `input` statement used above employs the syntax discussed in 1.1.2. In addition to this option for entering data within SAS, there is a GUI-based data entry/editing tool called the Table Editor. It can be accessed using the mouse through the Tools menu, or by using the `viewtable` command on the SAS command line.

**R**

```
x = numeric(10)
data.entry(x)
or
x1 = c(1, 1, 1.4, 123)
x2 = c(2, 3, 2, 4.5)
```

*Note:* The `data.entry()` function invokes a spreadsheet that can be used to edit or otherwise change a vector or dataframe. In this example, an empty numeric vector of length 10 is created to be populated. The `data.entry()` function differs from the `edit()` function,

which leaves the objects given as argument unchanged, returning a new object with the desired edits (see also the `fix()` function).

## 1.2 Output

### 1.2.1 Displaying data

*Example: 6.6.2*

See 2.1.3 (values of variables in a dataset).

#### SAS

```
title1 'Display of variables';
footnote1 'A footnote';
proc print data=ds;
  var x1 x3 xk x2;
  format x3 dollar10.2;
run;
```

*Note:* For `proc print` the `var` statement selects variables to be included. The `format` statement, as demonstrated, can alter the appearance of the data; here  $x_3$  is displayed as a dollar amount with 10 total digits, two of them to the right of the decimal. The keyword `_numeric_` can replace the variable name and will cause all of the numerical variables to be displayed in the same format. See A.6.3 for further discussion, as well as A.6.2, 11.1, and A.6.1 for ways to limit which observations are displayed. The `var` statement, as demonstrated, ensures the variables are displayed in the desired order. The `title` and `footnote` statements and related statements `title1`, `footnote2`, etc., allow headers and footers to be added to each output page. Specifying the command with no argument will remove the title or footnote from subsequent output.

SAS also provides `proc report` and `proc tabulate` to create more customized output.

#### R

```
dollarcents = function(x)
  return(paste("$", format(round(x*100, 0)/100, nsmall=2), sep=""))
data.frame(x1, dollarcents(x3), xk, x2)
```

or

```
ds[,c("x1", "x3", "xk", "x2")]
```

*Note:* A function can be defined to format a vector as U.S. dollar and cents by using the `round()` function (see 3.2.4) to control the number of digits (2) to the right of the decimal. Alternatively, named variables from a dataframe can be printed. The `cat()` function can be used to concatenate values and display them on the console (or route them to a file using the `file` option). More control on the appearance of printed values is available through use of `format()` (control of digits and justification), `sprintf()` (use of C-style string formatting) and `prettyNum()` (another routine to format using C-style specifications).

### 1.2.2 Number of digits to display

*Example: 2.6.1*

SAS lacks an option to control how many significant digits are displayed in procedure output, in general (an exception is `proc means`). For reporting purposes, one should save the output as a dataset using `ODS`, then use the `format` statement (1.2.1, A.6.3) with `proc print` to display the desired precision, as demonstrated in 6.6.2.

**R**

```
options(digits=n)
```

*Note:* The `options(digits=n)` command can be used to change the default number of decimal places to display in subsequent R output. To affect the actual significant digits in the data, use the `round()` function (see 3.2.4).

**1.2.3 Save a native dataset****SAS**

```
libname libref "dir_location";  
  
data libref.sasfilename;  
set ds;  
run;
```

*Example: 2.6.1*

*Note:* A SAS dataset can be read back into SAS using a `set` statement with a `libref` (see 1.1.1).

**R**

```
save(robject, file="savedfile")
```

*Note:* An object (typically a dataframe or a list of objects) can be read back into R using `load()` (see 1.1.1).

**1.2.4 Creating datasets in text format****SAS**

```
proc export data=ds outfile='file_location_and_name'  
    dbms=csv;           /* comma-separated values */  
  
...dbms=tab;           /* tab-separated values */  
...dbms=dlm;           /* arbitrary delimiter; default is space,  
                      others with delimiter= statement */
```

**R**

```
write.csv(ds, file="full_file_location_and_name")
```

or

```
library(foreign)  
write.table(ds, file="full_file_location_and_name")
```

*Note:* The `sep` option to `write.table()` can be used to change the default delimiter (space) to an arbitrary value.

**1.2.5 Creating Excel spreadsheets****SAS**

```
data help;  
set "c:\book\help.sas7bdat";  
run;  
  
proc export data=ds outfile="dir_location/filename.xls"  
    dbms=excel;  
run;
```

or

```
proc export data=ds
  outfile = "dir_location/filename.xls" dbms=excel;
  sheet="sheetname";
run;
```

*Note:* The latter code demonstrates adding a sheet to an existing Excel workbook. Documentation can be found at SAS Products; SAS/ACCESS; SAS/ACCESS Interface to PC files: Reference; Import and Export Wizards and Procedures; File Format-Specific Reference for the IMPORT and EXPORT Procedures.

There are several other methods for doing this in SAS. A possibly simpler way would be to use the `libname` statement, but platform dependence makes this method less desirable.

## R

```
library(WriteXLS)
HELP = read.csv("http://www.amherst.edu/~nhorton/sasr2/datasets/help.csv")
WriteXLS("HELP", ExcelFileName="newhelp.xls")
```

*Note:* The `WriteXLS` package provides this functionality. It uses Perl (Practical extraction and report language, <http://www.perl.org>) and requires an external installation of Perl to function. After installing Perl, this requires running the operating system command `cpan -i Text::CSV_XS` at the command line.

## 1.2.6 Creating files for use by other packages

*Example: 2.6.1*

See also 1.2.8 (write XML).

## SAS

```
libname ref spss 'filename.sav'; /* SPSS */
libname ref bmdp 'filename.dat'; /* BMDP */
libname ref v6 'filename.ssd01'; /* SAS version 6 */
libname ref xport 'filename.xpt'; /* SAS export */
libname ref xml 'filename.xml'; /* XML */

data ref.filename_without_extension;
set ds;
or
proc export data=ds outfile='file_location_and_name'
  dbms=csv; /* comma-separated values */

...dbms=dbf; /* dbase 5,IV,III */
...dbms=excel; /* excel */
...dbms=dta; /* Stata */
...dbms=tab; /* tab-separated values */
...dbms=access; /* Access table */
...dbms=dlm; /* arbitrary delimiter; default is space,
  others with delimiter=char statement */
```

*Note:* The `libname` statements above refer to file names, rather than directory names. The extensions shown above are those conventionally used but the option specification determines the file type that is created. Some of the above rely on the SAS/ACCESS product. See on-line help: SAS Products; SAS/ACCESS; SAS/ACCESS Interface to PC files: Reference; Import and Export Wizards and Procedures; The EXPORT Procedure.

**R**

```
library(foreign)
write.dta(ds, "filename.dta")
write.dbf(ds, "filename.dbf")
write.foreign(ds, "filename.dat", "filename.sas", package="SAS")
```

*Note:* Support for writing dataframes in R is provided in the `foreign` package. It is possible to write files directly in Stata format (see `write.dta()`) or DBF format (see `write.dbf()`) or create files with fixed fields as well as the code to read the file from within Stata, SAS, or SPSS using `write.foreign()`.

As an example with a dataset with two numeric variables  $X_1$  and  $X_2$ , the call to `write.foreign()` creates one file with the data and the SAS command file `filename.sas`, with the following contents.

```
data ds;
  infile "file.dat" dsd lrecl=79;
  input x1 x2;
run;
```

This code uses `proc format` (2.2.19) statements in SAS to store string (character) variables. Similar code is created for SPSS using `write.foreign()` with the appropriate package option.

## 1.2.7 Creating HTML formatted output

**SAS**

```
ods html file="filename.html";
...
ods html close;
```

*Note:* Any output generated between an `ods html` statement and an `ods html close` statement will be included in an HTML (hyper-text markup language) file (A.7.2). By default this will be displayed in an internal SAS window; the optional `file` option shown above will cause the output to be saved as a file.

**R**

```
library(prettyR)
htmlize("script.R", title="mytitle", echo=TRUE)
```

*Note:* The `htmlize()` function within the `prettyR` package can be used to produce HTML (hypertext markup language) from a script file (see B.2.1). The `cat()` function is used inside the script file (here denoted by `script.R`) to generate output. The `hwriter` package also supports writing R objects in HTML format. In addition, the R Markdown system using the `knitr` package and the `knit2html()` function can create HTML files as an option for reproducible analysis (11.3).

## 1.2.8 Creating XML datasets and output

**SAS**

```
libname ref xml 'dir_location\filename.xml';

data ref.filename_without_extension;
  set ds;
run;
```

```
or  
ods docbook file='filename.xml';  
...  
ods close;
```

*Note:* The `libname` statement can be used to write a SAS dataset to an XML-formatted file. It refers to a file name, rather than a directory name. The file extension `xml` is conventionally used but the `xml` specification, rather than the file extension, determines the file type that is created.

The `ods docbook` statement, in contrast, can be used to generate an XML file displaying procedure output; the file is formatted according to the OASIS DocBook DTD (document type definition).

## R

In R, the `XML` package provides support for writing XML files (see 1.1.9, write foreign files and Further resources).

## 1.3 Further resources

Introductions to data input and output in SAS can be found in [32] and [25]. Similar developments in R are accessibly presented in [187]. Paul Murrell's *Introduction to Data Technologies* text [124] provides a comprehensive introduction to XML, SQL, and other related technologies and can be found at <http://www.stat.auckland.ac.nz/~paul/ItDT> (see also Nolan and Temple Lang [127]).



# Chapter 2

## Data management

This chapter reviews important data management tasks, including dataset structure, derived variables, and dataset manipulations.

### 2.1 Structure and meta-data

#### 2.1.1 Access variables from a dataset

In SAS, every data step or procedure refers to a dataset explicitly or implicitly. Any variable in that dataset is available without further reference. In R, variable references must contain the name of the object which includes the variable, unless the object is `attached`; see below.

R

```
with(ds, mean(x))  
mean(ds$x)
```

*Note:* The `with()` and `within()` functions provide a way to access variables within a dataframe. In addition, the variables can be accessed directly using the `$` operator. Many functions (e.g., `lm()`) allow specification of a dataset to be accessed using the `data` option.

The command `attach()` will make the variables within the named dataset available in the workspace, while `detach()` will remove them from the workspace (see also `conflicts()`). The Google R Style Guide [56] states that “the possibilities for creating errors when using `attach()` are numerous. Avoid it.” We concur.

#### 2.1.2 Names of variables and their types

SAS

```
proc contents data=ds;  
run;
```

*Example: 2.6.1*

R

```
str(ds)
```

*Note:* The command `sapply(ds, class)` will return the names and classes (e.g., numeric, integer, or character) of each variable within a dataframe, while running `summary(ds)` will provide an overview of the distribution of each column.

### 2.1.3 Values of variables in a dataset

**SAS**

```
proc print data=ds (obs=nrows);
  var x1 ... xk;
run;
```

*Example: 2.6.2*

*Note:* The integer `nrows` for the `obs=nrows` option specifies how many rows to display, while the `var` statement selects variables to be displayed (A.6.1). Omitting the `obs=nrows` option or `var` statement will cause all rows and all variables in the dataset to be displayed, respectively.

**R**

```
print(ds)
or
View(ds)
or
edit(ds)
or
ds[1:10,]
ds[,2:3]
```

*Note:* The `print()` function lists the contents of the dataframe (or any other object), while the `View()` function opens a navigable window with a read-only view. The contents can be changed using the `edit()` function. Alternatively, any subset of the dataframe can be displayed on the screen using indexing, as in the final example. In the first example, the first 10 records are displayed, while in the second, the second and third variables. Variables can also be specified by name using a character vector index (see B.4.2). The `head()` function can be used to display the first (or, using `tail()`, last) values of a vector, dataset, or other object.

### 2.1.4 Label variables

As with the values of the categories, sometimes it is desirable to have a longer, more descriptive variable name (see [formatting variables](#), 2.2.19). In general, we do not recommend using this feature, as it tends to complicate communication between data analysts and other readers of output.

**SAS**

```
data ds;
  ...
  label x="This is the label for the variable 'x'";
run;
```

*Note:* The label is displayed instead of the variable name in all procedure output (except `proc print`, unless the `label` option is used) and can also be seen in `proc contents` (2.1.2).

Some procedures also allow `label` statements with identical syntax, in which case the label is used only for that procedure.

**R**

```
comment(x) = "This is the label for the variable 'x'"
```

*Note:* The label for the variable can be extracted using `comment(x)` with no assignment or via `attribute(x)$comment`.

### 2.1.5 Add comment to a dataset or variable

*Example: 2.6.1*

To help facilitate proper documentation of datasets, it can be useful to provide some annotation or description.

#### SAS

```
data ds (label="This is a comment about the dataset");
...
```

*Note:* The label can be viewed using `proc contents` (2.1.2) and retrieved as data using `ODS` (see A.7).

#### R

```
comment(ds) = "This is a comment about the dataset"
```

*Note:* The `attributes()` function (see B.4.7) can be used to list all attributes, including any `comment()`, while the `comment()` function without an argument on the right hand side will display the comment, if present.

## 2.2 Derived variables and data manipulation

This section describes the creation of new variables as a function of existing variables in a dataset.

### 2.2.1 Add derived variable to a dataset

*Example: 2.6.3*

#### SAS

```
data ds2;
set ds;
  newvar = myfunction(oldvar1, oldvar2, ...);
run;
```

*Note:* In the above, `myfunction` could be any of the functions listed in Chapter 3, might use simple operands, or combine these two options. By default, any variables to which values are assigned in a data step are saved into the output data step. To save a dataset with a new variable with the name of the original dataset, use `data ds; set ds;....` This is generally bad practice, as the original dataset is then lost, and the original data might be altered by other commands in the data step.

#### R

```
ds = transform(ds, newvar=myfunction(oldvar1, oldvar2, ...))
```

or

```
ds$newvar = myfunction(ds$oldvar1, ds$oldvar2, ...)
```

*Note:* In these equivalent examples, the new variable is added to the original dataframe. While care should be taken whenever dataframes are overwritten, this may be less risky because the addition of the variables is not connected with other changes.

### 2.2.2 Rename variables in a dataset

#### SAS

```
data ds2;
set ds (rename = (old1=new1 old2=new2 ...));
...
```

```
or
data ds;
...
rename old=new;
```

**R**

```
library(reshape)
ds = rename(ds, c("old1"="new1", "old2"="new2"))
or
names(ds)[names(ds)=="old1"] = "new1"
names(ds)[names(ds)=="old2"] = "new2"
or
ds = within(ds, {new1 = old1; new2 = old2; rm(old1, old2)})
```

*Note:* The `names()` function provides a list of names associated with an object (see B.4.6). This approach is an efficient way to undertake this task, as it involves no reading or copying of data, just a change of the names. The `edit()` function can be used to view names and edit values.

### 2.2.3 Create string variables from numeric variables

**SAS**

```
data ...;
  stringx = input(numericx, $char.);
run;
```

*Note:* Applying any function which operates on a string (or character) variable when given a numeric variable will force it to be treated as a character variable. As an example, concatenating (see 2.2.10) two numeric variables (i.e., `v3 = v1||v2`) will result in a string. See A.6.3 for a discussion of informats, which apply variable types when reading in data.

**R**

```
stringx = as.character(numericx)
typeof(stringx)
typeof(numericx)
```

*Note:* The `typeof()` function can be used to verify the type of an object; possible values include `logical`, `integer`, `double`, `complex`, `character`, `raw`, `list`, `NULL`, `closure` (function), `special`, and `builtin` (see B.4.7).

### 2.2.4 Create categorical variables from continuous variables

*Examples:* 2.6.3 and 7.10.6

**SAS**

```
data ...;
  if x ne . then newcat = (x ge minval) + (x ge cutpoint1) +
    ... + (x ge cutpointn);
run;
```

*Note:* Each expression within parentheses is a logical test returning 1 if the expression is true, 0 otherwise. If the initial condition is omitted then a missing value for `x` will return the value of 0 for `newcat`. More information about missing value coding can be found in 11.4.4 (see 4.1.2 for more about conditional execution).

**R**

```
newcat1 = (x >= cutpoint1) + ... + (x >= cutpointn)
or
```

```
newcat = cut(x, breaks=c(minval, cutpoint1, ..., cutpointn),
             labels=c("Cut0", "Cut1", ..., "Cutn"), right=FALSE)
```

*Note:* In the first implementation, each expression within parentheses is a logical test returning 1 if the expression is true, 0 if not true, and NA if x is missing. More information about missing value coding can be found in 11.4.4. The `cut()` function provides a more general framework (see also `cut_number()` from the `ggplot2` package).

## 2.2.5 Recode a categorical variable

A categorical variable may need to be recoded to have fewer levels.

### SAS

```
data ...;
  newcat = (oldcat in (val1, val2, ..., valn)) +
            (oldcat in (...)) + ...;
run;
```

*Note:* The `in` function can also accept quoted strings as input. It returns a value of 1 if any of the listed values is equal to the tested value. Section 2.2.11 has more information about set operations.

### R

```
tmpcat = oldcat
tmpcat[oldcat==val1] = newval1
tmpcat[oldcat==val2] = newval1
...
tmpcat[oldcat==valn] = newvaln
newcat = as.factor(tmpcat)
```

or

```
library(memisc)
newcat1=cases(
  "newval1"= oldcat==val1 | oldcat==val2,
  "newval2"= oldcat==valn
)
```

*Note:* Creating the variable can be undertaken in multiple steps. A copy of the old variable is first made, then multiple assignments are made for each of the new levels, for observations matching the condition inside the index (see B.4.2). In the final step, the categorical variable is coerced into a factor (class) variable. Alternatively, the `cases()` function from the `memisc` package can be used to create the factor vector in one operation, by specifying the Boolean conditions.

## 2.2.6 Create a categorical variable using logic

*Example:* 2.6.3

Here we create a trichotomous variable `newvar` which takes on a missing value if the continuous non-negative variable `oldvar` is less than 0, 0 if the continuous variable is 0, value 1 for subjects in group A with values greater than 0 but less than 50 and for subjects in group B with values greater than 0 but less than 60, or value 2 with values above those thresholds (more information about missing value coding can be found in 11.4.4).

**SAS**

```
data ...;
  if oldvar le 0 then newvar=.;
  else if oldvar eq 0 then newvar=0;
  else if (oldvar lt 50 and group eq "A") or
    (oldvar lt 60 and group eq "B")
    then newvar=1;
  else newvar=2;
run;
```

**R**

```
tmpvar = rep(NA, length(oldvar))
tmpvar[oldvar==0] = 0
tmpvar[oldvar>0 & oldvar<50 & group=="A"] = 1
tmpvar[oldvar>0 & oldvar<60 & group=="B"] = 1
tmpvar[oldvar>=50 & group=="A"] = 2
tmpvar[oldvar>=60 & group=="B"] = 2
newvar = as.factor(tmpvar)
```

or

```
library(memisc)
tmpvar = cases(
  "0" = oldvar==0,
  "1" = (oldvar>0 & oldvar<50 & group=="A") |
    (oldvar>0 & oldvar<60 & group=="B"),
  "2" = (oldvar>=50 & group=="A") |
    (oldvar>=60 & group=="B"))
```

*Note:* Creating the variable is undertaken in multiple steps in the first approach. A vector of the correct length is first created containing missing values. Values are updated if they match the conditions inside the vector index (see B.4.2). Care needs to be taken in the comparison of `oldvar==0` if non-integer values are present (see 3.2.5).

The `cases()` function from the `memisc` package provides a straightforward syntax for derivations of this sort. The `%in%` operator can also be used to test whether a string is included in a larger set of possible values (see 2.2.11 and `help("%in%")`).

## 2.2.7 Create numeric variables from string variables

**SAS**

```
data ...;
  numericx = input(stringx, integer.decimal);
run;
or
proc sort data=ds; by stringx; run;

data ds2;
set ds;
  by stringx;
  retain numericx 0;
  if first.stringx then
    numericx = numericx + 1;
run;
```

*Note:* In the first set of code, the string variable records numbers as character strings, and the code converts the storage type for these values. In the argument to the `input` function,

`integer` is the number of characters in the string, while `decimal` is an optional specification of how many characters appear after a decimal point. Applying any numeric function to a variable will force it to be treated as numeric. For example: a `numericx = stringx * 1.0` statement will also make `numericx` a numeric variable. See also A.6.3 for a discussion of informats, which apply variable types when reading in data.

The second set of code can be used when the values in the string variable are arbitrary (i.e., at least some values include non-numeric characters) and are long or numerous enough to make enumeration for coding based on logic awkward. We convert the different string values to numbers alphabetically, so that the first value is coded as number 1, and so forth. The `proc sort` and `set ...; by ...;` construction creates internal variables `first/byvar` and `last/byvar` that indicate the first and last observations with each value of `byvar`. Here these indicate when the value of the string value has changed. We use this information in concert with the `retain` statement (as in, e.g., 12.2) to record the same number for each level of the string variable.

#### R

```
numericx = as.numeric(stringx)
typeof(stringx)
typeof(numericx)
or
stringf = factor(stringx)
numericx = as.numeric(stringf)
```

*Note:* As above, the first set of code can be used when the string variable records numbers as character strings, and the code converts the storage type for these values.

The second set of code can be used when the values in the string variable are arbitrary and may be awkward to enumerate for coding based on logical operations.

The `typeof()` function can be used to verify the type of an object (see 2.2.3 and B.4.7).

## 2.2.8 Extract characters from string variables

#### SAS

```
data ...;
  get2through4 = substr(x, 2, 3);
run;
```

*Note:* The syntax functions as follows: name of the variable, start character place, how many characters to extract. The last parameter is optional. When omitted, all characters after the location specified in the second space will be extracted.

#### R

```
get2through4 = substr(x, start=2, stop=4)
```

*Note:* The arguments to `substr()` specify the input vector, start character position, and end character position.

## 2.2.9 Length of string variables

#### SAS

```
data ...;
  len = length(stringx);
run;
```

*Note:* In this example, `len` is a variable containing the number of characters in `stringx` for each observation in the dataset, excluding trailing blanks. Trailing blanks can be included through use of the `lengthc` function.

**R**

```
len = nchar(stringx)
```

*Note:* The `nchar()` function returns a vector of lengths of each of the elements of the string vector given as argument, as opposed to the `length()` function (2.3.4) that returns the number of elements in a vector.

### 2.2.10 Concatenate string variables

**SAS**

```
data ...;
  newcharvar = x1 || " VAR2 " || x2;
run;
```

*Note:* The above SAS code creates a character variable `newcharvar` containing the character variable  $X_1$  (which may be coerced from a numeric variable) followed by the string " VAR2 " then the character variable  $X_2$ . By default, no spaces are added.

**R**

```
newcharvar = paste(x1, " VAR2 ", x2, sep="")
```

*Note:* The above R code creates a character variable `newcharvar` containing the character vector  $X_1$  (which may be coerced from a numeric object) followed by the string " VAR2 " then the character vector  $X_2$ . The `sep=""` option leaves no additional separation character between these three strings.

### 2.2.11 Set operations

**SAS**

```
data ...;
  ny_in_ne = ("NY" in ("MA", "CT", "RI", "VT", "NH", "ME"));
  ma_in_ne = ("MA" in ("MA", "CT", "RI", "VT", "NH", "ME"));
run;
```

*Note:* The value of the first variable is 0, and the second is 1. The `in` operator also works with numeric values, as in 2.2.5.

**R**

```
newengland = c("MA", "CT", "RI", "VT", "ME", "NH")
"NY" %in% newengland
"MA" %in% newengland
```

*Note:* The first statement would return `FALSE`, while the second one would return `TRUE`. The `%in%` operator also works with numeric vectors (see `help("%in%")`). Vector functions for set-like operations include `union()`, `setdiff()`, `setequal()`, `intersect()`, `unique()`, `duplicated()`, and `match()`.

### 2.2.12 Find strings within string variables

SAS

```
data ...;
  /* where is the first occurrence of "pat"? */
  match = find(stringx, "pat");
  /* where is the first occurrence of "pat" after startpos? */
  matchafter = find(stringx, "pat", startpos);
  /* how many times does "pat" appear? */
  howmany = count(stringx, "pat");
run;
```

*Note:* Without the option `startpos`, `find` returns the start character for the first appearance of `pat`. If `startpos` is positive, the search starts at `startpos`, if it is negative, the search is to the left, starting at `startpos`. If `pat` is not found or `startpos=0`, then `match=0`.

R

```
matches = grep("pat", stringx)
positions = regexpr("pat", stringx)

> x = c("abc", "def", "abcdef", "defabc")
> grep("abc", x)
[1] 1 3 4
> regexpr("abc", x)
[1] 1 -1 1 4
attr(.,"match.length")
[1] 3 -1 3 3
> regexpr("abc", x) < 0
[1] FALSE TRUE FALSE FALSE
```

*Note:* The function `grep()` returns a list of elements in the vector given by `stringx` that match the given pattern, while the `regexpr()` function returns a numeric list of starting points in each string in the list (with -1 if there was no match). Testing `positions < 0` generates a vector of binary indicator of matches (TRUE=no match, FALSE=a match). The regular expressions available within `grep` and other related routines are quite powerful. As an example, Boolean OR expressions can be specified using the | operator. A comprehensive description of these operators can be found using `help(regex)`.

### 2.2.13 Find approximate strings

SAS

```
data ds;
  levdist = complev(string1, string2)
  geddist = compged(string1, string2)
  diffchar = compare(string1, string2)
run;
```

*Note:* Both `string1` and `string2` can be variables or fixed strings; in the latter case the string should be enclosed in quotes. The `levdist` function returns the Levenshtein edit distance (total number of insertions, deletions, and substitutions required to transform one string into another) between the two strings. The `compdist` function applies different and user-modifiable costs to different editing operations. For example, removing a blank might be less costly than removing a doubled character. The `compare` function returns a value of 0 if the strings are identical, or the leftmost character at which they differ, otherwise. All these functions accept various modifiers for case, blanks, quotation marks, etc.

**R**

```
agrep(pattern, c(string1, ..., stringk", max.distance=n)
```

*Note:* The support within the `agrep()` function is more rudimentary: it calculates the Levenshtein edit distance (total number of insertions, deletions, and substitutions required to transform one string into another) and it returns the indices of the elements of the second argument that are within `n` edits of `pattern` (see 2.2.12). By default the threshold is 10% of the pattern length.

```
x = c("I ask a favour", "Pardon my error", "You are my favorite")
> agrep("favor", x, max.distance=1)
[1] 1 3
```

**2.2.14 Replace strings within string variables****SAS**

```
data ds2;
  subloc = find(oldstring, "oldpat");
  newstring = oldstring;
  if subloc ne 0 then
    substr(newstring, subloc, length_oldpat) = "newpat";
run;
```

*Example:* 12.2

*Note:* When used on the left side of the = sign, the `substr` function replaces characters starting at the second argument and for the number of characters at the third argument, with the string on the right hand side. Here we find the location of the pattern to be replaced, copy the variable, and replace the sought pattern, if present.

**R**

```
newstring = gsub("oldpat", "newpat", oldstring)
or
x = "oldpat123"
substr(x, start=1, stop=6) = "newpat"
```

**2.2.15 Split strings into multiple strings****SAS**

```
data ds;
  do until(word=' ');
    count+1;
    word = scan(stringx, count);
    output;
  end;
run;
```

*Note:* The `scan` function examines its first argument and returns the word requested in the second argument. The code shown will thus extract each word and produce an output dataset with a row for each word. An alternate approach would be to use arrays (4.1.5) to get each word into the current row. Options to the `scan` function allow the user to define delimiters and how quotes, delimiters within quotes, etc., are handled.

**R**

```
strsplit(string, splitchar)
```

*Note:* The function `strsplit()` returns a list, each element of which is a vector containing the parts of the input, split at each occurrence of `splitchar`. If the input is a single character

string, this is a list of one vector. If `split` is the null string, then the function returns a list of vectors of single characters.

```
> x = "this is a test"
> strsplit(x, " ")
[[1]]
[1] "this" "is"    "a"     "test"
> strsplit(x,"")
[[1]]
[1] "t"  "h"  "i"  "s"  " "  "i"  "s"  " "  "a"  " "  "t"  "e"  "s"  "t"
```

### 2.2.16 Remove spaces around string variables

#### SAS

```
data ...;
  nolead = trail(stringx);
  notrail = trim(stringx);
  noleadortrail = strip(stringx);
  leftalign = left(stringx);
run;
```

*Note:* Spaces are removed from the beginning (`trail`), end (`trim`), or beginning and end `strip`, or moved from the beginning to the end (`left`).

#### R

```
noleadortrail = sub(' +$', ' ', sub('^\s+', ' ', stringx))
```

*Note:* The arguments to `sub()` consist of a regular expression, a substitution value, and a vector. In the first step, leading spaces are removed, then a separate call to `sub()` is used to remove trailing spaces (in both cases replacing the spaces with the null string). If instead of spaces all trailing whitespaces (e.g., tabs, space characters) should be removed, the regular expression '`' +$'`' should be replaced by '`'[[:space:]]+$'`'.

### 2.2.17 Upper to lower case

#### SAS

```
data ...;
  lowercasex = lowercase(x);
run;
or
data ...;
  lowercasex = translate(x, "abcdefghijklmnopqrstuvwxyz",
                         "ABCDEFGHIJKLMNOPQRSTUVWXYZ") ;
run;
```

*Note:* The `upcase` function makes all characters upper case. Arbitrary translations from sets of characters can be made using the `translate` function.

#### R

```
lowercasex = tolower(x)
or
lowercasex = chartr("ABCDEFGHIJKLMNOPQRSTUVWXYZ",
                     "abcdefghijklmnopqrstuvwxyz", x)
```

*Note:* The `toupper()` function can be used to convert to upper case. Arbitrary translations from sets of characters can be made using the `chartr()` function. The `iconv()` supports more complex encodings (e.g., from ASCII to other languages).

### 2.2.18 Lagged variable

A lagged variable has the value of that variable in a previous row (typically the immediately previous one) within that dataset. The value of lag for the first observation will be missing (see 11.4.4).

#### SAS

```
data ...;
  xlag1 = lag(x);
run;
or
data ...;
  xlagk = lagk(x);
run;
```

*Note:* In the latter case, the variable `xlagk` contains the value of `x` from the  $k$ th preceding observation. The value of  $k$  can be any integer less than 101: the first `k` observations will have a missing value.

If executed conditionally, only observations with computed values are included. In other words, the statement `if (condition) then xlag1 = lag(x)` results in the variable `xlag1` containing the value of `x` from the most recently processed observation *for which condition was true*. This is a common cause of confusion.

#### R

```
lag1 = c(NA, x[1:(length(x)-1)])
```

*Note:* This expression creates a one observation lag, with a missing value in the first position, and the first through second to last observation for the remaining entries (see `lag()`). We can write a function to create lags of more than one observation.

```
lagk = function(x, k) {
  len = length(x)
  if (!floor(k)==k) {
    cat("k must be an integer")
  } else if (k<1 | k>(len-1)) {
    cat("k must be between 1 and length(x)-1")
  } else {
    return(c(rep(NA, k), x[1:(len-k)]))
  }
}
> lagk(1:10, 5)
[1] NA NA NA NA NA  1  2  3  4  5
```

### 2.2.19 Formatting values of variables

*Example: 6.6.2*

Sometimes it is useful to display category names that are more descriptive than variable names. In general, we do not recommend using this feature (except potentially for graphical output), as it tends to complicate communication between data analysts and other readers of output (see labeling variables, 2.1.4). In this example, character labels are associated with a numeric variable (0=Control, 1=Low Dose, and 2=High Dose).

**SAS**

```
proc format;
  value dosegroup 0 = 'Control' 1 = 'Low Dose' 2 = 'High Dose';
run;
```

*Note:* Many procedures accept a `format x dosegroup.` statement (note trailing '.'); this syntax will accept formats designed by the user with the `proc format` statement, as well as built-in formats (see 1.2.1). Categorizations of a variable can also be imposed using `proc format`, but this can be cumbersome. In all cases, a new variable should be created as described in 2.2.4 or 2.2.5.

**R**

```
> x = c(0, 0, 1, 1, 2); x
[1] 0 0 1 1 2
> x = factor(x, 0:2, labels=c("Control", "Low Dose", "High Dose")); x
[1] Control Control Low Dose Low Dose High Dose
Levels: Control Low Dose High Dose
```

*Note:* The `names()` function can be used to associate a variable with the identifier (which is by default the observation number). As an example, this can be used to display the name of a region with the value taken by a particular variable measured in that region.

## 2.2.20 Perl interface

Perl is a high-level general purpose programming language [161]. SAS supports Perl regular expressions in the data step via the `prxparse`, `prxmatch`, `prxchange`, `prxparen`, and `prxposn` functions. Details on their use can be found in the on-line help: Contents; SAS Products; Base SAS; SAS Functions and CALL Routines; Dictionary of SAS Functions and CALL Routines, under the names listed above. The `RSPerl` package provides a bidirectional interface between Perl and R.

## 2.2.21 Accessing databases using SQL (structured query language)

*Example:* 12.5

The Structured Query Language (SQL) is a flexible language for accessing and modifying databases, data warehouses, and distributed systems. These interfaces are particularly useful when analyzing large datasets, since databases are highly optimized for certain complex operations, such as merges (joins).

An interface to external databases in SAS is available using the SAS/ACCESS product, which also supports connections to Hadoop, ODBC, and other systems. The `RODBC`, `RMySQL`, and `RSQLite` packages provide similar functionality in R [141]. The `dplyr` package is a version of the `plyr` package which is specialized for remote datastores. The `RMongo` package provides an interface to NoSQL Mongo databases (<http://www.mongodb.org>). Access and analysis of a large external database is demonstrated in 12.5.

Selections and other operations can be made on internal datasets (SAS) and dataframes (R) using an SQL-interface with `proc sql` and the `sqldf` package, respectively.

## 2.3 Merging, combining, and subsetting datasets

A common task in data analysis involves the combination, collation, and subsetting of datasets. In this section, we review these techniques for a variety of situations.

### 2.3.1 Subsetting observations

**SAS**

```
data ...;
  if x eq 1;
run;
or
data ...;
  where x eq 1;
run;
or
data ...;
  set ds (where= (x eq 1));
run;
```

*Example: 2.6.4*

*Note:* These examples create a new dataset consisting of observations where  $x = 1$ . The `if` statement has an implied “then output.” The `where` syntax also works within procedures to limit the included observations to those that meet the condition, without creating a new dataset (see 7.10.9).

**R**

```
smallds = subset(ds, x==1)
or
smallds = ds[x==1,]
```

*Note:* This example creates a subset of a dataframe consisting of observations where  $X = 1$ . In addition, many functions allow specification of a `subset=expression` option to carry out a procedure on observations that match the expression (see 8.7.6).

### 2.3.2 Drop or keep variables in a dataset

*Example: 2.6.1*

It is often desirable to prune extraneous variables from a dataset to simplify analyses. This can be done by specifying a set to keep or a set to drop.

**SAS**

```
data ds;
  ...
  keep x1 xk;
  ...
run;
or
data ds;
  set old_ds (keep=x1 xk);
  ...
run;
```

*Note:* The complementary syntax `drop` can also be used, both as a statement in the data step and as a `data` statement option.

**R**

```
subset(ds, select=c("x1", "xk"))
or
ds[,c("x1", "xk")] # select x1 and xk
or
```

```
subset(ds, select = -c(x2, x5)) # drop x2 and x5
```

*Note:* The first examples create a new dataframe consisting of the variables `x1` and `xk`. An alternative is to specify the variables to be excluded (in this case the second). The last example can be used to drop specified variables.

More sophisticated ways of listing the variables to be kept are available. For example, the command `ds[,grep("x1|^pat", names(ds))]` would keep `x1` and all variables starting with `pat` (see 2.2.12).

### 2.3.3 Random sample of a dataset

It is sometimes useful to sample a subset (here quantified as *nsamp*) of observations without replacement from a larger dataset (see random number seed, 3.1.3).

#### SAS

```
/* sample without replacement */
proc surveyselect data=ds out=outds n=nsamp method=srs; run;

/* sample with replacement */
proc surveyselect data=ds out=outds n=nsamp method=urs; run;

/* sample with replacement to observed sample size (bootstrap) */
proc surveyselect data=ds out=outds rate=1 method=urs; run;
or
/* sample without replacement */
data ds2;
set ds;
order = uniform(0);
run;

proc sort data=ds2;
by order;
run;

data ds3;
set ds2;
if _n_ le nsamp;
run;
```

*Note:* The former code shows how `proc surveyselect` can be used in various ways. A `strata` statement can be used to sample, e.g., with replacement but maintaining certain margins.

The second set of code shows a manual method. Note that if the second `data` step is omitted, the observations have been randomly reordered. It is also possible to generate a random sample in a single data step by generating a uniform random variate for each observation in the original data but using an `if` statement to retain only those which meet a criteria which changes with the number retained.

#### R

```
library(mosaic)
newds = resample(ds, size=nsamp, replace=FALSE)
```

*Note:* By default, the `resample()` function in the `mosaic` package creates a sample without replacement from a dataframe or vector (the built-in `sample()` function cannot directly sample a dataframe). The `replace=TRUE` option can be used to override this (e.g., when bootstrapping; see 11.4.3).

### 2.3.4 Observation number

**SAS**

```
data ...;
  x = _n_;
run;
```

*Example: 2.6.2*

*Note:* The variable `_n_` is created automatically by SAS and counts the number of lines of data that have been input into the `data` step. It is a temporary variable that it is not stored in the dataset unless a new variable is created (as demonstrated in the above code).

**R**

```
> y = c("abc", "def", "ghi")
> x = 1:length(y)
> x
[1] 1 2 3
> ds = data.frame(x, y)
> nrow(ds)
[1] 3
```

*Note:* The `length()` function returns the number of elements in a vector. This can be used in conjunction with the `:` operator (4.1.3) to create a vector with the integers from 1 to the sample size. The `nrow()` function returns the number of rows for a dataframe (B.4.6). Observation numbers might also be set as case labels as opposed to the row number (see `names()`).

### 2.3.5 Keep unique values

See also 2.3.6 (duplicated values).

**SAS**

```
proc sort data=ds out=newds nodupkey;
  by x1 ... xk;
run;
```

*Example: 2.6.2*

*Note:* The dataset `newds` contains all the variables in the dataset `ds`, but only one row for each unique value across  $x_1 x_2 \dots x_k$ .

The `sort` procedure is unusual in SAS in that it will overwrite a dataset and lose data by default, in this application, if the `out=` option is omitted.

**R**

```
uniquevalues = unique(x)
uniquevalues = unique(data.frame(x1, ..., xk))
```

*Note:* The `unique()` function returns each of the unique values represented by the vector or dataframe denoted by `x`.

### 2.3.6 Identify duplicated values

See also 2.3.5 (unique values).

**SAS**

```
proc sort data=ds out=ds2 nouniquekey;
  by x1 ... xk;
run;
```

*Note:* The output dataset `ds2` contains all observations that have duplicated `by` values. Thus if the input dataset contains only two identical observations, the output dataset will also contain both observations.

The `sort` procedure is unusual in SAS in that it will overwrite a dataset and lose data by default, in this application, if the `out=` option is omitted.

## R

```
duplicated(x)
```

*Note:* The `duplicated()` function returns a logical vector indicating a replicated value. Note that the first occurrence is not a replicated value. Thus `duplicated(c(1,1))` returns FALSE TRUE.

### 2.3.7 Convert from wide to long (tall) format

*Example:* 7.10.9

Data are often found in a different shape than that required for analysis. One example of this is commonly found in longitudinal measures studies. In this setting it is convenient to store the data in a wide or multivariate format with one line per observation, containing typically subject invariant factors (e.g., gender), as well as a column for each repeated outcome. An example is given below.

```
id female inc80 inc81 inc82
1   0      5000  5500  6000
2   1      2000  2200  3300
3   0      3000  2000  1000
```

Here the income for 1980, 1981, and 1982 are included in one row for each id.

In contrast, SAS and R tools for repeated measures analyses (7.4.2) typically require a row for each repeated outcome, as demonstrated below.

```
id year female inc
1  80   0      5000
1  81   0      5500
1  82   0      6000
2  80   1      2000
2  81   1      2200
2  82   1      3300
3  80   0      3000
3  81   0      2000
3  82   0      1000
```

In this section and in (2.3.8) below, we show how to convert between these two forms of this example data.

## SAS

```
data long;
set wide;
array incarray [3] inc80 - inc82;
do year = 80 to 82;
  inc = incarray[year - 79];
  output;
end;
drop inc80 - inc82;
run;
```

or

```

data long;
set wide;
  year=80; inc=inc80; output;
  year=81; inc=inc81; output;
  year=82; inc=inc82; output;
  drop inc80 - inc82;
run;
or
proc transpose data=wide out=long_i;
  var inc80 - inc82;
  by id female;
run;

data long;
set long_i;
  year=substr(_name_, 4, 2)*1.0;
  drop _name_;
  rename col1=inc;
run;

```

*Note:* The `year=substr()` statement in the last data step is required if the value of `year` must be numeric. The remainder of that step makes the desired variable name appear and removes extraneous information.

## R

```

long = reshape(wide, idvar="id", varying=list(names(wide)[3:5]),
  v.names="inc", timevar="year", times=80:82, direction="long")

```

*Note:* The list of variables to transpose is provided in the list `varying`, creating `year` as the time variable with values specified by `times` (see `library(reshape)` for more flexible dataset transformations).

### 2.3.8 Convert from long (tall) to wide format

See also 2.3.7 (reshape from wide to tall).

*Example:* 7.10.9

## SAS

```

proc transpose data=long out=wide (drop=_name_) prefix=inc;
  var inc;
  id year;
  by id female;
run;

```

*Note:* The `(drop=_name_)` option prevents the creation of an unneeded variable in the `wide` dataset.

## R

```

wide = reshape(long, v.names="inc", idvar="id", timevar="year",
  direction="wide")

```

*Note:* This example assumes that the dataset `long` has repeated measures on `inc` for subject `id` determined by the variable `year`. See also `library(reshape)` for more flexible dataset transformations.

### 2.3.9 Concatenate and stack datasets

#### SAS

```
data newds;
set ds1 ds2;
run;
```

*Note:* The datasets `ds1` and `ds2` are assumed to previously exist. The newly created dataset `newds` has as many rows as the sum of rows in `ds1` and `ds2`, and as many columns as unique variable names across the two input datasets.

#### R

```
newds = rbind(ds1, ds2)
```

*Note:* The result of `rbind()` is a dataframe with as many rows as the sum of rows in `ds1` and `ds2`. Dataframes given as arguments to `rbind()` must have the same column names. The similar `cbind()` function makes a dataframe with as many columns as the sum of the columns in the input objects. A similar function (`c()`) operates on vectors.

### 2.3.10 Sort datasets

#### SAS

```
proc sort data=ds;
  by x1 ... xk;
run;
```

*Example: 2.6.4*

*Note:* The keyword `descending` can be inserted before any variable to sort that variable from high to low (see 11.1).

#### R

```
sortds = ds[order(x1, x2, ..., xk),]
```

*Note:* The R command `sort()` can be used to sort a vector, while `order()` can be used to sort dataframes by selecting a new permutation of order for the rows. The `decreasing` option can be used to change the default sort order (for all variables). The command `sort(x)` is equivalent to `x[order(x)]`. As an alternative, a numeric variable can be reversed by specifying `-x1` instead of `x1`.

### 2.3.11 Merge datasets

*Example: 7.10.11*

Merging datasets is commonly required when data on single units are stored in multiple tables or datasets. We consider a simple example where variables `id`, `year`, `female`, and `inc` are available in one dataset, and variables `id` and `maxval` in a second. For this simple example, with the first dataset given as:

	<code>id</code>	<code>year</code>	<code>female</code>	<code>inc</code>
1	80	0		5000
1	81	0		5500
1	82	0		6000
2	80	1		2000
2	81	1		2200
2	82	1		3300
3	80	0		3000

```
3 81 0      2000
3 82 0      1000
```

and the second given below:

```
id maxval
2 2400
1 1800
4 1900
```

the desired merged dataset would look like:

	<b>id</b>	<b>year</b>	<b>female</b>	<b>inc</b>	<b>maxval</b>
1	1	81	0	5500	1800
2	1	80	0	5000	1800
3	1	82	0	6000	1800
4	2	82	1	3300	2400
5	2	80	1	2000	2400
6	2	81	1	2200	2400
7	3	82	0	1000	NA
8	3	80	0	3000	NA
9	3	81	0	2000	NA
10	4	NA	NA	NA	1900

in R, or equivalently, as below in SAS.

	<b>id</b>	<b>year</b>	<b>female</b>	<b>inc</b>	<b>maxval</b>
1	1	81	0	5500	1800
2	1	80	0	5000	1800
3	1	82	0	6000	1800
4	2	82	1	3300	2400
5	2	80	1	2000	2400
6	2	81	1	2200	2400
7	3	82	0	1000	.
8	3	80	0	3000	.
9	3	81	0	2000	.
10	4	.	.	.	1900

### SAS

```
proc sort data=ds1; by x1 ... xk;
run;

proc sort data=ds2; by x1 ... xk;
run;

data newds;
merge ds1 ds2;
  by x1 ... xk;
run;
```

For example, the result desired in the note above can be created as follows, assuming the two datasets are named `ds1` and `ds2`.

```
proc sort data=ds1; by id; run;

proc sort data=ds2; by id; run;

data newds;
merge ds1 ds2;
  by id;
run;
```

*Note:* The `by` statement in the `data` step describes the matching criteria, in that every observation with a unique set of  $X_1$  through  $X_k$  in `ds1` will be matched to every observation with the same set of  $X_1$  through  $X_k$  in `ds2`. The output dataset will have as many columns as there are uniquely named variables in the input datasets and as many rows as unique values across  $X_1$  through  $X_k$ . The `by` statement can be omitted, which results in the  $n$ th row of each dataset contributing to the  $n$ th row of the output dataset, though this is rarely desirable. If matched rows have discrepant values for a commonly named variable, the value in the later-named dataset is used in the output dataset.

## R

```
newds = merge(ds1, ds2, by="id", all=TRUE)
```

*Note:* The `all` option specifies that extra rows will be added to the output for any rows that have no matches in the other dataset. Multiple variables can be specified in the `by` option; if this is left out, all variables in both datasets are used.

## 2.4 Date and time variables

For SAS, variables in the date formats are integers counting the number of days since January 1, 1960. In R, the date functions return a `Date` class that represents the number of days since January 1, 1970. The R function `as.numeric()` can be used to create a numeric variable with the number of days since 1/1/1970. (See also the `chron` and `lubridate` packages).

### 2.4.1 Create date variable

See also 1.1.4 (read more complex files).

#### SAS

```
data ...;
  dayvar = input("04/29/2014", mmddyy10.);
  dayvar2 = mdy(4, 29, 2014);
  todays_date = today();
run;
```

*Note:* The value of both `dayvar` and `dayvar2` is the integer number of days between January 1, 1960 and April 29, 2014. The value of `todays_date` is the integer number of days between January 1, 1960 and the day the current instance of SAS was opened.

## R

```
dayvar = as.Date("2014-04-29")
todays_date = as.Date(Sys.time())
```

*Note:* The return value of `as.Date()` is a `Date` class object. If converted to numeric `dayvar`, it represents the number of days between January 1, 1970 and April 29, 2014, while `todays_date` is the integer number of days since January 1, 1970 (see `ISOdate()`).

## 2.4.2 Extract weekday

### SAS

```
data ...;
  wkday = weekday(datevar);
run;
```

*Note:* The `weekday` function returns an integer representing the weekday, 1=Sunday, ..., 7=Saturday.

### R

```
wkday = weekdays(datevar)
```

*Note:* `wkday` contains a string with the name of the weekday of the `Date` object.

## 2.4.3 Extract month

### SAS

```
data ...;
  monthval = month(datevar);
run;
```

*Note:* The `month` function returns an integer representing the month, 1=January, ..., 12=December.

### R

```
monthval = months(datevar)
```

*Note:* The function `months()` returns a string with the name of the month of the `Date` object.

## 2.4.4 Extract year

### SAS

```
data ...;
  yearval = year(datevar);
run;
```

*Note:* The variable `yearval` is years counted in the Common Era (CE, also called AD).

### R

```
yearval = substr(as.POSIXct(datevar), 1, 4)
```

*Note:* The `as.POSIXct()` function returns a string representing the date, with the first four characters corresponding to the year.

## 2.4.5 Extract quarter

### SAS

```
data ...;
  qtrval = qtr(datevar);
run;
```

*Note:* The return values for `qtrval` are 1, 2, 3, or 4.

**R**

```
qtrval = quarters(datevar)
```

*Note:* The function `quarters()` returns a string representing the quarter of the year (e.g., "Q1" or "Q2") given by the Date object.

### 2.4.6 Create time variable

*Example:* 12.4.2

See also 4.3.1 (timing commands).

**SAS**

```
data ...;
  timevar_1960 = datetime();
  timevar_midnight = time();
run;
```

*Note:* The variable `timevar_1960` contains the number of seconds since midnight, December 31, 1959. The variable `timevar_midnight` contains the number of seconds since the most recent midnight.

**R**

```
> arctime = as.POSIXlt("2014-04-29 17:15:45 NZDT")
> arctime
[1] "2014-04-29 17:15:45"
> now = Sys.time()
> now
[1] "2014-04-01 10:12:11 EST"
```

*Note:* The objects `arctime` and `now` can be compared with the subtraction operator to monitor elapsed time.

## 2.5 Further resources

Comprehensive introductions to data management in SAS can be found in [32] and [25]. Similar developments in R are accessibly presented in [187]. Frank Harrell's `Hmisc` package [62] provides a number of useful data management routines.

## 2.6 Examples

To help illustrate the tools presented in this and related chapters, we apply many of the entries to the HELP RCT data. SAS and R code can be downloaded from <http://www.amherst.edu/~nhorton/sasr2/examples>.

### 2.6.1 Data input and output

We begin by reading the dataset (1.1.5), keeping only the variables that are needed (2.3.2).

```

proc import
  datafile='c:/book/help.csv'
  out=dsprelim
  dbms=dlm;
  delimiter=',';
  getnames=yes;
run;

data ds;
set dsprelim;
  keep id cesd f1a -- f1t i1 i2 female treat;
run;

> options(digits=3)
> options(width=72) # narrow output
> ds = read.csv("http://www.amherst.edu/~nhorton/sasr2/datasets/help.csv")
> newds = subset(ds, select=c("cesd","female","i1","i2","id","treat","f1a",
  "f1b","f1c","f1d","f1e","f1f","f1g","f1h","f1i","f1j","f1k","f1l",
  "f1m","f1n","f1o","f1p","f1q","f1r","f1s","f1t"))

```

We can then show a summary of the dataset. In SAS, we use the ODS system (A.7) to reduce the length of the output.

```

options ls=70; /* narrows width to stay in grey box */
ods select attributes;
proc contents data=ds;
run;
ods select all;

```

#### The CONTENTS Procedure

Data Set Name	WORK.DS	Observations	453
Member Type	DATA	Variables	26
Engine	V9	Indexes	0
Created	Sunday, January 12, 2014 11:10:06 AM	Observation Length	208
Last Modified	Sunday, January 12, 2014 11:10:06 AM	Deleted Observations	0
Protection		Compressed	NO
Data Set Type		Sorted	NO
Label			
Data Representation	WINDOWS_64		
Encoding	wlatin1 Western (Windows)		

The default output prints a line for each variable with its name and additional information; the **short** option below limits the output to just the names of the variable.

```

options ls=70; /* narrows width to stay in grey box */
ods select variablesshort;
proc contents data=ds short;
run;
ods select all;

The CONTENTS Procedure
      Alphabetic List of Variables for WORK.DS
cesd f1a f1b f1c f1d f1e f1f f1g f1h f1i f1j f1k f1l f1m f1n f1o f1p
f1q f1r f1s f1t female i1 i2 id treat
> names(newds)

[1] "cesd"    "female"  "i1"       "i2"       "id"       "treat"   "f1a"
[8] "f1b"     "f1c"      "f1d"      "f1e"      "f1f"      "f1g"      "f1h"
[15] "f1i"     "f1j"      "f1k"      "f1l"      "f1m"      "f1n"      "f1o"
[22] "f1p"     "f1q"      "f1r"      "f1s"      "f1t"

> str(newds[,1:10]) # structure of the first 10 variables

'data.frame':   453 obs. of  10 variables:
 $ cesd : int  49 30 39 15 39 6 52 32 50 46 ...
 $ female: int  0 0 0 1 0 1 1 0 1 0 ...
 $ i1    : int  13 56 0 5 10 4 13 12 71 20 ...
 $ i2    : int  26 62 0 5 13 4 20 24 129 27 ...
 $ id    : int  1 2 3 4 5 6 7 8 9 10 ...
 $ treat : int  1 1 0 0 0 1 0 1 0 1 ...
 $ f1a   : int  3 3 3 0 3 1 3 1 3 2 ...
 $ f1b   : int  2 2 2 0 0 0 1 1 2 3 ...
 $ f1c   : int  3 0 3 1 3 1 3 2 3 3 ...
 $ f1d   : int  0 3 0 3 3 3 1 3 1 0 ...

```

```
> summary(newds[,1:10]) # summary of the first 10 variables

      cesd      female       i1       i2
Min.   : 1.0   Min.   :0.000   Min.   : 0.0   Min.   : 0.0
1st Qu.:25.0  1st Qu.:0.000  1st Qu.: 3.0   1st Qu.: 3.0
Median :34.0  Median :0.000  Median :13.0  Median :15.0
Mean   :32.8  Mean   :0.236  Mean   :17.9  Mean   :22.6
3rd Qu.:41.0  3rd Qu.:0.000  3rd Qu.:26.0  3rd Qu.:32.0
Max.   :60.0  Max.   :1.000  Max.   :142.0 Max.   :184.0

      id      treat      f1a      f1b
Min.   : 1   Min.   :0.000   Min.   :0.00   Min.   :0.00
1st Qu.:119  1st Qu.:0.000  1st Qu.:1.00  1st Qu.:0.00
Median :233  Median :0.000  Median :2.00  Median :1.00
Mean   :233  Mean   :0.497  Mean   :1.63  Mean   :1.39
3rd Qu.:348  3rd Qu.:1.000  3rd Qu.:3.00  3rd Qu.:2.00
Max.   :470  Max.   :1.000  Max.   :3.00  Max.   :3.00

      f1c      f1d
Min.   :0.00   Min.   :0.00
1st Qu.:1.00  1st Qu.:0.00
Median :2.00  Median :1.00
Mean   :1.92  Mean   :1.56
3rd Qu.:3.00  3rd Qu.:3.00
Max.   :3.00  Max.   :3.00
```

Displaying the first few rows of data can give a more concrete sense of what is in the dataset.

```
proc print data=ds (obs=3) width=minimum,  
run;
```

```
> head(newds, n=3)
```

	cesd	female	i1	i2	id	treat	f1a	f1b	f1c	f1d	f1e	f1f	f1g	f1h	f1i	f1j
1	49	0	13	26	1	1	3	2	3	0	2	3	3	0	2	3
2	30	0	56	62	2	1	3	2	0	3	3	2	0	0	3	0
3	39	0	0	0	3	0	3	2	3	0	2	2	1	3	2	3
	f1k	f1l	f1m	f1n	f1o	f1p	f1q	f1r	f1s	f1t						
1	3	0	1	2	2	2	2	3	3	2						
2	3	0	0	3	0	0	0	2	0	0						
3	1	0	1	3	2	0	0	3	2	0						

Saving the dataset in native format (1.2.3) will ease future access. We also add a comment (2.1.5) to help later users understand what is in the dataset.

```

libname book 'c:/temp';
data book.ds (label = "HELP baseline dataset");
set ds;
run;

> comment(newds) = "HELP baseline dataset"
> comment(newds)

[1] "HELP baseline dataset"

> save(ds, file="savedfile")

```

Saving it in a foreign format (1.1.9), say Microsoft Excel or comma separated value format, will allow access to other tools for analysis and display.

```

proc export data=ds replace
  outfile="c:/temp/ds.xls"
  dbms=excel;
run;

> write.csv(ds, file="ds.csv")

```

Getting data into SAS format from R is particularly useful; note that the R code below generates an ASCII dataset and a SAS command file to read it into SAS.

```

> library(foreign)
> write.foreign(newds, "file.dat", "file.sas", package="SAS")

```

## 2.6.2 Data display

We begin by consideration of the CESD (Center for Epidemiologic Statistics) measure of depressive symptoms for this sample at baseline.

```

proc print data=ds (obs=10);
  var cesd;
run;

Obs          cesd
  1            49
  2            30
  3            39
  4            15
  5            39
  6              6
  7            52
  8            32
  9            50
 10           46

```

The indexing mechanisms in R (see B.4.2) are helpful in extracting subsets of a vector.

```

> with(newds, cesd[1:10])

[1] 49 30 39 15 39   6 52 32 50 46

> with(newds, head(cesd, 10))

[1] 49 30 39 15 39   6 52 32 50 46

```

It may be useful to know how many high values there are and to which observations they belong.

```

proc print data=ds;
  where cesd gt 56;
  var cesd;
run;

Obs      cesd
 64      57
116      58
171      57
194      60
231      58
295      58
387      57

> with(newds, which(cesd > 56))

[1] 64 116 171 194 231 295 387

> subset(newds, cesd > 56, select=c("cesd"))

      cesd
64      57
116      58
171      57
194      60
231      58
295      58
387      57

```

Similarly, it may be useful to examine the observations with the lowest values:

```

proc sort data=ds out=dss1;
  by cesd;
run;

proc print data=dss1 (obs=4);
  var id cesd i1 treat;
run;

Obs      id      cesd      i1      treat
 1      233      1      3      0
 2      418      3     13      0
 3      139      3      1      0
 4      95       4      9      1

> with(newds, sort(cesd)[1:4])

[1] 1 3 3 4

> with(newds, which.min(cesd))

[1] 199

```

### 2.6.3 Derived variables and data manipulation

Suppose the dataset arrived with only the individual CESD questions and not the sum. We would need to create the CESD score. In SAS, we'll do this using an array (4.1.5) to aid the

recoding of the four questions which are asked “backwards,” meaning that high values of the response are counted for fewer points.<sup>1</sup> In R we’ll approach the recoding of the flipped questions by reading the CESD items into a new object. To demonstrate other tools, we’ll also see if there’s any missing data (11.4.4) and generate some other statistics about the question responses.

```
options ls = 70;
proc means data=ds min q1 median q3 max mean std n nmiss;
  var f1g;
run;
```

The MEANS Procedure

Analysis Variable : f1g

	Lower		Upper	
Minimum	Quartile	Median	Quartile	Maximum
0	1.0000000	2.0000000	3.0000000	3.0000000

Analysis Variable : f1g

		N	
Mean	Std Dev	N	Miss
1.7300885	1.0953143	452	1

```
> library(mosaic)
> tally(~ is.na(f1g), data=newds)
```

TRUE	FALSE	Total
1	452	453

```
> favstats(~ f1g, data=newds)
```

min	Q1	median	Q3	max	mean	sd	n	missing
0	1	2	3	3	1.73	1.1	452	1

We utilize the `tally()` and `favstat()` functions from the `mosaic` package to display the distribution and number of missing values.

Now we’re ready to create the score. We’ll generate the sum of the non-missing items, which effectively imputes 0 for the missing values, as well as a version that imputes the mean of the observed values instead.

---

<sup>1</sup>According to the coding instructions found at <http://www.amherst.edu/~nhorton/sasr2/cesd.pdf>.

```

data cesd;
set ds;
/* list of backwards questions */
array backwards [*] f1d f1h f1l f1p;
/* for each, subtract the stored value from 3 */
do i = 1 to dim(backwards);
  backwards[i] = 3 - backwards[i];
end;
/* this generates the sum of the non-missing questions */
newcesd = sum(of f1a -- f1t);
/* This counts the number of missing values, per person */
nmisscesd = nmiss(of f1a -- f1t);
/* this gives the sum, imputing the mean of non-missing */
imputemeancesd = mean(of f1a -- f1t) * 20;
run;

> # reverse code f1d, f1h, f1l and f1p
> cesditems = with(newds, cbind(f1a, f1b, f1c, (3 - f1d), f1e, f1f, f1g,
      (3 - f1h), f1i, f1j, f1k, (3 - f1l), f1m, f1n, f1o, (3 - f1p),
      f1q, f1r, f1s, f1t))
> nmisscesd = apply(is.na(cesditems), 1, sum)
> ncesditems = cesditems
> ncesditems[is.na(cesditems)] = 0
> newcesd = apply(ncesditems, 1, sum)
> imputemeancesd = 20/(20-nmisscesd)*newcesd

```

It is prudent to review the results when deriving variables. We'll check our recreated CESD score against the one which came with the dataset. To ensure that missing data has been correctly coded, we print the subjects with any missing questions.

```

proc print data=cesd (obs=20);
  where nmisscesd gt 0;
  var cesd newcesd nmisscesd imputemeancesd;
run;

Obs      cesd      newcesd      nmisscesd      imputemeancesd
  4        15        15          1        15.7895
  17       19        19          1        20.0000
  87       44        44          1        46.3158
  101      17        17          1        17.8947
  154      29        29          1        30.5263
  177      44        44          1        46.3158
  229      39        39          1        41.0526

> cbind(newcesd, newds$cesd, nmisscesd, imputemeancesd)[nmisscesd>0,]

      newcesd      nmisscesd      imputemeancesd
[1,]      15 15          1        15.8
[2,]      19 19          1        20.0
[3,]      44 44          1        46.3
[4,]      17 17          1        17.9
[5,]      29 29          1        30.5
[6,]      44 44          1        46.3
[7,]      39 39          1        41.1

```

The output shows that the original dataset was created with unanswered questions counted as if they had been answered with a zero. This conforms to the instructions provided with the CESD, but might be questioned on theoretical grounds.

It is often necessary to create a new variable using logic (2.2.6). In the HELP study, many subjects reported extreme amounts of drinking (as the baseline measure was taken while they were in detox). Here, an ordinal measure of alcohol consumption (abstinent, moderate, high-risk) is created using information about average consumption per day in the 30 days prior to detox (*i1*, measured in standard drink units) and maximum number of drinks per day in the 30 days prior to detox (*i2*). The number of drinks required for each category differs for men and women according to NIAAA guidelines for physicians [126].

```

data ds2;
set ds;
  if i1 eq 0 then drinkstat="abstinent";
  if (i1 eq 1 and i2 le 3 and female eq 1) or
    ((i1 eq 1) or (i1 eq 2)) and i2 le 4 and female eq 0
    then drinkstat="moderate";
  if (((i1 gt 1) or (i2 gt 3)) and female eq 1) or
    ((i1 gt 2) or (i2 gt 4)) and female eq 0
    then drinkstat="highrisk";
  if nmiss(i1,i2,female) ne 0 then drinkstat="";
run;

> library(memisc)
> newds = transform(newds, drinkstat=
  cases(
    "abstinent" = i1==0,
    "moderate" = (i1>0 & i1<=1 & i2<=3 & female==1) /
      (i1>0 & i1<=2 & i2<=4 & female==0),
    "highrisk" = ((i1>1 | i2>3) & female==1) |
      ((i1>2 | i2>4) & female==0)))
> library(mosaic)
```

It is always prudent to check the results of derived variables. As a demonstration, we display the observations in the last 6 rows of the data.

```

proc print data=ds2 (firstobs=365 obs=370);
  var i1 i2 female drinkstat;
run;
```

Obs	i1	i2	female	drinkstat
365	6	24	0	highrisk
366	6	6	0	highrisk
367	0	0	0	abstinent
368	0	0	1	abstinent
369	8	8	0	highrisk
370	32	32	0	highrisk

```
> tmpds = subset(newds, select=c("i1", "i2", "female", "drinkstat"))
> tmpds[365:370,]

   i1 i2 female drinkstat
365 6 24      0  highrisk
366 6 6       0  highrisk
367 0 0       0 abstinent
368 0 0       1 abstinent
369 8 8       0  highrisk
370 32 32     0  highrisk
```

It is also useful to focus such checks on a subset of observations. Here we show the drinking data for moderate female drinkers.

```
proc print data=ds2;
  where drinkstat eq "moderate" and female eq 1;
  var i1 i2 female drinkstat;
run;
```

Obs	i1	i2	female	drinkstat
116	1	1	1	moderate
137	1	3	1	moderate
225	1	2	1	moderate
230	1	1	1	moderate
264	1	1	1	moderate
266	1	1	1	moderate
394	1	1	1	moderate

```
> subset(tmpds, drinkstat=="moderate" & female==1)
```

i1	i2	female	drinkstat
116	1	1	moderate
137	1	3	moderate
225	1	2	moderate
230	1	1	moderate
264	1	1	moderate
266	1	1	moderate
394	1	1	moderate

Basic data description is an early step in analysis. Here we calculate relevant summaries of drinking and gender.

```
proc freq data=ds2;
  tables drinkstat;
run;
```

#### The FREQ Procedure

drinkstat	Frequency	Percent	Cumulative	Cumulative
			Frequency	Percent
abstinent	68	15.01	68	15.01
highrisk	357	78.81	425	93.82
moderate	28	6.18	453	100.00

```

> with(tmpds, sum(is.na(drinkstat)))
[1] 0

> mytab = rbind(tally(~ drinkstat, data=tmpds),
+                 tally(~ drinkstat, format="percent", data=tmpds))
> row.names(mytab) = c("count", "percent")
> mytab

      abstinent moderate highrisk Total
count       68     28.00    357.0   453
percent     15      6.18     78.8   100

proc freq data=ds2;
  tables drinkstat*female;
run;

The FREQ Procedure
Table of drinkstat by female
drinkstat      female
Frequency |
Percent      |
Row Pct      |
Col Pct      |      0|      1|  Total
-----+-----+
abstinent  |     42 |     26 |     68
           |  9.27 |  5.74 | 15.01
           | 61.76 | 38.24 |
           | 12.14 | 24.30 |
-----+-----+
highrisk   |    283 |     74 |    357
           | 62.47 | 16.34 | 78.81
           | 79.27 | 20.73 |
           | 81.79 | 69.16 |
-----+-----+
moderate   |     21 |      7 |     28
           |  4.64 |  1.55 |  6.18
           | 75.00 | 25.00 |
           |  6.07 |  6.54 |
-----+-----+
Total      346     107     453
           76.38   23.62  100.00

```

```
> tally(~ drinkstat + female, data=tmpds)

      female
drinkstat      0   1 Total
  abstinent  42  26   68
  moderate   21    7   28
  highrisk  283  74  357
  Total     346 107  453

> tally(~ female | drinkstat, data=tmpds)

      drinkstat
female  abstinent moderate highrisk
  0        0.618    0.750   0.793
  1        0.382    0.250   0.207
  Total    1.000    1.000   1.000
```

To display gender in a more direct fashion, we create a new character variable. Note that in these quoted strings, both SAS and R are case sensitive.

```
data ds3;
set ds;
  if female eq 1 then gender="Female";
  else if female eq 0 then gender="Male";
run;

proc freq data=ds3;
  tables female gender;
run;
```

#### The FREQ Procedure

female	Frequency	Percent	Cumulative	Cumulative
			Frequency	Percent
0	346	76.38	346	76.38
1	107	23.62	453	100.00

gender	Frequency	Percent	Cumulative	Cumulative
			Frequency	Percent
Female	107	23.62	107	23.62
Male	346	76.38	453	100.00

```
> newds = transform(newds,
  gender=factor(female, c(0,1), c("Male","Female")))
> tally(~ female + gender, margin=FALSE, data=newds)

      gender
female Male Female
  0    346    0
  1     0   107
```

### 2.6.4 Sorting and subsetting datasets

It is often useful to sort datasets (2.3.10) by the order of a particular variable (or variables). Here we sort by CESD and drinking.

```
proc sort data=ds;
  by cesd i1;
run;

proc print data=ds (obs=5);
  var id cesd i1;
run;

Obs      id      cesd      i1
 1      233      1      3
 2      139      3      1
 3      418      3     13
 4      251      4      4
 5      95       4      9

> newds = ds[order(ds$cesd, ds$i1),]
> newds[1:5,c("cesd", "i1", "id")]

      cesd i1  id
199    1  3 233
394    3  1 139
349    3 13 418
417    4  4 251
85     4  9  95
```

It is sometimes necessary to create data that is a subset (2.3.1) of other data. Here we make a dataset which only includes female subjects. First, we create the subset and calculate a summary value in the resulting dataset.

```
data females;
set ds;
  where female eq 1;
run;

proc means data=females mean maxdec=1;
  var cesd;
run;
```

The MEANS Procedure  
Analysis Variable : cesd

Mean

-----  
36.9  
-----

```
> females = subset(ds, female==1)
> with(females, mean(cesd))

[1] 36.9

> # an alternative approach
> mean(ds$cesd[ds$female==1])

[1] 36.9
```

To test the subsetting, we then display the mean for both genders.

```
proc sort data=ds;
  by female;
run;

proc means data=ds mean maxdec=2;
  by female;
  var cesd;
run;
```

```
female=0
The MEANS Procedure
Analysis Variable : cesd
```

```
Mean
-----
31.60
-----
```

```
female=1
Analysis Variable : cesd
```

```
Mean
-----
36.89
-----
```

```
> with(ds, tapply(cesd, female, mean))

 0    1
31.6 36.9

> library(mosaic)
> mean(cesd ~ female, data=ds)

 0    1
31.6 36.9
```

# Chapter 3

## Statistical and mathematical functions

This chapter reviews key statistical, probability, mathematical, and matrix functions.

### 3.1 Probability distributions and random number generation

SAS and R can calculate quantiles and cumulative distribution values as well as generate random numbers for a large number of distributions. Random variables are commonly needed for simulation and analysis. SAS includes comprehensive random number generation through the `rand` function, while R provides a series of `r`-commands.

Both packages allow specification of a seed for the random number generator. This is important to allow replication of results (e.g., while testing and debugging). Information about random number seeds can be found in 3.1.3.

Table 3.1 summarizes support for quantiles, cumulative distribution functions, and random numbers. More information on probability distributions within R can be found in the CRAN probability distributions task view (<http://cran.r-project.org/web/views/Distributions.html>).

#### 3.1.1 Probability density function

Both R and SAS use similar syntax for a variety of distributions. Here we use the normal distribution as an example; others are shown in Table 3.1.

**SAS**

```
data ...;
  y = cdf('NORMAL', 1.96, 0, 1);
run;
```

**R**

```
y = pnorm(1.96, mean=0, sd=1)
```

Table 3.1: Quantiles, probabilities, and pseudo-random number generation: distributions available in SAS and R

Distribution	R DISTNAME	SAS DISTNAME
Beta	<code>beta</code>	BETA
Beta-binomial	<code>betabin*</code>	
binomial	<code>binom</code>	BINOMIAL
Cauchy	<code>cauchy</code>	CAUCHY
chi-square	<code>chisq</code>	CHISQUARE
exponential	<code>exp</code>	EXPONENTIAL
F	<code>f</code>	F
gamma	<code>gamma</code>	GAMMA
geometric	<code>geom</code>	GEOMETRIC
hypergeometric	<code>hyper</code>	HYPERGEOMETRIC
inverse normal	<code>inv.gaussian*</code>	IGAUSS <sup>+</sup>
Laplace	<code>laplace*</code>	LAPLACE
logistic	<code>logis</code>	LOGISTIC
lognormal	<code>lnorm</code>	LOGNORMAL
negative binomial	<code>nbinom</code>	NEGBINOMIAL
normal	<code>norm</code>	NORMAL
Poisson	<code>pois</code>	POISSON
Student's <i>t</i>	<code>t</code>	T
Uniform	<code>unif</code>	UNIFORM
Weibull	<code>weibull</code>	WEIBULL

Note: For R, prepend `d` to the command to compute density functions of a distribution `dDISTNAME(xvalue, parm1, ..., parmn)`, `p` for the cumulative distribution function, `pDISTNAME(xvalue, parm1, ..., parmn)`, `q` for the quantile function `qDISTNAME(prob, parm1, ..., parmn)`, and `r` to generate random variables `rDISTNAME(nrand, parm1, ..., parmn)` where in the last case a vector of `nrand` values is the result. For SAS, random variates can be generated from the `rand` function: `rand('DISTNAME', parm1, ..., parmn)`, the areas to the left of a value via the `cdf` function: `cdf('DISTNAME', quantile, parm1, ..., parmn)`, and the quantile associated with a probability (the inverse CDF) via the `quantile` function: `quantile('DISTNAME', probability, parm1, ..., parmn)`, where the number of `parms` varies by distribution. Details are available through the on-line help: Contents; SAS Products; Base SAS; SAS Functions and CALL Routines; Dictionary of SAS Functions and CALL Routines; RAND Function. Note that within the `rand` function, SAS is case sensitive.

\* The `betabin()`, `inv.gaussian()`, and `laplace()` families of distributions are available using the VGAM package.

<sup>+</sup> The inverse normal is not available in the `rand` function; inverse normal variates can be generated by taking the inverse of normal random variates.

### 3.1.2 Quantiles of a probability density function

Both R and SAS use similar syntax for a variety of distributions. Here we use the normal distribution as an example; others are shown in Table 3.1.

**SAS**

```
data ...;
  y = quantile('NORMAL', .975, 0, 1);
run;
```

**R**

```
y = qnorm(.975, mean=0, sd=1)
```

### 3.1.3 Setting the random number seed

SAS includes comprehensive random number generation through the `rand` function. For variables created this way, an initial seed is selected automatically by SAS based on the system clock. Sequential calls use a seed derived from this initial seed. To generate a replicable series of random variables, use the `call streaminit` function before the first call to `rand`. The `rand` functions use the Mersenne–Twister random number generator developed by Matsumoto and Nishimura [115].

**SAS**

```
call streaminit(42);
```

*Note:* A set of separate SAS functions for random number generation includes `normal`, `ranbin`, `rancau`, `ranexp`, `rangam`, `rannor`, `ranpoi`, `rantbl`, `rantri`, `ranuni`, and `uniform`. These functions use a multiplicative congruential random number generator, which generally has less desirable properties than the Mersenne–Twister algorithm used for the `rand` function [41]. For these functions, calling with an argument of (0) is equivalent to calling the `rand` function without first running `call streaminit`; an initial seed is generated from the system clock. Calling the same functions with an integer greater than 0 as argument is equivalent to running `call streaminit` before an initial use of `rand`. In other words, this will result in a series of variates based on the first specified integer. Note that `call streaminit` or specifying an integer to one of the specific functions need only be performed once per `data` step; all seeds within that `data` step will be based on that seed.

In R, the default behavior is a seed based on the system clock. To generate a replicable series of variates, first run `set.seed(seedval)` where `seedval` is a single integer for the default Mersenne–Twister random number generator.

**R**

```
set.seed(42)
set.seed(Sys.time())
```

*Note:* More information can be found using `help(.Random.seed)`.

### 3.1.4 Uniform random variables

**SAS**

```
data ...;
  x1 = uniform(seed);
  x2 = rand('UNIFORM');
run;
```

*Note:* The variables  $x_1$  and  $x_2$  are uniform on the interval (0,1). The `ranuni()` function is a synonym for `uniform()`.

**R**

```
x = runif(n, min=0, max=1)
```

*Note:* The arguments specify the number of variables to be created and the range over which they are distributed.

### 3.1.5 Multinomial random variables

**SAS**

```
data ...;
  x1 = rantbl(seed, p1, p2, ..., pk);
  x2 = rand('TABLE', p1, p2, ..., pk);
run;
```

*Note:* The variables  $x_1$  and  $x_2$  take the value  $i$  with probability  $p_i$  and value  $k + 1$  with value  $1 - \sum_{i=1}^k p_i$ .

**R**

```
library(Hmisc)
x = rMultinom(matrix(c(p1, p2, ..., pr), 1, r), n)
```

*Note:* The function `rMultinom()` from the `Hmisc` package allows the specification of the desired multinomial probabilities ( $\sum_r p_r = 1$ ) as a  $1 \times r$  matrix. The final parameter is the number of variates to be generated. See also `rmultinom()` in the `stats` package.

### 3.1.6 Normal random variables

*Example: 3.4.1*

**SAS**

```
data ...;
  x1 = normal(seed);
  x2 = rand('NORMAL', mu, sigma);
run;
```

*Note:* The variable  $X_1$  is a standard normal ( $\mu = 0$  and  $\sigma = 1$ ), while  $X_2$  is normal with specified mean and standard deviation. The function `rannor()` is a synonym for `normal()`.

**R**

```
x1 = rnorm(n)
x2 = rnorm(n, mean=mu, sd=sigma)
```

*Note:* The arguments specify the number of variables to be created and (optionally) the mean and standard deviation (default  $\mu = 0$  and  $\sigma = 1$ ).

### 3.1.7 Multivariate normal random variables

For the following, we first create a  $3 \times 3$  covariance matrix. Then we generate 1000 realizations of a multivariate normal vector with the appropriate correlation or covariance.

**SAS**

```

data Sigma (type=cov);
infile cards;
input _type_ $ _Name_ $ x1 x2 x3;
cards;
cov   x1      3 1 2
cov   x2      1 4 0
cov   x3      2 0 5
;
run;

proc simnormal data=sigma out=outtest2 numreal=1000;
  var x1 x2 x3;
run;

```

*Note:* The type=cov option to the data step defines Sigma as a special type of SAS dataset which contains a covariance matrix in the format shown. A similar type=corr dataset can be used with a correlation matrix instead of a covariance matrix.

**R**

```

library(MASS)
mu = rep(0, 3)
Sigma = matrix(c(3, 1, 2,
                1, 4, 0,
                2, 0, 5), nrow=3)
xvals = mvrnorm(1000, mu, Sigma)
apply(xvals, 2, mean)
or
rmultnorm = function(n, mu, vmat, tol=1e-07)
# a function to generate random multivariate Gaussians
{
  p = ncol(vmat)
  if (length(mu) != p)
    stop("mu vector is the wrong length")
  if (max(abs(vmat - t(vmat))) > tol)
    stop("vmat not symmetric")
  vs = svd(vmat)
  vsqrt = t(vs$v %*% (t(vs$u) * sqrt(vs$d)))
  ans = matrix(rnorm(n * p), nrow=n) %*% vsqrt
  ans = sweep(ans, 2, mu, "+")
  dimnames(ans) = list(NULL, dimnames(vmat)[[2]])
  return(ans)
}
xvals = rmultnorm(1000, mu, Sigma)
apply(xvals, 2, mean)

```

*Note:* The returned object xvals, of dimension  $1000 \times 3$ , is generated from the variance covariance matrix denoted by Sigma, which has first row and column (3,1,2). An arbitrary mean vector can be specified using the c() function.

Several techniques are illustrated in the definition of the rmultnorm function. The first lines test for the appropriate arguments and return an error if the conditions are not satisfied. The singular value decomposition (see 3.3.15) is carried out on the variance covariance matrix, and the sweep function is used to transform the univariate normal random variables

generated by `rnorm` to the desired mean and covariance. The `dimnames()` function applies the existing names (if any) for the variables in `vmat`, and the result is returned.

### 3.1.8 Truncated multivariate normal random variables

See also 4.1.1.

No built-in function exists to do this in SAS, but it is straightforward to code.

#### SAS

```
data trunc;
upper = 1;
lower = -2;
do i = 1 to 100;
  do until (x ne .);
    x = rand("NORMAL");
    if x > upper or x lt lower then x = .;
  end;
  output;
end;
run;
```

#### R

```
library(tmvtnorm)
x = rtmvnorm(n, mean, sigma, lower, upper)
```

*Note:* The arguments specify the number of variables to be created, the mean and standard deviation, and a vector of lower and upper truncation values.

### 3.1.9 Exponential random variables

#### SAS

```
data ...;
  x1 = ranexp(seed);
  x2 = rand('EXPONENTIAL');
run;
```

*Note:* The expected value of both  $X_1$  and  $X_2$  is 1: for exponentials with expected value  $k$ , multiply the generated value by  $k$ .

#### R

```
x = rexp(n, rate=lambda)
```

*Note:* The arguments specify the number of variables to be created and (optionally) the inverse of the mean (default  $\lambda = 1$ ).

### 3.1.10 Other random variables

*Example: 3.4.1*

The list of probability distributions supported within SAS and R can be found in Table 3.1. In addition to these distributions, the inverse probability integral transform can be used to generate arbitrary random variables with invertible cumulative density function  $F$  (exploiting the fact that  $F^{-1} \sim U(0, 1)$ ). As an example, consider the generation of random variates from an exponential distribution with rate parameter  $\lambda$ , where  $F(X) = 1 - \exp(-\lambda X) = U$ . Solving for  $X$  yields  $X = -\log(1 - U)/\lambda$ . If we generate a  $\text{Uniform}(0, 1)$  variable, we can use this relationship to generate an exponential with the desired rate parameter.

**SAS**

```
data ds;
  lambda = 2;
  uvar = uniform(42);
  expvar = -1 * log(1-uvar)/lambda;
run;
```

**R**

```
lambda = 2
expvar = -log(1-runif(1))/lambda
```

## 3.2 Mathematical functions

### 3.2.1 Basic functions

See also 2.2 (derived variables) and 2.2.11 (sets).

**SAS**

```
data ...;
  minx = min(x1, ..., xk);
  maxx = max(x1, ..., xk);
  meanx = mean(x1, ..., xk);
  modulusx = mod(x1, x2);
  stddevx = std(x1, ..., xk);
  sumx = sum(x1, ..., xk)
  absolutevaluex = abs(x);
  etothex = exp(x);
  xtothey = x**y;
  squareroottx = sqrt(x);
  naturallogx = log(x);
  logbase10x = log10(x);
  logbase2x = log2(x);
run;
```

*Note:* The first five functions operate on a row-by-row basis within SAS (the equivalent within R operates on a column-wise basis).

**R**

```
minx = min(x)
maxx = max(x)
meanx = mean(x)
modx = x1 %% x2
stddevx = sd(x)
absolutevaluex = abs(x)
squarerootx = sqrt(x)
etothex = exp(x)
xtothey = x^y
naturallogx = log(x)
logbase10x = log10(x)
logbase2x = log2(x)
logbasearbx = log(x, base=42)
```

*Note:* The first five functions operate on a column-wise basis in R (the equivalent within SAS operates on a row-wise basis).

### 3.2.2 Trigonometric functions

**SAS**

```
data ...;
  sinx = sin(x);
  sinpi = sin(constant('PI'));
  cosx = cos(x);
  tanx = tan(x);
  arccosx = arcos(x);
  arcsinx = arsin(x);
  arctanx = atan(x);
  arctanxy = atan2(x, y);
run;
```

**R**

```
sin(pi)
cos(0)
tan(pi/4)
acos(x)
asin(x)
atan(x)
atan2(x, y)
```

### 3.2.3 Special functions

**SAS**

```
data ...;
  betaxy = beta(x, y);
  gammax = gamma(x);
  factorialn = fact(n);
  nchooser = comb(n, r);
  npermr = perm(n, r);
run;
```

**R**

```
betaxy = beta(x, y)
gammax = gamma(x)
factorialn = factorial(n)
nchooser = choose(n, r)

library(gtools)
nchooser = length(combinations(n, r)[,1])
npermr = length(permuations(n, r)[,1])
```

*Note:* The `combinations()` and `permuations()` functions return a list of possible combinations and permutations: the count equivalent to the SAS functions above can be calculated through use of the `length()` function given the first column of the output.

### 3.2.4 Integer functions

See also 1.2.2 (rounding and number of digits to display).

**SAS**

```
data ...;
  nextintx = ceil(x);
  justintx = floor(x);
  roundx = round(x1, x2);
  roundint = round(x, 1);
  movetozero = int(x);
run;
```

*Note:* The value of `roundx` is  $X_1$ , rounded to the nearest  $X_2$ . The value of `movetozero` is the same as `justint` if  $x > 0$  or `nextint` if  $x < 0$ .

**R**

```
nextintx = ceiling(x)
justintx = floor(x)
round2dec = round(x, 2)
roundint = round(x)
keep4sig = signif(x, 4)
movetozero = trunc(x)
```

*Note:* The second parameter of the `round()` function determines how many decimal places to round. The value of `movetozero` is the same as `justint` if  $x > 0$  or `nextint` if  $x < 0$ .

### 3.2.5 Comparisons of floating point variables

Because certain floating point values of variables do not have exact decimal equivalents, there may be some error in how they are represented on a computer. For example, if the true value of a particular variable is  $1/7$ , the approximate decimal is 0.1428571428571428. For some operations (for example, tests of equality), this approximation can be problematic.

**SAS**

```
data ds;
  x1 = ((1/7) eq .142857142857);
  x2 = (fuzz((1/7) - .142857142857) eq 0);
run;
```

*Note:* In the above example,  $x_1 = 0$ ,  $x_2 = 1$ . If the argument to `fuzz` is less than  $10^{-12}$ , then the result is the nearest integer.

**R**

```
> all.equal(0.1428571, 1/7)
[1] "Mean relative difference: 3.000000900364093e-07"
> all.equal(0.1428571, 1/7, tolerance=0.0001)
[1] TRUE
```

*Note:* The tolerance option for the `all.equal()` function determines how many decimal places to use in the comparison of the vectors or scalars (the default tolerance is set to the underlying lower level machine precision).

### 3.2.6 Complex numbers

Support for operations on complex numbers is available in R.

**R**

```
(0+1i)^2
```

*Note:* The above expression is equivalent to  $i^2 = -1$ . Additional support is available through the `complex()` function and related routines (see also `Re()` and `Im()`).

### 3.2.7 Derivatives

Rudimentary support for finding derivatives is available within R. These functions are particularly useful for high-dimensional optimization problems (see 3.2.9).

**R**

```
library(mosaic)
D(x^2 ~ x)
```

*Note:* The `D()` function within the `mosaic` package returns a function, which can be evaluated or plotted using `plotFun()`. Second (or higher order) derivatives can be found by repeatedly applying the `D` function with respect to  $X$ . This function (as well as `deriv()`) is useful in numerical optimization (see the `nls()`, `optim()` and `optimize()` functions).

### 3.2.8 Integration

*Example:* 10.1.6

Rudimentary support for calculus, including the evaluation of integrals, is available within R.

**R**

```
library(mosaic)
antiD(2*x ~ x)
```

### 3.2.9 Optimization problems

SAS and R can be used to solve optimization (maximization) problems. As an extremely simple example, consider maximizing the area of a rectangle with perimeter equal to 20. Each of the sides can be represented by  $x$  and  $10-x$ , with the area of the rectangle equal to  $x * (10 - x)$ .

**SAS**

```
proc iml;
  start f_area(x);
    f = x*(10-x);
    return (f);
  finish f_area;
  con = {0, 10};
  x = {2} ;
  optn = {1, 2};
  call nlpcg(rc, xres, "f_area", x, optn, con);
quit;
```

*Note:* The above uses conjugate gradient optimization. Several additional optimization routines are provided in `proc iml` (see the on-line help: Contents; SAS Products; SAS/IML; SAS/IML User's Guide; Nonlinear Optimization Examples).

**R**

```
f = function(x) { return(x*(10-x)) }
optimize(f, interval=c(0, 10), maximum=TRUE)
```

*Note:* Other optimization functions available within R include `nls()`, `uniroot()`, `optim()` and `constrOptim()` (see the CRAN task view on optimization and mathematical programming).

## 3.3 Matrix operations

Matrix operations are often needed in statistical analysis. For SAS, proc iml (a separate product from SAS/STAT), is needed to treat data as a matrix. Within R, matrices can be created using the `matrix()` function (see B.4.5): matrix operations are then immediately available.

Here, we briefly outline the process needed to read a SAS dataset into SAS/IML as a matrix, perform some function, then make the result available as a SAS native dataset. Throughout this section, we use capital letters to emphasize that a matrix is described, though `proc iml` is not case sensitive (unlike R).

```
proc iml;
  use ds;
  read all var{x1 ... xk} into Matrix_x;
  ... /* perform a function of some sort */
  print Matrix_x; /* print the matrix to the output window */
  create newds from Matrix_x;
  append from Matrix_x;
quit;
```

*Note:* Calls to `proc iml` end with a `quit` statement, rather than a `run` statement.

In addition to the routines described below, the `Matrix` package in R is particularly useful for manipulation of large as well as sparse matrices.

### 3.3.1 Create matrix from vector

In this entry, we demonstrate creating a  $2 \times 2$  matrix consisting of the first four nonzero integers:

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}.$$

#### SAS

```
proc iml;
  A = {1 2, 3 4};
quit;
```

#### R

```
A = matrix(c(1, 2, 3, 4), nrow=2, ncol=2, byrow=TRUE)
```

### 3.3.2 Combine vectors or matrices

We demonstrate creating a matrix from a set of conformable column vectors or smaller datasets. Note the close relationship between this and merging data (2.3.11).

#### SAS

```
proc iml;
  A = {1 2 3};
  B = {4 5 6};
  AHB = a || b;
  AVB = a // b;
quit;
```

*Note:* The `||` operator concatenates horizontally, while the `//` operator stacks vertically.

**R**

```
A = cbind(x1, ..., xk)
A2 = c(x1, ..., xk)
```

*Note:* A is a matrix with columns made up of all the columns of  $x_1, \dots, x_k$ . A2 is a vector of all of the elements in  $x_1$ , followed by all of the elements of  $x_2$ , etc. The `rbind()` function can be used to combine the matrices as rows instead of columns, making a  $k \times n$  matrix (see also the `unstack()` and `stack()` commands).

**3.3.3 Matrix addition****SAS**

```
proc iml;
  A = {1 2, 3 4};
  B = A + A;
quit;
```

**R**

```
A = matrix(c(1, 2, 3, 4), nrow=2, ncol=2, byrow=TRUE)
B = A + A
```

**3.3.4 Transpose matrix****SAS**

```
proc iml;
  A = {1 2, 3 4};
  transA = A`;
  transA_2 = t(A);
quit;
```

*Note:* Both `transA` and `transA_2` contain the transpose of A.

**R**

```
A = matrix(c(1, 2, 3, 4), nrow=2, ncol=2, byrow=TRUE)
transA = t(A)
```

*Note:* If a data frame is transposed in this manner, it will be converted to a matrix (which forces a single class for the objects within it).

**3.3.5 Find the dimension of a matrix or dataset****SAS**

```
proc iml;
  A = matrix(c(1, 2, 3, 4), nrow=2, ncol=2, byrow=TRUE);
  dimension(A);
quit;
or
```

```
%macro dschars(ds);
  %global nvars nobs;
  %let dsid=%sysfunc(open(&ds.,IN));
  %let nobs=%sysfunc(attrn(&dsid,nobs));
  %let nvars=%sysfunc(attrn(&dsid,nvars));
  %let rc=%sysfunc(close(&dsid));
%mend;

%dschars(sashelp.class)
%put There are &nobs obs and &nvars vars;
```

*Note:* The former code shows how to get the number of rows and columns in an SAS/IML matrix. The latter shows how to find the number of observations and variables in a dataset. In SAS, making them available programmatically is not especially natural. We need to both find the required values and assign the values to macro variables. Here we use the `%sysfunc` macro. The `dschars` macro calls the `sysfunc` macro to open the dataset, get the desired information, and close the dataset. The `%global` statement avoids the usual macro scoping and makes the results available outside the macro, while the `%let` statements make macro variables out of the results of the calls to `sysfunc`. The `sashelp.class` dataset is part of the standard SAS installation. It has 19 observations on 5 variables.

**R**

```
A = matrix(c(1, 2, 3, 4), nrow=2, ncol=2, byrow=TRUE)
dim(A)
```

*Note:* The `dim()` function returns the dimension (number of rows and columns) for both matrices and dataframes, but does not work for vectors. The `length()` function returns the length of a vector or the number of elements in a matrix.

### 3.3.6 Matrix multiplication

**SAS**

```
proc iml;
  A = {1 2, 3 4};
  Asquared = A * A;
quit;
```

**R**

```
A = matrix(c(1, 2, 3, 4), nrow=2, ncol=2, byrow=TRUE)
Asquared = A %*% A
```

### 3.3.7 Invert matrix

**SAS**

```
proc iml;
  A = {1 2, 3 4};
  Ainv = inv(A);
quit;
```

**R**

```
A = matrix(c(1, 2, 3, 4), nrow=2, ncol=2, byrow=TRUE)
Ainv = solve(A)
```

### 3.3.8 Component-wise multiplication

Unlike the matrix multiplication in 3.3.6, the result of this operation is scalar multiplication of each element in the matrix. For example, the component-wise multiplication of

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \text{ with itself yields } \begin{pmatrix} 1 & 4 \\ 9 & 16 \end{pmatrix}.$$

**SAS**

```
proc iml;
  A = {1 2, 3 4};
  newmat = A#A;
quit;
```

**R**

```
A = matrix(c(1, 2, 3, 4), nrow=2, ncol=2, byrow=TRUE)
newmat = A * A
```

### 3.3.9 Create submatrix

**SAS**

```
proc iml;
  A = {1 2 3 4, 5 6 7 8, 9 10 11 12};
  Asub = a[2:3, 3:4];
quit;
```

**R**

```
A = matrix(1:12, nrow=3, ncol=4, byrow=TRUE)
Asub = A[2:3, 3:4]
```

### 3.3.10 Create a diagonal matrix

**SAS**

```
proc iml;
  A = {1 2, 3 4};
  diagMat = diag(A);
quit;
```

*Note:* For matrix  $A$ , this results in a matrix with the same diagonals, but with all off-diagonals set to 0. For a vector argument, the function generates a matrix with the vector values as the diagonals and all off-diagonals 0.

**R**

```
A = matrix(c(1, 2, 3, 4), nrow=2, ncol=2, byrow=TRUE)
diagMat = diag(c(1, 4))    # argument is a vector
diagMat = diag(diag(A))    # A is a matrix
```

*Note:* For a vector argument, the `diag()` function generates a matrix with the vector values as the diagonals and all off-diagonals 0. For matrix  $A$ , the `diag()` function creates a vector of the diagonal elements (see 3.3.11); a diagonal matrix with these diagonal entries, but all off-diagonals set to 0, can be created by running the `diag()` with this vector as argument.

### 3.3.11 Create a vector of diagonal elements

**SAS**

```
proc iml;
  A = {1 2, 3 4};
  diagVals = vecdiag(A);
quit;
```

*Note:* The vector diagVals contains the diagonal elements of matrix A.

**R**

```
A = matrix(c(1, 2, 3, 4), nrow=2, ncol=2, byrow=TRUE)
diagVals = diag(A)
```

### 3.3.12 Create a vector from a matrix

**SAS**

```
proc iml;
  A = {1 2, 3 4};
  newvec = shape(A, 1);
quit;
```

*Note:* This makes a row vector from all the values in the matrix.

**R**

```
A = matrix(c(1, 2, 3, 4), nrow=2, ncol=2, byrow=TRUE)
newvec = c(A)
```

### 3.3.13 Calculate the determinant

**SAS**

```
proc iml;
  A = {1 2, 3 4};
  detval = det(A);
quit;
```

**R**

```
A = matrix(c(1, 2, 3, 4), nrow=2, ncol=2, byrow=TRUE)
det(A)
```

### 3.3.14 Find eigenvalues and eigenvectors

**SAS**

```
proc iml;
  A = {1 2, 3 4};
  Aeval = eigval(A);
  Aevec = eigvec(A);
quit;
```

**R**

```
A = matrix(c(1, 2, 3, 4), nrow=2, ncol=2, byrow=TRUE)
Aev = eigen(A)
Aeval = Aev$values
Aevec = Aev$vectors
```

*Note:* The `eigen()` function in R returns a list consisting of the eigenvalues and eigenvectors, respectively, of the matrix given as the argument.

### 3.3.15 Find the singular value decomposition

The singular value decomposition of a matrix  $A$  is given by  $A = U * \text{diag}(Q) * V^T$  where  $U^T U = V^T V = VV^T = I$  and  $Q$  contains the singular values of  $A$ .

**SAS**

```
proc iml;
  A = {1 2, 3 4};
  call svd(U, Q, V, A);
quit
```

**R**

```
A = matrix(c(1, 2, 3, 4), nrow=2, ncol=2, byrow=TRUE)
svdres = svd(A)
U = svdres$u
Q = svdres$d
V = svdres$v
```

*Note:* The `svd()` function returns a list with components corresponding to a vector of singular values, a matrix with columns corresponding to the left singular values, and a matrix with columns containing the right singular values.

## 3.4 Examples

To help illustrate the tools presented in this chapter, we apply some of the entries in examples. SAS and R code can be downloaded from <http://www.amherst.edu/~nhorton/sasr2/examples>.

### 3.4.1 Probability distributions

To demonstrate more tools, we leave the HELP dataset and show examples of how data can be generated within each programming environment. We will generate values (3.1.6) from the normal and  $t$  distribution densities; note that the probability density functions are not hard-coded into SAS as they are within R.

```
data dists;
  do x = -4 to 4 by .1;
    normal_01 = sqrt(2 * constant('PI'))**(-1) * exp(-1 * ((x*x)/2)) ;
    dfval = 1;
    t_1df = (gamma((dfval +1)/2) / (sqrt(dfval * constant('PI')) *
      gamma(dfval/2))) * (1 + (x*x)/dfval)**(-1 * ((dfval + 1)/2));
    output;
  end;
run;
```

```
> x = seq(from=-4, to=4.2, length=100)
> normval = dnorm(x, 0, 1)
> dfval = 1
> tval = dt(x, df=dfval)
```

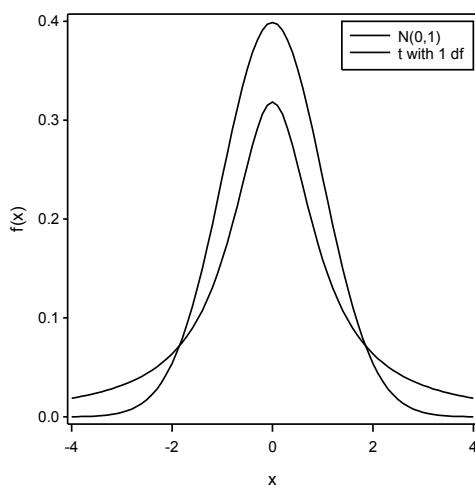
Figure 3.1 displays a plot of these distributions in SAS and R.

```
legend1 label=none position=(top inside right) frame down=2
  value = ("N(0,1)" tick=2 "t with 1 df");
axis1 label=(angle=90 "f(x)") minor=none order=(0 to .4 by .1);
axis2 minor=none order=(-4 to 4 by 2);
symbol1 i=j v=none l=1 c=black w=5;
symbol2 i=j v=none l=21 c=black w=5;
proc gplot data= dists;
  plot (normal_01 t_1df) * x / overlay legend=legend1
    vaxis=axis1 haxis=axis2;
run; quit;

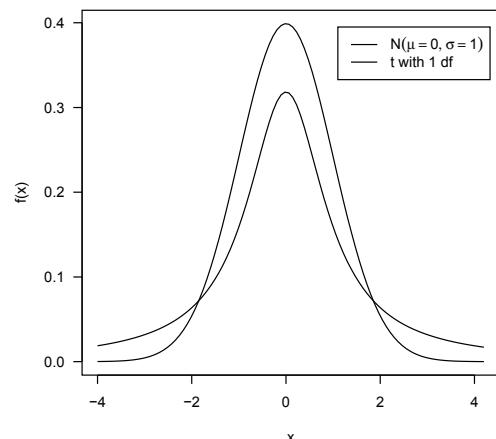
> plot(x, normval, type="n", ylab="f(x)", las=1)
> lines(x, normval, lty=1, lwd=2)
> lines(x, tval, lty=2, lwd=2)
> legend(1.1, .395, lty=1:2, lwd=2,
  legend=c(expression(N(mu == 0,sigma == 1)),
  paste("t with ", dfval, " df", sep="")))
```

The `xpnorm()` function within the `mosaic` package may be useful for teaching purposes, to display information about the normal density function (see Figure 3.2).

```
> library(mosaic)
> xpnorm(1.96, mean=0, sd=1)
```



(a) SAS



(b) R

Figure 3.1: Comparison of standard normal and  $t$  distribution with 1 degree of freedom (df)

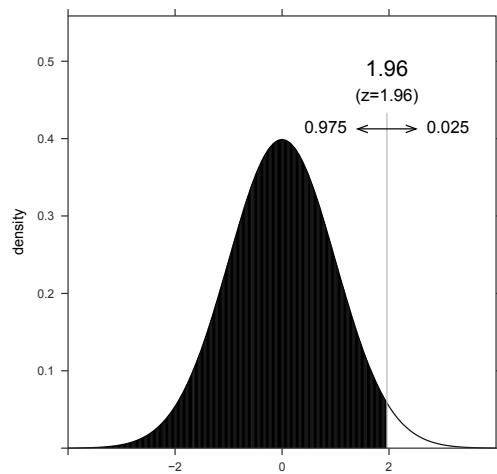


Figure 3.2: Descriptive plot of the normal distribution

## Chapter 4

# Programming and operating system interface

This chapter reviews programming functions as well as interface to the underlying operating system.

### 4.1 Control flow, programming, and data generation

Programming is an area where SAS and R are quite different. Here we show some basic aspects of programming. We include parallel code for each language, while noting that some actions have no straightforward analogue in the other language.

#### 4.1.1 Looping

SAS

```
data;
  do i = i1 to i2;
    x = normal(0);
    output;
  end;
run;
```

*Example: 11.2*

*Note:* The above code generates a new dataset with  $i_2 - i_1 + 1$  standard normal variates, with seed based on the system clock (3.1.6). The generic syntax for looping includes three parts: 1) a `do varname = values` statement; 2) the statements to be executed within the loop; 3) an `end` statement. As with all programming languages, users should be careful about modifying the index during processing. Other options include `do while` and `do until`. The `values` on the right of the `=` in the `do` statement can be consecutive integers if `values` are `values1 to values2`, as shown, or can step by values other than 1, using this syntax: `do i = i1 to i2 by byval`. To step across specified values, use statements like `do i = k1, ..., kn`.

R

```
x = numeric(i2-i1+1)  # create placeholder
for (i in 1:length(x)) {
  x[i] = rnorm(1) # this is slow and inefficient!
}
```

or (preferably)

```
x = rnorm(i2-i1+1) # this is far better
```

*Note:* Most tasks in R that could be written as a loop are often dramatically faster if they are encoded as a vector operation (as in the second and preferred option above). Examples of situations where loops in R are particularly useful can be found in 11.1 and 11.2. The `along.with` option for `seq()` and the `seq_along()` function can also be helpful.

More information on control structures for looping and conditional processing such as `while` and `repeat` can be found in `help(Control)`.

#### 4.1.2 Conditional execution

##### SAS

```
data ds;
  if expression1 then expression2 else expression3;
run;
```

or

```
if expression1 then expression2;
else if expression3 then expression4;
...
else expressionn;
```

or

```
if expression1 then do;
  ...
end;
else if expression2 then expression3;
...
```

*Note:* There is no limit on the number of conditions tested in the `else` statements, which always refer back to the most recent `if` statement. Once a condition in this sequence is met, the remaining conditions are not tested. Listing conditions in decreasing order of occurrence will therefore result in more efficient code.

The `then` code is executed if the `expression` following the `if` has a non-missing, non-zero value. So, for example, the statement `if 1 then y = x**2` is valid syntax, equivalent to the statement `y=x**2`. Good programming style is to make each tested expression be a logical test, such as `x eq 1` returning 1 if the expression is true and 0 otherwise. SAS includes mnemonics `lt`, `le`, `eq`, `ge`, `gt`, and `ne` for `<`,  `$\leq$` , `=`,  `$\geq$` ,  `$>$` , and  `$\neq$` , respectively. The mnemonic syntax cannot be used for assignment, and it is recommended style to reserve `=` for assignment and use only the mnemonics for testing.

The `do-end` block is the equivalent of `{ }` in the R code below. Any group of `data` step statements can be included in a `do-end` block.

##### R

```
if (expression1) { expression2 }
or
if (expression1) { expression2 } else { expression3 }
or
ifelse(expression, x, y)
```

*Note:* The `if` statement, with or without `else`, tests a single logical statement; it is not an elementwise (vector) function. If `expression1` evaluates to TRUE, then `expression2` is evaluated. The `ifelse()` function operates on vectors and evaluates the expression given as `expression` and returns `x` if it is TRUE and `y` otherwise (see comparisons, B.4.2). An expression can include multi-command blocks of code (in brackets). The `switch()` function may also be useful for more complicated tasks.

### 4.1.3 Sequence of values or patterns

*Example:* 10.1.3

It is often useful to generate a variable consisting of a sequence of values (e.g., the integers from 1 to 100) or a pattern of values (1 1 1 2 2 2 3 3 3). This might be needed to generate a variable consisting of a set of repeated values for use in a simulation or graphical display.

As an example, we demonstrate generating data from a linear regression model of the form:

$$E[Y|X_1, X_2] = \beta_0 + \beta_1 X_1 + \beta_2 X_2, \quad Var(Y|X) = 9, \quad Corr(X_1, X_2) = 0.$$

SAS

```
data ds;
  do x = start to stop by step;
  ...
  end;
run;
```

Note: The values `start` and `stop` are the smallest and largest values `x` will take. If `by step` is omitted, the values of `x` will be incremented by 1.

The following code implements the model described above for  $n = 200$ . The value 42 below is an arbitrary seed (3.1.3) [1] used for random number generation. The datasets `ds1` and `ds2` will be identical. However such values are generated, it would be wise to use `proc freq` (5.3.1) to check whether the intended results were achieved.

```
data ds1;
  beta0 = -1; beta1 = 1.5; beta2 = .5; rmse = 3;
  /* note multiple statements on previous line */
  do x1 = 1 to 2;
    do x2 = 1 to 2;
      do obs = 1 to 50;
        y = beta0 + beta1*x1 + beta2*x2 + normal(42)*rmse;
        output;
      end;
    end;
  end;
run;
or
data ds2;
  beta0 = -1; beta1 = 1.5; beta2 = .5; rmse = 3;
  do i = 1 to 200;
    x1 = (i gt 100) + 1;
    x2 = (((i gt 50) and (i le 100)) or (i gt 150)) + 1;
    y = beta0 + beta1*x1 + beta2*x2 + normal(42)*rmse;
    output;
  end;
run;
```

R

```
# generate
seq(from=i1, to=i2, length.out=nvals)
seq(from=i1, to=i2, by=1)
seq(i1, i2)
i1:i2

rep(value, times=nvals)
```

or

```
rep(value, each=nvals)
```

*Note:* The `seq` function creates a vector of length `val` if the `length.out` option is specified. If the `by` option is included, the length is approximately  $(i_2 - i_1)/byval$ . The `i1:i2` operator is equivalent to `seq(from=i1, to=i2, by=1)`. The `rep` function creates a vector of length `nvals` with all values equal to `value`, which can be a scalar, vector, or list. The `each` option repeats each element of `value` `nvals` times. The default is `times`.

The following code implements the model described above for  $n = 200$ .

```
> n = 200
> x1 = rep(c(0,1), each=n/2)      # x1 resembles 0 0 0 ... 1 1 1
> x2 = rep(c(0,1), n/2)          # x2 resembles 0 1 0 1 ... 0 1
> beta0 = -1; beta1 = 1.5; beta2 = .5;
> rmse = 3
> table(x1, x2)
  x2
x1   0   1
  0 50 50
  1 50 50
> y = beta0 + beta1*x1 + beta2*x2 + rnorm(n, mean=0, sd=rmse)
> lm(y ~ x1 + x2)
```

#### 4.1.4 Referring to a range of variables

*Example:* 2.6.3

For functions such as `mean()` it is often desirable to list variables to be averaged without listing them all by name. SAS provides two ways of doing this. First, variables stored adjacently can be referred to as a range `vara -- varb` (with two hyphens). Variables with sequential numerical suffices can be referred to as a range `varname1 - varnamek` (with a single hyphen) regardless of the storage location. The key thing to bear in mind is that the part of the name before the number must be identical for all variables. This shorthand syntax also works in procedures.

No straightforward equivalent exists in R, though variables stored adjacently can be referred to by the index of their column number.

#### SAS

```
data ...;
  meanadjacentx = mean(of x1 -- xk);
  meannamedx = mean(of x1 - xk);
run;
```

*Note:* The former code will return the mean of all the variables stored between  $x_1$  and  $x_k$ . The latter will return the mean of  $x_1 \dots x_k$ , if they all exist.

#### 4.1.5 Perform an action repeatedly over a set of variables

*Examples:* 2.6.3 and 7.10.9

It is often necessary to perform a given function for a series of variables. Here the square of each of a list of variables is calculated as an example.

In SAS, this can be accomplished using arrays.

**SAS**

```

data ...;
  array arrayname1 [arraylen] x1 x2 ... xk;
  array arrayname2 [arraylen] z1 ... zk;
  do i = 1 to arraylen;
    arrayname2[i] = arrayname1[i]**2;
  end;
run;

```

*Note:* In the above example,  $z_i = x_i^2, i = 1 \dots k$ , for every observation in the dataset. The variable **arraylen** is an integer. It can be replaced by a \* symbol, which implies that the dimension of the array is to be calculated automatically by SAS from the number of elements. Elements (variables in the array) are listed after the brackets. Arrays can also be multidimensional, when multiple dimensions are specified (separated by commas) within the brackets. This can be useful, for example, when variables contain a matrix for each observation in the dataset.

Variables can be created by definition in the array statement, meaning that in the above code, the variable **x2** need not exist prior to the first **array** statement. The function **dim(arrayname1)** returns the number of elements in the array and can be used in place of the variable **arraylen** to loop over arrays declared with the \* syntax.

**R**

```

l1 = c("x1", "x2", ..., "xk")
l2 = c("z1", "z2", ..., "zk")
for (i in 1:length(l1)) {
  assign(l2[i], eval(as.name(l1[i]))^2)
}

```

*Note:* It is not straightforward to refer to objects within R without evaluating those objects. Assignments to R objects given symbolically can be made using the **assign()** function. Here a non-obvious use of the **eval()** function is used to evaluate an expression after the string value in **l1** is coerced to be a symbol. This allows the values of the character vectors **l1** and **l2** to be evaluated (see **help(assign)**, **eval()**, and **substitute()**).

#### 4.1.6 Grid of values

*Example: 12.6*

It may be useful to generate all combinations of two or more vectors.

**SAS**

```

data ds;
do x1 = 1 to 3;
  do x2 = "M","F";
    output;
  end;
end;
run;

proc print data=ds; run;

```

Obs	x1	x2
1	1	M
2	1	F
3	2	M
4	2	F
5	3	M
6	3	F

**R**

```
> expand.grid(x1=1:3, x2=c("M", "F"))
   x1 x2
1  1  M
2  2  M
3  3  M
4  1  F
5  2  F
6  3  F
```

*Note:* The `expand.grid()` function takes two or more vectors or factors and returns a data frame. The first factor varies fastest. The resulting object is a matrix.

### 4.1.7 Debugging

For SAS, we will discuss only the data step and macros. The SAS log is the first option for debugging all code and for procedures about the only method.

**SAS**

```
data ds;
...
put ...;
or
data ds / debug;
...
```

*Note:* The first very basic option is to print values from the data step into the log, using the `put` statement. For macros, the equivalent is the `%put` statement. A more formal option for data steps is the `debug` option, which opens a debugging tool. See the on-line help: SAS Products; Base SAS; Base SAS Utilities: Reference; DATA Step Debugger. There is no equivalent for macros, but two system options can be particularly useful in finding errors in macros. An `options mprint;` statement will cause the statements that the macro generates to be printed in the log. The `symbolgen` option will report the value of macro variables in the log whenever they are called in the code.

**R**

```
browser() # create a breakpoint
debug(function) # enter the debugger when function called
```

*Note:* When a function flagged for debugging is called, the function can be executed one statement at a time. At the prompt, commands can be entered (`n` for next, `c` for continue, `where` for traceback, `Q` for quit) or expressions can be evaluated (see `browser()` and `trace()`). A debugging environment is available within RStudio. The debugger may be invoked by setting a breakpoint by clicking to the left of the line number in an R script, or pressing Shift+F9. Profiling of the execution of R expressions can be undertaken using the `Rprof()` function (see also `summaryRprof()` and `tracemem()`). RStudio provides a series of additional debugging tools.

### 4.1.8 Error recovery

In SAS, there is no general method to assess whether a command functioned correctly. Instead, to determine this kind of thing programmatically, one should submit code and examine the expected results to determine if they match expectations. Useful tools in this

setting include the `call symput` function (4.2.1) which can generate macro variables from within data steps.

## R

```
try(expression, silent=FALSE)
stopifnot(expr1, ..., exprk)
```

*Note:* The `try()` function runs the given `expression` and traps any errors that may arise (displaying them on the standard error output device). The function `geterrmessage()` can be used to display any errors. The `stopifnot()` function runs the given expressions and returns an error message if all are not true (see `stop()` and `message()`).

## 4.2 Functions and macros

A strength of both R and SAS is their extensibility. In this section, we provide an introduction to defining and calling functions and macros.

### 4.2.1 SAS macros

SAS does not provide a simple means for constructing functions which can be integrated with procedures, although the `fcmp` procedure can be useful for shortening data step code, as demonstrated in <http://tinyurl.com/sasrblog-poisson>. However, SAS does provide a text-replacement capability called the SAS Macro Language, which can simplify and shorten code. The language also includes looping capabilities. We demonstrate a simple macro to change the predictor in a simple linear regression example.

```
%macro reg1 (pred=);
  proc reg data=ds;
    model y = &pred;
  run;
%mend reg1;
```

In this example, we define a new macro (named `reg1`), which accepts a single parameter that will be passed to it in the macro call. When we refer to `&pred` within the macro, it will be replaced with the submitted text when the macro is called. In this case, the macro will run `proc reg` (6.1.1) with the submitted text interpreted as the name of the predictor of the outcome `y`. This macro would be called as follows, for example, if the predictor were named `x1`.

```
%reg1(pred=x1);
```

When the `%macro reg1...` statements and the `%reg1(pred=x1)` statement are run, SAS executes the following.

```
proc reg data=ds;
  model y = x1;
run;
```

If four separate regressions were required, they could then be run in four statements.

```
%reg1(pred=x1);
%reg1(pred=x2);
%reg1(pred=x3);
%reg1(pred=x4);
```

As with the Output Delivery System A.7, SAS macros are a much broader topic than can be fully explored here. For a full introduction to their uses and capabilities, see the on-line help: Contents; SAS Products; Base SAS; SAS Macro Language: Reference. Further examples of SAS macros can be found in 11.4.2, 12.4.4, and 12.1.3.

## 4.2.2 R functions

A new function in R is defined by the syntax `function(arglist) body`. The `body` is made up of a series of R commands (or expressions), typically separated by line breaks and enclosed in curly braces. Here, we create a function to calculate the estimated confidence interval (CI) for a mean, as in 5.1.6.

```
# calculate a t confidence interval for a mean
ci.calc = function(x, ci.conf=.95) {
  sampsize = length(x)
  tcrit = qt(1-((1-ci.conf)/2), sampsize - 1)
  mymean = mean(x)
  mysd = sd(x)
  return(list(civals=c(mymean-tcrit*mysd/sqrt(sampsize),
    mymean+tcrit*mysd/sqrt(sampsize)), ci.conf=ci.conf))
}
```

Here the appropriate quantile of the T distribution is calculated using the `qt()` function, and the appropriate confidence interval is calculated and returned as a list. The function is stored in the object `ci.calc`, which can then be used like any other function. For example, `ci.calc(x1)` will print the CI and confidence level for the object `x1`. We also demonstrate the syntax for providing a default value, so that the confidence level in the preceding example is 0.95. User-written functions nest just as pre-existing functions do: `ci.conf(rnorm(100), 0.9)` will return the CI and report that the confidence limit is 0.9 for 100 normal random variates. In this example, we explicitly `return()` a list of return values. If no return statement is provided, the results of the last expression evaluation are returned.

```
> ci.calc(x)
$civals
[1] 0.624 12.043
$ci.conf
[1] 0.95
```

If only the lower confidence interval is needed, this can be saved as an object.

```
> lci = ci.calc(x)$civals[1]
> lci
[1] 0.624
```

The default confidence level is 95%; this can be changed by specifying a different value.

```
> ci.calc(x, ci.conf=.90)
$civals
[1] 1.799 10.867

$ci.conf
[1] 0.9
```

This is equivalent to running `ci.calc(x, .90)`, since `ci.conf` is the second argument to the function.

Other examples of defined R functions can be found in 2.2.18 and 5.7.4.

## 4.3 Interactions with the operating system

### 4.3.1 Timing commands

See also 2.4.6 (time variables).

**SAS**

```
options stimer;
options fullstimer;
```

*Note:* These options request that a subset (`stimer`) or all available (`fullstimer`) statistics are reported in the SAS log. We are not aware of a simple way to get the statistics except by reading the log.

**R**

```
system.time(expression)
```

*Note:* The `expression` (e.g., call to any user or system defined function, see B.4.1) given as argument to the `system.time()` function is evaluated, and the user, system, and total (elapsed) time are returned (see `proc.time()`).

### 4.3.2 Suspend execution for a time interval

**SAS**

```
data ds;
  sleeping=sleep(numseconds);
...
```

*Note:* If SAS is running interactively through its GUI, a small window will appear with the scheduled awakening time.

**R**

```
Sys.sleep(numseconds)
```

*Note:* The command `Sys.sleep()` will pause execution for `numseconds`, with minimal system impact.

### 4.3.3 Execute a command in the operating system

**SAS**

```
x;
or
x 'OS command';
or
data ...;
  call system("OS command");
run;
```

*Note:* An example `OS command` in Windows would be `x 'dir'`. The statement consisting of just `x` will open a command window. Related statements are `x1, x2, ..., x9`, which allow up to nine separate operating system tasks to be executed simultaneously.

The `x` command need not be in a data step and cannot be executed conditionally. In other words, if it appears as a consequence in an `if` statement, it will be executed regardless of whether or not the test in the `if` statement is true. Use the `call system` statement as shown to execute conditionally. This syntax to open a command window may not be available in all operating systems.

**R**

```
system("ls")
```

*Note:* The command `ls` lists the files in the current working directory (see 4.3.7 to capture this information). When R is running under Windows, the `shell()` command can be used to start a command window.

### 4.3.4 Command history

The SAS log contains all processed SAS code, as well as notes on execution. There is no separate record of the commands without the notes. The SAS log is one of the output files when SAS is run in batch.

**R**

```
savehistory()
loadhistory()
history()
```

*Note:* The command `savehistory()` saves the history of recent commands, which can be re-loaded using `loadhistory()` or displayed using `history()`. The `timestamp()` function can be used to add a date and time stamp to the history.

### 4.3.5 Find working directory

**SAS**

```
x;
```

*Note:* This will open a command window; the current directory in this window is the working directory. The working directory can also be found using the method shown in 4.3.7 using the `cd` command in Windows or the `pwd` command in Linux.

The current directory is displayed by default in the status line at the bottom of the SAS window.

**R**

```
getwd()
```

*Note:* The command `getwd()` displays the current working directory.

### 4.3.6 Change working directory

**SAS**

```
/* Windows, Unix */
x 'cd dir_location';
```

*Note:* This can also be done interactively by double-clicking the display of the current directory in the status line at the bottom of the SAS window.

**R**

```
setwd("dir_location")
```

*Note:* The command `setwd()` changes the current working directory to the (absolute or relative) pathname given as argument (see `file.choose()`). This can also be done interactively under Windows and Mac OS X by selecting the **Change Working Directory** option under the **Misc** menu, or similar options on the **Session** menu in RStudio.

### 4.3.7 List and access files

#### SAS

```
filename filehandle pipe 'dir /b'; /* Windows */
filename filehandle pipe 'ls'; /* Unix or Mac OS X */

data ds;
  infile filehandle truncover;
  input x $20.;
run;
```

*Note:* The `pipe` is a special file type which passes the characters inside the single quote to the operating system when read using the `infile` statement, then reads the result. The above code lists the contents of the current directory. The dataset `ds` contains a single character variable `x` with the file names. The file handle can be no longer than eight characters.

#### R

```
list.files()
```

*Note:* The `list.files()` command returns a character vector of file names in the current directory (by default). Recursive listings are also supported. The function `file.choose()` provides an interactive file browser and can be given as an argument to functions such as `read.table()` (1.1.2) or `read.csv()` (1.1.5). Related file operation functions include `file.access()`, `file.exists()`, `file.info()` (see `help(files)`) and `Sys.glob()` for wild-card expansion).



# Chapter 5

## Common statistical procedures

This chapter describes how to generate univariate summary statistics (such as means, variances, and quantiles) for continuous variables, display and analyze frequency tables and cross-tabulations of categorical variables, and carry out a variety of one and two sample procedures.

### 5.1 Summary statistics

#### 5.1.1 Means and other summary statistics

SAS

*Example: 5.7.1*

```
proc means data=ds keyword1 ... keywordn;
  var x1 ... xk;
run;
or
proc summary data=ds;
  var x1 ... xk;
  output out=newds keyword1= keyword2(x2)=newname
        keyword3(x3 x4)=newnamea newnameb;
run;

proc print data=newds;
run;
or
proc univariate data=ds;
  var x1 ... xk;
run;
```

*Note:* The **univariate** procedure generates a number of statistics by default, including the mean, standard deviation, skewness, and kurtosis.

The **means** and **summary** procedures accept a number of **keywords**, including **mean**, **median**, **var**, **stdev**, **min**, **max**, **sum**. These procedures are identical except that **proc summary** produces no printed output, only an output dataset, while **proc means** can produce both printed output and a dataset. The **output** statement syntax is **keyword=**, in which case the summary statistic shares the name of the variable summarized, **keyword(varname)=newname** in which case the summary statistic takes the new name, or **keyword(varname1 ... varnamek)= newname1 ... newnamek**, which allows the naming of many summary statistic variables at once. These options become valuable especially when

summarizing within subgroups (11.1). The `maxdec` option to the `proc means` statement controls the number of decimal places printed.

## R

```
mean(x)
```

or

```
library(mosaic)
favstats(x, data=ds)
```

*Note:* The `mean()` function accepts a numeric vector or a numeric dataframe as arguments (date objects are also supported). Similar functions in R include `median()` (see 5.1.4 for more quantiles), `var()`, `sd()`, `min()`, `max()`, `sum()`, `prod()`, and `range()` (note that the latter returns a vector containing the minimum and maximum value). The `weighted.mean()` function can be used to calculate weighted means. The `rowMeans()` and `rowSums()` functions (and their equivalents for columns) can be helpful for some calculations. The `favstats()` function in the `mosaic` package provides a concise summary of the distribution of a variable (including the number of observations and missing values).

### 5.1.2 Other moments

*Example:* 5.7.1

While skewness and kurtosis are less commonly reported than the mean and standard deviation, they can be useful at times. Skewness is defined as the third moment around the mean and characterizes whether the distribution is symmetric ( $\text{skewness}=0$ ). Kurtosis is a function of the fourth central moment. It characterizes peakedness, where the normal distribution has a value of 3 and smaller values correspond to a more rounded peak and shorter, thinner tails.

## SAS

```
ods output moments=moments_of_x;
proc univariate data=ds;
    var x;
run;
```

*Note:* In the code above, the the ODS system is used to make a dataset with the results from `proc univariate`. They will also be displayed in the output. SAS subtracts 3 from the kurtosis, so that the normal distribution has a kurtosis of 0 (this is sometimes called excess kurtosis).

## R

```
library(moments)
skewness(x)
kurtosis(x)
```

*Note:* The `moments` package facilitates the calculation of skewness and kurtosis within R as well as higher order moments (see `all.moments()`).

### 5.1.3 Trimmed mean

## SAS

```
proc univariate data=ds trimmed=frac;
    var x;
run;
```

*Note:* The parameter `frac` is the proportion of observations above and below the mean to exclude, or a number (greater than 1), in which case `number` observations will be excluded.

Multiple variables may be specified. This statistic can be saved into a dataset using **ODS** (see A.7).

## R

```
mean(x, trim=frac)
```

*Note:* The value **frac** can take on range 0 to 0.5 and specifies the fraction of observations to be trimmed from each end of **x** before the mean is computed (**frac**=0.5 yields the median).

### 5.1.4 Quantiles

#### SAS

```
proc univariate data=ds;
  var x1 ... xk;
  output out=newds pctlpts=2.5, 95 to 97.5 by 1.25
        pctlpre=p pctlnames=2_5 95 96_125 97_5;
run;
```

*Example:* 5.7.1

*Note:* This creates a new dataset with the 2.5, 95, 96.25, 97.5 values stored in variables named p2.5, p95, p96\_125, and p97\_5. The 1st, 5th, 10th, 25th, 50th, 75th, 90th, 95th, and 99th can be obtained more directly from **proc means**, **proc summary**, and **proc univariate**.

Details and options regarding calculation of quantiles in **proc univariate** can be found in SAS on-line help: Contents; SAS Products; SAS Procedures; UNIVARIATE; Details; Calculating Percentiles.

## R

```
quantile(x, c(.025, .975))
quantile(x, seq(from=.95, to=.975, by=.0125))
```

*Note:* Details regarding the calculation of quantiles in **quantile()** can be found using **help(quantile)**. The **ntiles()** function in the **mosaic** package can facilitate the creation of groups of roughly equal sizes.

### 5.1.5 Centering, normalizing, and scaling

#### SAS

```
proc standard data=ds out=ds2 mean=0 std=1;
  var x1 ... xk;
run;
```

*Note:* The output dataset named in the **out** option contains all of the data from the original dataset, with the standardized version of each variable named in the **var** statement stored in place of the original. Either the **mean** or the **std** option may be omitted.

## R

```
scale(x)
```

or

```
(x-mean(x))/sd(x)
```

*Note:* The default behavior of **scale()** is to create a Z-score transformation. The **scale()** function can operate on matrices and dataframes, and allows the specification of a vector of the scaling parameters for both center and scale (see **sweep()**, a more general function).

### 5.1.6 Mean and 95% confidence interval

#### SAS

```
proc means data=ds lclm mean uclm;
  var x;
run;
```

*Note:* Calculated statistics can be saved using an `output` statement or using `proc summary` as in 5.1.1 or using `ODS`.

#### R

```
tcrit = qt(.975, df=length(x)-1)
mean(x) + c(-tcrit, tcrit)*sd(x)/sqrt(length(x))
or
t.test(x)$conf.int
```

*Note:* While the appropriate 95% confidence interval can be generated in terms of the mean and standard deviation, it is more straightforward to use the `t.test()` function to calculate the relevant quantities.

### 5.1.7 Proportion and 95% confidence interval

#### SAS

*Example:* 11.2

```
proc freq data=ds;
  tables x / binomial;
run;
```

*Note:* The `binomial` option requests the exact Clopper–Pearson confidence interval based on the F distribution [26], an approximate confidence interval, and a test that the probability of the first level of the variable equals some value (= 0.5 by default). If  $x$  has more than two levels, the probability estimated and tested is the probability of the first level vs. all the others combined. Additional confidence intervals are available as options to the `binomial` option.

#### R

```
binom.test(sum(x), length(x))
prop.test(sum(x), length(x))
```

*Note:* The `binom.test()` function calculates an exact Clopper–Pearson confidence interval based on the F distribution [26] using the first argument as the number of successes and the second argument as the number of trials, while `prop.test()` calculates an approximate confidence interval by inverting the score test. Both allow specification of the probability under the null hypothesis. The `conf.level` option can be used to change the default confidence level.

### 5.1.8 Maximum likelihood estimation of parameters

*Example:* 5.7.1

See also 3.1.1 (probability density functions).

There are no generic functions for generating maximum likelihood estimates (MLEs) of distributions in SAS.

#### R

```
library(MASS)
fitdistr(x, "densityfunction")
```

*Note:* Options for `densityfunction` include `beta`, `cauchy`, `chi-squared`, `exponential`, `f`, `gamma`, `geometric`, `log-normal`, `lognormal`, `logistic`, `negative binomial`, `normal`, `Poisson`, `t`, or `weibull`.

## 5.2 Bivariate statistics

### 5.2.1 Epidemiologic statistics

SAS

```
proc freq data=ds;
  tables x*y / relrisk;
run;
```

*Example: 5.7.3*

R

```
sum(x==0&y==0)*sum(x==1&y==1)/(sum(x==0&y==1)*sum(x==1&y==0))
```

or

```
tab1 = table(x, y)
tab1[1,1]*tab1[2,2]/(tab1[1,2]*tab1[2,1])
```

or

```
glm1 = glm(y ~ x, family=binomial)
exp(glm1$coef[2])
```

or

```
library(epitools)
oddsratio.fisher(x, y)
oddsratio.wald(x, y)
riskratio(x, y)
riskratio.wald(x, y)
```

*Note:* The `epitab()` function in the `epitools` package provides a general interface to many epidemiologic statistics, while `expand.table()` can be used to create individual level data from a table of counts (see generalized linear models, 7.1).

### 5.2.2 Test characteristics

The sensitivity of a test is defined as the probability that someone with the disease ( $D=1$ ) tests positive ( $T=1$ ), while the specificity is the probability that someone without the disease ( $D=0$ ) tests negative ( $T=0$ ). For a dichotomous screening measure, the sensitivity and specificity can be defined as  $P(D = 1, T = 1)/P(D = 1)$  and  $P(D = 0, T = 0)/P(D = 0)$ , respectively (see also receiver operating characteristic curves, 8.5.7).

SAS

```
proc freq data=ds;
  tables d*t / out=newds;
run;

proc means data=newds nway;
  by d;
  var count;
  output out=newds2 sum=sumdlev;
run;
```

```

data newds3;
merge newds newds2;
  by d;
  retain sens spec;
  if D eq 1 and T=1 then sens=count/sumdlev;
  if D eq 0 and T=0 then spec=count/sumdlev;
  if sens ge 0 and spec ge 0;
run;

```

*Note:* The above code creates a dataset with a single line containing the sensitivity, specificity, and other data, given a test positive indicator *t* and disease indicator *d*. Sensitivity and specificity across all unique cut-points of a continuous measure *T* can be calculated as follows.

```

proc summary data=ds;
  var d;
  output out=sumdisease sum=totaldisease n=totalobs;
run;

proc sort data=ds;
  by descending t;
run;

data ds2;
set ds;
  if _n_ eq 1 then set sumdisease;
  retain sumdplus 0 sumdminus 0;
  sumdplus = sumdplus + d;
  sumdminus = sumdminus + (d eq 0);
  sens = sumdplus/totaldisease;
  one_m_spec = sumdminus/(totalobs - totaldisease);
run;

```

In the preceding code, `proc summary` (5.1.1) is used to find the total number with the disease and in the dataset, and to save this data in a dataset named `sumdisease`. The data is then sorted in descending order of the test score *t*. In the final step, the disease and total number of observations are read in and the current number of true positives and negatives accrued as the value of *t* decreases. The conditional use of the `set` statement allows the summary values for disease and subjects to be included for each line of the output dataset; the `retain` statement allows values to be kept across entries in the dataset and optionally allows the initial value to be set. The final dataset contains the sensitivity `sens` and 1 minus the specificity `one_m_spec`. This approach would be more complicated if tied values of the test score were possible.

## R

```

sens = sum(D==1&T==1)/sum(D==1)
spec = sum(D==0&T==0)/sum(D==0)

```

*Note:* Sensitivity and specificity for an outcome *D* can be calculated for each value of a continuous measure *T* using the following code.

```
library(ROCR)
pred = prediction(T, D)
diagobj = performance(pred, "sens", "spec")
spec = slot(diagobj, "y.values")[[1]]
sens = slot(diagobj, "x.values")[[1]]
cut = slot(diagobj, "alpha.values")[[1]]
diagmat = cbind(cut, sens, spec)
head(diagmat, 10)
```

*Note:* The `ROCR` package facilitates the calculation of test characteristics, including sensitivity and specificity. The `prediction()` function takes as arguments the continuous measure and outcome. The returned object can be used to calculate quantities of interest (see `help(performance)` for a comprehensive list). The `slot()` function is used to return the desired sensitivity and specificity values for each cut score, where `[[1]]` denotes the first element of the returned list (see `help(list)` and `help(Extract)`).

### 5.2.3 Correlation

SAS

```
proc corr data=ds;
  var x1 ... xk;
run;
```

*Examples:* 5.7.2 and 8.7.7

*Note:* Specifying `spearman` or `kendall` as an option to `proc corr` generates the Spearman or Kendall correlation coefficients, respectively. The `with` statement can be used to generate correlations only between the `var` and `with` variables, as in 5.7.2, rather than among all the `var` variables. This can save space as it avoids replicating correlations above and below the diagonal of the correlation matrix.

R

```
pearsoncorr = cor(x, y)
spearmancorr = cor(x, y, method="spearman")
kendalltau = cor(x, y, method="kendall")
or
```

```
cormat = cor(cbind(x1, ..., xk))
```

*Note:* Specifying `method="spearman"` or `method="kendall"` as an option to `cor()` generates the Spearman or Kendall correlation coefficients, respectively. A matrix of variables (created with `cbind()`) can be used to generate the correlation between a set of variables. To emulate the `with` statement in SAS, subsets of the returned correlation matrix can be selected, as demonstrated in 5.7.2. This can save space as it avoids replicating correlations above and below the diagonal of the correlation matrix. The `use` option for `cor()` specifies how missing values are handled (either `"all.obs"`, `"complete.obs"`, or `"pairwise.complete.obs"`). The `cor.test()` function can carry out a test (or calculate the confidence interval) for a correlation.

### 5.2.4 Kappa (agreement)

SAS

```
proc freq data=ds;
  tables x * y / agree;
run;
```

*Note:* The `agree` statement produces  $\kappa$  and weighted  $\kappa$  and their asymptotic standard errors and confidence intervals, as well as McNemar's test for  $2 \times 2$  tables and Bowker's test of symmetry for tables with more than two levels [18].

**R**

```
library(irr)
kappa2(data.frame(x, y))
```

*Note:* The `kappa2()` function takes a dataframe (see B.4.6) as argument. Weights can be specified as an option.

## 5.3 Contingency tables

### 5.3.1 Display cross-classification table

*Example: 5.7.3*

Contingency tables show the group membership across categorical (grouping) variables. They are also known as cross-classification tables, cross-tabulations, and two-way tables.

**SAS**

```
proc freq data=ds;
  tables x * y;
run;
```

*Note:* N-way tables can be generated with additional `* varname` terms.

**R**

```
mytab = table(y, x)
addmargins(mytab)
prop.table(mytab, 1)
or
xtabs(~ y + x)
or
library(mosaic)
tally(~ y + x, data=ds)
or
library(prettyR)
xtab(y ~ x, data=ds)
```

*Note:* The `exclude=NULL` option can be used with the `table()` function to include categories for missing values. The `addmargins()` function adds (by default) the row and column totals to a table, while `prop.table()` can be used to calculate row totals (with option 1) and column totals (with option 2). The `colSums()`, `colMeans()` functions (and their equivalents for rows) can be used to efficiently calculate sums and means for numeric vectors. The `xtabs()` function can be used to create a contingency table from cross-classifying factors. The `tally()` function in the `mosaic` package supports a modeling language for categorical tables, including a `|` operator to stratify by a third variable. Options for the `tally()` function include `format="percent"` or `format="proportion"`. Further configuration of table display is provided in the `prettyR` package `xtab()` function.

### 5.3.2 Displaying missing value categories in a table

**SAS**

```
proc freq data=ds;
  tables x1*x2 / missprint;
run;
```

**R**

```
table(x1, x2, useNA="ifany")
```

**5.3.3 Pearson chi-square statistic****SAS**

*Example: 5.7.3*

```
proc freq data=ds;
  tables x * y / chisq;
run;
```

*Note:* For  $2 \times 2$  tables the output includes both unadjusted and continuity-corrected tests.

**R**

```
chisq.test(x, y)
```

*Note:* The `chisq.test()` command can accept either two class vectors or a matrix with counts. By default a continuity correction is used (the option `correct=FALSE` turns this off). A version with more verbose output (e.g., expected cell counts) can be found in the `xchisq.test()` function in the `mosaic` package.

**5.3.4 Cochran–Mantel–Haenszel test**

The Cochran–Mantel–Haenszel test gives an assessment of the relationship between  $X_2$  and  $X_3$ , stratified by (or controlling for)  $X_1$ . The analysis provides a way to adjust for the possible confounding effects of  $X_1$  without having to estimate parameters for them.

**SAS**

```
proc freq data=ds;
  tables x1 * x2 * x3 / cmh;
run;
```

*Note:* The `cmh` option produces Cochran–Mantel–Haenszel statistics and, when both  $X_2$  and  $X_3$  have two values, it generates estimates of the common odds ratio, common relative risks, and the Breslow–Day test for homogeneity of the odds ratios. More complex models can be fit using the generalized linear model methodology described in Chapter 7.

**R**

```
mantelhaen.test(x2, x3, x1)
```

**5.3.5 Cramér's V**

Cramér's V (or phi coefficient) is a measure of association for nominal variables.

**SAS**

```
proc freq data=ds;
  tables x1 * x2 / chisq;
run;
```

*Note:* Cramér's V is added to the output when the `chisq` option is specified.

**R**

```
library(vcd)
assocstats(table(x, y))
```

### 5.3.6 Fisher's exact test

SAS

```
proc freq data=ds;
  tables x * y / exact;
run;
```

or

```
proc freq data=ds;
  tables x * y;
  exact fisher / mc n=bnum;
run;
```

*Example: 5.7.3*

*Note:* The former requests only the exact  $p$ -value; the latter generates a Monte Carlo  $p$ -value, an asymptotically equivalent test based on `bnum` random tables simulated using the observed margins. The Monte Carlo  $p$ -value can be considerably less compute-intensive for large sample sizes.

R

```
fisher.test(y, x)
```

or

```
fisher.test(ymat)
```

*Note:* The `fisher.test()` command can accept either two class vectors or a matrix with counts (here denoted by `ymat`). For tables with many rows and/or columns,  $p$ -values can be computed using Monte Carlo simulation using the `simulate.p.value` option. The Monte Carlo  $p$ -value can be considerably less compute-intensive for large sample sizes.

### 5.3.7 McNemar's test

McNemar's test tests the null hypothesis that the proportions are equal across matched pairs, for example, when two raters assess a population.

SAS

```
proc freq data=ds;
  tables x * y / agree;
run;
```

R

```
mcnemar.test(y, x)
```

*Note:* The `mcnemar.test()` command can accept either two class vectors or a matrix with counts.

## 5.4 Tests for continuous variables

### 5.4.1 Tests for normality

SAS

```
proc univariate data=ds normal;
  var x;
run;
```

*Note:* This generates the Shapiro–Wilk test, the Kolmogorov–Smirnov test, the Anderson–Darling test, and the Cramér–von Mises test.

**R**

```
shapiro.test(x)
```

*Note:* The `nortest` package includes a number of additional tests of normality.

**5.4.2 Student's *t* test**

*Example: 5.7.4*

**SAS**

```
proc ttest data=ds;
  class x;
  var y;
run;
```

*Note:* The variable  $X$  takes on two values. The output contains both equal and unequal-variance  $t$  tests, as well as a test of the null hypothesis of equal variance.

**R**

```
t.test(y1, y2)
or
t.test(y ~ x, data=ds)
```

*Note:* The first example for the `t.test()` command displays how it can take two vectors ( $y_1$  and  $y_2$ ) as arguments to compare, or in the latter example a single vector corresponding to the outcome ( $y$ ), with another vector indicating group membership ( $x$ ) using a formula interface (see B.4.7 and 6.1.1). By default, the two-sample  $t$  test uses an unequal variance assumption. The option `var.equal=TRUE` can be added to specify an equal variance assumption. The command `var.test()` can be used to formally test equality of variances.

**5.4.3 Test for equal variances**

The assumption of equal variances among the groups in analysis of variance and the two-sample  $t$  test can be assessed via Levene's test.

**SAS**

```
proc glm data=ds;
  class x;
  model y=x;
  means x / hovtest=levene;
run;
```

*Note:* Other options to assess equal variance include Bartlett's test, the Brown and Forsythe version of Levene's test, and O'Brien's test, which is effectively a modification of Levene's test.

**R**

```
library(lawstat)
levene.test(y, x, location="mean")
bartlett.test(y ~ x)
```

*Note:* Options to the `levene.test()` function provide other variants of the test.

### 5.4.4 Nonparametric tests

SAS

```
proc npar1way data=ds wilcoxon edf median;
  class y;
  var x;
run;
```

*Example: 5.7.4*

*Note:* Many tests can be requested as options to the `proc npar1way` statement. Here we show a Wilcoxon test, a Kolmogorov–Smirnov test, and a median test, respectively. Exact tests can be generated by using an `exact` statement with these names, e.g., the `exact median` statement will generate the exact median test.

R

```
wilcox.test(y1, y2)
ks.test(y1, y2)

library(coin)
median_test(y ~ x)
```

*Note:* By default, the `wilcox.test()` function uses a continuity correction in the normal approximation for the *p*-value. The `ks.test()` function does not calculate an exact *p*-value when there are ties. The median test shown will generate an exact *p*-value with the `distribution="exact"` option.

### 5.4.5 Permutation test

SAS

```
proc npar1way data=ds;
  class y;
  var x;
  exact scores=data;
run;
```

or

```
proc npar1way data=ds;
  class y;
  var x;
  exact scores=data / mc n=bnum;
run;
```

*Example: 5.7.4*

*Note:* Any test described in 5.4.4 can be named in place of `scores=data` to get an exact test based on those statistics. The `mc` option generates an empirical *p*-value (asymptotically equivalent to the exact *p*-value) based on `bnum` Monte Carlo replicates.

R

```
library(coin)
oneway_test(y ~ as.factor(x), distribution=approximate(B=bnum))
or
library(mosaic)
obs = t.test(y ~ x)$statistic
res = do(10000) * t.test(y ~ shuffle(x))
tally(~ res$stat > abs(obs))
```

*Note:* The `oneway_test` function in the `coin` package implements a variety of permutation-based tests (see the `exactRankTests` package). The `distribution=approximate` syntax generates an empirical *p*-value (asymptotically equivalent to the exact *p*-value) based on

`bnum` Monte Carlo replicates. The `do()` function along with the `shuffle()` functions in the `mosaic` package can also be used to undertake permutation tests (see the package's resampling vignette at CRAN for details).

### 5.4.6 Logrank test

*Example: 5.7.5*

See also 8.5.11 (Kaplan–Meier plot) and 7.5.1 (Cox proportional hazards model).

#### SAS

```
proc phreg data=ds;
  model timevar*cens(0) = x;
run;
or
proc lifetest data=ds;
  time timevar*cens(0);
  strata x;
run;
```

*Note:* If `cens` is equal to 0, then `proc phreg` and `proc lifetest` treat `time` as the time of censoring, otherwise it is the time of the event. The default output from `proc lifetest` includes the logrank and Wilcoxon tests. Other tests, corresponding to different weight functions, can be produced with the `test` option to the `strata` statement. These include `test=fleming( $\rho_1, \rho_2$ )`, a superset of the G-rho family of Fleming and Harrington [43], which simplifies to the G-rho family when  $\rho_2 = 0$ .

#### R

```
library(survival)
survdiff(Surv(timevar, cens) ~ x)
```

*Note:* Other tests within the G-rho family of Fleming and Harrington [43] are supported by specifying the `rho` option.

## 5.5 Analytic power and sample size calculations

Many simple settings lend themselves to analytic power calculations, where closed form solutions are available. Other situations may require an empirical calculation, where repeated simulation is undertaken (see 11.2).

It is straightforward to find power or sample size (given a desired power) for two sample comparisons of either continuous or categorical outcomes. We show simple examples for comparing means and proportions in two groups and supply additional information on analytic power calculation available for more complex methods.

#### SAS

```
/* find sample size for two-sample t-test */
proc power;
  twosamplemeans groupmeans=(0 0.5) stddev=1 power=0.9 ntotal=.;
run;

/* find power for two-sample t-test */
proc power;
  twosamplemeans groupmeans=(0 0.5) stddev=1 power=. ntotal=200;
run;
```

The latter call generates the following output:

```
The POWER Procedure
Two-sample t Test for Mean Difference
    Fixed Scenario Elements
Distribution          Normal
Method               Exact
Group 1 Mean         0
Group 2 Mean         0.5
Standard Deviation   1
Total Sample Size    200
Number of Sides      2
Null Difference      0
Alpha                0.05
Group 1 Weight       1
Group 2 Weight       1

Computed Power
Power 0.940
/* find sample size for two-sample test of proportions */
proc power;
    twosamplefreq test=pchi ntotal=. groupproportions=(.1 .2) power=0.9;
run;
/* find power for two-sample test of proportions */
proc power;
    twosamplefreq test=pchi ntotal=200 groupproportions=(.1 .2) power=.;
run;
```

*Note:* The power procedure also allows power calculations for the Wilcoxon rank-sum test, the log-rank and related tests for censored data, paired tests of means and proportions, correlations, and for ANOVA and linear and logistic regression. The syntax is similar with the desired output of power, total sample size, effect size, alpha level, or variance listed with a missing value (a period after the equals sign).

## R

```
# find sample size for two-sample t-test
power.t.test(delta=0.5, power=0.9)

# find power for two-sample t-test
power.t.test(delta=0.5, n=100)
```

The latter call generates the following output.

```
Two-sample t test power calculation
    n = 100
    delta = 0.5
    sd = 1
    sig.level = 0.05
    power = 0.9404272
    alternative = two.sided
NOTE: n is number in *each* group

# find sample size for two-sample test of proportions
power.prop.test(p1=.1, p2=.2, power=.9)

# find power for two-sample test of proportions
power.prop.test(p1=.1, p2=.2, n=100)
```

*Note:* The `power.t.test()` function requires exactly four of the five arguments (sample size in each group, power, difference between groups, standard deviation, and significance level) to be specified. Default values exist for `sd=1` and `sig.level=0.05`. Other power calculation functions can be found in the `pwr` package.

## 5.6 Further resources

Comprehensive introductions to using SAS to fit common statistical models can be found in [25] and [32]. Similar methods in R are accessibly presented in [187]. A readable introduction to permutation-based inference can be found in [55]. A vignette on resampling-based inference using R can be found at <http://cran.r-project.org/web/packages/mosaic/vignettes/Resampling.pdf>. Collett [27] provides an accessible introduction to survival analysis.

## 5.7 Examples

To help illustrate the tools presented in this chapter, we apply many of the entries to the HELP data. SAS and R code can be downloaded from <http://www.amherst.edu/~nhorton/sasr2/examples>.

### 5.7.1 Summary statistics and exploratory data analysis

We begin by reading the dataset.

```
filename myurl
  url 'http://www.amherst.edu/~nhorton/sasr2/datasets/help.csv' lrecl=704;

proc import
  datafile=myurl
  out=ds dbms=dlm;
  delimiter=',';
  getnames=yes;
run;
```

The `lrecl` statement is needed due to the long lines in the csv file.

```
> options(digits=3)
> options(width=72) # narrows output to stay in the grey box
> ds = read.csv("http://www.amherst.edu/~nhorton/sasr2/datasets/help.csv")
```

A first step would be to examine some univariate statistics (5.1.1) for the baseline CESD (Center for Epidemiologic Statistics measure of depressive symptoms) score. In SAS, univariate statistics are produced by `proc univariate`, `proc means`, and others.

```
options ls=70; * narrow output to stay in grey box;
proc means data=ds maxdec=2 min p5 q1 median q3 p95 max
    mean std range skew kurtosis;
var cesd;
run;
```

The MEANS Procedure

Analysis Variable : cesd

Minimum	5th Pctl	Lower Quartile	Median	Upper Quartile
1.00	10.00	25.00	34.00	41.00

Analysis Variable : cesd

95th Pctl	Maximum	Mean	Std Dev	Range
53.00	60.00	32.85	12.51	59.00

Analysis Variable : cesd

Skewness	Kurtosis
-0.26	-0.44

In R, we can use functions which produce a set of statistics, such as `fivenum()`, or request them singly.

```
> with(ds, mean(cesd))
[1] 32.8

> library(mosaic)
> mean(~ cesd, data=ds)

[1] 32.8

> with(ds, median(cesd))
[1] 34

> with(ds, range(cesd))
[1] 1 60

> sd(~ cesd, data=ds)
[1] 12.5

> var(~ cesd, data=ds)
[1] 157

> favstats(~ cesd, data=ds)
   min Q1 median Q3 max mean    sd    n missing
   1 25      34  41  60 32.8 12.5 453      0

> library(moments)
> with(ds, skewness(cesd))

[1] -0.26

> with(ds, kurtosis(cesd))

[1] 2.55
```

We can also generate desired quantiles. Here, we find the deciles (5.1.4).

```

ods select none;
proc univariate data=ds;
  var cesd;
  output out=deciles pctlpts= 0 to 100 by 10 pctlpre=p_;
run;
ods select all;

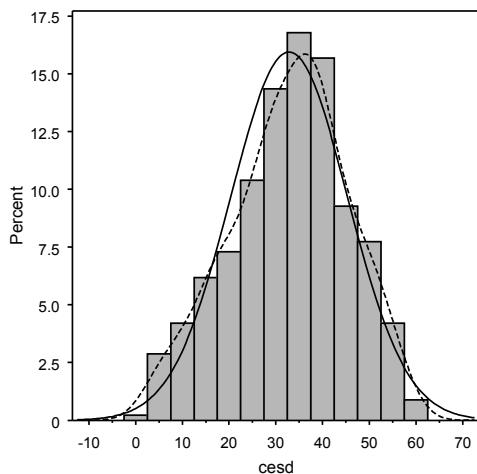
options ls=74;
proc print data=deciles;
run;

Obs  p_0   p_10  p_20  p_30  p_40  p_50  p_60  p_70  p_80  p_90  p_100
1    1     15    22    27    30    34    37    40    44    49    60
> with(ds, quantile(cesd, seq(from=0, to=1, length=11)))

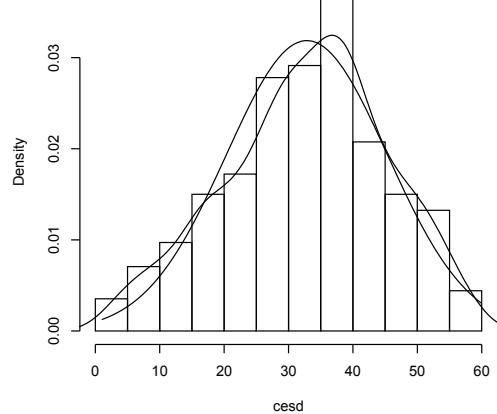
 0%  10%  20%  30%  40%  50%  60%  70%  80%  90% 100%
 1.0 15.2 22.0 27.0 30.0 34.0 37.0 40.0 44.0 49.0 60.0

```

Graphics can allow us to easily review the whole distribution of the data. Here we generate a histogram (8.1.4) of CESD, overlaid with its empirical PDF (8.1.5) and the closest-fitting normal distribution (see Figure 5.1). In SAS, the other results of `proc univariate` have been suppressed by selecting only the graphics output using an `ods select` statement (note the different y-axes generated).



(a) SAS



(b) R

Figure 5.1: Density plot of depressive symptom scores (CESD) plus superimposed histogram and normal distribution

```

ods select histogram;
proc univariate data=ds;
  var cesd;
  histogram cesd / normal (color=black l=1) kernel(color=black l=21)
                 cfill=greyCC;
run; quit;
ods select all;

> with(ds, hist(cesd, main="", freq=FALSE))
> with(ds, lines(density(cesd), main="CESD", lty=2, lwd=2))
> xvals = with(ds, seq(from=min(cesd), to=max(cesd), length=100))
> with(ds, lines(xvals, dnorm(xvals, mean(cesd), sd(cesd)), lwd=2))

```

## 5.7.2 Bivariate relationships

We can calculate the correlation (5.2.3) between CESD and MCS and PCS (mental and physical component scores). First, we show the default correlation matrix.

```

ods select pearsoncorr;
proc corr data=ds;
  var cesd mcs pcs;
run;

```

The CORR Procedure

```

Pearson Correlation Coefficients, N = 453
  Prob > |r| under H0: Rho=0

          cesd        mcs        pcs
cesd      1.00000    -0.68192    -0.29270
          <.0001      <.0001
mcs      -0.68192    1.00000    0.11046
          <.0001      0.0187
pcs      -0.29270    0.11046    1.00000
          <.0001      0.0187

> cormat = cor(with(ds, cbind(cesd, mcs, pcs)))
> cormat

      cesd     mcs     pcs
cesd  1.000 -0.682 -0.293
mcs   -0.682  1.000  0.110
pcs   -0.293  0.110  1.000

```

To save space, we can just print a subset of the correlations.

```
ods select pearsoncorr;
proc corr data=ds;
  var mcs pcs;
  with cesd;
run;
```

### The CORR Procedure

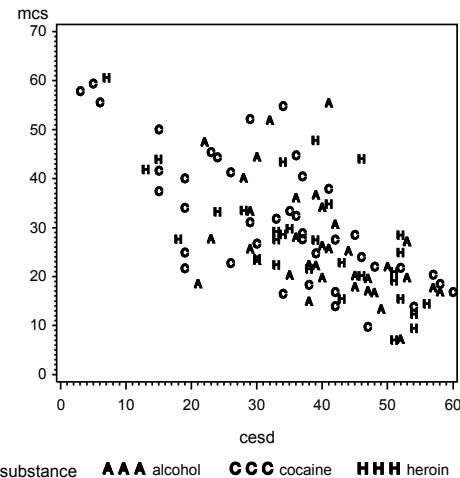
Pearson Correlation Coefficients, N = 453  
 Prob > |r| under H0: Rho=0

	mcs	pcs
cesd	-0.68192 <.0001	-0.29270 <.0001

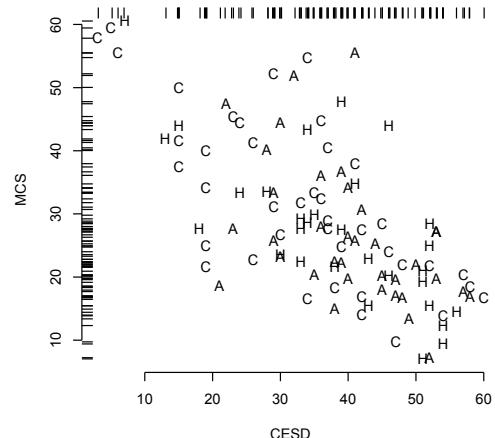
```
> cormat[c(2, 3), 1]
```

```
mcs      pcs
-0.682 -0.293
```

Figure 5.2 displays a scatterplot (8.3.1) of CESD and MCS, for the female subjects. The plotting character (9.1.2) is the primary substance (Alcohol, Cocaine, or Heroin). For R, a rug plot (9.1.8) is added to help demonstrate the marginal distributions; this is nontrivial in SAS.



(a) SAS



(b) R

Figure 5.2: Scatterplot of CESD and MCS for women, with primary substance shown as the plot symbol

```

symbol1 font=swiss v='A' h=.7 c=black;
symbol2 font=swiss v='C' h=.7 c=black;
symbol3 font=swiss v='H' h=.7 c=black;
proc gplot data=ds;
  where female=1;
  plot mcs*cesd=substance;
run; quit;

> with(ds, plot(cesd[female==1], mcs[female==1], xlab="CESD", ylab="MCS",
  type="n", bty="n"))
> with(ds, text(cesd[female==1&substance=="alcohol"],
  mcs[female==1&substance=="alcohol"], "A"))
> with(ds, text(cesd[female==1&substance=="cocaine"],
  mcs[female==1&substance=="cocaine"], "C"))
> with(ds, text(cesd[female==1&substance=="heroin"],
  mcs[female==1&substance=="heroin"], "H"))
> with(ds, rug(jitter(mcs[female==1]), side=2))
> with(ds, rug(jitter(cesd[female==1]), side=3))

```

### 5.7.3 Contingency tables

Here we display the cross-classification (contingency) table (5.3.1) of homeless at baseline by gender, calculate the observed odds ratio (OR, 5.2.1), and assess association using the Pearson  $\chi^2$  test (5.3.3) and Fisher's exact test (5.3.6). In SAS, this can be done with one call to `proc freq`.

```

proc freq data=ds;
  tables homeless*female / chisq exact relrisk;
run; quit;

```

The FREQ Procedure

Table of homeless by female

		female		Total
		0	1	
homeless	Frequency			
	Percent			
Row Pct				
Col Pct	0	1	Total	
0	177	67	244	
	39.07	14.79	53.86	
	72.54	27.46		
	51.16	62.62		
1	169	40	209	
	37.31	8.83	46.14	
	80.86	19.14		
	48.84	37.38		
Total	346	107	453	
	76.38	23.62	100.00	

**Statistics for Table of homeless by female**

Statistic	DF	Value	Prob
<hr/>			
Chi-Square	1	4.3196	0.0377
Likelihood Ratio Chi-Square	1	4.3654	0.0367
Continuity Adj. Chi-Square	1	3.8708	0.0491
Mantel-Haenszel Chi-Square	1	4.3101	0.0379
Phi Coefficient		-0.0977	
Contingency Coefficient		0.0972	
Cramer's V		-0.0977	

Fisher's exact test is provided by default with  $2 \times 2$  tables, so the `exact` statement is not required. The exact test result is shown.

**Statistics for Table of homeless by female**
**Fisher's Exact Test**

Cell (1,1) Frequency (F)	177
Left-sided Pr <= F	0.0242
Right-sided Pr >= F	0.9861
Table Probability (P)	0.0102
Two-sided Pr <= P	0.0456

**Statistics for Table of homeless by female**
**Estimates of the Relative Risk (Row1/Row2)**

Type of Study	Value	95% Confidence Limits	
<hr/>			
Case-Control (Odds Ratio)	0.6253	0.4008	0.9755
Cohort (Col1 Risk)	0.8971	0.8105	0.9930
Cohort (Col2 Risk)	1.4347	1.0158	2.0265

In R, the `tally()` function from the `mosaic` package displays contingency tables.

```
> tally(~ homeless + female, data=ds)

      female
homeless   0   1 Total
  0     177  67  244
  1     169  40  209
Total  346 107  453

> tally(~ homeless + female, format="percent", data=ds)

      female
homeless      0      1 Total
  0     39.07 14.79 53.86
  1     37.31  8.83 46.14
Total  76.38 23.62 100.00

> tally(~ homeless / female, data=ds)

      female
homeless      0      1
  0     0.512 0.626
  1     0.488 0.374
Total  1.000 1.000
```

We can easily calculate the odds ratio directly.

```
> or = with(ds, (sum(homeless==0 & female==0) *
+               sum(homeless==1 & female==1)) /
+             (sum(homeless==0 & female==1) *
+              sum(homeless==1 & female==0)))
> or
[1] 0.625
> library(epitools)
> oddsobject = with(ds, oddsratio.wald(homeless, female))
> oddsobject$measure

odds ratio with 95% C.I.
Predictor estimate lower upper
  0     1.000    NA    NA
  1     0.625  0.401  0.975

> oddsobject$p.value

two-sided
Predictor midp.exact fisher.exact chi.square
  0          NA          NA          NA
  1     0.0381     0.0456     0.0377
```

The  $\chi^2$  and Fisher's exact tests are fit in R using separate commands.

```
> chisqval = with(ds, chisq.test(homeless, female, correct=FALSE))
> chisqval

Pearson's Chi-squared test

data: homeless and female
X-squared = 4.32, df = 1, p-value = 0.03767

> with(ds, fisher.test(homeless, female))

Fisher's Exact Test for Count Data

data: homeless and female
p-value = 0.0456
alternative hypothesis: true odds ratio is not equal to 1
95 percent confidence interval:
0.389 0.997
sample estimates:
odds ratio
0.626
```

A graphical depiction of a table can be created: this can helpful as part of automated report generation and reproducible analysis (see 11.3). In SAS, this is most easily done using ODS output destinations and styles. Here we demonstrate the `html` destination and the `grayscaleprinter` style. A list of styles can be printed with `proc template; list styles; run;`.

```
ods html style=styles.grayscaleprinter;
proc freq data=ds;
  tables racegrp*substance / nocol norow nopercent;
run;
ods html close;

> library(gridExtra)
> mytab = tally(~ racegrp + substance, data=ds)
> plot.new()
> grid.table(mytab)
```

The results are shown in Figure 5.3.

Table of RACEGRP by SUBSTANCE				
RACEGRP	SUBSTANCE			
	alcohol	cocaine	heroin	Total
black	55	125	31	211
hispanic	17	7	26	50
other	9	7	10	26
white	96	13	57	166
<b>Total</b>	<b>177</b>	<b>152</b>	<b>124</b>	<b>453</b>

(a) SAS

	alcohol	cocaine	heroin	Total
black	55	125	31	211
hispanic	17	7	26	50
other	9	7	10	26
white	96	13	57	166
<b>Total</b>	<b>177</b>	<b>152</b>	<b>124</b>	<b>453</b>

(b) R

Figure 5.3: Graphical display of the table of substance by race/ethnicity

### 5.7.4 Two sample tests of continuous variables

We can assess gender differences in baseline age using a *t* test (5.4.2) and nonparametric procedures.

```
options ls=74; /* narrows output to stay in the grey box */
```

```
proc ttest data=ds;
  class female;
  var age;
run;
```

Variable: age

female	N	Mean	Std Dev	Std Err	Minimum	Maximum
0	346	35.4682	7.7501	0.4166	19.0000	60.0000
1	107	36.2523	7.5849	0.7333	21.0000	58.0000
Diff (1-2)		-0.7841	7.7116	0.8530		
female		Method		Mean	95% CL Mean	Std Dev
0			35.4682	34.6487 36.2877		7.7501
1			36.2523	34.7986 37.7061		7.5849
Diff (1-2)	Pooled		-0.7841	-2.4605 0.8923		7.7116
Diff (1-2)	Satterthwaite		-0.7841	-2.4483 0.8800		
female		Method		95% CL	Std Dev	
0			7.2125	8.3750		
1			6.6868	8.7637		
Diff (1-2)	Pooled		7.2395	8.2500		
Diff (1-2)	Satterthwaite					
Method		Variances		DF	t Value	Pr >  t
Pooled		Equal	451	-0.92	0.3585	
Satterthwaite		Unequal	179.74	-0.93	0.3537	
Equality of Variances						
Method	Num DF	Den DF	F Value	Pr > F		
Folded F	345	106	1.04	0.8062		

```
> ttres = t.test(age ~ female, data=ds)
> print(ttres)

Welch Two Sample t-test

data: age by female
t = -0.93, df = 180, p-value = 0.3537
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
-2.45 0.88
sample estimates:
mean in group 0 mean in group 1
35.5 36.3
```

The `names()` function can be used to identify the objects returned by the `t.test()` function (not displayed).

A permutation test can be run and used to generate a Monte Carlo  $p$ -value (5.4.5).

```
ods select datascoresmc;
proc npar1way data=ds;
  class female;
  var age;
  exact scores=data / mc n=9999 alpha=.05;
run;
ods exclude none;
```

#### Monte Carlo Estimates for the Exact Test

One-Sided Pr >= S	
Estimate	0.1830
95% Lower Conf Limit	0.1754
95% Upper Conf Limit	0.1906

Two-Sided Pr >=  S - Mean	
Estimate	0.3626
95% Lower Conf Limit	0.3532
95% Upper Conf Limit	0.3721

Number of Samples	9999
Initial Seed	221855001

```
> library(coin)
> oneway_test(age ~ as.factor(female),
  distribution=approximate(B=9999), data=ds)
```

#### Approximative 2-Sample Permutation Test

```
data: age by as.factor(female) (0, 1)
Z = -0.919, p-value = 0.3596
alternative hypothesis: true mu is not equal to 0
```

Both the Wilcoxon test and the Kolmogorov–Smirnov test (5.4.4) can be run with a single call to `proc freq`. Later, we'll include the D statistic from the Kolmogorov–Smirnov test and the associated  $p$ -value in a figure title. We'll use `ODS` to create a dataset containing these values.

```

ods output kolsmir2stats=age_female_ks_stats;
ods select wilcoxontest kolsmir2stats;
proc npar1way data=ds wilcoxon edf;
  class female;
  var age;
run;
ods select all;

```

#### Wilcoxon Two-Sample Test

Statistic 25288.5000

#### Normal Approximation

Z	0.8449
One-Sided Pr > Z	0.1991
Two-Sided Pr >  Z	0.3981

#### t Approximation

One-Sided Pr > Z	0.1993
Two-Sided Pr >  Z	0.3986

Z includes a continuity correction of 0.5.

#### Kolmogorov-Smirnov Two-Sample Test (Asymptotic)

KS	0.026755	D	0.062990
KSa	0.569442	Pr > KSa	0.9020

In R, these tests are obtained in separate function calls (see 5.4.4).

```
> with(ds, wilcox.test(age ~ as.factor(female), correct=FALSE))
```

#### Wilcoxon rank sum test

```

data: age by as.factor(female)
W = 17512, p-value = 0.3979
alternative hypothesis: true location shift is not equal to 0

```

```
> ksres = with(ds, ks.test(age[female==1], age[female==0]))
> print(ksres)
```

#### Two-sample Kolmogorov-Smirnov test

```

data: age[female == 1] and age[female == 0]
D = 0.063, p-value = 0.902
alternative hypothesis: two-sided

```

We can also plot estimated density functions (8.1.5) for age for both groups, and shade some areas (9.1.13) to emphasize how they overlap. SAS proc univariate with a by statement will generate density estimates for each group, but not overlay them. To get results similar to those available through R, we first generate the density estimates using proc kde (8.1.5) (suppressing all printed output).

```

proc sort data=ds;
  by female;
run;

ods select none;
proc kde data=ds;
  by female;
  univar age / out=kdeout;
run;
ods select all;

```

Next, we'll review the proc npar1way output which was saved as a dataset.

```
proc print data=age_female_ks_stats; run;
```

	V	c	n		c	n		
a								
r	L	V	V	L	V	V		
i	N	a	a	N	a	a		
a	a	b	l	a	b	l		
0	b	m	e	u	m	u		
b	l	e	l	e	l	e		
s	e	1	1	1	2	2		
1	age _KS_	KS	0.026755	0.026755	_D_	D	0.062990	0.062990
2	age _KSA_	KSa	0.569442	0.569442	P_KSA	Pr > KSa	0.9020	0.901979

Running proc contents (2.1.2, results not shown) reveals that the variable names prepended with 'c' are character variables. To get these values into a figure title, we use SAS Macro variables (A.8) created by the call symput function (4.2.1).

```

data _null_;
set age_female_ks_stats;
  if label2 eq 'D' then call symput('dvalue', substr(cvalue2, 1, 5));
    /* This makes a macro variable (which is saved outside any dataset)
       from a value in a dataset */
  if label2 eq 'Pr > KSa' then
    call symput('pvalue', substr(cvalue2, 1, 4));
run;

```

Finally, we construct the plot using proc gplot for the data with a title statement to include the Kolmogorov–Smirnov test results.

```

symbol1 i=j w=5 l=1 v=none c=black;
symbol2 i=j w=5 l=2 v=none c=black;
title "Test of ages: D=&dvalue p=&pvalue";
pattern1 color=grayBB;
proc gplot data=kdeout;
  plot density*value = female / legend areas=1 haxis=18 to 60 by 2;
run; quit;

```

In this code, the areas option to the plot statement makes SAS fill in the area under the first curve, while the pattern statement describes what color to fill in with.

In R, we create a function (4.2.2) to automate this task.

```
> plotdens = function(x,y, mytitle, mylab) {
  densx = density(x)
  densy = density(y)
  plot(densx, main=mytitle, lwd=3, xlab=mylab, bty="l")
  lines(densy, lty=2, col=2, lwd=3)
  xvals = c(densx$x, rev(densy$x))
  yvals = c(densx$y, rev(densy$y))
  polygon(xvals, yvals, col="gray")
}
```

The `polygon()` function is used to fill in the area between the two curves.

```
> mytitle = paste("Test of ages: D=", round(ksres$statistic, 3),
  " p=", round(ksres$p.value, 2), sep="")
> with(ds, plotdens(age[female==1], age[female==0], mytitle=mytitle,
  mylab="age (in years)"))
> legend(50, .05, legend=c("Women", "Men"), col=1:2, lty=1:2, lwd=2)
```

Results are shown in Figure 5.4.

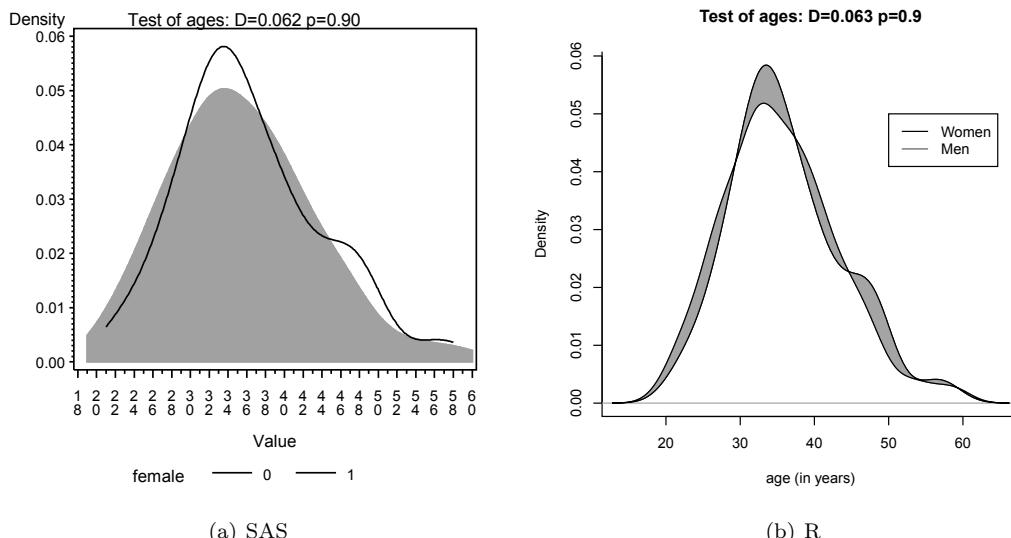


Figure 5.4: Density plot of age by gender

### 5.7.5 Survival analysis: logrank test

The logrank test (5.4.6) can be used to compare estimated survival curves between groups in the presence of censoring. Here we compare randomization groups with respect to `dayslink`, where a value of 0 for `linkstatus` indicates that the observation was censored, not observed, at the time recorded in `dayslink`.

```

ods select homtests;
proc lifetest data=ds;
  time dayslink*linkstatus(0);
  strata treat;
run;
ods select all;

Test of Equality over Strata

          Pr >
Test      Chi-Square    DF   Chi-Square
Log-Rank     84.7878      1    <.0001
Wilcoxon     87.0714      1    <.0001
-2Log(LR)   107.2920      1    <.0001

> library(survival)
> survobj = survdiff(Surv(dayslink, linkstatus) ~ treat,
  data=ds)
> print(survobj)

Call:
survdiff(formula = Surv(dayslink, linkstatus) ~ treat, data = ds)

n=431, 22 observations deleted due to missingness.

      N Observed Expected (O-E)^2/E (O-E)^2/V
treat=0 209       35     92.8     36.0     84.8
treat=1 222      128     70.2     47.6     84.8

Chisq= 84.8  on 1 degrees of freedom, p= 0

> names(survobj)

[1] "n"        "obs"      "exp"      "var"      "chisq"
[6] "na.action" "call"

```

# Chapter 6

# Linear regression and ANOVA

Regression and analysis of variance form the basis of many investigations. In this chapter we describe how to undertake many common tasks in linear regression (broadly defined), while Chapter 7 discusses many generalizations, including other types of outcome variables, longitudinal and clustered analysis, and survival methods.

Many SAS procedures and R commands can perform linear regression, as it constitutes a special case of which many models are generalizations. We present detailed descriptions for SAS `proc reg` and `proc glm` as well as for the R `lm()` command, as these offer the most flexibility and best output options tailored to linear regression in particular. While ANOVA can be viewed as a special case of linear regression, separate routines are available in SAS (`proc anova`) and R (`aov()`) to perform it. In addition, SAS `proc mixed` is needed for some calculations. We address these additional procedures only with respect to output that is difficult to obtain through the standard linear regression tools.

R supports a flexible modeling language implemented using formulas (see `help(formula)` and 6.1.1) for regression that is shared with the lattice graphics functions. Many of the routines available within R return or operate on `lm` class objects, which include objects such as coefficients, residuals, fitted values, weights, contrasts, model matrices, and the like (see `help(lm)`).

The CRAN statistics for the social sciences task view provides an excellent overview of methods described here and in Chapter 7.

## 6.1 Model fitting

### 6.1.1 Linear regression

SAS

```
proc glm data=ds;
  model y = x1 ... xk;
run;
or
proc reg data=ds;
  model y = x1 ... xk;
run;
```

*Example: 6.6.2*

*Note:* Both `proc glm` and `proc reg` support linear regression models, while `proc reg` provides more regression diagnostics. The `glm` procedure more easily allows categorical covariates.

**R**

```
mod1 = lm(y ~ x1 + ... + xk, data=ds)
summary(mod1)
summary.aov(mod1)
or
form = as.formula(y ~ x1 + ... + xk)
mod1 = lm(form, data=ds)
summary(mod1)
coef(mod1)
```

*Note:* The first argument of the `lm()` function is a formula object, with the outcome specified followed by the `~` operator then the predictors. It returns a linear model object. More information about the linear model `summary()` command can be found using `help(summary.lm)`. The `coef()` function extracts coefficients from a model (see also the `coefplot` package). The `biglm()` function in the `biglm` package can support model fitting to very large datasets. By default, stars are used to annotate the output of the `summary()` functions regarding significance levels: these can be turned off using the command `options(show.signif.stars=FALSE)`. The `biglm` package can be used to undertake estimation with larger datasets.

### 6.1.2 Linear regression with categorical covariates

*Example:* 6.6.2

See 6.1.4 (parameterization of categorical covariates).

**SAS**

```
proc glm data=ds;
  class x1;
  model y = x1 x2 ... xk;
run;
```

*Note:* The `class` statement specifies covariates that should be treated as categorical. The `glm` procedure uses reference cell coding; the reference category can be controlled using the `order` option to the `proc glm` statement, as in 7.10.11.

**R**

```
ds = transform(ds, x1f = as.factor(x1))
mod1 = lm(y ~ x1f + x2 + ... + xk, data=ds)
```

*Note:* The `as.factor()` command in R creates a categorical variable from a variable. By default, the lowest value (either numerically or lexicographically) is the reference value. The `levels` option for the `factor()` function can be used to select a particular reference value (see 2.2.19). Ordered factors can be constructed using the `ordered()` function.

### 6.1.3 Changing the reference category

**SAS**

```
proc sort data=ds;
  by classvar;
run;

proc glm data=ds order=data;
  class classvar;
  model y = classvar;
run;
```

or

```
proc genmod data=ds;
  class classvar (param=ref ref="level");
  model y = classvar;
run;
```

*Note:* The first code is necessary for procedures that use the more primitive `class` statement, as reviewed in 6.1.4. For these procedures, the default reference category is the last one. The `order` option can take other values, which may be useful. If the desired reference cell cannot be sorted to the end, it may be necessary to recode the category values to, e.g., A. Brown, B. Blue, and C. Green from Blue, Brown, and Green, before sorting. Sorting by descending `classvar` may also be useful. The second set of code will work in the `genmod`, `logistic`, and `surveylogistic` procedures.

**R**

```
ds = transform(ds, neworder = factor(classvar,
  levels=c("level", "otherlev1", "otherlev2")))
mod1 = lm(y ~ neworder, data=ds)
```

*Note:* The first level of a factor (by default that which appears first lexicographically) is the reference group. This can be modified through use of the `factor()` function.

#### 6.1.4 Parameterization of categorical covariates

*Example:* 6.6.5

SAS and R handle this issue in different ways. In R, `as.factor()` can be applied within any model-fitting function. Parameterization of the covariate can be controlled as below. For SAS, some procedures accept a `class` statement to declare that a covariate is to be treated as categorical. The following procedures will not accept a class statement: `arima`, `catmod`, `factor`, `lifetest`, `loess`, `mcmc`, `nlin`, `nlmixed`, `reg`, and `varclus`. For these procedures, indicator (or “dummy”) variables must be created in a data step, though this should be done with caution. The following procedures accept a class statement which applies reference cell or indicator variable coding (described as `contr.SAS()` in the R note below) to the listed variables: `proc anova`, `candisc`, `discrim`, `fmm`, `gam`, `glimmix`, `glm`, `mixed`, `quantreg`, `robustreg`, `stepdisc`, and `surveyreg`. The value used as the referent can often be controlled, usually as an `order` option to the controlling `proc`, as in 7.10.11. For these procedures, other parameterizations must be coded in a data step. The following procedures accept multiple parameterizations, using the syntax shown below for `proc logistic`: `proc genmod` (defaults to reference cell coding), `proc logistic` (defaults to effect coding), `proc phreg` (defaults to reference cell coding), and `proc surveylogistic` (defaults to effect coding).

**SAS**

```
proc logistic data=ds;
  class x1 (param=paramtype) x2 (param=paramtype);
  ...
run;
or
proc logistic data=ds;
  class x1 x2 / param=paramtype;
  ...
run;
```

*Note:* Available `paramtypes` include: 1) `orthpoly`, equivalent to `contr.poly()`; 2) `effect` (the default for `proc logistic` and `proc surveylogistic`), equivalent to `contr.sum()`; and 3) `ref`, equivalent to `contr.SAS()`. In addition, if the same parameterization is desired

for all of the categorical variables in the model, it can be added in a statement such as the second example. In this case, `param=glm` can be used to emulate the parameterization found in the other procedures which accept `class` statements and in `contr.SAS()` within R; this is the default for `proc genmod` and `proc phreg`.

## R

```
ds = transform(ds, x1f = as.factor(x1))
mod1 = lm(y ~ x1f, contrasts=list(x1f="contr.SAS"), data=ds)
```

*Note:* The `as.factor()` function creates a factor object, akin to how SAS treats class variables in `proc glm`. The `contrasts` option for the `lm()` function specifies how the levels of that factor object should be used within the function. The `levels` option to the `factor()` function allows specification of the ordering of levels (the default is lexicographic). An example can be found in 6.6.

The specification of the design matrix for analysis of variance and regression models can be controlled using the `contrasts` option. Examples of options (for a factor with four equally spaced levels) are given below.

<pre>&gt; contr.treatment(4)   2 3 4 1 0 0 0 2 1 0 0 3 0 1 0 4 0 0 1 &gt; contr.SAS(4)   1 2 3 1 1 0 0 2 0 1 0 3 0 0 1 4 0 0 0 &gt; contr.helmert(4)  [,1] [,2] [,3] 1    -1   -1   -1 2     1   -1   -1 3     0    2   -1 4     0    0    3</pre>	<pre>&gt; contr.poly(4)       .L    .Q    .C [1,] -0.671  0.5 -0.224 [2,] -0.224 -0.5  0.671 [3,]  0.224 -0.5 -0.671 [4,]  0.671  0.5  0.224 &gt; contr.sum(4)  [,1] [,2] [,3] 1     1    0    0 2     2    0    1 3     3    0    0 4     4   -1   -1</pre>
--	--

See `options("contrasts")` for defaults, and `contrasts()` or `lm()` to apply a contrast function to a factor variable. Support for reordering factors is available within the `factor()` function.

### 6.1.5 Linear regression with no intercept

#### SAS

```
proc glm data=ds;
  model y = x1 ... xk / noint;
run;
```

*Note:* The `noint` option works with many `model` statements.

## R

```
mod1 = lm(y ~ 0 + x1 + ... + xk, data=ds)
or
mod1 = lm(y ~ x1 + ... + xk - 1, data=ds)
```

### 6.1.6 Linear regression with interactions

SAS

```
proc glm data=ds;
  model y = x1 x2 x1*x2 x3 ... xk;
run;
or
proc glm data=ds;
  model y = x1|x2 x3 ... xk;
run;
```

*Example: 6.6.2*

*Note:* The | operator includes the product and all lower order terms, while the \* operator includes only the specified interaction. So, for example, `model y = x1|x2|x3` and `model y = x1 x2 x3 x1*x2 x1*x3 x2*x3 x1*x2*x3` are equivalent statements. The syntax above also works with any covariates designated as categorical using the `class` statement (6.1.2). The `model` statement for many procedures accepts this syntax.

R

```
mod1 = lm(y ~ x1 + x2 + x1:x2 + x3 + ... + xk, data=ds)
or
lm(y ~ x1*x2 + x3 + ... + xk, data=ds)
```

*Note:* The \* operator includes all lower order terms (in this case main effects), while the : operator includes only the specified interaction. So, for example, the commands `y ~ x1*x2*x3` and `y ~ x1 + x2 + x3 + x1:x2 + x1:x3 + x2:x3 + x1:x2:x3` are equivalent. The syntax also works with any covariates designated as categorical using the `as.factor()` command (see 6.1.2).

### 6.1.7 One-way analysis of variance

SAS

```
proc glm data=ds;
  class x;
  model y = x / solution;
run;
```

*Example: 6.6.5*

*Note:* The `solution` option to the `model` statement requests that the parameter estimates be displayed. Other procedures which fit ANOVA models include `proc anova` and `proc mixed`.

R

```
ds = transform(ds, xf=as.factor(x))
mod1 = aov(y ~ xf, data=ds)
summary(mod1)
anova(mod1)
```

*Note:* The `summary()` command can be used to provide details of the model fit. More information can be found using `help(summary.aov)`. Note that `summary.lm(mod1)` will display the regression parameters underlying the ANOVA model.

### 6.1.8 Analysis of variance with two or more factors

*Example: 6.6.5*

Interactions can be specified using the syntax introduced in 6.1.6 (see interaction plots, 8.5.2).

**SAS**

```
proc glm data=ds;
  class x1 x2;
  model y = x1 x2;
run;
```

*Note:* Other procedures which fit ANOVA models include `proc anova` and `proc mixed`.

**R**

```
aov(y ~ as.factor(x1) + as.factor(x2), data=ds)
```

## 6.2 Tests, contrasts, and linear functions of parameters

### 6.2.1 Joint null hypotheses: several parameters equal 0

As an example, consider testing the null hypothesis  $H_0 : \beta_1 = \beta_2 = 0$ .

**SAS**

```
proc reg data=ds;
  model ...;
  nametest: test x1=0, x2=0;
run;
```

*Note:* In the above, `nametest` is an arbitrary label which will appear in the output. Multiple `test` statements are permitted.

**R**

```
mod1 = lm(y ~ x1 + ... + xk, data=ds)
mod2 = lm(y ~ x3 + ... + xk, data=ds)
anova(mod2, mod1)
```

### 6.2.2 Joint null hypotheses: sum of parameters

As an example, consider testing the null hypothesis  $H_0 : \beta_1 + \beta_2 = 1$ .

**SAS**

```
proc reg data=ds;
  model ...;
  nametest: test x1 + x2 = 1;
run;
```

*Note:* The `test` statement is prefixed with an arbitrary `nametest` which will appear in the output. Multiple `test` statements are permitted.

**R**

```
mod1 = lm(y ~ x1 + ... + xk, data=ds)
covb = vcov(mod1)
coeff.mod1 = coef(mod1)
t = (coeff.mod1[2] + coeff.mod1[3] - 1)/
    sqrt(covb[2,2] + covb[3,3] + 2*covb[2,3])
pvalue = 2*(1-pt(abs(t), df=mod1$df))
```

*Note:* The `I()` function inhibits the interpretation of operators, to allow them to be used as arithmetic operators.

### 6.2.3 Tests of equality of parameters

*Example: 6.6.7*

As an example, consider testing the null hypothesis  $H_0 : \beta_1 = \beta_2$ .

#### SAS

```
proc reg data=ds;
  model ...;
  nametest: test x1=x2;
run;
```

*Note:* The **test** statement is prefixed with an arbitrary **nametest** which will appear in the output. Multiple **test** statements are permitted.

#### R

```
mod1 = lm(y ~ x1 + ... + xk, data=ds)
mod2 = lm(y ~ I(x1+x2) + ... + xk, data=ds)
anova(mod2, mod1)
or
library(gmodels)
estimable(mod1, c(0, 1, -1, 0, ..., 0))
or
mod1 = lm(y ~ x1 + ... + xk, data=ds)
covb = vcov(mod1)
coeff.mod1 = coef(mod1)
t = (coeff.mod1[2]-coeff.mod1[3])/sqrt(covb[2,2]+covb[3,3]-2*covb[2,3])
pvalue = 2*(1-pt(abs(t), mod1$df))
```

*Note:* The **I()** function inhibits the interpretation of operators, to allow them to be used as arithmetic operators. The **estimable()** function calculates a linear combination of the parameters. The more general R code below utilizes the same approach introduced in 6.2.1 for the specific test of  $\beta_1 = \beta_2$  (different coding would be needed for other comparisons).

### 6.2.4 Multiple comparisons

*Example: 6.6.6*

#### SAS

```
proc glm data=ds;
  class x1;
  model y = x1;
  lsmeans x1 / pdiff adjust=tukey;
run;
```

*Note:* The **pdiff** option requests *p*-values for the hypotheses involving the pairwise comparison of means. The **adjust** option adjusts these *p*-values for multiple comparisons. Other options available through **adjust** include **bon** (for Bonferroni) and **dunnett**, among others. SAS **proc mixed** also has an **adjust** option for its **lsmeans** statement. A graphical presentation of significant differences among levels can be obtained with the **lines** option to the **lsmeans** statement, as shown in 6.6.6.

#### R

```
mod1 = aov(y ~ x, data=ds)
TukeyHSD(mod1, "x")
```

*Note:* The **TukeyHSD()** function takes an **aov** object as an argument and calculates the pairwise comparisons of all of the combinations of the factor levels of the variable **x** (see the **multcomp** package).

### 6.2.5 Linear combinations of parameters

*Example: 6.6.7*

It is often useful to calculate predicted values for particular covariate values. Here, we calculate the predicted value  $E[Y|X_1 = 1, X_2 = 3] = \hat{\beta}_0 + \hat{\beta}_1 + 3\hat{\beta}_2$ .

#### SAS

```
proc glm data=ds;
  model y = x1 ... xk;
  estimate 'label' intercept 1 x1 1 x2 3;
run;
```

*Note:* The `estimate` statement is used to calculate a linear combination of parameters (and associated standard errors). The optional quoted text is a label which will be printed with the estimated function.

#### R

```
mod1 = lm(y ~ x1 + x2, data=ds)
newdf = data.frame(x1=c(1), x2=c(3))
predict(mod1, newdf, se.fit=TRUE, interval="confidence")
or
library(gmodels)
estimable(mod1, c(0, 1, 3))
or
library(mosaic)
myfun = makeFun(mod1)
myfun(x1=1, x2=3)
```

*Note:* The `predict()` command in R can generate estimates at any combination of parameter values, as specified as a dataframe that is passed as an argument. More information on this function can be found using `help(predict.lm)`.

## 6.3 Model diagnostics

### 6.3.1 Predicted values

*Example: 6.6.2*

#### SAS

```
proc reg data=ds;
  model ...;
  output out=newds predicted=predicted_varname;
run;
or
proc glm data=ds;
  model ...;
  output out=newds predicted=predicted_varname;
run;
```

*Note:* The `output` statement creates a new dataset and specifies variables to be included, of which the predicted values are an example. Others can be found using the on-line help: Contents; SAS Products; SAS Procedures; REG; OUTPUT.

#### R

```
mod1 = lm(y ~ x, data=ds)
predicted.varname = predict(mod1)
```

*Note:* The command `predict()` operates on any `lm` object and by default generates a vector of predicted values. Similar commands retrieve other regression output.

### 6.3.2 Residuals

SAS

```
proc glm data=ds;
  model ...;
  output out=newds residual=residual_varname;
run;
or
proc reg data=ds;
  model ...;
  output out=newds residual=residual_varname;
run;
```

*Example: 6.6.2*

*Note:* The **output** statement creates a new dataset and specifies variables to be included, of which the residuals are an example. Others can be found using the on-line help: Contents; SAS Products; SAS Procedures; REG; OUTPUT.

R

```
mod1 = lm(y ~ x, data=ds)
residual.varname = residuals(mod1)
```

*Note:* The command **residuals()** operates on any **lm** object and generates a vector of residuals. Other functions for **aov**, **glm**, or **lme** objects exist (see, for example, **help(residuals.glm)**).

### 6.3.3 Standardized and Studentized residuals

*Example: 6.6.2*

Standardized residuals are calculated by dividing the ordinary residual (observed minus expected,  $y_i - \hat{y}_i$ ) by an estimate of its standard deviation. Studentized residuals are calculated in a similar manner, where the predicted value and the variance of the residual are estimated from the model fit while excluding that observation. In SAS **proc glm** the standardized residual is requested by the **student** option, while the **rstudent** option generates the studentized residual.

SAS

```
proc glm data=ds;
  model ...;
  output out=newds student=standardized_resid_varname;
run;
or
proc reg data=ds;
  model ...;
  output out=newds rstudent=studentized_resid_varname;
run;
```

*Note:* The **output** statement creates a new dataset and specifies variables to be included, of which the Studentized residuals are an example. Both **proc reg** and **proc glm** include both types of residuals. Others can be found using the on-line help: Contents; SAS Products; SAS Procedures; REG; OUTPUT.

R

```
mod1 = lm(y ~ x, data=ds)
standardized.resid.varname = stdres(mod1)
studentized.resid.varname = studres(mod1)
```

*Note:* The **stdres()** and **studres()** functions operate on any **lm** object and generate a vector of studentized residuals (the former command includes the observation in the

calculation, while the latter does not). Similar commands retrieve other regression output (see `help(influence.measures)`).

### 6.3.4 Leverage

*Example:* 6.6.2

Leverage is defined as the diagonal element of the  $(X(X^T X)^{-1} X^T)$  or “hat” matrix.

#### SAS

```
proc glm data=ds;
  model ...;
  output out=newds h=leverage_varname;
run;
or
proc reg data=ds;
  model ...;
  output out=newds h=leverage_varname;
run;
```

*Note:* The `output` statement creates a new dataset and specifies variables to be included, of which the leverage values are one example. Others can be found using the on-line help: Contents; SAS Products; SAS Procedures; REG; OUTPUT.

#### R

```
mod1 = lm(y ~ x, data=ds)
leverage.varname = hatvalues(mod1)
```

*Note:* The command `hatvalues()` operates on any `lm` object and generates a vector of leverage values. Similar commands can be utilized to retrieve other regression output (see `help(influence.measures)`).

### 6.3.5 Cook’s D

*Example:* 6.6.2

Cook’s distance (D) is a function of the leverage (see 6.3.4) and the residual. It is used as a measure of the influence of a data point in a regression model.

#### SAS

```
proc glm data=ds;
  model ...;
  output out=newds cookd=cookd_varname;
run;
or
proc reg data=ds;
  model ...;
  output out=newds cookd=cookd_varname;
run;
```

*Note:* The `output` statement creates a new dataset and specifies variables to be included, of which the Cook’s distance values are an example. Others can be found using the on-line help: Contents; SAS Products; SAS Procedures; REG; OUTPUT.

#### R

```
mod1 = lm(y ~ x, data=ds)
cookd.varname = cooks.distance(mod1)
```

*Note:* The command `cooks.distance()` operates on any `lm` object and generates a vector of Cook’s distance values. Similar commands retrieve other regression output.

### 6.3.6 DFFITS

*Example: 6.6.2*

DFFITS are a standardized function of the difference between the predicted value for the observation when it is included in the dataset and when (only) it is excluded from the dataset. They are used as an indicator of the observation's influence.

#### SAS

```
proc reg data=ds;
  model ...;
  output out=newds dffits=dffits_varname;
run;
or
proc glm data=ds;
  model ...;
  output out=newds dffits=dffits_varname;
run;
```

*Note:* The `output` statement creates a new dataset and specifies variables to be included, of which the DFFITS values are an example. Others can be found using the on-line help: Contents; SAS Products; SAS Procedures; REG; OUTPUT.

#### R

```
mod1 = lm(y ~ x, data=ds)
dffits.varname = dffits(mod1)
```

*Note:* The command `dffits()` operates on any `lm` object and generates a vector of DFFITS values. Similar commands retrieve other regression output.

### 6.3.7 Diagnostic plots

*Example: 6.6.3*

#### SAS

```
proc reg data=ds;
  model ...;
  output out=newds predicted=pred_varname residual=resid_varname
        h=leverage_varname cookd=cookd_varname;
run;

proc gplot data=ds;
  plot resid_varname * pred_varname;
  plot resid_varname * leverage_varname;
run;
quit;
```

*Note:* To mimic R more closely, use a data step to generate the square root of residuals. QQ plots of residuals can be generated via `proc univariate`. It is not straightforward to plot lines of constant Cook's D on the residuals vs. leverage plot. The `reg` procedure will produce many diagnostic plots, as will `proc glm` with the `plots=diagnostics` option.

#### R

```
mod1 = lm(y ~ x, data=ds)
par(mfrow=c(2, 2)) # display 2 x 2 matrix of graphs
plot(mod1)
```

*Note:* The `plot.lm()` function (which is invoked when `plot()` is given a linear regression model as an argument) can generate six plots: 1) a plot of residuals against fitted values, 2) a Scale-Location plot of  $\sqrt{(Y_i - \hat{Y}_i)}$  against fitted values, 3) a normal Q-Q plot of the

residuals, 4) a plot of Cook's distances (6.3.5) versus row labels, 5) a plot of residuals against leverages (6.3.4), and 6) a plot of Cook's distances against leverage/(1-leverage). The default is to plot the first three and the fifth. The `which` option can be used to specify a different set (see `help(plot.lm)`).

### 6.3.8 Heteroscedasticity tests

#### SAS

```
proc model data=ds;
  parms int slope;
  y = int + slope * x;
  fit y / pagan=(1 x) white;
run; quit;
```

*Note:* In SAS, White's test [192] and the Breusch–Pagan test [19] can be found in the `model` procedure in the SAS/ETS product. Note the atypical syntax of `proc model`.

#### R

```
library(lmtest)
bptest(y ~ x1 + ... + xk, data=ds)
```

*Note:* The `bptest()` function in the `lmtest` package performs the Breusch–Pagan test for heteroscedasticity [19].

## 6.4 Model parameters and results

### 6.4.1 Parameter estimates

*Example: 6.6.2*

#### SAS

```
ods output parameterestimates=newds;
proc glm data=ds;
  model ... / solution;
run;
or
proc reg data=ds outest=newds;
  model ...;
run;
```

*Note:* The `ods output` statement (A.7.1) can be used to save any piece of SAS output as a SAS dataset. The `outest` option is specific to `proc reg`, though many other procedures accept similar syntax.

#### R

```
mod1 = lm(y ~ x, data=ds)
coeff.mod1 = coef(mod1)
```

*Note:* The first element of the vector `coeff.mod1` is the intercept (assuming that a model with an intercept was fit).

### 6.4.2 Standardized regression coefficients

Standardized coefficients from a linear regression model are the parameter estimates obtained when the predictors and outcomes have been standardized to have a variance of 1 prior to model fitting.

**SAS**

```
proc reg data=ds;
  model ... / stb;
run;
```

**R**

```
library(QuantPsyc)
mod1 = lm(y ~ x)
lm.beta(mod1)
```

### 6.4.3 Standard errors of parameter estimates

See 6.4.9 (covariance matrix).

**SAS**

```
proc reg data=ds outest=newds;
  model .../ outseb ...;
run;
```

or

```
ods output parameterestimates=newds;
proc glm data=ds;
  model .../ solution;
run;
```

*Note:* The `ods output` statement (A.7.1) can be used to save any piece of SAS output as a SAS dataset.

**R**

```
mod1 = lm(y ~ x, data=ds)
sqrt(diag(vcov(mod1)))
```

or

```
coef(summary(mod1))[,2]
```

*Note:* The standard errors are the second column of the results from `coef()`.

### 6.4.4 Confidence interval for parameter estimates

**SAS**

```
proc reg data=ds;
  model ... / clb;
run;
```

*Example:* 6.6.2

**R**

```
mod1 = lm(y ~ x, data=ds)
confint(mod1)
```

### 6.4.5 Confidence limits for the mean

These are the lower (and upper) confidence limits for the mean of observations with the given covariate values, as opposed to the prediction limits for individual observations with those values (see prediction limits, 6.4.6).

**SAS**

```
proc glm data=ds;
  model ...;
  output out=newds lclm=lcl_mean_varname;
run;
```

or

```
proc reg data=ds;
  model ...;
  output out=newds lclm=lcl_mean_varname;
run;
```

*Note:* The **output** statement creates a new dataset and specifies output variables to be included, of which the lower confidence limit values are one example. The upper confidence limits can be generated using the **uclm** option to the **output** statement. Other possibilities can be found using the on-line help: Contents; SAS Products; SAS Procedures; REG; OUTPUT.

**R**

```
mod1 = lm(y ~ x, data=ds)
pred = predict(mod1, interval="confidence")
lcl.varname = pred[,2]
```

*Note:* The lower confidence limits are the second column of the results from **predict()**. To generate the upper confidence limits, the user would replace **lclm** with **uclm** for SAS and access the third column of the **predict()** object in R. The command **predict()** operates on any **lm()** object, and with these options generates confidence limit values. By default, the function uses the estimation dataset, but a separate dataset of values to be used to predict can be specified. The **panel=panel.lmbands** option from the **mosaic** package can be added to an **xyplot()** call to augment the scatterplot with confidence interval and prediction bands.

#### 6.4.6 Prediction limits

These are the lower (and upper) prediction limits for “new” observations with the covariate values of subjects observed in the dataset, as opposed to confidence limits for the population mean (see confidence limits, 6.4.5).

**SAS**

```
proc glm data=ds;
  model ...;
  output out=newds lcl=lcl_varname;
run;
or
proc reg data=ds;
  model ...;
  output out=newds lcl=lcl_varname;
run;
```

*Note:* The **output** statement creates a new dataset and specifies variables to be included, of which the lower prediction limit values are an example. The upper limits can be requested with the **ucl** option to the **output** statement. Other possibilities can be found using the on-line help: Contents; SAS Products; SAS Procedures; REG; OUTPUT.

**R**

```
mod1 = lm(y ~ ..., data=ds)
pred.w.lowlim = predict(mod1, interval="prediction")[,2]
```

*Note:* This code saves the second column of the results from the **predict()** function into a vector. To generate the upper confidence limits, the user would access the third column of the **predict()** object in R. The command **predict()** operates on any **lm()** object, and with these options generates prediction limit values. By default, the function uses the estimation dataset, but a separate dataset of values to be used to predict can be specified.

### 6.4.7 R-squared

#### SAS

```
proc glm data=ds;
  model ...;
run;
```

*Note:* The coefficient of determination can be found as default output from `proc reg` or `proc glm`.

#### R

```
mod1 = lm(y ~ ..., data=ds)
summary(mod1)$r.squared
or
library(mosaic)
rsquared(mod1)
```

### 6.4.8 Design and information matrix

See 3.3 (matrices).

#### SAS

```
proc reg data=ds;
  model .../ xpx ...;
run;
or
proc glm data=ds;
  model .../ xpx ...;
run;
```

*Note:* A dataset containing the information ( $(X'X)$ ) matrix can be created using `ODS` by specifying either `proc` statement or by adding the option `outsscp=newds` to the `proc reg` statement.

#### R

```
mod1 = lm(y ~ x1 + ... + xk, data=ds)
XpX = t(model.matrix(mod1)) %*% model.matrix(mod1)
or
X = cbind(rep(1, length(x1)), x1, x2, ..., xk)
XpX = t(X) %*% X
rm(X)
```

*Note:* The `model.matrix()` function creates the design matrix from a linear model object. Alternatively, this quantity can be built up using the `cbind()` function to glue together the design matrix `X`. Finally, matrix multiplication (3.3.6) and the transpose function are used to create the information ( $(X'X)$ ) matrix.

### 6.4.9 Covariance matrix of parameter estimates

*Example:* 6.6.2

See 3.3 (matrices) and 6.4.3 (standard errors).

#### SAS

```
proc reg data=ds outest=newds covout;
run;
```

```
or
ods output covb=newds;
proc reg data=ds;
  model ... / covb ...;
```

**R**

```
mod1 = lm(y ~ x, data=ds)
vcov(mod1)
or
sumvals = summary(mod1)
covb = sumvals$cov.unscaled*sumvals$sigma^2
```

*Note:* Running `help(summary.lm)` provides details on return values.

#### 6.4.10 Correlation matrix of parameter estimates

See 3.3 (matrices) and 6.4.3 (standard errors).

**SAS**

```
ods output covb=lmcov corrb=lmcov ;
proc reg data=ds;
  model ... / covb corrb ...;
or
proc reg data=ds outest=newds covout;
  model ...;
  outest=outds;
run;
```

*Note:* the former can be used to generate either the covariance or correlation matrix, or both. The demonstrated ODS command will save the matrix or matrices as datasets. The latter uses the older method of generating SAS output as a dataset, but does not allow the generation of the correlation matrix.

**R**

```
mod1 = lm(y ~ x, data=ds)
mod1.cov = vcov(mod1)
mod1.cor = cov2cor(mod1.cov)
```

*Note:* The `cov2cor()` function is a convenient way to convert a covariance matrix into a correlation matrix.

## 6.5 Further resources

Accessible guides to linear regression in R and SAS can be found in [38] and [108], respectively. Cook [29] reviews regression diagnostics. Frank Harrell's `rms` (regression modeling strategies) package [63] features extensive support for regression modeling. The CRAN statistics for the social sciences task view provides an excellent overview of methods described here and in Chapter 7.

## 6.6 Examples

To help illustrate the tools presented in this chapter, we apply many of the entries to the HELP data. SAS and R code can be downloaded from <http://www.amherst.edu/~nhorton/sasr2/examples>.

We begin by reading in the dataset and keeping only the female subjects. In R, we prepare for later analyses by creating a version of `substance` as a factor variable (see 6.1.4).

```
proc import datafile='c:/book/help.dta'
    out=help_a dbms=dta;
run;

data help;
set help_a;
    if female;
run;

> options(digits=3)
> # read in Stata format
> library(foreign)
> ds = read.dta("help.dta", convert.underscore=FALSE)
> ds = transform(ds, sub = factor(substance,
    levels=c("heroin", "alcohol", "cocaine")))
> newds = subset(ds, female==1)
```

### 6.6.1 Scatterplot with smooth fit

As a first step to help guide fitting a linear regression, we create a scatterplot (8.3.1) displaying the relationship between age and the number of alcoholic drinks consumed in the period before entering detox (variable name: `i1`), as well as primary substance of abuse (alcohol, cocaine, or heroin).

Figure 6.1 displays a scatterplot of observed values for `i1` (along with separate smooth fits by primary substance). To improve legibility, the plotting region is restricted to those with number of drinks between 0 and 40 (see plotting limits, 9.2.9).

```
axis1 order = (0 to 40 by 10) minor=none;
axis2 minor=none;
legend1 label=none value=(h=1.5) shape=symbol(10,1.2)
    down=3 position=(top right inside) frame mode=protect;
symbol1 v=circle i=sm70s c=black l=1 h=1.1 w=5;
symbol2 v=diamond i=sm70s c=black l=33 h=1.1 w=5;
symbol3 v=square i=sm70s c=black l=8 h=1.1 w=5;
proc gplot data=help;
    plot i1*age = substance / vaxis=axis1 haxis=axis2 legend=legend1;
run; quit;
```

```

> with(newds, plot(age, i1, ylim=c(0,40), type="n", cex.lab=1.4,
+ cex.axis=1.4))
> with(newds, points(age[substance=="alcohol"], i1[substance=="alcohol"],
+ pch="a"))
> with(newds, lines(lowess(age[substance=="alcohol"]),
+ i1[substance=="alcohol"]), lty=1, lwd=2))
> with(newds, points(age[substance=="cocaine"], i1[substance=="cocaine"],
+ pch="c"))
> with(newds, lines(lowess(age[substance=="cocaine"]),
+ i1[substance=="cocaine"]), lty=2, lwd=2))
> with(newds, points(age[substance=="heroin"], i1[substance=="heroin"],
+ pch="h"))
> with(newds, lines(lowess(age[substance=="heroin"]),
+ i1[substance=="heroin"]), lty=3, lwd=2))
> legend(44, 38, legend=c("alcohol", "cocaine", "heroin"), lty=1:3,
+ cex=1.4, lwd=2, pch=c("a", "c", "h"))

```

The `pch` option to the `legend()` command can be used to insert plot symbols in R legends (Figure 6.1 displays the different line styles).

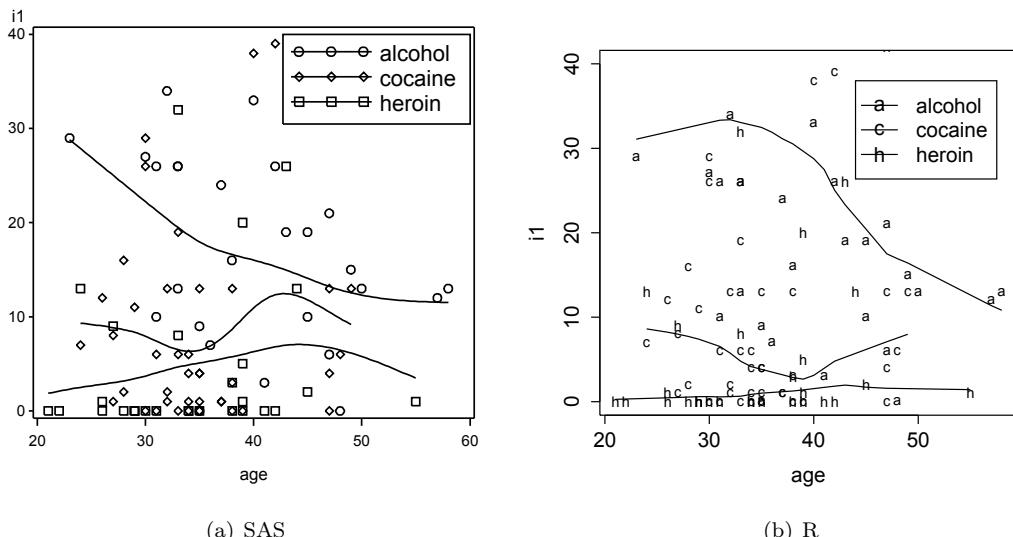


Figure 6.1: Scatterplot of observed values for age and I1 (plus smoothers by substance)

Not surprisingly, Figure 6.1 suggests that there is a dramatic effect of primary substance, with alcohol users drinking more than others. There is some indication of an interaction with age. It is important to note that SAS uses only the points displayed (i.e., within the specified axes) when smoothing, while R uses all points, regardless of whether they appear in the plot.

## 6.6.2 Linear regression with interaction

Next we fit a linear regression model (6.1.1) for the number of drinks as a function of age, substance, and their interaction (6.1.6). To assess the need for the interaction, we use the F test from the Type III sums of squares in SAS. In R, we additionally fit the model with

no interaction and use the `anova()` function to compare the models (the `drop1()` function could also be used). To save space, some results of `proc glm` have been suppressed using the `ods select` statement (see A.7).

```
options ls=74; /* reduces width of output to make it fit in gray area */
ods select overallanova modelanova parameterestimates;
proc glm data=help;
class substance;
model i1 = age substance age * substance / solution;
output out=helpout cookd=cookd_ch4 dffits=dffits_ch4
student=sresids_ch4 residual=resid_ch4
predicted=pred_ch4 h=lev_ch4;
run; quit;
ods select all;
```

The GLM Procedure

Dependent Variable: i1 i1

Source	DF	Sum of Squares	Mean Square	F Value	Pr > F
Model	5	12275.17570	2455.03514	9.99	<.0001
Error	101	24815.36635	245.69670		
Corrected Total	106	37090.54206			

The GLM Procedure

Dependent Variable: i1 i1

Source	DF	Type I SS	Mean Square	F Value	Pr > F
age	1	384.75504	384.75504	1.57	0.2137
substance	2	10509.56444	5254.78222	21.39	<.0001
age*substance	2	1380.85622	690.42811	2.81	0.0649

Source	DF	Type III SS	Mean Square	F Value	Pr > F
age	1	27.157727	27.157727	0.11	0.7402
substance	2	3318.992822	1659.496411	6.75	0.0018
age*substance	2	1380.856222	690.428111	2.81	0.0649

## The GLM Procedure

Dependent Variable: i1 i1

Parameter	Estimate	Standard			
		Error	t Value	Pr >  t	
Intercept	-7.77045212 B	12.87885672	-0.60	0.5476	
age	0.39337843 B	0.36221749	1.09	0.2801	
substance alcohol	64.88044165 B	18.48733701	3.51	0.0007	
substance cocaine	13.02733169 B	19.13852222	0.68	0.4976	
substance heroin	0.00000000 B	.	.	.	
age*substance alcohol	-1.11320795 B	0.49135408	-2.27	0.0256	
age*substance cocaine	-0.27758561 B	0.53967749	-0.51	0.6081	
age*substance heroin	0.00000000 B	.	.	.	

```
> options(show.signif.stars=FALSE)
> lm1 = lm(i1 ~ sub * age, data=newds)
> lm2 = lm(i1 ~ sub + age, data=newds)
> anova(lm2, lm1)
```

## Analysis of Variance Table

```
Model 1: i1 ~ sub + age
Model 2: i1 ~ sub * age
      Res.Df   RSS Df Sum of Sq    F Pr(>F)
1       103 26196
2       101 24815  2      1381 2.81  0.065
```

```
> summary.aov(lm1)

      Df Sum Sq Mean Sq F value Pr(>F)
sub        2 10810   5405  22.00 1.2e-08
age        1     84     84   0.34  0.559
sub:age    2  1381    690   2.81  0.065
Residuals 101 24815    246
```

There is some indication of a borderline significant interaction between age and substance group ( $p=0.065$ ).

In SAS, the `ods output` statement can be used to save any printed result as a SAS dataset. In the following code, all printed output from `proc glm` is suppressed, but the parameter estimates are saved as a SAS dataset, then printed using `proc print`. In addition, various diagnostics are saved via the the `output` statement.

```
ods select none;
ods output ParameterEstimates=helpmodelanova;
proc glm data=help;
class substance;
model i1 = age/substance / solution;
output out=helpout cookd=cookd_ch4 dfits=dffits_ch4
      student=sresids_ch4 residual=resid_ch4
      predicted=pred_ch4 h=lev_ch4;
run; quit;
ods select all;
```

```
proc print data=helpmodelanova;
  var parameter estimate stderr tvalue probt;
  format _numeric_ 6.3;
run;
```

Obs	Parameter	Estimate	StdErr	tValue	Probt
1	Intercept	-7.770	12.879	-0.603	0.548
2	age	0.393	0.362	1.086	0.280
3	substance alcohol	64.880	18.487	3.509	0.001
4	substance cocaine	13.027	19.139	0.681	0.498
5	substance heroin	0.000	.	.	.
6	age*substance alcohol	-1.113	0.491	-2.266	0.026
7	age*substance cocaine	-0.278	0.540	-0.514	0.608
8	age*substance heroin	0.000	.	.	.

In R, the `summary()` function provides similar information.

```
> summary(lm1)
```

Call:

```
lm(formula = i1 ~ sub * age, data = newds)
```

Residuals:

Min	1Q	Median	3Q	Max
-31.92	-8.25	-4.18	3.58	49.88

Coefficients:

	Estimate	Std. Error	t value	Pr(> t )
(Intercept)	-7.770	12.879	-0.60	0.54763
subalcohol	64.880	18.487	3.51	0.00067
subcocaine	13.027	19.139	0.68	0.49763
age	0.393	0.362	1.09	0.28005
subalcohol:age	-1.113	0.491	-2.27	0.02561
subcocaine:age	-0.278	0.540	-0.51	0.60813

Residual standard error: 15.7 on 101 degrees of freedom

Multiple R-squared: 0.331, Adjusted R-squared: 0.298

F-statistic: 9.99 on 5 and 101 DF, p-value: 8.67e-08

```
> confint(lm1)
```

	2.5 %	97.5 %
(Intercept)	-33.319	17.778
subalcohol	28.207	101.554
subcocaine	-24.938	50.993
age	-0.325	1.112
subalcohol:age	-2.088	-0.138
subcocaine:age	-1.348	0.793

It may also be useful to produce the table in L<sup>A</sup>T<sub>E</sub>X format. In SAS, we can do this using the `latex` destination for the `ODS` system. When compiled, the resulting table is displayed in Figure 6.2.

```
ods latex file="c:\book\table.tex" style=styles.printer;
proc print data = helpmodelanova; run;
ods latex close;
```

Obs	Dependent	Parameter	Estimate	Biased	StdErr	tValue	Probt
1	II	Intercept	5.31216647	1	6.87095132	0.77	0.4399
2	II	AGE	0.10665333	1	0.19987433	0.53	0.5939
3	II	SUBSTANCE alcohol	2.60085179	1	9.66168958	0.27	0.7879
4	II	SUBSTANCE cocaine	10.45473101	1	10.21750929	1.02	0.3068
5	II	SUBSTANCE heroin	0.00000000	1	.	.	.
6	II	AGE*SUBSTANCE alcohol	0.45042340	1	0.26525085	1.70	0.0902
7	II	AGE*SUBSTANCE cocaine	-0.21204498	1	0.29373772	-0.72	0.4707
8	II	AGE*SUBSTANCE heroin	0.00000000	1	.	.	.

Figure 6.2: SAS table produced with `latex` destination in ODS

In R, we can use the `xtable` package to display the regression results in L<sup>A</sup>T<sub>E</sub>X, as shown in Table 6.1.

```
> library(xtable)
> lmtab = xtable(lm1, digits=c(0,3,3,2,4), label="better",
+                 caption="Formatted results using the {\tt xtable} package")
> print(lmtab) # output the LaTeX
```

Table 6.1: Formatted results using the `xtable` package

	Estimate	Std. Error	t value	Pr(> t )
(Intercept)	-7.770	12.879	-0.60	0.5476
subalcohol	64.880	18.487	3.51	0.0007
subcocaine	13.027	19.139	0.68	0.4976
age	0.393	0.362	1.09	0.2801
subalcohol:age	-1.113	0.491	-2.27	0.0256
subcocaine:age	-0.278	0.540	-0.51	0.6081

There are many quantities of interest stored in the linear model object `lm1`, and these can be viewed or extracted for further use.

```
> names(summary(lm1))
[1] "call"          "terms"         "residuals"      "coefficients"
[5] "aliased"       "sigma"         "df"             "r.squared"
[9] "adj.r.squared" "fstatistic"    "cov.unscaled"

> summary(lm1)$sigma
[1] 15.7

> names(lm1)
[1] "coefficients" "residuals"      "effects"        "rank"
[5] "fitted.values" "assign"        "qr"            "df.residual"
[9] "contrasts"     "xlevels"       "call"          "terms"
[13] "model"
```

```
> coef(lm1)
(Intercept)      subalcohol      subcocaine      age subalcohol:age
-7.770          64.880          13.027          0.393          -1.113
subcocaine:age
-0.278

> vcov(lm1)

              (Intercept) subalcohol subcocaine   age subalcohol:age
(Intercept)      165.86    -165.86   -165.86 -4.548       4.548
subalcohol      -165.86     341.78    165.86  4.548      -8.866
subcocaine      -165.86    165.86    366.28  4.548      -4.548
age             -4.55        4.55     4.55  0.131      -0.131
subalcohol:age    4.55      -8.87    -4.55 -0.131       0.241
subcocaine:age    4.55      -4.55   -10.13 -0.131       0.131
                     subcocaine:age
(Intercept)          4.548
subalcohol          -4.548
subcocaine         -10.127
age                -0.131
subalcohol:age      0.131
subcocaine:age      0.291
```

### 6.6.3 Regression diagnostics

Assessing the model is an important part of any analysis. We begin by examining the residuals (6.3.2). First, we calculate the quantiles of their distribution (5.1.4), then display the smallest residual.

```
options ls=74;
proc means data=helpout min q1 median q3 max maxdec=2;
  var resid_ch4;
run;
```

The MEANS Procedure

Analysis Variable : resid\_ch4

	Lower		Upper	
Minimum	Quartile	Median	Quartile	Maximum
-31.92	-8.31	-4.18	3.69	49.88

```
> newds = transform(newds, pred = fitted(lm1))
> newds = transform(newds, resid = residuals(lm1))
> with(newds, quantile(resid))
```

0%	25%	50%	75%	100%
-31.92	-8.25	-4.18	3.58	49.88

We could examine the output, then condition to find the value of the residual that is less than  $-31$ . Instead the dataset can be sorted so the smallest observation is first and then print one observation.

```

proc sort data=helpout;
  by resid_ch4;
run;

proc print data=helpout (obs=1);
  var id age i1 substance pred_ch4 resid_ch4;
run;

      resid_
Obs   id    age    i1    substance   pred_ch4   ch4
      ch4
1    325    35     0    alcohol     31.9160   -31.9160

```

One way to print the largest value is to sort the dataset in the reverse order (2.3.10), then print just the first observation.

```

proc sort data=helpout;
  by descending resid_ch4;
run;

proc print data=helpout (obs=1);
  var id age i1 substance pred_ch4 resid_ch4;
run;

      resid_
Obs   id    age    i1    substance   pred_ch4   ch4
      ch4
1    9     50    71    alcohol     21.1185   49.8815
> tmpds = with(newds,
  data.frame(id, age, i1, sub, pred, resid, rstandard(lm1)))
> subset(tmpds, resid==max(resid))

  id age i1      sub pred resid rstandard.lm1.
9 9 50 71 alcohol 21.1 49.9          3.32

```

Graphical tools are one of the best ways to examine residuals. Figure 6.3 displays the default diagnostic plots (6.3) from the model (for R) and the Q-Q plot generated from the saved diagnostics (for SAS).

Sometimes in SAS it is necessary to clear out old graphics settings. This is easiest to do with the `goptions reset=all` statement (9.2.8).

```

goptions reset=all;

ods select univar;
proc univariate data=helpout;
  qqplot resid_ch4 / normal(mu=est sigma=est color=black);
run;

ods select all;

> oldpar = par(mfrow=c(2, 2), mar=c(4, 4, 2, 2)+.1)
> plot(lm1)
> par(oldpar)

```

In SAS, we get assorted diagnostic plots by default, but here we demonstrate a manual approach using the previously saved diagnostics. Figure 6.4 displays the empirical density of the standardized residuals, along with an overlaid normal density. The assumption that the residuals are approximately Gaussian does not appear to be tenable.

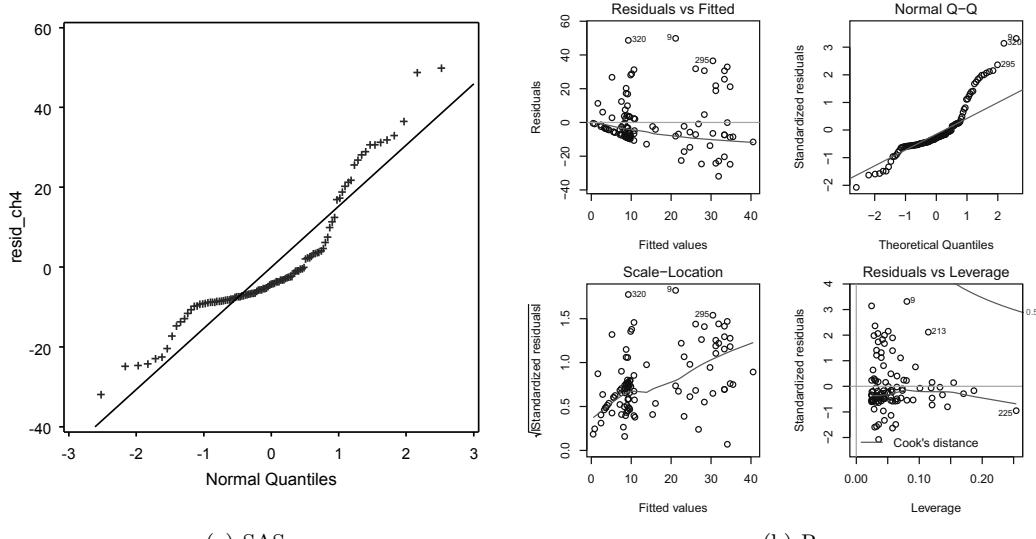


Figure 6.3: Q-Q plot from SAS, default diagnostics from R

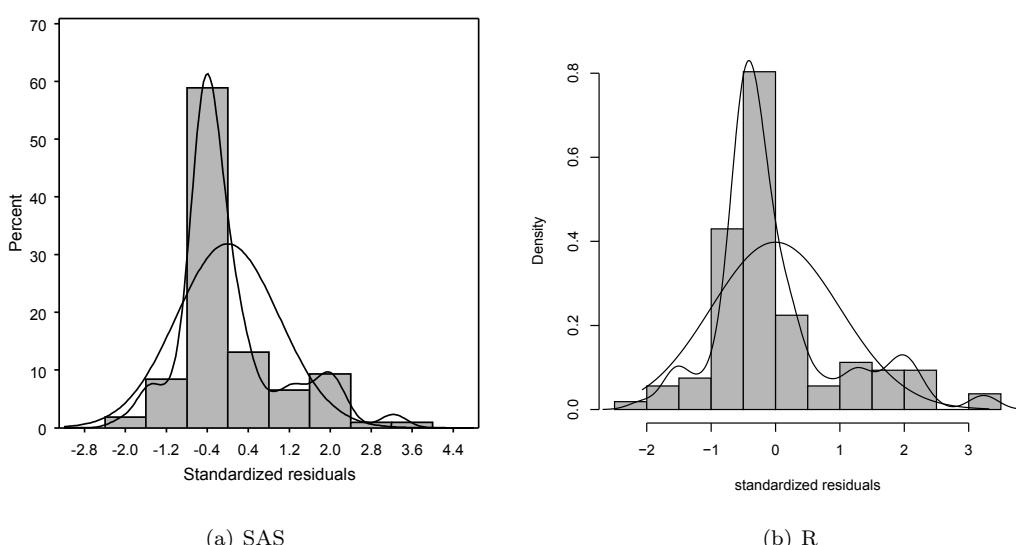


Figure 6.4: Empirical density of residuals, with superimposed normal density

```

axis1 label="Standardized residuals";
ods select "Histogram 1";
proc univariate data=helpout;
var sresids_ch4;
histogram sresids_ch4 / normal(mu=est sigma=est color=black)
kernel(color=black) haxis=axis1;
run;
ods select all;

```

```
> library(MASS)
> std.res = rstandard(lm1)
> hist(std.res, breaks=seq(-2.5, 3.5, by=.5), main="",
+       xlab="standardized residuals", col="gray80", freq=FALSE)
> lines(density(std.res), lwd=2)
> xvals = seq(from=min(std.res), to=max(std.res), length=100)
> lines(xvals, dnorm(xvals, mean(std.res), sd(std.res)), lty=2)
```

The residual plots indicate some potentially important departures from model assumptions, and further exploration should be undertaken.

#### 6.6.4 Fitting the regression model separately for each value of another variable

One common task is to perform identical analyses in several groups. Here, as an example, we consider separate linear regressions for each substance abuse group. In SAS, we show only the parameter estimates, using ODS.

```
ods select none;
proc sort data=help;
    by substance;
run;

ods output parameterestimates=helpsubstparams;
proc glm data=help;
    by substance;
    model i1 = age / solution;
run;
ods select all;
options ls=74;
proc print data=helpsubstparams;
run;
```

Obs	substance	Dependent Parameter	Estimate	StdErr	tValue	Probt
1	alcohol	i1 Intercept	57.10998953	18.00474934	3.17	0.0032
2	alcohol	i1 age	-0.71982952	0.45069028	-1.60	0.1195
3	cocaine	i1 Intercept	5.25687957	11.52989056	0.46	0.6510
4	cocaine	i1 age	0.11579282	0.32582541	0.36	0.7242
5	heroin	i1 Intercept	-7.77045212	8.59729637	-0.90	0.3738
6	heroin	i1 age	0.39337843	0.24179872	1.63	0.1150

For R, a matrix of the correct size is created, then a for loop is run for each unique value of the grouping variable.

```

> uniquevals = unique(newds$substance)
> numunique = length(uniquevals)
> formula = as.formula(i1 ~ age)
> p = length(coef(lm(formula, data=newds)))
> res = matrix(rep(0, numunique*p), p, numunique)
> for (i in 1:length(uniquevals)) {
+   res[,i] = coef(lm(formula,
+     data=subset(newds, substance==uniquevals[i])))
+ }
> rownames(res) = c("intercept", "slope")
> colnames(res) = uniquevals
> res

      heroin cocaine alcohol
intercept -7.770   5.257   57.11
slope       0.393   0.116   -0.72

```

### 6.6.5 Two-way ANOVA

Is there a statistically significant association between gender and substance abuse group with depressive symptoms? In SAS, we can make an interaction plot (8.5.2) by hand, as below, or proc glm will make a similar one automatically.

```

libname k 'c:/book';

proc sort data=k.help;
  by substance female;
run;

ods select none;
proc means data=k.help;
  by substance female;
  var cesd;
  output out=helpmean mean=;
run;
ods select all;
axis1 minor=none;
symbol1 i=j v=none l=1 c=black w=5;
symbol2 i=j v=none l=2 c=black w=5;
proc gplot data=helpmean;
  plot cesd*substance = female / haxis=axis1 vaxis=axis1;
run; quit;

```

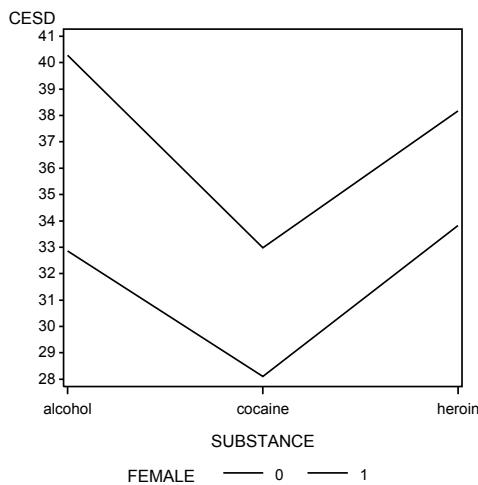
R has a function `interaction.plot()` to carry out this task. Figure 6.5 displays an interaction plot for CESD as a function of substance group and gender.

```

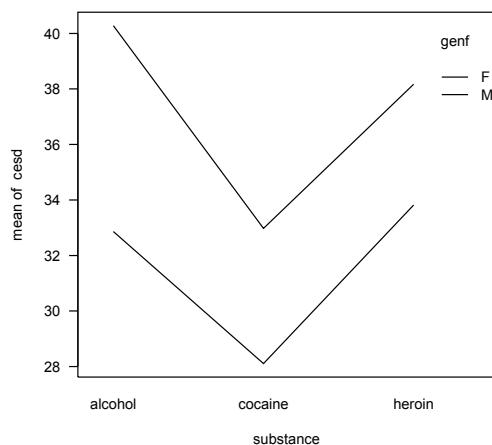
> ds = transform(ds, genf = as.factor(ifelse(female, "F", "M")))
> with(ds, interaction.plot(substance, genf, cesd,
+   xlab="substance", las=1, lwd=2))

```

There are indications of large effects of gender and substance group, but little suggestion of interaction between the two. The same conclusion is reached in Figure 6.6, which displays boxplots by substance group and gender.

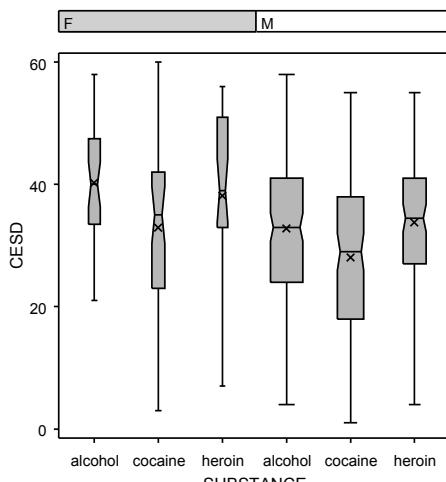


(a) SAS

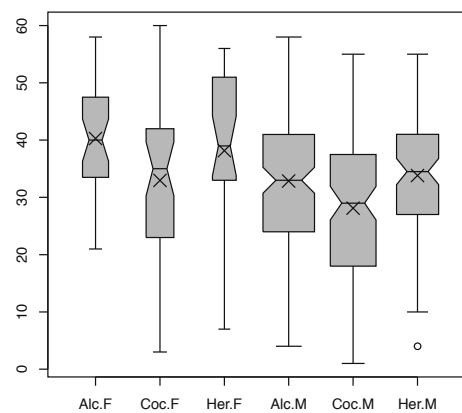


(b) R

Figure 6.5: Interaction plot of CESD as a function of substance group and gender



(a) SAS



(b) R

Figure 6.6: Boxplot of CESD as a function of substance group and gender

```

data h2; set k.help;
  if female eq 1 then sex='F';
  else sex='M';
run;
proc sort data=h2; by sex; run;

symbol1 v='x' c=black;
proc boxplot data=h2;
  plot cesd * substance(sex) / notches boxwidthscale=1;
run;

```

```
> library(memisc)
> ds = transform(ds, subs = cases(
+   "Alc" = substance=="alcohol",
+   "Coc" = substance=="cocaine",
+   "Her" = substance=="heroin"))
> boxout = with(ds,
+   boxplot(cesd ~ subs + genf, notch=TRUE, varwidth=TRUE,
+   col="gray80"))
> boxmeans = with(ds, tapply(cesd, list(subs, genf), mean))
> points(seq(boxout$n), boxmeans, pch=4, cex=2)
```

The width of each box is proportional to the size of the sample, with the notches denoting confidence intervals for the medians and X's marking the observed means. Next, we proceed to formally test whether there is a significant interaction through a two-way analysis of variance (6.1.8). In SAS, the Type III sums of squares table can be used to assess the interaction; we restrict output to this table to save space. In R we fit models with and without an interaction, and then compare the results. We also construct the likelihood ratio test manually.

```
options ls=74;
ods select modelanova;
proc glm data=k.help;
  class female substance;
  model cesd = female substance female*substance / ss3;
run;
```

The GLM Procedure

Dependent Variable: CESD

Source	DF	Type III SS	Mean Square	F Value	Pr > F
FEMALE	1	2463.232928	2463.232928	16.84	<.0001
SUBSTANCE	2	2540.208432	1270.104216	8.69	0.0002
FEMALE*SUBSTANCE	2	145.924987	72.962494	0.50	0.6075

```
> aov1 = aov(cesd ~ sub * genf, data=ds)
> aov2 = aov(cesd ~ sub + genf, data=ds)
> resid = residuals(aov2)
> anova(aov2, aov1)
```

Analysis of Variance Table

Model 1: cesd ~ sub + genf
Model 2: cesd ~ sub * genf
Res.Df RSS Df Sum of Sq F Pr(>F)
1 449 65515
2 447 65369 2 146 0.5 0.61

```

> options(digits=6)
> logLik(aov1)

'log Lik.' -1768.92 (df=7)

> logLik(aov2)

'log Lik.' -1769.42 (df=5)

> lldiff = logLik(aov1)[1] - logLik(aov2)[1]
> lldiff

[1] 0.505055

> 1 - pchisq(2*lldiff, df=2)

[1] 0.603472

> options(digits=3)
> summary(aov2)

```

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
sub	2	2704	1352	9.27	0.00011
genf	1	2569	2569	17.61	3.3e-05
Residuals	449	65515	146		

There is little evidence ( $p=0.61$ ) of an interaction, so this term can be dropped. For SAS, this means estimating the reduced model.

```

options ls=74; /* stay in gray box */
ods select overallanova parameterestimates;
proc glm data=k.help;
  class female substance;
  model cesd = female substance / ss3 solution;
run;

```

The GLM Procedure

Dependent Variable: CESD

Source	DF	Sum of Squares	Mean Square	F Value	Pr > F
Model	3	5273.13263	1757.71088	12.05	<.0001
Error	449	65515.35744	145.91394		
Corrected Total	452	70788.49007			

## The GLM Procedure

Dependent Variable: CESD

Parameter	Estimate	Standard		Pr >  t
		Error	t Value	
Intercept	39.13070331 B	1.48571047	26.34	<.0001
FEMALE 0	-5.61922564 B	1.33918653	-4.20	<.0001
FEMALE 1	0.00000000 B	.	.	.
SUBSTANCE alcohol	-0.28148966 B	1.41554315	-0.20	0.8425
SUBSTANCE cocaine	-5.60613722 B	1.46221461	-3.83	0.0001
SUBSTANCE heroin	0.00000000 B	.	.	.

The model was already fit in R to allow assessment of the interaction.

&gt; aov2

Call:

aov(formula = cesd ~ sub + genf, data = ds)

Terms:

	sub	genf	Residuals
Sum of Squares	2704	2569	65515
Deg. of Freedom	2	1	449

Residual standard error: 12.1

Estimated effects may be unbalanced

If results with the same referent categories used by SAS are desired, the default R design matrix (see 6.1.4) can be changed and the model re-fit.

```
> contrasts(ds$sub) = contr.SAS(3)
> aov3 = lm(cesd ~ sub + genf, data=ds)
> summary(aov3)
```

Call:

lm(formula = cesd ~ sub + genf, data = ds)

Residuals:

Min	1Q	Median	3Q	Max
-32.13	-8.85	1.09	8.48	27.09

Coefficients:

	Estimate	Std. Error	t value	Pr(> t )
(Intercept)	33.52	1.38	24.22	< 2e-16
sub1	5.61	1.46	3.83	0.00014
sub2	5.32	1.34	3.98	8.1e-05
genfM	-5.62	1.34	-4.20	3.3e-05

Residual standard error: 12.1 on 449 degrees of freedom

Multiple R-squared: 0.0745, Adjusted R-squared: 0.0683

F-statistic: 12 on 3 and 449 DF, p-value: 1.35e-07

The AIC criteria (7.8.3) can also be used to compare models. In SAS it is available in proc reg and proc mixed. Here we use proc mixed, omitting other output.

```
ods select fitstatistics;
proc mixed data=k.help method=ml;
  class female substance;
  model cesd = female|substance;
run; quit;
```

The Mixed Procedure

#### Fit Statistics

-2 Log Likelihood	3537.8
AIC (smaller is better)	3551.8
AICC (smaller is better)	3552.1
BIC (smaller is better)	3580.6

```
ods select fitstatistics;
proc mixed data=k.help method=ml;
  class female substance;
  model cesd = female substance;
run; quit;
ods select all;
```

The Mixed Procedure

#### Fit Statistics

-2 Log Likelihood	3538.8
AIC (smaller is better)	3548.8
AICC (smaller is better)	3549.0
BIC (smaller is better)	3569.4

> AIC(aov1)

[1] 3552

> AIC(aov2)

[1] 3549

The AIC criterion also suggests that the model without the interaction is most appropriate.

### 6.6.6 Multiple comparisons

We can also carry out multiple comparison (6.2.4) procedures to test each of the pairwise differences between substance abuse groups. In SAS this utilizes the `lsmeans` statement within `proc glm`.

```
ods select diff lsmeandiffcl lsmlines diffplot;
proc glm data=k.help;
  class substance;
  model cesd = substance;
  lsmeans substance / pdiff adjust=tukey cl lines;
run; quit;
ods select all;
```

The GLM Procedure  
 Least Squares Means  
 Adjustment for Multiple Comparisons: Tukey-Kramer

Least Squares Means for effect SUBSTANCE

Pr > |t| for H0: LSMean(i)=LSMean(j)

Dependent Variable: CESD

i/j	1	2	3
1		0.0009	0.9362
2	0.0009		0.0008
3	0.9362	0.0008	

The GLM Procedure  
 Least Squares Means  
 Adjustment for Multiple Comparisons: Tukey-Kramer

Least Squares Means for Effect SUBSTANCE

i	j	Difference Between Means	Simultaneous 95% Confidence Limits for LSMean(i)-LSMean(j)	
1	2	4.951829	1.753296	8.150362
1	3	-0.498086	-3.885335	2.889162
2	3	-5.449915	-8.950037	-1.949793

The GLM Procedure  
 Least Squares Means  
 Adjustment for Multiple Comparisons: Tukey-Kramer

Tukey-Kramer Comparison Lines for Least Squares Means of SUBSTANCE

LS-means with the same letter are not significantly different.

	CESD	LSMEAN	SUBSTANCE	Number
A	34.87097	heroin		3
A				
A	34.37288	alcohol		1
B	29.42105	cocaine		2

The above output demonstrates the results of the `lines` option using the `lsmeans` statement. The letter A shown on the left connecting the `heroin` and `alcohol` substances implies that there is not a statistically significant difference between these two groups. Since the `cocaine` substance has the letter B and no other group has one, the cocaine group is significantly different from each of the other groups. If instead the `cocaine` and `alcohol` substances **both** had a letter B attached, while the `heroin` and `alcohol` substances retained the letter A they have in the actual output, only the `heroin` and `cocaine` groups would differ significantly, while the `alcohol` group would differ from neither. This presentation becomes

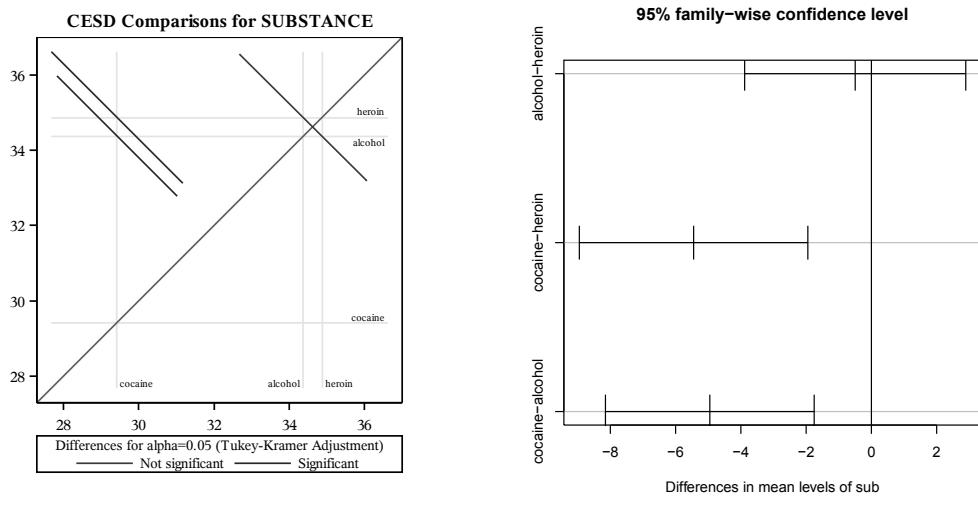


Figure 6.7: Pairwise comparisons

particularly useful as the number of groups increases. A graphical version called a `diffplot` is also produced; it is shown in Figure 6.7.

In R, we use the `TukeyHSD()` function.

```
> mult = TukeyHSD(aov(cesd ~ sub, data=ds), "sub")
> mult
```

```
Tukey multiple comparisons of means
 95% family-wise confidence level

Fit: aov(formula = cesd ~ sub, data = ds)

$sub
      diff    lwr    upr p adj
alcohol-heroin -0.498 -3.89  2.89 0.936
cocaine-heroin -5.450 -8.95 -1.95 0.001
cocaine-alcohol -4.952 -8.15 -1.75 0.001
```

The alcohol group and heroin group both have significantly higher CESD scores than the cocaine group, but the alcohol and heroin groups do not significantly differ from each other (95% CI for the difference ranges from  $-3.9$  to  $2.9$ ). Figure 6.7 provides a graphical display of the pairwise comparisons.

```
> plot(mult)
```

### 6.6.7 Contrasts

We can also fit contrasts (6.2.3) to test hypotheses involving multiple parameters. In this case, we can compare the CESD scores for the alcohol and heroin groups to the cocaine group. In SAS, to allow checking the contrast, we use the `e` option to the `estimate` statement.

```

ods select contrastcoef estimates;
proc glm data=k.help;
  class female substance;
  model cesd = female substance;
  output out=outanova residual=resid_ch4anova;
  estimate 'A+H = C?' substance 1 -2 1 / e;
run; quit;
ods select all;

```

The GLM Procedure

Coefficients for Estimate A+H = C?

	Row 1
Intercept	0
FEMALE 0	0
FEMALE 1	0
SUBSTANCE alcohol	1
SUBSTANCE cocaine	-2
SUBSTANCE heroin	1

The GLM Procedure

Dependent Variable: CESD

Parameter	Estimate	Standard Error	t Value	Pr >  t
A+H = C?	10.9307848	2.42008987	4.52	<.0001

```

> library(gmodels)
> levels(ds$sub)

[1] "heroin" "alcohol" "cocaine"

```

```
> fit.contrast(aov2, "sub", c(1,1,-2), conf.int=0.95 )
```

sub c=( 1 1 -2 )	Estimate	Std. Error	t value	Pr(> t )	lower CI	upper CI
	10.9	2.42	4.52	8.04e-06	6.17	15.7

As expected from the interaction plot (Figure 6.5), there is a statistically significant difference in this 1 degree of freedom comparison ( $p < 0.0001$ ).



# Chapter 7

# Regression generalizations and modeling

This chapter extends the discussion of linear regression introduced in Chapter 6 to include many commonly used statistical methods and models. Most SAS procedures mentioned in this chapter support the `class` statement for categorical covariates.

The CRAN statistics for the social sciences task view provide an excellent overview of methods described here and in Chapter 6.

## 7.1 Generalized linear models

Table 7.1 displays the options for SAS and R to specify link functions and family of distributions for generalized linear models [116]. Description of several specific generalized linear regression models (e.g., logistic and Poisson) can be found in subsequent sections of this chapter.

### SAS

```
proc genmod data=ds;
  model y = x1 ... xk / dist=familyname link=linkname;
run;
```

*Note:* The `class` statement in `proc genmod` is more flexible than that available in many other procedures, notably `proc glm`. However, the default behavior is the same as for `proc glm` (see 6.1.4).

### R

```
glmmod1 = glm(y ~ x1 + ... + xk, family="familyname"(link="linkname"),
  data=ds)
```

*Note:* More information on GLM families and links can be found using `help(family)`. Nested models can be compared using `anova(mymod2, mymod1, test="Chisq")`.

### 7.1.1 Logistic regression model

*Example: 7.10.1*

#### SAS

```
proc logistic data=ds;
  model y = x1 ... xk / or cl;
run;
```

Table 7.1: Generalized linear model distributions supported by SAS and R

Distribution	SAS PROC GENMOD	R <code>glm()</code>
Gaussian	<code>dist=normal</code>	<code>family="gaussian", link="identity", "log" or "inverse"</code>
binomial	<code>dist=binomial</code>	<code>family="binomial", link="logit", "probit", "cauchit", "log" or "cloglog"</code>
gamma	<code>dist=gamma</code>	<code>family="Gamma", link="inverse", "identity" or "log"</code>
Poisson	<code>dist=poisson</code>	<code>family="poisson", link="log", "identity" or "sqrt"</code>
inverse Gaussian	<code>dist=igaussian</code>	<code>family="inverse.gaussian", link="1/mu^2", "inverse", "identity" or "sqrt"</code>
multinomial	<code>dist=multinomial</code>	See <code>multinom()</code> in <code>nnet</code> package
negative binomial	<code>dist=negbin</code>	See <code>negative.binomial()</code> in <code>MASS</code> package
overdispersed	<code>dist=binomial</code> or <code>dist=multinomial</code> with <code>aggregate</code> , or <code>dist=poisson</code> <code>scale=deviance</code>	<code>family="quasi", link="logit", "probit", "cloglog", "identity", "inverse", "log", "1/mu^2" or "sqrt"</code> (see <code>glm.binomial.disp()</code> in the <code>dispm</code> package)

Note: For the `glm()` function in R, the available links for each distribution are listed. The following links are available for all distributions in SAS: `identity`, `log`, or `power(λ)` (where  $\lambda$  is specified by the user). For dichotomous outcomes, complementary log-log (`link=cloglog`), logit (`link=logit`), or probit (`link=probit`) are additionally available. For multinomial distributed outcomes, cumulative complementary log-log (`link=cumcl1`), cumulative logit (`link=cumlogit`), or cumulative probit (`link=cumprobit`) are available. Once the family and link functions have been specified, the variance function is implied (with the exception of the `quasi` family). In SAS, overdispersion is implemented using the `scale` option to the model statement. To allow overdispersion in Poisson, binomial, or multinomial models, use the option `scale=deviance`; the additional `aggregate` option is required for the binomial and multinomial. Any valid link listed above may be used.

or

```
proc logistic data=ds;
  model y(event='1') = x1 ... xk;
run;
or
proc logistic data=ds;
  model r/n = x1 ... xk / or cl; /* events/trials syntax */
run;
or
proc genmod data=ds;
  model y = x1 ... xk / dist=binomial link=logit;
run;
```

*Note:* While both procedures will fit logistic regression models, `proc logistic` is likely to be more useful for ordinary logistic regression than `proc genmod`. The former allows options such as those printed above in the first `model` statement, which produce the odds ratios (and their confidence limits) associated with the log-odds estimated by the model. It also produces the area under the ROC curve (the so-called “c” statistic) by default (see 8.5.7). Both procedures allow the logit, probit, and complementary log-log links, through the `link` option to the `model` statement; `proc genmod` must be used if other link functions are desired.

The `events/trials` syntax can be used to save storage space for data. In this case, observations with the same covariate values are stored as a single line of data, with the number of observations recorded in one variable (`trials`) and the number with the outcome in another (`events`).

The output from `proc logistic` and `proc genmod` prominently display the level of `y` that is being predicted. The `descending` option to the `proc` statement will reverse the order. Alternatively, the `model` statement in `proc logistic` allows you to specify the target level as shown in the second set of code.

The `class` statement in `proc genmod` is more flexible than that available in many other procedures, notably `proc glm`. Importantly, the default behavior is different than in `proc glm` (see 6.1.4).

## R

```
glm(y ~ x1 + ... + xk, binomial, data=ds)
or
library(rms)
lrm(y ~ x1 + ... + xk, data=ds)
```

*Note:* The `lrm()` function within the `rms` package provides the so-called “c” statistic (area under the ROC curve, see 8.5.7) and the Nagelkerke pseudo- $R^2$  index [125]. Nested models can be compared using `anova(mymod2, mymod1, test="Chisq")`.

### 7.1.2 Conditional logistic regression model

#### SAS

```
proc logistic data=ds;
  strata id;
  model y = x1 ... xk;
run;
or
proc logistic data=ds;
  strata id;
  model y = x1 ... xk;
  exact intercept x1;
run;
```

*Note:* The variable `id` identifies strata or matched sets of observations. An exact model can be fit using the `exact` statement with a list of covariates to be assessed using an exact test, including the intercept, as shown above.

## R

```
library(survival)
cmod = clogit(y ~ x1 + ... + xk + strata(id), data=ds)
```

*Note:* The variable `id` identifies strata or matched sets of observations. An exact model is fit by default.

### 7.1.3 Exact logistic regression

#### SAS

```
proc logistic data=ds;
  model y = x1 ... xk;
  exact intercept x1;
run;
```

*Note:* An exact test is generated for each variable listed in the `exact` statement, including, if desired, the intercept, as shown above. Not all covariates in the `model` statement need be included in the `exact` statement, but all covariates in the `exact` statement must be included in the `model` statement.

#### R

```
library(elrm)
ds = transform(ds, n=1)
elrmres = elrm(y/n ~ x1 + ... + xk, interest=~x1, iter=10000,
  burnIn=1000, data=ds)
```

*Note:* The `elrm()` function implements a modified MCMC algorithm to approximate exact conditional inference for logistic regression models [205] (see also the `logistiX` package).

### 7.1.4 Ordered logistic model

*Example:* 7.10.6

In this model the odds of each level of the outcome relative to all lower levels are calculated. The model requires that the proportional odds assumption holds.

#### SAS

```
proc genmod data=ds;
  model y = x1 ... xk / dist=multinomial;
run;
```

or

```
proc logistic data=ds;
  model y = x1 ... xk / link=cumlogit;
run;
```

*Note:* The `genmod` procedure applies a cumulative logit link by default when `dist=multinomial` is specified. The cumulative probit model is available with the `link=cprobit` option to the `model` statement in `proc genmod`. The `proc logistic` implementation provides a score test for the proportional odds assumption.

#### R

```
library(MASS)
polr(y ~ x1 + ... + xk, data=ds)
```

*Note:* The default link is logistic; this can be changed to probit, complementary log-log, or Cauchy using the `method` option (see also `ordered()`).

### 7.1.5 Generalized logistic model

*Example:* 7.10.7

#### SAS

```
proc logistic data=ds;
  model y = x1 ... xk / link=glogit;
run;
```

*Note:* Each level is compared to a reference level, which can be chosen using the `ref` option, e.g., `model y(ref='0') = x1 / link=glogit`.

**R**

```
library(VGAM)
mlogit = vglm(y ~ x1 + ... + xk, family=multinomial(), data=ds)
```

**7.1.6 Poisson model**

See 7.2.1 (zero-inflated Poisson).

*Example:* 7.10.2

**SAS**

```
proc genmod data=ds;
  model y = x1 ... xk / dist=poisson;
run;
```

*Note:* The default output from proc genmod includes useful methods to assess fit.

**R**

```
glm(y ~ x1 + ... + xk, poisson, data=ds)
```

*Note:* It is always important to check assumptions for models. This is particularly true for Poisson models, which are quite sensitive to model departures [73]. One way to assess the fit of the model is by comparing the observed and expected cell counts, and then calculating Pearson's chi-square statistic. This can be carried out using the `goodfit()` function.

**7.1.7 Negative binomial model**

See 7.2.2 (zero-inflated negative binomial).

*Example:* 7.10.4

**SAS**

```
proc genmod data=ds;
  model y = x1 ... xk / dist=negbin;
run;
```

**R**

```
library(MASS)
glm.nb(y ~ x1 + ... + xk, data=ds)
```

**7.1.8 Log-linear model**

Log-linear models are a flexible approach to analysis of categorical data [4]. A log-linear model of a three-dimensional contingency table denoted by  $X_1, X_2$ , and  $X_3$  might assert that the expected counts depend on a two-way interaction between the first two variables, but that  $X_3$  is independent of all the others:

$$\log(m_{ijk}) = \mu + \lambda_i^{X_1} + \lambda_j^{X_2} + \lambda_{ij}^{X_1, X_2} + \lambda_k^{X_3}$$

**SAS**

```
proc catmod data=ds;
  weight count;
  model x1*x2*x3 =_response_ / pred;
  loglin x1|x2 x3;
run;
```

*Note:* The variables listed in the `model` statement above describe the  $n$ -way table to be analyzed; the term `_response_` is a required keyword indicating a log-linear model. The `loglin` statement specifies the dependence assumptions. The `weight` statement is optional. If used, the `count` variable should contain the cell counts and can be used if the analysis is based on a summary dataset.

**R**

```
logres = loglin(table(x1, x2, x3), margin=list(c(1,2), c(3)), param=TRUE)
pvalue = 1 - with(logres, pchisq(lrt, df))
```

*Note:* The `margin` option specifies the dependence assumptions. In addition to the `loglin()` function, the `loglm()` function within the `MASS` package provides an interface for log-linear modeling.

## 7.2 Further generalizations

### 7.2.1 Zero-inflated Poisson model

*Example:* 7.10.3

Zero-inflated Poisson models can be used for count outcomes that generally follow a Poisson distribution but for which there are (many) more observed counts of 0 than would be expected. These data can be seen as deriving from a mixture distribution of a Poisson and a degenerate distribution with point mass at zero (see 7.2.2, zero-inflated negative binomial).

**SAS**

```
proc genmod data=ds;
  model y = x1 ... xk / dist=zip;
  zeromodel x2 ... xp;
run;
```

*Note:* The Poisson rate parameter of the model is specified in the `model` statement, with a default log link and alternate link functions available, as described in Table 7.1. The zero probability is modeled as a logistic regression of the covariates specified in the `zeromodel` statement. Support for zero-inflated Poisson models is also available within `proc countreg`.

**R**

```
library(pscl)
mod = zeroinfl(y ~ x1 + ... + xk | x2 + ... + xp, data=ds)
```

*Note:* The Poisson rate parameter of the model is specified in the usual way with a formula as argument to `zeroinfl()`. The default link is `log`. The zero probability is modeled as a function of the covariates specified after the “`|`” character. An intercept-only model can be fit by including `1` as the second model. Support for zero-inflated negative binomial and geometric models is available.

### 7.2.2 Zero-inflated negative binomial model

Zero-inflated negative binomial models can be used for count outcomes that generally follow a negative binomial distribution but for which there are (many) more observed counts of 0 than would be expected. These data can be seen as deriving from a mixture distribution of a negative binomial and a degenerate distribution with point mass at zero (see zero-inflated Poisson, 7.2.1).

**SAS**

```
proc countreg data=help2;
  model y = x1 ... xk / dist=zinb;
  zeromodel y ~ x2 ... xp;
run;
```

*Note:* The negative binomial rate parameter of the model is specified in the `model` statement. The zero probability is modeled as a function of the covariates specified after the `~` in the `zeromodel` statement.

**R**

```
library(pscl)
mod = zeroinfl(y ~ x1 + ... + xk | x2 + ... + xp, dist="negbin", data=ds)
```

*Note:* The negative binomial rate parameter of the model is specified in the usual way with a formula as argument to `zeroinfl()`. The default link is `log`. The zero probability is modeled as a function of the covariates specified after the “`|`” character. A single intercept for all observations can be fit by including `1` as the model.

### 7.2.3 Generalized additive model

**SAS**

*Example:* 7.10.8

```
proc gam data=ds;
  model y = spline(x1, df) loess(x2) spline2(x3, x4) ...
    param(x5 ... xk);
run;
```

*Note:* Specification of a spline or lowess term for variable `x1` is given by `spline(x1)` or `loess(x1)`, respectively, while a bivariate thin-plate spline fit can be included using `spline2(x1, x2)`. The degrees of freedom can be specified as in `spline(x1, df)`, following a comma in the variable function description, or estimated from the model using generalized cross-validation by including the `method=gcv` option in the model statement. If neither is specified, the default degrees of freedom of 4 is used. Any variables included in `param()` are fit as linear predictors with the usual syntax (6.1.6).

**R**

```
library(gam)
gam(y ~ s(x1, df) + lo(x2) + lo(x3, x4) + x5 + ... + xk, data=ds)
```

*Note:* Specification of a smooth term for variable `x1` is given by `s(x1)`, while a univariate or bivariate loess fit can be included using `lo(x1, x2)`. See `gam.s()` and `gam.lo()` within the `gam` package for details regarding specification of degrees of freedom or span, respectively. Polynomial regression terms can be fit using the `poly()` function.

### 7.2.4 Nonlinear least squares model

Nonlinear least squares models [163] can be fit flexibly within SAS and R. As an example, consider the income inequality model described by Sarabia and colleagues [150]:

$$Y = (1 - (1 - X)^p)^{(1/p)}$$

We provide a starting value (0.5) within the interior of the parameter space.

**SAS**

```
proc nlin data=ds;
  parms p=0.5;
  model y = (1 - ((1-x)**p))**(1/p);
run;
```

**R**

```
nls(y ~ (1 - (1-x)^{p})^{(1/p)}, start=list(p=0.5), trace=TRUE)
```

*Note:* Finding solutions for nonlinear least squares problems is often challenging (consult `help(nls)` for information on supported algorithms as well as 3.2.9, optimization).

## 7.3 Robust methods

### 7.3.1 Quantile regression model

*Example:* 7.10.5

Quantile regression predicts changes in the specified quantile of the outcome variable per unit change in the predictor variables, analogous to the change in the mean predicted in least squares regression. If the quantile so predicted is the median, this is equivalent to minimum absolute deviation regression (as compared to least squares regression minimizing the squared deviations).

#### SAS

```
proc quantreg data=ds;
  model y = x1 ... xk / quantile=0.75;
run;
```

*Note:* The `quantile` option specifies which quantile is to be estimated (here the 75th percentile). Median regression (i.e., `quantile=0.50`) is performed by default. If multiple quantiles are included (separated by commas), then they are estimated simultaneously; however, standard errors and tests are only carried out when a single quantile is provided.

#### R

```
library(quantreg)
quantmod = rq(y ~ x1 + ... + xk, tau=0.75, data=ds)
```

*Note:* The default for `tau` is 0.5, corresponding to median regression. If a vector is specified, the return value includes a matrix of results.

### 7.3.2 Robust regression model

Robust regression refers to methods for detecting outliers and/or providing stable estimates when they are present. Outlying variables in the outcome, predictor, or both are considered.

#### SAS

```
proc robustreg data=ds;
  model y = x1 ... xk / diagnostics leverage;
run;
```

*Note:* By default, M estimation is performed; other methods are accessed through the `method=` option to the `proc robustreg` statement, with valid methods including `lts`, `s`, and `mm`.

#### R

```
library(MASS)
rlm(y ~ x1 + ... + xk, data=ds)
```

*Note:* The `rlm()` function fits a robust linear model using M estimation. More information can be found in the CRAN robust statistical methods task view.

### 7.3.3 Ridge regression model

Ridge regression is used to deal with ill-conditional regression problems, particularly those due to multicollinearity (see also 7.8.5).

**SAS**

```
proc reg data=ds ridge=a to b by c;
  model y = x1 ... xk;
run;
```

*Note:* Each of the values  $a$ ,  $a+c$ ,  $a + 2c$ , ...,  $b$  is added to the diagonal of the cross-product matrix of  $X_1, \dots, X_k$ . Ridge regression estimates are the least squares estimates obtained using this new cross-product matrix.

**R**

```
library(MASS)
ridgemod = lm.ridge(y ~ x1 + ... + xk, lambda=seq(from=a, to=b, by=c),
  data=ds)
```

*Note:* Post-estimation functions supporting `ridgelm` objects include `plot()` and `select()`. A vector of ridge constants can be specified using the `lambda` option.

## 7.4 Models for correlated data

There is extensive support within SAS and R for correlated data regression models, including repeated measures, longitudinal, time series, clustered, and other related methods. Throughout this section we assume that repeated measurements are taken on a subject or cluster with a common value for the variable `id`.

### 7.4.1 Linear models with correlated outcomes

**SAS**

```
proc mixed data=ds;
  class id;
  model y = x1 ... xk;
  repeated / type=vartype subject=id;
run;
or
proc mixed data=ds;
  class id;
  model y = x1 ... xk / outpm=dsname;
  repeated ordervar / type=covtypename subject=id;
run;
```

*Example: 7.10.10*

*Note:* The `solution` option to the `model` statement can be used to get fixed effects parameter estimates in addition to ANOVA tables. The `repeated ordervar` syntax is used when observations within a cluster are 1) ordered (as in repeated measurements), 2) the placement in the order affects the covariance structure (as in most structures other than independence and compound symmetry), and 3) observations may be missing from the beginning or middle of the order. Predicted values for observations can be found using the `outpm` option to the `model` statement, as demonstrated in the second block of code. To add to the `outpm` dataset the outcomes and transformed residuals (scaled by the inverse Cholesky root of the marginal covariance matrix), add the `vcir` option to the `model` statement.

The structure of the covariance matrix of the observations is controlled by the `type` option to the `repeated` statement. Particularly useful structure options include `un` (unstructured), `cs` (compound symmetry), and `ar(1)` (first-order autoregressive). The full list is available through the on-line help: Contents; SAS Products; SAS Procedures; MIXED; Syntax; REPEATED.

**R**

```
library(nlme)
glsres = gls(y ~ x1 + ... + xk,
  correlation=corSymm(form = ~ ordervar | id),
  weights=varIdent(form = ~1 | ordervar), ds)
```

*Note:* The `gls()` function supports estimation of generalized least squares regression models with arbitrary specification of the variance covariance matrix. In addition to a formula interface for the mean model, the analyst specifies a within-group correlation structure as well as a description of the within-group heteroscedasticity structure (using the `weights` option). The statement `ordervar | id` implies that associations are assumed within `id`. Other covariance matrix options are available; see `help(corClasses)`.

### 7.4.2 Linear mixed models with random intercepts

See 7.4.3 (random slope models), 7.4.4 (random coefficient models), and 11.2 (empirical power calculations).

**SAS**

```
proc mixed data=ds;
  class id;
  model y = x1 ... xk;
  random int / subject=id;
run;
```

*Note:* The `solution` option to the `model` statement may be required to get fixed effects parameter estimates in addition to ANOVA tables. The `random` statement describes the design matrix for the random effects. Unlike the fixed effects design matrix, specified as usual with the `model` statement, the random effects design matrix includes a random intercept only if it is specified as above. The predicted random intercepts can be printed with the `solution` option to the `random` statement and saved into a dataset using the `ODS` system, e.g., an `ods output solutionr=reffs` statement. Predicted values for observations can be found using the `outp=datasetname` and `outpm=datasetname` options to the `model` statement; the `outp` dataset includes the predicted random effects in the predicted values while the `outpm` predictions include only the fixed effects.

**R**

```
library(nlme)
lmeint = lme(fixed= y ~ x1 + ... + xk, random = ~ 1 | id,
  na.action=na.omit, data=ds)
```

*Note:* Best linear unbiased predictors (BLUPs) of the sum of the fixed effects plus corresponding random effects can be generated using the `coef()` function, random effect estimates using the `random.effects()` function, and the estimated variance covariance matrix of the random effects using `VarCorr()` (see `fixef()` and `ranef()`). Normalized residuals (using a Cholesky decomposition, see pages 238–241 of Fitzmaurice et al. [42]) can be generated using the `type="normalized"` option when calling `residuals()` using an NLME option (more information can be found using `help(residuals.lme)`). A plot of the random effects can be created using `plot(lmeint)`.

### 7.4.3 Linear mixed models with random slopes

*Example:* 7.10.11

See 7.4.2 (random intercept models) and 7.4.4 (random coefficient models).

**SAS**

```
proc mixed data=ds;
  class id;
  model y = time x1 ... xk;
  random int time / subject=id type=covtypename;
run;
```

*Note:* The **solution** option to the **model** statement can be used to get fixed effects parameter estimates in addition to ANOVA tables. Random effects may be correlated with each other (though not with the residual errors for each observation). The structure of the covariance matrix of the random effects is controlled by the **type** option to the **random** statement. The option most likely to be useful is **type=un** (unstructured); by default, **proc mixed** uses the variance component (**type=vc**) structure, in which the random effects are uncorrelated with each other. The predicted random effects can be printed with the **solution** option to the **random** statement and saved into a dataset using the **ODS** system, e.g., an **ods output solutionr=reffs** statement. Predicted values for observations can be found using the **outp=datasetname** and **outpm=datasetname** options to the **model** statement; the **outp** dataset includes the predicted random effects in the predicted values while the **outpm** predictions include only the fixed effects.

**R**

```
library(nlme)
lmeslope = lme(fixed=y ~ time + x1 + ... + xk, random = ~ time | id,
  na.action=na.omit, data=ds)
```

*Note:* The default covariance for the random effects is unstructured (see **help(reStruct)** for other options). Best linear unbiased predictors (BLUPs) of the sum of the fixed effects plus corresponding random effects can be generated using the **coef()** function, random effect estimates using the **random.effects()** function, and the estimated variance covariance matrix of the random effects using **VarCorr()**. A plot of the random effects can be created using **plot(lmeint)**.

#### 7.4.4 More complex random coefficient models

We can extend the random effects models introduced in 7.4.2 and 7.4.3 to three or more subject-specific random parameters (e.g., a quadratic growth curve or spline/“broken stick” model [42]). We use *time<sub>1</sub>* and *time<sub>2</sub>* to refer to two generic functions of time.

**SAS**

```
proc mixed data=ds;
  class id;
  model y = time1 time2 x1 ... xk;
  random int time1 time2 / subject=id type=covtypename;
run;
```

*Note:* The **solution** option to the **model** statement can be used to get fixed effects parameter estimates in addition to ANOVA tables. Random effects may be correlated with each other, though not with the residual errors for each observation. The structure of the covariance matrix of the random effects is controlled by the **type** option to the **random** statement. The option most likely to be useful is **type=un** (unstructured); by default, **proc mixed** uses the variance component (**type=vc**) structure, in which the random effects are uncorrelated with each other. The predicted random effects can be printed with the **solution** option to the **random** statement and saved into a dataset using the **ODS** system, e.g., **ods output solutionr=reffs**. Predicted values for observations can be found using the **outp** and **outpm**

options to the `model` statement; the `outp` dataset includes the predicted random effects in the predicted values while the `outpm` predictions include only the fixed effects.

## R

```
library(nlme)
lmestick = lme(fixed= y ~ time1 + time2 + x1 + ... + xk,
  random = ~ time1 time2 | id, data=ds, na.action=na.omit)
```

*Note:* The default covariance for the random effects is unstructured (see `help(reStruct)` for other options). Best linear unbiased predictors (BLUPs) of the sum of the fixed effects plus corresponding random effects can be generated using the `coef()` function, random effect estimates using the `random.effects()` function, and the estimated variance covariance matrix of the random effects using `VarCorr()`. A plot of the random effects can be created using `plot(lmeint)`.

### 7.4.5 Multilevel models

Studies with multiple levels of clustering can be fit in SAS and R. In a typical example, a study might include schools (as one level of clustering) and classes within schools (a second level of clustering), with individual students within the classrooms providing a response. Generically, we refer to *level<sub>l</sub>* variables, which are identifiers of cluster membership at level *l*. Random effects at different levels are assumed to be uncorrelated with each other.

## SAS

```
proc mixed data=ds;
  class id;
  model y = x1 ... xk;
  random int / subject=level1;
  random int / subject=level2(level1);
run;
```

*Note:* Each `random` statement uses a `subject` option to describe a different clustering structure in the data. There's no theoretical limit to the complexity of the structure or the number of `random` statements, but practical difficulties in fitting the models may be encountered.

## R

```
library(nlme)
lmres = lme(fixed= y ~ x1 + ... + xk, random= ~ 1 | level1 / level2,
  data=ds)
```

*Note:* A model with *k* levels of clustering can be fit using the syntax: `level1 / ... / levelk`.

### 7.4.6 Generalized linear models with correlated outcomes

*Example:* 7.10.13

## SAS

```
proc glimmix data=ds;
  model y = time x1 ... xk / dist=familyname link=linkname;
  random residual / subject=id type=covtypename;
run;
```

*Note:* Observations sharing a value for `id` are correlated; otherwise, they are assumed independent. The structure of the covariance matrix of the random effects is controlled by the `type=` option to the `random` statement. The structure of the covariance matrix of the

observations is controlled by the `type=` option to the `repeated` statement. There are many available structures. The ones most likely to be useful include `un` (unstructured), `cs` (compound symmetry), and `ar(1)` (first-order autoregressive). The full list is available through the on-line help: Contents; SAS Products; Procedures; GLIMMIX; Syntax; RANDOM.

Note that these models and more general nonlinear models can be fit in SAS with the `nlmixed` procedure, as demonstrated in 10.1.3.

## R

```
library(lme4)
glmmres = glmer(y ~ x1 + ... + xk + (1|id), family=binomial(link="logit"),
  data=ds)
```

*Note:* See `help(family)` for other distribution families.

### 7.4.7 Generalized linear mixed models

#### SAS

*Examples:* 7.10.13 and 11.2

```
proc glimmix data=ds;
  model y = time x1 ... xk / dist=familyname link=linkname;
  random int time / subject=id type=covtypename;
run;
```

*Note:* Observations sharing a value for `id` are correlated; otherwise they are assumed independent. Random effects may be correlated with each other (though not with the residual errors for each observation). The structure of the covariance matrix of the random effects is controlled by the `type` option to the `random` statement. There are many available structures. The one most likely to be useful is `un` (unstructured). The full list is available through the on-line help: Contents; SAS Products; SAS Procedures; GLIMMIX; Syntax; RANDOM. All of the distributions and links shown in Table 7.1 are available, and additionally `dist` can be `beta`, `exponential`, `geometric`, `lognormal`, or `tcentral` ( $t$  distribution). An additional link is available for nominal categorical outcomes: `glogit`, the generalized logit. The `method=laplace` option to the `proc glimmix` statement will use a numeric integration method (this is likely to be time consuming).

## R

```
library(lme4)
glmmres = lmer(y ~ x1 + ... + xk + (1|id), family=familyval, data=ds)
```

*Note:* See `help(family)` for details regarding specification of distribution families and link functions.

### 7.4.8 Generalized estimating equations

#### SAS

*Example:* 7.10.12

```
proc genmod data=ds;
  model y = x1 ... xk;
  repeated subject=id / type=corrtype;
run;
```

*Note:* The `repeated ordervar` syntax should be used when observations within a cluster are a) ordered (as in repeated measurements), b) the placement in the order affects the covariance structure (as in most structures other than independence and compound symmetry), and c) observations may be missing from the beginning or middle of the order. The structure of the working covariance matrix of the observations is controlled by the `type` option to the `repeated` statement. The `corrtypes` include `ar` (first-order autoregressive),

`exch` (exchangeable), `ind` (independent), `mdep(m)` ( $m$ -dependent), `un` (unstructured), and `user` (a fixed, user-defined correlation matrix).

## R

```
library(gee)
geeres = gee(formula = y ~ x1 + ... + xk, id=id, data=ds,
             family=binomial, corstr="independence")
```

*Note:* The `gee()` function requires that the dataframe be sorted by subject identifier. Other correlation structures include "fixed", "stat\_M\_dep", "non\_stat\_M\_dep", "AR-M", and "unstructured". Note that the "unstructured" working correlation requires careful specification of ordering when missing data are monotone.

## 7.4.9 MANOVA

### SAS

```
proc glm data=ds;
  class x1;
  model y1 y2 y3 = x1;
  manova h=x1;
run;
```

*Note:* The number of categories in `x1` determines whether this is MANOVA or (if `x1` has two levels) the equivalent of Hotelling's  $T^2$  test.

## R

```
library(car)
mod = lm(cbind(y1, y2, y3) ~ x1, data=ds)
Anova(mod, type="III")
```

*Note:* The `car` package has a vignette which provides detailed examples, including a repeated measures ANOVA with details of use of the `idata` and `idesign` options. If the factor `x1` has two levels, this is the equivalent of a Hotelling's  $T^2$  test. The `Hotelling` package (due to James Curran) can also be used to calculate Hotelling's  $T^2$  statistic.

## 7.4.10 Time series model

Time series modeling is an extensive area with a specialized language and notation. We make only the briefest approach here. We display fitting an ARIMA (autoregressive integrated moving average) model for the first difference, with first-order autoregression and moving averages.

In SAS, the procedures to fit time series data are included in the SAS/ETS package. These provide extensive support for time series analysis. However, it is also possible to fit simple autoregressive models using `proc mixed`. We demonstrate the basic use of `proc arima` (from SAS/ETS). The CRAN time series task view provides an overview of support available for R.

### SAS

```
proc arima data=ds;
  identify var=x(1);
  estimate p=1 q=1;
run;
```

*Note:* In `proc arima`, the variable to be analyzed is specified in the `identify` statement, with differencing specified in parentheses. The `estimate` statement specifies the order of

the autoregression (p) and moving average (q). Prediction can be accomplished via the `forecast` statement.

## R

```
tsobj = ts(x, frequency=12, start=c(1992, 2))
arres = arima(tsobj, order=c(1, 1, 1))
```

*Note:* The `ts()` function creates a time series object, in this case for monthly time series data within the variable `x` beginning in February 1992 (the default behavior is that the series starts at time 1 and the number of observations per unit of time is 1). The `start` option is either a single number or a vector of two integers which specify a natural time unit and a number of samples into the time unit. The `arima()` function fits an ARIMA model with AR, differencing, and MA order all equal to 1.

## 7.5 Survival analysis

Survival, or failure time data, typically consist of the time until the event, as well as an indicator of whether the event was observed or censored at that time. Throughout, we denote the time of measurement with the variable `time` and censoring with a dichotomous variable `cens` = 1 if censored, or = 0 if observed. More information on survival (or failure time, or time-to-event) analysis in R can be found in the CRAN survival analysis task view (see B.6.2). Other entries related to survival analysis include 5.4.6 (log-rank test) and 8.5.11 (Kaplan–Meier plot).

### 7.5.1 Proportional hazards (Cox) regression model

#### SAS

*Example: 7.10.14*

```
proc phreg data=ds;
  model time*cens(1) = x1 ... xk;
run;
```

*Note:* SAS supports time varying covariates using programming statements within `proc phreg`. The `class` statement in `proc genmod` is more flexible than that available in many other procedures, notably `proc glm`. However, the default behavior is the same as for `proc glm` (see 6.1.4).

## R

```
library(survival)
survmod = coxph(Surv(time, cens) ~ x1 + ... + xk)
```

*Note:* The Efron estimator is the default; other choices including exact and Breslow can be specified using the `method` option. The `cph()` function within the `rms` package supports time varying covariates, while the `cox.zph()` function within the `survival` package allows testing of the proportionality assumption, as does the `simPH` package.

### 7.5.2 Proportional hazards (Cox) model with frailty

#### SAS

```
proc phreg data=ds;
  class id;
  model time*cens(1) = x1 x2 ...;
  random id / noclprint;
run;
```

*Note:* The `noclprint` option suppresses the printing of every level of the `id` variable.

**R**

```
library(survival)
coxph(Surv(time, cens) ~ x1 + ... + xk + frailty(id), data=ds)
```

*Note:* More information on specification of frailty models can be found using `help(frailty)`; support is available for t, Gamma, and Gaussian distributions.

### 7.5.3 Nelson–Aalen estimate of cumulative hazard

The Nelson–Aalen method provides a nonparametric estimator of the cumulative hazard rate function in censored data problems [193].

**SAS**

```
ods output productlimitestimates=naout;
proc lifetest data=ds nelson;
  time time*cens(1);
run;
```

*Note:* The `nelson` option generates the Nelson–Aalen estimates. The `ODS` syntax shown here will save the estimates into a new dataset.

**R**

```
calcna = function(time, event) {
  na.fit = survfit(coxph(Surv(time, event) ~ 1), type="aalen")
  jumps = c(0, na.fit$time, max(time))
  # need to be careful at the beginning and end
  surv = c(1, na.fit$surv, na.fit$surv[length(na.fit$surv)])
  
  # apply appropriate transformation
  neglogsurv = -log(surv)

  # create placeholder of correct length
  naest = numeric(length(time))
  for (i in 2:length(jumps)) {
    naest[which(time>=jumps[i-1] & time<=jumps[i])] =
      neglogsurv[i-1] # select the appropriate value
  }
  return(naest)
}
or
basehaz(coxph(Surv(time, event) ~ 1))
```

*Note:* We can do the necessary housekeeping in R, using the fact that the Nelson–Aalen estimate is just the negative log of the survival function (after specifying the `type="aalen"` option). Similar estimates can be generated using the `basehaz()` function.

### 7.5.4 Testing the proportionality of the Cox model

There are several methods for assessing whether the proportionality assumption holds.

**SAS**

```

proc phreg data=ds;
  model time*cens(1) = x1;
  output out=propcheck ressch=schres;
run;

proc sgplot data=propcheck;
  loess x=dayslink y=schres / clm;
run;
or
proc phreg data=ds;
  model time*cens(1) = x1;
  assess ph;
run;
or
proc phreg data=ds;
  model time*cens(1) = x1 x1_time;
  x1_time = x1 * log(time);
run;

```

*Note:* In the first code we use a visual inspection of the Schoenfeld residuals [159]. The **ressch=name** option to the **output** statement will add the Schoenfeld residuals to the output dataset. The **sgplot** procedure will generate the plot needed for visual inspection: a non-zero slope in any region of the plot suggests a lack of proportionality.

In the second code we use the built-in assessment, which applies the method of Lin, Wei, and Ying [105].

In the third code block we add a time-varying effect (7.5.5) of the predictor to the model. If this is statistically significant, the null hypothesis of proportionality has been rejected.

**R**

```

library(survival)
survmod = coxph(Surv(time, cens) ~ x1 + ... + xk)
cox.zph(survmod)
plot(cox.zph(survmod))

```

*Note:* The **cox.zph()** function supports a **plot** object to generate graphical displays which facilitate model assessment (see also John Fox's Cox regression appendix at <http://tinyurl.com/foxcox>).

### 7.5.5 Cox model with time-varying predictors

**SAS**

```

proc phreg data=ds;
  model time*cens(1) = x1 x1_time;
  x1_time = x1 * log(time);
run;

```

*Note:* Note that **proc phreg** allows data-step-like code to construct variables.

**R**

*Note:* Estimation of the Cox model with time-varying predictors in R requires the creation of a dataset with separate time periods for each occurrence of the time-varying predictor. More details can be found in John Fox's Cox regression appendix at <http://tinyurl.com/foxcox>.

## 7.6 Multivariate statistics and discriminant procedures

This section describes some commonly used multivariate, clustering methods, and discriminant procedures [113, 171].

In SAS, summaries of these topics and how to implement related methods are discussed in the on-line help: Contents; SAS Products; SAS/STAT User's Guide, under the headings "Introduction to Multivariate Procedures," "Introduction to Clustering Procedures," and "Introduction to Discriminant Procedures." The multivariate statistics, cluster analysis, and psychometrics task views on CRAN provide additional descriptions of functionality available within R.

### 7.6.1 Cronbach's $\alpha$

*Example: 7.10.15*

Cronbach's  $\alpha$  is a measure of internal consistency for a multi-item measure.

#### SAS

```
ods select cronbachalpha;
proc corr data=help alpha nomiss;
  var x1 -- xk;
run;
ods exclude none;
```

*Note:* The `nomiss` option is required to include only observations with all variables observed.

#### R

```
library(multilevel)
cronbach(cbind(x1, x2, ..., xk))
```

### 7.6.2 Factor analysis

*Example: 7.10.16*

Factor analysis is used to explain the variability of a set of measures in terms of underlying unobservable factors. The observed measures can be expressed as linear combinations of the factors, plus random error. Factor analysis is often used as a way to guide the creation of summary scores from individual items.

#### SAS

```
ods select orthrotfactpat factor.rotatedsolution.finalcommunwgt;
proc factor data=ds nfactors=3 method=ml rotate=varimax;
  var x1--xk;
run;
```

#### R

```
res = factanal(~ x1 + ... + xk, factors=3, scores="regression")
print(res, cutoff=0.45, sort=TRUE)
```

### 7.6.3 Recursive partitioning

*Example: 7.10.17*

Recursive partitioning is used to create a decision tree to classify observations from a dataset based on categorical predictors. Recursive partitioning is only available in SAS through SAS Enterprise Miner, a module not included with the educational license typically purchased by universities, or through relatively expensive third-party add-ons to SAS. Within R, this functionality is available within the `rpart` package.

**R**

```
library(rpart)
mod.rpart = rpart(y ~ x1 + ... + xk, method="class", data=ds)
printcp(mod.rpart)
plot(mod.rpart)
text(mod.rpart)
```

*Note:* The `partykit` package provides more control of the display of regression trees (see also the CRAN machine learning task view).

### 7.6.4 Linear discriminant analysis

*Example: 7.10.18*

Linear (or Fisher) discriminant analysis is used to find linear combinations of variables that can predict class membership.

**SAS**

```
ods select lineardiscfunc classifiedresub errorresub;
proc discrim data=ds out=ldaout;
  class y;
  var x1 x2 .. xk;
run;
```

**R**

```
library(MASS)
ngroups = length(levels(group))
ldamodel = lda(y ~ x1 + ... + xk,
  prior=rep(1/ngroups, ngroups))
print(ldamodel)
```

### 7.6.5 Latent class analysis

Latent class analysis is a technique used to classify observations based on patterns of categorical responses.

Support for this method in SAS is available through the `proc lca` and `proc lta` add-on routines created and distributed by the Methodology Center at Penn State University, which can be downloaded from <http://methodology.psu.edu/index.php/downloads/proclcalta>. While it's customary in R to use researcher-written routines, it's less so for SAS; the machinery which allows independently written procs thus has the potential to mislead users. It bears explicitly stating that third-party procs probably don't have the same level of robustness or support as those distributed by SAS Institute.

**SAS**

```
proc lca data=ds;
  nclass c;
  items x1 x2 ... xk;
  categories 2 2 2 2;
  seed 42;
  nstarts n;
run;
```

*Note:* The `nclass` statement requests the data be divided into `c` classes. The remaining statements transparently specify the variables to include, the number of categories per variable, and information about random starts. It's highly recommended to set `n` to a relatively large number of random starts, to ensure that the true maximum likelihood estimate is reached.

**R**

```
library(poLCA)
poLCA(cbind(x1, x2, ..., x3) ~ 1, maxiter=50000, nclass=k, nrep=n, data=ds)
```

*Note:* In this example, a  $k$  class model is fit. The `poLCA()` function requires that the variables are coded as positive integers. Other support for latent class models in R can be found in the `randomLCA` and the `MplusAutomation` packages.

### 7.6.6 Hierarchical clustering

*Example: 7.10.19*

Many techniques exist for grouping similar variables or similar observations. These groups, or clusters, can be overlapping or disjoint, and are sometimes placed in a hierarchical structure so that some disjoint clusters share a higher-level cluster. Within SAS, the procedures `cluster`, `fastclus`, and `modeclus` can be used to find clusters of observations; the `varclus` and `factor` procedures can be used to find clusters of variables. The `tree` procedure can be used to plot tree diagrams from hierarchical clustering results. In R, there are many packages which perform clustering. Clustering tools included with the R distribution as part of the `stats` package include `hclust()` and `kmeans()`. The function `dendrogram()`, also in the `stats` package, plots tree diagrams. The `cluster` package, included with the R distribution, contains functions `pam()`, `clara()`, and `diana()`. The CRAN clustering task view has more details.

**SAS**

```
proc varclus data=help outtree=treedisp centroid;
  var x1 ... xk;
run;
ods exclude none;

proc tree data=treedisp nclusters=5;
  height _varexp_;
run;
```

**R**

```
cormat = cor(cbind(x1, x2, ..., xk),
  use="pairwise.complete.obs")
hclustobj = hclust(dist(cormat))
```

## 7.7 Complex survey design

The appropriate analysis of sample surveys requires incorporation of complex design features, including stratification, clustering, weights, and finite population correction. These can be addressed in SAS and R for many common models. In this example, we assume that there are variables `psuvar` (cluster or primary sampling units), `stratum` (stratification variable), and `wt` (sampling weight). Code examples are given to estimate the mean of a variable `x1` as well as a linear regression model.

**SAS**

```
proc surveymeans data=ds rate=fpcvar;
  cluster psuvar;
  strata stratum;
  weight wt;
  var x1 ... xk;
run;
```

or

```
proc surveyreg data=ds rate=fpcvar;
  cluster psuvar;
  strata stratum;
  weight wt;
  model y = x1 ... xk;
run;
```

*Note:* The `surveymeans` and `surveyreg` procedures account for complex survey designs with equivalent functionality to `means` and `reg`, respectively. Other survey procedures in SAS include `surveyfreq` and `surveylogistic`, which emulate procedures `freq` and `logistic`. The survey procedures share a `strata` statement to describe the stratification variables, a `weight` statement to describe the sampling weights, a `cluster` statement to specify the PSU or cluster, and a `rate` option (for the `proc` statement) to specify a finite population correction as a count or dataset. Additional options allow specification of the total number of primary sampling units (PSUs) or a dataset with the number of PSUs in each stratum.

**R**

```
library(survey)
mydesign = svydesign(id=~psuvar, strata=~stratum, weights=~wt,
  fpc=~fpcvar, data=ds)
meanres = svymean(~ x1, mydesign)
regres = svyglm(y ~ x1 + ... + xk, design=mydesign)
```

*Note:* Thomas Lumley's `survey` package includes support for many models. Illustrated above are means and linear regression models, with specification of PSUs, stratification, weight, and FPC. The CRAN official statistics task view provides an overview of other implementations.

## 7.8 Model selection and assessment

### 7.8.1 Compare two models

*Example:* 6.6.5

Model comparison marks a key point of divergence for SAS and R. In general, most procedures in SAS fit a single model. Comparisons between models must be constructed by hand. An exception is “leave-one-out” models, in which a model identical to the one fit is considered, except that a single predictor is to be omitted. In this case, SAS offers “Type III” sums of squares tests, which can be printed by default or request in many modeling procedures. The R function `drop1()` computes a table of changes in fit. In addition, R offers functions which compare nested models using the `anova()` function. The Wald tests calculated by SAS and the likelihood ratio tests from `anova()` are identical in many settings, though they differ in general. In cases in which they differ, likelihood ratio tests are to be preferred.

**R**

```
mod1 = lm(y ~ x1 + ... + xk, data=ds)
mod2 = lm(y ~ x3 + ... + xk, data=ds)
anova(mod2, mod1)
or
drop1(mod2)
```

*Note:* The `anova()` command in R computes analysis of variance (or deviance) tables. When given one model as an argument, it displays the ANOVA table. When two (or more) nested models are given, it calculates the differences between them.

## 7.8.2 Log-likelihood

See 7.8.3 (AIC).

*Example:* 6.6.5

### SAS

```
proc mixed data=ds;
  model y = x1 ... xk;
run;
```

*Note:* Log-likelihood values are produced by various SAS procedures, but the means of requesting them can be idiosyncratic. The `mixed` procedure fits a superset of models available in `proc glm` and can be used to generate this quantity.

### R

```
mod1 = lm(y ~ x1 + ... + xk, data=ds)
logLik(mod1)
```

*Note:* The `logLik()` can operate on `glm`, `lm`, `multinom`, `nls`, `Arima`, `gls`, `lme`, and `nlme` objects, among others.

## 7.8.3 Akaike Information Criterion (AIC)

See 7.8.2 (log-likelihood).

*Example:* 6.6.5

### SAS

```
proc reg data=ds stats=aic;
  model y = x1 ... xk;
run;
```

*Note:* AIC values are available in various SAS procedures, but the means of requesting them can be idiosyncratic.

### R

```
mod1 = lm(y ~ x1 + ... + xk, data=ds)
AIC(mod1)
```

*Note:* The `AIC()` function includes support for `glm`, `lm`, `multinom`, `nls`, `Arima`, `gls`, `lme`, and `nlme` objects.

## 7.8.4 Bayesian Information Criterion (BIC)

See 7.8.3 (AIC).

### SAS

```
proc mixed data=ds;
  model y = x1 ... xk;
run;
```

*Note:* BIC values are presented by default in `proc mixed`.

### R

```
mod1 = lm(y ~ x1 + ... + xk, data=ds)
library(nlme)
BIC(mod1)
```

### 7.8.5 LASSO model

The LASSO (least absolute shrinkage and selection operator) is a model selection method for linear regression that minimizes the sum of squared errors subject to a constraint on the sum of the absolute value of the coefficients. This technique, due to Tibshirani [176] is particularly useful in data mining situations where a large number of predictors is being considered for inclusion in the model (see also 7.3.3).

#### SAS

```
proc glmselect;
  model y = x1 ... xk / selection=lasso;
```

*Note:* The `glmselect` procedure includes several other model selection algorithms, including stepwise procedures and least angle regression.

#### R

```
library(lars)
lars(y ~ x1 + ... + xk, data=ds, type="lasso")
```

*Note:* The `lars()` function also implements least angle regression and forward stagewise methods.

### 7.8.6 Hosmer–Lemeshow goodness of fit

#### SAS

```
proc logistic data=ds;
  model y = x1 x2 ... / lackfit;
run;
```

*Note:* The `lackfit` option generates a test based on 10 categories.

#### R

```
hosmerlem = function(y, yhat, g=10) {
  cutyhat = cut(yhat,
    breaks = quantile(yhat, probs=seq(0,
      1, 1/g)), include.lowest=TRUE)
  obs = xtabs(cbind(1 - y, y) ~ cutyhat)
  expect = xtabs(cbind(1 - yhat, yhat) ~ cutyhat)
  chisq = sum((obs - expect)^2/expect)
  P = 1 - pchisq(chisq, g-2)
  return(list(chisq=chisq, p.value=P))
}
```

*Note:* The test is straightforward to code directly. The `hosmerlem()` function accepts a vector of observed 0 and 1 outcomes and predicted probabilities. For a more refined version that accepts a model object as input, see <http://tinyurl.com/sasrblog-hosmer-lemeshow>.

### 7.8.7 Goodness of fit for count models

#### SAS

```
proc genmod data = ds;
  model y = x1 x2 ... / dist=poisson;
```

*Example:* 7.10.2

*Note:* As part of the default output, SAS provides the deviance and degrees of freedom. Under the null, the deviance has the  $\chi^2$  distribution with these degrees of freedom (DF). Informally, this means that deviance/DF should be around 1.

```
library(vcd)
poisfit = goodfit(x, "poisson")
```

The `goodfit()` function carries out a Pearson's  $\chi^2$  test of observed vs. expected counts. Other distributions supported include `binomial` and `nbinomial`.

Using the code below, R can also create a hanging rootogram [181] to assess the goodness of fit for count models. If the model fits well, then the bottom of each bar in the rootogram should be near zero.

```
library(vcd)
rootogram(poisfit)
```

## 7.9 Further resources

Many of the topics covered in this chapter are active areas of statistical research and many foundational articles are still useful. Here we provide references to texts which serve as accessible references.

Dobson and Barnett [34] is an accessible introduction to generalized linear models, while [116] remains a classic. Agresti [4] describes the analysis of categorical data. The CRAN statistics for the social sciences task view provides an overview of R support in this area.

Fitzmaurice, Laird, and Ware [42] is an accessible overview of mixed effects methods while [191] reviews these methods for a variety of statistical packages. A comprehensive review of the material in this chapter is incorporated in [39]. The text by Hardin and Hilbe [61] provides a review of generalized estimating equations. The CRAN analysis of spatial data task view provides a summary of tools to read, visualize, and analyze spatial data.

Collett [27] is an accessible introduction to survival analysis.

Manly [113] and Tabachnick and Fidell [171] provide a comprehensive introduction to multivariate statistics. Särndal, Swensson, and Wretman [152] provides a readable overview of the analysis of data from complex surveys.

## 7.10 Examples

To help illustrate the tools presented in this chapter, we apply many of the entries to the HELP data. SAS and R code can be downloaded from <http://www.amherst.edu/~nhorton/sasr2/examples>.

In general, SAS output is lengthier than R `summary()` results. We annotate the full output with named ODS objects for logistic regression (7.10.1), provide the bulk of results in some examples, and utilize ODS to reduce the output to a few key elements for the sake of brevity for most entries.

```
libname k "c:/book";

data help;
set k.help;
run;

> options(digits=3)
> options(show.signif.stars=FALSE)
> load("savedfile") # saved from previous chapter
```

The R dataset can be read in from a previously saved file (see p. 43 and 2.6.1).

### 7.10.1 Logistic regression

In this example we fit a logistic regression (7.1.1) where we model the probability of being homeless (spending one or more nights in a shelter or on the street in the past six months)

as a function of predictors.

We can specify the `param` option to make the SAS reference category match the default in R (see 6.1.4).

```
options ls=74; /* keep output in grey box */
proc logistic data=help descending;
  class substance (param=ref ref='alcohol');
  model homeless = female i1 substance sexrisk indtot;
run;
```

SAS produces a large number of distinct pieces of output by default. Here we reproduce the ODS name of each piece of output. Running `ods trace on / listing` before the procedure, as introduced in A.7, will display the ODS objects available. Each of these can be saved as a SAS dataset using these names with the `ods output` statement as on page 176.

First, SAS reports basic information about the model and the data in the ODS `modelinfo` output.

#### The LOGISTIC Procedure

##### Model Information

Data Set	WORK.HELP
Response Variable	HOMELESS
Number of Response Levels	2
Model	binary logit
Optimization Technique	Fisher's scoring

Then SAS reports the number of observations read and used, in the ODS `nobs` output. Note that missing data will cause these numbers to differ. Subsetting with the `where` statement (A.6.2) will cause the number of observations displayed here to differ from the number in the dataset.

Number of Observations Read	453
Number of Observations Used	453

The ODS `responseprofile` output tabulates the number of observations with each outcome, and, importantly, reports which level is being modeled as the event.

##### Response Profile

Ordered Value	HOMELESS	Total
		Frequency
1	1	209
2	0	244

Probability modeled is HOMELESS=1.

The ODS `classlevelinfo` output shows the coding for each `class` variable.

##### Class Level Information

Class	Value	Design	
		Variables	
SUBSTANCE	alcohol	0	0
	cocaine	1	0
	heroin	0	1

Whether the model converged is reported in the ODS `convergencestatus` output.

### Model Convergence Status

Convergence criterion (GCONV=1E-8) satisfied.

AIC (7.8.3) and other fit statistics are produced in the ODS fitstatistics output.

### Model Fit Statistics

Criterion	Intercept	
	Intercept Only	and Covariates
AIC	627.284	590.652
SC	631.400	619.463
-2 Log L	625.284	576.652

Tests reported in the ODS globaltests output assess the joint null hypothesis that all parameters except the intercept equal 0.

### Testing Global Null Hypothesis: BETA=0

Test	Chi-Square	DF	Pr > ChiSq
Likelihood Ratio	48.6324	6	<.0001
Score	45.6522	6	<.0001
Wald	40.7207	6	<.0001

The ODS type3 output contains tests for each covariate (including joint tests for **class** variables with two or more values) conditional on all other covariates being included in the model.

### Type 3 Analysis of Effects

Effect	DF	Wald	
		Chi-Square	Pr > ChiSq
FEMALE	1	1.0831	0.2980
I1	1	7.6866	0.0056
SUBSTANCE	2	4.2560	0.1191
SEXRISK	1	3.4959	0.0615
INDTOT	1	8.2868	0.0040

The ODS parameterestimates output shows the maximum likelihood estimates of the parameters, their standard errors, and Wald statistics and tests for the null hypothesis that the parameter value is 0. Note that in this table, as opposed to the previous one, each level (other than the referent) of any **class** variable is reported separately.

### Analysis of Maximum Likelihood Estimates

Parameter	DF	Estimate	Standard	Chi-Square	Pr > ChiSq
			Error		
Intercept	1	-2.1319	0.6335	11.3262	0.0008
FEMALE	1	-0.2617	0.2515	1.0831	0.2980
I1	1	0.0175	0.00631	7.6866	0.0056
SUBSTANCE cocaine	1	-0.5033	0.2645	3.6206	0.0571
SUBSTANCE heroin	1	-0.4431	0.2703	2.6877	0.1011
SEXRISK	1	0.0725	0.0388	3.4959	0.0615
INDTOT	1	0.0467	0.0162	8.2868	0.0040

The ODS `oddsratios` output shows the exponentiated parameter estimates and associated confidence limits.

#### Odds Ratio Estimates

Effect	Point Estimate	95% Wald Confidence Limits	
FEMALE	0.770	0.470	1.260
I1	1.018	1.005	1.030
SUBSTANCE cocaine vs alcohol	0.605	0.360	1.015
SUBSTANCE heroin vs alcohol	0.642	0.378	1.091
SEXRISK	1.075	0.997	1.160
INDTOT	1.048	1.015	1.082

The ODS `association` output shows various other statistics, including the area under the ROC curve, denoted "c" by SAS.

#### Association of Predicted Probabilities and Observed Responses

Percent Concordant	67.8	Somers' D	0.360
Percent Discordant	31.8	Gamma	0.361
Percent Tied	0.4	Tau-a	0.179
Pairs	50996	c	0.680

Within R, we use the `glm()` command to fit the logistic regression model.

```
> logres = glm(homeless ~ female + i1 + substance + sexrisk + indtot,
   binomial, data=ds)
> summary(logres)
```

Call:

```
glm(formula = homeless ~ female + i1 + substance + sexrisk +
    indtot, family = binomial, data = ds)
```

Deviance Residuals:

Min	1Q	Median	3Q	Max
-1.75	-1.04	-0.70	1.13	2.03

Coefficients:

	Estimate	Std. Error	z value	Pr(> z )
(Intercept)	-2.13192	0.63347	-3.37	0.00076
female	-0.26170	0.25146	-1.04	0.29800
i1	0.01749	0.00631	2.77	0.00556
substancecocaine	-0.50335	0.26453	-1.90	0.05707
substanceheroin	-0.44314	0.27030	-1.64	0.10113
sexrisk	0.07251	0.03878	1.87	0.06152
indtot	0.04669	0.01622	2.88	0.00399

(Dispersion parameter for binomial family taken to be 1)

Null deviance: 625.28 on 452 degrees of freedom  
 Residual deviance: 576.65 on 446 degrees of freedom  
 AIC: 590.7

Number of Fisher Scoring iterations: 4

If the parameter estimates are desired as a dataset, ODS can be used in SAS.

```
ods exclude all;
ods output parameterestimates=helplogisticbetas;
proc logistic data=help descending;
  class substance (param=ref ref='alcohol');
  model homeless = female i1 substance sexrisk indtot;
run;

ods exclude none;
options ls=74;
proc print data=helplogisticbetas;
run;
```

Obs	Variable	Class		DF	Estimate	StdErr	WaldChiSq	ChiSq	Prob	_ESTTYPE_
		Val0								
1	Intercept		1	-2.1319	0.6335		11.3262	0.0008		MLE
2	FEMALE		1	-0.2617	0.2515		1.0831	0.2980		MLE
3	I1		1	0.0175	0.00631		7.6866	0.0056		MLE
4	SUBSTANCE cocaine		1	-0.5033	0.2645		3.6206	0.0571		MLE
5	SUBSTANCE heroin		1	-0.4431	0.2703		2.6877	0.1011		MLE
6	SEXRISK		1	0.0725	0.0388		3.4959	0.0615		MLE
7	INDTOT		1	0.0467	0.0162		8.2868	0.0040		MLE

Similar information can be found in the `summary()` output object in R.

```
> names(summary(logres))
[1] "call"          "terms"         "family"        "deviance"
[5] "aic"           "contrasts"     "df.residual"   "null.deviance"
[9] "df.null"       "iter"          "deviance.resid" "coefficients"
[13] "aliased"       "dispersion"    "df"            "cov.unscaled"
[17] "cov.scaled"

> coeff.like.SAS = summary(logres)$coefficients
> coeff.like.SAS

              Estimate Std. Error z value Pr(>|z|)
(Intercept) -2.1319    0.63347 -3.37 0.000764
female       -0.2617    0.25146 -1.04 0.297998
i1            0.0175    0.00631  2.77 0.005563
substancecocaine -0.5033   0.26453 -1.90 0.057068
substanceheroin  -0.4431   0.27030 -1.64 0.101128
sexrisk        0.0725    0.03878  1.87 0.061518
indtot         0.0467    0.01622  2.88 0.003993
```

## 7.10.2 Poisson regression

In this example we fit a Poisson regression model (7.1.6) for `i1`, the average number of drinks per day in the 30 days prior to entering the detox center.

Because `proc genmod` lacks an easy way to specify the reference category, the R results have a different intercept and different effects for the abuse groups.

```

options ls=74;
ods exclude modelinfo nobs classlevels convergencestatus;
proc genmod data=help;
  class substance;
  model i1 = female substance age / dist=poisson;
run;

```

The GENMOD Procedure

#### Criteria For Assessing Goodness Of Fit

Criterion	DF	Value	Value/DF
Deviance	448	6713.8986	14.9864
Scaled Deviance	448	6713.8986	14.9864
Pearson Chi-Square	448	7933.2027	17.7080
Scaled Pearson X2	448	7933.2027	17.7080
Log Likelihood		16385.3197	
Full Log Likelihood		-4207.6544	
AIC (smaller is better)		8425.3089	
AICC (smaller is better)		8425.4431	
BIC (smaller is better)		8445.8883	

In the following output, the confidence limits for the parameter estimates, which appear by default in SAS, have been removed.

#### Analysis Of Maximum Likelihood Parameter Estimates

Parameter	DF	Estimate	Standard	Wald	Pr > ChiSq
			Error	Chi-Square	
Intercept	1	1.7767	0.0582	930.73	<.0001
FEMALE	1	-0.1761	0.0280	39.49	<.0001
SUBSTANCE	alcohol	1.1212	0.0339	1092.72	<.0001
SUBSTANCE	cocaine	0.3040	0.0381	63.64	<.0001
SUBSTANCE	heroin	0.0000	0.0000	.	.
AGE	1	0.0132	0.0015	82.52	<.0001
Scale	0	1.0000	0.0000		

NOTE: The scale parameter was held fixed.

```
> poisres = glm(i1 ~ female + substance + age, poisson, data=ds)
> summary(poisres)

Call:
glm(formula = i1 ~ female + substance + age, family = poisson,
     data = ds)

Deviance Residuals:
    Min      1Q  Median      3Q      Max 
-7.57   -3.69   -1.40    1.04   15.99 

Coefficients:
            Estimate Std. Error z value Pr(>|z|)    
(Intercept)  2.89785   0.05827  49.73 < 2e-16  
female       -0.17605   0.02802  -6.28  3.3e-10  
substancecocaine -0.81715   0.02776 -29.43 < 2e-16  
substanceheroin  -1.12117   0.03392 -33.06 < 2e-16  
age          0.01321   0.00145   9.08 < 2e-16  

(Dispersion parameter for poisson family taken to be 1)

Null deviance: 8898.9 on 452 degrees of freedom
Residual deviance: 6713.9 on 448 degrees of freedom
AIC: 8425
```

#### Number of Fisher Scoring iterations: 6

It is always important to check assumptions for models. This is particularly true for Poisson models, which are quite sensitive to model departures. There is support within R for a Pearson's  $\chi^2$  goodness of fit test.

```
> library(vcd)
> poisfit = with(ds, goodfit(e2b, "poisson"))
> summary(poisfit)
```

Goodness-of-fit test for poisson distribution

X<sup>2</sup> df P(> X<sup>2</sup>)

Likelihood Ratio 208 10 3.6e-39

The results indicate that the fit is poor ( $\chi^2_{10} = 208$ ,  $p < 0.0001$ ); the Poisson model does not appear to be tenable. This is also seen in the SAS output, which produces several assessments of goodness of fit by default. The deviance value per degree of freedom is high (14.99).

### 7.10.3 Zero-inflated Poisson regression

A zero-inflated Poisson regression model (7.2.1) might fit better. We'll allow a different probability of extra zeros per level of `female`.

```

options ls=74;
ods select parameterestimates zeroparameterestimates;
proc genmod data=help;
  class substance;
  model i1 = female substance age / dist=zip;
  zeromodel female;
run;

```

In the following output, the confidence limits for the parameter estimates, which appear by default in SAS, have been removed.

#### The GENMOD Procedure

##### Analysis Of Maximum Likelihood Parameter Estimates

Parameter	DF	Estimate	Standard	Wald	Pr > ChiSq	
			Error	Chi-Square		
Intercept	1	2.2970	0.0599	1471.50	<.0001	
FEMALE	1	-0.0680	0.0280	5.89	0.0153	
SUBSTANCE	alcohol	1	0.7609	0.0336	512.52	<.0001
SUBSTANCE	cocaine	1	0.0362	0.0381	0.90	0.3427
SUBSTANCE	heroin	0	0.0000	0.0000	.	.
AGE		1	0.0093	0.0015	39.86	<.0001
Scale	0	1.0000	0.0000			

NOTE: The scale parameter was held fixed.

##### Analysis Of Maximum Likelihood Zero Inflation Parameter Estimates

Parameter	DF	Estimate	Standard	Wald	Pr > ChiSq
			Error	Chi-Square	
Intercept	1	-1.9794	0.1646	144.57	<.0001
FEMALE	1	0.8430	0.2791	9.12	0.0025

> library(pscl)

```

> res = zeroinfl(i1 ~ female + substance + age | female, data=ds)
> res

```

Call:

```
zeroinfl(formula = i1 ~ female + substance + age | female, data = ds)
```

Count model coefficients (poisson with log link):

(Intercept)	female	substancecocaine	substanceheroin
3.05781	-0.06797	-0.72466	-0.76086
age			
0.00927			

Zero-inflation model coefficients (binomial with logit link):

(Intercept)	female
-1.979	0.843

Women are more likely to abstain from alcohol than men: they have more than double the odds of being in the zero-inflation group ( $p=0.0025$ ) and a smaller Poisson mean among

those in the Poisson distribution ( $p=0.015$ ). Other significant predictors include `substance` and `age`, though model assumptions for count models should always be carefully verified [73].

#### 7.10.4 Negative binomial regression

A negative binomial regression model (7.1.7) might also improve on the Poisson.

```
options ls=74;
ods exclude nobs convergencestatus classlevels modelinfo;
proc genmod data=help;
  class substance;
  model i1 = female substance age / dist=negbin;
run;
```

The GENMOD Procedure

##### Criteria For Assessing Goodness Of Fit

Criterion	DF	Value	Value/DF
Deviance	448	539.5973	1.2045
Scaled Deviance	448	539.5973	1.2045
Pearson Chi-Square	448	444.7227	0.9927
Scaled Pearson X2	448	444.7227	0.9927
Log Likelihood		18884.8073	
Full Log Likelihood		-1708.1668	
AIC (smaller is better)		3428.3336	
AICC (smaller is better)		3428.5219	
BIC (smaller is better)		3453.0290	

In the following output, the confidence limits for the parameter estimates, which appear by default in SAS, have been removed.

##### Analysis Of Maximum Likelihood Parameter Estimates

Parameter		DF	Estimate	Standard Error	Wald Chi-Square	Pr > ChiSq
Intercept		1	1.8681	0.2735	46.64	<.0001
FEMALE		1	-0.2689	0.1272	4.47	0.0346
SUBSTANCE	alcohol	1	1.1488	0.1393	68.03	<.0001
SUBSTANCE	cocaine	1	0.3252	0.1400	5.40	0.0202
SUBSTANCE	heroin	0	0.0000	0.0000	.	.
AGE		1	0.0107	0.0075	2.04	0.1527
Dispersion		1	1.2345	0.0897		

NOTE: The negative binomial dispersion parameter was estimated by maximum likelihood.

```

> library(MASS)
> nbres = glm.nb(i1 ~ female + substance + age, data=ds)
> summary(nbres)

Call:
glm.nb(formula = i1 ~ female + substance + age, data = ds,
       init.theta = 0.810015139, link = log)

Deviance Residuals:
    Min      1Q  Median      3Q     Max 
-2.414  -1.032  -0.278   0.241   2.808 

Coefficients:
            Estimate Std. Error z value Pr(>|z|)    
(Intercept)  3.01693   0.28928  10.43   < 2e-16  
female       -0.26887   0.12758  -2.11    0.035    
substancecocaine -0.82360   0.12904  -6.38   1.7e-10  
substanceheroin  -1.14879   0.13882  -8.28   < 2e-16  
age          0.01072   0.00725   1.48    0.139    
                                                        
(Dispersion parameter for Negative Binomial(0.81) family taken to be 1)

Null deviance: 637.82 on 452 degrees of freedom
Residual deviance: 539.60 on 448 degrees of freedom
AIC: 3428

Number of Fisher Scoring iterations: 1

Theta:  0.8100
Std. Err.: 0.0589

2 x log-likelihood: -3416.3340

```

Note that the R and SAS dispersion parameters are inverses of each other.

### 7.10.5 Quantile regression

In this section, we fit a quantile regression model (7.3.1) of the number of drinks (*i1*) as a function of predictors, modeling the 75th percentile (Q3).

```
ods select parameterestimates;
proc quantreg data=help;
  class substance;
  model i1 = female substance age / quantile=0.75;
run;
```

The QUANTREG Procedure

Parameter	DF	Estimate	95% Confidence	
			Limits	
Intercept	1	7.0000	-3.0697	18.5600
FEMALE	1	-2.9091	-7.5765	5.0956
SUBSTANCE alcohol	1	22.6364	16.7650	29.6974
SUBSTANCE cocaine	1	2.5455	-2.6155	10.0198
SUBSTANCE heroin	0	0.0000	0.0000	0.0000
AGE	1	0.1818	-0.2368	0.5867

```
> library(quantreg)
> quantres = rq(i1 ~ female + substance + age, tau=0.75, data=ds)
> summary(quantres)

Call: rq(formula = i1 ~ female + substance + age, tau = 0.75, data = ds)

tau: [1] 0.75

Coefficients:
            coefficients lower bd upper bd
(Intercept)    29.636      19.193     44.392
female        -2.909      -7.116      3.479
substancecocaine -20.091     -28.348    -15.460
substanceheroin   -22.636     -28.256    -19.115
age             0.182      -0.250      0.552

> detach("package:quantreg")
```

Estimating standard errors and confidence limits is nontrivial in these models, and it is thus unsurprising that the default approaches in R and SAS yield different confidence limits.

Because the `quantreg` package overrides needed functionality in other packages, we `detach()` it after running the `rq()` function (see B.4.6).

### 7.10.6 Ordered logistic

To demonstrate an ordinal logit analysis (7.1.4), we first create an ordinal categorical variable from the `sexrisk` variable, then model this three level ordinal variable as a function of `cesd` and `pcs`. Note that SAS and R use opposite coding of the reference group for the intercepts (so the estimates are of opposite sign).

```
data help3;
set help;
  sexriskcat = (sexrisk ge 2) + (sexrisk ge 6);
run;
```

```
ods select parameterestimates;
proc logistic data=help3 descending;
  model sexriskcat = cesd pcs;
run;
```

The LOGISTIC Procedure

#### Analysis of Maximum Likelihood Estimates

Parameter	DF	Estimate	Standard Error	Wald Chi-Square	Pr > ChiSq
Intercept 2	1	-0.9436	0.5607	2.8326	0.0924
Intercept 1	1	1.6697	0.5664	8.6909	0.0032
CESD	1	-0.00004	0.00759	0.0000	0.9963
PCS	1	0.00521	0.00881	0.3499	0.5542

```
> library(MASS)
> ds = transform(ds, sexriskcat =
+   as.factor(as.numeric(sexrisk) >= 2) +
+   as.numeric(sexrisk >= 6)))
> ologit = polr(sexriskcat ~ cesd + pcs, data=ds)
> summary(ologit)
```

Call:

```
polr(formula = sexriskcat ~ cesd + pcs, data = ds)
```

Coefficients:

	Value	Std. Error	t value
cesd	-3.72e-05	0.00761	-0.00489
pcs	5.23e-03	0.00876	0.59649

Intercepts:

	Value	Std. Error	t value
0 1	-1.669	0.562	-2.971
1 2	0.944	0.556	1.698

Residual Deviance: 871.76

AIC: 879.76

### 7.10.7 Generalized logistic model

We can fit a generalized logistic (7.1.5) model for the categorized `sexrisk` variable.

```
options ls=74; /* keep output in grey box */
ods select responseprofile parameterestimates;
proc logistic data=help3 descending;
  model sexriskcat = cesd pcs / link=glogit;
run;
```

## The LOGISTIC Procedure

## Response Profile

Ordered Value	sexriskcat	Total
		Frequency
1	2	151
2	1	244
3	0	58

Logits modeled use sexriskcat=0 as the reference category.

## Analysis of Maximum Likelihood Estimates

Parameter	sexriskcat	DF	Standard	Chi-Square	Pr > ChiSq
			Estimate		
Intercept	2	1	0.6863	0.9477	0.5244
Intercept	1	1	1.4775	0.8943	2.7292
CESD	2	1	-0.00672	0.0132	0.2610
CESD	1	1	-0.0133	0.0125	1.1429
PCS	2	1	0.0105	0.0149	0.4983
PCS	1	1	0.00851	0.0140	0.3670

```
> library(VGAM)
> mlogit = vglm(sexriskcat ~ cesd + pcs,
  family=multinomial(refLevel=1), data=ds)
```

```
> summary(mlogit)

Call:
vglm(formula = sexriskcat ~ cesd + pcs, family = multinomial(refLevel = 1),
      data = ds)

Pearson residuals:
          Min    1Q Median    3Q Max
log(mu[,2]/mu[,1]) -2 -0.6    0.8 0.8   1
log(mu[,3]/mu[,1]) -2 -0.4   -0.4 1.3   1

Coefficients:
            Estimate Std. Error z value
(Intercept):1  1.478     0.89     1.7
(Intercept):2  0.686     0.95     0.7
cesd:1        -0.013     0.01    -1.1
cesd:2        -0.007     0.01    -0.5
pcs:1         0.009     0.01     0.6
pcs:2         0.010     0.01     0.7

Number of linear predictors:  2

Names of linear predictors: log(mu[,2]/mu[,1]), log(mu[,3]/mu[,1])

Dispersion Parameter for multinomial family:  1

Residual deviance: 870 on 900 degrees of freedom

Log-likelihood: -435 on 900 degrees of freedom

Number of iterations: 5

> detach("package:VGAM")
```

Because the VGAM package overrides needed functionality in other packages, we `detach()` it after running the `vglm()` function (see B.4.6).

### 7.10.8 Generalized additive model

We can fit a generalized additive model (7.2.3) and generate a plot in `proc gam` (Figure 7.1).

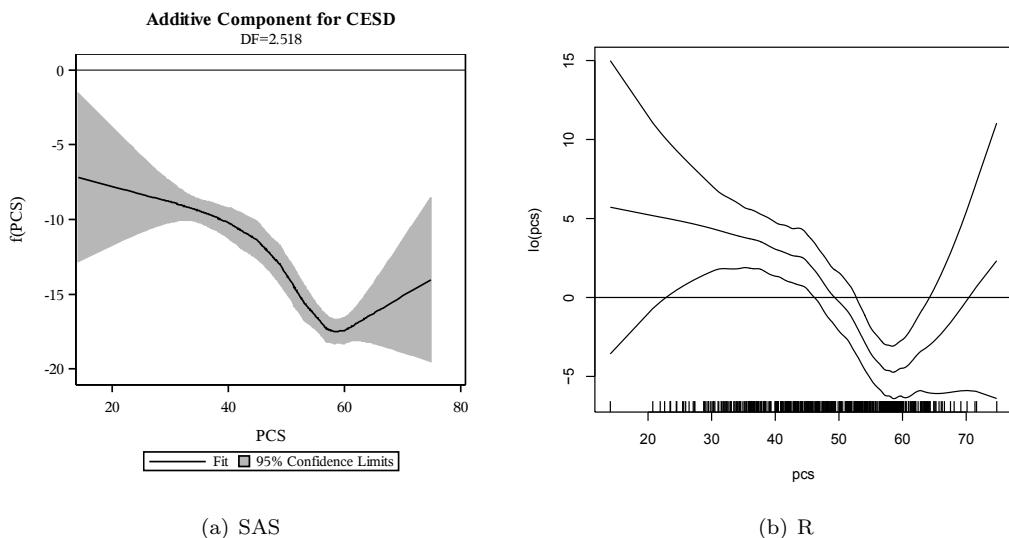


Figure 7.1: Scatterplots of smoothed association of PCS with CESD

```
ods select parameterestimates anodev smoothingcomponentplot;
proc gam data=help plots=components(clm);
  class substance;
  model cesd = param(female) loess(pcs) param(substance) / method=gcv;
run;
```

The GAM Procedure  
 Dependent Variable: CESD  
 Regression Model Component(s): FEMALE SUBSTANCE  
 Smoothing Model Component(s): loess(PCS)

Parameter	Parameter	Standard	t Value	Pr >  t
	Estimate	Error		
Intercept	46.35154	2.70268	17.15	<.0001
FEMALE	4.33966	1.30947	3.31	0.0010
SUBSTANCE alcohol	0.13996	1.36431	0.10	0.9183
SUBSTANCE cocaine	-3.80420	1.43848	-2.64	0.0085
SUBSTANCE heroin	0	.	.	.
Linear(PCS)	-0.27695	0.05275	-5.25	<.0001

Source	DF	Sum of	Chi-Square	Pr > ChiSq
		Squares		
Loess(PCS)	1.51843	1392.676342	10.2850	0.0031

```

> library(gam)
> gamreg= gam(cesd ~ female + lo(pcs) + substance, data=ds)
> summary(gamreg)

Call: gam(formula = cesd ~ female + lo(pcs) + substance, data = ds)
Deviance Residuals:
    Min      1Q  Median      3Q      Max 
-29.158 -8.136  0.811  8.226 29.250 

(Dispersion Parameter for gaussian family taken to be 135)

Null Deviance: 70788 on 452 degrees of freedom
Residual Deviance: 60288 on 445 degrees of freedom
AIC: 3519

Number of Local Scoring Iterations: 2

Anova for Parametric Effects
      Df Sum Sq Mean Sq F value Pr(>F)
female     1   2202    2202   16.2 6.5e-05
lo(pcs)    1   5099    5099   37.6 1.9e-09
substance  2   1437     718    5.3  0.0053
Residuals 445  60288     135

Anova for Nonparametric Effects
      Npar Df Npar F Pr(F)
(Intercept)
female
lo(pcs)       3.1   3.77  0.01
substance

> coefficients(gamreg)

      (Intercept)           female           lo(pcs) substance cocaine
                         46.524            4.339          -0.277        -3.956
      substanceheroin
                         -0.205

```

The `gam` package provides a `plot()` method to demonstrate the results.

```

> plot(gamreg, terms=c("lo(pcs)"), se=2, lwd=3)
> abline(h=0)

```

The estimated smoothing function is displayed in Figure 7.1. Note that the two implementations choose a different referent for the y-axis.

### 7.10.9 Reshaping a dataset for longitudinal regression

A wide (multivariate) dataset can be reshaped (2.3.7) into a tall (longitudinal) dataset. Here we create time-varying variables (with a suffix `tv`) as well as keep baseline values (without the suffix).

In SAS, we do this directly with an `output` statement, putting four lines in the `long` dataset for every line in the original dataset.

```

data long;
set help;
array cesd_a [4] cesd1 - cesd4;
array mcs_a [4] mcs1 - mcs4;
array i1_a [4] i11 - i14;
array g1b_a [4] g1b1 - g1b4;
do time = 1 to 4;
  cesdtv = cesd_a[time];
  mcstv = mcs_a[time];
  i1tv = i1_a[time];
  g1btv = g1b_a[time];
  output;
end;
run;

```

In R we use the `reshape()` command.

```

> long = reshape(ds, idvar="id",
+   varying=list(c("cesd1", "cesd2", "cesd3", "cesd4"),
+   c("mcs1", "mcs2", "mcs3", "mcs4"), c("i11", "i12", "i13", "i14"),
+   c("g1b1", "g1b2", "g1b3", "g1b4")),
+   v.names=c("cesdtv", "mcstv", "i1tv", "g1btv"),
+   timevar="time", times=1:4, direction="long")

```

We can check the resulting dataset by printing tables by time. In the code below, we use some options to `proc freq` to reduce the information provided by default.

```

proc freq data=long;
  tables g1btv*time / nocum norow nopercent;
run;

```

#### The FREQ Procedure

Table of g1btv by time

		Frequency				Total
Col	Pct	1	2	3	4	
0		219	187	225	245	876
		89.02	89.47	91.09	92.11	
1		27	22	22	21	92
		10.98	10.53	8.91	7.89	
Total		246	209	247	266	968

Frequency Missing = 844

```
> table(long$g1btv, long$time)
```

	1	2	3	4
0	219	187	225	245
1	27	22	22	21

We can look at the observations over time for a given individual.

```

proc print data=long;
  where id eq 1;
  var id time cesd cesdtv;
run;

Obs      ID      time      CESD      cesdtv
709      1       1        49        7
710      1       2        49        .
711      1       3        49        8
712      1       4        49        5

> subset(long, id==1, select=c("id", "time", "cesd", "cesdtv"))

  id time cesd cesdtv
1.1  1     1    49      7
1.2  1     2    49     NA
1.3  1     3    49      8
1.4  1     4    49      5

```

This process can be reversed, creating a wide dataset from a tall one. In SAS, we begin by using `proc transpose` to make a row for each variable with the four time points in it.

```

proc transpose data=long out=wide1 prefix=time;
by notsorted id;
var cesdtv mcstv i1tv g1btv;
id time;
run;

```

Note the `notsorted` option to the `by` statement, which allows us to skip an unneeded `proc sort` step and can be used because we know that all the observations for each `id` are stored adjacent to one another. This results in the following data.

```

proc print data=wide1 (obs=6);
run;

Obs      ID      _NAME_      time1      time2      time3      time4
1       2       cesdtv     11.0000      .          .          .
2       2       mcstv      41.7270      .          .          .
3       2       i1tv       8.0000      .          .          .
4       2       g1btv      0.0000      .          .          .
5       8       cesdtv     18.0000      .         25.0000      .
6       8       mcstv      36.0636      .         40.6260      .

```

To put the data for each variable onto one line, we merge the data with itself, taking the lines separately and renaming them along the way using the `where` and `rename` dataset options (A.6.1).

```

data wide (drop=_name_);
merge
wide1 (where = (_name_="cesdtv")
        rename = (time1=cesd1 time2=cesd2 time3=cesd3 time4=cesd4))
wide1 (where = (_name_="mcstv")
        rename = (time1=mcs1 time2=mcs2 time3=mcs3 time4=mcs4))
wide1 (where = (_name_="i1tv")
        rename = (time1=i11 time2=i12 time3=i13 time4=i14))
wide1 (where = (_name_="g1btv")
        rename = (time1=g1b1 time2=g1b2 time3=g1b3 time4=g1b4));
run;

```

The `merge` without a `by` statement simply places the data from sequential lines in each merged dataset next to each other in the new dataset. Since, here, they are different lines from the same dataset, we know that this is correct. In general, the ability to merge without a `by` variable in SAS can cause unintended consequences.

The final dataset is as desired.

```

proc print data=wide (obs=2);
  var id cesd1 - cesd4;
run;

Obs      ID      cesd1      cesd2      cesd3      cesd4
1         2         11          .          .          .
2         8         18          .          25          .

```

This is a cumbersome process, but more straightforward than a pure data step approach.

In contrast, converting to a wide format in R can be done with another call to `reshape()`.

```

> wide = reshape(long,
+   v.names=c("cesdtv", "mcstv", "i1tv", "g1btv"),
+   idvar="id", timevar="time", direction="wide")
> wide[c(2,8), c("id", "cesd", "cesdtv.1", "cesdtv.2", "cesdtv.3",
+   "cesdtv.4")]

      id cesd cesdtv.1 cesdtv.2 cesdtv.3 cesdtv.4
2.1  2   30       11       NA       NA       NA
8.1  8   32       18       NA       25       NA

```

### 7.10.10 Linear model for correlated data

Here we fit a general linear model for correlated data (modeling the covariance matrix directly, 7.4.1).

```

ods select rcorr covparms solutionf tests3;
proc mixed data=long;
  class time;
  model cesdtv = treat time / solution;
  repeated time / subject=id type=un rcorr=7;
run;

```

In this example, the estimated correlation matrix for the seventh subject is printed (this subject was selected because all four time points were observed).

The estimated elements of the variance-covariance matrix are printed row-wise.

Covariance Parameter Estimates						
Cov Parm	Subject	Estimate				
UN(1,1)	ID	207.21				
UN(2,1)	ID	125.11				
UN(2,2)	ID	221.29				
UN(3,1)	ID	131.74				
UN(3,2)	ID	158.39				
UN(3,3)	ID	205.36				
UN(4,1)	ID	97.8055				
UN(4,2)	ID	124.85				
UN(4,3)	ID	151.03				
UN(4,4)	ID	205.75				
Solution for Fixed Effects						
Effect	time	Estimate	Standard			
			Error	DF	t Value	Pr >  t
Intercept		21.2439	1.0709	381	19.84	<.0001
TREAT		-0.4795	1.3196	381	-0.36	0.7165
time	1	2.4140	0.9587	381	2.52	0.0122
time	2	2.6973	0.9150	381	2.95	0.0034
time	3	1.7545	0.6963	381	2.52	0.0121
time	4	0	.	.	.	.
Type 3 Tests of Fixed Effects						
Effect	Num DF		Den DF	F Value	Pr > F	
	DF	DF				
TREAT	1	381		0.13	0.7165	
time	3	381		3.53	0.0150	

```

> library(nlme)
> glsres = gls(cesdtv ~ treat + as.factor(time),
+               correlation=corSymm(form = ~ time | id),
+               weights=varIdent(form = ~ 1 | time),
+               na.action=na.omit, data=long)

```

```
> summary(glsres)

Generalized least squares fit by REML
Model: cesdtv ~ treat + as.factor(time)
Data: long
AIC  BIC logLik
7550 7623 -3760

Correlation Structure: General
Formula: ~time | id
Parameter estimate(s):
Correlation:
  1   2   3
2 0.584
3 0.639 0.743
4 0.474 0.585 0.735
Variance function:
Structure: Different standard deviations per stratum
Formula: ~1 | time
Parameter estimates:
  1   3   4   2
1.000 0.996 0.996 1.033

Coefficients:
            Value Std.Error t-value p-value
(Intercept) 23.66     1.098   21.55  0.000
treat        -0.48     1.320    -0.36  0.716
as.factor(time)2 0.28     0.941    0.30  0.763
as.factor(time)3 -0.66     0.841   -0.78  0.433
as.factor(time)4 -2.41     0.959   -2.52  0.012

Correlation:
          (Intr) treat  as.()2 as.()3
treat      -0.627
as.factor(time)2 -0.395  0.016
as.factor(time)3 -0.433  0.014  0.630
as.factor(time)4 -0.464  0.002  0.536  0.708

Standardized residuals:
      Min     Q1     Med     Q3    Max
-1.643 -0.874 -0.115  0.708  2.582

Residual standard error: 14.4
Degrees of freedom: 969 total; 964 residual

> anova(glsres)

Denom. DF: 964
            numDF F-value p-value
(Intercept)      1     1168 <.0001
treat           1       0  0.6887
as.factor(time)  3       4  0.0145
```

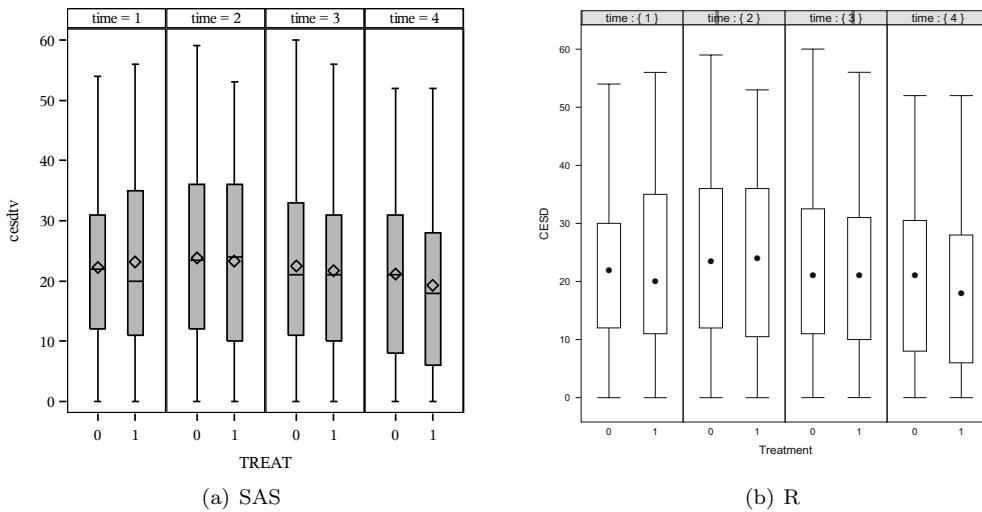


Figure 7.2: Side-by-side box plots of CESD by treatment and time

A set of side-by-side boxplots (8.2.2) by time can be generated using the following commands. Results are displayed in Figure 7.2.

```
proc sgpanel data=long;
  panelby time / columns=4;
  vbox cesdtv / category=treat;
run;

> library(lattice)
> bwplot(cesdtv ~ as.factor(treat) | time, xlab="TREAT",
  strip=strip.custom(strip.names=TRUE, strip.levels=TRUE),
  ylab="CESD", layout=c(4,1), col="black", data=long,
  par.settings=list(box.rectangle=list(col="black"),
  box.dot=list(col="black"), box.umbrella=list(col="black")))
```

### 7.10.11 Linear mixed (random slope) model

Here we fix a mixed effects, or random slope model (7.4.3). Note that in SAS a given variable can be either a `class` variable or not, within a procedure. In this example, we specify a categorical fixed effect of time but a random slope across time treated continuously. We do this by making a copy of the time variable in a new dataset. We save the estimated random effects for later examination, but use `ODS` to suppress their printing.

```
data long2;
set long;
  timecopy=time;
run;
```

To make the first time point the referent, as the R `lme()` function does by default, we first sort by time; then we use the `order=data` option to the `proc mixed` statement.

```

proc sort data= long2; by id descending time; run;

options ls=74;
ods output solutionr=reffs;
ods exclude modelinfo classlevels convergencestatus fitstatistics lrt
      dimensions nobs iterhistory solutionr;
proc mixed data=long2 order=data;
  class timecopy;
  model cesdtv = treat timecopy / solution;
  random int time / subject=id type=un vcorr=20 solution;
run;

```

The Mixed Procedure

#### Estimated V Correlation Matrix for Subject 20

Row	Col1	Col2	Col3	Col4
1	1.0000	0.6878	0.6210	0.5293
2	0.6878	1.0000	0.6694	0.6166
3	0.6210	0.6694	1.0000	0.6813
4	0.5293	0.6166	0.6813	1.0000

The Mixed Procedure

#### Covariance Parameter Estimates

Cov Parm	Subject	Estimate
UN(1,1)	ID	188.43
UN(2,1)	ID	-21.8938
UN(2,2)	ID	9.1731
Residual		61.5856

The Mixed Procedure

#### Solution for Fixed Effects

Effect	timecopy	Estimate	Standard			
			Error	DF	t Value	Pr >  t
Intercept		23.8843	1.1066	381	21.58	<.0001
TREAT		-0.4353	1.3333	292	-0.33	0.7443
timecopy	4	-2.5776	0.9438	292	-2.73	0.0067
timecopy	3	-1.0142	0.8689	292	-1.17	0.2441
timecopy	2	-0.06144	0.8371	292	-0.07	0.9415
timecopy	1	0	.	.	.	.

## The Mixed Procedure

## Type 3 Tests of Fixed Effects

Effect	Num	Den	F Value	Pr > F
	DF	DF		
TREAT	1	292	0.11	0.7443
timecopy	3	292	3.35	0.0195

To examine the predicted random effects, or BLUPs, we can look at the `reffs` dataset created by the ODS output statement and the `solution` option to the `random` statement. This dataset includes a `subject` variable created by SAS from the `subject` option in the `random` statement. It contains the same information as the `id` variable, but is encoded as a character variable and has some blank spaces in it. In order to easily print the predicted random effects for the subject with `id=1`, we condition using the `where` statement (A.6.2), removing the blanks using the `strip` function (2.2.16).

```
proc print data=reffs;
  where strip(subject) eq '1';
run;
```

Obs	Effect	Subject	Estimate	StdErr			
				Pred	DF	tValue	Probt
1	Intercept	1	-13.4805	7.4764	292	-1.80	0.0724
2	time	1	-0.02392	2.3267	292	-0.01	0.9918

We can check the predicted values for an individual (incorporating their predicted random effect) using the `outp` option as well as the marginal predicted mean from the `outpm` option to the `model` statement. Here we suppress all output, then print the observed and predicted values for one subject.

```
ods exclude all;
proc mixed data=long2 order=data;
  class timecopy;
  model cesdtv = treat timecopy / outp=lmp outpm=lmppm;
  random int time / subject=id type=un;
run;
ods select all;
```

The `outp` dataset has the predicted mean conditional on each subject. The `outpm` dataset has the marginal means. If we want to see them in the same dataset, we can merge them (2.3.11). Note that because the input dataset (`long2`) used in `proc mixed` was sorted, the output datasets are also sorted. Otherwise, a `proc sort` step would be needed for each dataset to be merged. Since both the datasets contain a variable `pred`, we rename one of the variables as we merge the datasets.

```

data lmmout;
merge lmmp lmmppm (rename = (pred=margpred));
  by id descending time;
run;

proc print data=lmmout;
  where id eq 1;
  var id time cesdtv pred margpred;
run;

Obs      ID      time      cesdtv      Pred      margpred
1        1        4          5      7.29524    20.8714
2        1        3          8      8.88264    22.4349
3        1        2          .      9.85929    23.3876
4        1        1          7      9.94464    23.4490

```

In R we mimic this process by creating an `as.factor()` version of time. As an alternative, we could nest the call to `as.factor()` within the call to `lme()`.

```

> long = transform(long, tf=as.factor(time))
> library(nlme)
> lmeslope = lme(fixed=cesdtv ~ treat + tf,
+                 random= ~ time | id, na.action=na.omit,
+                 data=long)
> print(lmeslope)

Linear mixed-effects model fit by REML
Data: long
Log-restricted-likelihood: -3772
Fixed: cesdtv ~ treat + tf
(Intercept)      treat          tf2          tf3          tf4
23.8843       -0.4353       -0.0615      -1.0142      -2.5776

Random effects:
Formula: ~time | id
Structure: General positive-definite, Log-Cholesky parametrization
          StdDev Corr
(Intercept) 13.73 (Intr)
time         3.03  -0.527
Residual     7.85

Number of Observations: 969
Number of Groups: 383

> anova(lmeslope)

      numDF denDF F-value p-value
(Intercept)     1    583   1163 <.0001
treat          1    381      0  0.7257
tf             3    583      3  0.0189

```

In R, we use the `random.effects` and `predict()` functions to find the predicted random effects and predicted values, respectively.

```

> reffs = random.effects(lmeslope)
> reffs[1,]

  (Intercept)    time
1           -13.5 -0.024

> predval = predict(lmeslope, newdata=long, level=0:1)
> predval[predval$id==1,]

  id predict.fixed predict.id
1.1 1          23.4      9.94
1.2 1          23.4      9.86
1.3 1          22.4      8.88
1.4 1          20.9      7.30

> vc = VarCorr(lmeslope)
> summary(vc)

  Variance   StdDev       Corr
9.17:1     3.03:1       :1
61.58:1    7.85:1     -0.527:1
188.43:1   13.73:1   (Intr):1

```

The `VarCorr()` function calculates the variances, standard deviations, and correlations between the random effects terms, as well as the within-group error variance and standard deviation.

### 7.10.12 Generalized estimating equations

We fit a GEE model (7.4.8), using an exchangeable working correlation matrix and empirical variance [102]. SAS supports nonmonotone missingness with unstructured working correlations, using the syntax below (results not shown).

```

proc genmod data=long2 descending;
  class timecopy id;
  model g1btv = treat time / dist=bin;
  repeated subject = id / within=timecopy type=un corrw;
run;

```

To show equivalence between the two systems, we fit the exchangeable correlation structure in SAS as well. In this case the `within` option to the `repeated` statement is not needed.

```

ods select geeemppest geewcorr;
proc genmod data=long2 descending;
  class id;
  model g1btv = treat time / dist=bin;
  repeated subject = id / type=exch corrw;
run;

```

## The GENMOD Procedure

 Analysis Of GEE Parameter Estimates  
 Empirical Standard Error Estimates

Parameter Estimate	Standard Error	95% Confidence		Z	Pr >  Z
		Limits			
Intercept	-1.8517	0.2723	-2.3854	-1.3180	-6.80 <.0001
TREAT	-0.0087	0.2683	-0.5347	0.5172	-0.03 0.9740
time	-0.1459	0.0872	-0.3168	0.0250	-1.67 0.0942

The `corrw` option requests the working correlation matrix be printed.

## Working Correlation Matrix

	Col1	Col2	Col3	Col4
Row1	1.0000	0.2994	0.2994	0.2994
Row2	0.2994	1.0000	0.2994	0.2994
Row3	0.2994	0.2994	1.0000	0.2994
Row4	0.2994	0.2994	0.2994	1.0000

```
> library(gee)
> sortlong = long[order(long$id),]
> geeres = gee(formula = g1btv ~ treat + time, id=id, data=sortlong,
+               family=binomial, na.action=na.omit, corstr="exchangeable")

(Intercept)      treat       time
-1.9649      0.0443     -0.1256
```

In addition to returning an object with results, the `gee()` function displays the coefficients from a model assuming that all observations are uncorrelated. This is also the default behavior for `proc genmod`, though we have suppressed printing these estimates here.

```
> coef(geeres)

(Intercept)      treat       time
-1.85169     -0.00874    -0.14593

> sqrt(diag(geeres$robust.variance))

(Intercept)      treat       time
0.2723       0.2683     0.0872

> geeres$working.correlation

 [,1]  [,2]  [,3]  [,4]
[1,] 1.000 0.299 0.299 0.299
[2,] 0.299 1.000 0.299 0.299
[3,] 0.299 0.299 1.000 0.299
[4,] 0.299 0.299 0.299 1.000
```

### 7.10.13 Generalized linear mixed model

Here we fit a GLMM (7.4.7), predicting recent suicidal ideation as a function of treatment, depressive symptoms (CESD), and time. All subjects are assumed to have their own random intercepts.

```
ods select parameterestimates;
proc glimmix data=long;
    model g1btv = treat cesdtv time / dist=bin solution;
    random int / subject=id;
run;
```

The GLIMMIX Procedure

#### Solutions for Fixed Effects

Effect	Standard		DF	t Value	Pr >  t
	Estimate	Error			
Intercept	-4.3572	0.4831	381	-9.02	<.0001
TREAT	-0.00749	0.2821	583	-0.03	0.9788
cesdtv	0.07820	0.01027	583	7.62	<.0001
time	-0.09253	0.1111	583	-0.83	0.4051

For many generalized linear mixed models, the likelihood has an awkward shape, and maximizing it can be difficult. In such cases, it is preferable to use the Laplace approximation to the marginal likelihood, rather than the penalized quasi-likelihood approximation used by default in `proc glimmix`; this can be requested using the `method=laplace` option to the `proc glimmix` statement.

```
ods select parameterestimates;
proc glimmix data=long method=laplace;
    model g1btv = treat cesdtv time / dist=bin solution;
    random int / subject=id;
run;
```

The GLIMMIX Procedure

#### Solutions for Fixed Effects

Effect	Standard		DF	t Value	Pr >  t
	Estimate	Error			
Intercept	-8.7630	1.2217	381	-7.17	<.0001
TREAT	-0.04170	0.6707	583	-0.06	0.9505
cesdtv	0.1018	0.01927	583	5.28	<.0001
time	-0.2426	0.1730	583	-1.40	0.1615

```

> library(lme4)
> glmmres = glmer(g1btv ~ treat + cesdtv + time + (1/id),
+                   family=binomial(link="logit"), data=long)
> summary(glmmres)

Generalized linear mixed model fit by maximum likelihood ['glmerMod']
  Family: binomial ( logit )
Formula: g1btv ~ treat + cesdtv + time + (1 | id)
Data: long

      AIC      BIC    logLik deviance
 480     504     -235      470

Random effects:
 Groups Name        Variance Std.Dev.
 id      (Intercept) 32.6      5.71
 Number of obs: 968, groups: id, 383

Fixed effects:
            Estimate Std. Error z value Pr(>|z|)
(Intercept) -8.7670    1.2808  -6.85  7.6e-12
treat       -0.0347    1.2167  -0.03   0.98
cesdtv       0.1017    0.0237   4.30  1.7e-05
time        -0.2430    0.1838  -1.32   0.19

Correlation of Fixed Effects:
          (Intr) treat  cesdtv
treat     -0.480
cesdtv    -0.641 -0.025
time      -0.365  0.009  0.028

```

### 7.10.14 Cox proportional hazards model

Here we fit a proportional hazards model (7.5.1) for the time to linkage to primary care, with randomization group, age, gender, and CESD as predictors.

```

options ls=74;
ods exclude modelinfo nobs classlevelinfo convergencestatus type3;
proc phreg data=help;
  class treat female;
  model dayslink*linkstatus(0) = treat age female cesd;
run;

```

The PHREG Procedure

Summary of the Number of Event and Censored Values

Total	Event	Censored	Percent
			Censored
431	163	268	62.18

Model Fit Statistics						
Criterion		Without Covariates		With Covariates		
-2 LOG L		1899.982		1805.368		
AIC		1899.982		1813.368		
SBC		1899.982		1825.743		

Testing Global Null Hypothesis: BETA=0						
Test		Chi-Square	DF	Pr > ChiSq		
Likelihood Ratio		94.6132	4	<.0001		
Score		92.3599	4	<.0001		
Wald		76.8717	4	<.0001		

Analysis of Maximum Likelihood Estimates						
Parameter	DF	Parameter Estimate	Standard Error	Chi-Square	Pr > ChiSq	Hazard Ratio
TREAT	0	1	-1.65185	0.19324	73.0737	<.0001
AGE		1	0.02467	0.01032	5.7160	0.0168
FEMALE	0	1	0.32535	0.20379	2.5489	0.1104
CESD		1	0.00235	0.00638	0.1363	0.7120

In R we request the Breslow estimator, for compatibility with SAS (the default is the Efron estimator, which is considered to be a better approximation in the case of many ties).

```
> library(survival)
> survobj = coxph(Surv(dayslink, linkstatus) ~ treat + age + female +
+ cesd, method="breslow", data=ds)
> print(survobj)
```

Call:

```
coxph(formula = Surv(dayslink, linkstatus) ~ treat + age + female +
cesd, data = ds, method = "breslow")
```

	coef	exp(coef)	se(coef)	z	p
treat	1.65186	5.217	0.19324	8.548	0.000
age	0.02467	1.025	0.01032	2.391	0.017
female	-0.32535	0.722	0.20379	-1.597	0.110
cesd	0.00235	1.002	0.00638	0.369	0.710

Likelihood ratio test=94.6 on 4 df, p=0 n= 431, number of events= 163  
(22 observations deleted due to missingness)

Note that the nomiss option is required in SAS to include only observations with all variables observed.

### 7.10.15 Cronbach's $\alpha$

We begin by calculating Cronbach's  $\alpha$  (7.6.1) for the 20 items comprising the CESD (Center for Epidemiologic Studies-Depression scale).

```

ods select cronbachalpha;
proc corr data=help alpha nomiss;
  var f1a -- f1t;
run;
ods exclude none;

The CORR Procedure

Cronbach Coefficient Alpha

Variables          Alpha
-----
Raw              0.760762
Standardized    0.764156

> library(multilevel)
> with(ds, cronbach(cbind(f1a, f1b, f1c, f1d, f1e, f1f, f1g, f1h,
+   f1i, f1j, f1k, f1l, f1m, f1n, f1o, f1p, f1q, f1r,
+   f1s, f1t)))

$Alpha
[1] 0.761

$N
[1] 446

```

The observed  $\alpha$  of 0.76 from the HELP study is relatively low: this may be due to ceiling effects for this sample of subjects recruited in a detoxification unit.

### 7.10.16 Factor analysis

Here we consider a maximum likelihood factor analysis (7.6.2) with varimax rotation for the individual items of the CESD (Center for Epidemiologic Studies–Depression) scale. The individual questions can be found in Table C.2, p. 381. We arbitrarily force three factors. Before beginning, we exclude observations with missing values.

```
ods select orthrotfactpat factor.rotatedsolution.finalcommunwgt;
proc factor data=help nfactors=3 method=ml rotate=varimax;
  var f1a--f1t;
run;
```

The FACTOR Procedure  
Rotation Method: Varimax

#### Rotated Factor Pattern

	Factor1	Factor2	Factor3
F1A	0.44823	-0.19780	0.12436
F1B	0.42744	-0.18496	0.12385
F1C	0.61763	-0.29675	0.21479
F1D	-0.25073	0.45456	-0.15236
F1E	0.51814	-0.11387	0.13228
F1F	0.66562	-0.33478	0.15433
F1G	0.47079	0.03520	0.10880
F1H	-0.07422	0.62158	-0.08435
F1I	0.46243	-0.32461	0.25433
F1J	0.49539	-0.22585	0.27949
F1K	0.52291	-0.11535	0.08873
F1L	-0.27558	0.63987	0.01191
F1M	0.28394	-0.03699	0.19061
F1N	0.48453	-0.33040	0.18281
F1O	0.26188	-0.06977	0.53195
F1P	-0.07338	0.75511	-0.13125
F1Q	0.45736	-0.07107	0.27039
F1R	0.61412	-0.28168	0.27696
F1S	0.23592	-0.16627	0.80228
F1T	0.48914	-0.26872	0.40136

Final Communality Estimates and Variable Weights  
Total Communality: Weighted = 15.332773 Unweighted = 7.811194

Variable	Communality	Weight
F1A	0.25549722	1.34316770
F1B	0.23225517	1.30252990
F1C	0.51565766	2.06467779
F1D	0.29270906	1.41401403
F1E	0.29893385	1.42636367
F1F	0.57894420	2.37499121
F1G	0.23471625	1.30675434
F1H	0.39897919	1.66400037
F1I	0.38389849	1.62312753
F1J	0.37453462	1.59881735
F1K	0.29461104	1.41765736
F1L	0.48551624	1.94346054
F1M	0.11832415	1.13419896
F1N	0.37735132	1.60602564
F1O	0.35641841	1.55382997
F1P	0.59280807	2.45558672
F1Q	0.28734113	1.40315708
F1R	0.53318869	2.14218252
F1S	0.72695038	3.66226205
F1T	0.47255864	1.89596701

```

> res = with(ds, factanal(~ f1a + f1b + f1c + f1d + f1e + f1f + f1g +
+ f1h + f1i + f1j + f1k + f1l + f1m + f1n + f1o + f1p + f1q + f1r +
+ f1s + f1t, factors=3, rotation="varimax", na.action=na.omit,
+ scores="regression"))
> print(res, cutoff=0.45, sort=TRUE)

Call:
factanal(x = ~f1a + f1b + f1c + f1d + f1e + f1f + f1g + f1h +
f1i + f1j + f1k + f1l + f1m + f1n + f1o + f1p + f1q + f1r +
f1s + f1t, factors = 3, na.action = na.omit, scores = "regression",
rotation = "varimax")

Uniquenesses:
   f1a    f1b    f1c    f1d    f1e    f1f    f1g    f1h    f1i    f1j    f1k    f1l
0.745 0.768 0.484 0.707 0.701 0.421 0.765 0.601 0.616 0.625 0.705 0.514
   f1m    f1n    f1o    f1p    f1q    f1r    f1s    f1t
0.882 0.623 0.644 0.407 0.713 0.467 0.273 0.527

Loadings:
          Factor1 Factor2 Factor3
f1c      0.618
f1e      0.518
f1f      0.666
f1k      0.523
f1r      0.614
f1h      -0.621
f1l      -0.640
f1p      -0.755
f1o          0.532
f1s          0.802
f1a
f1b
f1d      -0.454
f1g      0.471
f1i      0.463
f1j      0.495
f1m
f1n      0.485
f1q      0.457
f1t      0.489

          Factor1 Factor2 Factor3
SS loadings     3.847   2.329   1.636
Proportion Var  0.192   0.116   0.082
Cumulative Var 0.192   0.309   0.391

Test of the hypothesis that 3 factors are sufficient.
The chi square statistic is 289 on 133 degrees of freedom.
The p-value is 1.56e-13

```

Next, we interpret the item scores from the output. We see that the second factor loads on the reverse coded items (H, L, P, and D, see 2.6.3). Factor 3 loads on items O and S (*people were unfriendly* and *I felt that people dislike me*).

### 7.10.17 Recursive partitioning

In this example, we use recursive partitioning (section 7.6.3) to classify subjects based on their homeless status, using gender, drinking, primary substance, RAB sexrisk, MCS, and PCS as predictors.

```
> library(rpart)
> ds = transform(ds, sub = as.factor(substance))
> homeless.rpart = rpart(homeless ~ female + i1 + sub + sexrisk + mcs +
  pcs, method="class", data=ds)
> printcp(homeless.rpart)

Classification tree:
rpart(formula = homeless ~ female + i1 + sub + sexrisk + mcs +
  pcs, data = ds, method = "class")

Variables actually used in tree construction:
[1] female   i1       mcs      pcs      sexrisk

Root node error: 209/453 = 0.5

n= 453

      CP nsplit rel error xerror xstd
1 0.10      0     1.0    1.0  0.05
2 0.05      1     0.9    1.0  0.05
3 0.03      4     0.8    1.0  0.05
4 0.02      5     0.7    1.0  0.05
5 0.01      7     0.7    0.9  0.05
6 0.01      9     0.7    0.9  0.05

> library(partykit)
> plot(as.party(homeless.rpart))
```

Figure 7.3 displays the tree. To help interpret this model, we can explore the empirical proportion of subjects that were homeless among those with  $i1 < 3.5$  by `pcs` less than 31.94. This corresponds to Nodes 18 and 19 in the figure.

```
> home = with(ds, homeless[i1<3.5])
> pcslow = with(ds, pcs[i1<3.5]<=31.94)
> table(home, pcslow)

pcslow
home FALSE TRUE
  0     89     2
  1     31     5

> rm(home, pcslow)
```

Among this subset, 71.4% (5 of 7) of those with low PCS scores are homeless, while only 25.8% (31 of 120) of those with PCS scores above the threshold are homeless.

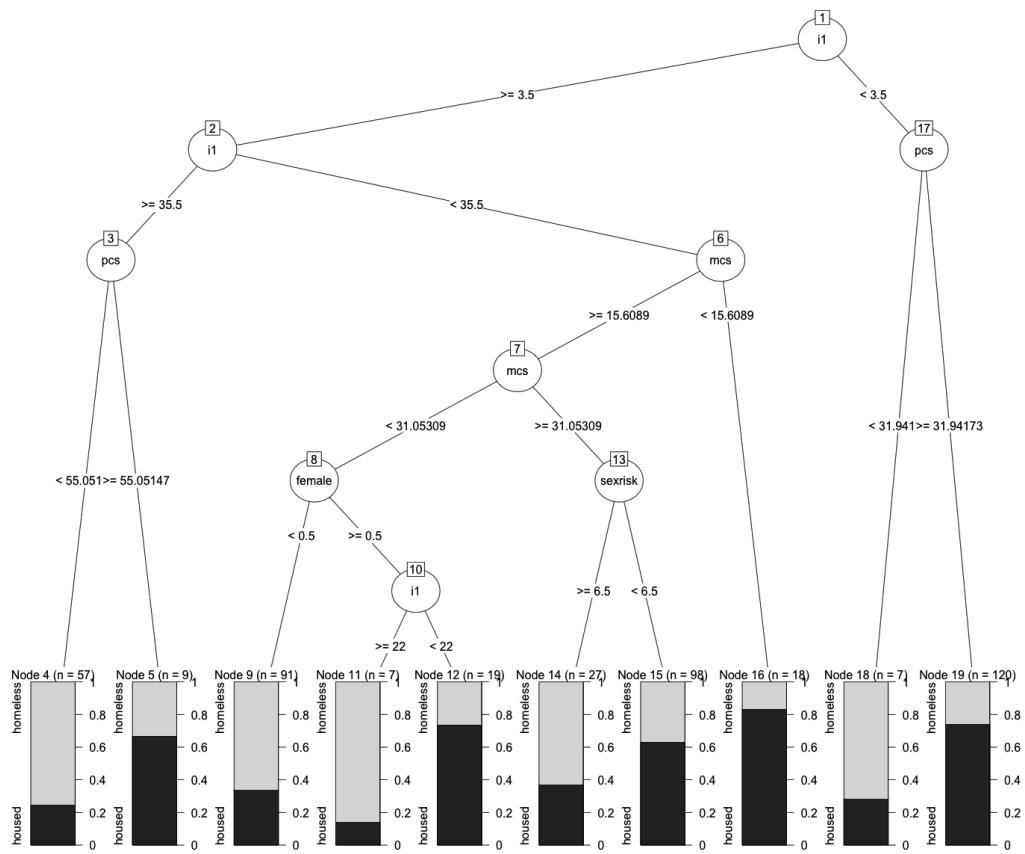


Figure 7.3: Recursive partitioning tree from R

### 7.10.18 Linear discriminant analysis

We use linear discriminant analysis (7.6.4) to distinguish between homeless and nonhomeless subjects, with a prior classification that half are in each group (default in SAS).

```
ods select lineardiscfunc classifiedresub errorresub;
proc discrim data=help out=ldaout;
  class homeless;
  var age cesd mcs pcs;
run;
```

The DISCRIM Procedure

#### Linear Discriminant Function for HOMELESS

Variable	0	1
Constant	-56.61467	-56.81613
AGE	0.76638	0.78563
CESD	0.86492	0.87231
MCS	0.68105	0.67569
PCS	0.74918	0.73750

The DISCRIM Procedure

Classification Summary for Calibration Data: WORK.HELP

Resubstitution Summary using Linear Discriminant Function

#### Number of Observations and Percent Classified into HOMELESS

From	Total		
HOMELESS	0	1	Total
0	142 58.20	102 41.80	244 100.00
1	89 42.58	120 57.42	209 100.00
Total	231 50.99	222 49.01	453 100.00
Priors	0.5	0.5	

#### Error Count Estimates for HOMELESS

	0	1	Total
Rate	0.4180	0.4258	0.4219
Priors	0.5000	0.5000	

```

> library(MASS)
> ngroups = length(unique(ds$homeless))
> ldamodel = with(ds, lda(homeless ~ age + cesd + mcs + pcs,
+ prior=rep(1/ngroups, ngroups)))
> print(ldamodel)

Call:
lda(homeless ~ age + cesd + mcs + pcs, prior = rep(1/ngroups,
ngroups))

Prior probabilities of groups:
 0   1
0.5 0.5

Group means:
  age cesd mcs  pcs
0 35.0 31.8 32.5 49.0
1 36.4 34.0 30.7 46.9

Coefficients of linear discriminants:
LD1
age  0.0702
cesd  0.0269
mcs -0.0195
pcs -0.0426

axis1 label="Prob(homeless eq 1)";

ods select "Histogram 1";
proc univariate data=ldaout;
  class homeless;
  var _1;
  histogram _1 / nmidpoints=20 haxis=axis1;
run;

> plot(ldamodel)

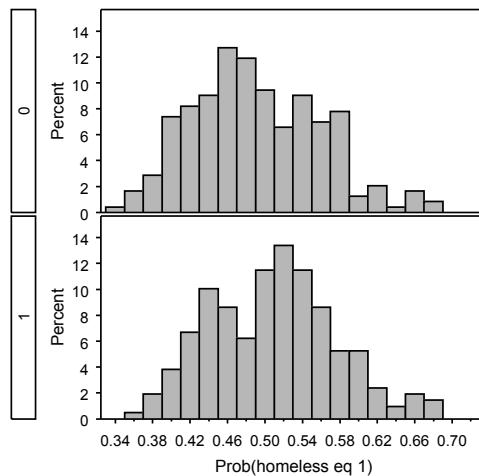
```

Details on the display of `lda` objects can be found using `help(plot.lda)`.

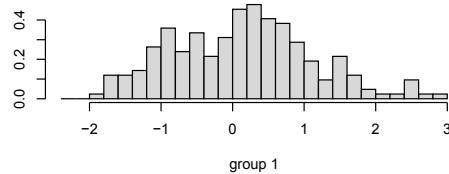
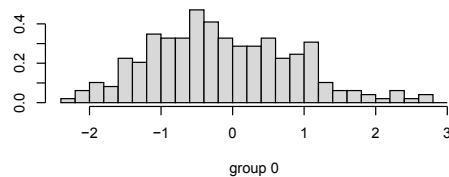
The results indicate that homeless subjects tend to be older, have higher CESD scores, and lower MCS and PCS scores. Figure 7.4 displays the distribution of linear discriminant function values by homeless status; the discrimination ability appears to be slight. The distribution of the linear discriminant function values is shifted to the right for the homeless subjects, though there is considerable overlap between the groups.

### 7.10.19 Hierarchical clustering

In this example, we use hierarchical clustering (7.6.6) to group continuous variables from the HELP dataset.



(a) SAS



(b) R

Figure 7.4: Graphical display of assignment probabilities or score functions from linear discriminant analysis by actual homeless status

```

ods exclude all;
proc varclus data=help outtree=treedisp centroid;
  var mcs pcs cesd i1 sexrisk;
run;
ods exclude none;

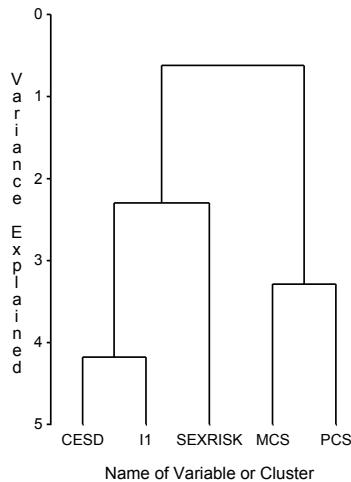
proc tree data=treedisp nclusters=5;
  height _varexp_;
run;

```

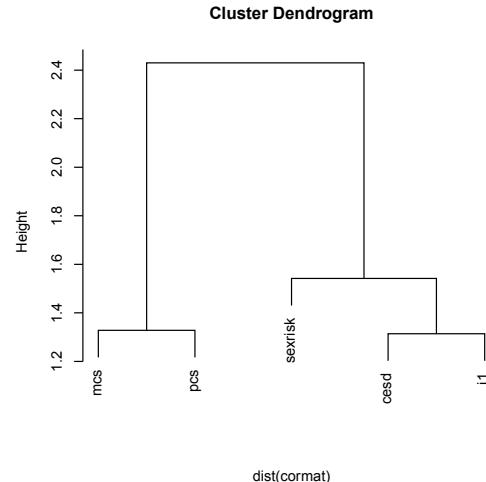
```
> cormat = with(ds, cor(cbind(mcs, pcs, cesd, i1, sexrisk),
  use="pairwise.complete.obs"))
> hclustobj = hclust(dist(cormat))
```

Figure 7.5 displays the clustering. Not surprisingly, the MCS and PCS variables cluster together, since they both utilize similar questions and structures. The CESD and I1 variables cluster together, while there is a separate node for SEXRISK.

```
> plot(hclustobj)
```



(a) SAS



(b) R

Figure 7.5: Results from hierarchical clustering

# Chapter 8

## A graphical compendium

This chapter provides a compendium of graphical displays. More details about configuration options can be found in Chapter 9. Because graphics are useful to visualize analyses, examples appear throughout the book.

Producing graphics for data analysis is simple and direct in both programs. Producing graphics for publication is more complex and typically requires a great deal of time to achieve the desired appearance. Our intent is to provide sufficient guidance that most effects can be achieved, but further investigation of the documentation and experimentation will doubtless be necessary for specific needs. There are a huge number of options: we aim to provide a roadmap as well as examples to illustrate the power of the package.

While many graphics in R can be generated using one command, figures are often built up element by element. For example, an empty box can be created with a specific set of x and y axis labels and tick marks, then points can be added with different printing characters. Text annotations can then be added, along with legends and other additional information (see 6.6.1). The CRAN graphics task view (<http://cran.r-project.org/web/views>) provides a comprehensive listing of functionality to create graphics in R.

A somewhat intimidating number of options are available, some of which can be specified using the `par()` graphics parameters (see 9.2), while others can be given as options to plotting commands (such as `plot()` or `lines()`).

R provides a number of graphics devices to support different platforms and formats. The default varies by platform (`Windows()` under Windows, `X11()` under Linux, and `quartz()` under modern Mac OS X distributions). A device is created automatically when a plotting command is run, or a device can be started in advance to create a file in a particular format (e.g., the `pdf()` device).

A series of powerful add-on packages to create sophisticated graphics is available within R. These include the `grid` package [123], the `lattice` package [151], and the `ggplot2` package [196]. Running `example()` for a specified function of interest is often helpful for commands shown in this chapter, as is `demo(graphics)`.

### 8.1 Univariate plots

#### 8.1.1 Barplot

While not typically an efficient graphical display, there are times when a barplot is appropriate to display counts by groups.

**SAS**

```
proc gchart data=ds;
  hbar x1 / sumvar=x2 type=mean;
run; quit;
or
proc sgplot data=ds;
  hbar x1 / response=x2 stat=mean;
run;
```

*Note:* The above code produces one bar for each level of  $X_1$  with the length determined by the mean of  $X_2$  in each level. Without the `type=mean` or `stat=mean` option, the length would be the sum of  $x_2$  in each level. With no options, the length of each bar is measured in the number of observations in each level of  $X_1$ . The `hbar` statement can be replaced by the `vbar` statement (with identical syntax) to make vertical bars, while the `hbar3d` and `vbar3d` (in `proc gchart` only) make bars with a three-dimensional appearance. Options in `proc gchart` allow display of reference lines, display of statistics, grouping by an additional variable, and many other possibilities. The `sgplot` procedure can also produce similar dot plots using the `dot` statement.

**R**

```
barplot(table(x1, x2), legend=c("grp1", "grp2"), xlab="X2")
or
library(lattice)
barchart(table(x1, x2, x3))
```

*Note:* The input for the `barplot()` function is given as the output of a one- or two-dimensional contingency table, while the `barchart()` function within the `lattice` package supports three-dimensional tables (see `example(barplot)` and `example(barchart)`). A similar `dotchart()` function produces a horizontal slot for each group with a dot reflecting the frequency.

### 8.1.2 Stem-and-leaf plot

*Example:* 6.6.3

Stem-and-leaf plots are text-based graphics that are particularly useful to describe the distribution of small datasets. They are often used for teaching purposes.

**SAS**

```
proc univariate plot data=ds;
  var x;
run;
```

*Note:* The stem-and-leaf plot display in SAS is accompanied by a box plot; the `plot` option also generates a text-based normal Q-Q plot. To produce only these plots, use an `ods select plots` statement before the `proc univariate` statement.

**R**

```
stem(x)
```

*Note:* The `scale` option can be used to increase or decrease the number of stems (default value is 1).

### 8.1.3 Dotplot

Dotplots (also called Wilkinson dotplots) are a simple introductory graphic useful for presenting numerical data when the data set is small [200]. They are often used for teaching purposes, as they help to identify clusters and gaps while conserving numerical information (in the same manner as a stem-and-leaf plot).

We are unaware of a procedure in SAS to replicate this. An approximation in SAS might be the stem-and-leaf plot (8.1.2). Alternatively, a macro by Holger Diedrich and Ulrike Grömping can be downloaded from [www.amherst.edu/~nhorton/sasr2/examples/dotplot.sas](http://www.amherst.edu/~nhorton/sasr2/examples/dotplot.sas).

**SAS**

```
%include "dotplot.sas";

%dotplot(x1, dat=ds);
```

*Note:* The dataset containing the variable `x1` is specified with the `dat` option. The macro file contains descriptions of other optional settings.

**R**

```
library(mosaic)
dotPlot(x)
```

### 8.1.4 Histogram

*Example:* 5.7.1

The example in 5.7.1 demonstrates how to annotate a histogram with an overlaid normal or kernel density estimate. Similar estimates are available for all other densities supported within R (see Table 3.1) and for the beta, exponential, gamma, lognormal, Weibull, and other densities within SAS.

**SAS**

```
proc univariate data=ds;
  histogram x1 ... xk;
run;
```

*Note:* The `sgplot` and `sgpanel` procedures also generate histograms, but allow fewer options.

**R**

```
hist(x)
```

*Note:* The default behavior for a histogram is to display frequencies on the vertical axis; probability densities can be displayed using the `freq=FALSE` option. The default title is given by `paste("Histogram of" , x)` where `x` is the name of the variable being plotted; this can be changed with the `main` option. The `histogram()` function in the `lattice` package provides an alternative implementation.

### 8.1.5 Density plots

Density plots are nonparametric estimates of the empirical probability density function (see also 8.2.3, overlaid density plots, and 8.3.5, bivariate density plots).

**SAS**

```
proc kde data=ds;
  univar x / plots=(density histdensity);
run;
```

or

```
proc univariate data=ds;
  histogram x / kernel;
run;
```

*Note:* The `proc univariate` code generates a graphic (as in 5.1), but omits details provided in the latter approach.

**R**

```
plot(density(x))
or
library(lattice)
densityplot(~ x2, data=ds)
```

*Note:* The help page for `density()` provides guidance for specifying the default window width and other options for `densityplot()`.

### 8.1.6 Empirical cumulative probability density plot

**SAS**

```
proc univariate data=ds;
  var x;
  cdfplot x;
run;
```

*Note:* The empirical density plot offered in `proc univariate` is not smoothed, but theoretical distributions can be superimposed as in the histogram plotted in 5.7.1 and using similar syntax. If a smoothed version is required, it may be necessary to estimate the PDF with `proc kde` and save the output (as shown in 5.7.1), then use it to find the corresponding CDF.

**R**

```
plot(ecdf(x))
```

*Note:* The `knots()` function can be used to determine when the empirical density function jumps.

### 8.1.7 Boxplot

See also 8.2.2 (side-by-side boxplots)

*Examples:* 6.6.5 and 7.10.10

**SAS**

```
data ds2;
set ds;
  int=1;
run;

proc boxplot data=ds;
  plot x * int;
run;
or
proc sgplot data=ds;
  vbox x;
run;
```

*Note:* The `boxplot` procedure is designed to produce side-by-side boxplots (8.2.2). To generate a single boxplot with this procedure, create a variable with the same value for all observations, as above, and make a side-by-side boxplot based on that variable. The `sgplot` procedure also allows the `hbox` statement, which produces a horizontal boxplot.

**R**

```
boxplot(x)
```

*Note:* The `boxplot()` function allows sideways orientation using the `horizontal=TRUE` option. The `lattice` package provides an alternative implementation using the `bwplot()` function.

### 8.1.8 Violin plots

Violin plots combine a boxplot and (doubled) kernel density plot. As far as we know, there are no procedures that generate violin plots in SAS. However, a similar notion would be to stack kernel density estimates.

#### SAS

```
proc sgpanel data=ds;
  panelby x1 / columns=1 ;
  density x2 / scale=percent type=kernel ;
run;
```

*Note:* It is assumed that the variable `x1` has relatively few levels.

#### R

```
library(vioplot)
vioplot(x2[x1==0], x2[x1==1])
```

*Note:* Here we assume that `x1` has two levels (0 and 1).

## 8.2 Univariate plots by grouping variable

### 8.2.1 Side-by-side histograms

#### SAS

```
proc sgpanel data=ds;
  panelby x1;
  histogram x2;
run;
```

*Note:* It is assumed that the variable `x1` has relatively few levels.

#### R

```
library(lattice)
histogram(~ x2 | x1)
```

### 8.2.2 Side-by-side boxplots

See also 8.1.7 (boxplots).

#### SAS

```
proc boxplot data=ds;
  plot y * x;
run;
or
proc boxplot data=ds;
  plot (y1 ... yk) * x (z1 ... zp);
run;
or
proc sgplot data=ds;
  vbox x / category=y;
run;
```

*Note:* The first, basic `proc boxplot` code generates a box describing  $Y$  for each level of  $X$ . The second, more general `proc boxplot` code generates a box for each of  $Y_1, Y_2, \dots, Y_k$  for

each level of  $X$ , further grouped by  $Z_1, Z_2, \dots, Z_p$ . The example in Figure 6.6 demonstrates customization.

The `proc sgplot` code results in boxes of  $x$  for each value of  $y$ ; the similar `hbox` statement makes horizontal boxplots. The `sgpanel` procedure can produce multiple side-by-side boxplots in one graphic using `vbox` or `hbox` statements similar to those shown for `proc sgplot`.

#### R

```
boxplot(y[x==0], y[x==1], y[x==2], names=c("X=0", "X=1", "X=2"))
or
library(lattice)
bwplot(y ~ x)
```

*Note:* The `boxplot()` function can be given multiple arguments of vectors to display or can use a formula interface (which will generate a boxplot for each level of the variable  $x$ ). A number of useful options are available, including `varwidth` to draw the boxplots with widths proportional to the square root of the number of observations in that group, `horizontal` to reverse the default orientation, `notch` to display notched boxplots, and `names` to specify a vector of labels for the groups. Boxplots can also be created using the `bwplot()` function in the `lattice` package.

### 8.2.3 Overlaid density plots

See also 8.1.5 (density plots) and 8.3.5 (bivariate density).

#### SAS

```
proc kde data=ds;
  univar x1 x2 / plots=densityoverlay;
run;
```

*Note:* This code shows the empirical densities for two variables.

#### R

```
library(lattice)
densityplot(~ x2, groups=x1, auto.key=TRUE)
```

*Note:* This code shows the density for a single variable plotted for each level of a second variable.

### 8.2.4 Bar chart with error bars

While the graphical display with a bar the height of which indicates the mean and vertical lines indicating the standard error is quite common, many find these displays troubling. We concur with graphics authorities such as Edward Tufte [179], who discourage their use, as does Frank Harrell's group at Vanderbilt (see [biostat.mc.vanderbilt.edu/wiki/Main/StatisticalPolicy](http://biostat.mc.vanderbilt.edu/wiki/Main/StatisticalPolicy)).

#### SAS

```
proc gchart data=ds;
  vbar x1 / group=x2 type=mean sumvar=x3 errorbar=top;
run;
```

*Note:* The code produces a bar for each level of  $x_1$  and  $x_2$ . The bar height will be the mean of  $x_3$  in that level of  $x_1$  and  $x_2$ .

**R**

```

library(lattice)
library(grid)
dynamitePlot = function(height, error,
  names=as.character(1:length(height)),
  significance=NA, ylim=c(0, maxLim), ...)
{
  if (missing(error)) { error = 0 }
  maxLim = 1.2 * max(mapply(sum, height, error))
  mError = min(c(error, na.rm=TRUE))
  barchart(height ~ names, ylim=ylim, panel=function(x,y,...) {
    panel.barchart(x, y, ...)
    grid.polyline(c(x,x), c(y, y+error), id=rep(x,2),
      default.units='native',
      arrow=arrow(angle=45, length=unit(mError, 'native')))
    grid.polyline(c(x,x), c(y, y-error), id=rep(x,2),
      default.units='native',
      arrow=arrow(angle=45, length=unit(mError, 'native')))
    grid.text(x=x, y=y + error + .05*maxLim, label=significance,
      default.units='native')
  }, ...)
}

```

*Note:* This graph is built up in parts using customized calls to the `barchart()` function. Much of the code (due to Randall Pruim) involves setting up the appropriate axis limits, as a function of the heights and error ranges, then drawing the lines adding the text using calls to `grid.polyline()` and `grid.text()`. The ... option to the function passes any additional arguments to the `panel.barchart()` function, to allow further customization (see the book by Sarkar [151]). Once defined, the function can be run using the following syntax.

```

Values = c(1, 2, 5, 4)
Errors = c(0.25, 0.5, 0.33, 0.12)
Names = paste("Trial", 1:4)
Sig = c("a", "a", "b", "b")
dynamitePlot(Values, Errors, names=Names, significance=Sig)

```

The `bgraph.CI()` function within the `sciplot` package provides similar functionality.

## 8.3 Bivariate plots

### 8.3.1 Scatterplot

*Example: 6.6.1*

See 8.3.2 (scatterplot with multiple y values) and 8.4.1 (matrix of scatterplots).

**SAS**

```

proc gplot data=ds;
  plot y*x;
run; quit;
or
proc sgscatter data=ds;
  plot y*x;
run;

```

or

```
proc sgplot data=ds;
  scatter y=y x=x;
run;
```

*Note:* The **gplot** procedure allows the finest control. The **sgplot** procedure allows adding features easily, as in 12.4.3. The **sgscatter** procedure is particularly useful for scatterplot matrices (8.4.1).

## R

```
plot(x, y)
```

*Note:* Many objects within R have default plotting functions (e.g., for a linear model object, **plot.lm()** is called). More information can be found using **methods(plot)**. Specifying **type="n"** causes nothing to be plotted (but sets up axes and draws boxes, see 3.4.1). This technique is often useful if a plot is built up part by part.

### 8.3.2 Scatterplot with multiple y values

See also 8.4.1 (matrix of scatterplots).

*Example:* 8.7.1

## SAS

```
proc gplot data=ds; /* create 1 plot with a single y axis */
  plot (y1 ... yk)*x / overlay;
run; quit;
```

or

```
proc gplot data=ds; /* create 1 plot with 2 separate y axes */
  plot y1*x;
  plot2 y2*x;
run; quit;
```

*Note:* The first code generates a single graphic with all the different  $Y$  values plotted. In this case a simple legend can be added with the **legend** option to the **plot** statement, e.g., **plot (y1 y2)\*x / overlay legend**. A fully controllable legend can be added with a **legend** statement as in Figure 3.1.

The second code generates a single graphic with two y axes. The scale for  $Y_1$  appears on the left and for  $Y_2$  on the right.

In either case, the **symbol** statements (see entries in 9.1) can be used to control the plotted values and add interpolated lines as in 8.7.1. SAS will plot each  $Y$  value in a different color and/or symbol by default. The **overlay** option and **plot2** statements are not mutually exclusive, so that several variables can be plotted on each  $Y$  axis scale.

Using the statement **plot (y1 ... yk)\*x** without the **overlay** option will create  $k$  separate plots, identical to  $k$  separate **proc gplot** procedures. Adding the **uniform** option to the **proc gplot** statement will create  $k$  plots with a common y axis scale.

## R

```
plot(x, y1, pch=pchval1)  # create 1 plot with single y-axis
points(x, y2, pch=pchval2)
...
points(x, yk, pch=pchvalk)
```

or

```
# create 1 plot with 2 separate y axes
addsecondy = function(x, y, origy, yname="Y2") {
  prevlimits = range(origy)
  axislimits = range(y)
  axis(side=4, at=prevlimits[1] + diff(prevlimits)*c(0:5)/5,
       labels=round(axislimits[1] + diff(axislimits)*c(0:5)/5, 1))
  mtext(yname, side=4)
  newy = (y-axislimits[1])/(diff(axislimits)/diff(prevlimits)) +
    prevlimits[1]
  points(x, newy, pch=2)
}
plottwoy = function(x, y1, y2, xname="X", y1name="Y1", y2name="Y2")
{
  plot(x, y1, ylab=y1name, xlab=xname)
  addsecondy(x, y2, y1, yname=y2name)
}
plottwoy(x, y1, y2, y1name="Y1", y2name="Y2")
```

*Note:* To create a figure with a single y axis value, it is straightforward to repeatedly call `points()` or other functions to add elements.

In the second example, two functions `addsecondy()` and `plottwoy()` are defined to add points on a new scale and an appropriate axis on the right. This involves rescaling and labeling the second axis (`side=4`) with 6 tick marks, as well as rescaling the `y2` variable.

### 8.3.3 Scatterplot with binning

When there are many observations, scatterplots can become difficult to read because many plot symbols will obscure one another. One solution to this problem is to use binned scatterplots. Other options are transparent plot symbols that display as darker areas when overlaid (8.3.4) and bivariate density plotting (8.1.5).

We're not aware of a SAS procedure which can do this easily. The `twodhist` macro found at [www.amherst.edu/~nhorton/sasr2/examples](http://www.amherst.edu/~nhorton/sasr2/examples) provides a crude version.

#### SAS

```
%include "twodhist.sas";

%twodhist(data=ds, x=x1, y=x2, nbinsx=30, nbinsy=30, nshades=9);
```

*Note:* The macro accepts an x and y variable and divides each axis into the specified number of bins.

#### R

```
library(hexbin)
plot(hexbin(x, y))
```

### 8.3.4 Transparent overplotting scatterplot

When there are many observations, scatterplots can become difficult to read because many plot symbols will obscure one another. One solution to this problem is to use transparent plot symbols that display as darker areas when overlaid. Other options are binned scatterplots (8.3.3) and bivariate density plotting (8.1.5).

**SAS**

```
proc sgplot data=ds;
    scatter x=x1 y=x2 / markerattrs=(symbol=CircleFilled size=.05in)
        transparency=0.85;
run;
```

**R**

```
plot(x1, x2, pch=19, col="#00000022", cex=0.1)
```

### 8.3.5 Bivariate density plot

**SAS**

```
proc kde data=ds; /* bivariate density */
    bivar x1 x2 / plots=contour;
run;
```

*Note:* The `kde` procedure includes kernel density estimation using a normal kernel. The `bivar` statement for `proc kde` will generate a joint empirical density estimate. The bandwidth can be controlled with the `bwm` option and the number of grid points by the `ngrid` option to the `univar` or `bivar` statements.

**R**

```
smoothScatter(x, y)
or
library(GenKern)
# bivariate density
op = KernSur(x, y, na.rm=TRUE)
image(op$xords, op$yords, op$zden, col=gray.colors(100), axes=TRUE,
      xlab="x var", ylab="y var")
```

*Note:* The `smoothScatter()` function provides a simple interface for a bivariate density plot. The default smoother for `KernSur()` can be specified using the `kernel` option (possible values include the default gaussian, rectangular, triangular, epanechnikov, biweight, cosine, or optcosine). Bivariate density support is provided with the `GenKern` package. Any of the three-dimensional plotting routines (see 8.4.4) can be used to visualize the results.

### 8.3.6 Scatterplot with marginal histograms

*Example: 8.7.3*

This is not easily done with built-in procedures in SAS. We present a solution below using code provided by SAS Institute, which is downloadable from [www.amherst.edu/~nhorton/sasr2/examples/scatterhist.sas](http://www.amherst.edu/~nhorton/sasr2/examples/scatterhist.sas). The code uses syntax not intended for casual users. See <http://tinyurl.com/sasrblog-margscat> for more comments. In the entry below, we assume the code has been downloaded and saved in the file `scatterhist.sas`.

**SAS**

```
%include "dir_location/scatterhist.sas";

proc sgrender data=ds template=scatterhist;
    dynamic yvar="y" xvar="x"
    title="MCS-PCS Relationship";
run;
```

*Note:* The `template` option reads in a template written by the code referenced above. The names of the variables to be plotted must be included in quotes.

**R**

```

scatterhist = function(x, y, xlab="x label", ylab="y label"){
  zones=matrix(c(2,0,1,3), ncol=2, byrow=TRUE)
  layout(zones, widths=c(4/5,1/5), heights=c(1/5,4/5))
  xhist = hist(x, plot=FALSE)
  yhist = hist(y, plot=FALSE)
  top = max(c(xhist$counts, yhist$counts))
  par(mar=c(3,3,1,1))
  plot(x,y)
  par(mar=c(0,3,1,1))
  barplot(xhist$counts, axes=FALSE, ylim=c(0, top), space=0)
  par(mar=c(3,0,1,1))
  barplot(yhist$counts, axes=FALSE, xlim=c(0, top), space=0, horiz=TRUE)
  par(oma=c(3,3,0,0))
  mtext(xlab, side=1, line=1, outer=TRUE, adj=0,
        at=.8 * (mean(x) - min(x))/(max(x)-min(x)))
  mtext(ylab, side=2, line=1, outer=TRUE, adj=0,
        at=(.8 * (mean(y) - min(y))/(max(y) - min(y))))
}
scatterhist(x, y)

```

## 8.4 Multivariate plots

### 8.4.1 Matrix of scatterplots

**SAS**

```

proc sgscatter data=ds;
  matrix x1 ... xk;
run;

```

*Example: 8.7.6*

*Note:* The `diagonal` option to the `matrix` statement allows the diagonal cells to show, for example, histograms with empirical density estimates. A similar effect can be produced with `proc sgpanel`, as demonstrated in 8.7.6.

**R**

```

pairs(data.frame(x1, ..., xk))

```

*Note:* The `pairs()` function is quite flexible, since it calls user specified functions to determine what to display on the lower triangle, diagonal, and upper triangle (see `examples(pairs)` for illustration of its capabilities). The `ggpairs()` function in the `GGally` package can be used to create a pairs plot with both continuous and categorical variables.

### 8.4.2 Conditioning plot

A conditioning plot is used to display a scatter plot for each level of one or two classification variables, as below.

**SAS**

```

proc sgpanel data=ds;
  panelby x2 x3;
  scatter x=x1 y=y;
run;

```

*Example: 8.7.2*

*Note:* A similar plot can be generated with a boxplot, histogram, or other contents in each cell of  $X_2 * X_3$  using other `sgplot` statements in place of the `scatter` statement.

**R**

```
coplot(y ~ x1 | x2*x3)
```

*Note:* The `coplot()` function displays plots of `y` and `x1`, stratified by `x2` and `x3`. All variables may be either numeric or factors.

### 8.4.3 Contour plots

A contour plot shows the Cartesian plain with similarly valued points linked by lines. The most familiar versions of such plots may be contour maps showing lines of constant elevations that are very useful for hiking.

**SAS**

```
proc gcontour data=ds;
  plot y * x = z;
run;
```

**R**

```
contour(x, y, z)
```

or

```
filled.contour(x, y, z)
```

*Note:* The `contour()` function displays a standard contour plot. The `filled.contour()` function creates a contour plot with colored areas between the contours.

### 8.4.4 3-D plots

Perspective or surface plots and needle plots can be used to visualize data in three dimensions. These are particularly useful when a response is observed over a grid of two-dimensional values.

**SAS**

```
proc g3d data=ds;
  scatter x*y=z;
run;
```

```
proc g3d data=ds;
  plot x*y=z;
run;
```

```
proc gcontour data=ds;
  plot x*y=z;
run;
```

*Note:* The `scatter` statement produces a needle plot, a 3-D scatterplot with lines drawn from the points down to the  $z = 0$  plane to help visualize the third dimension. The `grid` option to the `scatter` statement may help in clarifying the plot, while the needles can be omitted with the `noneedle` option. The `x` and `y` vars must be a grid for the `plot` statement in either the `g3d` (where it produces a surface plot) or the `gcontour` procedure; if not, the `g3grid` procedure can be used to smooth values. The `rotate` and `tilt` options to the `plot` and `scatter` statements will show the plot from a different perspective for the `g3d` procedure.

**R**

```
persp(x, y, z)
image(x, y, z)

library(scatterplot3d)
scatterplot3d(x, y, z)
```

*Note:* The values provided for `x` and `y` must be in ascending order.

## 8.5 Special purpose plots

### 8.5.1 Choropleth maps

**SAS**

*Example:* 12.3.3

```
proc gmap map=mapds data=ds all;
  id idvar;
  choro x1;
run; quit;
```

*Note:* The map dataset `mapds` is specially formatted to define the boundaries of the map region, and both `mapds` and `ds` have the variable `idvar`. The data set `ds` additionally contains the variable `x1`, and the procedure will assign each region a different color or pattern depending on the value of `x1`. Patterns and colors can be controlled with `pattern` statements.

**R**

```
library(ggmap)
mymap = map_data('state')  # need to add variable to plot
p0 = ggplot(map_data, aes(x=x, y=y, group=z)) +
  geom_polygon(aes(fill = cut_number(z, 5))) +
  geom_path(colour = 'gray', linestyle = 2) +
  scale_fill_brewer(palette = 'PuRd') +
  coord_map();
plot(p0)
```

*Note:* More examples of maps can be found in the `ggmap` package documentation.

### 8.5.2 Interaction plots

*Example:* 6.6.5

Interaction plots are used to display means by two variables (as in a two-way analysis of variance, 6.1.8).

**SAS**

```
proc glm data=ds;
  class x1 x2;
  model y = x1|x2;
run;
```

*Note:* In the above, the interaction plot is produced by default; the `ods select` statement can be used if only the graphic is desired. In addition, an interaction plot can be generated using the `means` and `gplot` procedures (as shown in 6.6.5).

**R**

```
interaction.plot(x1, x2, y)
```

*Note:* The default statistic to compute is the mean; other options can be specified using the `fun` option.

### 8.5.3 Plots for categorical data

A variety of less traditional plots can be used to graphically represent categorical data. While these tend to have a low data to ink ratio, they can be useful in figures with repeated multiples [178]. SAS macros to make some of them are available from Michael Friendly at <http://www.datavis.ca/sasmac>.

**R**

```
mosaicplot(table(x, y, z))
assocplot(table(x, y))
```

*Note:* The `mosaicplot()` function provides a graphical representation of a two-dimensional or higher contingency table, with the area of each box representing the number of observations in that cell. The `assocplot()` function can be used to display the deviations from independence for a two-dimensional contingency table. Positive deviations of observed minus expected counts are above the line and colored black, while negative deviations are below the line and colored red. Tables can be included in graphics using the `gridExtra` packages (see 5.7.3).

### 8.5.4 Circular plot

Circular plots are used to analyze data that wraps (e.g., directions expressed as angles, time of day on a 24 hour clock) [40, 86]. SAS macros for circular statistics, including a `circplot` macro, are available from Ulric Lund's webpage at Cal Poly San Luis Obispo.

**R**

```
library(circular)
plot.circular(x, stack=TRUE, bins=50)
```

### 8.5.5 Plot an arbitrary function

**SAS**

```
data ds;
do x = start to stop by step;
  y = some_function_of_x;
  output;
end;
run;

symbol1 i = j;
proc gplot;
  plot y * x;
run; quit;
```

*Example:* 10.1.6

*Note:* In the code above we use the `do` syntax shown in 4.1.3. The right hand side of `y =` is any function of `x`. The `plot` method is discussed in 9.1.5.

**R**

```
curve(expr, from=start, to=stop, n=number)
or
x = seq(from=start, to=stop, by=step)
y = expr(x)
plot(x, y)
```

*Note:* The `curve()` function can be used to plot an arbitrary function denoted by `expr`, with `n` values of `x` between `start` and `stop` (see 9.1.5). This can also be built up in parts by use of the `seq()` function. The `plotFun()` function in the `mosaic` package can be used to plot regression models or other functions generated using `makeFun()`.

### 8.5.6 Normal quantile-quantile plot

*Example: 6.6.3*

Quantile-quantile plots are a commonly used graphical technique to assess whether a univariate sample of random variables is consistent with a Gaussian (normal) distribution.

#### SAS

```
proc univariate data=ds plot;
  var x;
run;
or
proc univariate data=ds;
  var x;
  qqplot x;
run;
```

*Note:* The normal Q-Q plot from the `plot` option is a text-based version; it is accompanied by a stem-and-leaf and a box plot. The plot from the `qqplot` statement is a graphics version. Q-Q plots for other distributions are also available as options to the `qqplot` statement.

#### R

```
qqnorm(x)
qqline(x)
```

*Note:* The `qqline()` function adds a straight line which goes through the first and third quartiles.

### 8.5.7 Receiver operating characteristic (ROC) curve

*Example: 8.7.5*

See also 5.2.2 (diagnostic agreement) and 7.1.1 (logistic regression).

Receiver operating characteristic curves can be used to help determine the optimal cut-score to predict a dichotomous measure. This is particularly useful in assessing diagnostic accuracy in terms of sensitivity (the probability of detecting the disorder if it is present), specificity (the probability that a disorder is not detected if it is not present), and the area under the curve (AUC). The variable `x` represents a predictor (e.g., individual scores) and `y` a dichotomous outcome. There is a close connection between the idea of the ROC curve and goodness of fit for logistic regression, where the latter allows multiple predictors to be used. In SAS, ROC curves are embedded in `proc logistic`; to emulate the functions available in the R `ROCR` package [167], just use a single predictor in SAS `proc logistic`.

#### SAS

```
proc logistic data=ds plots(only)=roc;
  model y = x1 ... xk;
run;
```

*Note:* The `plots(only)` option is used to request only the ROC curve be produced, rather than the default inclusion of several additional plots. The probability cutpoint associated with each point on the ROC curve can be printed using `roc(id=prob)` in place of `roc` above.

#### R

```
library(ROCR)
pred = prediction(x, y)
perf = performance(pred, "tpr", "fpr")
plot(perf)
```

*Note:* The area under the curve (AUC) can be calculated by specifying "auc" as an argument when calling the `performance()` function.

### 8.5.8 Plot confidence intervals for the mean

#### SAS

```
symbol1 i=rlclm95 value=none;
proc gplot data=ds;
  plot y * x;
run;
```

*Note:* The `symbol` statement `i` option (synonym for `interpolation`) contains many useful options for adding features to scatterplots. The `rlclm95` selection requests a regression line plot, with 95% confidence limits for the mean. The `value=none` requests that the observations themselves not be plotted (see scatterplots, 8.3.1).

#### R

```
pred.w.clim = predict(lm(y ~ x), interval="confidence")
matplot(x, pred.w.clim, lty=c(1, 2, 2), type="l", ylab="predicted y")
or
library(mosaic)
xyplot(y ~ x, panel=panel.lmbands)
```

*Note:* The first entry produces fit and confidence limits at the original observations in the original order. If the observations aren't sorted relative to the explanatory variable `x`, the resulting plot will be a jumble. The `matplot()` function is used to generate lines, with a solid line (`lty=1`) for predicted values and a dashed line (`lty=2`) for the confidence bounds. The `panel.lmbands()` function in the `mosaic` package can also be used to plot these values.

### 8.5.9 Plot prediction limits from a simple linear regression

#### SAS

```
symbol1 i=rlcli95 l=2 value=none;
proc gplot data=ds;
  plot y * x;
run;
```

*Note:* The `symbol` statement `i` (synonym for `interpolation`) option contains many useful options for adding features to scatterplots. The `rlcli95` selection requests a regression line plot, with 95% confidence limits for the values. The `value=none` requests that the observations not be plotted (see 8.3.1).

#### R

```
pred.w.plim = predict(lm(y ~ x), interval="prediction")
matplot(x, pred.w.plim, lty=c(1, 2, 2), type="l", ylab="predicted y")
```

*Note:* This entry produces fit and confidence limits at the original observations in the original order. If the observations aren't sorted relative to the explanatory variable `x`, the resulting plot will be a jumble. The `matplot()` function is used to generate lines, with a solid line (`lty=1`) for predicted values and a dashed line (`lty=2`) for the confidence bounds.

### 8.5.10 Plot predicted lines for each value of a variable

Here we describe how to generate plots for a variable  $X_1$  versus  $Y$  separately for each value of the variable  $X_2$  (see conditioning plot, 8.4.2).

**SAS**

```
symbol1 i=rl value=none;
symbol2 i=rl value=none;
proc gplot data=ds;
  plot y*x1 = x2;
run;
```

*Note:* The **symbol** statement **i** (synonym for **interpolation**) option contains many useful options for adding features to scatterplots. The **rl** selection requests a regression line plot. The **value=none** requests that the observations not be plotted. The **= x2** syntax requests a different **symbol** statement be applied for each level of **x2** (see scatterplots, 8.3.1).

**R**

```
plot(x1, y, pch=" ") # create an empty plot of the correct size
abline(lm(y ~ x1, subset=x2==0), lty=1, lwd=2)
abline(lm(y ~ x1, subset=x2==1), lty=2, lwd=2)
...
abline(lm(y ~ x1, subset=x2==k), lty=k+1, lwd=2)
```

*Note:* The **abline()** function is used to generate lines for each of the subsets, with a solid line (**lty=1**) for the first group and a dashed line (**lty=2**) for the second (this assumes that  $X_2$  takes on values 0– $k$ , see 11.1). The **plotFun()** function in the **mosaic** package provides another way of adding lines or arbitrary curves to a plot.

### 8.5.11 Kaplan–Meier plot

See also 5.4.6 (log-rank test).

*Example:* 8.7.4

**SAS**

```
ods select survivalplot;
proc lifetest data=ds plots=s;
  time time*status(1);
  strata x;
run;
or
proc lifetest data=ds outsurv=survds;
  time time*status(1);
  strata x;
run;

symbol1 i=stepj r=kx;
proc gplot data=survds;
  plot survival*survtime = x;
run;
```

*Note:* The second approach demonstrates how to manually construct the plot. The survival estimates generated by **proc lifetest** are saved in a new dataset using the **outsurv** option to the **proc lifetest** statement; we suppose there are  $kx$  levels of **x**, the stratification variable.

For the plot, a step-function to connect the points is specified using the **i=stepj** option to the **symbol** statement. Finally, **proc gplot** with the **a\*b=c** syntax (9.1.2) is called. In this case, **survival\*survtime=x** will plot lines for each of the  $kx$  levels of **x**. Here, **survival** and **survtime** are variable names created by **proc lifetest**. Note that the **r=kx** option to the **symbol** statement is shorthand for typing in the same options for **symbol1**, **symbol2**, ..., **symbolkx** statements; here we repeat them for the  $kx$  strata specified in **x**.

**R**

```
library(survival)
fit = survfit(Surv(time, status) ~ as.factor(x), data=ds)
plot(fit, conf.int=FALSE, lty=1:length(unique(x)))
```

*Note:* The `Surv()` function is used to combine survival time and status, where `time` is length of follow-up (interval censored data can be accommodated via an additional parameter) and `status=1` indicates an event (e.g., death) while `status=0` indicates censoring. The model is stratified by each level of the group variable `x` (see adding legends, 9.1.15, and different line styles, 9.2.11). More information can be found in the CRAN survival analysis task view.

### 8.5.12 Hazard function plotting

The hazard function from right censored data can be estimated using kernel methods (see also 8.7.4).

The following SAS code relies on the `smooth` macro written by Paul Allison and provided in his text [7]. The macro is available from [www.amherst.edu/~nhorton/sasr2/examples/smooth.sas](http://www.amherst.edu/~nhorton/sasr2/examples/smooth.sas).

**SAS**

```
proc lifetest data=ds outsurv=os;
  time time*status(0);
  strata x;
run;

%include "c:/ken/sasmacros/smooth.sas";
%smooth(data=os, time=time, width=25);
```

*Note:* The `width` option is the bandwidth for the smoother used for estimating the function.

**R**

```
library(muhaz)
plot(muhaz(time, status))
```

*Note:* Estimation of hazards from multiple groups can be plotted together using the `lines()` command after running `muhaz()` on the other groups.

### 8.5.13 Mean-difference plots

The Tukey mean-difference plot, popularized in medical research as the “Bland–Altman” plot [8] plots the difference between two variables against their mean. This can be useful when they are two different methods for assessing the same quantity.

We’re unaware of a tool to do this directly in SAS and provide the following macro, which can be retyped or downloaded from [www.amherst.edu/~nhorton/sasr2/examples/baplot.sas](http://www.amherst.edu/~nhorton/sasr2/examples/baplot.sas).

**SAS**

```
%macro baplot(datain=,x=x,y=y);

/* calculate differences and averages */
data ba;
set &datain;
  bamean = (&x + &y)/2;;
  badiff = &y-&x;
run;

ods output summary=basumm;
ods select none;
proc means data=ba mean std;
  var badiff;
run;
ods select all;

data lines;
set basumm;
  call symput('bias',badiff_mean);
  call symput('hici',badiff_mean+(1.96 * badiff_stddev));
  call symput('loci',badiff_mean-(1.96 * badiff_stddev));
run;

symbol1 i=none v=dot h=.5;
title "Bland-Altman type plot of &x and &y";
title2 "&x and &y standardized";
proc gplot data=ba;
  plot badiff * bamean / vref=&bias &hici &loci lvref=3;
  label badiff = "difference" bamean="mean";
run;
%mend baplot;

%baplot(datain=ds, x=x1, y=x2);
```

*Note:* In data lines we take values for the mean and confidence limits calculated by proc summary and store them in macro variables using the powerful call symput function (4.2.1). The values are used in the vref option in the gplot code; vref draws reference line(s) on the vertical axis while lvref specifies a line type for the reference lines.

**R**

```
set.seed(42); n=500
x = rnorm(n)
y = x + rnorm(n)
baplot = function(x, y) {
  bamean = (x + y)/2
  badiff = (y - x)
  plot(badiff ~ bamean, pch=20, xlab="mean", ylab="difference")
  abline(h = c(mean(badiff), mean(badiff)+1.96 * sd(badiff),
              mean(badiff)-1.96 * sd(badiff)), lty=2)
}
baplot(x, y)
```

## 8.6 Further resources

The books by Tufte [177, 178, 179, 180] provide an excellent framework for graphical displays, some of which build on the work of Tukey [181]. Comprehensive and accessible books on R graphics include [123], [151] and [196].

## 8.7 Examples

To help illustrate the tools presented in this chapter, we apply many of the entries to the HELP data. SAS and R code can be downloaded from <http://www.amherst.edu/~nhorton/sasr2/examples>. We begin by reading in the data.

```
proc import
  datafile='c:/book/help.csv'
  out=ds
  dbms=dlm;
  delimiter=',';
  getnames=yes;
run;
> options(digits=3)
> ds = read.csv("help.csv")
```

### 8.7.1 Scatterplot with multiple axes

The following example creates a single figure that displays the relationship between CESD and the variables `indtot` (Inventory of Drug Abuse Consequences, InDUC) and `mcs` (Mental Component Score) for a subset of female alcohol-involved subjects. We specify two different y axes (8.3.2) for the figure.

```
axis1 minor=none;
axis2 minor=none order=(5 to 60 by 13.625);
axis3 minor=none order=(20, 40, 60);
symbol1 i=sm65s v=circle color=black l=1 w=5;
symbol2 i=sm65s v=triangle color=black l=2 w=5;
proc gplot data=ds;
  where female eq 1 and substance eq 'alcohol';
  plot indtot*cesd / vaxis=axis1 haxis=axis3;
  plot2 mcs*cesd / vaxis = axis2;
run; quit;
```

In the SAS code above, the `symbol` and `axis statements` are used to control the output and to add lines through the data. Note that three axes are specified and are associated with the various axes in the plot in the `vaxis` and `haxis` options to the `plot` and `plot2` statements. The `axis` statements can be omitted for a simpler graphic.

In R, a considerable amount of housekeeping is needed. The second y variable must be rescaled to the range of the original, and the axis labels and tick marks added on the right. To accomplish this, we write a function `plottwoy()`, which first makes the plot of the first (left axis) y against x, adds a lowess curve through that data, then calls a second function, `addsecondy()`.

```
> plottwoy = function(x, y1, y2, xname="X", y1name="Y1", y2name="Y2") {
  plot(x, y1, ylab=y1name, xlab=xname)
  lines(lowess(x, y1), lwd=3)
  addsecondy(x, y2, y1, yname=y2name)
}
```

The function `addsecondy()` does the work of rescaling the range of the second variable to that of the first, adds the right axis, and plots a lowess curve through the data for the rescaled `y2` variable.

```
> addsecondy = function(x, y, origy, yname="Y2") {
  prevlimits = range(origy)
  axislimits = range(y)
  axis(side=4, at=prevlimits[1] + diff(prevlimits)*c(0:5)/5,
       labels=round(axislimits[1] + diff(axislimits)*c(0:5)/5, 1))
  mtext(yname, side=4)
  newy = (y-axislimits[1])/(diff(axislimits)/diff(prevlimits)) +
    prevlimits[1]
  points(x, newy, pch=2)
  lines(lowess(x, newy), lty=2, lwd=3)
}
```

Finally, the newly defined functions can be run and Figure 8.1 generated.

```
> with(ds, plottwoy(cesd[female==1&substance=="alcohol"],
  indtot[female==1&substance=="alcohol"],
  mcs[female==1&substance=="alcohol"], xname="cesd",
  y1name="indtot", y2name="mcs"))
```

Note that the two graphics are not identical due to different y axes. In SAS it is difficult to select axis ranges exactly conforming to the range of the data, while our R function uses more of the space for data display.

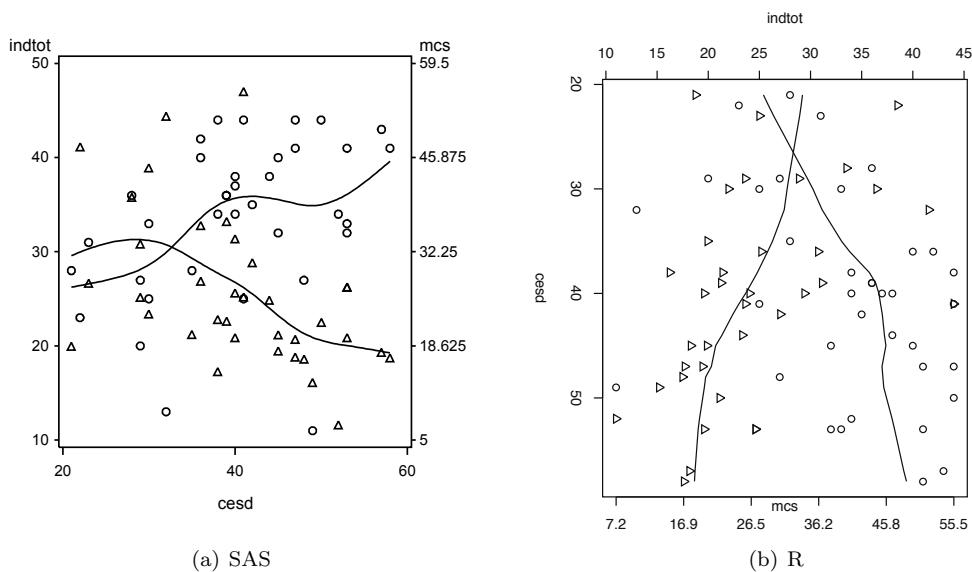


Figure 8.1: Plot of InDUC and MCS vs. CESD for female alcohol-involved subjects

## 8.7.2 Conditioning plot

Figure 8.2 displays a conditioning plot (8.4.2) with the association between MCS and CESD stratified by substance and report of suicidal thoughts (g1b).

```
proc sgpanel data=ds;
  panelby g1b substance / layout=lattice;
  pbspline x=cesd y=mcs;
run; quit;
```

For R, ensure that the necessary packages are loaded (B.6.1).

```
> library(lattice)
```

Then we can set up and generate the plot.

```
> ds = transform(ds, suicidal.thoughts = ifelse(g1b==1, "Y", "N"))
> coplot(mcs ~ cesd | suicidal.thoughts*substance,
  panel=panel.smooth, data=ds)
```

There is a similar association between CESD and MCS for each of the substance groups. Subjects with suicidal thoughts tended to have higher CESD scores, and the association between CESD and MCS was somewhat less pronounced than for those without suicidal thoughts.

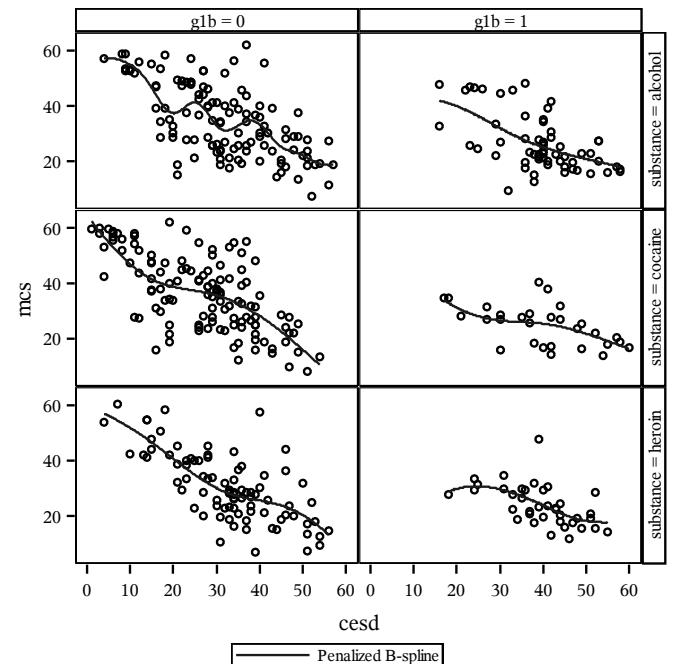
## 8.7.3 Scatterplot with marginal histograms

We can assess the univariate as well as bivariate distribution of the MCS and CESD scores using a scatterplot with a marginal histogram (8.3.6), as shown in Figure 8.3.

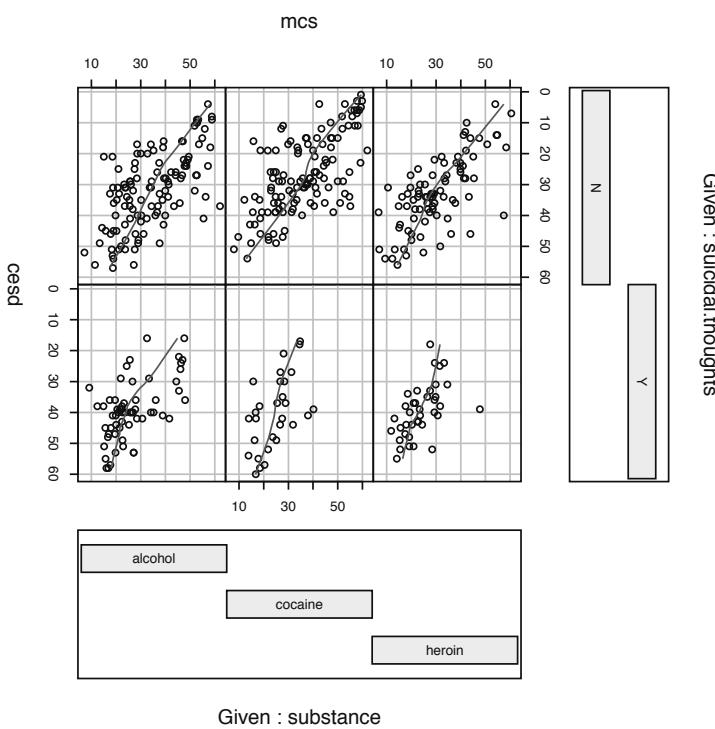
```
%include "c:\book\scatterhist.sas"
proc sgrender data=ds template=scatterhist;
  dynamic YVAR="PCS" XVAR="MCS";
run;
```

The R implementation utilizes the `layout()` function to create the graphic (9.2.3).

```
> scatterhist = function(x, y, xlab="x label", ylab="y label"){
  zones=matrix(c(3,1,2,4), ncol=2, byrow=TRUE)
  layout(zones, widths=c(4/5,1/5), heights=c(1/5,4/5))
  par(mar=c(0,0,0,0))
  plot(type="n",x=1, y =1, bty="n",xaxt="n", yaxt="n")
  text(x=1,y=1,paste0("nobs = ",min(length(x), length(y))), cex =1.8)
  xhist = hist(x, plot=FALSE)
  yhist = hist(y, plot=FALSE)
  top = max(c(xhist$counts, yhist$counts))
  par(mar=c(3,3,1,1))
  plot(x,y)
  par(mar=c(0,3,1,1))
  barplot(xhist$counts, axes=FALSE, ylim=c(0, top), space=0)
  par(mar=c(3,0,1,1))
  barplot(yhist$counts, axes=FALSE, xlim=c(0, top), space=0, horiz=TRUE)
  par(oma=c(3,3,0,0))
  mtext(xlab, side=1, line=1, outer=TRUE, adj=0,
        at=.8 * (mean(x) - min(x))/(max(x)-min(x)))
  mtext(ylab, side=2, line=1, outer=TRUE, adj=0,
        at=(.8 * (mean(y) - min(y))/(max(y) - min(y))))
}
> with(ds, scatterhist(mcs, pcs, xlab="MCS", ylab="PCS"))
```

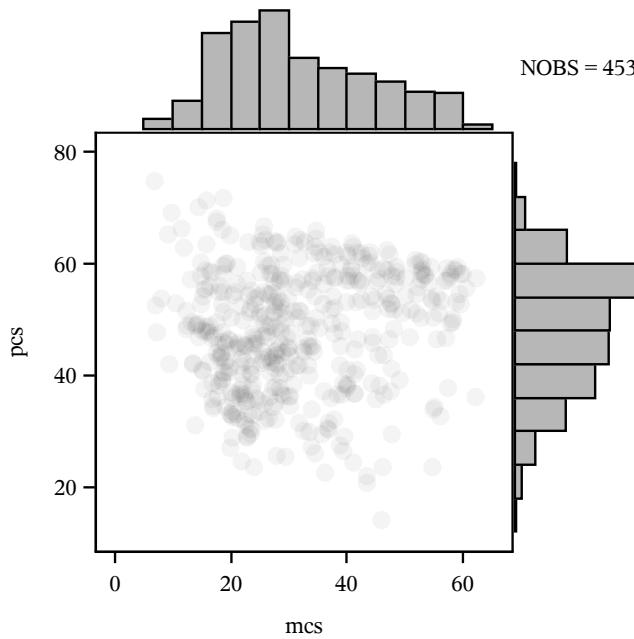


(a) SAS

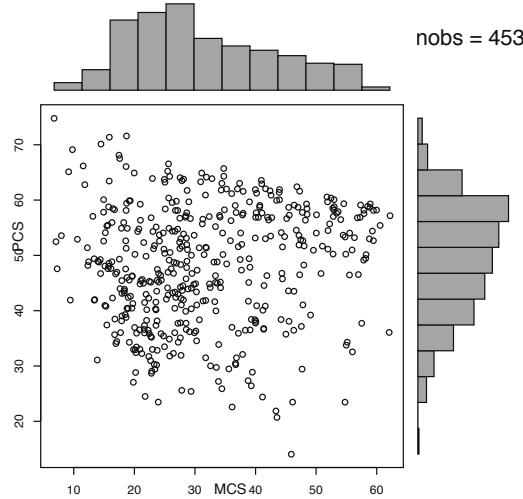


(b) R

Figure 8.2: Association of MCS and CESD, stratified by substance and report of suicidal thoughts



(a) SAS



(b) R

Figure 8.3: Association of MCS and CESD with marginal histograms

### 8.7.4 Kaplan–Meier plot

The main outcome of the HELP study was time to linkage to primary care, as a function of randomization group. This can be displayed using a Kaplan–Meier plot (see 8.5.11). For SAS, detailed information regarding the Kaplan–Meier estimator at each time point can be found by omitting the `ods select` statement, and for R by using `summary(survobj)`. Figure 8.4 displays the estimates, with + signs indicating censored observations.

```
ods select censoredsummary survivalplot;
proc lifetest data=ds plots=s(test);
  time dayslink*linkstatus(0);
  strata treat;
run;
```

The LIFETEST Procedure

#### Summary of the Number of Censored and Uncensored Values

Stratum	treat	Total	Failed	Censored	Percent Censored
1	0	209	35	174	83.25
2	1	222	128	94	42.34
<hr/>		Total	431	163	268
					62.18

NOTE: 22 observations with invalid time, censoring, or strata values were deleted.

```
> library(survival)
> survobj = survfit(Surv(dayslink, linkstatus) ~ treat, data=ds)
> print(survobj)

Call: survfit(formula = Surv(dayslink, linkstatus) ~ treat, data = ds)

22 observations deleted due to missingness
      records n.max n.start events median 0.95LCL 0.95UCL
treat=0     209    209     209     35     NA     NA     NA
treat=1     222    222     222    128    120     79    272
> plot(survobj, lty=1:2, lwd=2, col=c(4,2))
> title("Product-Limit Survival Estimates")
> legend(250, .75, legend=c("Control", "Treatment"), lty=c(1,2), lwd=2,
  col=c(4,2), cex=1.4)
```

As reported previously [76, 149], there is a highly statistically significant effect of treatment, with approximately 55% of clinic subjects linking to primary care, as opposed to only 15% of control subjects.

### 8.7.5 ROC curve

Receiver operating characteristic (ROC) curves are used for diagnostic agreement (5.2.2 and 8.5.7) as well as assessing goodness of fit for logistic regression (7.1.1). In SAS, they can be created using `proc logistic`. In R, we use the `ROCR` package. Figure 8.5 displays the receiver operating characteristic curve predicting suicidal thoughts using the CESD measure of depressive symptoms.

```
ods graphics on;
ods select roccurve;
proc logistic data=ds descending plots(only)=roc;
  model g1b = cesd;
run;
ods graphics off;
```

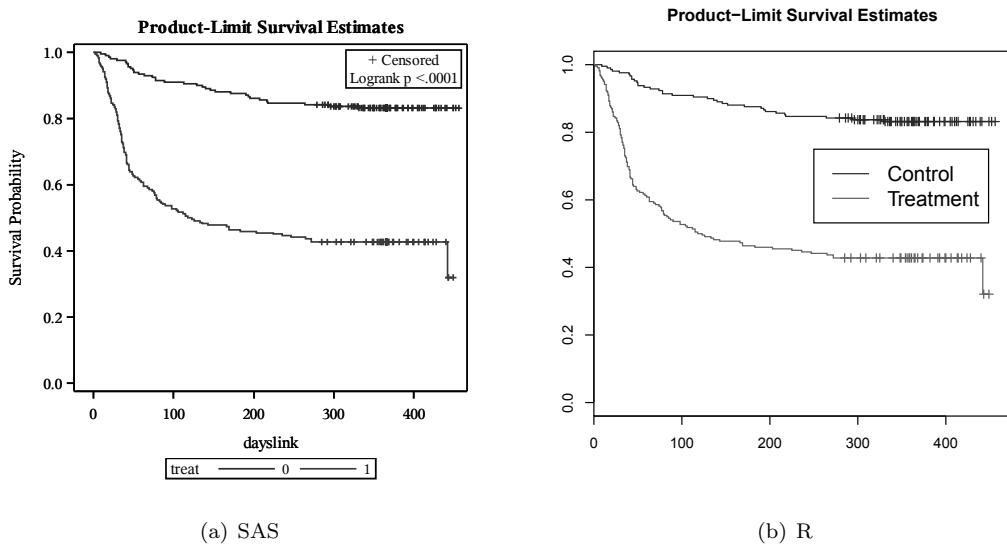


Figure 8.4: Kaplan–Meier estimate of time to linkage to primary care by randomization group

The `descending` option changes the behavior of `proc logistic` to model the probability that the outcome is 1; the default models the probability that the outcome is 0.

Using R, we first load the `ROCR` package, create a prediction object, and retrieve the area under the curve (AUC) to use in Figure 8.5.

```
> library(ROCR)
> pred = with(ds, prediction(cesd, g1b))
> auc = slot(performance(pred, "auc"), "y.values")[[1]]
```

We can then plot the ROC curve, adding a display of cutoffs for particular CESD values ranging from 20 to 50. These values are offset from the ROC curve using the `text.adj` option.

If the continuous variable (in this case `cesd`) is replaced by the predicted probability from a logistic regression model, multiple predictors can be included.

```
> plot(performance(pred, "tpr", "fpr"),
      print.cutoffs.at=seq(from=20, to=50, by=5),
      text.adj=c(1, -.5), lwd=2)
> lines(c(0, 1), c(0, 1))
> text(.6, .2, paste("AUC=", round(auc,3), sep=""), cex=1.4)
> title("ROC Curve for Model")
```

## 8.7.6 Pairs plot

We can qualitatively assess the associations between some of the continuous measures of mental health, physical health, and alcohol consumption using a pairsplot or scatterplot matrix (8.4.1). To make the results clearer, we display only the female subjects.

For SAS, the new `sgscatter` procedure provides a simple way to produce this. The results of the following code are included in Figure 8.6.

```
proc sgscatter data=ds;
  where female eq 1;
  matrix cesd mcs pcs i1 / diagonal=(histogram kernel);
run; quit;
```

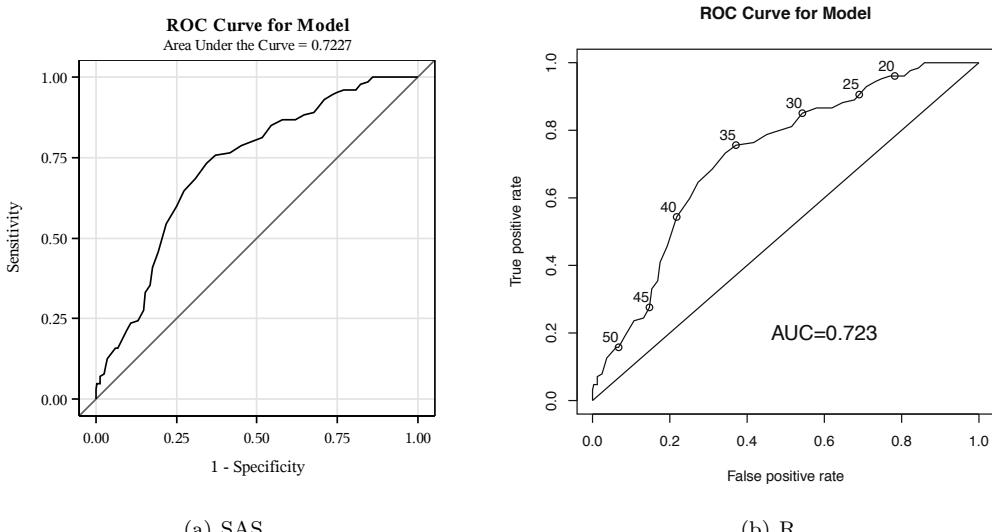


Figure 8.5: Receiver operating characteristic curve for the logistical regression model predicting suicidal thoughts using the CESD as a measure of depressive symptoms (sensitivity = true positive rate; 1-specificity = false positive rate)

If fitted curves in the pairwise scatterplots are required, the following code will produce a similar matrix, with LOESS curves in each cell and less helpful graphs in the diagonals (results not shown).

```
proc sgscatter data=ds;
  where female eq 1;
  compare x = (cesd mcs pcs i1)
    y = (cesd mcs pcs i1) / loess;
run; quit;
```

For complete control of the figure, the `sgscatter` procedure will not suffice and more complex coding is necessary; we would begin with SAS macros written by Michael Friendly and available from his web site at York University (<http://www.datavis.ca>).

For R, a simple version with only the scatterplots could be generated easily with the `pairs()` function (results not shown):

```
> pairs(c(ds[72:74], ds[67]))
or
> pairs(ds[c("pcs", "mcs", "cesd", "i1")])
```

Here instead we demonstrate building a figure using several functions. We begin with a function `panel.hist()` to display the diagonal entries (in this case, by displaying a histogram).

```
> panel.hist = function(x, ...)
{
  usr = par("usr"); on.exit(par(usr))
  par(usr=c(usr[1:2], 0, 1.5))
  h = hist(x, plot=FALSE)
  breaks = h$breaks; nB = length(breaks)
  y = h$counts; y = y/max(y)
  rect(breaks[-nB], 0, breaks[-1], y, col="cyan", ...)
}
```

Another function is created to create a scatterplot along with a fitted line.

```
> panel.lm = function(x, y, col=par("col"), bg=NA, pch=par("pch"),
  cex=1, col.lm="red", ...)
{
  points(x, y, pch=pch, col=col, bg=bg, cex=cex)
  ok = is.finite(x) & is.finite(y)
  if (any(ok))
    abline(lm(y[ok] ~ x[ok]))
}
```

These functions are called (along with the built-in `panel.smooth()` function) to display the results. Figure 8.6 displays the pairsplot of CESD, MCS, PCS, and I1, with histograms along the diagonals. For R, smoothing splines are fit on the lower triangle, linear fits on the upper triangle, using code fragments derived from `example(pairs)`.

```
> pairs(~ cesd + mcs + pcs + i1, subset=(female==1),
  lower.panel=panel.smooth, diag.panel=panel.hist,
  upper.panel=panel.lm, data=ds)
```

There is an indication that CESD, MCS, and PCS are interrelated, while I1 appears to have modest associations with the other variables.

### 8.7.7 Visualize correlation matrix

One visual analysis which might be helpful to display would be the pairwise correlations. We approximate this in SAS by plotting a confidence ellipse for the observed data. This approach allows an assessment of whether the linear correlation is an appropriate statistic to consider.

In the code below, we demonstrate some options for the `sgscatter` procedure. The `ellipse` option draws confidence ellipses at the requested  $\alpha$ -level, here chosen arbitrarily to mimic R. The `start` option also mimics R by making the diagonal begin in the lower left; the top left is the default. The `markeratrs` option controls aspects of the appearance of plots generated with the `sgscatter`, `sgpanel`, and `sgplot` procedures.

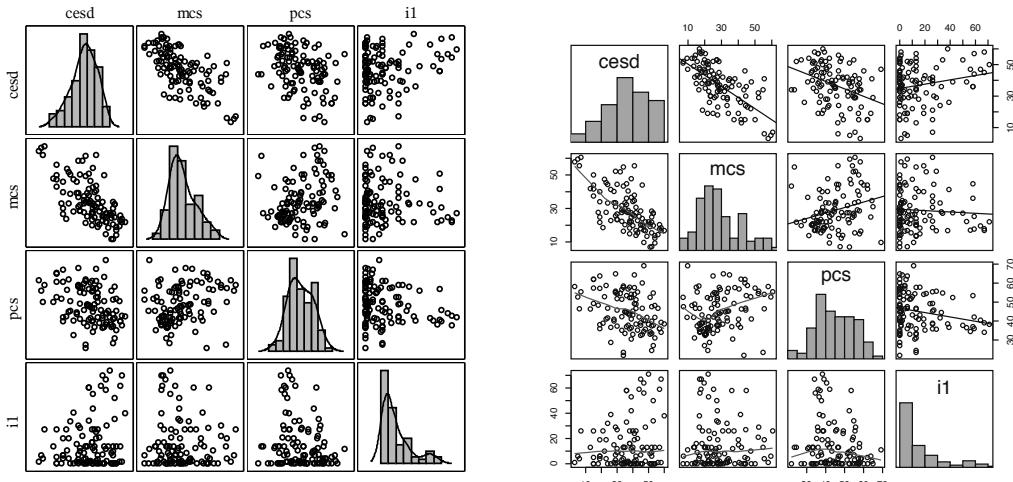


Figure 8.6: Pairsplot of variables from the HELP dataset

```

proc sgscatter data=ds;
  matrix mcs pcs pss_fr drugrisk cesd indtot i1 sexrisk /
    ellipse=(alpha=.25) start=bottomleft
    markerattrs=(symbol=circlefilled size=4);
run; quit;

```

In R, we utilize the approach used by Sarkar to recreate Figure 13.5 of the *Lattice: Multivariate data visualization with R* book [151]. Other examples in that reference help to motivate the power of the lattice package far beyond what is provided by `demo(lattice)`.

```

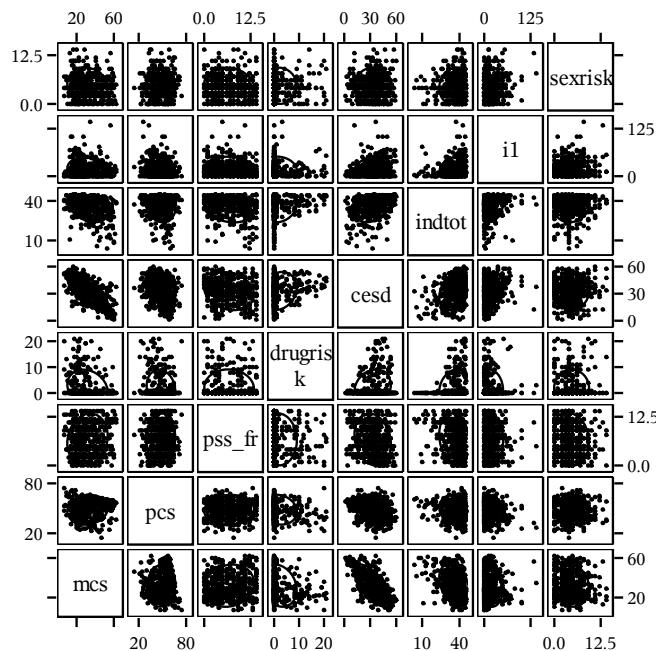
> cormat = with(ds, cor(cbind(mcs, pcs, pss_fr, drugrisk,
+ cesd, indtot, i1, sexrisk), use="pairwise.complete.obs"))
> oldopt = options(digits=2)
> cormat

      mcs     pcs pss_fr drugrisk     cesd indtot      i1 sexrisk
mcs     1.000  0.110  0.138 -0.2058 -0.682 -0.38 -0.087 -0.1061
pcs     0.110  1.000  0.077 -0.1411 -0.293 -0.13 -0.196  0.0239
pss_fr  0.138  0.077  1.000 -0.0390 -0.184 -0.20 -0.070 -0.1128
drugrisk -0.206 -0.141 -0.039  1.0000  0.179  0.18 -0.100 -0.0055
cesd    -0.682 -0.293 -0.184  0.1789  1.000  0.34  0.176  0.0157
indtot  -0.381 -0.135 -0.198  0.1807  0.336  1.00  0.202  0.1132
i1      -0.087 -0.196 -0.070 -0.0999  0.176  0.20  1.000  0.0881
sexrisk -0.106  0.024 -0.113 -0.0055  0.016  0.11  0.088  1.0000

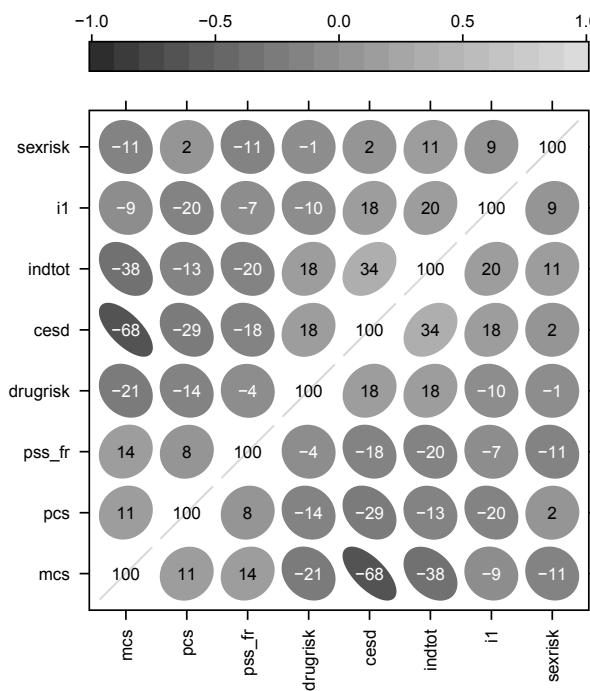
> options(olddopt)
> ds$drugrisk[is.na(ds$drugrisk)] = 0
> panel.corrgram = function(x, y, z, at, level=0.9, label=FALSE, ...)
{
  require("ellipse", quietly=TRUE)
  zcol = level.colors(z, at=at, col.regions=gray.colors)
  for (i in seq(along=z)) {
    ell = ellipse(z[i], level=level, npoints=50,
                  scale=c(.2, .2), centre=c(x[i], y[i]))
    panel.polygon(ell, col=zcol[i], border=zcol[i], ...)
  }
  if (label)
    panel.text(x=x, y=y, lab=100*round(z, 2), cex=0.8,
               col=ifelse(z < 0, "white", "black"))
}
> library(ellipse)
> library(lattice)
> print(levelplot(cormat, at=do.breaks(c(-1.01, 1.01), 20),
+                 xlab=NULL, ylab=NULL, colorkey=list(space = "top",
+                                           col=gray.colors), scales=list(x=list(rot = 90)),
+                 panel=panel.corrgram, label=TRUE))

```

The SAS plot in Figure 8.7 suggests that some of these linear correlations might not be useful measures of association, while the R plot allows a consistent frame of reference for the many correlations.



(a) SAS



(b) R

Figure 8.7: Visual display of correlations and associations

# Chapter 9

# Graphical options and configuration

This chapter describes how to annotate graphical displays and change defaults to create publication-quality figures, as well as details regarding how to output graphics in a variety of file formats (9.3).

## 9.1 Adding elements

In R, it is relatively simple to add features to graphs which have been generated by one of the functions discussed in Chapter 8. For example, adding an arbitrary line to a graphic requires only one function call with the two endpoints as arguments (9.1.1). In SAS, such additions are made using a specially formatted dataset called an `annotate` dataset; see 12.3 for an example. These datasets contain certain required variable names and values. Perfecting a graphic for publication may require `annotate` datasets, a powerful low-level tool. Their use is made somewhat easier by a suite of SAS macros, the `annotate` macros provided with SAS/GRAFPH. To use the macros, you must first enable them in the following way.

```
%annomac;
```

The macros can be called to draw a line between two points, plot a circle, and so forth. This is accomplished by creating an `annotate` dataset and calling the macros within it.

```
data annods;  
  %system(x, y, s);  
  ...  
run;
```

Here the ellipses refer to additional `annotate` macros. The `system` macro is useful in getting the macros to work as desired; it defines how the values of `x` and `y` in later `annotate` macros are interpreted as well as the size of the plotted values. For example, to measure in terms of the graphics output area, use the value 3 for the first two parameters in the `system` macro. This can be useful for drawing outside the axes. More frequently, we find that using the coordinate system of the plot itself is most convenient; using the value 2 for each parameter will implement this.

### 9.1.1 Arbitrary straight line

*Example:* 6.6.1

#### SAS

```
%annomac;
data annods;
  %system(2,2,2);
  %line(xvalue_1, yvalue_1, xvalue_2, yvalue_2, colorspec, linetype, .01);
run;

proc gplot data=ds;
  plot x*y / anno=annods;
run; quit;
```

*Note:* See 9.1 for an overview of annotate datasets. The `line` macro draws a line from `(xvalue_1, yvalue_1)` to `(xvalue_2, yvalue_2)`. The line will have the color (9.2.13) specified by `colorspec` and be solid or dashed (9.2.11), as specified in `linetype`. The final entry specifies the width of the line, here quite narrow. Another approach would be to add the endpoint values to the original dataset, then use the `symbol` statement and the `a*b=c` syntax of `proc gplot` (9.1.2).

#### R

```
plot(x, y)
lines(point1, point2)
or
abline(intercept, slope)
```

*Note:* The `lines()` function draws lines between successive pairs of locations specified by `point1` and `point2`, which are each vectors with values for the `x` and `y` axes. The `abline()` function draws a line based on the slope-intercept form. Vertical or horizontal lines can be specified using the `v` or `h` option to `abline()`.

### 9.1.2 Plot symbols

*Example:* 5.7.2

#### SAS

```
symbol1 value=valuename;
symbol1 value='plottext';
symbol1 font=fontname value=plottext;
proc gplot data=ds;
  ...
run;
or
proc gplot data=ds;
  plot y*x = groupvar;
run; quit;
```

*Note:* The specific characters plotted in `proc gplot` can be controlled using the `value` option to a preceding `symbol` statement, as demonstrated in Figure 5.2. The `valuenames` available include `dot`, `point`, `plus`, `diamond`, and `triangle`. They can also be colored with the `color` option and their size changed with the `height` option. A full list of plot symbols can be found in the on-line help: Contents; SAS Products; SAS/GRAPH; Statements; SYMBOL. The list appears approximately two-thirds of the way through the entry. Additionally, any font character or string can be plotted, if enclosed in quotes as in the second `symbol` statement example, or without the quotes if a `font` option is specified as in the third example.

In the second set of code, a unique plot symbol or color is printed for each value of the variable `groupvar`. If there are many values, for example, if `groupvar` is continuous, the results can be confusing.

**R**

```
plot(x, y, pch=pchval)
or
points(x, y, string, pch=pchval)
or
library(lattice)
xyplot(x ~ y, group=factor(groupvar), data=ds)
or
library(ggplot2)
qplot(x, y, col=factor(groupvar), shape=factor(groupvar), data=ds)
```

*Note:* The `pch` option requires either a single character or an integer code. Some useful values include 20 (dot), 46 (point), 3 (plus), 5 (diamond), and 2 (triangle) (running `example(points)` will display more possibilities). The size of the plotting symbol can be changed using the `cex` option. The vector function `text()` adds the value in the variable `string` to the plot at the specified location. The examples using `xyplot()` and `qplot()` will also generate scatterplots with different plot symbols for each level of `groupvar`.

### 9.1.3 Add points to an existing graphic

See also 9.1.2 (specifying plotting character).

*Example:* 6.6.1

**SAS**

```
%annomac;
data annods;
  %system(2, 2, 2);
  %circle(xvalue, yvalue, radius);
run;

proc gplot data=ds;
  plot x*y / anno=annods;
run; quit;
```

*Note:* See 9.1 for an introduction to `annotate` datasets. The `circle` macro draws a circle with the center at `(xvalue, yvalue)` and with a radius determined by the last parameter. A suitably small radius will plot a point. Another approach is to add a value to the original dataset, then use the `symbol` statement and the `a*b=c` syntax of `proc gplot` (9.1.2).

**R**

```
plot(x, y)
points(x, y)
```

### 9.1.4 Jitter points

*Example:* 5.7.2

Jittering is the process of adding a negligible amount of noise to each observed value so that the number of observations sharing a value can be easily discerned. This can be accomplished in a data step within SAS or using the built-in function within R.

**SAS**

```
data ds;
set newds;
jitterx = x + ((uniform(0) * .4) - .2);
run;
```

*Note:* The above code replicates the default behavior of R, assuming `x` has a minimum distance between values of 1.

**R**

```
jitterx = jitter(x)
```

*Note:* The default value for the range of the random uniforms is 40% of the smallest difference between values.

### 9.1.5 Regression line fit to points

**SAS**

```
symbol1 interpol=rl;
proc gplot data=ds;
plot y*x;
run;
or
proc sgplot data=ds;
reg x=x y=y;
run;
```

*Note:* For `proc gplot`, related interpolations which can be specified in the `symbol` statement are `rq` (quadratic fit) and `rc` (cubic fit). Note also that confidence limits for the mean or for individual predicted values can be plotted by appending `clm` or `cli` after `rx` (see 8.5.8 and 8.5.9). The type of line can be modified as described in 9.2.11. For the `proc sgplot` approach, confidence limits can be requested with the `clm` and/or `cli` options to the `reg` statement; polynomial regression curves can be plotted using the `degree` option. Similar plots can be generated by `proc reg` and by the `sgscatter` and `sgpanel` procedures.

**R**

```
plot(x, y)
abline(lm(y ~ x))
```

*Note:* The `abline()` function accepts regression objects with a single predictor as input.

### 9.1.6 Smoothed line

See also 7.10.8 (generalized additive models).

*Example:* 5.7.2

**SAS**

```
symbol1 interpol=splines;
proc gplot data=ds;
plot y*x;
run;
or
proc loess data=ds;
model y = x;
or
proc gam data=ds plots=all;
model y = x;
```

or

```
proc sgplot;
  loess x=x y=y;
run;
```

*Note:* The **spline** interpolation in the **symbol** statement smooths a plot using cubic splines with continuous second derivatives. Other smoothing **interpolation** options include **sm**, which uses a cubic spline which minimizes a linear combination of the sum of squares of the residuals and the integral of the square of the second derivative. In that case, an integer between 0 and 99 appended to the **sm** controls the smoothness. Another option is **interpol=lx**, which uses a Lagrange interpolation of degree  $x$ , where  $x = 1, 3, 5$ . For all of these smoothers, using the **s** suffix to the method sorts the data internally. If the data are previously sorted, this is not needed. The **sgplot** procedure also offers penalized B-spline smoothing via the **pbspline** statement; the **sgpanel** procedure also includes these smoothers.

**R**

```
plot(...)
  lines(lowess(x, y))
```

*Note:* The **f** parameter to **lowess()** can be specified to control the proportion of points which influence the local value (larger values give more smoothness). The **supsmu()** (Friedman's "super smoother") and **loess()** (local polynomial regression fitting) functions are alternative smoothers.

### 9.1.7 Normal density

*Example: 6.6.3*

A normal density plot can be added as an annotation to a histogram or empirical density.

**SAS**

```
proc sgplot data=ds;
  density x;
run;
```

or

```
proc univariate data=ds;
  histogram x / normal;
run;
```

*Note:* The **sgplot** procedure will draw the estimated normal curve without the histogram, as shown. The histogram can be added using the **histogram** statement; the order of the statements determines which element is plotted on top of the other(s). The **univariate** procedure allows many more distributional curves to be fit; it will generate copious text output unless that is suppressed with the **ods select** statement.

**R**

```
hist(x)
xvals = seq(from=min(x), to=max(x), length=100)
lines(pnorm(xvals, mean(x), sd(x)))
```

### 9.1.8 Marginal rug plot

*Example: 5.7.2*

A rug plot displays the marginal distribution on one of the margins of a scatterplot. While this is possible in SAS, using **annotate** datasets or **proc sgrender**, it is nontrivial.

**R**

```
rug(x, side=sideval)
```

*Note:* The `rug()` function adds a marginal plot to one of the sides of an existing plot (`sideval=1` for bottom (default), `2` for left, `3` for top, and `4` for right side).

**9.1.9 Titles****SAS**

*Example: 5.7.4*

```
title 'Title text';
or
title1 "Main title";
title2 "subtitle";
```

*Note:* The `title` statement is not limited to graphics, but will also print titles on text output. To prevent any title from appearing after having specified one, use a `title` statement with no quoted title text. Up to 99 numbered `title` statements are allowed. For graphic applications, font characteristics can be specified with options to the `title` statement.

**R**

```
title(main="main", sub="sub", xlab="xlab", ylab="ylab")
```

*Note:* The title commands refer to the main title, subtitle, x axis, and y axis, respectively. Some plotting commands (e.g., `hist()`) create titles by default, and the appropriate option within those routines needs to be specified when calling them.

**9.1.10 Footnotes****SAS**

```
footnote 'footnote text';
or
footnote1 "Main footnote";
footnote2 "subtitle";
```

*Note:* The `footnote` statement in SAS is not limited to graphics, but will also print footnotes on text output. To prevent any title from appearing after having specified one, use a `footnote` statement with no quoted footnote text. Up to 10 numbered `footnote` statements are allowed. For graphic applications, font characteristics can be specified with options to the `footnote` statement.

**R**

```
title(sub="sub")
```

*Note:* The `sub` option for the `title()` function generates a subtitle.

**9.1.11 Text****SAS**

*Example: 5.7.2, 8.7.5*

```
%annomac;
data annods;
  %system(2,2,3);
  %label(xvalue, yvalue, "text", color, angle, rotate, size,
        font, position);
run;

proc gplot data=ds;
  plot x*y / anno=annods;
run; quit;
```

**Note:** See 9.1 for an introduction to `annotate` datasets. The `label` macro draws the text given by `text` at `(xvalue, yvalue)`, though a character variable can also be specified if the quotes are omitted. The remainder of the parameters which define the text are generally self-explanatory, with the exception of `size`, which is a numeric value measured in terms of the size of the graphics area, and `position`, which specifies the location of the specified point relative to the printed text. A value of 5 centers the text on the specified point. Fonts available include SAS and system fonts; a default typical SAS font is `swiss`. SAS font information can be found in the on-line help: Contents; SAS Products; SAS/GRAFH; SAS/GRAFH Reference; Controlling the Appearance of Your Output; Specifying Fonts in SAS/GRAFH programs.

## R

```
text(x, y, labels)
```

**Note:** Each value of the character vector `labels` is displayed at the specified (X,Y) coordinate. The `adj` option can be used to change text justification to the left, center (default), or right of the coordinate. The `srt` option can be used to rotate text, while `cex` controls the size of the text. The `font` option to `par()` allows specification of plain, bold, italic, or bold italic fonts (see the `family` option to specify the name of a different font family).

## 9.1.12 Mathematical symbols

*Example: 3.4.1*

In SAS, mathematical symbols can be plotted using the text plotting method described in 9.1.11, specifying a font containing math symbols. These can be found in the documentation: Contents; SAS Products; SAS/GRAFH; Concepts; Fonts. Useful fonts include the `math` and `greek` fonts. Putting equations with subscripts and superscripts into a plot, or mixing fonts, can be very time consuming.

## R

```
plot(x, y)
text(x, y, expression(mathexpression))
```

**Note:** The `expression()` argument can be used to embed mathematical expressions and symbols (e.g.,  $\mu = 0$ ,  $\sigma^2 = 4$ ) in graphical displays as text, axis labels, legends, or titles. See `help(plotmath)` for more details on the form of `mathexpression` and `example(plotmath)` for examples.

## 9.1.13 Arrows and shapes

*Examples: 5.7.4 and 8.7.6*

### SAS

```
%annomac;
data annods;
  %system(2,2,3);
  %arrow(xvalue_1, yvalue_1, xvalue_2, yvalue_2, color,
         linetype, size, angle, font);
  %rect(xvalue_1, yvalue_1, xvalue_2, yvalue_2, color, linetype,
        size);
run;

proc gplot data=ds;
  plot x*y / anno=annods;
run; quit;
```

*Note:* See 9.1 for an introduction to `annotate` datasets. The `arrow` macro draws an arrow from `(xvalue_1, yvalue_1)` to `(xvalue_2, yvalue_2)`. The `size` is a numeric value measured in terms of the size of the graphics area. The `rect` macro draws a rectangle with opposite corners at `(xvalue_1, yvalue_1)` and `(xvalue_2, yvalue_2)`. The type of line drawn is determined by the value of `linetype`, as discussed in 9.2.11, and the color is determined by the value of `color`, as discussed in 9.2.13.

## R

```
arrows(x, y)
rect(xleft, ybottom, xright, ytop)
polygon(x, y)

library(plotrix)
draw.circle(x, y, r)
```

*Note:* The `arrows()`, `rect()`, and `polygon()` functions take vectors as arguments and create the appropriate object with vertices specified by each element of those vectors.

### 9.1.14 Add grid

A rectangular grid can sometimes be helpful to add to an existing plot.

#### SAS

```
proc gplot data=ds;
  plot x*y / autohref autovref;
run; quit;
or
proc sgplot data=ds;
  scatter x=x1 y=x2;
  xaxis grid;
  yaxis grid;
run; quit;
```

*Note:* Omitting one or the other option will leave only horizontal or vertical reference lines.

## R

```
grid(nx=num, ny=num)
```

*Note:* The `nx` and `ny` options control the number of cells in the grid. If they are specified as `NULL`, the grid aligns with the tick marks. The default shading is light gray, with a dotted line. Further support for complex grids is available within the `grid.lines()` function in the `grid` package.

### 9.1.15 Legend

*Examples:* 3.4.1 and 5.7.4

**SAS**

```
legend1 mode=share position=(bottom right inside)
  across=ncols frame label=("Legend Title" h=3) value=("Grp1" "Grp2");

proc gplot data=ds;
  plot y*x=group / legend=legend1;
run;
```

*Note:* The `legend` statement controls all aspects of how the legend will look and where it will be placed. Legends can be attached to many graphics in a manner similar to that demonstrated here for `proc gplot`. Here we show the most commonly used options. An

example of using the `legend` statement can be found in Figure 6.1. The `mode` option determines whether the legend shares the graphic output region with the graphic (shown above); other options reserve space or prevent other plot elements from interfering with the graphic. The `position` option places the legend within the plot area. The `across` option specifies the number of columns in the legend. The `frame` option draws a box around the legend. The `label` option describes the text of the legend title, while the `value` option describes the text printed with legend items. Fuller description of the legend statement is provided in the on-line help: Contents; SAS Products; SAS/GRAFH; Statements; LEGEND.

**R**

```
plot(x, y)
legend(xval, yval, legend=c("Grp1", "Grp2"), lty=1:2, col=3:4)
```

*Note:* The `legend()` command can be used to add a legend at the location (`xval`, `yval`) to distinguish groups on a display. Line styles (9.2.11) and colors (9.2.13) can be used to distinguish the groups. A vector of legend labels, line types, and colors can be specified using the `legend`, `lty`, and `col` options, respectively.

### 9.1.16 Identifying and locating points

**SAS**

```
symbol1 pointlabel="#label";
proc gplot data=ds;
  plot y*x;
run;
quit;

data newds;
set ds;
  label = 'alt=' || x || "," || y;
run;

ods html;
proc gplot data=newds;
  plot y*x / html=label;
run; quit;
ods html close;
```

*Note:* The former code will print the values of the variable `label` on the plot. The variable `label` must appear in the dataset used in the `proc gplot` statement. Note that this can result in messy plots, and it is advisable when there are many observations to choose or create a `label` variable with mostly missing values.

The latter code will make the value of *X* and *Y* appear when the mouse hovers over a plotted data point, as long as the HTML output destination is used. Any text or variable value can be displayed in place of the value of `label`, which in the above entry specifies the observed values of *x* and *y*.

**R**

```
locator(n)
```

*Note:* The `locator()` function identifies the position of the cursor when the mouse button is pressed. An optional argument `n` specifies how many values to return. The `identify()` function works in the same fashion, but returns the point closest to the cursor.

## 9.2 Options and parameters

Many options can be given to plots. In many SAS procedures, these are implemented using `goptions`, `symbol`, `axis`, `legend`, or other statements. Details on these statements can be found in the on-line help: Contents; SAS Products; SAS/GRAFH; Statements.

In R, many options are arguments to `plot()`, `par()`, or other high-level functions. Many of these options are described in the documentation for the `par()` function.

### 9.2.1 Graph size

#### SAS

```
goptions hsize=Xin vsize=Yin;
```

or

```
ods graphics width=Xin height=Yin;
```

*Note:* The size in `goptions` can be specified as above in inches (`in`) or as centimeters (`cm`). The size in the `ods graphics` statement can also be specified as (`cm`), millimeters (`mm`), standard typesetting dimensions (`em`, `en`), or printer's points (`pt`).

#### R

```
pdf("filename.pdf", width=Xin, height=Yin)
```

*Note:* The graph size is specified as an optional argument when starting a graphics device (e.g., `pdf()`, 9.3.1), with arguments `Xin` and `Yin` given in inches.

### 9.2.2 Grid of plots per page

*Example:* 6.6.3

See also 9.2.3.

#### SAS

```
proc gplot ... gout=myplots;
  plot ...;
run; quit;

proc gchart ... gout=myplots;
  vbar ...;
run; quit;

proc greplay gout=myplots igout=myplots nofs
  tc=sashelp.templt template=h2;
  device win;
  treplay 1:gplot 2:gchart;
quit;
or
proc sgpanel data=ds;
  panelby catvar / layout=rowlattice rows=n;
  series x=x y=y;
run; quit;
```

*Note:* In the first set of code, we show a general approach based on SAS/GRAFH. There, we save each plotted figure into a graphics library (`myplots`), then “replay” them using `proc greplay`. The `template` describes how the graphics area will be divided. SAS provides several templates, descriptions of which can be displayed in the log using `proc greplay nofs`

`tc=sashelp.templt; list tc; run;`. Custom templates can also be designed; for information on this, see the on-line help: Contents; SAS Products; SAS Procedures; GREPLAY.

In the second set of code, we show how `proc sgpanel` can be used to make a regular grid of similar plots, one for each value of some variable(s). These are listed in the `panelby` statement. In the example, a time series plot is displayed, but any other graphic that `proc sgpanel` will make can be displayed.

#### R

```
par(mfrow=c(a, b))
or
par(mfcol=c(a, b))
```

*Note:* The `mfrow` option specifies that plots will be drawn in an  $a \times b$  array by row (by column for `mfcol`).

### 9.2.3 More general page layouts

*Example:* 8.7.3

See also 9.2.2.

#### SAS

In SAS, more general layouts can be generated using the `proc greplay` approach discussed in 9.2.2, where in this case it is likely that a custom template will be required, rather than one provided with the program. The `greplay` procedure can be used to write custom templates as well as to replay plots. The syntax is too cumbersome to describe here, but examples can be found in the on-line help for `proc greplay`: Contents; SAS Products; SAS Procedures; GREPLAY. Regular grids of plots are discussed in 9.2.2. Scatterplot matrices (8.4.1) can be generated using `proc sgscatter`, and conditioning plots (8.4.2) can be made using `proc sgpanel`.

#### R

```
oldpar = par(no.readonly=TRUE)
layout(mat, widths=wvec, heights=hvec)
layout.show()
#fill the layout with plots
par(oldpar)
```

*Note:* The `layout()` function divides the graphics device into rows and columns, the relative sizes of which are specified by the `widths` and `heights` vectors. The number of rows and columns, plus the locations in the matrix to which the figures will be plotted, are specified by `mat`. The elements of the `mat` matrix are integers showing the order in which generated plots fill the cells. Larger and smaller figures can be plotted by repeating some integer values. The `layout.show(n)` function plots the outline of the next `n` figures. Other options to arrange plots on a device include `par(mfrow)`, for regular grids of plots, and `split.screen()`. The `lattice` package provides a different implementation to multiple plots (see the `position` option).

## 9.2.4 Fonts

### SAS

```
proc fontreg mode=all;
  fontpath "c:/windows/fonts";
run;

goptions ftext="Comic Sans MS";

title font="Lucida Handwriting" "My font titles";

axis1 label=(font="Goudy Old Style" h=2 "The first axis")
      value = ( h = 2 font= "Comic Sans MS");

symbol2 font="Agency FB/bo" v='C' h=.7 c=black;
```

*Note:* The `fontreg` procedure only needs to be run once and allows the use of any TrueType or Adobe Type 1 font installed on the computer. (Results may differ by OS.)

The `goptions` syntax will make the specified font the default on following SAS/Graph graphics. The remaining statements show how to specify different fonts for particular aspects of the graphic.

### R

```
pdf(file="plot.pdf")
par(family="Palatino")
plot(x, y)
dev.off()
```

*Note:* Supported postscript families include `AvantGarde`, `Bookman`, `Courier`, `Helvetica`, `Helvetica-Narrow`, `NewCenturySchoolbook`, `Palatino`, and `Times` (see `?postscript`).

## 9.2.5 Point and text size

*Example:* 6.6.5

### SAS

```
goptions htext=Xin;
title 'titletext' h=Xin;
axis label = ('labeltext' h=Xin);
axis value = ('valuetext' h=Yin);
```

*Note:* For many graphics statements which produce text, the `h` option controls the size of the printed characters. The default metric is graphic cells, but absolute values in inches and centimeters can also be used as in the `axis` statements shown. The `htext` option to the `goptions` statement affects all text in graphic output unless changed for a specific graphic element.

### R

```
plot(x, y, cex=cexval)
```

*Note:* The `cex` options specifies how much the plotting text and symbols should be magnified relative to the default value of 1 (see `help(par)` for details on how to specify this for axes, labels, and titles, e.g., `cex.axis`).

## 9.2.6 Box around plots

*Example:* 5.7.4

In SAS, some graphics-generating statements accept a `frame` or a (default) `noframe` option, which will draw or prevent drawing a box around the plot.

**R**

```
plot(x, y, bty=btyval)
```

*Note:* Control for the box around the plot can be specified using `btyval`, where if the argument is one of `o` (the default), `l`, `7`, `c`, `u`, or `]`, the resulting box resembles the corresponding character, while a value of `n` suppresses the box.

**9.2.7 Size of margins**

*Example: 6.6.3*

Within SAS, the margin options define the printable area of the page for graphics and text. For R, these control how tight plots are to the printable area.

**SAS**

```
options bottommargin=3in topmargin=4cm leftmargin=1 rightmargin=1;
```

*Note:* The default units are inches; a trailing `cm` indicates centimeters, while a trailing `in` makes inches the explicit metric.

**R**

```
par(mar=c(bot, left, top, right),    # inner margin
     oma=c(bot, left, top, right))  # outer margin
```

*Note:* The vector given to `mar` specifies the number of lines of margin around a plot: the default is `c(5, 4, 4, 2) + 0.1`. The `oma` option specifies additional lines outside the entire plotting area (the default is `c(0,0,0,0)`). Other options to control margin spacing include `omd` and `omi`.

**9.2.8 Graphical settings**

*Example: 6.6.3*

**SAS**

```
goptions reset=all;
```

*Note:* Many graphical settings are specified using the `goptions` statement. The above usage will revert all values to the SAS defaults.

**R**

```
# change values, while saving old
oldvalues = par(...)

# restore old values for graphical settings
par(oldvalues)
```

**9.2.9 Axis range and style**

*Examples: 6.6.1 and 8.7.1*

**SAS**

```
axis1 order = (x1, x2 to x3 by x4, x5);
axis2 order = ("value1" "value2" ... "valuен");
```

*Note:* Axis statements are associated with vertical or horizontal axes using `vaxis` or `haxis` options in various procedures. For an example, see Figure 8.1 in 8.7.1. Multiple options to the `axis` statement can be listed, as in Figure 6.1.

**R**

```
plot(x, y, xlim=c(minx, maxx), ylim=c(miny, maxy), xaxs="i", yaxs="i")
```

*Note:* The `xaxs` and `yaxs` options control whether tick marks extend beyond the limits of the plotted observations (default) or are constrained to be internal ("i"). More control is available through the `axis()` and `mtext()` functions.

### 9.2.10 Axis labels, values, and tick marks

SAS

*Example: 3.4.1*

```
axis1 label=("Text for axis label" angle=90 color=red
            font=swiss height=2 justify=right rotate=180);
axis1 value=("label1" "label2")
axis1 major=(color=blue height=1.5cm width=2);
axis1 minor=none;
```

*Note:* Axis statements are associated with vertical or horizontal axes using `vaxis` or `haxis` options in various procedures. For example, in `proc gplot`, one might use a `plot y*x / vaxis=axis1 haxis=axis2` statement. Multiple options to the `axis` statement can be listed, as in Figure 6.1.

In the `label` option above we show the text options available for graphics which apply to both `legend` and `axis` statements, and to `title` statements when graphics are produced. The `angle` option specifies the angle of the line along which the text is printed; the default depends on which `axis` is described. The `color` and `font` options are discussed in 9.2.13 and 9.1.11, respectively. The `height` option specifies the text size; it is measured in graphic cells, but can be specified with the number of units, for example, `height=1cm`. The `justify` option can take values of `left`, `center`, or `right`. The `rotate` option rotates each character in place. The `value` option describes the text which labels the tick marks, and takes the same parameters described for the `label` option.

The `major` and `minor` options take the same parameters; `none` will omit either labeled (`major`) or unlabeled (`minor`) tick marks. The `width` option specifies the thickness of the tick in multiples of the default.

R

```
plot(x, y, lab=c(x, y, len), # number of tick marks
      las=lasval,      # orientation of tick marks
      tck=tckval,      # length of tick marks
      tcl=tclval,      # length of tick marks
      xaxp=c(x1, x2, n),   # coordinates of the extreme tick marks
      yaxp=c(x1, x2, n),   # coordinates of the extreme tick marks
      xlab="X axis label", ylab="Y axis label")
```

*Note:* Options for `las` include 0 for always parallel, 1 for always horizontal, 2 for perpendicular, and 3 for vertical.

### 9.2.11 Line styles

SAS

*Example: 6.6.3*

```
symbol1 interpol=itype line=ltyval;
```

*Note:* The `interpol` option to the `symbol` statement, which can be shortened to simply `i`, specifies what kind of line should be plotted through the data. Options include smoothers, step functions, linear regressions, and more. The `line` option (which can be shortened to `l`) specifies a solid line (by default, `ltyval=1`) or various dashed or dotted lines (`ltyval 2 ... 33`). A list of line types with associated code can be found in the on-line documentation: Contents; SAS Products; SAS/GRAPH; Statements; SYMBOL. The line types do not have a separate entry, but appear near the end of the long description of the `symbol` statement.

R

```
plot(...)
lines(x, y, lty=ltyval)
```

*Note:* Supported line type values include 0=blank, 1=solid (default), 2=dashed, 3=dotted, 4=dotdash, 5=longdash, and 6=twodash.

### 9.2.12 Line widths

**SAS**

*Example: 3.4.1*

```
symbol interpol=interpol_type width=lwdval;
```

*Note:* When a line through the data is requested using the `interpol` option, the thickness of the line, in multiples of the default thickness, can be specified by the `width` option, for which `w` is a synonym. The default thickness depends on display hardware.

**R**

```
plot(...)  
lines(x, y, lwd=lwdval)
```

*Note:* The default for `lwd` is 1; the value of `lwdval` must be positive.

### 9.2.13 Colors

**SAS**

*Example: 5.7.4*

```
symbol1 c=colval cl=colval cv=colval;  
axis1 label=(color=colval);
```

*Note:* Colors in SAS can be specified in a variety of ways. Some typical examples of applying colors are shown, but many features of plots can be colored. If precise control is required, `colval` can be specified using a variety of schemes, as described in the on-line documentation: Contents; SAS Products; SAS/GRAFH; SAS/GRAFH Reference; Controlling the Appearance of Your Output; Using Colors in SAS/GRAFH Programs. For more casual choice of colors, color names such as `blue`, `black`, `red`, `purple`, `strongblue`, or `lightred` can be used.

**R**

```
plot(...)  
lines(x, y, col=colval)
```

*Note:* For more information on setting colors, see the `Color Specification` section within `help(par)`. The `colors()` function lists available colors, while the `colors.plot()` function within the `epitools` package displays a matrix of colors, and `colors.matrix()` returns a matrix of color names. The `display.brewer.all()` function within the `RColorBrewer` package is particularly useful for selecting a set of complementary colors for a palette.

### 9.2.14 Log scale

**SAS**

```
axis1 logbase=base logstyle=expand;
```

*Note:* The `logbase` option scales the axis according to the log of the specified base; valid base values include `e`, `pi`, or a number. The `logstyle` option produces plots with tick marks labeled with the value of the base (`logstyle=power`) or the base raised to that value (`logstyle=expand`).

**R**

```
plot(x, y, log=logval)
```

*Note:* A natural log scale can be specified using the `log` option to `plot()`, where `log="x"` denotes only the `x` axis, `"y"` only the `y` axis, and `"xy"` for both.

### 9.2.15 Omit axes

**SAS**

```
axis1 style=0 major=none minor=none label="" value=none;
```

*Note:* To remove an axis entirely in SAS, it is necessary to request each element of the axis not be drawn, as shown here.

**R**

```
plot(x, y, xaxt="n", yaxt="n")
```

## 9.3 Saving graphs

It is straightforward to export graphics in a variety of formats. In addition to what is described below, RStudio allows export of a plot in multiple formats along with full control over the size and resolution.

### 9.3.1 PDF

**SAS**

```
ods pdf file="filename.pdf";
proc gplot data=ds;
  ...
ods pdf close;
or
filename filehandle "filename.pdf";
goptions gsfname=filehandle device=pdf gsfmode=replace;

proc gplot data=ds;
  ...
run;
```

*Note:* In both versions above, the `filename` can include a directory location as well as a name. The `device` option specifies formatting of the graphic; the many valid options can be viewed using `proc gdevice`, and key options are presented in this section. The `gsfmode=replace` option allows SAS to create and/or overwrite the graphic. The `filehandle` may not be more than eight characters long.

The `ods pdf` statement will place graphics and text output from procedures into the pdf file generated.

**R**

```
pdf("file.pdf")
plot(...)
dev.off()
```

*Note:* The `dev.off()` function is used to close a graphics device.

### 9.3.2 Postscript

**SAS**

```
ods ps file="filename.ps";
proc gplot data=ds;
  ...
run;
ods ps close;
```

```
or
filename filehandle "filename.ps";
goptions gsfname=filehandle device=ps gsfmode=replace;

proc gplot data=ds;
  ...
run;
```

*Note:* In both versions above, the `filename` can include a directory location as well as a name. The `device` option specifies formatting of the graphic; the many valid options can be viewed using `proc gdevice`, and key options are presented in this section. The `gsfmode=replace` option allows SAS to create and/or overwrite the graphic. The `filehandle` may not be more than eight characters long.

The `ods ps` statement will place graphics and text output from procedures into the file generated.

## R

```
postscript("file.ps")
plot(...)
dev.off()
```

*Note:* The `dev.off()` function is used to close a graphics device.

### 9.3.3 RTF

The Rich Text Format (RTF) is a file format developed for cross-platform document sharing. Many word processors are able to read and write RTF documents.

The following SAS commands will create a file in this format containing the graphic; any text generated by procedures will also appear in the RTF file if they are executed between the `ods rtf` and `ods rtf close` statements.

#### SAS

```
ods rtf file="filename.rtf";
proc gplot data=ds;
  ...
run;
ods rtf close;
```

*Note:* The `filename` can include a directory location as well as a name.

#### R

```
library(rtf)
output = "file.doc"
rtf = RTF(output)
addParagraph(rtf, "This is a plot.\n")
addPlot(rtf, plot.fun=plot, width=6, height=6, res=300, x, y)
done(rtf)
```

*Note:* This example adds text and the equivalent of `plot(x, y)` to an RTF file. The `rtf` package vignette provides additional details for formatting graphs and tables.

### 9.3.4 JPEG

#### SAS

```
filename filehandle "filename.jpg";
goptions gsfname=filehandle device=jpeg gsfmode=replace;

proc gplot data=ds;
  ...
run;
```

*Note:* The `filename` can include a directory location as well as a name. The `device` option specifies formatting of the graphic; valid options can be viewed using `proc gdevice`. The `gsfmode=replace` option allows SAS to create and/or overwrite the graphic. The `filehandle` may not be more than eight characters long.

#### R

```
jpeg("filename.jpg")
plot(...)
dev.off()
```

*Note:* The `dev.off()` function is used to close a graphics device.

### 9.3.5 Windows Metafile (WMF)

#### SAS

```
filename filehandle "filename.wmf";
goptions gsfname=filehandle device=wmf gsfmode=replace;

proc gplot data=ds;
  ...
run;
```

*Note:* The `filename` can include a directory location as well as a name. The `device` option specifies formatting of the graphic; valid options can be viewed using `proc gdevice`. The `gsfmode=replace` option allows SAS to create and/or overwrite the graphic. The `filehandle` may not be more than eight characters long.

#### R

```
win.metafile("file.wmf")
plot(...)
dev.off()
```

*Note:* The function `win.metafile()` is only supported under Windows. Functions which generate multiple plots are not supported. The `dev.off()` function is used to close a graphics device.

### 9.3.6 Bitmap image file (BMP)

#### SAS

```
filename filehandle "filename.bmp";
goptions gsfname=filehandle device=bmp gsfmode=replace;

proc gplot data=ds;
  ...
run;
```

*Note:* The `filename` can include a directory location as well as a name. The `device` option specifies formatting of the graphic; valid options can be viewed using `proc gdevice`. The `gsfmode=replace` option allows SAS to create and/or overwrite the graphic. The `filehandle` may not be more than eight characters long.

**R**

```
bmp("filename.bmp")
plot(...)
dev.off()
```

*Note:* The `dev.off()` function is used to close a graphics device.

### 9.3.7 Tagged image file format (TIFF)

**SAS**

```
filename filehandle "filename.tif";
goptions gsfname=filehandle device=tiffp300 gsfmode=replace;

proc gplot data=ds;
  ...
run;
```

*Note:* The `filename` can include a directory location as well as a name. The `device` option specifies formatting of the graphic; valid options can be viewed using `proc gdevice`. The `gsfmode=replace` option allows SAS to create and/or overwrite the graphic. The `filehandle` may not be more than eight characters long. Many types of TIFF can be generated; the above `device` specifies a color plot with 300 dpi.

**R**

```
tiff("filename.tiff")
plot(...)
dev.off()
```

*Note:* The `dev.off()` function is used to close a graphics device.

### 9.3.8 Portable Network Graphics (PNG)

**SAS**

```
filename filehandle "filename.png";
goptions gsfname=filehandle device=png gsfmode=replace;

proc gplot data=ds;
  ...
run;
```

*Note:* The `filename` can include a directory location as well as a name. The `device` option specifies formatting of the graphic; valid options can be viewed using `proc gdevice`. The `gsfmode=replace` option allows SAS to create and/or overwrite the graphic. The `filehandle` may not be more than eight characters long.

The `ODS` graphics system works by creating a PNG file which is stored in the current directory or the directory and then creating output in the desired format. Using the `ods output` statement for the formatted output options in this section will also result in a PNG file.

**R**

```
png("filename.png")
plot(...)
dev.off()
```

*Note:* The `dev.off()` function is used to close a graphics device.

### 9.3.9 Closing a graphic device

*Example:* 8.7.4

There is no analog in SAS for this concept. In R, the following code closes a graphics window. This is particularly useful when a graphics file is being created.

**R**

```
dev.off()
```

# Chapter 10

## Simulation

Simulations provide a powerful way to answer questions and explore properties of statistical estimators and procedures. In this chapter, we will explore how to simulate data in a variety of common settings and apply some of the techniques introduced earlier.

### 10.1 Generating data

#### 10.1.1 Generate categorical data

Simulation of data from continuous probability distributions is straightforward using the functions detailed in 3.1.1. Simulating from categorical distributions can be done manually or using some available functions.

```
data test;
p1 = .1; p2 = .2; p3 = .3;
do i = 1 to 10000;
  x = uniform(0);
  mycat1 = (x ge 0) + (x gt p1) + (x gt p1 + p2)
            + (x gt p1 + p2 + p3);
  mycat2 = rantbl(0, .5, .4, .05);
  mycat3 = rand("TABLE", .3, .3, .4);
  output;
end;
run;
```

```
proc freq data=test;
  tables mycat1 mycat2 mycat3;
run;
```

The FREQ Procedure

			Cumulative Frequency	Cumulative Percent
mycat1	Frequency	Percent		
1	1014	10.14	1014	10.14
2	1966	19.66	2980	29.80
3	2950	29.50	5930	59.30
4	4070	40.70	10000	100.00

		Cumulative Frequency	Cumulative Percent
mycat2	Frequency	Percent	
1	5003	50.03	5003
2	4064	40.64	9067
3	481	4.81	9548
4	452	4.52	10000

		Cumulative Frequency	Cumulative Percent
mycat3	Frequency	Percent	
1	2907	29.07	2907
2	3017	30.17	5924
3	4076	40.76	10000

The first argument to the `rantbl` function is the seed. The remaining arguments are the probabilities for the categories; if they sum to more than 1, the excess is ignored. If they sum to less than 1, the remainder is used for another category. The same is true for `rand("TABLE", ...)`.

```
> options(digits=3)
> options(width=72) # narrow output
> p = c(.1,.2,.3)
> x = runif(10000)
> mycat1 = numeric(10000)
> for (i in 0:length(p)) {
+   mycat1 = mycat1 + (x >= sum(p[0:i]))
+ }
> table(mycat1)

mycat1
  1    2    3    4
984 2035 3030 3951
```

```
> mycat2 = cut(runif(10000), c(0, 0.1, 0.3, 0.6, 1))
> summary(mycat2)

(0,0.1] (0.1,0.3] (0.3,0.6] (0.6,1]
989      1999      3002      4010

> mycat3 = sample(1:4, 10000, rep=TRUE, prob=c(.1,.2,.3,.4))
> table(mycat3)

mycat3
 1   2   3   4
1013 2064 2970 3953
```

The `cut()` function (2.2.4) bins continuous data into categories with both endpoints defined by the arguments. Note that the `min()` and `max()` functions can be particularly useful here in the outer categories. The `sample()` function as shown treats the values 1,2,3,4 as a dataset and samples from the dataset 10,000 times with the probability of selection defined in the `prob` vector.

### 10.1.2 Generate data from a logistic regression

Here we show how to simulate data from a logistic regression (7.1.1). Our process is to generate the linear predictor, then apply the inverse link, and finally draw from a distribution with this parameter. This approach is useful in that it can easily be applied to other generalized linear models (7.1). Here we make the intercept  $-1$ , the slope  $0.5$ , and generate 5,000 observations.

```
data test;
intercept = -1;
beta = .5;
do i = 1 to 5000;
  xtest = normal(12345);
  linpred = intercept + (xtest * beta);
  prob = exp(linpred)/ (1 + exp(linpred));
  ytest = uniform(0) lt prob;
  output;
end;
run;
```

Sometimes the voluminous SAS output can be useful, but here we just want to demonstrate that the parameter estimates are more or less accurate. The `ODS` system provides a way to choose only specific output elements.

```
ods select parameterestimates;
proc logistic data=test;
  model ytest(event='1') = xtest;
run;
ods select all;
```

```
The LOGISTIC Procedure
Analysis of Maximum Likelihood Estimates
```

Parameter	DF	Estimate	Standard Error	Wald	
				Chi-Square	Pr > ChiSq
Intercept	1	-1.0925	0.0338	1047.8259	<.0001
xtest	1	0.4978	0.0346	207.1199	<.0001

```
> intercept = -1
> beta = 0.5
> n = 5000
> xtest = rnorm(n, mean=1, sd=1)
> linpred = intercept + (xtest * beta)
> prob = exp(linpred)/(1 + exp(linpred))
> ytest = ifelse(runif(n) < prob, 1, 0)
```

While the `summary()` of a `glm` object is more concise than the default SAS output, we can display just the estimated values of the coefficients from the logistic regression model using the `coef()` function (see 6.4.1).

```
> coef(glm(ytest ~ xtest, family=binomial))

(Intercept)      xtest
-1.018        0.479
```

### 10.1.3 Generate data from a generalized linear mixed model

In this example, we generate data from a generalized linear mixed model (7.4.7) with a dichotomous outcome. We generate 1500 clusters, denoted by `id`. There is one predictor with a common value for all observations in a cluster ( $X_1$ ). Each observation within the cluster has an order indicator (denoted by  $X_2$ ) which has a linear effect (`beta_2`), and there is an additional predictor which varies among observations ( $X_3$ ). The dichotomous outcome  $Y$  is generated from these predictors using a logistic link incorporating a normal distributed random intercept for each cluster.

```
data sim;
  sigbsq=4; beta0=-2; beta1=1.5; beta2=0.5; beta3=-1; n=1500;
  do i = 1 to n;
    x1 = (i lt (n+1)/2);
    randint = normal(0) * sqrt(sigbsq);
    do x2 = 1 to 3 by 1;
      x3 = uniform(0);
      linpred = beta0 + beta1*x1 + beta2*x2 + beta3*x3 + randint;
      expit = exp(linpred)/(1 + exp(linpred));
      y = (uniform(0) lt expit);
      output;
    end;
  end;
run;
```

This model can be fit using `proc nlmixed` (7.4.6) or `proc glimmix` (7.4.7). For large datasets, `proc nlmixed` (which uses numerical approximation to calculate the integral) can take a prohibitively long time to fit, and convergence can sometimes be problematic.

```

options ls=64;
ods select parameterestimates;

proc nlmixed data=sim qpoints=50;
parms b0=1 b1=1 b2=1 b3=1;
eta = b0 + b1*x1 + b2*x2 + b3*x3 + bi1;
mu = exp(eta)/(1 + exp(eta));
model y ~ binary(mu);
random bi1 ~ normal(0, g11) subject=i;
predict mu out=predmean;
run;
ods select all;

```

The NLMIXED Procedure

#### Parameter Estimates

Standard						
Parameter	Estimate	Error	DF	t Value	Pr >  t	Alpha
b0	-1.9946	0.1688	1499	-11.82	<.0001	0.05
b1	1.4866	0.1418	1499	10.49	<.0001	0.05
b2	0.5358	0.05153	1499	10.40	<.0001	0.05
b3	-0.9630	0.1596	1499	-6.04	<.0001	0.05
g11	4.1258	0.4064	1499	10.15	<.0001	0.05

#### Parameter Estimates

Parameter	Lower	Upper	Gradient
b0	-2.3257	-1.6636	-0.00007
b1	1.2085	1.7648	0.000043
b2	0.4347	0.6369	-0.00006
b3	-1.2760	-0.6500	-4.63E-7
g11	3.3286	4.9231	-0.00001

On the other hand, `proc glimmix` frequently fails to reach convergence using the default maximization technique. We show below how to use a maximization technique that is often effective. We also show how to implement the Laplace approximation for the likelihood. This has better properties than the default pseudo-likelihood technique, but is not available for some more complex models.

```

ods select parameterestimates covparms;

proc glimmix data=sim order=data method=laplace;
  nloptions maxiter=100 technique=dbldog;
  model y = x1 x2 x3 / solution dist=bin;
  random int / subject=i;
run;

ods select all;

```

The GLIMMIX Procedure  
Covariance Parameter Estimates

Standard			
Cov Parm	Subject	Estimate	Error
Intercept	i	3.3195	0.3317

#### Solutions for Fixed Effects

Standard					
Effect	Estimate	Error	DF	t Value	Pr >  t
Intercept	-1.9500	0.1634	1498	-11.93	<.0001
x1	1.4567	0.1332	2998	10.93	<.0001
x2	0.5197	0.05060	2998	10.27	<.0001
x3	-0.9317	0.1553	2998	-6.00	<.0001

Discrepancies between the two sets of estimates arise mainly from the differences between the numeric integration in `proc nlmixed` and the use of the Laplace approximation in `proc glimmix`.

The R simulation uses the approach introduced in 4.1.3 applied to a more complex setting, with each of the components built up part by part.

```

> n = 1500; p = 3; sigbsq = 4
> beta = c(-2, 1.5, 0.5, -1)
> id = rep(1:n, each=p)          # 1 1 ... 1 2 2 ... 2 ... n
> x1 = as.numeric(id < (n+1)/2) # 1 1 ... 1 0 0 ... 0
> randint = rep(rnorm(n, 0, sqrt(sigbsq)), each=p)
> x2 = rep(1:p, n)             # 1 2 ... p 1 2 ... p ...
> x3 = runif(p*n)
> linpred = beta[1] + beta[2]*x1 + beta[3]*x2 + beta[4]*x3 + randint
> expit = exp(linpred)/(1 + exp(linpred))
> y = runif(p*n) < expit      # generate a logical as our outcome

```

We fit the model using the `glmer()` function from the `lme4` package.

```

> library(lme4)
> glmmres = glmer(y ~ x1 + x2 + x3 + (1/id), family=binomial(link="logit"))
> summary(glmmres)

Generalized linear mixed model fit by maximum likelihood ['glmerMod']
Family: binomial ( logit )
Formula: y ~ x1 + x2 + x3 + (1 | id)

      AIC      BIC  logLik deviance
5323     5355    -2657      5313

Random effects:
 Groups Name        Variance Std.Dev.
 id     (Intercept) 2.6       1.61
 Number of obs: 4500, groups: id, 1500

Fixed effects:
            Estimate Std. Error z value Pr(>|z|)
(Intercept) -2.0118    0.1406  -14.3 < 2e-16 ***
x1          1.4084    0.1127   12.5 < 2e-16 ***
x2          0.4640    0.0451   10.3 < 2e-16 ***
x3         -0.8168    0.1409   -5.8 6.8e-09 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Correlation of Fixed Effects:
  (Intr) x1     x2
x1 -0.436
x2 -0.665  0.050
x3 -0.440 -0.052 -0.035

```

#### 10.1.4 Generate correlated binary data

Another way to generate correlated dichotomous outcomes  $Y_1$  and  $Y_2$  is based on the probabilities corresponding to the  $2 \times 2$  table. Given these cell probabilities, the variable probabilities can be expressed as a function of the marginal probabilities and the desired correlation, using the methods of Lipsitz and colleagues [107]. Here we generate a sample of 1000 values where:  $P(Y_1 = 1) = .15$ ,  $P(Y_2 = 1) = .25$ , and  $\text{Corr}(Y_1, Y_2) = 0.40$ .

```

data test;
p1=.15; p2=.25; corr=0.4;
p1p2=corr*sqrt(p1*(1-p1)*p2*(1-p2)) + p1*p2;
do i = 1 to 10000;
cat=rand('TABLE', 1-p1-p2+p1p2, p1-p1p2, p2-p1p2);
y1=0;
y2=0;
if cat=2 then y1=1;
else if cat=3 then y2=1;
else if cat=4 then do;
y1=1;
y2=1;
end;
output;
end;
run;

> p1 = .15; p2 = .25; corr = 0.4; n = 10000
> p1p2 = corr*sqrt(p1*(1-p1)*p2*(1-p2)) + p1*p2
> library(Hmisc)
> vals = rMultinom(matrix(c(1-p1-p2+p1p2, p1-p1p2, p2-p1p2, p1p2),
nrow=1, ncol=4), n)
> y1 = rep(0, n); y2 = rep(0, n)    # put zeroes everywhere
> y1[vals==2 | vals==4] = 1          # and replace them with ones
> y2[vals==3 | vals==4] = 1          # where needed
> rm(vals, p1, p2, p1p2, corr, n)   # cleanup

```

The generated data is close to the desired values.

```

options ls = 68;
proc corr data=test;
var y1 y2;
run;

```

#### The CORR Procedure

2 Variables:		y1	y2	
				Simple Statistics
Variable	N	Mean	Std Dev	Sum
y1	10000	0.14630	0.35342	1463
y2	10000	0.24550	0.43040	2455

#### Simple Statistics

Variable	Minimum	Maximum
y1	0	1.00000
y2	0	1.00000

Pearson Correlation Coefficients, N = 10000  
Prob > |r| under H0: Rho=0

	y1	y2
y1	1.00000	0.38451
		<.0001
y2	0.38451	1.00000
		<.0001

```
> cor(y1, y2)
[1] 0.412

> table(y1)

y1
 0    1
8476 1524

> table(y2)

y2
 0    1
7398 2602
```

### 10.1.5 Generate data from a Cox model

To simulate data from a Cox proportional hazards model (7.5.1), we need to model the hazard functions for both time to event and time to censoring. In this example, we use a constant baseline hazard, but this can be modified by specifying other `scale` parameters for the Weibull random variables.

```
data simcox;
  beta1 = 2;
  beta2 = -1;
  lambdaT = 0.002; *baseline hazard;
  lambdaC = 0.004; *censoring hazard;
  do i = 1 to 10000;
    x1 = normal(0);
    x2 = normal(0);
    linpred = exp(-beta1*x1 - beta2*x2);
    t = rand("WEIBULL", 1, lambdaT * linpred);
    * time of event;
    c = rand("WEIBULL", 1, lambdaC);
    * time of censoring;
    time = min(t, c);    * time of first?;
    censored = (c lt t); * 1 if censored;
    output;
  end;
run;
```

```
> # generate data from Cox model
> n = 10000
> beta1 = 2; beta2 = -1
> lambdaT = .002          # baseline hazard
> lambdaC = .004 # hazard of censoring
> x1 = rnorm(n)    # standard normal
> x2 = rnorm(n)
> # true event time
> T = rweibull(n, shape=1, scale=lambdaT*exp(-beta1*x1-beta2*x2))
> C = rweibull(n, shape=1, scale=lambdaC)  #censoring time
> time = pmin(T,C) #observed time is min of censored and true
> censored = (time==C) # set to 1 if event is censored
> # fit Cox model
> library(survival)
> survobj = coxph(Surv(time, (1-censored))~ x1 + x2, method="breslow")
```

These parameters generate data where approximately 40% of the observations are censored.

Note that `proc phreg` and `coxph()` expect different things: a censoring indicator and an observed event indicator, respectively. Here we made a censoring indicator in both simulations, though this leads to the somewhat awkward syntax shown in the `coxph()` function.

The `phreg` procedure (7.5.1) will describe the censoring patterns as well as the results of fitting the regression model.

```
options ls = 68;
ods select censoredsummary parameterestimates;
proc phreg data=simcox;
  model time*censored(1) = x1 x2;
run;
```

#### The PHREG Procedure

##### Summary of the Number of Event and Censored Values

			Percent
Total	Event	Censored	Censored
10000	5999	4001	40.01

##### Analysis of Maximum Likelihood Estimates

Parameter	DF	Estimate	Standard	Chi-Square	Pr > ChiSq	Hazard
						Ratio
x1	1	1.99434	0.02230	7997.1857	<.0001	7.347
x2	1	-0.98394	0.01563	3962.9682	<.0001	0.374

In R we tabulate the censoring indicator, then display the results as well as the associated 95% confidence intervals.

```

> table(censored)

censored
FALSE  TRUE
 5968  4032

> print(survobj)

Call:
coxph(formula = Surv(time, (1 - censored)) ~ x1 + x2, method = "breslow")

      coef exp(coef) se(coef)    z p
x1  2.006    7.433   0.0222 90.4 0
x2 -0.988    0.373   0.0157 -62.7 0

Likelihood ratio test=11490  on 2 df, p=0  n= 10000, number of events= 5968

> confint(survobj)

      2.5 % 97.5 %
x1  1.96  2.049
x2 -1.02 -0.957

```

The results are similar to the true parameter values.

### 10.1.6 Sampling from a challenging distribution

When the cumulative density function for a probability distribution can be inverted, it is simple to draw a sample from the distribution using the probability integral transform (3.1.10). However, when the form of the distribution is complex, this approach may be difficult or impossible. The Metropolis–Hastings algorithm [117] is a Markov Chain Monte Carlo (MCMC) method for obtaining samples from a variable with an arbitrary probability density function.

The MCMC algorithm uses a series of draws from a more common distribution, choosing at random which of these proposed draws to accept as draws from the target distribution. The probability of acceptance is calculated so that after the process has converged the accepted draws form a sample from the desired distribution. A further discussion can be found in Section 11.3 of *Probability and Statistics: The Science of Uncertainty* [37] or Section 1.9 of Gelman et al. [52].

Evans and Rosenthal [37] consider a distribution with probability density function:

$$f(y) = c \exp(-y^4)(1 + |y|)^3,$$

where  $c$  is a normalizing constant and  $y$  is defined on the whole real line.

We find the acceptance probability  $\alpha(x, y)$  in terms of two densities, the desired  $f(y)$  and  $q(x, y)$ , a proposal density. For the proposal density, we use the normal with mean equal to the previous value and unit variance, and find that

$$\begin{aligned} \alpha(x, y) &= \min \left\{ 1, \frac{f(y)q(y, x)}{f(x)q(x, y)} \right\} \\ &= \min \left\{ 1, \frac{c \exp(-y^4)(1 + |y|)^3(2\pi)^{-1/2} \exp(-(y-x)^2/2)}{c \exp(-x^4)(1 + |x|)^3(2\pi)^{-1/2} \exp(-(x-y)^2/2)} \right\} \end{aligned}$$

$$= \min \left\{ 1, \frac{\exp(-y^4 + x^4)(1 + |y|)^3}{(1 + |x|)^3} \right\}$$

To begin the process, we pick an arbitrary value for  $X_1$ . The Metropolis–Hastings algorithm chooses  $X_{n+1}$  as follows:

1. Generate  $y$  from a normal( $X_n$ , 1).
2. Compute  $\alpha(x, y)$  as above.
3. With probability  $\alpha(x, y)$ , let  $X_{n+1} = y$  (use proposal value). Otherwise, with probability  $1 - \alpha(x, y)$ , let  $X_{n+1} = X_n = x$  (repeat previous value).

The code below uses the first 5,000 iterations as a burn-in period, then generates 50,000 samples using this procedure. Only every 10th value from these 50,000 is saved, to reduce autocorrelation. This process is known as “thinning.”

```
data mh;
burnin=5000; numvals=5000; thin=10;
x = normal(0);
do i = 1 to (burnin + (numvals * thin));
  y = normal(0) + x;
  switchprob = min(1, exp(-y**4 + x**4) *
    (1 + abs(y))**3 * (1 + abs(x))**(-3));
  if uniform(0) lt switchprob then x = y;
  * if we don't change x, the previous value is kept
    in the data step by default-- no code needed;
  if (i gt burnin) and mod(i-burnin, thin) = 0 then output;
  * only start saving if we're past the burn-in period,
    then keep every 10th to thin;
end;
run;
```

In R, we begin by writing a function to calculate the acceptance probability.

```
> alphafun = function(x, y) {
  return(exp(-y^4+x^4)*(1+abs(y))^3*
    (1+abs(x))^(-3))
}
```

Generating the samples in R uses a `for()` loop.

```
> burnin=5000; numvals=5000; thin = 10
> xn = numeric(burnin + numvals*thin)
> xn[1] = rnorm(1)          # starting value
> for (i in 2:(burnin + numvals*thin)) {
  propy = rnorm(1, xn[i-1], 1)
  alpha = min(1, alphafun(xn[i-1], propy))
  xn[i] = sample(c(propy, xn[i-1]), 1, prob=c(alpha, 1-alpha))
}
> sample = xn[5000 + (1:numvals) * 10]
```

We can compare the true distribution to the empirical PDF estimated from the random draws. To do that, we need the normalizing constant  $c$ , which we calculate using R to integrate the distribution over the positive real line.

```
> f = function(x) {
  exp(-x^4)*(1+abs(x))^3
}
> integral = integrate(f, 0, Inf)
> c = 2 * integral$value; c

[1] 6.81
```

We find that  $c = 6.81$ .

In SAS, we'll get a kernel density estimate (8.1.5) of our draws from the distribution using proc kde.

```
ods select none;

proc kde data=mh;
  univar x / out=mhepdf;
run;
```

```
ods exclude none;
```

Then we can add in the calculated values of the true pdf using the formula.

```
data mh2;
set mhepdf;
  true = 6.809610784**(-1) * exp(-value**4) * (1 + abs(value))**3;
run;
```

Finally, we can plot them together.

```
legend1 position=(bottom center inside)
  across=1 noframe label=none value=(h=1.5);
axis1 label=(angle=90 "Density");
symbol1 i=j l=1 w=3 c=black;
symbol2 i=j l=21 w=3 c=black;
proc gplot data=mh2;
  plot (density true)*value / overlay vaxis = axis1 legend=legend1;
  label value="x" density="Simulated" true="True";
run;
```

The results are displayed in Figure 10.1, with the dashed line indicating the true distribution and the solid line the empirical PDF estimated from the simulated variates. The normalizing constant is used to plot density using the curve() function.

```
> pdfeval = function(x) {
  return(1/6.809610784*exp(-x^4)*(1+abs(x))^3)
}
> curve(pdfeval, from=-2, to=2, lwd=2, lty=2, type="l",
  ylab="probability", xlab="Y")
> lines(density(sample), lwd=2, lty=1)
> legend(-1, .15, legend=c("True", "Simulated"),
  lty=2:1, lwd=2, cex=1.8, bty="n")
```

Care is always needed when using MCMC methods. This example was particularly well behaved, in that the proposal distribution is large compared to the distance between the two modes. Section 6.2 of Lavine [96] and Gelman et al. [52] provide accessible discussions of dangers and diagnostics.

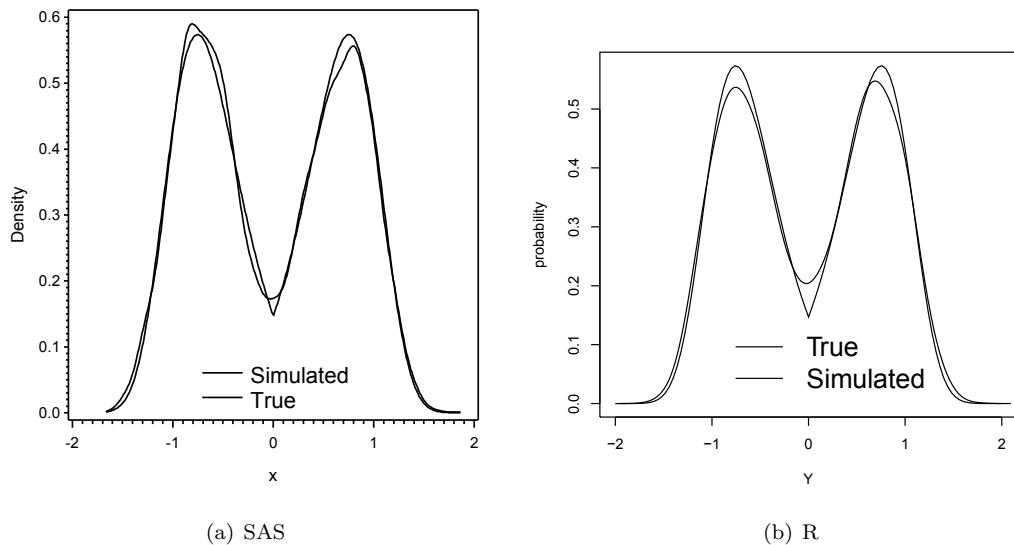


Figure 10.1: Plot of true and simulated distributions

## 10.2 Simulation applications

### 10.2.1 Simulation study of Student's $t$ test

A common rule of thumb is that the sampling distribution of the mean is approximately normal for samples of 30 or larger. Tim Hesterberg has argued ([home.comcast.net/~timhesterberg/articles/JSM08-n30.pdf](http://home.comcast.net/~timhesterberg/articles/JSM08-n30.pdf)) that the  $n \geq 30$  rule may need to be retired. He demonstrates that for sample sizes much larger than typically thought necessary, the one-sided error rate of the  $t$  test is not appropriately preserved. We explore this by generating  $n = 500$  random exponential variables with mean 1 and carrying out a one-sample Student's  $t$  test. We'll repeat this 1000 times and examine the coverage of the lower and upper confidence limits.

In SAS, we first make the dataset.

```
data test;
do sim = 1 to 1000;
  do i = 1 to 500;
    x = ranexp(42);
    output;
  end;
end;
run;
```

Next we perform the  $t$  test, saving the relevant output using the ODS system.

```
ods select none;
ods output conflimits=ttci;
proc ttest data=test h0=1 sides=2;
  by sim;
  var x;
run;
ods exclude none;
```

Then we process that dataset to determine whether the confidence interval failed to enclose the true value and report the proportion of times this was the case.

```
data summtt;
set ttci;
lower = (lowerclmean > 1);
upper = (upperclmean < 1);
run;

proc means data=summtt mean;
var lower upper;
run;
```

Variable	Mean
lower	0.0180000
upper	0.0410000

In R we make the data and perform the test in a `for()` loop. The `foreach` package could be used to speed up the computations if multiple cores are available.

```
> set.seed(42)
> numsim = 1000; n=500
> lower = numeric(numsim); upper = numeric(numsim)
> for (i in 1:numsim) {
  x = rexp(n, rate=1) # skewed
  testresult = t.test(x, mu=1)
  lower[i] = testresult$conf.int[1] > 1
  upper[i] = testresult$conf.int[2] < 1
}
```

We use the `tally()` function (5.1.7) to display the proportions of times that the true parameter is not captured.

```
> library(mosaic)
> tally(~ lower, format="percent")

  0      1 Total
97.8    2.2 100.0

> tally(~ upper, format="percent")

  0      1 Total
97.3    2.7 100.0
```

We observe that the test is rejecting more than it should on the upper end of the interval (though the overall two-sided alpha level is preserved).

An alternative approach would be to generate a matrix of random exponentials and then use `apply()` to process the computations without a `for` loop (results not shown):

```
> runttest = function(x) {return(CI=confint(t.test(x, mu=1)))}
> xmat = matrix(rexp(numsim*n), nrow=n)
> results = apply(xmat, 2, runttest)
> tally(~ results[2,> 1])
> tally(~ results[3,< 1])
```

### 10.2.2 Diploma (or hat-check) problem

Smith College is a selective women's liberal arts college in Northampton, MA. One tradition acquired since the college was founded in 1871 is to give every graduating student a diploma at random (or more accurately, in a haphazard fashion). At the end of the ceremony, a circle is formed, and students repeatedly pass the diplomas to the person next to them, stepping out once they've received their own diploma. This problem, also known as the *hat-check* problem, is featured in Mosteller[121]. Variants provide great fodder for probability courses.

The analytic solution for the expected number of students who receive their diplomas in the initial disbursement is very straightforward. Let  $X_i$  be the event that the  $i$ th student receives their diploma.  $E[X_i] = 1/n$  for all  $i$ , since the diplomas are assumed uniformly distributed. If  $T$  is defined as the sum of all of the events  $X_1$  through  $X_n$ ,  $E[T] = n*1/n = 1$  by the rules of expectations. It is sometimes surprising to students that this result does not depend on  $n$ . The variance is trickier, since the outcomes are not independent (if one student receives their diploma, the probability that others will increases).

Simulation-based problem solving is increasingly common as a means to complement and enhance analytic solutions [119, 71]. Here we simulate the number of students who receive their diploma and calculate the variance of that quantity. We simulate  $n = 650$  students and repeat the experiment 10,000 times.

In SAS we prepare by making a dataset with all  $650 \times 10,000$  diplomas by looping through the 10,000 simulations, creating 650 diplomas each time. We also assign a random number to each diploma. Then we sort on the random value within each simulation.

```
data dips;
do sim = 1 to 10000;
  do diploma = 1 to 650;
    randorder = uniform(0);
    output;
  end;
end;
run;

proc sort data=dips;
  by sim randorder;
run;
```

This gives the 650 diplomas a random order, within each simulation.

Next, we assign student ID numbers sequentially within the randomly ordered (with respect to diploma number) dataset and mark whether the diploma number matches the student ID. We'll print out a few rows to demonstrate.

```

data match;
set dips;
student = mod(_n_, 650);
match = (student eq diploma);
run;

proc print data = match (obs = 6); run;

Obs      sim      diploma      randorder      student      match
 1        1        462        0.000119        1          0
 2        1        202        0.000590        2          0
 3        1        451        0.001929        3          0
 4        1        496        0.005497        4          0
 5        1        355        0.008108        5          0
 6        1        374        0.011383        6          0

```

Finally, we count how many times the diploma matches the student using `proc summary` (5.1.1). It would also be relatively easy to count this manually within the data step.

```

proc summary data=match;
by sim;
var match;
output out=summatch sum=totalmatches;
run;

```

We can generate a table of the results using `proc freq` (5.1.7) and the mean and standard deviation using `proc means` (5.1.1).

```

proc freq data=summatch;
tables totalmatches / nocum;
run;

```

The FREQ Procedure

totalmatches	Frequency	Percent
0	3721	37.21
1	3692	36.92
2	1837	18.37
3	563	5.63
4	149	1.49
5	32	0.32
6	6	0.06

```

proc means data=summatch mean std var;
var totalmatches;
run;

```

The MEANS Procedure

Analysis Variable : totalmatches

Mean	Std Dev	Variance
0.9847000	0.9905372	0.9811640

In R, we begin by defining a function to carry out one of the simulations. The `students` vector is generated and then permuted using the `sample()` function (see 2.3.3) to represent

the diplomas received by the ordered students. The `==` operator (B.4.2) is used to compare each of the elements of the vectors, and the number of matches is returned. The `replicate()` function is used to run this multiple times and save the result.

```
> givedips = function(n) {
  students = 1:n
  diploma = sample(students, n)
  return(sum(students==diploma))
}

> res = replicate(10000, givedips(650))

> table(res)

res
  0   1   2   3   4   5   6 
3714 3745 1749  612 147  26   7 

> library(mosaic)
> favstats(res)

  min  Q1 median  Q3 max  mean    sd    n missing
  0    0      1    2    6 0.984 0.991 10000      0
```

The expected value and standard deviation of the number of students who receive their diplomas in the random disbursement are both 1.

### 10.2.3 Monty Hall problem

The Monty Hall problem illustrates a simple setting where intuition is often misleading. The situation is based on the TV game show “Let’s Make a Deal.” First, Monty (the host) puts a prize behind one of three doors. Then the player chooses a door. Next (without moving the prize) Monty opens an unselected door, revealing that the prize is not behind it. The player may then switch to the other nonselected door. Should the player switch?

Many people see that there are now two doors to choose between and feel that since Monty can always open a nonprize door, there’s still equal probability for each door. If that were the case, the player might as well keep the original door. This intuition is so attractive that when Marilyn Vos Savant asserted that the player should switch (in her *Parade* magazine column), there were reportedly 10,000 letters asserting she was wrong.

A correct intuitive route is to observe that Monty’s door is fixed. The probability that the player has the right door is  $1/3$  before Monty opens the nonprize door, and remains  $1/3$  after that door is open. This means that the probability the prize is behind one of the other doors is  $2/3$ , both before and after Monty opens the nonprize door. After Monty opens the nonprize door, the player gets a  $2/3$  chance of winning by switching to the remaining door. If the player wants to win, they should switch doors.

One way to prove to yourself that switching improves your chances of winning is through simulation. In fact, even deciding how to code the problem may be enough to convince yourself to switch.

In SAS, we assign the prize to a door, then make an initial guess. If the guess was right, Monty can open either door. We’ll switch to the other door. Rather than have Monty choose a door, we’ll choose one, under the assumption that Monty opened the other one. We do this with a `do until` loop (4.1.1). If our initial guess was wrong, Monty will open the only remaining nonprize door, and when we switch we’ll be choosing the prize door.

```

data mh;
do i = 1 to 10000;
    prize = rand("TABLE", .333, .333);
    * Monty puts the prize behind a random door;
    initial_guess = rand("TABLE", .333, .333);
    * We make a random initial guess;
    * if the initial guess is right, Monty can
        open either of the others;
    * which means that player can switch to either
        of the other doors;
    if initial_guess eq prize then do;
        new_guess = initial_guess;
        do until (new_guess ne initial_guess);
            new_guess = rand("TABLE", .333, .333);
        end;
    end;
    if initial_guess ne prize then new_guess = prize;
    output;
end;
run;

```

To see what happened, we'll summarize the resulting dataset. If our initial guess was right, we'll win by keeping it. If switching leads to a win, the new guess is where the prize is. We create new variables to indicate winning using a logical test (as in 2.2).

```

data mh2;
set mh;
    win_by_keep = (initial_guess eq prize);
    win_by_switch = (new_guess eq prize);
run;

proc means data=mh2 mean;
    var win_by_keep win_by_switch;
run;

```

Variable	Mean
win_by_keep	0.3331000
win_by_switch	0.6669000

In R, we write two helper functions. In one, Monty opens a door, choosing at random among the nonchosen doors if the initial choice was correct, or choosing the one nonselected non-prize door if the initial choice was wrong. The other function returns the door chosen by swapping. We use the `sample()` function (2.3.3) to randomly pick one value. We then use these functions on each trial with the `apply()` statement (B.5.2).

```
> numsim = 10000
> doors = 1:3
> opendoor = function(x) {
  # input x is a vector with two values
  # first element is winner, second is choice
  if (x[1]==x[2])    # guess was right
    return(sample(doors[-c(x[1])], 1))
  else return(doors[-c(x[1],x[2])])
}
> opendoor(c(1, 1))  # can return 2 or 3
[1] 3
> opendoor(c(1, 2))  # must return 3!
[1] 3
```

As an reminder of how this function works, Monty can choose either doors 2 or 3 to open when the winning door is initially chosen. When the winning door and initial choice differ (as in the latter example), there is only one door which can be opened.

```
> swapdoor = function(x) { return(doors[-c(x[1], x[2])]) }
> swapdoor(c(1,1))

[1] 2 3

> swapdoor(c(1,2))

[1] 3
```

The `swapdoor()` function works in a similar fashion. Once these parts are in place, the simulation is straightforward.

```
> winner = sample(doors, numsim, replace=TRUE)
> choice = sample(doors, numsim, replace=TRUE)
> open = apply(cbind(winner, choice), 1, opendoor)
> newchoice = apply(cbind(open, choice), 1, swapdoor)

> cat("Without switching, won ",
      round(sum(winner==choice)/numsim*100, 1),
      " percent of the time.\n", sep="")
```

Without switching, won 33 percent of the time.

```
> cat("With switching, won ",
      round(sum(winner==newchoice)/numsim*100, 1),
      " percent of the time.\n", sep="")
```

With switching, won 67 percent of the time.

## 10.3 Further resources

Rizzo [142] provides a comprehensive introduction to statistical computing in R, while [72] and [71] describe the use of R for simulation studies.

# Chapter 11

## Special topics

In this chapter we demonstrate some key programming and statistical techniques that statisticians may encounter in daily practice.

### 11.1 Processing by group

One of the most common needs in analytic practice is to replicate analyses for subgroups within the data. For example, one may need to stratify a linear regression by gender or repeat a modeling exercise multiple times for each replicate in a simulation experiment.

In SAS the basic tool for this is the `by` statement. Accepted by most analytic procedures, this runs the procedure on subgroups of the specified `by` variable(s). The data should have been previously sorted on those variables using the `sort` procedure.

In R, the basic tools for replication include the `by()` function and the `apply()` family of functions (B.5.2). The syntax for these functions can be complicated, however, and various packages exist that can replicate and enhance the functionality provided by `apply()`. One of these is the `plyr` package developed by Hadley Wickham, demonstrated below; another is the `doBy` package.

#### Means by group

*Examples:* 5.7.4 and 2.6.4

The simplest and possibly most common task is to generate simple statistics by a grouping variable. Here we simulate some data to demonstrate.

```
data test;
do cat = "blue", "red";
  do i = 1 to 50;
    x = normal(1966);
    output;
  end;
end;
run;
```

We can get the mean for the whole group with the `summary` procedure (5.1.1).

```

proc summary data=test;
  output out=whole mean=meanx;
  var x;
run;

proc print data=whole;
run;

Obs      _TYPE_      _FREQ_      meanx
 1          0          100      -.006787841

```

To implement by-group processing, first sort the data on the grouping variable (2.3.10), then just insert the by statement into the proc `summary` code.

```

proc sort data=test;
  by cat;
run;

proc summary data=test;
  by cat;
  output out=newds mean=meanx;
  var x;
run;

proc print data=newds;
run;

```

Obs	cat	_TYPE_	_FREQ_	meanx
1	blue	0	50	0.004504
2	red	0	50	-0.018079

The summary statistics for each by group are included in output and in datasets created by the procedure (see 5.1.1 for a discussion of `output` statement syntax).

In R, we use the `tapply()` function.

```

> options(digits=3)
> cat = rep(c("blue", "red"), each=50)
> y = rnorm(100)
> tapply(y, cat, mean)

  blue     red
-0.0568  0.0616

> xtabs(~ ave(y, as.factor(cat), FUN=mean))

ave(y, as.factor(cat), FUN = mean)
-0.0568  0.0616
      50      50

> library(mosaic)
> mean(y ~ cat)

  blue     red
-0.0568  0.0616

```

The `tapply()` function applies the function given as the third argument (in this case `mean()`) to the vector in the first argument (`y`) stratified by every unique set of values

of the list of factors in the second argument ( $x$ ). It returns a vector of that length with the results of the function. Similar functionality is available using the `by()` or the `ave()` functions (see `example(ave)`), the latter of which returns a vector of the same length as  $x$  with each element equal to the mean of the subset of observations with the factor level specified by  $y$ . Many functions (e.g., `mean()`, `median()`, and `favstats()`) within the `mosaic` package support a lattice-style modeling language for summary statistics.

### Linear models stratified by each value of a grouping variable

Replicating a linear regression within subgroups in SAS is remarkably similar to getting means within each subgroup. We'll use the `HELP` data to assess the relationship between age and drinking, by gender.

In SAS, we begin by sorting the data by `female`.

```
proc sort data="c:/book/help.sas7bdat";
  by female;
run;
```

Note that this `sort` changes the stored dataset in the operating system.

Now that the dataset is sorted, we write the `proc reg` code, inserting a `by female` statement, and saving the parameter estimates with the `ODS` system.

```
ods select none;
ods output parameterestimates=params;
proc reg data="c:/book/help.sas7bdat";
  by female;
  model i1 = age;
run;
ods select all;
```

The result is a data set with parameter estimates for each gender. Note that if the `by` variable has many distinct values, output may be voluminous.

```
options ls=64;
proc print data=params; run;
```

Obs	FEMALE	Model	Dependent	Variable	DF	Estimate
1	0	MODEL1	I1	Intercept	1	-3.69320
2	0	MODEL1	I1	AGE	1	0.63493
3	1	MODEL1	I1	Intercept	1	5.82858
4	1	MODEL1	I1	AGE	1	0.25118

Obs	StdErr	tValue	Probt
1	4.98628	-0.74	0.4594
2	0.13735	4.62	<.0001
3	8.86598	0.66	0.5124
4	0.23943	1.05	0.2965

In R, we begin by showing a way to do this from scratch, i.e., without convenience functions. It's often a useful programming exercise to code such routines, rather than relying on existing functions or packages.

```

> ds = read.csv("c:/book/help.csv")
> uniquevals = unique(ds$female)
> numunique = length(uniquevals)
> formula = as.formula(i1 ~ age)
> p = length(coef(lm(formula, data=ds)))
> params = matrix(rep(0, numunique*p), nrow=p, ncol=numunique)
> for (i in 1:length(uniquevals)) {
  cat("grouping: ", i, "\n")
  params[,i] = coef(lm(formula, data=subset(ds,
    female==uniquevals[i])))
}

grouping: 1
grouping: 2

> rownames(params) = c("Intercept", "Age")
> colnames(params) = ifelse(uniquevals==0, "male", "female")
> params

      male female
Intercept -3.693  5.829
Age        0.635  0.251

```

In the above code, separate regressions are fit for each value of the grouping variable `z` through use of a `for` loop. This requires the creation of a matrix of results `params` to be set up in advance, of the appropriate dimension (number of rows equal to the number of parameters ( $p = k + 1$ ) for the model, and number of columns equal to the number of levels for the grouping variable `z`). Within the loop, the `lm()` function is called, and the coefficients from each fit are saved in the appropriate column of the `params` matrix.

A simpler and more elegant approach is to use the `dplyr()` and `ldply()` functions from Hadley Wickham's `plyr` package.

```

> library(plyr)
> models = dplyr(ds, "female", function(df) lm(i1~age, data=df))
> ldply(models, coef)

      female (Intercept)    age
1         0       -3.69  0.635
2         1        5.83  0.251

```

The `dplyr()` function splits a dataframe, applies a function to each of the parts, and returns the results in a list. The `ldply()` function reverses this process: it splits a list, applies a function to each element, and returns a dataframe. Note that we define the function within the call to `dplyr()`, without giving it a name. This is often a useful technique in `apply()`-like functions (B.5.2).

## 11.2 Simulation-based power calculations

In some settings, analytic power calculations (5.5) may not be readily available. A straightforward alternative is to estimate power empirically, simulating data from the proposed design under given assumptions regarding the alternative.

We consider a study of children clustered within families. Each family has three children; in some families all three children have an exposure of interest, while in others just one child is exposed. In the simulation, we assume that the outcome is multivariate normal with a

higher mean for those with the exposure, and 0 for those without. A compound symmetry correlation is imposed, with equal variances for each child. We assess the power to detect an exposure effect where the intended analysis uses a random intercept model (7.4.2) to account for the clustering within families.

With this simple covariance structure it is trivial to generate correlated errors directly, as in the SAS code below; an alternative which could be used with more complex structures in SAS is `proc simnormal` (3.1.7).

```
data simpower1;
  effect = 0.35; /* effect size */
  corr = 0.4; /* desired correlation */
  covar = (corr)/(1 - corr); /* implied covariance given variance = 1*/
  numsim = 1000; /* number of datasets to simulate */
  numfams = 100; /* number of families in each dataset */
  numkids = 3; /* each family */
  do simnum = 1 to numsim; /* make a new dataset for each simnum */
    do famid = 1 to numfams; /* make numfams families in each dataset */
      inducecorr = normal(42)* sqrt(covar);
        /* this achieves the desired
           correlation between kids within family */
      do kidnum = 1 to numkids; /* generate each kid */
        exposed = ((kidnum eq 1) or (famid le numfams/2));
          /* assign kid to be exposed */
        x = (exposed * effect) +
          (inducecorr + normal(0))/sqrt(1 + covar);
        output;
      end;
    end;
  end;
run;
```

In the previous code, the integer provided as an argument in the initial use of the `normal` function sets the seed used for all calls to the pseudo-random number generator, so that the results can be exactly replicated, if necessary (see 3.1.3). Next, we run the desired model on each of the simulated datasets, using the `by` statement (11.1) and saving the estimated fixed effects parameters using the ODS system (A.7).

```
ods select none;
ods output solutionf=simres;
proc mixed data=simpower1 order=data;
by simnum;
  class exposed famid;
  model x = exposed / solution;
  random int / subject=famid;
run;
ods select all;
```

Finally, we process the resulting output dataset to generate an indicator of rejecting the null hypothesis of no exposure effect.

```
data powerout;
set simres;
  where exposed eq 1;
  reject=(probt lt 0.05);
run;
```

The proportion of rejections shown in the results of proc freq is the empirical estimate of power. The binomial option to proc freq (5.1.7) provides the asymptotic and exact CI for the estimated power.

```
proc freq data=powerout;
  tables reject / binomial (level='1');
run;
```

#### The FREQ Procedure

reject	Frequency	Percent	Cumulative	Cumulative
			Frequency	Percent
0	147	14.70	147	14.70
1	853	85.30	1000	100.00

#### Binomial Proportion

reject = 1

Proportion	0.8530
ASE	0.0112
95% Lower Conf Limit	0.8311
95% Upper Conf Limit	0.8749

#### Exact Conf Limits

95% Lower Conf Limit	0.8295
95% Upper Conf Limit	0.8744

Test of H0: Proportion = 0.5

ASE under H0 0.0158

Z 22.3257

One-sided Pr > Z <.0001

Two-sided Pr > |Z| <.0001

Sample Size = 1000

In R, we specify the correlation matrix directly and simulate the multivariate normal.

```
> library(MASS)
> library(nlme)
> # initialize parameters and building blocks
> effect = 0.35      # effect size
> corr = 0.4          # intrafamilial correlation
> numsim = 1000
> n1fam = 50          # families with 3 exposed
> n2fam = 50          # families with 1 exposed and 2 unexposed
> # 3x3 compound symmetry matrix
> vmat = matrix(c
+   ( 1,      corr, corr,
+     corr, 1      , corr,
+     corr, corr, 1      ), nrow=3, ncol=3)
> # 1 1 1 ... 1 0 0 0 ... 0
> x = c(rep(1, n1fam), rep(1, n1fam), rep(1, n1fam),
+       rep(1, n2fam), rep(0, n2fam), rep(0, n2fam))
> # 1 2 ... n1fam 1 2 ... n1fam ...
> id = c(1:n1fam, 1:n1fam, 1:n1fam,
+       (n1fam+1:n2fam), (n1fam+1:n2fam), (n1fam+1:n2fam))
> power = rep(0, numsim) # initialize vector for results
```

The concatenate function (`c()`) is used to glue together the appropriate elements of the design matrices and underlying correlation structure.

```
> for (i in 1:numsim) {
  # all three exposed
  grp1 = mvrnorm(n1fam, c(effect, effect, effect), vmat)

  # only first exposed
  grp2 = mvrnorm(n2fam, c(effect, 0, 0), vmat)

  # concatenate the output vector
  y = c(grp1[,1], grp1[,2], grp1[,3],
        grp2[,1], grp2[,2], grp2[,3])

  group = groupedData(y ~ x | id) # specify dependence structure
  res = lme(group, random = ~ 1) # fit random intercept model
  pval = summary(res)$tTable[2,5] # grab results for main parameter
  power[i] = pval<=0.05 # is it statistically significant?
}
```

This yields the following power estimate (and confidence interval due to simulation).

```
> cat("\nEmpirical power for effect size of ", effect,
     " is ", round(sum(power)/numsim,3), ".\n", sep="")

Empirical power for effect size of 0.35 is 0.846.

> cat("95% confidence interval is",
      round(prop.test(sum(power), numsim)$conf.int, 3), "\n")

95% confidence interval is 0.822 0.868
```

## 11.3 Reproducible analysis and output

The key idea of reproducible analysis is that data analysis code, results, and interpretation should all be located together. This stems from the concept of “literate programming” (in the sense of Knuth [89]) and facilitates transparent and repeatable analysis [97, 53]. Reproducible analysis systems, which are becoming more widely adopted [13], help to provide a clear audit trail and automate report creation. Ultimately, the goal is to avoid post-analysis cut and paste processing, which has a high probability of introducing errors.

There are various implementations of reproducible analysis for SAS and R [101, 97, 203], several of which made the production of this book possible. Each of these systems functions by allowing the analyst to combine code and text into a single file. This file is processed to extract the code, run it through the statistical systems in batch mode, collect the results, then integrate the text, code, output, and graphical displays into the final document.

In SAS, the `report` procedure can be used to make customized tabular presentations that can reliably be regenerated. These reports can be a sort of primitive reproducible analysis, in the sense that they may not require additional word processing before use. However, including text is awkward, and the result is not likely to be attractive. It is also possible to generate very crude reproducible analyses using the `ODS` system to generate data analysis and the `print` procedure to produce text surrounding the output, with the `ods rtf` statement used to generate a file that can be read by many word processing programs.

For SAS users who know L<sup>A</sup>T<sub>E</sub>X, the SASweave program (written by Russell Lenth [101]) is a literate programming solution that integrates attractively formatted code, results, and

text [101]. (Lenth also wrote statweave, which supports other languages and environments.) SASweave has the attractive feature that it can include both SAS and R commands in a single document, using original code for SAS and relying for R on the Sweave system (due to Leisch [97]). Most of the results displayed in this book were generated using SASweave.

Another option for SAS users who use L<sup>A</sup>T<sub>E</sub>X is the `statrep` style developed by SAS for documentation purposes (see the paper by Tim Arnold and Warren Kuhfeld, available from SAS and at [www.amherst.edu/~nhorton/sasr2/examples/statrep.pdf](http://www.amherst.edu/~nhorton/sasr2/examples/statrep.pdf)).

The options available in R are more extensive, and they are an active area of development. The most powerful and flexible of these is the `knitr` package (due to Yihui Xie [203]). The package can be used with L<sup>A</sup>T<sub>E</sub>X but another useful option is to use Markdown as the text-formatting part of the reproducible analysis. Markdown is a simplified markup language that generates HTML code for web viewing. The `knitr` package is well-integrated with RStudio, which provides one-click access to Markdown and knitr. More details can be found in [203] and [50] as well as the CRAN reproducible analysis task view (see also <http://yihui.name/knitr>).

As an example of how these systems work, we consider a minimal analysis in Markdown using data from the built-in `cars` data frame.

Figure 11.1 displays a sample Markdown input file. The file is given a title (`R Markdown example`) by following it with a line of equal signs. The title and helpful text below it are the content provided when opening a new `R Markdown` document in RStudio. Simple markup (such as bolding) is added through use of the `**` characters before and after the word `Help`. Blocks of code are begun using the ````{r}` command and closed with a ````` command (three back quotes). In this example, the correlation between two variables is calculated and a scatterplot is generated.

The formatted output can be generated and displayed by clicking the `Knit HTML` button in RStudio, or by using the commands in the following code block.

### R Markdown example

---

```
This is an R Markdown document. Markdown is a simple formatting
syntax for authoring web pages (click the **Help** toolbar button).
```

```
When you click the **Knit HTML** button a web page will be
generated that includes both content as well as the output
of any embedded R code chunks within the document. You can
embed an R code chunk like this:
```

```
```{r}
with(cars, cor(speed, dist))
```
```

You can also embed plots, for example:

```
```{r fig.height=4}
library(lattice)
xyplot(dist ~ speed, type=c("p", "smooth"), data=cars)
```
```

Figure 11.1: Sample Markdown input file

```
library(markdown)
knit("filename.Rmd")    # creates filename.md
markdownToHTML("filename.md", "filename.html")
browseURL("filename.html")
```

The `knit()` function extracts the R commands from a specially formatted input file (called `filename.Rmd`), evaluates them, and integrates the resulting output, including text and graphics, into an intermediate file (`filename.md`). This file is then processed, using the `markdownToHTML()` function to make a final display file, in HTML format. A screenshot of the results of performing these steps on the `.Rmd` file displayed in Figure 11.1 is displayed in Figure 11.2.

The `knit()` function operates, by default, on the convention that input files ending with `.Rmd` generate a `.md` (Markdown) file, and files ending with `.Rnw` generate a `.tex` (L<sup>A</sup>T<sub>E</sub>X) file.

A L<sup>A</sup>T<sub>E</sub>X file can be generated using the following commands, where `filename.Rnw` is a L<sup>A</sup>T<sub>E</sub>X file with specific codes indicating the presence of R statements.

## R Markdown example

This is an R Markdown document. Markdown is a simple formatting syntax for authoring web pages (click the **Help** toolbar button).

When you click the **Knit HTML** button a web page will be generated that includes both content as well as the output of any embedded R code chunks within the document. You can embed an R code chunk like this:

```
with(cars, cor(speed, dist))
```

```
## [1] 0.8069
```

You can also embed plots, for example:

```
library(lattice)
xyplot(dist ~ speed, type = c("p", "smooth"), data = cars)
```

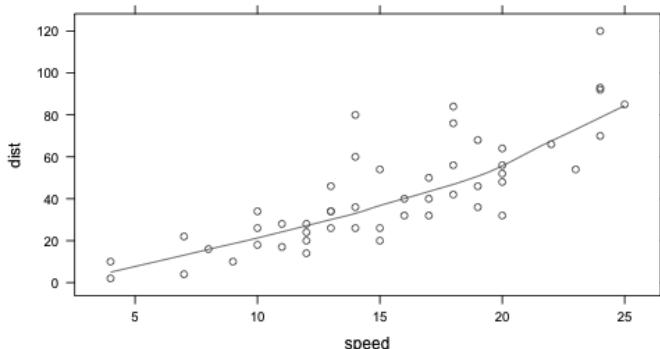


Figure 11.2: Formatted output from R Markdown example

```
library(knitr)
knit("filename.Rnw")
```

The resulting `filename.tex` file could then be compiled with `pdflatex` in the operating system, resulting in a PDF file. This is done automatically using the `Compile to PDF` button in RStudio.

It's often useful to evaluate the code separately. The `Stangle()` function creates a file containing the code chunks and omitting the text. The resulting file could be run as a script using `source()` and would generate just the results seen in the woven document. The `spin()` function in the `knitr` package takes a formatted R script and produces an R Markdown document. This can be helpful for those moving from the use of scripts to more structured Markdown files.

RStudio also supports R Presentations using a variant of the R Markdown language. Instructions and an example can be found by opening a new `R presentations` document in RStudio.

## 11.4 Advanced statistical methods

In this section we discuss implementations of modern statistical methods and techniques, including Bayesian methods, propensity score analysis, missing data methods, and estimation of finite mixture models.

### 11.4.1 Bayesian methods

Bayesian methods are increasingly commonly utilized, and implementations of many models are available in both SAS and R. For SAS, the on-line documentation is a valuable resource: Contents; SAS Products; SAS/STAT User's guide; Introduction to Bayesian Analysis Procedures. For R, the CRAN task view on Bayesian inference provides an overview of the packages that incorporate some aspect of Bayesian methodologies.

We focus here on Markov chain Monte Carlo (MCMC) methods for model fitting, which are quite general and much more flexible than closed form solutions. Diagnosis of convergence is a critical part of any MCMC model fitting (see Gelman et. al., [52] for an accessible introduction). In SAS, convergence diagnosis is built in to procedures that allow fitting models by MCMC. In R, support for model assessment is provided, for example, in the `coda` (Convergence Diagnosis and Output Analysis) package written by Kate Cowles and others.

In SAS, we can use the `bayes` statement available in some procedures or implement general models using the `mcmc` procedure.

```

/* generalized linear models */
/* example: negative binomial */
proc genmod data=ds;
  model y = x1 x2 / dist=nb;
  bayes nbi=2000 nmc=10000 ...;
run;

/* general models via proc MCMC */
/* example: Poisson random effects model */
proc mcmc data=ds nmc=10000 thin=10 ...;
  parms fixedint beta1 gscale;
  prior fixedint ~ normal(0, var=10000);
  prior beta1 ~ laplace(0, scale=50);
  prior gscale ~ igamma(.01 , scale=.01) ;
  random rint ~ gamma(shape=1, scale=gscale) subject=i initial=0.0001;
  mu = exp(fixedint + rint + beta1*covar);
  model y ~ poisson(mu);
run;

```

The `bayes` statement in `proc genmod` allows Bayesian linear, logistic, and Poisson regression, as well as regression with any other outcome distribution described in 7.1. One key option to the `bayes` statement is `coeffprior`, used for specifying the prior.

The `mcmc` procedure allows very general models indeed to be fit. The syntax is very natural: the `parms` statement describes the parameters that will be sampled (and their starting values), while the `prior` statements define each parameter's prior distribution. There is also a `hyperprior` statement available, though it is not demonstrated here. An arbitrary number of additional statements describe how the parameters and covariates interact. Finally, a `model` statement describes the distribution of the outcome and its relationship to the covariates and parameters. In the code above, the `random` statement is used to introduce a random effect; this is a shortcut to a model that can be built with the other statements only with considerable effort.

In either the `bayes` or `proc mcmc` statements, the `nbi` and `nmc` options specify the number of burn-in samples and the number of iterations for inference, respectively. The `diagnostics` and `plots` options generate diagnostic and interpretive information.

```

library(MCMCpack)
# linear regression
mod1 = MCMCregress(formula, burnin=1000, mcmc=10000, data=ds)

# logistic regression
mod2 = MCMClogit(formula, burnin=1000, mcmc=10000, data=ds)

# Poisson regression
mod3 = MCMCpoisson(formula, burnin=1000, mcmc=10000, data=ds)

```

The CRAN task view on Bayesian inference provides an overview of the packages that incorporate some aspect of Bayesian methodologies. Table 11.1 displays modeling functions available within the `MCMCpack` package (including the three listed above). By default, the prior mean and precision are set to 0, equivalent to an improper uniform distribution.

More general MCMC models can also be fit in R, typically in packages that call stand-alone MCMC software such as OpenBUGS, JAGS, or WinBUGS. These packages include `BRugs`, `R2WinBUGS`, `rjags`, `R2jags`, and `runjags`.

Table 11.1: Bayesian modeling functions available within the `MCMCpack` package

|                                  |  |
|----------------------------------|--|
| <code>MCMCbinaryChange()</code>  | MCMC for a Binary Multiple Changepoint Model   |
| <code>MCMCdynamiEI()</code>      | MCMC for Quinn's Dynamic Ecological Inference Model  |
| <code>MCMCdynamIRT1d()</code>    | MCMC for Dynamic One Dimensional Item Response Theory Model  |
| <code>MCMCfactanal()</code>      | MCMC for Normal Theory Factor Analysis Model   |
| <code>MCMChierEI()</code>        | MCMC for Wakefield's Hierarchical Ecological Inference Model   |
| <code>MCMCirt1d()</code>         | MCMC for One Dimensional Item Response Theory Model  |
| <code>MCMCirtHier1d()</code>     | MCMC for Hierarchical One Dimensional Item Response Theory Model, Covariates Predicting Latent Ideal Point (Ability) |
| <code>MCMCirtKd()</code>         | MCMC for K-Dimensional Item Response Theory Model  |
| <code>MCMCirtKdHet()</code>      | MCMC for Heteroskedastic K-Dimensional Item Response Theory Model  |
| <code>MCMCirtKdRob()</code>      | MCMC for Robust K-Dimensional Item Response Theory Model   |
| <code>MCMClogit()</code>         | MCMC for Logistic Regression   |
| <code>MCMCmetrop1R()</code>      | Metropolis Sampling from User-Written R Function   |
| <code>MCMCmixfactanal()</code>   | MCMC for Mixed Data Factor Analysis Model  |
| <code>MCMCmnl()</code>           | MCMC for Multinomial Logistic Regression   |
| <code>MCMCoprobit()</code>       | MCMC for Ordered Probit Regression   |
| <code>MCMCordfactanal()</code>   | MCMC for Ordinal Data Factor Analysis Model  |
| <code>MCMCpoisson()</code>       | MCMC for Poisson Regression  |
| <code>MCMCpoissonChange()</code> | MCMC for a Poisson Regression Changepoint Model  |
| <code>MCMCprobit()</code>        | MCMC for Probit Regression   |
| <code>MCMCquantreg()</code>      | Bayesian Quantile Regression Using Gibbs Sampling  |
| <code>MCMCregress()</code>       | MCMC for Gaussian Linear Regression  |
| <code>MCMCSVDrreg()</code>       | MCMC for SVD Regression  |
| <code>MCMCtobit()</code>         | MCMC for Gaussian Linear Regression with a Censored Dependent Variable   |

### Logistic regression via MCMC

One use for Bayesian logistic regression might be in the case of complete or quasi-complete separation. Loosely, this occurs when all the subjects in some level of the exposure variables have the same outcome status. Here we simulate such data and demonstrate how to use Bayesian MCMC methods to fit the model. The simulated data have 100 trials in each of two levels of a predictor, with 0 and 5 events in the two levels. Note that the classical

estimated odds ratio is infinity, or undefined, and that different software implementations behave unpredictably in this instance.

```
options ls=64;

/* make the data */
data testmcmc;
  x=0; count=0; n=100; output;
  x=1; count=5; n=100; output;
run;

ods select postsummaries;
proc genmod descending data=testmcmc;
  model count/n = x / dist=binomial link=logit;
  bayes seed=12182002 nbi=100 nmc=2000
    coeffprior=normal(var=25)
    statistics(percent=2.5,50,97.5)=summary;
run;
ods select all;
```

The GENMOD Procedure

#### Bayesian Analysis

##### Posterior Summaries

| Parameter | N    | Standard |           |
|-----------|------|----------|-----------|
|           |      | Mean     | Deviation |
| Intercept | 2000 | -6.5397  | 1.7475    |
| x         | 2000 | 3.4649   | 1.7954    |

##### Posterior Summaries

| Parameter | Percentiles |         |         |
|-----------|-------------|---------|---------|
|           | 2.5%        | 50%     | 97.5%   |
| Intercept | -10.5758    | -6.2445 | -3.8757 |
| x         | 0.4625      | 3.2342  | 7.5514  |

The `bayes` statement has options to control many aspects of the MCMC process. We select a short burn-in and a small number of cycles, for speed in this demonstration setting. The prior distribution and precision are chosen to match `MCMClogit()` in R.

```
> events.0=0 # for X = 0
> events.1=5 # for X = 1
> x = as.factor(c(rep(0,100), rep(1,100)))
> y = c(rep(0,100-events.0), rep(1,events.0),
       rep(0, 100-events.1), rep(1, events.1))
> library(MCMCpack)
> logmcmc = MCMClogit(y ~ x, burnin=100, mcmc=2000, b0=0, B0=.04)
```

```
> summary(logmcmc)

Iterations = 101:2100
Thinning interval = 1
Number of chains = 1
Sample size per chain = 2000

1. Empirical mean and standard deviation for each variable,
   plus standard error of the mean:

              Mean     SD Naive SE Time-series SE
(Intercept) -6.50  1.70    0.0381      0.0931
x1          3.42  1.77    0.0395      0.1009

2. Quantiles for each variable:

        2.5%   25%   50%   75% 97.5%
(Intercept) -10.606 -7.48 -6.26 -5.30 -3.97
x1           0.708  2.15  3.15  4.52  7.50
```

Under the default normal prior, the mean of the prior is set with `b0`; `B0` is the prior precision. The `burnin` and `mcmc` options define the number of iterations discarded before inference iterations are captured, and the number of iterations for inference, respectively.

SAS and `MCMlogit()` provide similar estimates of the posterior means; the prior is contributing a fair amount of information in this difficult setting.

## Poisson regression

In addition to problematic examples such as the quasi-complete separation discussed on page 292, it may be desirable to consider Bayesian techniques in more conventional settings. Here we demonstrate a Poisson regression using the HELP data set.

```
proc import datafile='c:/book/help.csv'
  out=help dbms=dlm;
  delimiter=',';
  getnames=yes;
run;
```

```

ods select postintervals;
proc genmod data=help;
  class substance (param=reference ref='alcohol');
  model i1 = female substance age / dist=poisson;
  bayes nbi=100 nmc=2000;
run;
ods select all;

```

The GENMOD Procedure

#### Bayesian Analysis

##### Posterior Intervals

| Parameter        | Alpha | Equal-Tail Interval | HPD Interval    |
|------------------|-------|---------------------|-----------------|
| Intercept        | 0.050 | 2.7835 3.0157       | 2.7908 3.0216   |
| female           | 0.050 | -0.2301 -0.1233     | -0.2265 -0.1215 |
| substancecocaine | 0.050 | -0.8752 -0.7640     | -0.8753 -0.7641 |
| substanceheroin  | 0.050 | -1.1860 -1.0551     | -1.1876 -1.0585 |
| age              | 0.050 | 0.0103 0.0162       | 0.0103 0.0159   |

Diagnostic plots are generated by default.

In R, we use the MCMCpoisson() function.

```

> library(MCMCpack)
> posterior = with(ds,
+   MCMCpoisson(i1 ~ female + as.factor(substance) + age,
+   burnin=100, mcmc=2000))
> summary(posterior)

```

Iterations = 101:2100  
 Thinning interval = 1  
 Number of chains = 1  
 Sample size per chain = 2000

1. Empirical mean and standard deviation for each variable,  
 plus standard error of the mean:

|                             | Mean    | SD      | Naive SE | Time-series SE |
|-----------------------------|---------|---------|----------|----------------|
| (Intercept)                 | 2.8888  | 0.06255 | 1.40e-03 | 0.005864       |
| female                      | -0.1728 | 0.03109 | 6.95e-04 | 0.002888       |
| as.factor(substance)cocaine | -0.8140 | 0.02817 | 6.30e-04 | 0.002329       |
| as.factor(substance)heroin  | -1.1166 | 0.03477 | 7.78e-04 | 0.003252       |
| age                         | 0.0135  | 0.00155 | 3.47e-05 | 0.000147       |

2. Quantiles for each variable:

|                             | 2.5%    | 25%     | 50%     | 75%     | 97.5%   |
|-----------------------------|---------|---------|---------|---------|---------|
| (Intercept)                 | 2.7703  | 2.8454  | 2.8871  | 2.9349  | 3.0095  |
| female                      | -0.2355 | -0.1946 | -0.1734 | -0.1508 | -0.1105 |
| as.factor(substance)cocaine | -0.8676 | -0.8325 | -0.8134 | -0.7935 | -0.7615 |
| as.factor(substance)heroin  | -1.1856 | -1.1391 | -1.1153 | -1.0922 | -1.0485 |
| age                         | 0.0105  | 0.0124  | 0.0135  | 0.0146  | 0.0163  |

Default plots are available for MCMC objects returned by MCMCpack. These can be displayed using the command `plot(posterior)`.

### 11.4.2 Propensity scores

Propensity scores can be used to attempt causal inference in an observational setting where there are potential confounding factors [144, 145]. Here we consider comparisons of the physical component scores (PCSs) for homeless vs. nonhomeless subjects in the HELP study. Does homelessness make people less physically competent?

First, we examine the observed difference in PCS between homeless and homed.

```
ods select parameterestimates;
```

```
proc glm data="C:/book/help.sas7bdat" order=data;
  class homeless;
  model pcs = homeless / solution;
run; quit;
```

```
ods select all;
```

The GLM Procedure

Dependent Variable: PCS

| Parameter  | Estimate      | Error      | t Value | Pr >  t |
|------------|---------------|------------|---------|---------|
| Intercept  | 49.00082904 B | 0.68801845 | 71.22   | <.0001  |
| HOMELESS 1 | -2.06404896 B | 1.01292210 | -2.04   | 0.0422  |
| HOMELESS 0 | 0.00000000 B  | .          | .       | .       |

```
> lm1 = lm(pcs ~ homeless, data=ds)
> summary(lm1)
```

Call:

```
lm(formula = pcs ~ homeless, data = ds)
```

Residuals:

| Min    | 1Q    | Median | 3Q   | Max   |
|--------|-------|--------|------|-------|
| -34.93 | -7.90 | 0.64   | 8.39 | 25.81 |

Coefficients:

|             | Estimate | Std. Error | t value | Pr(> t )   |
|-------------|----------|------------|---------|------------|
| (Intercept) | 49.001   | 0.688      | 71.22   | <2e-16 *** |
| homeless    | -2.064   | 1.013      | -2.04   | 0.042 *    |

---

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

Residual standard error: 10.7 on 451 degrees of freedom

Multiple R-squared: 0.00912, Adjusted R-squared: 0.00693

F-statistic: 4.15 on 1 and 451 DF, p-value: 0.0422

We see statistically significant lower PCSs in the homeless ( $p = 0.042$ ). However, subjects were not randomized to homelessness. Homelessness may be a result of confounding factors that are associated with homelessness and cause reduced physical competence. If we want to make causal inference about the effects of homelessness, we need to adjust for these confounders.

## Regression adjustment

One approach to this problem involves controlling for possible confounders (in this case, age, gender, number of drinks, and MCS score) in a multiple regression model (6.1.1).

```
ods select parameterestimates;
```

```
proc glm data="C:/book/help.sas7bdat" order=data;
  class homeless female;
  model pcs = homeless age female i1 mcs / solution;
run;
```

```
ods select all;
```

The GLM Procedure

Dependent Variable: PCS

| Parameter  | Estimate      | Standard   |         |         |
|------------|---------------|------------|---------|---------|
|            |               | Error      | t Value | Pr >  t |
| Intercept  | 54.25705249 B | 2.66785940 | 20.34   | <.0001  |
| HOMELESS 1 | -1.14706552 B | 0.99794019 | -1.15   | 0.2510  |
| HOMELESS 0 | 0.00000000 B  | .          | .       | .       |
| AGE        | -0.26592570   | 0.06410301 | -4.15   | <.0001  |
| FEMALE 0   | 3.95518533 B  | 1.15141690 | 3.44    | 0.0006  |
| FEMALE 1   | 0.00000000 B  | .          | .       | .       |
| I1         | -0.08078748   | 0.02537684 | -3.18   | 0.0016  |
| MCS        | 0.07032292    | 0.03807251 | 1.85    | 0.0654  |

```
> lm2 = lm(pcs ~ homeless + age + female + i1 + mcs, data=ds)
> summary(lm2)
```

Call:

```
lm(formula = pcs ~ homeless + age + female + i1 + mcs, data = ds)
```

Residuals:

| Min    | 1Q    | Median | 3Q   | Max   |
|--------|-------|--------|------|-------|
| -35.77 | -6.67 | 0.41   | 7.67 | 26.59 |

Coefficients:

|             | Estimate | Std. Error | t value | Pr(> t )    |
|-------------|----------|------------|---------|-------------|
| (Intercept) | 58.2122  | 2.5667     | 22.68   | < 2e-16 *** |
| homeless    | -1.1471  | 0.9979     | -1.15   | 0.25099     |
| age         | -0.2659  | 0.0641     | -4.15   | 4e-05 ***   |
| female      | -3.9552  | 1.1514     | -3.44   | 0.00065 *** |
| i1          | -0.0808  | 0.0254     | -3.18   | 0.00156 **  |
| mcs         | 0.0703   | 0.0381     | 1.85    | 0.06540 .   |

---

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

Residual standard error: 10.2 on 447 degrees of freedom

Multiple R-squared: 0.112, Adjusted R-squared: 0.102

F-statistic: 11.2 on 5 and 447 DF, p-value: 3.21e-10

Controlling for the other predictors has caused the parameter estimate to attenuate to the point that it is no longer statistically significant ( $p = 0.25$ ). While controlling for other confounders may be effective in this problem, other situations may be more vexing, particularly if the dataset is small and the number of measured confounders is large. In such settings, the propensity score (the probability of being homeless, conditional on other factors), can be used. Typical applications include regression adjustment for the propensity to exposure, matching on the propensity to exposure, and stratification into similar levels of propensity. Propensity scores also allow easy investigation of the overlap in covariate space, a requirement for effective multiple regression adjustment that is often ignored. Here we demonstrate estimating the propensities, using them in a regression adjustment, and matching. Assessment of overlap resembles the assessment of the linear discriminant analysis in 7.10.18. For an example of stratification, see <http://tinyurl.com/sasrblog-propensity>.

### Estimating the propensity score

A typical way to estimate the propensity score is to model the exposure as a function of covariates in a logistic regression model (7.1.1).

```
ods select none;

proc logistic data="C:/book/help.sas7bdat" order=data;
  class homeless female;
  model homeless(event='1') = age female i1 mcs;
  output out = propen pred=propensity;
run;

ods select all;
```

The new dataset `propen` has the original data plus the new variable `propensity`.

In R, we use a `formula` object (see 6.1.1) to specify the model.

```
> form = formula(homeless ~ age + female + i1 + mcs)
> glm1 = glm(form, family=binomial, data=ds)
> propensity = glm1$fitted
```

The `glm1` object has a `fitted` element that contains the predicted probability from the logistic regression. For ease of use we extract it to a new object.

### Regression adjustment for propensity

The simplest use of the propensity score is to include it as a continuous covariate in a regression model.

```

ods select parameterestimates;

proc glm data=propopen order=data;
  class homeless;
  model pcs = homeless propensity/ solution;
run; quit;

ods select all;

The GLM Procedure

Dependent Variable: PCS

Parameter           Estimate      Standard
Intercept          54.53896207 B    1.82526760   t Value    Pr > |t|
HOMELESS 1         -1.17849749 B    1.03815099   -1.14     0.2569
HOMELESS 0         0.00000000 B    .
propensity        -12.88925969   3.94156548   -3.27     0.0012

> lm3 = lm(pcs ~ homeless + propensity, data=ds)
> summary(lm3)

Call:
lm(formula = pcs ~ homeless + propensity, data = ds)

Residuals:
    Min      1Q Median      3Q      Max
-34.03   -7.62    0.93    8.24   25.65

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 54.54       1.83    29.88 <2e-16 ***
homeless    -1.18       1.04   -1.14    0.2569
propensity  -12.89      3.94   -3.27    0.0012 **
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 10.6 on 450 degrees of freedom
Multiple R-squared:  0.0321,    Adjusted R-squared:  0.0278
F-statistic: 7.47 on 2 and 450 DF,  p-value: 0.000645

```

As with the multiple regression model, controlling for the propensity also leads to an attenuated estimate of the homeless coefficient.

### Matching on propensity score

Another approach matches exposed and unexposed (homeless and nonhomeless) subjects with similar propensity scores. This typically generates a sample that is approximately balanced on the terms included in the propensity model. Since a confounded effect requires a disequilibrium of the confounders between the groups, this can be an effective treatment.

In SAS, the matching process is relatively complex. As far as we are aware, there are no SAS procedures designed for this purpose. We utilize macros developed by Jon Kosanke and Erik Bergstrahl at the Mayo Clinic Division of Biomedical Statistics and Informatics

and enhanced slightly by ourselves. They are available from [www.amherst.edu/~nhorton/sasr2/examples](http://www.amherst.edu/~nhorton/sasr2/examples). The **dist** macro calculates the pairwise distances between observations, while the **vmatch** macro makes matches based on the distances, finding the closest set of matches while minimizing the overall distance. The latter macro calls upon the **nobs** macro.

```
%include "C:\ken\sasmacros\vmatch.sas";
%include "C:\ken\sasmacros\dist.sas";
%include "C:\ken\sasmacros\nobs.sas";

%dist(data=prop2, group=homeless, id=id, mvars=propensity,
      wts=1, vmatch=Y, a=1, b=1, lilm=201, dmax=0.1,
      outm=mp1_b, summatch=n, printm=N,
      mergeout="c:/book\mpropopen.sas7bdat");
```

The three SAS macros are well documented by text included at the top of their respective files, as is common with SAS macros that authors share. In the preceding code, the parameter values in the first line are relatively self-explanatory. The **wts** parameter allows multiple matching variables to be weighted differently. The **dmax** parameter specifies the maximum distance acceptable for a match. We arbitrarily decide that the propensities must be within 0.1 to make a match (our results would differ if other criteria were specified for the matching). The remaining parameters request the matching, ask for one and only one match per case, for all the cases to be matched (if possible), suppress printed output, and name the dataset to contain output. The output dataset contains the variable **matched**, an indicator that the observation is part of a matched set.

The comparison between the matched groups can be made using the **mpropopen** dataset, after restricting the analysis to the matched observations using the **where** statement (2.3.1).

```
ods select parameterestimates;

proc glm data="c:\book\mpropopen.sas7bdat" order=data;
  where matched eq 1;
  class homeless;
  model pcs = homeless / solution;
run;

ods select all;
```

#### The GLM Procedure

Dependent Variable: PCS

| Parameter  | Estimate      | Error      | Standard |         |
|------------|---------------|------------|----------|---------|
|            |               |            | t Value  | Pr >  t |
| Intercept  | 47.17540798 B | 0.75406901 | 62.56    | <.0001  |
| HOMELESS 0 | 1.84260270 B  | 1.06641462 | 1.73     | 0.0848  |
| HOMELESS 1 | 0.00000000 B  | .          | .        | .       |

The effect of homelessness on PCS no longer reaches the threshold of statistical significance. The matching and comparison are straightforward to do in R using the **Matching** package.

```
> library(Matching)
> rr = with(ds, Match(Y=pcs, Tr=homeless, X=propensity, M=1))
> summary(rr)

Estimate... -0.80207
AI SE..... 1.4448
T-stat..... -0.55516
p.val..... 0.57878

Original number of observations..... 453
Original number of treated obs..... 209
Matched number of observations..... 209
Matched number of observations (unweighted). 252
```

We see that the causal estimate of  $-0.80$  in the matched comparison is not significantly different from zero ( $p = 0.58$ ), which is similar to the results from the other approaches that accounted for the possible confounders. Differences between R and SAS here are due mainly to sophisticated techniques used by the **Matching** package, including the use of matching with replacement.

### Assessing balance after matching

It would be wise to make a further investigation of whether the matching “worked,” in the sense of making the groups more similar with respect to the potential confounders.

For example, here are the means and standard deviations among the whole sample (including just two covariates for space reasons).

```
proc means data=propen mean std;
  class homeless;
  var age pcs;
run;
```

| The MEANS Procedure |     |          |            |            |  |
|---------------------|-----|----------|------------|------------|--|
|                     |     | N        |            |            |  |
| HOMELESS            | Obs | Variable | Mean       | Std Dev    |  |
| 0                   | 244 | AGE      | 35.0409836 | 7.1657594  |  |
|                     |     | PCS      | 49.0008290 | 10.8287793 |  |
| 1                   | 209 | AGE      | 36.3684211 | 8.2609576  |  |
|                     |     | PCS      | 46.9367801 | 10.6510842 |  |

In comparison, here are the values in the matched sample.

```
proc means data="c:/book/mpropen.sas7bdat" mean std;
  where matched;
  class homeless;
  var age pcs;
run;
```

The MEANS Procedure

| HOMELESS | N   | Obs | Variable   | Mean       | Std Dev |
|----------|-----|-----|------------|------------|---------|
| 0        | 204 | AGE | 35.5392157 | 7.3325935  |         |
|          |     | PCS | 49.0180107 | 11.0194001 |         |
| 1        | 204 | AGE | 36.1470588 | 8.1305809  |         |
|          |     | PCS | 47.1754080 | 10.5152180 |         |

Note that while balance was improved for both covariates, there remains some difference between the groups.

The `MatchBalance()` function can be used to describe the distribution of the predictors (by homeless status) before and after matching (to save space, only the results for `age` are displayed).

```
> longout = capture.output(MatchBalance(form, match.out=rr,
  nboots=10, data=ds))
> write(longout[1:20], file = "")

***** (V1) age *****

          Before Matching           After Matching
mean treatment.....      36.368            36.368
mean control.....        35.041            36.423
std mean diff.....      16.069           -0.65642

mean raw eQQ diff.....    1.5981           0.94841
med  raw eQQ diff.....      1                  1
max  raw eQQ diff.....      7                  10

mean eCDF diff.....     0.037112          0.022581
med  eCDF diff.....      0.026365          0.019841
max  eCDF diff.....      0.10477           0.083333

var ratio (Tr/Co).....    1.329            1.2671
T-test p-value.....      0.070785          0.93902
KS Bootstrap p-value..      0.1                0.3
KS Naive p-value.....     0.16881           0.34573
KS Statistic.....        0.10477           0.083333
```

The `capture.output()` function is used to send the voluminous output to a character string, so that only a subset can be displayed. After matching, the age variables had distributions that were considerably closer to each other.

The `Match()` function can also be used to generate a dataset containing only the matched observations (see the `index.treated` and `index.control` components of the `Match` object).

### 11.4.3 Bootstrapping

Bootstrapping is a powerful and elegant approach to estimating the sampling distribution of statistics. It can be implemented even in many situations where asymptotic results are difficult to find or otherwise unsatisfactory [35]. Bootstrapping proceeds using three steps: first, resample the dataset (with replacement) many times (typically on the order of 10,000); then calculate the desired statistic from each resampled dataset; finally, use the distribution of the resampled statistics to estimate the standard error of the statistic (normal approximation method) or construct a confidence interval using quantiles of that distribution (percentile method). There are several ways to easily do bootstrapping in R. In SAS, the best way is to use a suite of SAS macros developed by SAS and available at <http://www.amherst.edu/~nhorton/sasr2/examples/jackboot.sas>.

As an example, we consider estimating the standard error and 95% confidence interval for the coefficient of variation (CV), defined as  $\sigma/\mu$ , for a random variable  $X$ . We'll generate normal data with a mean and variance of 1.

```
/* make some data */
data test;
do i = 1 to 1000;
  x = normal(0) + 1;
  output;
end;
run;

> x = rnorm(1000, mean=1)
```

Note that for both packages, the user must provide code to calculate the statistic of interest; this must be done in a macro (in SAS) or in a function (in R).

The %boot macro in the `jackboot.sas` file requires that you write a %analyze macro, which must generate an output dataset.

```
/* create macro that generates the desired statistic */
%macro analyze(data=, out=);
proc summary data=&data;
  var x;
  output out=&out (drop=_freq_ _type_) cv=cv_x;
run;
%mend;
```

Bootstrap results for all variables in the dataset output from the %analyze macro will be calculated. The `drop` dataset option in the `summary` procedure removes some character variables from this output dataset so that statistics are not reported on them. See 4.2.1 for more information on SAS macros.

In R we define a function to calculate the desired statistic.

```
> covfun = function(x) { # multiply CV by 100
  return(100*sd(x)/mean(x))
}
```

We multiply the statistic by 100 to match SAS.

To read in the `jackboot.sas` file that contains the %boot macro, we use the %include statement, which is equivalent to typing the contents of the file.

```
/* download "jackboot.sas" from
   http://www.amherst.edu/~nhorton/sasr2/examples */
%include 'c:/sasmacros/jackboot.sas';

%boot(data=test, samples=2000, print=0, chart=0);
%bootci(PERCENTILE);
```

The syntax for the `%boot` macro is fairly self-explanatory. The `jackboot.sas` file also has macros for calculating various confidence intervals from the bootstrap estimates. You can generate all available CI with `%allci()` or request specific CI with `%bootci(method)`, where *method* can be one of PERCENTILE, HYBRID, T, BC, or BCA. Here we choose the percentile method because the sampling distribution for the coefficient of variation is nonnormal. Note that `proc summary` calculates the coefficient of variation as  $100 * sd/mean$ .

| Obs | Name | Statistic | Approximate |       | Approximate |       |
|-----|------|-----------|-------------|-------|-------------|-------|
|     |      |           | Observed    | Lower | Confidence  | Upper |
|     |      |           |             | Limit |             | Limit |
| 1   | cv_x | 97.6362   | 91.1355     |       | 104.953     | 95    |

| Obs | Method for           |          | Number of Resamples |
|-----|----------------------|----------|---------------------|
|     | Confidence           | Interval |                     |
| 1   | Bootstrap percentile |          | 2000                |

In R, we use the `do()` function from the `mosaic` package.

```
> options(digits=4)
> covfun(x)

[1] 97.59

> library(mosaic)
> res = do(2000) * covfun(resample(x))
> quantile(res$result, c(.025, .975))

2.5% 97.5%
90.73 105.09
```

The percentile interval is simple to calculate from the observed bootstrapped statistics. If the distribution of the bootstrap samples is approximately normally distributed, a *t* interval could be created by calculating the standard deviation of the bootstrap samples and finding the appropriate multiplier for the confidence interval (more information can be found in the `mosaic` package resampling vignette). Plotting the bootstrap sample estimates is helpful to determine the form of the bootstrap distribution [69]. The `do()` function provides a natural syntax for repetition (see 2.3.3 and `replicate()`). The `boot` package also provides a rich set of routines for bootstrapping, including support for bias corrected and accelerated intervals.

#### 11.4.4 Missing data

##### Account for missing values

Missing values are ubiquitous in most real-world investigations. Both SAS and R feature support for missing value codes, though there are important distinctions which need to be kept in mind by an analyst, particularly when deriving new variables or fitting models.

In SAS, the default missing value code for numeric data is `.`, which has a numeric value of negative infinity. There are 27 other predefined missing value codes (`._`, `.a` ... `.z`), which can be used, for example, to record different reasons for missingness. The missing value code for character data, for assignment, is `" "` (quote blank quote), displayed as a blank.

Listwise deletion is the default behavior for most multivariate procedures in SAS. That is, observations with missing values for any variables named in the procedure are omitted from all calculations. Data step functions are different: functions defined with mathematical

operators (+ - \* \*\*) will result in a missing value if any operand has a missing value, but named functions, such as `mean(x1, x2)` will return the function applied to the nonmissing values.

In R, missing values are denoted by `NA`, a logical constant of length 1 which has no numeric equivalent. The missing value code is distinct from the character string value "NA". The default behavior for most R functions is to return `NA` if any of the input vectors have any missing values.

The following code demonstrates some behavior and functions related to missing data in SAS.

```
data missdemo;
  missing = (x1 eq .);
  x2 = (1 + 2 + .)/3;
  x3 = mean(1, 2, .);

  if x4 = 999 then x4 = .;

  x5 = n(1, 2, 49, 123, .);
  x6 = nmiss(x2, x3);

  x7 = 1;
  if x7 ne .;
  if x7 ne . then output;
run;

proc print data=missdemo;
run;
```

| Obs | missing | x1 | x2 | x3  | x4 | x5 | x6 | x7 |
|-----|---------|----|----|-----|----|----|----|----|
| 1   | 1       | .  | .  | 1.5 | .  | 4  | 1  | 1  |

The variable `missing` has a value of 1 if  $X_1$  is missing, 0 otherwise. Values of  $x_4$  that were previously coded 999 are now marked as missing. The `n` function returns the number of nonmissing values. The `nmiss` function returns the number of missing values among its arguments. The last two statements have identical meanings. They will remove all observations for which  $X_7$  contains a missing value.

A similar exploration for R may be helpful.

```
> x = c(1, 2, NA)
> mean(x)

[1] NA

> mean(x, na.rm=TRUE)

[1] 1.5

> sum(na.omit(x))

[1] 3

> sum(!is.na(x))

[1] 2
```

The `na.rm` option is used within the `mean()` function to override the default behavior, omit missing values, and calculate the result on the complete cases. Many other functions allow

the specification of an `na.action` option (e.g., for the `lm()` function). Common `na.action` functions include `na.exclude()`, `na.omit()`, and `na.fail()` (see also `na.action()` and `options("na.action")`). The `!` (not) Boolean operator allows counting of the number of observed values (since `is.na()` returns a logical set to TRUE if an observation is missing).

The `na.omit()` function returns the dataframe with missing values omitted (if a value is missing for a given row, all observations are removed, also called listwise deletion; see also `naresid()`).

Functions such as `scan()` and `read.table()` have the default argument `na.strings="NA"`. This can be used to recode on input for situations where a numeric missing value code has been used. The `table()` function provides the `exclude=NULL` option to include a category for missing values. R has other kinds of “missing” values, corresponding to floating point standards (see also the `is.infinite()` and `is.nan()` functions).

```
> # remap values of x with missing value code of 999 to missing
> w = c(1, 2, 999)
> w[w==999] = NA
> w
```

```
[1] 1 2 NA
```

or

```
> w = c(1,2,999)
> is.na(w) = w==999 # set 999's to missing
> w
```

```
[1] 1 2 NA
```

Arbitrary numeric missing values (999 in this example) can be mapped to R missing value codes using indexing and assignment. Here all values of `x` that are 999 are replaced by the missing value code of NA.

The `na.pattern()` function in the `Hmisc` package can be used to determine the different patterns of missing values in a dataset.

### Account for missing data using multiple imputation

Here we demonstrate some of the capabilities for fitting incomplete data regression models using multiple imputation [146, 154, 74] implemented with chained equation models [183, 136, 182].

In this example we replicate an analysis from 7.10.1 in a version of the HELP dataset that includes missing values for several of the predictors. While not part of the regression model of interest, the `mcs` and `pcs` variables are included in the imputation models, which may make the missing at random assumption more plausible [28].

In SAS, we begin by importing the data.

```
filename localf "c:/book/helpmiss.csv" lrecl=704;

proc import replace datafile=localf out=help dbms=dlm;
  delimiter=',';
  getnames=yes;
run;
```

Next we use the `mi` procedure without imputing any values, by setting `nimpute=0`. This prints a summary of the missing data patterns.

```

options ls=64;
ods select misspattern;
proc mi data=help n impute=0;
  var homeless female i1 sexrisk indtot mcs pcs;
run;
ods select all;

```

The MI Procedure

#### Missing Data Patterns

| Group | homeless | female | i1 | sexrisk | indtot | mcs | pcs | Freq |
|-------|----------|--------|----|---------|--------|-----|-----|------|
| 1     | X        | X      | X  | X       | X      | X   | X   | 454  |
| 2     | X        | X      | X  | X       | X      | .   | .   | 2    |
| 3     | X        | X      | X  | X       | .      | X   | X   | 13   |
| 4     | X        | X      | X  | .       | .      | X   | X   | 1    |

#### Missing Data Patterns

#### -----Group Means-----

| Group | Percent | homeless | female   | i1        |
|-------|---------|----------|----------|-----------|
| 1     | 96.60   | 0.462555 | 0.237885 | 17.920705 |
| 2     | 0.43    | 1.000000 | 0        | 13.000000 |
| 3     | 2.77    | 0.461538 | 0.230769 | 31.307692 |
| 4     | 0.21    | 1.000000 | 0        | 13.000000 |

#### Missing Data Patterns

#### -----Group Means-----

| Group | sexrisk  | indtot    | mcs       | pcs       |
|-------|----------|-----------|-----------|-----------|
| 1     | 4.638767 | 35.729075 | 31.662403 | 48.018233 |
| 2     | 7.000000 | 35.500000 | .         | .         |
| 3     | 4.153846 | .         | 27.832265 | 49.931599 |
| 4     | .        | .         | 28.452675 | 49.938469 |

Since the pattern of missingness is nonmonotone, our options for imputing within SAS are somewhat limited. In the code below, we impute using fully conditional specification (a chained equations approach). An alternative to proc `mi` would be to use IVEware, a free suite of SAS macros [137].

```

proc mi data=help n impute=20 out=helpmi20 noprint;
  class homeless female;
  fcs logistic regpmm;
  var homeless female i1 sexrisk indtot mcs pcs;
run;

```

By default, the `mi` procedure will use discriminant analyses for listed `class` variables, but only continuous variables can be used as covariates in the discriminant models. Instead, we specify that logistic regressions should be used. It's possible to request different models for each variable and different prediction variables as well. The `regpmm` option specifies that predictive mean matching will be used for the non-`class` variables [67, 157].

The output dataset `helpmi20` has 20 completed versions of the original dataset, along with an additional variable, `_imputation_`, which identifies the completed versions. We use the `by` statement in SAS to fit a logistic regression within each completed dataset.

```

ods select none;
ods output parameterestimates=helpmipe covb=helpmicovb;
proc logistic data=helpmi20 descending;
by _imputation_;
model homeless(event='1')=female i1 sexrisk indtot / covb;
run;
ods select all;

```

Note the use of the ods select none statement to suppress all printed output and the ods output statement to save the parameter estimates and their estimated covariance matrix for use in multiple imputation. These are especially useful to prevent many pages of output being generated.

The multiple imputation inference is performed in proc mianalyze.

```

ods select varianceinfo;
proc mianalyze parms=helpmipe covb=helpmicovb;
modeleffects intercept female i1 sexrisk indtot;
run;

```

#### The MIANALYZE Procedure

##### Variance Information

| -----Variance----- |              |             |             |        |
|--------------------|--------------|-------------|-------------|--------|
| Parameter          | Between      | Within      | Total       | DF     |
| intercept          | 0.007286     | 0.346758    | 0.354408    | 40776  |
| female             | 0.000150     | 0.059400    | 0.059557    | 2.7E6  |
| i1                 | 3.8889244E-8 | 0.000031467 | 0.000031508 | 1.13E7 |
| sexrisk            | 0.000005046  | 0.001277    | 0.001282    | 1.11E6 |
| indtot             | 0.000006035  | 0.000245    | 0.000251    | 29817  |

##### Variance Information

| Parameter | in | Relative | Fraction | Efficiency |
|-----------|----|----------|----------|------------|
|           |    | Increase | Missing  |            |
| intercept |    | 0.022062 | 0.021634 | 0.998919   |
| female    |    | 0.002659 | 0.002653 | 0.999867   |
| i1        |    | 0.001298 | 0.001296 | 0.999935   |
| sexrisk   |    | 0.004150 | 0.004135 | 0.999793   |
| indtot    |    | 0.025897 | 0.025308 | 0.998736   |

```

options ls = 68;
ods select parameterestimates;
proc mianalyze parms=helpmipe covb=helpmicovb;
  modeleffects intercept female i1 sexrisk indtot;
run;
ods select all;

```

The MIANALYZE Procedure

#### Parameter Estimates

| Parameter | Estimate  | Std Error | 95% Confidence Limits |
|-----------|-----------|-----------|-----------------------|
| intercept | -2.523978 | 0.595322  | -3.69082 -1.35713     |
| female    | -0.244341 | 0.244044  | -0.72266 0.23398      |
| i1        | 0.023114  | 0.005613  | 0.01211 0.03412       |
| sexrisk   | 0.059424  | 0.035804  | -0.01075 0.12960      |
| indtot    | 0.048737  | 0.015844  | 0.01768 0.07979       |

#### Parameter Estimates

| Parameter | DF     | Minimum   | Maximum   |
|-----------|--------|-----------|-----------|
| intercept | 40776  | -2.703885 | -2.369444 |
| female    | 2.7E6  | -0.269855 | -0.225265 |
| i1        | 1.13E7 | 0.022717  | 0.023494  |
| sexrisk   | 1.11E6 | 0.055376  | 0.064592  |
| indtot    | 29817  | 0.044616  | 0.053491  |

#### Parameter Estimates

##### t for H0:

| Parameter | Theta0 | Parameter=Theta0 | Pr >  t |
|-----------|--------|------------------|---------|
| intercept | 0      | -4.24            | <.0001  |
| female    | 0      | -1.00            | 0.3167  |
| i1        | 0      | 4.12             | <.0001  |
| sexrisk   | 0      | 1.66             | 0.0970  |
| indtot    | 0      | 3.08             | 0.0021  |

Let's compare with the complete case analysis, the SAS default.

```

ods select parameterestimates;
proc logistic data=help;
  model homeless(event='1')=female i1 sexrisk indtot;
run;
ods select all;

```

The LOGISTIC Procedure

#### Analysis of Maximum Likelihood Estimates

| Parameter | DF | Estimate | Error   | Standard   |            | Wald | Pr > ChiSq |
|-----------|----|----------|---------|------------|------------|------|------------|
|           |    |          |         | Chi-Square | Pr > ChiSq |      |            |
| Intercept | 1  | -2.5278  | 0.5965  | 17.9575    | <.0001     |      |            |
| female    | 1  | -0.2401  | 0.2473  | 0.9426     | 0.3316     |      |            |
| i1        | 1  | 0.0232   | 0.00578 | 16.1050    | <.0001     |      |            |
| sexrisk   | 1  | 0.0562   | 0.0363  | 2.4012     | 0.1212     |      |            |
| indtot    | 1  | 0.0493   | 0.0158  | 9.6974     | 0.0018     |      |            |

In R we begin by reading in the data then using the `na.pattern()` function from the `Hmisc` package to characterize the patterns of missing values.

```

> ds =
  read.csv("c:/book/helpmiss.csv")
> smallds = with(ds, data.frame(homeless, female, i1, sexrisk, indtot,
  mcs, pcs))

> summary(smallds)

   homeless      female       i1      sexrisk
Min.   :0.000   Min.   :0.000   Min.   : 0.0   Min.   : 0.00
1st Qu.:0.000   1st Qu.:0.000   1st Qu.: 3.0   1st Qu.: 3.00
Median :0.000   Median :0.000   Median :13.0   Median : 4.00
Mean   :0.466   Mean   :0.236   Mean   :18.3   Mean   : 4.64
3rd Qu.:1.000   3rd Qu.:0.000   3rd Qu.:26.0   3rd Qu.: 6.00
Max.   :1.000   Max.   :1.000   Max.   :142.0  Max.   :14.00
                           NA's   :1

  indtot        mcs         pcs
Min.   : 4.0   Min.   : 6.8   Min.   :14.1
1st Qu.:32.0  1st Qu.:21.7  1st Qu.:40.3
Median :37.5  Median :28.6  Median :48.9
Mean   :35.7  Mean   :31.5  Mean   :48.1
3rd Qu.:41.0  3rd Qu.:40.6  3rd Qu.:57.0
Max.   :45.0  Max.   :62.2  Max.   :74.8
NA's   :14    NA's   :2     NA's   :2

> library(Hmisc)
> na.pattern(smallds)

pattern
0000000 0000011 0000100 0001100
      454      2      13      1

```

There are 14 subjects missing `indtot`, 2 missing `mcs` as well as `pcs`, and 1 missing `sexrisk`. In terms of patterns of missingness, there are 454 observations with complete data, 2 missing both `mcs` and `pcs`, 13 missing `indtot` alone, and 1 missing `sexrisk` and `indtot`. Fitting a logistic regression model (7.1.1) using the available data ( $n=456$ ) yields the following results.

```

> glm(homeless ~ female + i1 + sexrisk + indtot, binomial,
  data=smallds)

Call: glm(formula = homeless ~ female + i1 + sexrisk + indtot,
  family = binomial, data = smallds)

```

#### Coefficients:

| (Intercept) | female  | i1     | sexrisk | indtot |
|-------------|---------|--------|---------|--------|
| -2.5278     | -0.2401 | 0.0232 | 0.0562  | 0.0493 |

Degrees of Freedom: 455 Total (i.e. Null); 451 Residual  
 (14 observations deleted due to missingness)

Null Deviance: 630

Residual Deviance: 586 AIC: 596

Next, the `mice()` function within the `mice` package is used to impute missing values for `sexrisk`, `indtot`, `mcs`, and `pcs`. These results are combined using `glm.mids()`, and results are pooled and reported. Note that by default, all variables within the `smallds` data frame

are included in each of the chained equations (so that `mcs` and `pcs` are used as predictors in each of the imputation models).

```
> library(mice)
> imp = mice(smallds, m=20, maxit=25, seed=42, print=FALSE)
> summary(pool(glm.mids(homeless ~ female + i1 + sexrisk +
+ indtot, family=binomial, data=imp)))
```

|             | est     | se       | t     | df  | Pr(> t ) | lo      | 95      | hi | 95 | nmis |
|-------------|---------|----------|-------|-----|----------|---------|---------|----|----|------|
| (Intercept) | -2.5238 | 0.59165  | -4.27 | 457 | 2.42e-05 | -3.6865 | -1.3611 | NA |    |      |
| female      | -0.2449 | 0.24379  | -1.00 | 463 | 3.16e-01 | -0.7239 | 0.2342  | 0  |    |      |
| i1          | 0.0232  | 0.00561  | 4.14  | 463 | 4.20e-05 | 0.0122  | 0.0342  | 0  |    |      |
| sexrisk     | 0.0587  | 0.03580  | 1.64  | 461 | 1.02e-01 | -0.0116 | 0.1291  | 1  |    |      |
| indtot      | 0.0488  | 0.01572  | 3.10  | 456 | 2.04e-03 | 0.0179  | 0.0797  | 14 |    |      |
|             | fmi     | lambda   |       |     |          |         |         |    |    |      |
| (Intercept) | 0.01446 | 0.010155 |       |     |          |         |         |    |    |      |
| female      | 0.00537 | 0.001076 |       |     |          |         |         |    |    |      |
| i1          | 0.00513 | 0.000836 |       |     |          |         |         |    |    |      |
| sexrisk     | 0.00898 | 0.004683 |       |     |          |         |         |    |    |      |
| indtot      | 0.01610 | 0.011793 |       |     |          |         |         |    |    |      |

The summary includes the number of missing observations as well as the fraction of missing information (fmi). While the results are qualitatively similar, they do differ, which is not surprising given the different imputation models used. Support for other missing data models is available in the `mix` and `mitools` packages.

### 11.4.5 Finite mixture models with concomitant variables

Finite mixture models (FMMs) can be used in settings where some unmeasured classification separates the observed data into groups with different exposure/outcome relationships. One familiar example of this is a zero-inflated model (7.2.1), where some observations come from a degenerate distribution with all mass at 0. In that case the exposure/outcome relationship is less interesting in the degenerate distribution group, but there would be considerable interest in the estimated probability of group membership. Another possibly familiar setting is the estimation of a continuous density as a mixture of normal distributions.

More generally, there could be several groups, with “concomitant” covariates predicting group membership. Each group might have different sets of predictors and outcomes from different distribution families. On the other hand, in a “homogeneous” mixture setting, all groups have the same distributional form, but with different parameter values. If the covariates in the model are the same, this setting is similar to an ordinary regression model where every observed covariate is interacted with the (unobserved) group membership variable.

We’ll demonstrate with a simulated dataset. We create a variable `x` that predicts both group membership and an outcome `y` with different linear regression parameters depending on group. The mixing probability follows a logistic regression with intercept = -1 and slope (log odds ratio) = 2.

The intercept and slope for the outcome are (0, 1) and (3, 1.2) for the groups, respectively. We leave as an exercise for the reader to explore the consequences of naively fitting the model which ignores the mixture.

In SAS, we use the `fmm` procedure to fit these models. As hinted at above, the generality implied by the models is fairly vast, and the FMM procedure includes a lot of this generality. The most obvious limitation to `proc fmm` is that it requires independent observations.

```

data fmmtest;
do i = 1 to 5000;
  x = normal(1492);
  group = (exp(-1 + 2*x)/(1 + exp(-1 + 2*x))) gt uniform(0);
  y = (group * 3) + ((1 + group/5) * x) + normal(0) * sqrt(1);
  output;
end;
run;

```

Then we can use proc fmm to fit the model.

```

ods select parameterestimates mixingprobs;
proc fmm data=fmmtest;
  model y = x / k=2 equate=scale;
  probmodel x;
run;

```

In the **model** statement, the option **k** defines how many groups (or “components”) are to be included. There are also options **kmin** and **kmax** which will cause to be fit models with various numbers of components and report results of one of them based on some desired criterion. The **equate** option allows the user to force some elements of the component distributions to be equal. Here we force the residual variance to be equal, since that’s the model that generated our data. The **probmodel** statement describes the concomitant model. The default settings model a logistic (or generalized logit) regression using the listed covariates. The parameter estimates for the components are shown in the **parameterestimates** output.

#### The FMM Procedure

##### Parameter Estimates for 'Normal' Model

| Component | Effect    | Standard |         |         |         |
|-----------|-----------|----------|---------|---------|---------|
|           |           | Estimate | Error   | z Value | Pr >  z |
| 1         | Intercept | 2.9685   | 0.04655 | 63.77   | <.0001  |
| 1         | x         | 1.2326   | 0.03803 | 32.41   | <.0001  |
| 2         | Intercept | -0.02270 | 0.02474 | -0.92   | 0.3590  |
| 2         | x         | 1.0132   | 0.02386 | 42.47   | <.0001  |
| 1         | Variance  | 0.9942   | 0.02436 |         |         |
| 2         | Variance  | 0.9942   | 0.02436 |         |         |

Note that the routine has done a great job of estimating the parameters we input for each component. The parameter estimates for the concomitant model are shown in the **mixingprobs** output.

#### The FMM Procedure

##### Parameter Estimates for Mixing Probabilities

| Effect    | Standard |         |         |         |
|-----------|----------|---------|---------|---------|
|           | Estimate | Error   | z Value | Pr >  z |
| Intercept | -0.9691  | 0.05562 | -17.42  | <.0001  |
| x         | 1.9987   | 0.07593 | 26.32   | <.0001  |

Since we used the default logistic regression, these are the intercept and log odds ratio for group membership; again, the parameters used to simulate the data have been recovered with admirable fidelity.

Simulating the data in R is similar to the process in SAS.

```
> set.seed(1492)
> x = rnorm(5000)
> probgroup1 = exp(-1 + 2*x)/(1 + exp(-1 + 2*x))
> group = ifelse(probgroup1 > runif(5000), 1, 0)
> y = (group * 3) + ((1 + group/5) * x) + rnorm(5000)
```

To fit the model, we'll use the `flexmix` package, a quite general tool written by Gruen, Leisch [98], and Sarkar (other options are described in the CRAN finite mixture models task view).

```
> library(flexmix)
> mixout.fm=flexmix(y ~ x, k=2, model=FLXMRglmfix(y ~ x, varFix=TRUE),
  concomitant=FLXPmultinom(~ x))
```

The `flexmix()` function uses a variety of special objects that are created by other functions the package provides. Here we use the `FLXMRglmfix()` function to force equal variances across the components and the `FLXPmultinom()` function to define the logistic regression on the covariate `x` for the concomitant model.

The results can be generated from the output object with the `parameters()` function. By default, it prints the parameters for the model in each component.

```
> parameters(mixout.fm)

      Comp.1   Comp.2
coef.(Intercept) 3.010 -0.0301
coef.x           1.167  0.9900
sigma            0.999  0.9988
```

Using the `which="concomitant"` option generates the parameter estimates for the concomitant model.

```
> parameters(mixout.fm, which="concomitant")

      1       2
(Intercept) 0  0.999
x           0 -1.906
```

As with the SAS results, impressive accuracy has been achieved. Note that the concomitant variable model is predicting membership in the other group, as compared with SAS, so the signs are reversed from the generating model.

## 11.5 Further resources

Comprehensive descriptions of reproducible analysis tools and workflow can be found in [203] and [50]. Gelman et al. [52] is an accessible introduction to Bayesian inference, while Albert [5] focuses on the use of R for Bayesian computations. Rubin's review [146] and Schafer's book [154] provide overviews of multiple imputation, while [183, 136, 182] describe chained equation models. Review of software implementations of missing data models can be found in [75, 74].



# Chapter 12

## Case studies

In this chapter, we explore several case studies that demonstrate the statistical computing strengths and potential of these two packages. This includes data management tasks, reading more complex files, creating maps, data scraping, manipulating larger datasets, and an optimization problem.

### 12.1 Data management and related tasks

#### 12.1.1 Finding two closest values in a vector

Suppose we need to find the closest pair of observations for some variable. This might arise if we were concerned that some data had been accidentally duplicated. In this case study, we return the IDs of the two closest observations and their distance from each other. We'll first create some sample data and sort it, recognizing that the smallest difference must come between two observations adjacent after sorting.

We begin by generating data (3.1.6), along with some subject identifiers (2.3.4).

```
data ds;
do id = 1 to 8;
  x = normal(42);
  output;
end;
run;

> options(digits=3)
> ds = data.frame(x=rnorm(8), id=1:8)
```

Then, we sort the data. In SAS, we use `proc sort`. In R, the `order()` function (2.3.10) is used to keep track of the sorted random variables.

```
proc sort data=ds; by x; run;
```

```
proc print data=ds; run;
```

| Obs | id | x        |
|-----|----|----------|
| 1   | 8  | -1.79681 |
| 2   | 4  | -0.32100 |
| 3   | 3  | -0.20874 |
| 4   | 1  | -0.02860 |
| 5   | 7  | 0.25504  |
| 6   | 2  | 0.53803  |
| 7   | 5  | 0.62084  |
| 8   | 6  | 1.33267  |

```
> options(digits=3)
> ds = ds[order(ds$x),]
> ds
```

| x | id      |
|---|---------|
| 8 | -2.3031 |
| 6 | -1.5407 |
| 2 | -0.9816 |
| 1 | -0.3854 |
| 3 | -0.0208 |
| 7 | 0.0718  |
| 5 | 0.1008  |
| 4 | 1.1977  |

To find the smallest distance between the sorted observations in SAS we'll use data step programming. The approach is somewhat complex, and we annotate within the code. A proc sql solution would be shorter.

```
data smallest;
set ds end=eof;      /* eof is a variable; = 1 if last row */
retain smalldist last lastid smallid largeid;
/* variables we keep between processing observations
   in the ds dataset */
if _n_ gt 1 then do; /* no diff for the first obs */
   smalldist = min(smalldist, x - last);
   /* is distance smaller now than last time? */
   if smalldist = x - last then do;      /* if so */
      smallid = lastid;
      /* the current smaller id is the one */
      /* from the last row */
   largeid = id;
   /* and the current larger id is the */
   /* one in this row */
end;
last = x;            /* if this row turns out to be the */
lastid = id;          /* smaller end of the smallest pair */
/* we'll need to know x and the id */
if eof then output;
run;
```

The explicit use of `output` in the last line means that SAS does not output a row into the `smallest` dataset except when the condition is true. In this case, that is when we reach the last row of the `ds` dataset.

```
proc print data=smallest;
  var smalldist smallid largeid;
run;
```

| Obs | smalldist | smallid | largeid |
|-----|-----------|---------|---------|
| 1   | 0.082809  | 2       | 5       |

In R we can use the `diff()` function to get the differences between observations. The `which.min()` function extracts the index (location within the vector) of the smallest value. We apply this function to the `diffx` vector to find the location and extract that location from the `id` vector.

```
> diffx = diff(ds$x)
> min(diffx)

[1] 0.0289

> with(ds, id[which.min(diffx)]) # first val
[1] 7

> with(ds, id[which.min(diffx) + 1]) # second val
[1] 5
```

### 12.1.2 Tabulate binomial probabilities

Suppose we wanted to assess the probability  $P(X = x)$  for a binomial random variate with  $n = 10$  and with  $p = .81, .84, \dots, .99$ . This could be helpful, for example, in various game settings.

In SAS, we find the probability that  $X = x$  using differences in the CDF calculated via the `cdf` function (3.1.1). We loop through the various binomial probabilities  $p$  and observed successes  $j$  using the `do ... end` structure (4.1.1). Finally, we store the probabilities in appropriate variables via an `array` statement (4.1.5).

```
data table (drop=j);
array probs [11] prob0 prob1 - prob10;
do p = .81 to .99 by .03;
  do j = 0 to 10;
    if j eq 1 then probs[j+1] = cdf("BINOMIAL", 0, p, 10);
    else probs[j+1] = cdf("BINOMIAL", j, p, 10)
      - cdf("BINOMIAL", j-1, p, 10);
  end;
  output;
end;
run;
```

Note that the use of the single dash (-) in the `array` statement causes the variables `prob1`, `prob2`, ..., `prob10` to both be created and included in the array.

```

options ls=64;
proc print data=table noobs;
  var p prob0 prob1 - prob10;
  format _numeric_ 3.2;
run;

          p
      p   p   p   p   p   p   p   p   p   p   r
      r   r   r   r   r   r   r   r   r   r   o
      o   o   o   o   o   o   o   o   o   o   b
      b   b   b   b   b   b   b   b   b   b   1
p   0   1   2   3   4   5   6   7   8   9   0
.81 .00 .00 .00 .00 .02 .08 .19 .30 .29 .12
.84 .00 .00 .00 .00 .01 .05 .15 .29 .33 .17
.87 .00 .00 .00 .00 .00 .03 .10 .25 .37 .25
.90 .00 .00 .00 .00 .00 .01 .06 .19 .39 .35
.93 .00 .00 .00 .00 .00 .00 .02 .12 .36 .48
.96 .00 .00 .00 .00 .00 .00 .01 .05 .28 .66
.99 .00 .00 .00 .00 .00 .00 .00 .00 .09 .90

```

In R, we make a vector of the binomial probabilities, using the `:` operator (2.3.4) to generate a sequence of integers. After creating an empty matrix (3.3) to hold the table results, we loop (4.1.1) through the binomial probabilities, calling the `dbinom()` function (3.1.1) to find the probability that the random variable takes on that particular value. This calculation is nested within the `round()` function (3.2.4) to reduce the digits displayed. Finally, we include the vector of binomial probabilities with the results using `cbind()`.

```

> p = .78 + (3 * 1:7)/100
> allprobs = matrix(nrow=length(p), ncol=11)
> for (i in 1:length(p)) {
+   allprobs[i,] = round(dbinom(0:10, 10, p[i]),2)
+ }
> table = cbind(p, allprobs)
> table

```

|      | p    | 0    | 1    | 2    | 3    | 4    | 5    | 6    | 7    | 8    | 9 |
|------|------|------|------|------|------|------|------|------|------|------|---|
| [1,] | 0.81 | 0.00 | 0.00 | 0.00 | 0.02 | 0.08 | 0.19 | 0.30 | 0.29 | 0.12 |   |
| [2,] | 0.84 | 0.00 | 0.00 | 0.00 | 0.01 | 0.05 | 0.15 | 0.29 | 0.33 | 0.17 |   |
| [3,] | 0.87 | 0.00 | 0.00 | 0.00 | 0.00 | 0.03 | 0.10 | 0.25 | 0.37 | 0.25 |   |
| [4,] | 0.90 | 0.00 | 0.00 | 0.00 | 0.00 | 0.01 | 0.06 | 0.19 | 0.39 | 0.35 |   |
| [5,] | 0.93 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.02 | 0.12 | 0.36 | 0.48 |   |
| [6,] | 0.96 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.01 | 0.05 | 0.28 | 0.66 |   |
| [7,] | 0.99 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.09 | 0.90 |   |

### 12.1.3 Calculate and plot a running average

The “Law of Large Numbers” concerns the convergence of the arithmetic average to the expected value, as sample sizes increase. This is an important topic in mathematical statistics. The convergence (or lack thereof, for certain distributions) can easily be visualized [72].

In SAS, we’ll write a macro to generate the running average for a given distribution, given a number of observations.

```
%macro runave(outdata=, n=, distparms=) ;
data &outdata;
call streaminit(1984);
do i = 1 to &n;
  x = rand &distparms;
  runtot = sum(x, runtot);
  avex = runtot/i;
  output;
end;
run;
%mend runave;
```

In order to keep the macro as generic as possible, we'll use the `rand` function (3.1). The specification of the desired distribution is passed to the macro via the `distparms` parameter.

In R, we define a function (4.2.2) to calculate the running average for a given vector, allowing for variates from many distributions to be generated.

```
> runave = function(n, gendist, ...) {
  x = gendist(n, ...)
  avex = numeric(n)
  for (k in 1:n) {
    avex[k] = mean(x[1:k])
  }
  return(data.frame(x, avex))
}
```

The `runave()` function takes at a minimum two arguments: a sample size `n` and function (4.2.2) denoted by `gendist` that is used to generate samples from a distribution (3.1). In addition, other options for the function can be specified, using the `...` syntax (see 4.2.2). This is used, for example, to specify the degrees of freedom for the samples generated for the *t* distribution in the next code block. The loop in the `runave()` function could be eliminated through use of the `cumsum()` function applied to the vector given as argument, and then divided by a vector of observation numbers.

Next, we generate the data, using our new macro and function. To make sure we have a nice example, we first set a fixed seed (3.1.3). Recall that because the expectation of a Cauchy random variable is undefined [143] the sample average does not converge to the center, while a *t* distribution with more than 1 df does.

```
libname k 'c:/temp';
%runave(outdata=k.cauchy, n=1000, distparms= ("CAUCHY"));
%runave(outdata=k.t4, n=1000, distparms= ('T',4));
> vals = 1000
> set.seed(1984)
> cauchy = runave(vals, rcauchy)
> t4 = runave(vals, rt, 4)
```

Now we can plot the results. In SAS we put the two datasets together using a `set` statement. The `in=` option lets us identify which dataset each observation came from. We use this to generate the `dist` variable that describes the distribution. To plot the two running averages, we use the `plot a * b = c` syntax (9.1.2) to treat the two values of `dist` separately. The `symbol` statement (9.2.11) is used to draw lines connecting the consecutive values with different line types.

```

data c_t4;
set k.cauchy k.t4 (in=t4);
if t4 then dist="t with 4 df";
else dist="Cauchy";
run;

symbol1 i=j c=black l=1 w=3 v=none;
symbol2 i=j c=black l=21 w=3 v=none;
proc gplot data=c_t4;
plot avex * i=dist / vref=0;
run; quit;

```

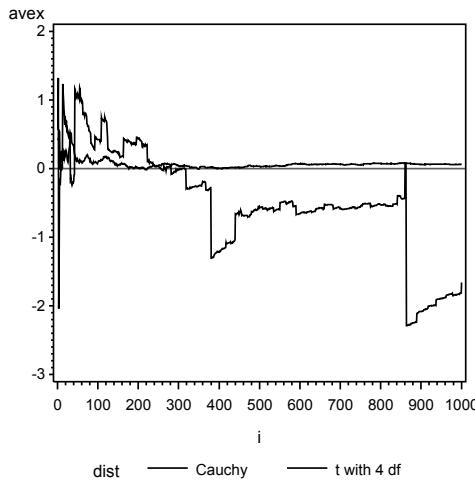
The `vref=0` option in the `plot` statement draws a horizontal reference line at 0 on the vertical axis. Note that SAS produces a legend by default.

In R, we begin with an empty plot with the correct axis limits, using the `type="n"` specification (8.3.1). We add the running average using the `lines()` function (9.1.1) and varying the line style (9.2.11) and thickness (9.2.12) with the `lty` and `lwd` specifications, respectively. Finally we specify a title (9.1.9) and a legend (9.1.15). The results are displayed in Figure 12.1.

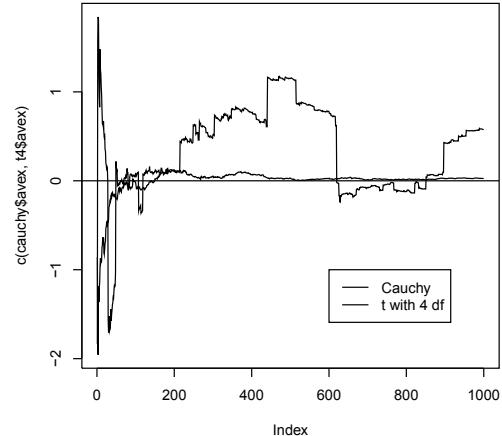
```

> plot(c(cauchy$avex, t4$avex), xlim=c(1, vals), type="n")
> lines(1:vals, cauchy$avex, lty=1, lwd=2)
> lines(1:vals, t4$avex, lty=2, lwd=2)
> abline(0, 0)
> legend(vals*.6, -1, legend=c("Cauchy", "t with 4 df"),
  lwd=2, lty=c(1, 2))

```



(a) SAS



(b) R

Figure 12.1: Running average for Cauchy and  $t$  distributions

### 12.1.4 Create a Fibonacci sequence

The Fibonacci numbers have many mathematical relationships and have been discovered repeatedly in nature. They are constructed as the sum of the previous two values, initialized with the values 1 and 1.

In SAS we can generate the sequence in a data step.

```
data fibo;
do i = 1 to 10;
    fib = sum(fib, lag(fib));
    if i eq 1 then fib = 1;
    output;
end;
run;

proc print data=fibo;
run;
```

| Obs | i  | fib |
|-----|----|-----|
| 1   | 1  | 1   |
| 2   | 2  | 1   |
| 3   | 3  | 2   |
| 4   | 4  | 3   |
| 5   | 5  | 5   |
| 6   | 6  | 8   |
| 7   | 7  | 13  |
| 8   | 8  | 21  |
| 9   | 9  | 34  |
| 10  | 10 | 55  |

In R, it's convenient to use a `for` loop, though other approaches (e.g., recursion) could be used.

```
> len = 10
> fibvals = numeric(len)
> fibvals[1] = 1
> fibvals[2] = 1
> for (i in 3:len) {
+     fibvals[i] = fibvals[i-1] + fibvals[i-2]
+ }
> fibvals
[1] 1 1 2 3 5 8 13 21 34 55
```

## 12.2 Read variable format files

Sometimes datasets are stored in variable format. For example, US Census boundary files (available from <http://www.census.gov/geo/www/cob/index.html>) are available in both proprietary and ASCII formats. An example ASCII file describing the counties of Massachusetts is available on the book web site (<http://www.amherst.edu/~nhorton/sasr2>). The first few lines are reproduced here.

```
1      -0.709816806854972E+02      0.427749187746914E+02
-0.709148990000000E+02      0.428865890000000E+02
-0.709148860000000E+02      0.428865640000000E+02
-0.709148860000000E+02      0.428865640000000E+02
-0.709027680000000E+02      0.428865300000000E+02
...
-0.709148990000000E+02      0.428865890000000E+02
END
```

The first line contains an identifier for the county (linked with a county name in an additional file) and a latitude and longitude centroid within the polygon representing the county defined by the remaining points. The remaining points on the boundary do not contain the identifier. After the lines with the points, a line containing the word “END” is included. In addition, the county boundaries contain different numbers of points.

Reading this kind of data requires some care in programming. We begin with SAS.

```
filename census1
  "c:/book/co25_d00.dat";

data pcts cents;
infile census1;
retain cntyid;
input @1 endind $3. @; /* the trailing '@' means to keep this line */
if endind ne 'END' then do;
  input @7 neglat $1. @; /* if this line does not say 'END', then
                           check to see if the 7th character is '-' */
  if neglat eq '-' then do; /* if so, it has a boundary point */
    input @7 x y;
    const = 1;             /* Used as choropleth value in the map */
    output pcts;           /* write out to boundary dataset */
  end;
else if neglat ne '-' then do; /* if not, it must be the centroid */
  input @9 cntyid 2. x y ;
  output cents;           /* write it to the centroid dataset */
end;
run;
```

Two datasets are defined in the `data` statement, and explicit `output` statements are used to specify which lines are output to which dataset. The `@` designates the position on the line which is to be read and also “holds” the line for further reading after the end of an `input` statement.

The county names, which can be associated by the county identifier, are stored in another dataset.

```
filename census2
  "c:/book/co25_d00a.dat";

data cntynames;
infile census2 DSD;
format cntyname $17. ;
input cntyid 2. cntyname $;
run;
```

To get the names onto the map, we have to merge the centroid location dataset with the county names dataset. They have to be sorted first.

```
proc sort data=cntynames; by cntyid; run;
proc sort data=cents; by cntyid; run;
```

Note that in the preceding code we depart from the convention of requiring a new line for every statement; simple procedures like these are a convenient place to reduce the line length of the code.

In R we begin by reading in all of the input lines, keeping track of how many counties have been observed (based on how many lines include END). This information is needed for housekeeping purposes when collecting map points for each county.

```
> # read in the data
> input =
  readLines("c:/book/co25_d00.dat",
            n=-1)
> # figure out how many counties, and how many entries
> num = length(grep("END", input))
> allvals = length(input)
> numentries = allvals-num
> # create vectors to store data
> county = numeric(numentries);
> lat = numeric(numentries)
> long = numeric(numentries)
```

Each line of the input file is processed in turn.

```
> curval = 0  # number of counties seen so far
> # loop through each line
> for (i in 1:allvals) {
  if (input[i]=="END") {
    curval = curval + 1
  } else {
    # remove extraneous spaces
    nospace = gsub("[ ]+", " ", input[i])
    # remove space in first column
    nospace = gsub("^ ", "", nospace)
    splitstring = as.numeric(strsplit(nospace, " ")[[1]])
    len = length(splitstring)
    if (len==3) { # new county
      curcounty = splitstring[1]; county[i-curval] = curcounty
      lat[i-curval] = splitstring[2]; long[i-curval] = splitstring[3]
    } else if (len==2) { # continue current county
      county[i-curval] = curcounty; lat[i-curval] = splitstring[1]
      long[i-curval] = splitstring[2]
    }
  }
}
```

The `strsplit()` function is used to parse the input file. Lines containing `END` require incrementing the count of counties seen to date. If the line indicates the start of a new county, the new county number is saved. If the line contains two fields (another set of latitudes and longitudes), then this information is stored in the appropriate index (`i-curval`) of the output vectors.

Next we read in a dataset of county names. Later we'll plot the Massachusetts counties and annotate the plot with the names of the counties.

```
> # read county names
> countynames =
  read.table("c:/book/co25_d00a.dat",
             header=FALSE)
> names(countynames) = c("county", "countyname")
```

## 12.3 Plotting maps

### 12.3.1 Massachusetts counties, continued

In SAS, we're ready to merge the two datasets. At the same time, we'll include the variables needed by the `annotate` facility to put data from the dataset onto the map. The variables `function`, `style`, `color`, `position`, `when`, `size`, and the `?sys` variables all describe aspects of the text to be placed onto the plot.

```
data nameloc;
length function style color $ 8 position $ 1 text $ 20;
retain xsy s ysy "2" hsys "3" when "a";
merge cntynames cents;
by cntyid;
  function="label"; style="swiss"; text=cntyname; color="black";
  size=3; position="5";
  output;
run;
```

Finally, we can make the map. The `annotate` option (9.1) tells SAS to use the named dataset to mark up the map.

```
pattern1 value=empty;
proc gmap map=pcts data=pcts;
  choro const / nolegend coutline=black annotate=nameloc;
  id cntyid;
run; quit;
```

The `pattern` statement can be used to control the fill colors when creating choropleth maps. Here we specify that no fill is needed. Results are displayed in Figure 12.2.

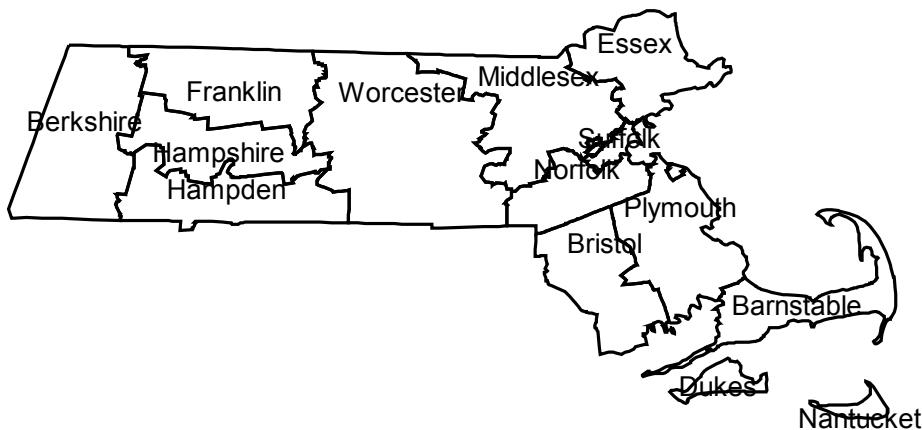


Figure 12.2: Massachusetts counties

To create the map in R, we begin by determining the plotting region, creating the plot of boundaries, then adding the county names at the internal point that was provided. Since the first set of points is in the interior of the county, these are not included in the values given to the `polygon` function (see indexing, B.4.2).

```

> xvals = c(min(lat), max(lat))
> yvals = c(range(long))
> plot(xvals, yvals, pch=" ", xlab="", ylab="", xaxt="n", yaxt="n")
> counties = unique(county)
> for (i in 1:length(counties)) {
  # first element is an internal point
  polygon(lat[county==counties[i]][-1], long[county==counties[i]][-1])
  # plot name of county using internal point
  text(lat[county==counties[i]][1], long[county==counties[i]][1],
       countynames$countyname[i])
}

```

The plots from SAS and R differ only with respect to the default font used, so we display the results only once, in Figure 12.2. Many other maps as well as more sophisticated projections are supported with the `maps` package (see also the CRAN spatial statistics task view).

### 12.3.2 Bike ride plot

The Pioneer Valley of Massachusetts, where we both live, is a wonderful place to take a bike ride. In combination with technology to track GPS coordinates, time, and altitude, information regarding outings can be displayed. The data used here can be downloaded, as demonstrated below, from <http://www.amherst.edu/~nhorton/sasr2/datasets/cycle.csv>.

In R, a map can be downloaded for a particular area from Google Maps, then plotted in conjunction with latitude/longitude coordinates using functions in the `ggmap` package. These routines are built on top of the `ggplot2` (grammar of graphics) package. We found good locations for Amherst using trial and error and plotted the bike ride GPS signals with the map.

```
> library(ggmap)
> options(digits=4)
> amherst = c(lon=-72.52, lat=42.36)
> mymap = get_map(location=amherst, zoom=13, color="bw")
> myride =
    read.csv("http://www.amherst.edu/~nhorton/sasr2/datasets/cycle.csv")
> head(myride, 2)
```

|   | Time                | Ride.Time    | Ride.Time..secs. | Stopped.Time            |                  |
|---|---------------------|--------------|------------------|-------------------------|------------------|
| 1 | 2010-10-02 16:26:54 | 0:00:01      | 0.9              | 0:00:00                 |                  |
| 2 | 2010-10-02 16:27:52 | 0:00:59      | 58.9             | 0:00:00                 |                  |
|   | Stopped.Time..secs. | Latitude     | Longitude        | Elevation..feet.        | Distance..miles. |
| 1 | 0                   | 42.32        | -72.51           | 201                     | 0.00             |
| 2 | 0                   | 42.32        | -72.51           | 159                     | 0.04             |
|   | Speed..miles.h.     | Pace         | Pace..secs.      | Average.Speed..miles.h. | Average.Pace     |
| 1 | NA                  | NA           |                  | 0.00                    | 0:00:00          |
| 2 | 2.73                | 0:21:56      | 1316             | 2.69                    | 0:22:17          |
|   | Average.Pace..secs. | Climb..feet. | Calories         |                         |                  |
| 1 | 0                   | 0            | 0                |                         |                  |
| 2 | 1337                | 0            | 1                |                         |                  |

```
> ggrepel::geom_text_repel + geom_point(aes(x=Longitude, y=Latitude), data=myride)
```

The results are shown in Figure 12.3. Relatively poor cell phone service leads to sparsity in the points in the middle of the figure.

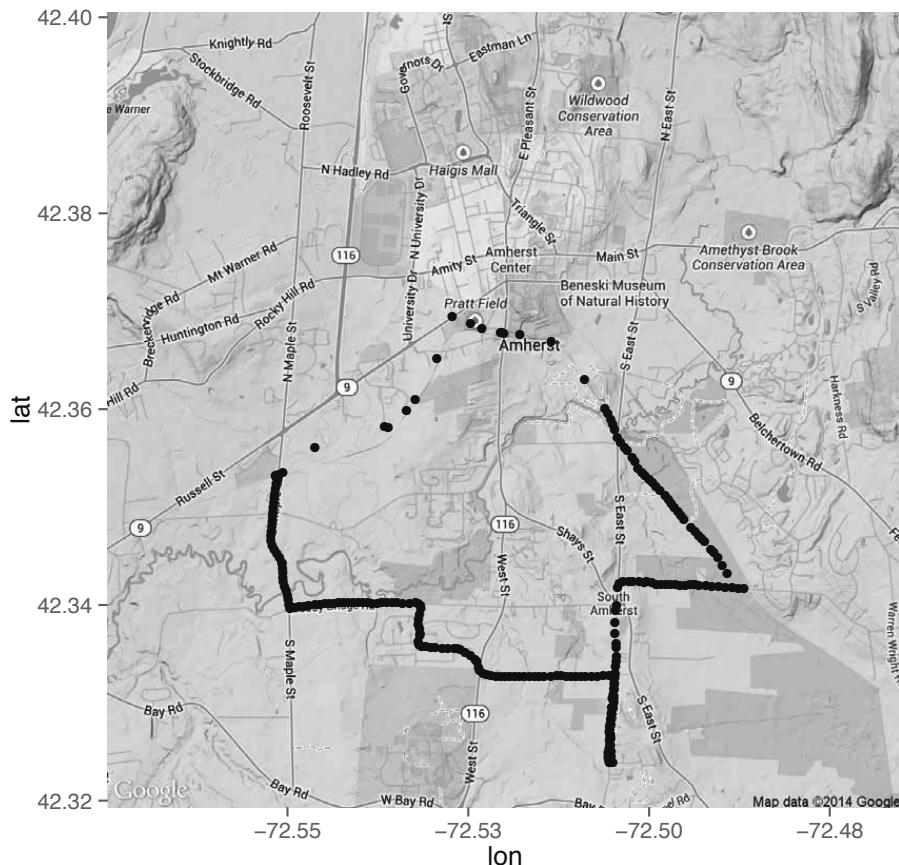


Figure 12.3: Bike plot with map background

As far as we know, there is no simple way to do this with SAS. A close approximation can be made, however, by plotting the points onto a background image of the desired map. In this case, we used the Google map generated by R as in the previous example, using `ggmap()` without the `geom_point()` function. That file is saved as `c:/book/mapback.jpeg` and inserted as the background using the `iframe` option below. Considerable trial and error was required to correctly align the points with the background.

```

filename bike
  'c:/book/cycle.csv';

proc import datafile=bike out=ride dbms=dlm;
  delimiter=',';
  getnames=yes;
run;

symbol1 c=black v=dot h=.5 i=none;
axis1 color=white order = (-72.58,-72.461) offset=(0)
  value= (c=black) major = (c= black);
axis2 color=white order = (42.315, 42.405) offset=(0)
  value= (c=black) major = (c= black);
proc gplot data=ride;
  plot latitude * longitude / haxis=axis1 vaxis=axis2
    imagestyle=fit iframe = "c:/book/mapback.jpeg";
;
run;

```

The results are quite similar to the R results in Figure 12.3. For more complex multi-dimensional graphics made with the same data, see <http://tinyurl.com/sasrblog-bikeride> and <http://tinyurl.com/sasrblog-bikeride-redux>.

### 12.3.3 Choropleth maps

Choropleth maps (8.5.1) are helpful for visualizing geographic data. In this example, we use data from the built-in R dataset, `USArrests`, which includes United States arrests in 1973 per 100,000 inhabitants in various categories by state.

To use the data in SAS, we'll save it to an external file in R. We'll use the Stata format for the external file (see 1.2.6).

```

> library(foreign)
> USArrests.st =
  transform(USArrests, region=tolower(rownames(USArrests)))
> write.dta(USArrests.st, "c:\\book\\USArrests.dta",
  convert.factors="string")

```

Note that the state names are provided only as the row names in the R dataframe. It's easier to get them out of R if they are instead stored as a variable. The `transform()` function above adds them to the dataset (2.2). The default is to convert string variables to numbers with Stata labels containing the strings. These are correctly imported to SAS as value labels, but value labels are not useful for our purposes. The `convert.factors="string"` option is used to retain the state names directly instead. Then we can read it into SAS (see 1.1.9).

```
proc import datafile="C:\book\usarrests.dta"
  out=usarrests dbms=dta replace;
run;
```

```
proc print data=usarrests (obs=5); run;
```

| Obs | Urban  |         |     |      |            |
|-----|--------|---------|-----|------|------------|
|     | Murder | Assault | Pop | Rape | region     |
| 1   | 13.2   | 236     | 58  | 21.2 | alabama    |
| 2   | 10.0   | 263     | 48  | 44.5 | alaska     |
| 3   | 8.1    | 294     | 80  | 31.0 | arizona    |
| 4   | 8.8    | 190     | 50  | 19.5 | arkansas   |
| 5   | 9.0    | 276     | 91  | 40.6 | california |

To make the map, we'll use a built-in US map provided with SAS. This comes with the two-letter US postal codes identifying the states. To match with the lower-case state names in the input dataset, we'll use the `stnamel` function to convert the postal codes to long names and the `lcase` function (2.2.17) to match the values coming from R.

Then the `gmap` procedure makes the choropleth.

```
pattern1 v=s c=grayff;
pattern2 v=s c=grayda;
pattern3 v=s c=grayaa;
pattern4 v=s c=gray68;
pattern5 v=s c=gray22;
data mymap;
set maps.us;
  region = lcase(stnamel(statecode));
run;

proc gmap data=usarrests map=mymap;
  id region;
  choro murder / levels=5;
run; quit;
```

Note that the map dataset and the plot dataset remain separate, but are linked by a commonly named variable specified in the `id` statement. The `pattern` statements change the colors from the default blue shades to print-friendly grays. SAS maps are stored with variables `x` and `y` describing the boundary points. If longitude and latitude values are available, a variety of projections can be applied using the `gproject` procedure.

The results are displayed in Figure 12.4.

In R, we'll use the `ggmap` package. Its functions build on the `ggplot2` package, which implements ideas related to the “grammar of graphics” [196]. The package uses a syntax where specific elements of the plot are added to the final product using special functions connected by the `+` symbol. Some additional work is needed to merge the dataset with the state information (2.3.11) and to sort the resulting dataframe (2.3.10) so that the shape data for the states is plotted in order.

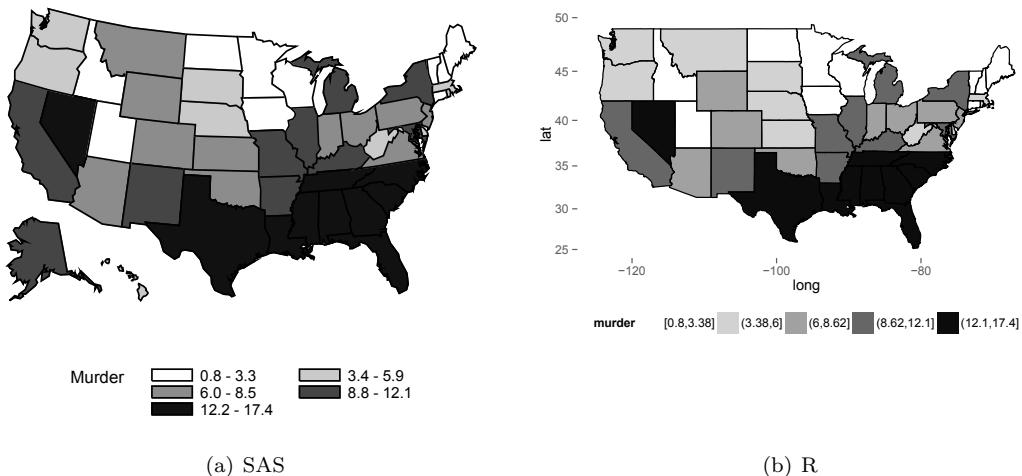


Figure 12.4: Choropleth map

```
> library(ggmap)
> USArests.st = transform(USArests.st, murder = cut_number(Murder, 5))
> us_state_map = map_data('state')
> map_data = merge(USArests.st, us_state_map, by="region")
> map_data = map_data[order(map_data$order),]
> p0 = ggplot(map_data, aes(x=long, y=lat, group=group)) +
  geom_polygon(aes(fill = murder)) +
  geom_path(colour='black') +
  theme(legend.position = "bottom",
        panel.background=element_rect(fill="transparent",
                                       color=NA)) +
  scale_fill_grey(start=1, end = .1) + coord_map()
> plot(p0)
```

The `scale_fill_grey()` function changes the colors from the default unordered multiple colors to an ordered and print-friendly black and white (see also `scale_file_brewer`). The `ggmap` package uses the Mercator projection (see `coord_map()` in the `ggplot2` package and `mapproject` in the `mapproject` package).

Note that the binning algorithms used by the SAS `gmap` procedure and the `cut_number()` function differ slightly, so that some states are shifted. As always, the choice of groupings can have an impact on the message conveyed by the graphical display.

## 12.4 Data scraping and visualization

In the next sections, we automate data harvesting from the web, by “scraping” a URL, then reading a datafile with two lines per observation, and plotting the results as time series data. The data being harvested and displayed are the sales ranks from Amazon for the “Cartoon Guide to Statistics”[54].

### 12.4.1 Scraping data from HTML files

We can find the Amazon Sales Rank for a book by downloading the html code for a desired web page and searching for the appropriate line. The code to do this relies heavily on 1.1.4 (reading more complex data files) as well as 2.2.14 (replacing strings).

An example can be found at <http://www.amherst.edu/~nhorton/sasr2/datasets/cartoon.html>. Many thousands of lines into the file, we find the line we're looking for.

```
#8,048 in Books (<a href="http://www.amazon.com/best-sellers-books-
Amazon/zgbs/books/ref=pd_dp_ts_b_1">See Top 100 in Books</a>)
```

(We've inserted a line break to allow for printing). If you want to see this, find out how to **View Source** in your browser. In Mozilla Firefox, this is in the **Web Developer** tab.

Unfortunately, the line number where it appears changes periodically. Thus, to find the line, we need to read every line of the file and parse it until we find the correct line. Our approach will be to first look for the line with the expression **<b>See Top 100 in Books</b>**. In SAS, once we've found the line, we can look for the # symbol, and the numbers between there and the text **in Books**.

```
filename amazon url "http://tinyurl.com/cartoonguide";
```

```
data grabit;
infile amazon truncover;
input @1 line $256.;

if count(line, "See Top 100 in Books") gt 0 then do;
  rankchar = substr(line, find(line, "#") + 1,
    find(line, "in Books") - find(line, "#") - 2);
  rank = input(rankchar, comma9.);
  output;
end;
run;
```

The code works by reading in the first 256 characters of each line, as a single variable. Then we use the **count** function to determine if this is the correct line. Next, we use the **substr** function to make a new variable with the characters after the # symbol and before the “in Books”. Finally, we read the number in as a character string, formatting it afterwards to accommodate the commas.

Our approach in R is similar, except that we'll use some different functions to isolate the number, as annotated within the code below. To help in comprehending the code, readers are encouraged to run the commands on a line-by-line basis, then look at the resulting value.

```
> # grab contents of web page
> urlcontents = readLines("http://tinyurl.com/cartoonguide")
> # find line with sales rank
> linenum = suppressWarnings(grep("See Top 100 in Books", urlcontents))
> # split line into multiple elements
> linevals = strsplit(urlcontents[linenum], ' ')[[1]]
> # find element with sales rank number
> entry = grep("#", linevals)
> charrank = linevals[entry] # snag that entry
> charrank = substr(charrank, 2, nchar(charrank)) # kill '#' at start
> charrank = gsub(',', '', charrank) # remove commas
> salesrank = as.numeric(charrank) # make it numeric
> cat("salesrank=", salesrank, "\n")
```

In our experience, the format of Amazon’s book pages changes often. The code above may not work on current pages, but could be tested on the example page mentioned above, at <http://www.amherst.edu/~nhorton/sasr2/datasets/cartoon.html>. More sophisticated approaches to web scraping can be found in Nolan and Temple Lang [127].

### 12.4.2 Reading data with two lines per observation

The code from 12.4.1 was run regularly on a server by calling R in batch mode (see B.2.2), with results stored in a cumulative file. While a date-stamp was added, it was included in the file on a different line. The file (accessible at <https://www.amherst.edu/~nhorton/sasr2/datasets/cartoon.txt>) has the following form.

```
Wed Oct  9 16:00:04 EDT 2013
salesrank= 3269
Wed Oct  9 16:15:02 EDT 2013
salesrank= 4007
```

To read these data into SAS, we’ll check the first character of each line: if it is an “s”, we have a line with a rank, not a date in it. We’ll read the two types of lines with different input statements.

```
filename salesdat url
  "http://www.amherst.edu/~nhorton/sasr2/datasets/cartoon.txt";

data sales;
infile salesdat;
retain dow month date time edt year;
input @1 type $1 @;
if type ne 's' then do;
  input @1 dow $ Month $ date time $ edt $ year;
end;
else do;
  input @12 rank;
  datetime = compress(date//month//year//"/"//time);
  salestime = input(datetime,datetime18.);
  if timepart(salestime) lt (8 * 60 * 60) or
    timepart(salestime) gt (18 * 60 * 60) then night=1;
  else night = 0;
  output;
end;
run;
```

To implement our plan, we use the advanced features of the `input` statement, as in 12.2. Later, we’ll use the time of day in our plot. For clarity, we first construct a character variable with the time, then convert it to a SAS date-time variable (see 2.4) using the `input` function. We define “night” as times between 6:00 PM and 8:00 AM, and calculate it by extracting just the time from the SAS date-time formatted variable we construct from the information in the file.

We can then examine the first few lines of the file. First, let’s see how SAS handles the date-time variable.

```
proc print data=sales (obs=4);
  var datetime salestime rank;
run;

Obs      datetime          salestime   rank
 1  30Sep2013/00:00:03  1696118403  5151
 2  30Sep2013/00:15:03  1696119303  5151
 3  30Sep2013/00:30:03  1696120203  4162
 4  30Sep2013/00:45:03  1696121103  4162
```

Those `salestime` values look like the counts of seconds they are supposed to be. But let's also see whether they converted correctly.

```
proc print data=sales (obs=4);
  var datetime salestime rank;
  format salestime datetime18. ;
run;
```

| Obs | datetime           | salestime        | rank |
|-----|--------------------|------------------|------|
| 1   | 30Sep2013/00:00:03 | 30SEP13:00:00:03 | 5151 |
| 2   | 30Sep2013/00:15:03 | 30SEP13:00:15:03 | 5151 |
| 3   | 30Sep2013/00:30:03 | 30SEP13:00:30:03 | 4162 |
| 4   | 30Sep2013/00:45:03 | 30SEP13:00:45:03 | 4162 |

The character and date-time versions match.

In R, we begin by reading the file, then we calculate the number of entries by dividing the file's length by two. Next, two empty vectors of the correct length and type are created to store the data. Once this preparatory work is completed, we loop (4.1.1) through the file, reading in the odd-numbered lines as date/time values from the Eastern U.S. time zone, with daylight savings applied. The `gsub()` function (2.2.14) replaces matches determined by regular expression matching. In this situation, it is used to remove the time zone from the line before this processing. These date/time values are read into the `timeval` vector. Even-numbered lines are read into the `rank` vector, after removing the strings `salesrank=` and `NA` (again using two calls to `gsub()`). Finally, we make a dataframe (B.4.6) from the two vectors and display the first few lines using the `head()` function (1.2.1).

```
> library(RCurl)
> myurl =
  getURL("https://www3.amherst.edu/~nhorton/sasr2/datasets/cartoon.txt",
         ssl.verifypeer=FALSE)
> file = readLines(textConnection(myurl))
> n = length(file)/2
> rank = numeric(n)
> timeval = as.POSIXlt(rank, origin="1960-01-01")
> for (i in 1:n) {
  timeval[i] = as.POSIXlt(gsub('EST', '',
                               gsub('EDT', '', file[(i-1)*2+1])),
                           tz="EST5EDT", format="%a %b %d %H:%M:%S %Y")
  rank[i] = as.numeric(gsub('NA', '',
                            gsub('salesrank= ', '', file[i*2])))
}
> timerank = data.frame(timeval, rank)
```

Note that the file is being read from an HTTPS (Hypertext Transfer Protocol Secure) connection (1.1.12) and string data is converted to date and time variables (2.4.6). The first four entries of the file are given below.

```
> head(timerank, 4)

      timeval rank
1 2013-09-30 00:00:03 5151
2 2013-09-30 00:15:03 5151
3 2013-09-30 00:30:03 4162
4 2013-09-30 00:45:03 4162
```

### 12.4.3 Plotting time series data

While it is straightforward to make a simple plot of the data from 12.4.2 using code discussed in 8.3.1, we'll augment the display by indicating whether the rank was recorded in the nighttime (eastern U.S. time) or not. Then we'll color the nighttime ranks differently from the daytime ranks.

In SAS, we already made the nighttime indicator. We'll use the `sgplot` procedure (8.3.1) to make the plot.

```
proc sgplot data=sales;
  series y=rank x=salestime / lineattrs=(thickness=2 color=black);
  scatter y=rank x=salestime / group=night grouporder=ascending
    markerattrs=(symbol=circlefilled) name="night";
  refline "30SEP13/23:59:59"dt / axis=x lineattrs=(thickness=2
    color=black pattern=shortdash);
  keylegend "night" / title="Night" location=inside position=top;
  xaxis label=" ";
  format salestime datetime8. ;
run;
```

The plot is composed of several pieces. The `series` statement makes a time series plot, or a line plot, connecting the values in sequence. The `scatter` statement adds symbols for the observed ranks, plotting day and night values in different colors using the `group` option. The `refline` statement adds a vertical line at the end of September. Note the use of the trailing `dt` to convert the text to the date-time format that the x axis values are recorded in. The `keylegend` statement improves the default legend by bringing it into the plot area. The remaining options are fairly self-explanatory.

For R, we begin by creating a new variable reflecting the date-time at the midnight before we started collecting data. We then coerce the time values to numeric values using the `as.numeric()` function (2.2.7) while subtracting that midnight value. Next, we call the `hour()` function in the `lubridate` package (2.4) to get the hour of measurement.

```
> library(lubridate)
> timeofday = hour(timeval)
> night = rep(0,length(timeofday)) # vector of zeroes
> night[timeofday < 8 | timeofday > 18] = 1
```

Then we can build the plot.

```
> plot(timeval, rank, type="l", xlab="", ylab="Amazon Sales Rank")
> points(timeval[night==1], rank[night==1], pch=20, col="black")
> points(timeval[night==0], rank[night==0], pch=20, col="red")
> legend(as.POSIXlt("2013-10-03 00:00:00 EDT"), 6000,
  legend=c("day", "night"), col=c("red", "black"), pch=c(20,20))
> abline(v=as.numeric(as.POSIXlt("2013-10-01 00:00:00 EST")), lty=2)
```

The time series plot is requested by the `type="l"` option and symbols for the ranks added with calls to the `points()` function. The `abline()` function adds a reference line at the start of October. The results for both SAS and R are displayed in Figure 12.5.

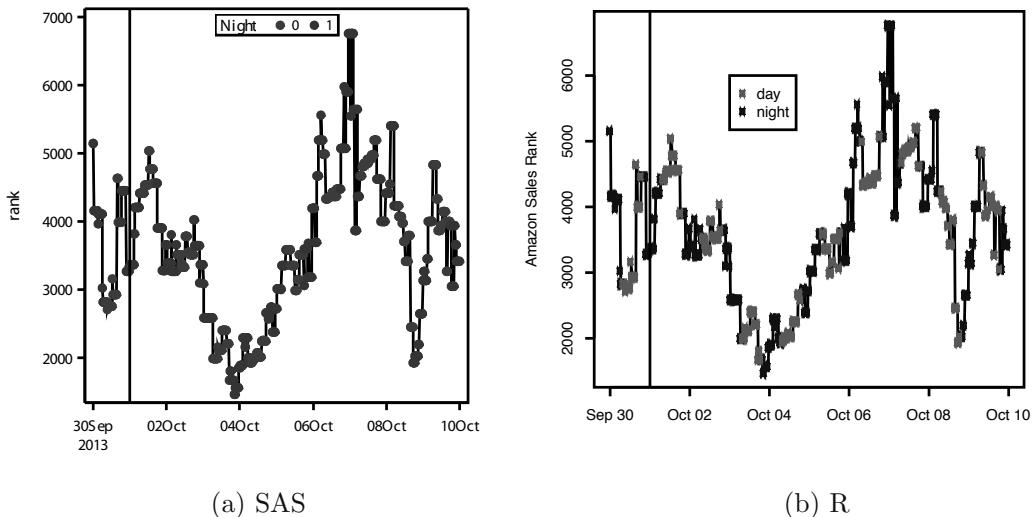


Figure 12.5: Sales plot

#### 12.4.4 URL APIs and truly random numbers

Usually, we're content to use a pseudo-random number generator. But sometimes we may want numbers that are actually random. An example might be for randomizing treatment status in a randomized controlled trial. The site [Random.org](http://Random.org) provides truly random numbers based on radio static. For long simulations which need a huge number of random numbers, the quota system at [Random.org](http://Random.org) may preclude its use. But for small to moderate needs, it can be used to provide truly random numbers. In addition, you can purchase larger quotas if need be.

The site provides application programming interfaces (APIs) for several types of information. We'll demonstrate how to use these to pull vectors of uniform (0,1) random numbers (of  $10^{-9}$  precision) and to check the quota. To generate random variates from other distributions, you can use the inverse probability integral transform (3.1.10).

In SAS, we'll make a macro to grab the desired number of random values and save them in a dataset. The challenging bit is to pass the desired number of random numbers off to the API, through the macro system. This is hard because the API includes the special characters ?, ", and, most notably, &. The ampersand is used by the macro system to denote the start of a macro variable and is used in APIs to indicate that an additional parameter follows.

To avoid processing these characters as part of the macro syntax, we have to enclose them within the macro quoting function `%nrstr()`. We use this approach twice, for the fixed pieces of the API, and between them insert the macro variable that contains the number of random numbers desired. Also note that the sequence "%" is used to produce the quotation mark. Then, to unmask the resulting character string and use it as intended, we `%unquote()` it. Note that the line breaks printed below in the filename statement must be removed for the code to work.

```
%macro rands (outds=ds, nrands=);
filename randsite url %unquote(
  %nrstr(%"http://www.random.org/integers/?num=
  &nrands
  %nrstr(&min=0&max=1000000000&col=1&base=10&
  format=plain&rnd=new%"));
proc import datafile=randsite out=&outds dbms=dlm replace;
getnames=no;
run;

data &outds;
set &outds;
  var1 = var1 / 1000000000;
run;
%mend rands;
```

Running the macro and examining the output is trivial.

```
%rands(nrands=7, outds=myrs);

proc print data=myrs; run;
```

| Obs | VAR1        |
|-----|-------------|
| 1   | 0.502746213 |
| 2   | 0.247134785 |
| 3   | 0.620425172 |
| 4   | 0.932627004 |
| 5   | 0.266144436 |
| 6   | 0.967032193 |
| 7   | 0.751058844 |

The companion macro to find the quota is slightly simpler, since we don't need to insert the number of random numbers in the middle of the URL. Here, we show the quota in the SAS log; the `file print` syntax, shown in <http://tinyurl.com/sasrblog-robust>, can be used to send it to the output instead.

```
%macro quotacheck;
filename randsite url
%unquote(%nrstr(%
  "http://www.random.org/quota/?format=plain%"));
proc import datafile=randsite out = __qc dbms = dlm replace;
getnames = no;
run;

data _null_;
set __qc;
put "Remaining quota is " var1 "bytes";
run;
%mend quotacheck;
```

```
Remaining quota is 996040 bytes
```

Two R functions are shown below. While the problem isn't as difficult as in SAS, it is necessary to enclose the character string for the URL in the `as.character()` function (1.1.4).

```

> truerand = function(numrand) {
  read.table(as.character(paste("http://www.random.org/integers/?num=",
    numrand, "&min=0&max=1000000000&col=1&base=10&format=plain&rnd=new",
    sep="")))/1000000000
}
> quotacheck = function() {
  line = as.numeric(readLines(
    "http://www.random.org/quota/?format=plain"))
  return(line)
}
> truerand(7)

      V1
1 0.780
2 0.804
3 0.502
4 0.377
5 0.537
6 0.580
7 0.135

> quotacheck()

[1] 1e+06

```

## 12.5 Manipulating bigger datasets

In this example, we consider analysis of the Data Expo 2009 commercial airline flight dataset [197], which includes details of  $n = 123,534,969$  flights from 1987 to 2008. We consider the number of flights originating from Bradley airport (code BDL, serving Hartford, CT and Springfield, MA). Because of the size of the data, we will demonstrate use of a database system accessed using a structured query language (SQL) [172].

Full details are available on the Data Expo website (<http://stat-computing.org/dataexpo/2009/sqlite.html>) regarding how to download the Expo data as comma separated files (1.6 gigabytes of compressed, 12 gigabytes uncompressed through 2008), set up and index a database (19 gigabytes), then access it from within R.

In SAS, analysis can be undertaken on an internal or external database server running MySQL. The following code extracts the sum total of flights from Bradley International Airport, by day, month, and year. Here the SELECT statement specifies the five variables to be included (one of which is the count of flights), the name of the table, what values to include (only BDL), and what level to aggregate (unique day).

```

proc sql;
  connect to mysql (user="tlehrer" server="hen3ry.mgh.edu"
    password="FakePW1" dbname="airlines");
  execute (create table ds as
    SELECT DayofMonth, Month, Year, Origin,
      sum(1) as numFlights FROM ontime WHERE Origin=BDL
      GROUP BY DayofMonth,Month,Year)
  by mysql;
  disconnect from mysql;
quit;

```

A similar system for allowing access to databases is SQLite, a self-contained, serverless, transactional SQL database engine. To use this with R, the analyst installs the `sqlite` software library (<http://sqlite.org>). Next the input files must be downloaded to the local machine, a database set up (by running `sqlite3 ontione.sqlite3`) at the shell command line), creating a table with the appropriate fields, loading the files using a series of `.import` statements, then speeding up access by adding indexing. Then the `RSQLite` package can be used to create a connection to the database.

```
> library(RSQLite)
> con = dbConnect("SQLite", dbname = "/Home/Airlines/ontime.sqlite3")
> ds = dbGetQuery(con, "SELECT DayofMonth, Month, Year, Origin,
+     sum(1) as numFlights FROM ontione WHERE Origin='BDL'
+     GROUP BY DayofMonth, Month, Year")
> # returns a data frame with 7,763 rows and 5 columns
> ds = transform(ds, date =
+     as.Date(paste(Year, "-", Month, "-", DayofMonth, sep="")))
> ds = transform(ds, weekday = weekdays(date))
> ds = ds[order(ds$date),]
> mondays = subset(ds, weekday=="Monday")
> library(lattice)
> xyplot(numFlights ~ date, xlab="", ylab="number of flights on Monday",
+     type="l", col="black", lwd=2, data=mondays)
```

As in the SAS example, the `SELECT` statement specifies the five variables to be included (one of which is the count of flights), the name of the table `ontime`, what flights to include (only those originating at BDL), and what level to aggregate (unique day). The results are plotted in Figure 12.6. Similar functionality is provided for MySQL databases using the `RMySQL` package.

## 12.6 Constrained optimization: the knapsack problem

The website [http://rosettacode.org/wiki/Knapsack\\_Problem](http://rosettacode.org/wiki/Knapsack_Problem) describes a fanciful trip by a traveler to Shangri La. Upon leaving, the traveler is allowed to take as much of three valuable items as they like, as long as they fit in a knapsack. A maximum of 25 weights can be taken, with a total volume of 25 cubic units. The weights, volumes, and values of the three items are given in Table 12.1.

How can the traveler maximize the value of the items? It is straightforward to calculate the solutions using brute force, by iterating over all possible combinations and eliminating those that are over weight or too large to fit.

In SAS, this task is undertaken as a data step.

Table 12.1: Weights, volume, and values for the knapsack problem

| Item    | Weight | Volume | Value |
|---------|--------|--------|-------|
| Panacea | 0.3    | 2.5    | 3000  |
| Ichor   | 0.2    | 1.5    | 1800  |
| Gold    | 2.0    | 0.2    | 2500  |

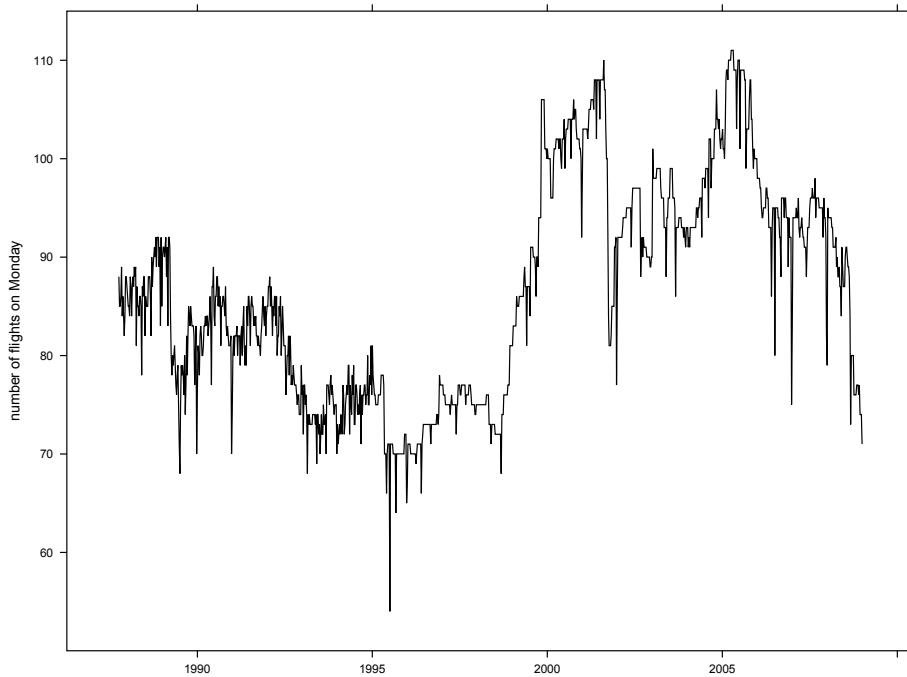


Figure 12.6: Number of flights departing Bradley airport on Mondays over time

```

data one;
  wtpanacea=0.3;      wtichor=0.2;      wtgold=2.0;
  volpanacea=0.025;   volichor=0.015;   volgold=0.002;
  valpanacea=3000;    valichor=1800;    valgold=2500;
  maxwt=25;           maxvol=0.25;

/* find upper bound for looping */
maxpanacea = floor(min(maxwt/wtpanacea, maxvol/volpanacea));
maxichor = floor(min(maxwt/wtichor, maxvol/volichor));
maxgold = floor(min(maxwt/wtgold, maxvol/volgold));

/* loop */
do panacea = 0 to maxpanacea;
  do ichor = 0 to maxichor;
    do gold = 0 to maxgold;
      output;
    end;
  end;
end;
run;

```

The resulting dataset includes improper values: combinations with too much weight or volume. We prune them out in a separate data step where we also calculate the total weight and volume implied. We discard values over the limit using the subsetting `if` statement

(2.3.1). Note that these statements could have been included in the innermost do loop above. We put them in a separate data step for clarity of presentation.

```
data two; set one;
  totalweight = wtpanacea*panacea + wtichor*ichor + wtgold*gold;
  totalvolume = volpanacea*panacea + volichor*ichor + volgold*gold;
  if (totalweight le maxwt) and (totalvolume le maxvol);
  vals = valpanacea*panacea + valichor*ichor + valgold*gold;
run;
```

To find the maximum value that fits in the knapsack, we can sort the dataset and print the results. Here we show the top five values.

```
proc sort data=two;
  by descending vals;
run;

proc print data=two (obs=5) noobs;
  var panacea ichor gold vals totalweight;
run;
```

| panacea | ichor | gold | vals  | totalweight |
|---------|-------|------|-------|-------------|
| 0       | 15    | 11   | 54500 | 25.0        |
| 3       | 10    | 11   | 54500 | 24.9        |
| 6       | 5     | 11   | 54500 | 24.8        |
| 9       | 0     | 11   | 54500 | 24.7        |
| 1       | 13    | 11   | 53900 | 24.9        |

We can maximize the value at the minimum weight with 9 panacea and 11 gold.

In R, we define a number of support functions, then run over all possible values of the knapsack contents (after `expand.grid()` generates the list). The `findvalue()` function checks the constraints and sets the value to 0 if they are not satisfied, and otherwise calculates them for the set. The `apply()` function (see 2.6.4) is used to run a function for each item of a vector.

```
> # Define constants and useful functions
> weight = c(0.3, 0.2, 2.0)
> volume = c(2.5, 1.5, 0.2)
> value = c(3000, 1800, 2500)
> maxwt = 25
> maxvol = 25
```

```

> # minimize the grid points we need to calculate
> max.items = floor(pmin(maxwt/weight, maxvol/volume))
> # useful functions
> getvalue = function(n) sum(n*value)
> getweight = function(n) sum(n*weight)
> getvolume = function(n) sum(n*volume)
> # main function: return 0 if constraints not met,
> # otherwise return the value of the contents, and their weight
> findvalue = function(x) {
  thisweight = apply(x, 1, getweight)
  thisvolume = apply(x, 1, getvolume)
  fits = (thisweight <= maxwt) &
    (thisvolume <= maxvol)
  vals = apply(x, 1, getvalue)
  return(data.frame(panacea=x[,1], ichor=x[,2], gold=x[,3],
    value=fits*vals, weight=thisweight,
    volume=thisvolume))
}
> # Find and evaluate all possible combinations
> combs = expand.grid(lapply(max.items, function(n) seq.int(0, n)))
> values = findvalue(combs)

```

Now we can display the solutions.

```

> max(values$value)

[1] 54500

> values[values$value==max(values$value),]

  panacea ichor gold value weight volume
2067      9     0   11 54500   24.7   24.7
2119      6     5   11 54500   24.8   24.7
2171      3    10   11 54500   24.9   24.7
2223      0    15   11 54500   25.0   24.7

```

The first solution (with 9 panacea), no ichor, and 11 gold) satisfies the volume constraint, maximizes the value, and also minimizes the weight. More sophisticated approaches are available using the `lpSolve` package for linear/integer problems.

# Appendix A

## Introduction to SAS

The SAS™ system is a programming and data analysis package developed and marketed by SAS Institute, Cary NC (SAS). SAS markets many products which are modular parts of an integrated environment. In this book we address software available in the Base SAS, SAS/STAT, SAS/GRAFH, SAS/ETS, and SAS/IML products. Base SAS provides a wide range of data management and analysis tools, while SAS/STAT and SAS/GRAFH provide support for more sophisticated statistics and graphics, respectively. We touch briefly on the IML (interactive matrix language) module, which provides extensive matrix functions and manipulation, and the SAS/ETS module, which supports time series tools and other specialized procedures. All of these products are typically included in educational institution installations, for which SAS offers discounts.

Another option is SAS “OnDemand” (<http://support.sas.com/learn/ondemand>), a cloud-based system. In general, pricing information can be obtained only by contacting SAS directly, as noted at <http://www.sas.com/nextsteps>. SAS does market a system limited to Base SAS, SAS/STAT, and SAS/GRAFH, the current licensing fee for which is \$8,700.

### A.1 Installation

Once licensed, a set of installation disks or a USB key is mailed; this package includes detailed installation instructions tailored to the operating system for which the license was obtained. Also necessary is a special “setinit” file sent from SAS which functions as a password allowing installation of licensed products. An updated setinit file is sent upon purchase of a license renewal.

### A.2 Running SAS and a sample session

Once installed, a recommended step for a new user is to start SAS and run a sample session. Starting SAS in a GUI environment opens a SAS window, as displayed in Figure A.1.

The window is divided into two panes. On the left is a navigation pane with Results and Explorer tabs, while on the right is an interactive windowing environment with Editor, Log, and Output Windows. Effectively, the right-hand pane is like a limited graphical user interface (GUI) in itself. There are multiple windows, any one of which may be maximized, minimized, or closed. Their contents can also be saved to the operating system or printed. Depending on the code submitted, additional windows may open in this area. To open a window, click on its name at the bottom of the right-hand pane; to maximize or minimize

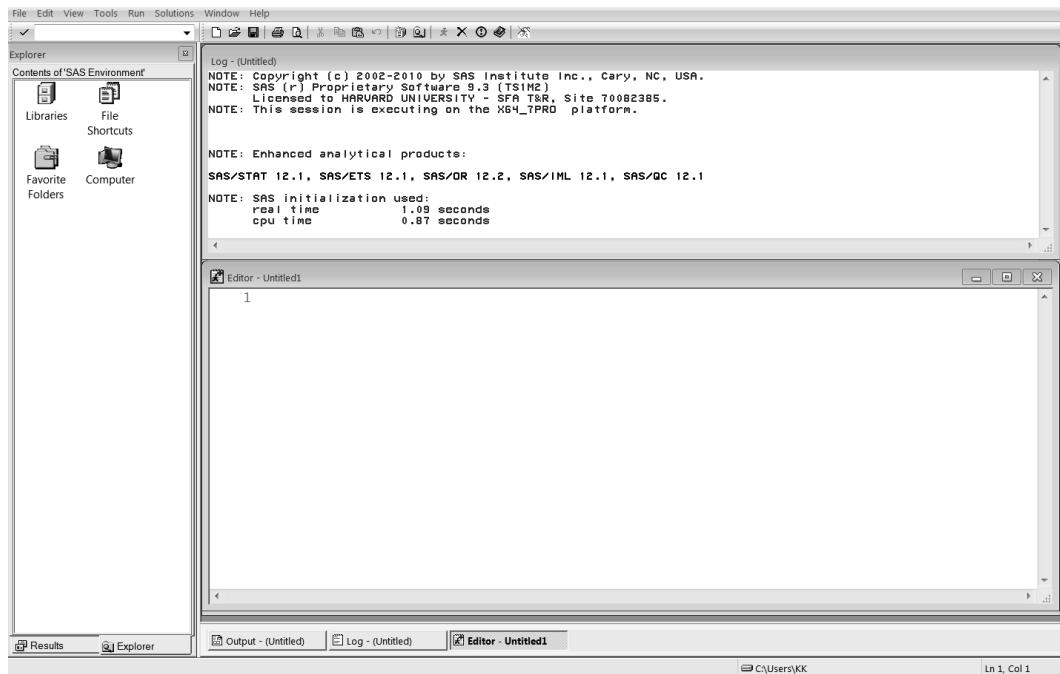


Figure A.1: SAS Windows interface

within the SAS GUI, click on the standard icons your operating system uses for these actions.

On starting SAS, the cursor will appear in the Editor window. Commands such as those in the sample session which follows are typed there. They can also be read into the window from previously saved text files using *File; Open Program* from the menu bar. Typing the code doesn't do anything, even if there are carriage returns in it. To run code, it must be *submitted*; this can be done by clicking the submit button in the GUI as in Figure A.2 or using keyboard shortcuts. After code is submitted, SAS processes the code. Results are not displayed in the Editor window, but in the Results window, and comments from SAS on the commands which were run are displayed in the Log window. If output lines (typically analytic results) are generated, the Output window will jump to the front.

In the left-hand pane, the Explorer tab can be used to display datasets created within the current SAS session or found in the operating system. The datasets are displayed in a spreadsheet-like format. Navigation within the Explorer pane uses idioms familiar to users of GUI-based operating systems. The Results tab allows users to navigate among the output generated during the current SAS session.

As a sample session, consider the following SAS code, which generates 100 normal variates (see 3.1.6) and 100 uniform variates (see 3.1.4), displays the first five of each (see 1.2.1), and calculates series of summary statistics (see 5.1.1). These commands would be typed directly into the Editor window.

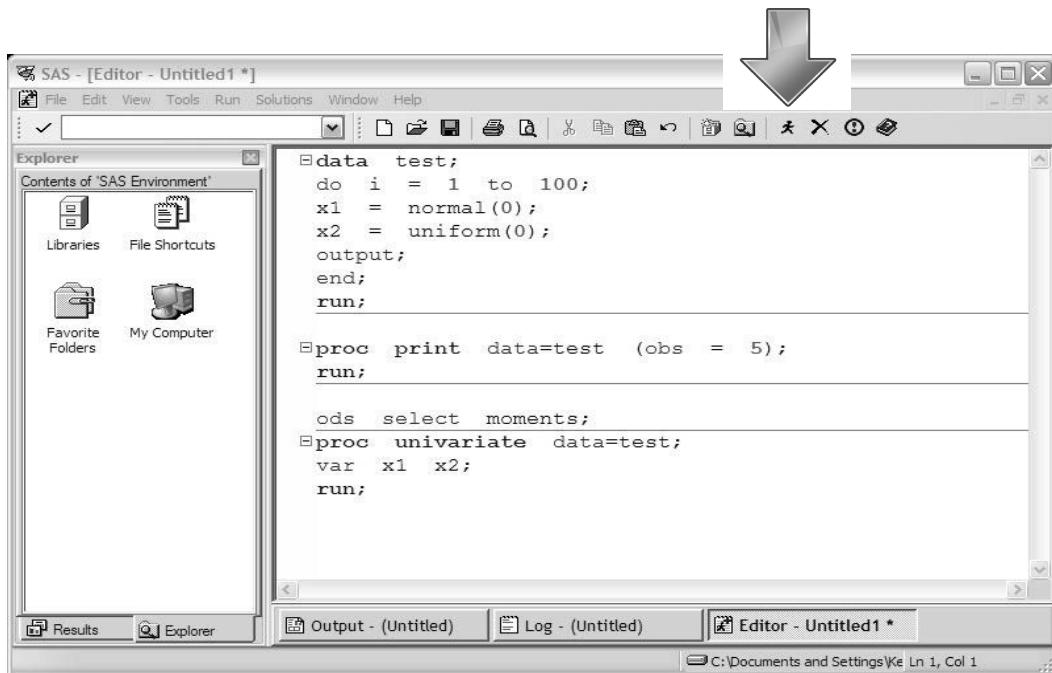


Figure A.2: Running a SAS program

```

/* This is the sample session */
data test;
do i = 1 to 100;
x1 = normal(0);
x2 = uniform(0);
output;
end;
run;

proc print data=test (obs=5);
run;

ods select moments;
proc univariate data=test;
var x1 x2;
run;

```

A user can run a section of code by selecting it using the mouse and clicking the “running figure” (submit) icon near the right end of the toolbar, as shown in Figure A.2. Clicking the submit button when no text is selected will run all of the contents of the window. As demonstrated, text typed between /\* and \*/ are comments, not interpreted by SAS. This code is available for download from the book website: <http://www.amherst.edu/~nhorton/sasr2/examples/sampsess.sas>. Additional code from the text can be found at <http://www.amherst.edu/~nhorton/sasr2/examples>.

We discuss each block of code in the sample session.

```

data test;
do i = 1 to 100;
  x1 = normal(0);
  x2 = uniform(0);
  output;
end;
run;

```

After selecting and submitting the above code there is no output, since none was requested, but the log window will contain some new information:

```

1  data test;
2    do i = 1 to 100;
3      x1 = normal(0);
4      x2 = uniform(0);
5      output;
6    end;
7  run;

```

NOTE: The dataset WORK.TEST has 100 observations and 3 variables.

NOTE: DATA statement used (Total process time):

|           |              |
|-----------|--------------|
| real time | 0.01 seconds |
| cpu time  | 0.01 seconds |

This indicates that the commands ran without incident, creating a dataset called WORK.TEST with 100 rows and 3 columns (one for *i*, one for *x1*, and one for *x2*). The line numbers shown at the left can be used in debugging code.

Next consider the *proc print* code.

```

proc print data=test (obs=5);
run;

```

When these commands are submitted, the Results window will show a table like the one shown in Figure A.3. Note that only five observations are shown because *obs=5* was specified (A.6.1). Omitting this will result in all 100 lines of data printing.

| <b>Obs</b> | <b>i</b> | <b>x1</b> | <b>x2</b> |
|------------|----------|-----------|-----------|
| <b>1</b>   | 1        | -1.18513  | 0.31978   |
| <b>2</b>   | 2        | -0.38488  | 0.08027   |
| <b>3</b>   | 3        | 0.08394   | 0.37353   |
| <b>4</b>   | 4        | 0.05027   | 0.45145   |
| <b>5</b>   | 5        | 0.54218   | 0.55179   |

Figure A.3: Results from *proc print*

| <b>The UNIVARIATE Procedure</b> |            |                         |            |
|---------------------------------|------------|-------------------------|------------|
| <b>Variable: x1</b>             |            |                         |            |
| <b>Moments</b>                  |            |                         |            |
| <b>N</b>                        | 100        | <b>Sum Weights</b>      | 100        |
| <b>Mean</b>                     | -0.1042773 | <b>Sum Observations</b> | -10.42773  |
| <b>Std Deviation</b>            | 1.02828212 | <b>Variance</b>         | 1.05736412 |
| <b>Skewness</b>                 | -0.2217538 | <b>Kurtosis</b>         | -0.5177453 |
| <b>Uncorrected SS</b>           | 105.766423 | <b>Corrected SS</b>     | 104.679047 |
| <b>Coeff Variation</b>          | -986.10356 | <b>Std Error Mean</b>   | 0.10282821 |

Figure A.4: Results from proc univariate

Finally, data are summarized by submitting the lines specifying the `univariate` procedure.

```
ods select moments;
proc univariate data=test;
  var x1 x2;
run;
ods select all;
```

The Results window will display the table shown in Figure A.4 (and a similar table will be created for x2).

As with the `obs=5` specified in the `proc print` statement above, the `ods select moments` statement causes a subset of the default output to be printed. By default, SAS often generates voluminous output that can be hard for new users to digest and would take up many pages of a book. We use the ODS system (A.7) to select pieces of the output throughout the book.

For each of these submissions, additional information is presented in the Log window. While some users may ignore the Log window unless the code did not work as desired, it is always a good practice to examine the log carefully, as it contains warnings about unexpected behavior as well as descriptions of errors which cause the code to execute incorrectly or not at all. In addition, it provides confirmation about expected behavior.

Note that the contents of the Editor, Log, and Result windows can be saved in typical GUI fashion by bringing the window to the front and using *File; Save* through the menus.

In addition to the HTML-formatted Results window, SAS can also print results to the Output window. Output in the Output window is text based, which may be advantageous in some settings. It is also faster. To have SAS send results to both the Output and Results windows, use the GUI menus: Tools; Options; Preferences; Results tab; then click the **Listing** box.

Figure A.5 shows the appearance of the SAS window after running the sample program and clicking the Output window. The Output window can be scrolled through to find results, or the Results tab shown in the left-hand pane can be used to find particular pieces of output more quickly.

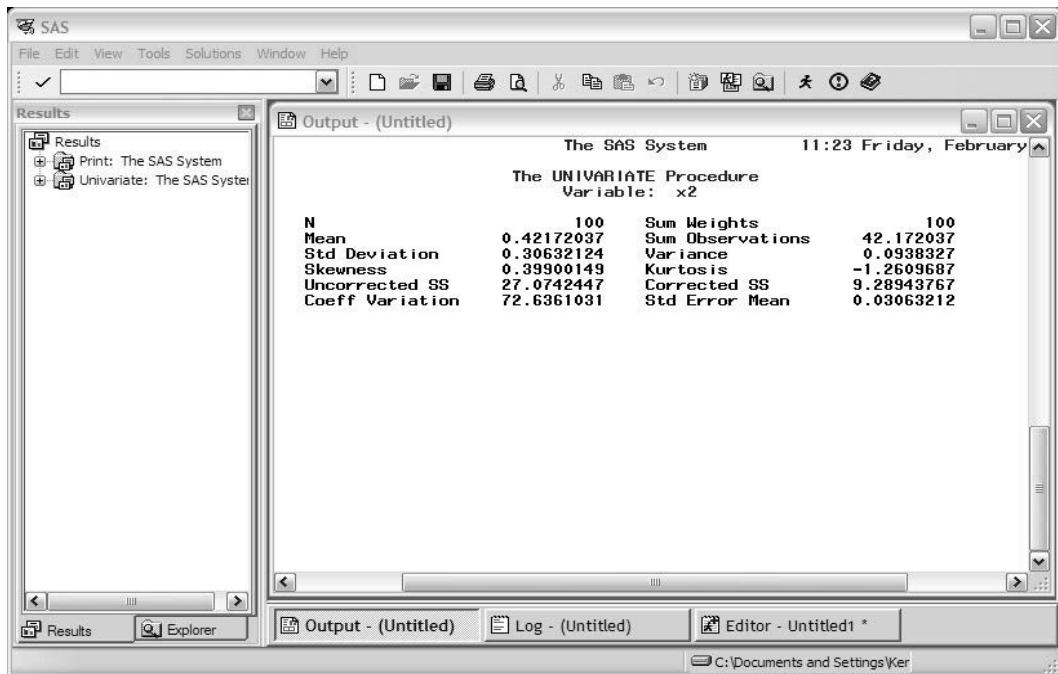


Figure A.5: The SAS window after running the sample session code

Figure A.6 shows the view of the dataset found through the Explorer window by clicking through *Libraries*; *Work*; *Test*. Datasets not assigned to permanent storage in the operating system (see 1.2.3) are kept in a temporary SAS library called the “Work” library.

### A.3 Learning SAS and getting help

This book is not intended as an introduction to SAS. There are, however, numerous tools available for learning SAS. At least three of these are built into the program. Under the Help menu in the Menu bar are “Getting Started with SAS Software” and “Learning SAS Programming.” In the on-line help, under the **Contents** tab is “Learning to Use SAS.” For those interested in learning about SAS but without access to a working version, an on-line option is the excellent UCLA statistics website, which includes the “SAS Starter Kit” (<http://www.ats.ucla.edu/stat/sas/sk/default.htm>). SAS Institute also offers several ways to get help. The central place to start is their web site where the front page for support is currently <http://support.sas.com/techsup>, which has links to discussion forums, support documents, and instructions for submitting an e-mail or phone request for technical support.

Complete documentation is included with SAS installation by default. Clicking the icon of the book with a question mark in the GUI (Figure A.7) will open a new window with a tool for viewing the documentation (Figure A.8). While there are **Contents**, **Index**, **Search**, and **Favorites** tabs in the help tool, we generally use the **Contents** tab as a starting point. Expanding the **SAS Products** folder here will open a list of SAS products (i.e., Base SAS, SAS/STAT, etc.), as well as a link to a list of SAS procedures. Detailed documentation for the desired procedure can be found under the product which provides

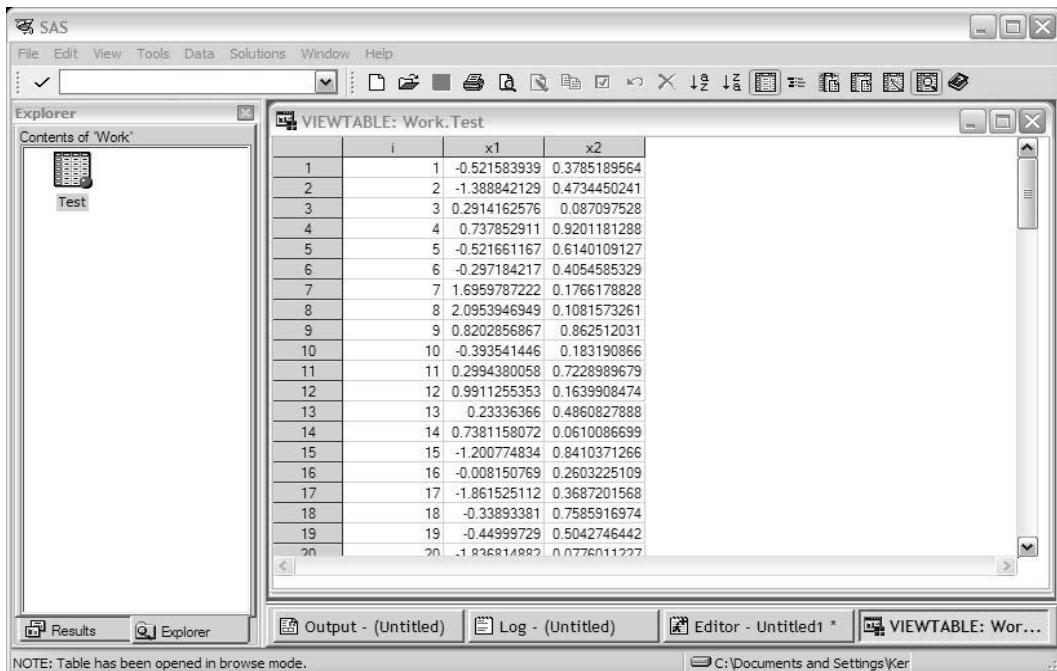


Figure A.6: The SAS Explorer window

access to that `proc` or directly in the list of procedures. In the text, we provide occasional pointers to the on-line help, using the folder structure of the help tool to provide directions to these documents.

## A.4 Fundamental elements of SAS syntax

The SAS syntax can be broken into three main elements: the data step, procedures, and global statements. The data step is used to manage and manipulate data. Procedures are generally ways to do some kind of analysis and get results. Users of SAS refer to procedures as “procs.” Global statements are generally used to set parameters and make optional choices that apply to the output of one or more procedures.

A typical data step might read as follows.

```
data newtest;
set test;
logx = log(x);
run;
```

In this code a new variable named `logx` is created by taking the natural log of the variable `x`. The `data` step works by applying the instructions listed, sequentially, to each line of the input dataset, which in this case is named using the `set` statement, then writing that line of data out to the dataset named in the `data` statement. Data steps and procedures are typically multi-statement collections. Both are terminated with a `run` statement. As shown above, statements in SAS are separated by semicolons, meaning that carriage returns and line breaks are ignored. When SAS reads the `run` statement in the example (when it reaches the “;” after the word `run`), it writes out the processed line of data, then repeats

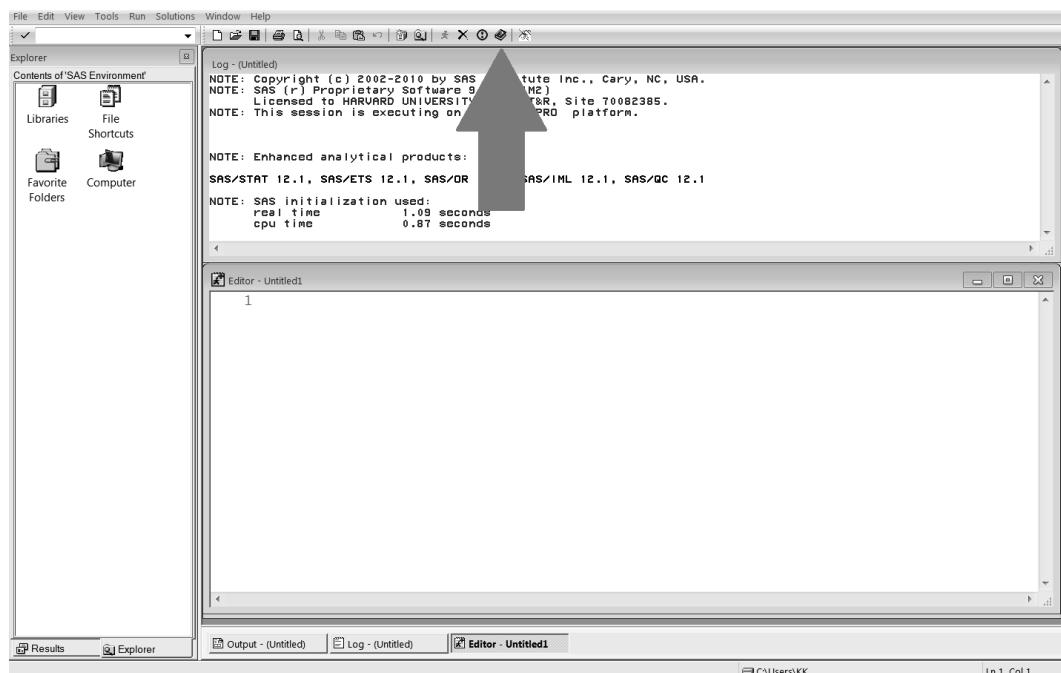


Figure A.7: Opening the on-line help

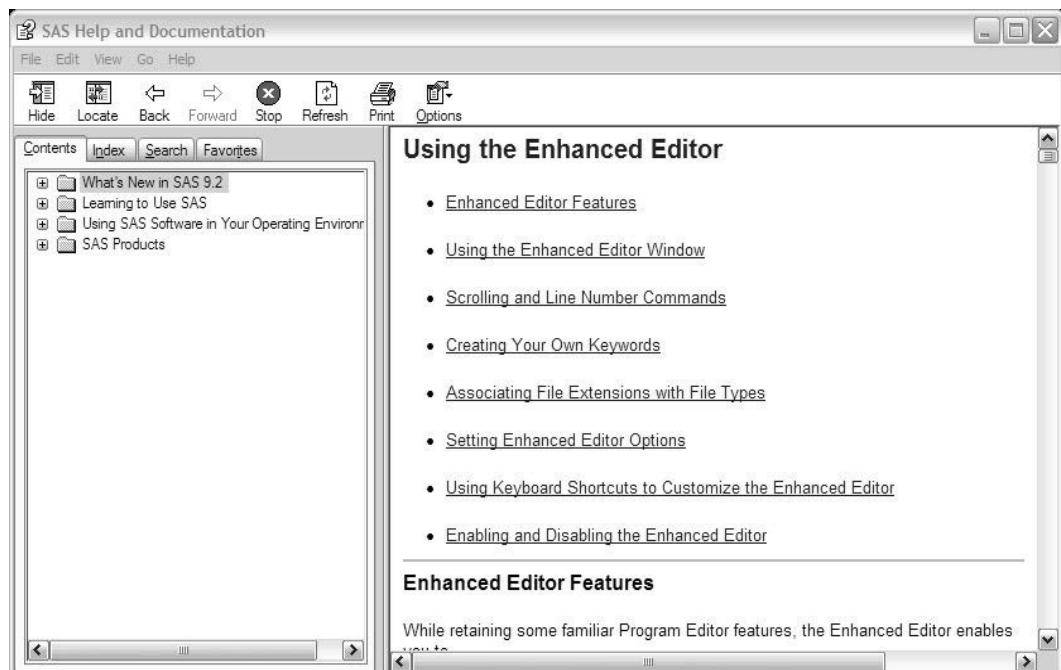


Figure A.8: The SAS Help and Documentation window

the statements for each line of the input data. In this example, a line of data is read from the `test` dataset, the `logx` variable is generated, and the line of data (including `logx`, `x`, and any other data stored in `test`) is written to the new dataset `newtest`.

A typical procedure in SAS might read as follows.

```
proc glm data=newtest;
  model y = logx / solution;
run;
```

Many procedures require multiple statements to function. For example, the `glm` procedure requires both a `proc glm` statement and a `model` statement.

Here, we show the two ways that *options* can be specified in SAS. One way is by simply listing optional syntax after the statement name. In the `proc glm` (6.1.1) statement above, we specify, using the `data` option, that the dataset that should be used is the `newtest` dataset. Without this option SAS defaults to using the most recently created dataset. As a matter of style, we always specify the dataset using the `data` option, which can be used with any and all `procs`. Naming datasets explicitly in each procedure minimizes errors and makes code clearer.

The `model` statement shown demonstrates another way that options are specified, namely, after a forward slash. In general, this syntax is used when the main body of the statement may include separate words. For example, the slash in the `model` statement above separates the model specification (`y = logx`) from the `solution` option that requests the parameter estimates in addition to the default ANOVA table.

We refer to any SAS code appearing between semicolons generically as “statements.” Most statements appear within data steps or procs. Global statements are special statements that need not appear within a data step or a proc. An example would be the following code.

```
options ls=78 ps=60 nocenter;
```

This `options` statement affects the formatting of output pages, limiting the line length to 78 characters per line and 60 lines per page, while removing the default centering.

## A.5 Work process: The cognitive style of SAS

A typical SAS work session involves first writing a `data` step or loading a saved command file (conventionally saved with a `.sas` extension) which might read in or perhaps modify a saved dataset. Then a `proc` is written to perform a desired analysis. The output is examined, and based on the results, the `data` step is modified to generate new variables, the `proc` is edited to choose new options, new `procs` are written, or some subset of these steps is repeated. At the end of the session, the dataset might be saved in the native SAS format, the commands saved in text format, and the results printed onto paper or saved (conventionally with a `.lst` extension). Alternatively, reproducible analysis tools (11.3) might be employed to keep the code, results, and interpretation together.

## A.6 Useful SAS background

### A.6.1 Dataset options

In addition to `data` steps for manipulating data, SAS allows on-the-fly modification of datasets. This approach, while less than ideal for documentation, can be a useful way to reduce code length: rather than create a new dataset with a subset of observations, or with a renamed variable, this can be done simultaneously with specifying the dataset to be used in a procedure. The syntax for these commands, called “dataset options” in SAS documentation, is to list them in parentheses after naming the dataset. So, for example, to

exclude extraneous variables in a dataset from an analysis dataset, the following code could possibly save time if the dataset were large.

```
proc ttest data=test2 (keep=x y);
  class x;
  var y;
run;
```

Another useful dataset option limits the number of observations used from the named dataset.

```
proc ttest data=test2 (obs=60);
  class x;
  var y;
run;
```

A full list of dataset options can be found in the on-line documentation: Contents; SAS Products; Base SAS; SAS Data Set options: Reference; Data Set Options Dictionary.

## A.6.2 Subsetting

It is often convenient to restrict the membership in a dataset or run analyses on a subset of observations. There are three main ways we do this in SAS. One is through the use of a subsetting **if** statement in a data step. The syntax for this is simply

```
data ...;
set ...;
  if condition;
run;
```

where **condition** is a logical statement such as **x eq 2**. (See 4.1.2 for a discussion of logical operators.) This includes only observations for which the condition is true, because when an **if** statement does not include a **then**, the implied **then** clause is interpreted as “then output this line to the dataset; otherwise do not output it.”

A second approach is a **where** statement. This can be used in a **data** step or in a procedure.

```
proc ... data=ds;
  where condition;
...
run;
```

Finally, there is also a **where** dataset option which can be used in a **data** step or in a procedure; the syntax here is slightly different.

```
proc ... data=ds (where=(condition));
...
run;
```

The differences between the **where** statement and the **where** dataset option are subtle and beyond our scope here. However, it is generally computationally cheaper to use a **where** approach than a subsetting **if**.

## A.6.3 Formats and informats

SAS provides special tools for displaying variables or reading them in when they have complicated or unusual constructions in raw text. A good example for this is dates, for which June 27, 2009 might be written as, for example, **6-27-09**, **27-6-09**, **06/27/2009**, and so on. SAS stores dates as the integer number of days since December 31, 1959. To convert one of the aforementioned expressions to the desired storage value, **17710**, one could use an *informat* to describe the way the data is written. For example, if the data were stored

as the above expressions, the informats `mmddyy8.`, `ddmmyy8.`, and `mmddyy10.`, respectively, would read them correctly as 17710. An example of reading in dates is shown in 1.1.2.

In contrast, displaying data in styles other than that in which it is stored is done using the `informat`'s inverse, the `format`. The format for display can be specified within a `proc`. For example, if we plan a time series plot of `x*time` and want the `x` axis labeled in quarters (i.e., 2010Q3), we could use the following code, where the `time` variable is the integer-valued date.

```
proc gplot data=ds;
  plot x*time;
  format time yyq6. ;
run;
```

Another example is deciding how many decimal digits to display. For example, if you want to display two decimal places for variable `p` and three for variable `x`, you could use the following code. This topic is also discussed in 1.2.1.

```
proc print data=ds;
  var p x;
  format p 4.2 x 5.3;
run;
```

More information on informats and formats can be found in the on-line documentation: Contents; SAS Products; Base SAS; SAS Formats and Informats.

## A.7 Accessing and controlling SAS output: the Output Delivery System

SAS does not provide access to most of the internal objects used in calculating results. Instead, it provides specific access to many objects of interest through various procedure statements. The ways to find these objects can be idiosyncratic, and we have tried to highlight the most commonly needed objects in the text. This situation is roughly equivalent to the need in R to know the full name of an object before it can be accessed.

A general way to access and control output within SAS is through the `output delivery system` or (redundantly, as in “ATM machine”) the `ODS` system. This is a very powerful and flexible system for accessing procedure results and controlling printed output. We use the `ODS` system mainly for two tasks: 1) to save procedure output into explicitly named datasets and 2) to suppress some printed output from procedures which generate lengthy output. In addition, we discuss using the `ODS` system to save output in useful file formats such as portable document format (PDF), hypertext markup language (HTML), or the rich text format (RTF) which many word processing programs can read. We note that ODS has other uses beyond the scope of this book and encourage readers to spend time familiarizing themselves with it.

### A.7.1 Saving output as datasets and controlling output

Using ODS to save output or control the printed results involves two steps; first, finding out the name by which the `ODS` system refers to the output, and second, requesting that the dataset be saved as a SAS dataset or including or excluding it as output. The names used by the `ODS` system can be most easily found by running an `ods trace / listing` statement (later reversed using an `ods trace off` statement). The ODS `outputname` thus identified can be saved as a dataset using an `ods output outputname=newname` statement. Specific pieces of output can be excluded using an `ods exclude outputname1 outputname2 ... outputnamek` statement or all but desired pieces excluded using the `ods select`

`outputname1 outputname2 ... outputnamek` statement. These statements are each submitted before the procedure code which generates the output concerned. The `exclude` and `select` statements can be reversed using an `ods exclude none` or `ods select all` statement.

For example, to save the result of the *t* test performed by `proc ttest` (5.4.2), the following code could be used. First, generate some data for the test.

```
data test2;
  do i = 1 to 100;
    if i lt 51 then x=1;
    else x=0;
    y = normal(0) + x;
    output;
  end;
run;
```

Then, run the *t* test, including the `ods trace on / listing` statement to learn the names used by the ODS system.

```
ods trace on / listing;
proc ttest data=test2;
  class x;
  var y;
run;
ods trace off;
```

This will create the following output.

#### The TTEST Procedure

Variable: y

Output Added:

```
-----
Name:      Statistics
Label:     Statistics
Template:  Stat.TTest.Statistics
Path:      Ttest.y.Statistics
-----
```

| x          | N  | Mean    | Std Dev | Std Err | Minimum | Maximum |
|------------|----|---------|---------|---------|---------|---------|
| 0          | 50 | -0.0253 | 0.8473  | 0.1198  | -1.7148 | 1.8998  |
| 1          | 50 | 0.9700  | 0.9937  | 0.1405  | -0.9282 | 3.0741  |
| Diff (1-2) |    | -0.9953 | 0.9234  | 0.1847  |         |         |

|               |                       |         |         |         |         |
|---------------|-----------------------|---------|---------|---------|---------|
| Variable:     | y                     |         |         |         |         |
| Output Added: |                       |         |         |         |         |
| -----         |                       |         |         |         |         |
| Name:         | ConfLimits            |         |         |         |         |
| Label:        | Confidence Limits     |         |         |         |         |
| Template:     | Stat.TTest.ConfLimits |         |         |         |         |
| Path:         | Ttest.y.ConfLimits    |         |         |         |         |
| -----         |                       |         |         |         |         |
| x             | Method                | Mean    | 95% CL  | Mean    | Std Dev |
| 0             |                       | -0.0253 | -0.2661 | 0.2155  | 0.8473  |
| 1             |                       | 0.9700  | 0.6876  | 1.2524  | 0.9937  |
| Diff (1-2)    | Pooled                | -0.9953 | -1.3618 | -0.6288 | 0.9234  |
| Diff (1-2)    | Satterthwaite         | -0.9953 | -1.3619 | -0.6287 |         |
| x             | Method                | 95% CL  | Std Dev |         |         |
| 0             |                       | 0.7078  | 1.0558  |         |         |
| 1             |                       | 0.8301  | 1.2383  |         |         |
| Diff (1-2)    | Pooled                | 0.8103  | 1.0736  |         |         |
| Diff (1-2)    | Satterthwaite         |         |         |         |         |
| Variable:     | y                     |         |         |         |         |
| Output Added: |                       |         |         |         |         |
| -----         |                       |         |         |         |         |
| Name:         | TTests                |         |         |         |         |
| Label:        | T-Tests               |         |         |         |         |
| Template:     | Stat.TTest.TTests     |         |         |         |         |
| Path:         | Ttest.y.TTests        |         |         |         |         |
| -----         |                       |         |         |         |         |
| Method        | Variances             | DF      | t Value | Pr >  t |         |
| Pooled        | Equal                 | 98      | -5.39   | <.0001  |         |
| Satterthwaite | Unequal               | 95.612  | -5.39   | <.0001  |         |

```
Variable: y

Output Added:
-----
Name:      Equality
Label:     Equality of Variances
Template:  Stat.TTest.Equality
Path:      Ttest.y.Equality
-----
```

### Equality of Variances

| Method   | Num DF | Den DF | F Value | Pr > F |
|----------|--------|--------|---------|--------|
| Folded F | 49     | 49     | 1.38    | 0.2680 |

Note that failing to issue the `ods trace off` command will result in continued annotation of every piece of output. Similarly, when using the `ods exclude` and `ods select` statements, it is good practice to conclude each procedure with an `ods select all` or `ods exclude none` statement so that later output will not be affected.

The previous output shows that the *t* test itself (including the tests assuming equal and unequal variances) appears in output which the ODS system calls `ttests`, so the following code demonstrates how the test results can be saved into a new dataset. Here we assign the new dataset the name `appendixattest`.

```
ods output ttests=appendixattest;
proc ttest data=test2;
  class x;
  var y;
run;

proc print data=appendixattest;
run;
```

The `proc print` code results in the following output.

| Obs | Variable | Method        | Variances | tValue | DF     | ProbT  |
|-----|----------|---------------|-----------|--------|--------|--------|
| 1   | y        | Pooled        | Equal     | -5.39  | 98     | <.0001 |
| 2   | y        | Satterthwaite | Unequal   | -5.39  | 95.612 | <.0001 |

To run the *t* test and print only these results, the following code would be used.

```
ods select ttests;
proc ttest data=test2;
  class x;
  var y;
run;
ods select all;
```

Variable: y

| Method        | Variances | DF     | t Value | Pr >  t |
|---------------|-----------|--------|---------|---------|
| Pooled        | Equal     | 98     | -5.39   | <.0001  |
| Satterthwaite | Unequal   | 95.612 | -5.39   | <.0001  |

This application is especially useful when running simulations, as it allows the results of procedures to be easily stored for later analysis.

The foregoing barely scratches the surface of what is possible using **ODS**. For further information, refer to the on-line help: Contents; SAS Products; Base SAS; SAS Output Delivery System User's Guide.

### A.7.2 Output file types and ODS destinations

The other main use of the **ODS** system is to generate output in a variety of file types. By default, SAS output is printed in a **Results** window in the internal GUI. When run in batch mode, or when saving the contents of the output window using the GUI, this output is saved as a plain text file with a **.lst** extension. The **ODS** system provides a way to save SAS output in a more attractive form. As discussed in 9.3, procedure output and graphics can be saved to named output files by using commands of the following form.

```
ods destinationname file="filename.ext";
```

Valid **destinationnames** include **pdf**, **rtf**, **latex**, and others. SAS refers to these file types as “destinations.” It is possible to have multiple destinations open at the same time. For destinations other than **listing** (the Output window), the **destination** must be closed before the results can be seen. This is done using the following statement.

```
ods destinationname close;
```

Note that the **listing** destination can also be closed. If there are no output destinations open, no results will be displayed.

## A.8 SAS macro variables

SAS also includes what are known as *macro variables*. Unlike SAS macros, macro variables are values that exist during SAS runs and are not stored within datasets. One way to assign a value to a macro variable is the **%let** statement.

```
%let macrovar=chars;
```

Note that the **%let** statement need not appear within a **data** step; it is a global statement. The value is stored as a string of characters, and can be referred to as **&macrovar**.

```
data ds;
  newvar=&macrovar;
run;
```

or

```
title "This is the &macrovar";
```

In the above example, the double quotes in the **title** statement allow the text within to be processed to assess whether macro variables are present, and to replace them with text, if so. Enclosing the title text in single quotes will result in **&macrovar** appearing in the title, while the code above will replace **&macrovar** with the value of the **macrovar** macro variable.

While this basic application of macro variables is occasionally useful in coding, a more powerful application is to generate the macro variables within a SAS **data** step. This can be done using a **call symput** function, as shown in 5.7.4.

```
data _null_;
 $\dots$ 
call symput('macrovar', x);
run;
```

This makes a new macro variable named **macrovar** which has the value of the data set variable **x**. The **\_null\_** dataset is a special SAS dataset which is not saved. It is efficient to use it when there is no need for a stored dataset. The formulation here can be used, for

example, to calculate a value during a data step and pass it to the title of a figure. We demonstrate the use of `call symput` in section 5.7.4.

## A.9 Miscellanea

Official documentation provided by SAS refers to, for example `PROC GLM`. However, SAS is not case sensitive, with a few exceptions. In this text we use lower case throughout. We find lower case easier to read, and prefer the ease of typing (both for coding and book composition) in lower case.

Since statements are separated by semicolons, multiple statements may appear on one line and statements may span lines. We usually type one statement per line in the text (and in practice), however. This prevents statements being overlooked among others appearing in the same line. In addition, we indent statements within a data step or proc, to clarify the grouping of related commands.

We prefer the fine control available through text-based commands. However, some people may prefer a point-and-click interface to the analytic tools available. SAS provides various applications for such an approach which can be accessed through the **Solutions** menu in the GUI.

SAS includes both `run` and `quit` statements. The `run` statement tells SAS to act on the code submitted since the most recent `run` statement (or since startup, if there has been no `run` statement submitted thus far). Some procedures allow multiple steps within the procedure without having to end it; key procedures which allow this are `proc gplot` and `proc reg`. This might be useful for model fitting and diagnostics with particularly large datasets in `proc reg`. In general we find it a nuisance in graphics procedures, because the graphics are sometimes not actually drawn until the `quit` statement is entered. In the examples, we use the `run` statement in general and the `quit` statement when necessary, without further comment.

We find the SAS GUI to be a comfortable work environment and an aid to productivity. However, SAS can be easily run in batch mode. To use SAS this way, compose code in the text editor of your choice. Save the file (a `.sas` extension would be appropriate), then find it in the operating system. In Windows, a right-click on the file will bring up a list of potential actions, one of which is “Batch Submit with SAS”. If this option is selected, SAS will run the file without opening the GUI. The output will be saved in the same directory with the same name but with a `.lst` extension; the log will be saved in the same directory with the same name but with a `.log` extension. Both of these files are plain text files.

## Appendix B

# Introduction to R and RStudio

This chapter provides a (brief) introduction to R and RStudio. R is a free, open-source software environment for statistical computing and graphics [81, 135]. RStudio is an open-source integrated development environment for R that adds many features and productivity tools for R. The chapter includes a short history, installation information, a sample session, background on fundamental structures and actions, information about help and documentation, and other important topics.

R is a general purpose package that includes support for a wide variety of modern statistical and graphical methods (many of which have been contributed by users). It is available for most UNIX platforms, Windows, and MacOS. The R Foundation for Statistical Computing holds and administers the copyright of R software and documentation. R is available under the terms of the Free Software Foundation's GNU General Public License in source code form.

RStudio facilitates use of R by integrating R help and documentation, providing a workspace browser and data viewer, and supporting syntax highlighting, code completion, and smart indentation. It integrates reproducible analysis with Sweave, knitr and R Markdown (see 11.3) as well as slide presentations, and includes a debugging environment (see 4.1.7). RStudio also provides support for multiple projects as well as an interface to source code control systems such as GitHub. It has become the default interface for many R users, including the authors.

RStudio is available as a client (standalone) for Windows, Mac OS X, and Linux, and there is also a server version. Commercial products and support are available in addition to the open-source offerings (see <http://www.rstudio.com/ide> for details).

The first versions of R were written by Ross Ihaka and Robert Gentleman at the University of Auckland, New Zealand, while current development is coordinated by the R Development Core Team, a group of international volunteers. As of January 2014 this group consisted of Douglas Bates, John Chambers, Peter Dalgaard, Seth Falcon, Robert Gentleman, Kurt Hornik, Stefano Iacus, Ross Ihaka, Friedrich Leisch, Uwe Ligges, Thomas Lumley, Martin Maechler, Duncan Murdoch, Paul Murrell, Martyn Plummer, Brian Ripley, Deepayan Sarkar, Duncan Temple Lang, Luke Tierney, and Simon Urbanek. Many hundreds of other people have contributed to the development of R or developed add-on libraries and packages.

R is similar to the S language, a flexible and extensible statistical environment originally developed in the 1980s at AT&T Bell Labs (now Alcatel-Lucent). Insightful Corporation has continued the development of S in their commercial software package S-PLUS™.

New users are encouraged to download and install R from the Comprehensive R archive network (CRAN, <http://www.r-project.org>, see B.1) and install RStudio from <http://www.rstudio.com/ide>.

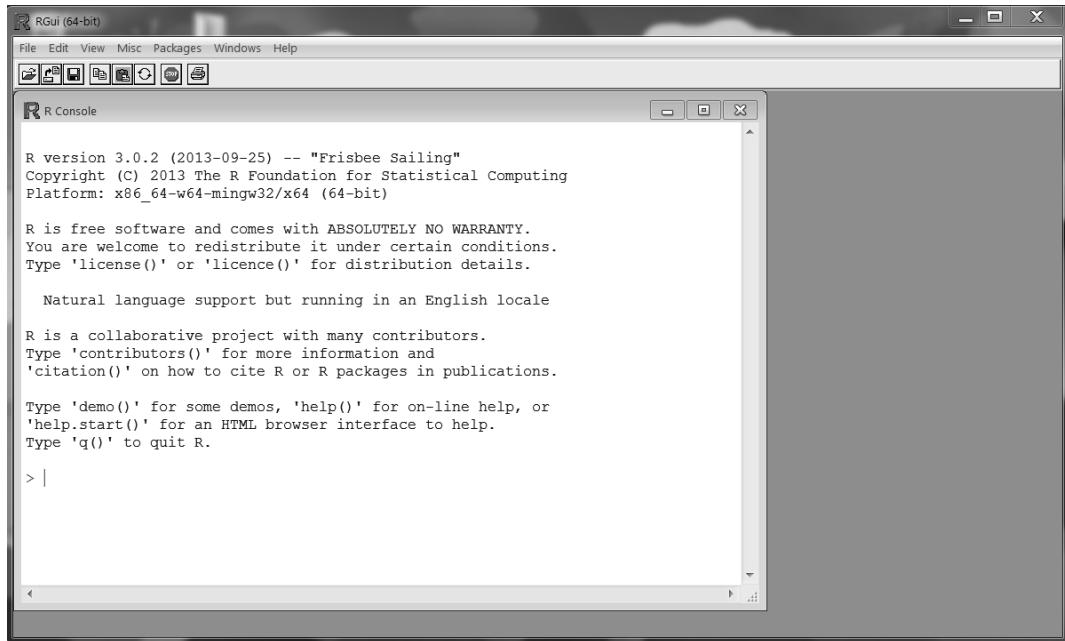


Figure B.1: R Windows graphical user interface

//[www.rstudio.com/ide](http://www.rstudio.com/ide). The sample session in the appendix of the *Introduction to R* document, also available from CRAN (see B.2), is highly recommended.

## B.1 Installation

The home page for the R project, located at <http://r-project.org>, is the best starting place for information about the software. It includes links to CRAN, which features pre-compiled binaries as well as source code for R, add-on packages, documentation (including manuals, frequently asked questions, and the R newsletter) as well as general background information. Mirrored CRAN sites with identical copies of these files exist all around the world. Updates to R and packages are regularly posted on CRAN. In addition to the instructions for installation under Windows and Mac OS X, R and RStudio are also available for multiple Linux implementations.

### B.1.1 Installation under Windows

Versions of R for Windows XP and later, including 64-bit versions, are available at CRAN. The distribution includes **Rgui.exe**, which launches a self-contained windowing system that includes a command-line interface, **Rterm.exe** for a command-line interface only, **Rscript.exe** for batch processing only, and **R.exe**, which is suitable for batch or command-line use. A screenshot of the R graphical user interface (GUI) can be found in Figure B.1. More information on Windows-specific issues can be found in the CRAN *R for Windows FAQ* (<http://cran.r-project.org/bin/windows/base/rw-FAQ.html>).

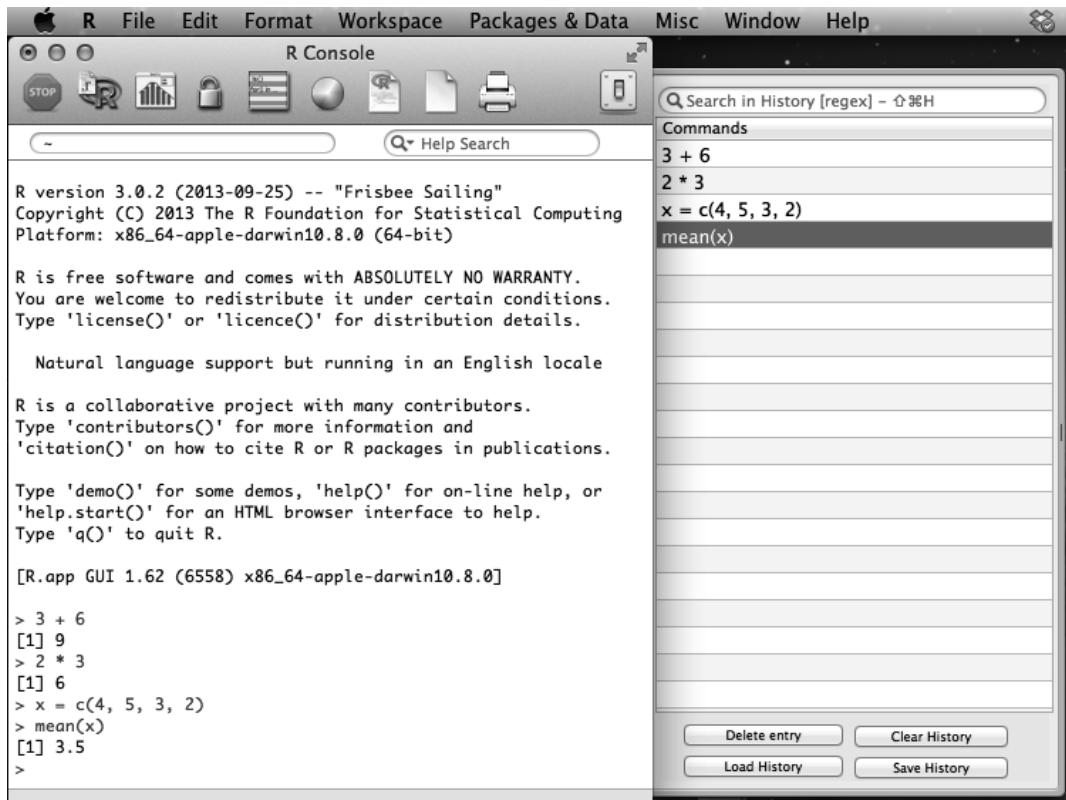


Figure B.2: R Mac OS X graphical user interface

### B.1.2 Installation under Mac OS X

A version of R for Mac OS X 10.6 and higher is available at CRAN. This is distributed as a disk image containing the installer. In addition to the graphical interface version, a command line version (particularly useful for batch operations) can be run as the command `R`. A screenshot of the graphical interface can be found in Figure B.2.

More information on Macintosh-specific issues can be found in the CRAN *R for Mac OS X FAQ* (<http://cran.r-project.org/bin/macosx/RMacOSX-FAQ.html>).

### B.1.3 RStudio

RStudio for MacOS, Windows, or Linux can be downloaded from <http://www.rstudio.com/ide>. RStudio requires R to be installed on the local machine. A server version (accessible from web browsers) is also available for download. Documentation of the advanced features in the system is available on the RStudio web site. A screenshot of the RStudio interface can be found in Figure B.3.

### B.1.4 Other graphical interfaces

Other graphical user interfaces for R include the R Commander project [45], Deducer (<http://www.deducer.org>), and the SOCR (Statistics Online Computational Resource) project (<http://www.socr.ucla.edu>).

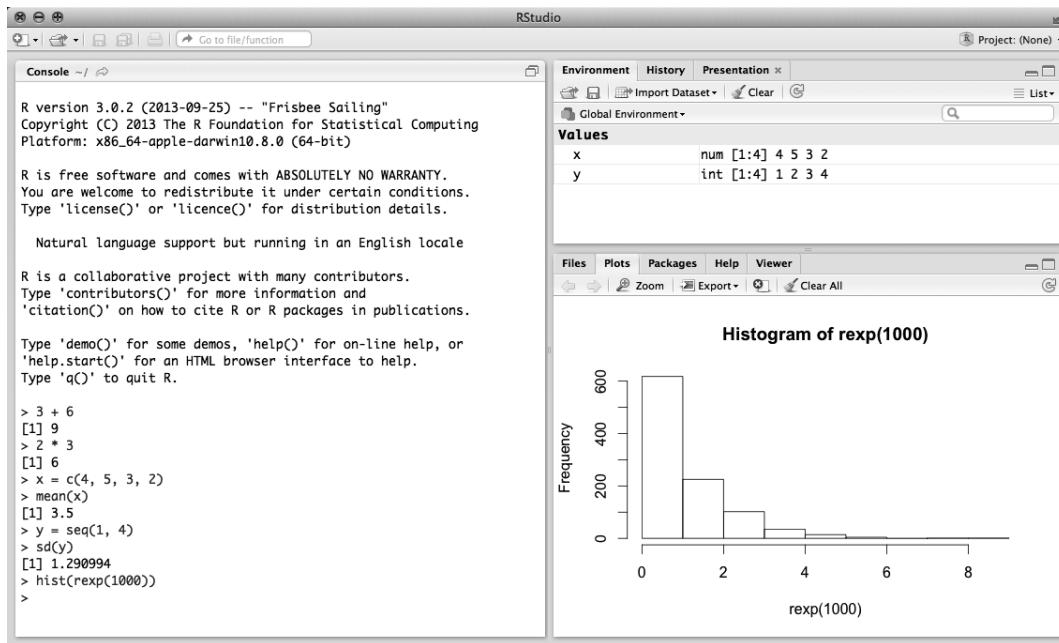


Figure B.3: RStudio graphical user interface

## B.2 Running R and sample session

Once installation is complete, the recommended next step for a new user would be to start R and run a sample session. An example from the command line interface within Mac OS X is given in Figure B.4.

The `>` character is the command prompt, and commands are executed once the user presses the RETURN or ENTER key. R can be used as a calculator (as seen from the first two commands on lines 1 and 3). New variables can be created (as on lines 5 and 8) using the assignment operator `=`. If a command generates output (as on lines 6 and 11), then it is printed on the screen, preceded by a number indicating place in the vector (this is particularly useful if output is longer than one line, e.g., lines 23–24). Saved data (here assigned the name `ds`) is read into R on line 15, then summary statistics are calculated (lines 16–17) and individual observations are displayed (lines 23–24). The `$` operator allows access to objects within a dataframe. Alternatively, the `with()` function can be used to access objects within a dataset.

Unlike SAS, R is case sensitive.

```
> x = 1:3
> X = seq(2, 4)
> x

[1] 1 2 3

> X

[1] 2 3 4
```

A very comprehensive sample session in R can be found in Appendix A of *An Introduction to R* [186] (<http://cran.r-project.org/doc/manuals/R-intro.pdf>).

```
% R
R version 3.0.2 (2013-09-25) -- "Frisbee Sailing"
Copyright (C) 2013 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin10.8.0 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

1 > 3 + 6
2 [1] 9
3 > 2 * 3
4 [1] 6
5 > x = c(4, 5, 3, 2)
6 > x
7 [1] 4 5 3 2
8 > y = seq(1, 4)
9 > y
10 [1] 1 2 3 4
11 > mean(x)
12 [1] 3.5
13 > sd(y)
14 [1] 1.290994
15 > ds = read.csv("http://www.amherst.edu/~nhorton/sasr2/datasets/help.csv")
16 > mean(ds$age)
17 [1] 35.65342
18 > mean(age)
19 Error in mean(age) : object "age" not found
20 > with(ds, mean(age))
21 [1] 35.65342
22 > ds$age[1:30]
23 [1] 37 37 26 39 32 47 49 28 50 39 34 58 53 58 60 36 28 35 29 27 27
24 [22] 41 33 34 31 39 48 34 32 35
25 > q()
26 Save workspace image? [y/n/c]: n
```

Figure B.4: Sample session in R

### B.2.1 Replicating examples from the book and sourcing commands

To help facilitate reproducibility, R commands can be bundled into a plain text file, called a “script” file, which can be executed using the `source()` command. The optional argument `echo=TRUE` for the `source()` command can be set to display each command and its output.

The book web site cited above includes the R source code for the examples. The sample session in Figure B.4 can be executed by running:

```
> source("http://www.amherst.edu/~nhorton/sasr2/examples/sampsess.R",
  echo=TRUE)
```

while most of the examples at the end of each chapter can be executed by running:

```
> source("http://www.amherst.edu/~nhorton/sasr2/examples/chapterXX.R",
  echo=TRUE)
```

where XX is replaced by the desired chapter number. In many cases, add-on packages (see B.6.1) need to be installed prior to running the examples. To facilitate this process, we have created a script file to load them in one step.

```
> source("http://www.amherst.edu/~nhorton/sasr2/examples/install.R",
  echo=TRUE)
```

If needed libraries are not installed (B.6.1), the example code will generate error messages.

## B.2.2 Batch mode

In addition, R can be run in batch (noninteractive) mode from a command line interface:

```
% R CMD BATCH file.R
```

This will run the commands contained within `file.R` and put all output into `file.Rout`. To use R in batch mode under Windows, users need to include `R.exe` in their path (see the Windows R FAQ and section B.1.1).

## B.3 Learning R and getting help

An excellent starting point for new R users can be found in the *Introduction to R*, available from CRAN ([r-project.org](http://r-project.org)).

The system features extensive on-line documentation, though as with SAS, it can sometimes be challenging to comprehend. Each command in R has an associated help file that describes usage, lists arguments, provides details of actions, references, lists other related functions, and includes examples of its use. The help system is invoked using the command:

```
?function
```

or

```
help(function)
```

where `function` is the name of the function of interest. As an example, the help file for the `mean()` function is accessed by the command `help(mean)`. The output from this command is provided in Figure B.5.

It describes the `mean()` function as a generic function for the (trimmed) arithmetic mean, with arguments `x` (an R object), `trim` (the fraction of observations to trim, with default=0; setting `trim=0.5` is equivalent to calculating the median), and `na.rm` (should missing values be deleted, default is `na.rm=F`).

Some commands (e.g., `if`) are reserved, so `?if` will not generate the desired documentation. Running `?"if"` will work (see also `?Reserved` and `?Control`). Other reserved words include `else`, `repeat`, `while`, `function`, `for`, `in`, `next`, `break`, `TRUE`, `FALSE`, `NULL`, `Inf`, `NaN`, and `NA`.

The `RSiteSearch()` function will search for key words or phrases in many places (including the search engine at <http://search.r-project.org>). A screenshot of the results of the command `RSiteSearch("eta squared anova")` can be found in Figure B.6. The `RSeek.org` site can also be helpful in finding more information and examples.

Examples of many functions are available using the `example()` function.

```

mean                  package:base          R Documentation

Arithmetic Mean

Description:
  Generic function for the (trimmed) arithmetic mean.

Usage:
  mean(x, ...)

  ## Default S3 method:
  mean(x, trim = 0, na.rm = FALSE, ...)

Arguments:
  x: An R object.  Currently there are methods for numeric/logical
     vectors and date, date-time and time interval objects.
     Complex vectors are allowed for 'trim = 0', only.

  trim: the fraction (0 to 0.5) of observations to be trimmed from
     each end of 'x' before the mean is computed. Values of trim
     outside that range are taken as the nearest endpoint.

  na.rm: a logical value indicating whether 'NA' values should be
     stripped before the computation proceeds.

  ...: further arguments passed to or from other methods.

Value:
  If 'trim' is zero (the default), the arithmetic mean of the values
  in 'x' is computed, as a numeric or complex vector of length one.
  If 'x' is not logical (coerced to numeric), numeric (including
  integer) or complex, 'NA_real_' is returned, with a warning.

  If 'trim' is non-zero, a symmetrically trimmed mean is computed
  with a fraction of 'trim' observations deleted from each end
  before the mean is computed.

References:
  Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) The New S
  Language. Wadsworth & Brooks/Cole.

See Also:
  'weighted.mean', 'mean.POSIXct', 'colMeans' for row and column
  means.

Examples:
  x <- c(0:10, 50)
  xm <- mean(x)
  c(xm, mean(x, trim = 0.10))

```

Figure B.5: Documentation on the `mean()` function

**R Site Search**

Query:   [How to search]

Display:  Description:  Sort:

Target:

Functions  
 Vignettes  
 Task views

For problems WITH THIS PAGE (not with R) contact [baron@psych.upenn.edu](mailto:baron@psych.upenn.edu).

**Results:**

References:

- **views:** [ eta: 1 ] [ squared: 2 ] [ anova: 7 ] [ TOTAL: 0 ]
- **vignettes:** [ eta: 49 ] [ squared: 406 ] [ anova: 193 ] [ TOTAL: 5 ]
- **functions:** [ eta: 962 ] [ squared: 3358 ] [ anova: 1992 ] [ TOTAL: 14 ]

Total 19 documents matching your query.

1. **R: Measures of Partial Association (Eta-squared) for Linear...** (score: 28)
 

Author: unknown  
 Date: Thu, 05 Sep 2013 10:24:33 -0500  
 Measures of Partial Association (Eta-squared) for Linear Models Description Usage Arguments Details Value Author(s) References See Also Examples page for etasq {heplots} etasq {heplots} R Documentation <http://finzi.psych.upenn.edu/R/library/heplots/html/etasq.html> (4,148 bytes)
2. **R: Make nice ANOVA table for printing.** (score: 9)
 

Author: unknown  
 Date: Sat, 07 Dec 2013 07:14:08 -0500  
 Make nice ANOVA table for printing. Description Usage Arguments Details Value Author(s) References See Also Examples page for nice.anova {afex} nice.anova {afex} R Documentation These functions prod <http://finzi.psych.upenn.edu/R/library/afex/html/nice.anova.html> (6,554 bytes)
3. **R: Effect size calculations for ANOVAs** (score: 6)
 

Author: unknown  
 Date: Wed, 20 Nov 2013 08:27:15 -0500  
 Effect size calculations for ANOVAs Description Usage Arguments Details Value Warning Author(s) See Also Examples page for etaSquared {lsr} etaSquared {lsr} R Documentation Calculates eta-squared an <http://finzi.psych.upenn.edu/R/library/lsr/html/etaSquared.html> (3,770 bytes)

Figure B.6: Display after running `RSiteSearch("eta squared anova")`

```
> example(mean)

mean> x <- c(0:10, 50)

mean> xm <- mean(x)

mean> c(xm, mean(x, trim = 0.10))
[1] 8.75 5.50
```

Other useful resources are `help.start()`, which provides a set of online manuals, and `help.search()`, which can be used to look up entries by description. The `apropos()` command returns any functions in the current search list that match a given pattern (which facilitates searching for a function based on what it does, as opposed to its name).

Other resources for help available from CRAN include the R-help mailing list (see also B.7, support). New users are also encouraged to read the R FAQ (frequently asked questions) list.

## B.4 Fundamental structures and objects

Here we provide a brief introduction to R data structures.

### B.4.1 Objects and vectors

Almost everything in R is an object, which may be initially disconcerting to a new user. An object is simply something that R can operate on. Common objects include vectors, matrices, arrays, factors (see 2.2.19), dataframes (akin to datasets in SAS), lists, and functions.

The basic variable structure is a vector. Vectors can be created using the `<-` or `=` assignment operators (which assigns the evaluated expression on the right-hand side of the operator to the object name on the left-hand side).

```
> x <- c(5, 7, 9, 13, -4, 8)
> x = c(5, 7, 9, 13, -4, 8) # equivalent
```

The above code creates a vector of length 6 using the `c()` function to concatenate scalars (2.2.10). The `=` operator must be used for the specification of options for functions. Other assignment operators exist, as well as the `assign()` function (see 4.1.5 or `help("<-")` for more information). The `rm()` command can be used to remove objects. The `exists()` function can be utilized to determine whether an object exists.

### B.4.2 Indexing

Since vector operations are so common in R, it is important to be able to access (or index) elements within these vectors. Many different ways of indexing vectors are available. Here, we introduce several of these, using the above example. The command `x[2]` would return the second element of `x` (the scalar 7), and `x[c(2,4)]` would return the vector (7,13). The expressions `x[c(T,T,T,T,F)]`, `x[1:5]`, and `x[-6]` would all return a vector consisting of the first five elements in `x`; the last specifies all elements except the 6th. Knowledge and basic comfort with these approaches to vector indexing are important to effective use of R, as they can help with computational efficiency.

Vectors are recycled if needed, for example, when comparing each of the elements of a vector to a scalar, as shown below.

```
> x>8
[1] FALSE FALSE TRUE TRUE FALSE FALSE
```

The above expression demonstrates the use of comparison operators (see `?Comparison`).

Only the third and fourth elements of `x` are greater than 8. The function returns a logical value of either `TRUE` or `FALSE` (see `?Logic`).

A count of elements meeting the condition can be generated using the `sum()` function.

```
> sum(x>8)
```

```
[1] 2
```

The following commands create a vector of values greater than 8.

```
> largerthan8 = x[x>8]
> largerthan8
```

```
[1] 9 13
```

Here the expression `x[x>8]` can be interpreted as “the elements of `x` for which `x` is greater than 8.” This is a difficult construction for some new users. Examples of its application in the book can be found in 11.4.4 and 2.6.2.

Other comparison operators include `==` (equal), `>=` (greater than or equal), `<=` (less than or equal) and `!=` (not equal). Care needs to be taken in the comparison using `==` if noninteger values are present (see 3.2.5).

### B.4.3 Operators

There are many operators defined in R to carry out a variety of tasks. Many of these were demonstrated in the sample session (assignment, arithmetic) and above examples (comparison). Arithmetic operations include `+`, `-`, `*`, `/`, `^` (exponentiation), `%%` (modulus), and `&/&` (integer division). More information about operators can be found using the help system (e.g., `?"+"`). Background information on other operators and precedence rules can be found using `help(Syntax)`.

R supports Boolean operations (OR, AND, NOT, and XOR) using the `|`, `||`, `&`, `!` operators and the `xor()` function. The `|` is an “or” operator that operates on each element of a vector, while the `||` is another “or” operator stops evaluation the first time that the result is true (see `?Logic`).

### B.4.4 Lists

Lists in R are generic objects that can contain other objects. List members can be named, or referenced using numeric indices (using the `[[` operator).

```
> newlist = list(x1="hello", x2=42, x3=TRUE)
> is.list(newlist)

[1] TRUE

> newlist

$x1
[1] "hello"

$x2
[1] 42

$x3
[1] TRUE

> newlist[[2]]

[1] 42

> newlist$x2

[1] 42
```

The `unlist()` function can be used to flatten (make a vector out of) the elements in a list (see also `relist()`).

```
> unlisted = unlist(newlist)
> unlisted

  x1      x2      x3
"hello"    "42"   "TRUE"
```

Note that unlisted objects are coerced (see 2.2.3) to a common type (in this case `character`).

### B.4.5 Matrices

Matrices are rectangular objects with two dimensions (see 3.3). We can create a  $2 \times 3$  matrix, display it, and test for its type.

```
> A = matrix(x, 2, 3)
> A

 [,1] [,2] [,3]
[1,]    5    9   -4
[2,]    7   13    8

> is.matrix(A)    # is A a matrix?

[1] TRUE

> is.vector(A)

[1] FALSE

> is.matrix(x)

[1] FALSE
```

Note that comments are supported within R (any input given after a `#` character is ignored).

Indexing for matrices is done in a similar fashion as for vectors, albeit with a second dimension (denoted by a comma).

```
> A[2,3]

[1] 8

> A[,1]

[1] 5 7

> A[1,]

[1] 5 9 -4
```

### B.4.6 Dataframes

Analysis datasets are often stored in a dataframe, which is more general than a matrix. This rectangular object, similar to a dataset in SAS, can be thought of as a matrix with columns of vectors of different types (as opposed to a matrix, which consists of vectors of the same type). The functions `read.csv()` (see 1.1.5) and `read.table()` (see 1.1.2) return dataframe objects. A simple dataframe can be created using the `data.frame()` command. Access to sub-elements is achieved using the `$` operator, as shown below (see also `help(Extract)`).

In addition, operations can be performed by column (e.g., calculation of sample statistics).

```
> y = rep(11, length(x))
> y

[1] 11 11 11 11 11 11 11 11

> ds = data.frame(x, y)

> ds

  x  y
1 5 11
2 7 11
3 9 11
4 13 11
5 -4 11
6 8 11

> ds$x[3]

[1] 9
```

We can check to see if an object is a data frame with `is.data.frame()`.

Note that the use of `data.frame()` differs from the use of `cbind()`, which yields a matrix object (unless `cbind()` is given data frames as inputs).

```
> newmat = cbind(x, y)
> newmat

  x  y
[1,] 5 11
[2,] 7 11
[3,] 9 11
[4,] 13 11
[5,] -4 11
[6,] 8 11

> is.data.frame(newmat)

[1] FALSE
```

```
> is.matrix(newmat)
```

```
[1] TRUE
```

Dataframes can be thought of as the equivalent of datasets in SAS. They can be created from matrices using `as.data.frame()`, while matrices can be constructed from dataframes using `as.matrix()`.

Dataframes can be attached to the workspace using the `attach(ds)` command (see 2.1.1), though this is discouraged [56]. After this command, individual columns in `ds` can be referenced directly by name (e.g., `x` instead of `ds$x`). Name conflicts are a common problem with `attach()` (see `conflicts()`).

The `search()` function lists attached packages and objects. To avoid cluttering the name-space, the command `detach(ds)` should be used once a dataframe is no longer needed.

The `with()` and `within()` commands (see 2.1.1) can be used to simplify reference to an object within a dataframe without attaching.

Sometimes it's desirable to remove a package (B.6.1) from the workspace. For example, a package might define a function (4.2.2) with the same name as an existing function. Packages can be detached using the syntax `detach("package:PKGNAME")`, where `PKGNAME` is the name of the package (see, for example, 7.10.5).

The names of all variables within a given dataset (or more generally for sub-objects within an object) are provided by the `names()` command. The names of all objects defined within an R session can be generated using the `objects()` and `ls()` commands, which return a vector of character strings. RStudio includes an `Environment` tab that lists all the objects in the current environment.

The `print()` and `summary()` functions can be used to display simple and more complex descriptions, respectively, of an object. Running `print(object)` at the command line is equivalent to just entering the name of the object, i.e., `object`.

## B.4.7 Attributes and classes

Objects have a set of associated attributes (such as names of variables, dimensions, or classes) which can be displayed or sometimes changed. While a powerful concept, this can often be initially confusing. For example, we can find the dimension of the matrix defined earlier:

```
> attributes(A)

$dim
[1] 2 3
```

Other types of objects within R include lists (ordered objects that are not necessarily rectangular), regression models (objects of class `lm`), and formulae (e.g.,  $y \sim x_1 + x_2$ ). Examples of the use of formulas can be found in 5.4.2 and 6.1.1.

R supports object-oriented programming (see `help(UseMethod)`). As a result, objects within R have an associated “Class” attribute, which affects default behaviors for some operations on that object. Many functions have special capabilities when operating on a particular class. For example, when `summary()` is applied to an `lm` object, the `summary.lm()` function is called, while `summary.aov()` is called when an `aov` object is given as argument. The `class()` function returns the classes to which an object belongs, while the `methods()` function displays all of the classes supported by a function (e.g., `methods(summary)`).

The `attributes()` command displays the attributes associated with an object, while the `typeof()` function provides information about the object (e.g., logical, integer, double, complex, character, and list).

## B.4.8 Options

The `options()` function in R can be used to change various default behaviors, for example, the default number of digits to display in output (`options(digits=n)` where `n` is the preferred number). Defaults described in the book include `digits`, `show.signif.stars`, and `width`. The previous options are returned when `options()` is called (see 8.7.7), to allow them to be restored. The command `help(options)` lists all of the settable options.

# B.5 Functions

## B.5.1 Calling functions

Fundamental actions within R are carried out by calling functions (either built-in or user defined). Multiple arguments may be given, separated by commas. The function carries out

operations using the provided arguments then returns values (an object such as a vector or list) that are displayed (by default) or which can be saved by assignment to an object.

As an example, the `quantile()` function takes a vector and returns the minimum, 25th percentile, median, 75th percentile and maximum, though if an optional vector of quantiles is given, those are calculated instead.

```
> vals = rnorm(1000) # generate 1000 standard normals
> quantile(vals)

 0%    25%    50%    75%   100%
-2.8478 -0.6288  0.0802  0.7634  3.2597

> quantile(vals, c(.025, .975))

 2.5% 97.5%
-1.94  2.12
```

Return values can be saved for later use.

```
> res = quantile(vals, c(.025, .975))
> res[1]

 2.5%
-1.94
```

Options are available for many functions. These are named arguments for the function and are generally added after the other arguments, also separated by commas. The documentation specifies the default action if named arguments (options) are not specified. For the `quantile()` function, there is a `type()` option which allows specification of one of nine algorithms for calculating quantiles. Setting `type=3` specifies the “nearest even order statistic” option, which is the default for SAS.

```
> res = quantile(vals, c(.025, .975), type=3)
```

Some functions allow a variable number of arguments. An example is the `paste()` function (see usage in 2.2.10). The calling sequence is described in the documentation as follows.

```
> paste(..., sep=" ", collapse=NULL)
```

To override the default behavior of a space being added between elements output by `paste()`, the user can specify a different value for `sep`.

## B.5.2 The `apply` family of functions

Operations within R are most efficiently carried out using vector or list operations rather than looping. The `apply()` function can be used to perform many actions that would be implemented within a data step (A.4) within SAS. While somewhat subtle, the power of the vector language can be seen in this example. The `apply()` command is used to calculate column means or row means of the previously defined matrix in one fell swoop:

```
> A
 [,1] [,2] [,3]
[1,]    5    9   -4
[2,]    7   13    8

> apply(A, 2, mean)
[1] 6 11 2

> apply(A, 1, mean)
[1] 3.33 9.33
```

Option 2 specifies that the mean should be calculated for each column, while option 1 calculates the mean of each row. Here we see some of the flexibility of the system, as functions in R (such as `mean()`) are also objects that can be passed as arguments to functions.

Other related functions include `lapply()`, which is helpful in avoiding loops when using lists, `sapply()` (see 2.1.2), `mapply()`, and `vapply()` to do the same for dataframes, matrices, and vectors, respectively, and `tapply()` (11.1) performs an action on subsets of an object. The `foreach` and `plyr` package provides equivalent formulations for parallel execution (see also the `parallel` package).

## B.6 Add-ons: packages

### B.6.1 Introduction to packages

Additional functionality in R is added through packages, which consist of libraries of bundled functions, datasets, examples, vignettes, and help files that can be downloaded from CRAN. The function `install.packages()` or the windowing interface under *Packages and Data* must be used to download and install packages. RStudio provides an easy to use *Packages* tab to install and load packages.

The `library()` function can be used to load a previously installed package (i.e., one that is included in the standard release of R or has been previously made available through use of the `install.packages()` function). As an example, to install and load Frank Harrell's `Hmisc` package, two commands are needed:

```
> install.packages("Hmisc")
> library(Hmisc)
```

Once a package has been installed, it can be loaded for use in a session of R by executing the function `library(libraryname)`. If a package is not installed, running the `library()` command will yield an error. Here we try to load the `Zelig` package (which had not yet been installed):

```
> library(Zelig)

Error in library(Zelig) : there is no package called 'Zelig'
```

```
> install.packages("Zelig")
trying URL 'ftp.osuosl.org/pub/cran/bin/macosx/contrib/Zelig_4.2-1.tgz'
Content type 'application/x-gzip' length 3374792 bytes (3.2 Mb)
opened URL
=====
downloaded 3.2 Mb
The downloaded binary packages are in
/var/folders/2j/RtmpXPJ4o0/downloaded_packages

> library(Zelig)

ZELIG (Versions 4.2-1, built: 2013-09-12)
+-----+
| Please refer to http://gking.harvard.edu/zelig for full |
| documentation or help.zelig() for help with commands and |
| models support by Zelig. |
| Zelig project citations: |
|   Kosuke Imai, Gary King, and Olivia Lau. (2009). |
|   ``Zelig: Everyone's Statistical Software,''
|   http://gking.harvard.edu/zelig
|   and
|   Kosuke Imai, Gary King, and Olivia Lau. (2008).
|   ``Toward A Common Framework for Statistical Analysis
|   and Development,'' Journal of Computational and
|   Graphical Statistics, Vol. 17, No. 4 (December)
|   pp. 892-913.
+-----+
Attaching package: 'Zelig'
```

A user can test whether a package is available by running `require(packagename)`; this will load the library if it is installed, and generate a warning message if it is not (as opposed to `library()`, which will return an error, see 4.1.8). This is particularly useful in functions or reproducible analysis.

The `update.packages()` function should be run periodically to ensure that packages are up-to-date. The `sessionInfo()` command displays the version of R that is running as well as information on all loaded packages.

As of January 2014, there were more than 5,000 packages available from CRAN. This represents a tremendous investment of time and code by many developers [46]. While each of these has met a minimal standard for inclusion, it is important to keep in mind that packages within R are created by individuals or small groups, and not endorsed by the R core group. As a result, they do not necessarily undergo the same level of testing and quality assurance that the core R system does.

## B.6.2 CRAN task views

The *Task Views* on CRAN (<http://cran.r-project.org/web/views>) are a very useful resource for finding packages. These are listings of relevant packages within a particular application area (such as multivariate statistics, psychometrics, or survival analysis). Table B.1 displays the task views available as of January 2014.

Table B.1: CRAN task views

|                             |   |
|-----------------------------|---|
| Bayesian                    | Bayesian inference                                  |
| ChemPhys                    | Chemometrics and computational physics              |
| Clinical Trials             | Design, monitoring, and analysis of clinical trials |
| Cluster                     | Cluster analysis & finite mixture models            |
| DifferentialEquations       | Differential equations                              |
| Distributions               | Probability distributions                           |
| Econometrics                | Computational econometrics                          |
| Environmetrics              | Analysis of ecological and environmental data       |
| Experimental Design         | Design and analysis of experiments                  |
| Finance                     | Empirical finance                                   |
| Genetics                    | Statistical genetics                                |
| Graphics                    | Graphic displays, devices, and visualization        |
| gR                          | Graphical models in R                               |
| High Performance Computing  | High-performance and parallel computing             |
| Machine Learning            | Machine and statistical learning                    |
| Medical Imaging             | Medical image analysis                              |
| MetaAnalysis                | Meta-analysis                                       |
| Multivariate                | Multivariate statistics                             |
| Natural Language Processing | Natural language processing                         |
| Numerical Mathematics       | Numerical mathematics                               |
| Official Statistics         | Official statistics & survey methodology            |
| Optimization                | Optimization and mathematical programming           |
| Pharmacokinetics            | Analysis of pharmacokinetic data                    |
| Psychometrics               | Psychometric models and methods                     |
| Reproducible Research       | Reproducible research                               |
| Robust                      | Robust statistical methods                          |
| Social Sciences             | Statistics for the social sciences                  |
| Spatial                     | Analysis of spatial data                            |
| Spatio Temporal             | Handling and analyzing spatio-temporal data         |
| Survival                    | Survival analysis                                   |
| Time Series                 | Time series analysis                                |
| Web Technologies            | Web technologies and service                        |

### B.6.3 Installed libraries and packages

Running the command `library(help="libraryname")` will display information about an installed package. Entries in the book that utilize packages include a line specifying how to access that library (e.g., `library(foreign)`). As of January 2014, the R distribution comes with the following packages:

**base** Base R functions

**compiler** R byte code compiler

**datasets** Base R datasets

**grDevices** Graphics devices for base and grid graphics

**graphics** R functions for base graphics

**grid** A rewrite of the graphics layout capabilities, plus some support for interaction

**methods** Formally defined methods and classes for R objects, plus other programming tools

**parallel** Support for parallel computation, including by forking and by sockets, and random-number generation

**splines** Regression spline functions and classes

**stats** R statistical functions

**stats4** Statistical functions using S4 classes

**tcltk** Interface and language bindings to Tcl/Tk GUI elements

**tools** Tools for package development and administration

**utils** R utility functions

These are available without having to run the `library()` command and are effectively part of R.

#### B.6.4 Packages referenced in this book

Other packages utilized in the book include:

**biglm** Bounded memory linear and generalized linear models [112]

**boot** Bootstrap functions [20] (recommended)

**BRugs** R interface to the OpenBUGS MCMC software [175]

**car** Companion to Applied Regression [47]

**chron** Chronological objects [83]

**circular** Circular statistics [3]

**coda** Output analysis and diagnostics for Markov Chain Monte Carlo simulations [132]

**coefplot** Plots coefficients from fitted models [92]

**coin** Conditional inference procedures in a permutation test framework [79]

**dispmod** Dispersion models [162]

**dBy** Groupwise summary statistics, LSmeans, and general linear contrasts [70]

**dplyr** Plyr specialized for dataframes: faster and with remote datastores [194]

**ellipse** Functions for drawing ellipses and ellipse-like confidence regions [122]

**elrm** Exact logistic regression via MCMC [205]

**epitools** Epidemiology tools [9]

**exactRankTests** Exact distributions for rank and permutation tests [78]

**flexmix** Flexible mixture modeling [98]

**foreach** Foreach looping construct for R [139]

**foreign** Read data stored by Minitab, S, SAS, SPSS, Stata, Systat, Weka, dBase [134] (recommended)

**gam** Generalized additive models [64]

**gdata** Various R programming tools for data manipulation [189]

**gee** Generalized estimation equation solver [21]

**GenKern** Functions for generating and manipulating binned kernel density estimates [109]

- GGally** Extension to ggplot2 [158]
- ggmap** A package for spatial visualization with Google Maps and OpenStreetMap [87]
- ggplot2** An implementation of the Grammar of Graphics [196]
- gmodels** Various R programming tools for model fitting [188]
- gridExtra** Functions for grid graphics [10]
- gtools** Various R programming tools [190]
- hexbin** Hexagonal binning routines [22]
- Hmisc** Harrell miscellaneous [62]
- Hotelling** Hotelling's T-squared test and variants [30]
- hwriter** HTML writer: outputs R objects in HTML format [129]
- irr** Various coefficients of interrater reliability and agreement [48]
- knitr** A general-purpose package for dynamic report generation in R [202]
- lars** Least angle regression, LASSO, and forward stagewise [65]
- lattice** Lattice graphics [151] (recommended)
- lawstat** An R package for biostatistics, public policy, and law [51]
- lme4** Linear mixed-effects models [12]
- lmtest** Testing linear regression models [206]
- logistiX** Exact logistic regression including Firth correction for binary covariates [66]
- lpSolve** Interface to Lp\_solve v. 5.5 to solve linear/integer programs [16]
- lubridate** Makes dealing with dates a little easier [57]
- maps** Draw geographical maps [15]
- markdown** Markdown rendering for R [6]
- MASS** Support functions and datasets for Venables and Ripley's MASS [185] (recommended)
- Matching** Multivariate and propensity score matching with balance optimization [164]
- Matrix** Sparse and dense matrix classes and methods [11] (recommended)
- MCMCpack** Markov chain Monte Carlo (MCMC) package [114]
- memisc** Tools for survey data, graphics, programming, statistics, and simulation [36]
- mice** Multivariate imputation by chained equations [184]
- mitools** Tools for multiple imputation of missing data [111]
- mix** Estimation/multiple imputation for mixed categorical and continuous data [155]
- moments** Moments, cumulants, skewness, kurtosis, and related tests [91]
- mosaic** Project MOSAIC statistics and mathematics teaching utilities [133]
- MplusAutomation** Automating Mplus model estimation and interpretation [60]
- muhaz** Hazard function estimation in survival analysis [68]
- multcomp** Simultaneous inference in general parametric models [77]
- multilevel** Multilevel functions [17]
- nlme** Linear and nonlinear mixed effects models [130] (recommended)
- nnet** Feed-forward neural networks and multinomial log-linear models [185] (recommended)

- nortest** Tests for normality [58]
- partykit** A toolkit for recursive partytioning [80]
- plotrix** Various plotting functions [99]
- plyr** Tools for splitting, applying, and combining data [198]
- poLCA** Polytomous variable latent class analysis [106]
- prettyR** Pretty descriptive stats [100]
- pscl** Political science computational laboratory, Stanford University [82]
- pwr** Basic functions for power analysis [23]
- QuantPsyc** Quantitative psychology tools [44]
- quantreg** Quantile regression [90]
- R2jags** A package for running jags from R [170]
- R2WinBUGS** Running WinBUGS and OpenBUGS from R [169]
- randomLCA** Random effects latent class analysis [14]
- RCurl** General network (HTTP/FTP) client interface for R [93]
- reshape** Flexibly reshape data [195]
- rjags** Bayesian graphical models using MCMC [131]
- RMongo** MongoDB client for R [24]
- rms** Regression modeling strategies [63]
- RMySQL** R interface to the MySQL database [84]
- ROCR** Visualizing the performance of scoring classifiers [168]
- RODBC** An ODBC database interface [140]
- rpart** Recursive partitioning [173] (recommended)
- RSQlite** SQLite interface for R [85]
- rtf** Rich text format (RTF) output [156]
- runjags** Interface utilities, parallel computing methods, and additional distributions for MCMC models in JAGS [33]
- sas7bdat** SAS database reader [166]
- scatterplot3d** 3D scatter plot [104]
- sciplot** Scientific graphing functions for factorial designs [120]
- simPH** Tools for simulating and plotting quantities of interest estimated from Cox proportional hazards models [49]
- sqldf** Perform SQL selects on R dataframes [59]
- survey** Analysis of complex survey samples [110]
- survival** Survival analysis [174] (recommended)
- tmvtnorm** Truncated multivariate normal and Student *t* distribution [199]
- vcd** Visualizing categorical data [118]
- VGAM** Vector generalized linear and additive models [204]
- vioplot** Violin plot [2]
- WriteXLS** Cross-platform Perl based R function to create Excel spreadsheets [160]

**XML** Tools for parsing and generating XML [94]

**xtable** Export tables to L<sup>A</sup>T<sub>E</sub>X or HTML [31]

**Zelig** Everyone's statistical software [128]

These must be downloaded, installed, and loaded prior to use (see `install.packages()`, `require()` and `library()`), though the recommended packages are included in most distributions of R. To facilitate the process of loading the other packages, we have created a script file to load these in one step (see B.2.1).

### B.6.5 Datasets available with R

A number of datasets are available within the `datasets` package. The `data()` function lists these, while the optional `package` option can be used to regenerate datasets from within a specific package.

## B.7 Support and bugs

Since R is a free software project written by volunteers, there are no paid support options available directly from the R Foundation. A number of groups provide commercial support for R and related systems, including Revolution Analytics and RStudio. In addition, extensive resources are available to help users.

In addition to the manuals, publications, FAQs, newsletter, task views, and books listed on the [www.r-project.org](http://www.r-project.org) web page, there are a number of mailing lists that exist to help answer questions. Because of the volume of postings, it is important to carefully read the posting guide at <http://www.r-project.org/posting-guide.html> prior to submitting a question. These guidelines are intended to help leverage the value of the list, to avoid embarrassment, and to optimize the allocation of limited resources to technical issues.

As in any general purpose statistical software package, some bugs exist. More information about the process of determining whether and how to report a problem can be found using `help(bug.report)` (please also review the R FAQ).



# Appendix C

## The HELP study dataset

### C.1 Background on the HELP study

Data from the HELP (Health Evaluation and Linkage to Primary Care) study are used to illustrate many of the entries in R and SAS. The HELP study was a clinical trial for adult inpatients recruited from a detoxification unit. Patients with no primary care physician were randomized to receive a multidisciplinary assessment and a brief motivational intervention or usual care, with the goal of linking them to primary medical care. Funding for the HELP study was provided by the National Institute on Alcohol Abuse and Alcoholism (R01-AA10870, Samet PI) and the National Institute on Drug Abuse (R01-DA10019, Samet PI).

Eligible subjects were adults, who spoke Spanish or English, reported alcohol, heroin, or cocaine as their first or second drug of choice, and either resided in proximity to the primary care clinic to which they would be referred, or were homeless. Patients with established primary care relationships they planned to continue, significant dementia, specific plans to leave the Boston area that would prevent research participation, failure to provide contact information for tracking purposes, or pregnancy were excluded.

Subjects were interviewed at baseline during their detoxification stay, and follow-up interviews were undertaken every 6 months for 2 years. A variety of continuous, count, discrete, and survival time predictors and outcomes were collected at each of these five occasions.

The details of the randomized trial along with the results from a series of additional analyses have been published [149, 138, 76, 103, 88, 148, 147, 165, 95, 201].

### C.2 Roadmap to analyses of the HELP dataset

Table C.1 summarizes the analyses illustrated using the HELP dataset. These analyses are intended to help illustrate the methods described in the book. Interested readers are encouraged to review the published data from the HELP study for substantive analyses.

Table C.1: Analyses undertaken using the HELP dataset

| Description             | section (page) |
|-------------------------|----------------|
| Data input and output   | 2.6.1 (p. 39)  |
| Summarize data contents | 2.6.1 (p. 39)  |
| Data display            | 2.6.2 (p. 43)  |

|  |                  |
|--|------------------|
| Derived variables and data manipulation  | 2.6.3 (p. 45)    |
| Sorting and subsetting                   | 2.6.4 (p. 51)    |
| Summary statistics                       | 5.7.1 (p. 97)    |
| Exploratory data analysis                | 5.7.1 (p. 97)    |
| Bivariate relationship                   | 5.7.2 (p. 101)   |
| Contingency tables                       | 5.7.3 (p. 103)   |
| Two-sample tests                         | 5.7.4 (p. 107)   |
| Survival analysis (logrank test)         | 5.7.5 (p. 112)   |
| Scatterplot with smooth fit              | 6.6.1 (p. 129)   |
| Linear regression with interaction       | 6.6.2 (p. 130)   |
| Regression diagnostics                   | 6.6.3 (p. 135)   |
| Fitting stratified regression models     | 6.6.4 (p. 138)   |
| Two-way analysis of variance (ANOVA)     | 6.6.5 (p. 139)   |
| Multiple comparisons                     | 6.6.6 (p. 144)   |
| Contrasts                                | 6.6.7 (p. 146)   |
| Logistic regression                      | 7.10.1 (p. 172)  |
| Poisson regression                       | 7.10.2 (p. 176)  |
| Zero-inflated Poisson regression         | 7.10.3 (p. 178)  |
| Negative binomial regression             | 7.10.4 (p. 180)  |
| Quantile regression                      | 7.10.5 (p. 181)  |
| Ordinal logit                            | 7.10.6 (p. 182)  |
| Multinomial logit                        | 7.10.7 (p. 183)  |
| Generalized additive model               | 7.10.8 (p. 185)  |
| Reshaping datasets                       | 7.10.9 (p. 187)  |
| General linear model for correlated data | 7.10.10 (p. 190) |
| Random effects model                     | 7.10.11 (p. 193) |
| Generalized estimating equations model   | 7.10.12 (p. 197) |
| Generalized linear mixed model           | 7.10.13 (p. 199) |
| Proportional hazards regression model    | 7.10.14 (p. 200) |
| Cronbach $\alpha$                        | 7.10.15 (p. 201) |
| Factor analysis                          | 7.10.16 (p. 202) |
| Recursive partitioning                   | 7.10.17 (p. 205) |
| Linear discriminant analysis             | 7.10.18 (p. 206) |
| Hierarchical clustering                  | 7.10.19 (p. 208) |
| Scatterplot with multiple y axes         | 8.7.1 (p. 230)   |
| Conditioning plot                        | 8.7.2 (p. 232)   |
| Scatterplot with marginal histogram      | 8.7.3 (p. 232)   |
| Kaplan–Meier plot                        | 8.7.4 (p. 234)   |
| ROC curve                                | 8.7.5 (p. 235)   |
| Pairs plot                               | 8.7.6 (p. 236)   |
| Visualize correlation matrix             | 8.7.7 (p. 238)   |
| By group processing                      | 11.1 (p. 283)    |
| Bayesian regression                      | 11.4.1 (p. 290)  |
| Propensity score modeling                | 11.4.2 (p. 296)  |
| Multiple imputation                      | 11.4.4 (p. 306)  |

### C.3 Detailed description of the dataset

The Institutional Review Board of Boston University Medical Center approved all aspects of the study, including the creation of the de-identified dataset. Additional privacy protection was secured by the issuance of a Certificate of Confidentiality by the Department of Health and Human Services.

A de-identified dataset containing the variables utilized in the end of chapter examples is available for download at the book web site:

<http://www.amherst.edu/~nhorton/sasr2/datasets/help.csv>.

Variables included in the HELP dataset are described in Table C.2. A full copy of the study instruments can be found at <http://www.amherst.edu/~nhorton/help>.

Table C.2: Annotated description of variables in the HELP dataset

| VARIABLE    | DESCRIPTION  | VALUES         | NOTE   |
|-------------|--|----------------|--|
| a15a        | number of nights in overnight shelter in past 6 months                               | 0–180          | see also <b>homeless</b>   |
| a15b        | number of nights on the street in past 6 months                                      | 0–180          | see also <b>homeless</b>   |
| age         | age at baseline (in years)   | 19–60          |  |
| anystatus   | use of any substance post-detox  | 0=no,<br>1=yes | see also <b>daysany</b>  |
| cesd*       | Center for Epidemiologic Studies Depression scale                                    | 0–60           | higher scores indicate more depressive symptoms; see also <b>f1a–f1t</b> |
| d1          | how many times hospitalized for medical problems (lifetime)                          | 0–100          |  |
| daysany     | time (in days) to first use of any substance post-detox                              | 0–268          | see also <b>anystatus</b>  |
| daysdrink   | time (in days) to first alcoholic drink post-detox                                   | 0–270          | see also <b>drinkstatus</b>  |
| dayslink    | time (in days) to linkage to primary care  | 0–456          | see also <b>linkstatus</b>   |
| drinkstatus | use of alcohol post-detox  | 0=no,<br>1=yes | see also <b>daysdrink</b>  |
| drugrisk*   | Risk-Assessment Battery (RAB) drug risk score  | 0–21           | higher scores indicate riskier behavior; see also <b>sexrisk</b>         |
| e2b*        | number of times in past 6 months entered a detox program                             | 1–21           |  |
| f1a         | I was bothered by things that usually don't bother me                                | 0–3#           |  |
| f1b         | I did not feel like eating; my appetite was poor                                     | 0–3#           |  |
| f1c         | I felt that I could not shake off the blues even with help from my family or friends | 0–3#           |  |
| f1d         | I felt that I was just as good as other people                                       | 0–3#           |  |

|                   |  |                     |  |
|-------------------|--|---------------------|--|
| <b>f1e</b>        | I had trouble keeping my mind on what I was doing                                | 0-3#                |  |
| <b>f1f</b>        | I felt depressed   | 0-3#                |  |
| <b>f1g</b>        | I felt that everything I did was an effort                                       | 0-3#                |  |
| <b>f1h</b>        | I felt hopeful about the future  | 0-3#                |  |
| <b>f1i</b>        | I thought my life had been a failure   | 0-3#                |  |
| <b>f1j</b>        | I felt fearful   | 0-3#                |  |
| <b>f1k</b>        | My sleep was restless  | 0-3#                |  |
| <b>f1l</b>        | I was happy  | 0-3#                |  |
| <b>f1m</b>        | I talked less than usual   | 0-3#                |  |
| <b>f1n</b>        | I felt lonely  | 0-3#                |  |
| <b>f1o</b>        | People were unfriendly   | 0-3#                |  |
| <b>f1p</b>        | I enjoyed life   | 0-3#                |  |
| <b>f1q</b>        | I had crying spells  | 0-3#                |  |
| <b>f1r</b>        | I felt sad   | 0-3#                |  |
| <b>f1s</b>        | I felt that people dislike me  | 0-3#                |  |
| <b>f1t</b>        | I could not get going  | 0-3#                |  |
| <b>female</b>     | gender of respondent   | 0=male,<br>1=female |  |
| <b>g1b*</b>       | experienced serious thoughts of suicide (last 30 days)                           | 0=no,<br>1=yes      |  |
| <b>homeless*</b>  | 1 or more nights on the street or shelter in past 6 months                       | 0=no,<br>1=yes      | see also <b>a15a</b> and <b>a15b</b>                           |
| <b>i1*</b>        | average number of drinks (standard units) consumed per day (in the past 30 days) | 0-142               | see also <b>i2</b>   |
| <b>i2</b>         | maximum number of drinks (standard units) consumed per day (in the past 30 days) | 0-184               | see also <b>i1</b>   |
| <b>id</b>         | random subject identifier  | 1-470               |  |
| <b>indtot*</b>    | Inventory of Drug Use Consequences (InDUC) total score                           | 4-45                |  |
| <b>linkstatus</b> | post-detox linkage to primary care   | 0=no,<br>1=yes      | see also <b>dayslink</b>                                       |
| <b>mcs*</b>       | SF-36 Mental Component Score   | 7-62                | higher scores indicate better functioning; see also <b>pcs</b> |
| <b>pcrec*</b>     | number of primary care visits in past 6 months                                   | 0-2                 | see also <b>linkstatus</b> , not observed at baseline          |
| <b>pcs*</b>       | SF-36 Physical Component Score   | 14-75               | higher scores indicate better functioning; see also <b>mcs</b> |
| <b>pss_fr</b>     | perceived social supports (friends)  | 0-14                |  |
| <b>satreat</b>    | any BSAS substance abuse treatment at baseline                                   | 0=no,<br>1=yes      |  |

|                  |   |                                      |   |
|------------------|---|--------------------------------------|---|
| <b>sexrisk*</b>  | Risk-Assessment Battery (RAB)<br>sex risk score | 0–21                                 | higher scores indicate<br>riskier behavior; see<br>also <b>drugrisk</b> |
| <b>substance</b> | primary substance of abuse                      | alcohol,<br>cocaine,<br>or heroin    |   |
| <b>treat</b>     | randomization group                             | 0=usual<br>care,<br>1=HELP<br>clinic |   |

Notes: Observed range is provided (at baseline) for continuous variables.

\* denotes variables measured at baseline and followup (e.g., **cesd** is baseline measure, **cesd1** is measured at 6 months, and **cesd4** is measured at 24 months).

#: For each of the 20 items in HELP section F1 (CESD), respondents were asked to indicate how often they behaved this way during the past week (0 = rarely or none of the time, less than 1 day; 1 = some or a little of the time, 1–2 days; 2 = occasionally or a moderate amount of time, 3–4 days; or 3 = most or all of the time, 5–7 days); items **f1d**, **f1h**, **f1l**, and **f1p** were reverse coded.



# References

- [1] D. Adams. *The Hitchhiker's Guide to the Galaxy*. Pan Books, 1979.
- [2] D. Adler. *vioplot: Violin plot*, 2005. R package version 0.2.
- [3] C. Agostinelli and U. Lund. *R package circular: Circular Statistics (version 0.4-7)*, 2013.
- [4] A. Agresti. *Categorical Data Analysis*. John Wiley & Sons, Hoboken, NJ, 2002.
- [5] J. Albert. *Bayesian Computation with R*. Springer, New York, 2008.
- [6] J. J. Allaire, J. Horner, V. Marti, and N. Porte. *markdown: Markdown rendering for R*, 2013. R package version 0.6.3.
- [7] P. D. Allison. *Survival Analysis Using SAS: A Practical Guide (second edition)*. SAS Institute, 2010.
- [8] D. G. Altman and J.M. Bland. Measurement in medicine: the analysis of method comparison studies. *The Statistician*, 32:307–317, 1983.
- [9] T. J. Aragon. *epitools: Epidemiology Tools*, 2012. R package version 0.5-7.
- [10] B. Auguie. *gridExtra: Functions in Grid Graphics*, 2012. R package version 0.9.1.
- [11] D. Bates and M. Maechler. *Matrix: Sparse and Dense Matrix Classes and Methods*, 2013. R package version 1.1-0.
- [12] D. Bates, M. Maechler, B. Bolker, and S. Walker. *lme4: Linear Mixed-Effects Models Using Eigen and S4*, 2013. R package version 1.0-5.
- [13] B. Baumer, M. Çetinkaya Rundel, A. Bray, L. Loi, and N.J. Horton. R markdown: Integrating a reproducible analysis tool into introductory statistics. *Technology Innovations in Statistics Education*, 8(1), 2014.
- [14] K. Beath. *randomLCA: Random Effects Latent Class Analysis*, 2013. R package version 0.8-7.
- [15] R. A. Becker, A. R. Wilks, R. Brownrigg, and T. P. Minka. *maps: Draw Geographical Maps*, 2013. R package version 2.3-6.
- [16] M. Berkelaar. *lpSolve: Interface to Lp\_solve v. 5.5 to Solve Linear/Integer Programs*, 2013. R package version 5.6.7.
- [17] P. Bliese. *multilevel: Multilevel Functions*, 2013. R package version 2.5.

- [18] A. H. Bowker. Bowker's test for symmetry. *Journal of the American Statistical Association*, 43:572–574, 1948.
- [19] T. S. Breusch and A. R. Pagan. A simple test for heteroscedasticity and random coefficient variation. *Econometrica*, 47, 1979.
- [20] A. Canty and B. Ripley. *boot: Bootstrap R (S-Plus) Functions*, 2013. R package version 1.3-9.
- [21] V. J. Carey. *gee: Generalized Estimation Equation Solver*, 2012. R package version 4.13-18.
- [22] D. Carr, N. Lewin-Koh, and M. Maechler. *hexbin: Hexagonal Binning Routines*, 2013. R package version 1.26.3.
- [23] S. Champely. *pwr: Basic Functions for Power Analysis*, 2012. R package version 1.1.1.
- [24] T. Chheng. *RMongo: MongoDB Client for R*, 2013. R package version 0.0.25.
- [25] R. P. Cody and J. K. Smith. *Applied Statistics and the SAS Programming Language*. Prentice Hall, 1997.
- [26] D. Collett. *Modelling Binary Data*. Chapman & Hall, London, 1991.
- [27] D. Collett. *Modeling Survival Data in Medical Research (second edition)*. CRC Press, Boca Raton, FL, 2003.
- [28] L. M. Collins, J. L. Schafer, and C.-M. Kam. A comparison of inclusive and restrictive strategies in modern missing data procedures. *Psychological Methods*, 6(4):330–351, 2001.
- [29] R. D. Cook. *Residuals and Influence in Regression*. Chapman & Hall, London, 1982.
- [30] J. M. Curran. *Hotelling's T-squared Test and Variants*, 2013. R package version 1.0-2.
- [31] D. B. Dahl. *xtable: Export Tables to LaTeX or HTML*, 2013. R package version 1.7-1.
- [32] L. D. Delwiche and S. J. Slaughter. *The Little SAS Book: A Primer (third edition)*. SAS Publishing, 2003.
- [33] M. J. Denwood. runjags: An R package providing interface utilities, parallel computing methods and additional distributions for MCMC models in JAGS. *Journal of Statistical Software*, in review.
- [34] A. J. Dobson and A. Barnett. *An Introduction to Generalized Linear Models (third edition)*. CRC Press, Boca Raton, FL, 2008.
- [35] B. Efron and R. J. Tibshirani. *An Introduction to the Bootstrap*. Chapman & Hall, London, 1993.
- [36] M. Elff. *memisc: Tools for Management of Survey Data, Graphics, Programming, Statistics, and Simulation*, 2013. R package version 0.96-9.
- [37] M. J. Evans and J. S. Rosenthal. *Probability and Statistics: the Science of Uncertainty*. W H Freeman and Company, New York, 2004.
- [38] J. J. Faraway. *Linear Models with R*. CRC Press, Boca Raton, FL, 2004.

- [39] J. J. Faraway. *Extending the Linear model with R: Generalized Linear, Mixed Effects and Nonparametric Regression Models*. CRC Press, Boca Raton, FL, 2005.
- [40] N. I. Fisher. *Statistical Analysis of Circular Data*. Cambridge University Press, 1996.
- [41] G. S. Fishman and L. R. Moore. A statistical evaluation of multiplicative congruential generators with modulus  $(2^{31} - 1)$ . *Journal of the American Statistical Association*, 77:29–136, 1982.
- [42] G. M. Fitzmaurice, N. M. Laird, and J. H. Ware. *Applied Longitudinal Analysis*. John Wiley & Sons, Hoboken, NJ, 2004.
- [43] T. R. Fleming and D. P. Harrington. *Counting Processes and Survival Analysis*. John Wiley & Sons, Hoboken, NJ, 1991.
- [44] T. D. Fletcher. *QuantPsyc: Quantitative Psychology Tools*, 2012. R package version 1.5.
- [45] J. Fox. The R Commander: a basic graphical user interface to R. *Journal of Statistical Software*, 14(9), 2005.
- [46] J. Fox. Aspects of the social organization and trajectory of the R Project. *The R Journal*, 1(2):5–13, December 2009.
- [47] John Fox and Sanford Weisberg. *An R Companion to Applied Regression (second edition)*. Sage, Thousand Oaks, CA, 2011.
- [48] M. Gamer, J. Lemon, I. Fellows, and P. Singh. *irr: Various Coefficients of Interrater Reliability and Agreement*, 2012. R package version 0.84.
- [49] C. Gandrud. *simPH: Tools for Simulating and Plotting Quantities of Interest Estimated from Cox Proportional Hazards Models*, 2013. R package version 0.8.5.
- [50] C. Gandrud. *Reproducible Research with R and RStudio*. CRC Press, Boca Raton, FL, 2014.
- [51] J. L. Gastwirth, Y. R. Gel, W. L. Wallace Hui, V. Lyubchich, W. Miao, and K. Noguchi. *lawstat: An R Package for Biostatistics, Public Policy, and Law*, 2013. R package version 2.4.1.
- [52] A. Gelman, J. B. Carlin, H. S. Stern, and D. B. Rubin. *Bayesian Data Analysis (second edition)*. Chapman & Hall, London, 2004.
- [53] R. Gentleman and D. Temple Lang. Statistical analyses and reproducible research. *Journal of Computational and Graphical Statistics*, 16(1):1–23, 2007.
- [54] L. Gonick. *Cartoon Guide to Statistics*. HarperPerennial, New York, 1993.
- [55] P. I. Good. *Permutation Tests: A Practical Guide to Resampling Methods for Testing Hypotheses*. Springer-Verlag, New York, 1994.
- [56] Google. R style guide. <http://google-styleguide.googlecode.com/svn/trunk/Rguide.xml>, date accessed 10/29/2013, 2013.
- [57] G. Grolemund and H. Wickham. Dates and times made easy with lubridate. *Journal of Statistical Software*, 40(3):1–25, 2011.
- [58] J. Gross and U. Ligges. *nortest: Tests for Normality*, 2012. R package version 1.0-2.

- [59] G. Grothendieck. *sqldf: Perform SQL Selects on R Data Frames*, 2012. R package version 0.4-6.4.
- [60] M. Hallquist and J. Wiley. *MplusAutomation: Automating Mplus Model Estimation and Interpretation*, 2013. R package version 0.6-2.
- [61] J. W. Hardin and J. M. Hilbe. *Generalized Estimating Equations*. CRC Press, Boca Raton, FL, 2002.
- [62] F. E. Harrell. *Hmisc: Harrell Miscellaneous*, 2013. R package version 3.13-0.
- [63] F. E. Harrell. *rms: Regression Modeling Strategies*, 2013. R package version 4.1-0.
- [64] T. Hastie. *gam: Generalized Additive Models*, 2013. R package version 1.09.
- [65] T. Hastie and B. Efron. *lars: Least Angle Regression, Lasso and Forward Stagewise*, 2013. R package version 1.2.
- [66] G. Heinze and T. Ladner. *logistix: Exact logistic regression including Firth correction*, 2013. R package version 1.0-1.
- [67] D. F. Heitjan and R. J. A. Little. Multiple imputation for the Fatal Accident Reporting System. *Applied Statistics*, 40:13–29, 1991.
- [68] K. Hess and R. Gentleman. *muhaz: Hazard Function Estimation in Survival Analysis*, 2010. R package version 1.2.5.
- [69] T. C. Hesterberg, D. S. Moore, S. Monaghan, A. Clipson, and R. Epstein. *Bootstrap Methods and Permutation Tests*. W.C. Freeman, 2005.
- [70] S. Højsgaard and U. Halekoh. *doBy: Groupwise Summary Statistics, LSmeans, General Linear Contrasts, Various Utilities*, 2013. R package version 4.5-10.
- [71] N. J. Horton. I hear, I forget. I do, I understand: A modified Moore-method mathematical statistics course. *The American Statistician*, 67(3):219–228, 2013.
- [72] N. J. Horton, E. R. Brown, and L. Qian. Use of R as a toolbox for mathematical statistics exploration. *The American Statistician*, 58(4):343–357, 2004.
- [73] N. J. Horton, E. Kim, and R. Saitz. A cautionary note regarding count models of alcohol consumption in randomized controlled trials. *BMC Medical Research Methodology*, 7(9), 2007.
- [74] N. J. Horton and K. P. Kleinman. Much ado about nothing: A comparison of missing data methods and software to fit incomplete data regression models. *The American Statistician*, 61:79–90, 2007.
- [75] N. J. Horton and S. R. Lipsitz. Multiple imputation in practice: comparison of software packages for regression models with missing variables. *The American Statistician*, 55(3):244–254, 2001.
- [76] N. J. Horton, R. Saitz, N. M. Laird, and J. H. Samet. A method for modeling utilization data from multiple sources: Application in a study of linkage to primary care. *Health Services and Outcomes Research Methodology*, 3:211–223, 2002.
- [77] T. Hothorn, F. Bretz, and P. Westfall. Simultaneous inference in general parametric models. *Biometrical Journal*, 50(3):346–363, 2008.

- [78] T. Hothorn and K. Hornik. *exactRankTests: Exact Distributions for Rank and Permutation Tests*, 2013. R package version 0.8-27.
- [79] T. Hothorn, K. Hornik, M. A. van de Wiel, and A. Zeileis. Implementing a class of permutation tests: The coin package. *Journal of Statistical Software*, 28(8):1–23, 2008.
- [80] T. Hothorn and A. Zeileis. *partykit: A Toolkit for Recursive Partytioning*, 2013. R package version 0.1-6.
- [81] R. Ihaka and R. Gentleman. R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics*, 5(3):299–314, 1996.
- [82] S. Jackman. *pscl: Classes and Methods for R Developed in the Political Science Computational Laboratory, Stanford University*, 2012. R package version 1.04.4.
- [83] D. James and K. Hornik. *chron: Chronological Objects Which Can Handle Dates and Times*, 2013. R package version 2.3-44. S original by David James, R port by Kurt Hornik.
- [84] D. A. James and S. DebRoy. *RMySQL: R Interface to the MySQL Database*, 2012. R package version 0.9-3.
- [85] D. A. James and S. Falcon. *RSQLite: SQLite Interface for R*, 2013. R package version 0.11.4.
- [86] S. R. Jammalamadaka and A. Sengupta. *Topics in Circular Statistics*. World Scientific, 2001.
- [87] D. Kahle and H. Wickham. *ggmap: A Package for Spatial Visualization with Google Maps and OpenStreetMap*, 2013. R package version 2.3.
- [88] S. G. Kertesz, N. J. Horton, P. D. Friedmann, R. Saitz, and J. H. Samet. Slowing the revolving door: Stabilization programs reduce homeless persons substance use after detoxification. *Journal of Substance Abuse Treatment*, 24:197–207, 2003.
- [89] D. Knuth. Literate programming. *CSLI Lecture Notes*, 27, 1992.
- [90] R. Koenker. *quantreg: Quantile Regression*, 2013. R package version 5.05.
- [91] L. Komsta and F. Novomestky. *moments: Moments, Cumulants, Skewness, Kurtosis and Related Tests*, 2012. R package version 0.13.
- [92] J. P. Landers. *coefplot: Plots Coefficients from Fitted Models*, 2013. R package version 1.2.0.
- [93] D. Temple Lang. *RCurl: General Network (HTTP/FTP/...) Client Interface for R*, 2013. R package version 1.95-4.1.
- [94] D. Temple Lang. *XML: Tools for Parsing and Generating XML within R and S-Plus*, 2013. R package version 3.95-0.2.
- [95] M. J. Larson, R. Saitz, N. J. Horton, C. Lloyd-Travaglini, and J. H. Samet. Emergency department and hospital utilization among alcohol and drug-dependent detoxification patients without primary medical care. *American Journal of Drug and Alcohol Abuse*, 32:435–452, 2006.

- [96] M. Lavine. *Introduction to Statistical Thought*. <http://www.math.umass.edu/~lavine/Book/book.html>, 2005.
- [97] F. Leisch. Sweave: Dynamic generation of statistical reports using literate data analysis. In Wolfgang Härdle and Bernd Rönz, editors, *Compstat 2002 — Proceedings in Computational Statistics*, pages 575–580. Physica Verlag, Heidelberg, 2002.
- [98] F. Leisch. FlexMix: A general framework for finite mixture models and latent class regression in R. *Journal of Statistical Software*, 11(8):1–18, 2004.
- [99] J. Lemon. Plotrix: a package in the red light district of R. *R-News*, 6(4):8–12, 2006.
- [100] J. Lemon and P. Grosjean. *prettyR: Pretty Descriptive Stats*, 2013. R package version 2.0-7.
- [101] R. Lenth and S. Højsgaard. Reproducible statistical analysis with multiple languages. *Computational Statistics*, 26(3):419–426, 2011.
- [102] K.-Y. Liang and S. L. Zeger. Longitudinal data analysis using generalized linear models. *Biometrika*, 73:13–22, 1986.
- [103] J. Liebschutz, J. B. Savetsky, R. Saitz, N. J. Horton, C. Lloyd-Travaglini, and J. H. Samet. The relationship between sexual and physical abuse and substance abuse consequences. *Journal of Substance Abuse Treatment*, 22(3):121–128, 2002.
- [104] U. Ligges and M. Mächler. Scatterplot3d: an R package for visualizing multivariate data. *Journal of Statistical Software*, 8(11):1–20, 2003.
- [105] D. Y. Lin, L. J. Wei, and Z. Ying. Checking the Cox model with cumulative sums of martingale-based residuals. *Biometrika*, 80:557–572, 1993.
- [106] D. A. Linzer and J. B. Lewis. poLCA: An R package for polytomous variable latent class analysis. *Journal of Statistical Software*, 42(10):1–29, 2011.
- [107] S. R. Lipsitz, N. M. Laird, and D. P. Harrington. Maximum likelihood regression methods for paired binary data. *Statistics in Medicine*, 9:1517–1525, 1990.
- [108] R. Littell, W. W. Stroup, and R. Freund. *SAS For Linear Models (fourth edition)*. SAS Publishing, 2002.
- [109] D. Lucy and R. Aykroyd. *GenKern: Functions for Generating and Manipulating Binned Kernel Density Estimates*, 2013. R package version 1.2-60.
- [110] T. Lumley. Analysis of complex survey samples. *Journal of Statistical Software*, 9(1):1–19, 2004.
- [111] T. Lumley. *mitools: Tools for Multiple Imputation of Missing Data*, 2012. R package version 2.2.
- [112] T. Lumley. *biglm: Bounded Memory Linear and Generalized Linear Models*, 2013. R package version 0.9-1.
- [113] B. F. J. Manly. *Multivariate Statistical Methods: A Primer (third edition)*. CRC Press, Boca Raton, FL, 2004.
- [114] A. D. Martin, K. M. Quinn, and J. H. Park. MCMCpack: Markov Chain Monte Carlo in R. *Journal of Statistical Software*, 42(9):22, 2011.

- [115] M. Matsumoto and T. Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8:8–30, 1998.
- [116] P. McCullagh and J. A. Nelder. *Generalized Linear Models*. Chapman & Hall, London, 1989.
- [117] N. Metropolis, A.W. Rosenbluth, A.H. Teller, and E. Teller. Equations of state calculations by fast computing machines. *Journal of Chemical Physics*, 21(6):1087–1092, 1953.
- [118] D. Meyer, A Zeileis, and Kurt Hornik. The strucplot framework: Visualizing multi-way contingency tables with vcd. *Journal of Statistical Software*, 17(3):1–48, 2006.
- [119] J. D. Mills. Using computer simulation methods to teach statistics: A review of the literature. *Journal of Statistics Education*, 10(1), 2002.
- [120] M. Morales. *sciplot: Scientific Graphing Functions for Factorial Designs*, 2012. R package version 1.1-0.
- [121] F. Mosteller. *Fifty Challenging Problems in Probability with Solutions*. Dover Publications, 1987.
- [122] D. Murdoch and E. D. Chow. *ellipse: Functions for Drawing Ellipses and Ellipse-Like Confidence Regions*, 2013. R package version 0.3-8.
- [123] P. Murrell. *R Graphics*. Chapman & Hall, London, 2005.
- [124] P. Murrell. *Introduction to Data Technologies*. Chapman & Hall, London, 2009.
- [125] N. J. D. Nagelkerke. A note on a general definition of the coefficient of determination. *Biometrika*, 78(3):691–692, 1991.
- [126] National Institutes of Alcohol Abuse and Alcoholism, Bethesda, MD. *Helping Patients Who Drink Too Much*, 2005.
- [127] D. Nolan and D. Temple Lang. *XML and Web Technologies for Data Sciences with R*. Springer, New York, 2014.
- [128] M. Owen, K. Imai, G. King, and O. Lau. *Zelig: Everyone’s Statistical Software*, 2013. R package version 4.2-1.
- [129] G. Pau. *hwriter: HTML Writer: Outputs R Objects in HTML Format*, 2010. R package version 1.3.
- [130] J. Pinheiro, D. Bates, S. DebRoy, and D. Sarkar. *nlme: Linear and Nonlinear Mixed Effects Models*, 2013. R package version 3.1-113.
- [131] M. Plummer. *rjags: Bayesian Graphical Models Using MCMC*, 2013. R package version 3-11.
- [132] M. Plummer, N. Best, K. Cowles, and K. Vines. Coda: Convergence diagnosis and output analysis for MCMC. *R News*, 6(1):7–11, 2006.
- [133] R. Pruim, D. Kaplan, and N. J. Horton. *mosaic: Project MOSAIC (mosaic-web.org) Statistics and Mathematics Teaching Utilities*, 2014. R package version 0.8-18.

- [134] R Core Team. *foreign: Read Data Stored by Minitab, S, SAS, SPSS, Stata, Systat, Weka, dBase, ...*, 2013. R package version 0.8-57.
- [135] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, 2013.
- [136] T. E. Raghunathan, J. M. Lepkowski, J. van Hoewyk, and P. Solenberger. A multivariate technique for multiply imputing missing values using a sequence of regression models. *Survey Methodology*, 27(1):85–95, 2001.
- [137] T. E. Raghunathan, P. W. Solenberger, and J. V. Hoewyk. IVEware: imputation and variance estimation software. <http://www.isr.umich.edu/src/smp/ive>, accessed October 29, 2013, 2013.
- [138] V. W. Rees, R. Saitz, N. J. Horton, and J. H. Samet. Association of alcohol consumption with HIV sex and drug risk behaviors among drug users. *Journal of Substance Abuse Treatment*, 21(3):129–134, 2001.
- [139] Revolution Analytics and S. Weston. *foreach: Foreach Looping Construct for R*, 2013. R package version 1.4.1.
- [140] B. Ripley and M. Lapsley. *RODBC: ODBC Database Access*, 2013. R package version 1.3-10.
- [141] B. D. Ripley. Using databases with R. *R News*, 1(1):18–20, 2001.
- [142] M. L. Rizzo. *Statistical Computing with R*. CRC Press, Boca Raton, FL, 2007.
- [143] J. P. Romano and A. F. Siegel. *Counterexamples in Probability and Statistics*. Duxbury Press, 1986.
- [144] P. R. Rosenbaum and D. B. Rubin. Reducing bias in observational studies using subclassification on the propensity score. *Journal of the American Statistical Association*, 79:516–524, 1984.
- [145] P. R. Rosenbaum and D. B. Rubin. Constructing a control group using multivariate matched sampling methods that incorporate the propensity score. *The American Statistician*, 39:33–38, 1985.
- [146] D. B. Rubin. Multiple imputation after 18+ years. *Journal of the American Statistical Association*, 91:473–489, 1996.
- [147] R. Saitz, N. J. Horton, M. J. Larson, M. Winter, and J. H. Samet. Primary medical care and reductions in addiction severity: a prospective cohort study. *Addiction*, 100(1):70–78, 2005.
- [148] R. Saitz, M. J. Larson, N. J. Horton, M. Winter, and J. H. Samet. Linkage with primary medical care in a prospective cohort of adults with addictions in inpatient detoxification: Room for improvement. *Health Services Research*, 39(3):587–606, 2004.
- [149] J. H. Samet, M. J. Larson, N. J. Horton, K. Doyle, M. Winter, and R. Saitz. Linking alcohol and drug dependent adults to primary medical care: A randomized controlled trial of a multidisciplinary health intervention in a detoxification unit. *Addiction*, 98(4):509–516, 2003.
- [150] J.-M. Sarabia, E. Castillo, and D. J. Slottje. An ordered family of Lorenz curves. *Journal of Econometrics*, 91:43–60, 1999.

- [151] D. Sarkar. *Lattice: Multivariate Data Visualization with R*. Springer, New York, 2008.
- [152] C.-E. Särndal, B. Swensson, and J. Wretman. *Model Assisted Survey Sampling*. Springer-Verlag, New York, 1992.
- [153] SAS Institute. *SAS/STAT Software: Changes and Enhancements, Release 9.4*, 2013.
- [154] J. L. Schafer. *Analysis of Incomplete Multivariate Data*. Chapman & Hall, London, 1997.
- [155] J. L. Schafer. *mix: Estimation/Multiple Imputation for Mixed Categorical and Continuous Data*, 2010. R package version 1.0-8.
- [156] M. E. Schaffer. *rtf: Rich Text Format Output*, 2013. R package version 0.4-11.
- [157] N. Schenker and J. M. G. Taylor. Partially parametric techniques for multiple imputation. *Computational Statistics and Data Analysis*, 22(4):425–446, 1996.
- [158] B. Schloerke, J. Crowley, D. Cook, H. Hofmann, H. Wickham, F. Briatte, and M. Marbach. *GGally: Extension to ggplot2*, 2013. R package version 0.4.4.
- [159] D. Schoenfeld. Residuals for the proportional hazards regression model. *Biometrika*, 69:239–241, 1982.
- [160] M. Schwartz. *WriteXLS: Cross-Platform Perl Based R Function to Create Excel 2003 (XLS) and Excel 2007 (XLSX) Files*, 2013. R package version 3.2.2.
- [161] R. L. Schwartz, b. d. foy, and T. Phoenix. *Learning Perl (sixth edition)*. O'Reilly and Associates, 2011.
- [162] L. Scrucca. *dispmmod: Dispersion Models*, 2012. R package version 1.1.
- [163] G. A. F. Seber and C. J. Wild. *Nonlinear Regression*. John Wiley & Sons, Hoboken, NJ, 1989.
- [164] J. S. Sekhon. Multivariate and propensity score matching software with automated balance optimization: The Matching package for R. *Journal of Statistical Software*, 42(7):1–52, 2011.
- [165] C. W. Shanahan, A. Lincoln, N. J. Horton, R. Saitz, M. J. Larson, and J. H. Samet. Relationship of depressive symptoms and mental health functioning to repeat detoxification. *Journal of Substance Abuse Treatment*, 29:117–123, 2005.
- [166] M. S. Shotwell. *sas7bdat: SAS Database Reader*, 2012. R package version 0.3.
- [167] T. Sing, O. Sander, N. Beerenwinkel, and T. Lengauer. ROCR: visualizing classifier performance in R. *Bioinformatics*, 21(20):3940–3941, 2005.
- [168] T. Sing, O. Sander, N. Beerenwinkel, and T. Lengauer. ROCR: visualizing classifier performance in R. *Bioinformatics*, 21(20):7881, 2005.
- [169] S. Sturtz, U. Ligges, and A. Gelman. R2WinBUGS: A package for running WinBUGS from R. *Journal of Statistical Software*, 12(3):1–16, 2005.
- [170] Y.-S. Su and M. Yajima. *R2jags: A Package for Running jags from R*, 2013. R package version 0.03-11.

- [171] B. G. Tabachnick and L. S. Fidell. *Using Multivariate Statistics (fifth edition)*. Allyn & Bacon, 2007.
- [172] S. M. M. Tahaghoghi and H. E. Williams. *Learning MySQL*. O'Reilly Media: Sebastopol, CA, 2006.
- [173] T. Therneau, B. Atkinson, and B. Ripley. *rpart: Recursive Partitioning*, 2013. R package version 4.1-4.
- [174] T. M. Therneau and P. M. Grambsch. *Modeling Survival Data: Extending the Cox Model*. Springer, New York, 2000.
- [175] A. Thomas, B. O'Hara, U. Ligges, and S. Sturtz. Making BUGS open. *R News*, 6(1):12–17, 2006.
- [176] R. Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society B*, 58(1), 1996.
- [177] E. R. Tufte. *Envisioning Information*. Graphics Press, Cheshire, CT, 1990.
- [178] E. R. Tufte. *Visual Explanations: Images and Quantities, Evidence and Narrative*. Graphics Press, Cheshire, CT, 1997.
- [179] E. R. Tufte. *Visual Display of Quantitative Information (second edition)*. Graphics Press, Cheshire, CT, 2001.
- [180] E. R. Tufte. *Beautiful Evidence*. Graphics Press, Cheshire, CT, 2006.
- [181] J. W. Tukey. *Exploratory Data Analysis*. Addison Wesley, 1977.
- [182] S. van Buuren. *Flexible Imputation of Missing Data*. CRC Press, Boca Raton, FL, 2012.
- [183] S. van Buuren, H. C. Boshuizen, and D. L. Knook. Multiple imputation of missing blood pressure covariates in survival analysis. *Statistics in Medicine*, 18:681–694, 1999.
- [184] S. van Buuren and K. Groothuis-Oudshoorn. mice: Multivariate imputation by chained equations in R. *Journal of Statistical Software*, 45(3):1–67, 2011.
- [185] W. N. Venables and B. D. Ripley. *Modern Applied Statistics with S (fourth edition)*. Springer, New York, 2002.
- [186] W. N. Venables, D. M. Smith, and the R Core Team. An introduction to R: Notes on R: A programming environment for data analysis and graphics, version 3.0.2. <http://cran.r-project.org/doc/manuals/R-intro.pdf>, accessed October 27, 2013, 2013.
- [187] J. Verzani. *Using R For Introductory Statistics*. CRC Press, Boca Raton, FL, 2005.
- [188] G. R. Warnes. *gmodels: Various R Programming Tools for Model Fitting*, 2013. R package version 2.15.4.1.
- [189] G. R. Warnes, B. Bolker, G. Gorjanc, G. Grothendieck, A. Korosec, T. Lumley, D. MacQueen, A. Magnusson, and J. Rogers. *gdata: Various R Programming Tools for Data Manipulation*, 2013. R package version 2.13.2.
- [190] G. R. Warnes, B. Bolker, and T. Lumley. *gtools: Various R Programming Tools*, 2013. R package version 3.1.1.

- [191] B. West, K. B. Welch, and A. T. Galecki. *Linear Mixed Models: A Practical Guide Using Statistical Software*. CRC Press, Boca Raton, FL, 2006.
- [192] H. White. A heteroskedasticity-consistent covariance matrix estimator and a direct test for heteroskedasticity. *Econometrica*, 48:817–838, 1980.
- [193] I. R. White and P. Royston. Imputing missing covariate values for the Cox model. *Statistics in Medicine*, 28:1982–1998, 2009.
- [194] H. Wickham. *Plyr specialised for data frames: faster and with remote datastores*. In process.
- [195] H. Wickham. Reshaping data with the reshape package. *Journal of Statistical Software*, 21(12), 2007.
- [196] H. Wickham. *ggplot2: Elegant Graphics for Data Analysis*. Springer, New York, 2009.
- [197] H. Wickham. ASA 2009 data expo. *Journal of Computational and Graphical Statistics*, 20(2):281–283, 2011.
- [198] H. Wickham. The Split-Apply-Combine strategy for data analysis. *Journal of Statistical Software*, 40(1):1–29, 2011.
- [199] S. Wilhelm and B. G. Manjunath. *tmvtnorm: Truncated Multivariate Normal and Student t Distribution*, 2013. R package version 1.4-8.
- [200] L. Wilkinson. Dot plots. *The American Statistician*, 53(3):276–281, 1999.
- [201] J. D. Wines, R. Saitz, N. J. Horton, C. Lloyd-Travaglini, and J. H. Samet. Overdose after detoxification: a prospective study. *Drug and Alcohol Dependence*, 89:161–169, 2007.
- [202] Y. Xie. *knitr: A General-Purpose Package for Dynamic Report Generation in R*, 2013. R package version 1.5.
- [203] Y. Xie. *Dynamic Documents with R and knitr*. CRC Press, Boca Raton, FL, 2014.
- [204] T. W. Yee. The VGAM package for categorical data analysis. *Journal of Statistical Software*, 32(10):1–34, 2010.
- [205] D. Zamar, B. McNeney, and J. Graham. elrm: Software implementing exact-like inference for logistic regression models. *Journal of Statistical Software*, 21(3), 2007.
- [206] A. Zeileis and T. Hothorn. Diagnostic checking in regression relationships. *R News*, 2(3):7–10, 2002.



Retaining the same accessible format as the popular first edition, **SAS and R: Data Management, Statistical Analysis, and Graphics, Second Edition** explains how to easily perform an analytical task in both SAS and R, without having to navigate through the extensive, idiosyncratic, and sometimes unwieldy software documentation. The book covers many common tasks, such as data management, descriptive summaries, inferential procedures, regression analysis, and graphics, along with more complex applications.

This edition now covers RStudio, a powerful and easy-to-use interface for R. It incorporates a number of additional topics, including application program interfaces (APIs), database management systems, reproducible analysis tools, Markov chain Monte Carlo (MCMC) methods, and finite mixture models. It also includes extended examples of simulations and many new examples.

Through the extensive indexing and cross-referencing, users can directly find and implement the material they need. SAS users can look up tasks in the SAS index and then find the associated R code while R users can benefit from the R index in a similar manner. Numerous example analyses demonstrate the code in action and facilitate further exploration.

## Features

- Presents parallel examples in SAS and R to demonstrate how to use the software and derive identical answers regardless of software choice
- Takes users through the process of statistical coding from beginning to end
- Contains worked examples of basic and complex tasks, offering solutions to stumbling blocks often encountered by new users
- Includes an index for each software, allowing users to easily locate procedures
- Shows how RStudio can be used as a powerful, straightforward interface for R
- Covers APIs, reproducible analysis, database management systems, MCMC methods, and finite mixture models
- Incorporates extensive examples of simulations
- Provides the SAS and R example code, datasets, and more online



**CRC Press**  
Taylor & Francis Group  
an informa business  
[www.crcpress.com](http://www.crcpress.com)

6000 Broken Sound Parkway, NW  
Suite 300, Boca Raton, FL 33487  
711 Third Avenue  
New York, NY 10017  
2 Park Square, Milton Park  
Abingdon, Oxon OX14 4RN, UK

K19040

ISBN: 978-1-4665-8449-5  
9 781466 584495

90000