

Data Mining

Desktop Survival Guide

Togaware Watermark For Data Mining Survival

Copyright © 2006-2008 Graham Williams

Togaware Series of Open Source Desktop Survival Guides

This innovative series presents open source and freely available software tools and techniques with a focus on the tasks they are built for, ranging from working with the GNU/Linux operating system, through common desktop productivity tools, to sophisticated data mining applications. Each volume aims to be self contained, and slim, presenting the information in an easy to follow format without overwhelming the reader with details.

Togaware Watermark For Data Mining Survival

Series Titles

Data Mining with Rattle
R for the Data Miner
Text Mining with Rattle
Debian GNU/Linux
OpenMoko and the Neo 1973

Data Mining

Desktop Survival Guide

Graham Williams

Togaware.com

Togaware Watermark For Data Mining Survival

The procedures and applications presented in this book have been included for their instructional value. They have been tested but are not guaranteed for any particular purpose. Neither the publisher nor the author offer any warranties or representations, nor do they accept any liabilities with respect to the programs and applications.

The book, as you see it presently, is a work in progress, and different sections are progressed depending on feedback. Please send comments, suggestions, updates, and criticisms to Graham.Williams@togaware.com.

I hope you find it useful!

Togaware Watermark For Data Mining Survival

Printed 18th January 2008

Copyright © 2006-2007 by Graham Williams

ISBN 0-9757109-2-3

Togaware Watermark For Data Mining Survival

*Where knowledge is power, data is the fuel and data mining
the engine room for delivering that knowledge.*

Togaware Watermark For Data Mining Survival

Dedication

Togaware Watermark For Data Mining Survival

Togaware Watermark For Data Mining Survival

Contents

I Data Mining with Rattle	1
1 Introduction	3
1.1 Data Mining	4
1.2 Types of Analysis	4
1.3 Data Mining Applications	4
1.4 A Framework for Modelling	4
1.5 Agile Data Mining	5
2 Rattle Data Miner	7
2.1 Installing GTK, R, and Rattle	8
2.1.1 Quick Start Install	9
2.1.2 Installation Details	10
2.2 The Initial Interface	15
2.3 Interacting with Rattle	16
2.4 Menus and Buttons	18
2.4.1 Project Menu and Buttons	19
2.4.2 Edit Menu	19

2.4.3 Tools Menu and Toolbar	19
2.4.4 Settings	20
2.4.5 Help	20
2.5 Paradigms	20
2.6 Interacting with Plots	23
2.7 Summary	24
3 Sourcing Data	25
3.1 Nomenclature	25
3.2 Loading Data	26
3.3 CSV Data	27
3.4 ARFF Data	32
3.5 ODBC Sourced Data	35
3.6 R Data	37
3.7 R Dataset	37
3.8 Data Entry	39
4 Selecting Data	41
4.1 Sampling Data	41
4.2 Variable Roles	43
4.3 Automatic Role Identification	44
4.4 Weights Calculator	45
5 Exploring Data	47
5.1 Summarising Data	48

5.1.1	Summary	49
5.1.2	Describe	50
5.1.3	Basics	51
5.1.4	Kurtosis	52
5.1.5	Skewness	54
5.1.6	Missing	55
5.2	Exploring Distributions	57
5.2.1	Box Plot	60
5.2.2	Histogram	62
5.2.3	Cumulative Distribution Plot	64
5.2.4	Benford's Law	65
5.2.5	Bar Plot	70
5.2.6	Dot Plot	70
5.2.7	Mosaic Plot	70
5.3	Sophisticated Exploration with GGobi	71
5.3.1	Scatterplot	72
5.3.2	Data Viewer: Identifying Entities in Plots	75
5.3.3	Other Options	76
5.3.4	Further GGobi Documentation	77
5.4	Correlation Analysis	78
5.4.1	Hierarchical Correlation	82
5.4.2	Principal Components	82
5.5	Single Variable Overviews	82

6 Transforming Data	83
6.1 Normalising Data	85
6.1.1 Recenter	85
6.1.2 Scale [0,1]	88
6.1.3 Rank	88
6.1.4 Median/MAD	88
6.2 Impute	88
6.2.1 Zero/Missing	90
6.2.2 Mean/Median/Mode	90
6.2.3 Constant	92
6.3 Remap	92
6.3.1 Binning	92
6.3.2 Indicator Variables	92
6.3.3 Join Categoricals	95
6.3.4 Math Transforms	95
6.4 Outliers	95
6.5 Cleanup	95
6.5.1 Delete Ignored	97
6.5.2 Delete Selected	97
6.5.3 Delete Missing	97
6.5.4 Delete Entities with Missing	97
7 Building Classification Models	99
7.1 Building Models	100

7.2	Risk Charts	101
7.3	Decision Trees	105
7.3.1	Tutorial Example	107
7.3.2	Formalities	107
7.3.3	Tuning Parameters	107
7.4	Boosting	109
7.4.1	Tutorial Example	110
7.4.2	Formalities	111
7.4.3	Tuning Parameters	111
7.5	Random Forests	111
7.5.1	Tutorial Example For Data Mining Survival	113
7.5.2	Formalities	118
7.5.3	Tuning Parameters	118
7.6	Support Vector Machine	119
7.7	Logistic Regression	121
7.8	Bibliographic Notes	123
8	Unsupervised Modelling	125
8.1	Cluster Analysis	125
8.1.1	KMeans	125
8.1.2	Export KMeans Clusters	125
8.1.3	Discriminant Coordinates Plot	126
8.1.4	Number of Clusters	126
8.2	Hierarchical Clusters	128

8.3 Association Rules	129
8.3.1 Basket Analysis	129
8.3.2 General Rules	131
9 Evaluation	135
9.1 The Evaluate Tab	136
9.2 Confusion Matrix	138
9.2.1 Measures	138
9.2.2 Graphical Measures	138
9.3 Lift	139
9.4 ROC Curves	140
9.5 Precision versus Recall	140
9.6 Sensitivity versus Specificity	140
9.7 Scoring	140
9.8 Calibration Curves	141
10 Issues	143
10.1 Model Selection	143
10.2 Overfitting	144
10.3 Imbalanced Classification	144
10.3.1 Sampling	145
10.3.2 Cost Based Learning	146
10.4 Model Deployment and Interoperability	146
10.4.1 SQL	146
10.4.2 PMML	147

10.5 Bibliographic Notes	147
11 Moving into R	149
11.1 The Current Rattle State	149
11.2 Data	151
11.3 Samples	151
11.4 Projects	153
11.5 The Rattle Log	153
11.6 Further Tuning Models	154
12 Troubleshooting	157
12.1 Cairo: cairo_pdf_surface_create could not be located	157
12.2 A factor has new levels	158
II R for the Data Miner	159
13 R: The Language	161
13.1 Obtaining and Installing R	164
13.1.1 Installing on Debian GNU/Linux	165
13.1.2 Installing on MS/Windows	165
13.1.3 Install MS/Windows Version Under GNU/Linux .	165
13.2 Interacting With R	166
13.2.1 Basic Command Line	167
13.2.2 Emacs and ESS	169
13.2.3 Windows, Icons, Mouse, Pointer—WIMP	170

13.3 Evaluation	173
13.4 Help	174
13.5 Assignment	176
13.6 Libraries and Packages	177
13.6.1 Searching for Objects	177
13.6.2 Package Management	178
13.6.3 Information About a Package	180
13.6.4 Testing Package Availability	181
13.6.5 Packages and Namespaces	182
13.7 Basic Programming in R	183
13.7.1 Folders and Files	183
13.7.2 Flow Control	184
13.7.3 Functions	184
13.7.4 Apply	185
13.7.5 Methods	185
13.7.6 Objects	186
13.7.7 System	187
13.7.8 Misc	190
13.7.9 Internet	190
13.8 Memory Management	191
13.8.1 Memory Usage	191
13.8.2 Garbage Collection	193
13.8.3 Errors	194
13.9 Frivolous	194

13.9.1 Sudoku	194
13.10 Further Resources	197
13.10.1 Using R	197
13.10.2 Specific Purposes	197
14 Data	199
14.1 Data Types	199
14.1.1 Numbers	200
14.1.2 Strings	201
14.1.3 Logical	203
14.1.4 Dates and Times	203
14.1.5 Space	205
14.2 Data Structures	205
14.2.1 Vectors	205
14.2.2 Arrays	206
14.2.3 Lists	207
14.2.4 Sets	207
14.2.5 Matrices	207
14.2.6 Data Frames	208
14.2.7 General Manipulation	210
14.3 Loading Data	215
14.3.1 Interactive Responses	215
14.3.2 Interactive Data Entry	215
14.3.3 Available Datasets	217

14.3.4 CSV Data Used In The Book	218
14.4 Saving Data	222
14.4.1 Formatted Output	224
14.4.2 Automatically Generate Filenames	224
14.5 Using SQLite	225
14.6 ODBC Data	226
14.6.1 Database Connection	226
14.6.2 Excel	228
14.6.3 Access	229
14.7 Clipboard Data	229
14.8 Map Data	230
14.9 Other Data Formats	232
14.9.1 Fixed Width Data	232
14.9.2 Global Positioning System	233
14.10 Documenting a Dataset	233
14.11 Common Data Problems	233
15 Graphics in R	235
15.1 Basic Plot	237
15.2 Controlling Axes	239
15.3 Arrow Axes	240
15.4 Legends and Points	241
15.4.1 Colour	243
15.5 Symbols	244

15.6 Multiple Plots	245
15.7 Other Graphic Elements	246
15.8 Maths in Labels	247
15.9 Making an Animation	248
15.10 Animated Mandelbrot	249
15.11 Adding a Logo to a Graphic	251
15.12 Graphics Devices Setup	251
15.12.1 Screen Devices	251
15.12.2 Multiple Devices	252
15.12.3 File Devices	252
15.12.4 Multiple Plots	254
15.12.5 Copy and Print Devices	255
15.13 Graphics Parameters	255
15.13.1 Plotting Region	256
15.13.2 Locating Points on a Plot	256
15.13.3 Scientific Notation and Plots	256
16 Understanding Data	259
16.1 Single Variable Overviews	260
16.1.1 Textual Summaries	260
16.1.2 Multiple Line Plots	262
16.1.3 Separate Line Plots	264
16.1.4 Pie Chart	265
16.1.5 Fan Plot	267

16.1.6 Stem and Leaf Plots	267
16.1.7 Histogram	270
16.1.8 Barplot	272
16.1.9 Trellis Histogram	273
16.1.10 Histogram Uneven Distribution	274
16.1.11 Density Plot	275
16.1.12 Basic Histogram	277
16.1.13 Basic Histogram with Density Curve	278
16.1.14 Practical Histogram	280
16.2 Multiple Variable Overviews	281
16.2.1 Pivot Tables	281
16.2.2 Scatterplot	284
16.2.3 Scatterplot with Marginal Histograms	286
16.2.4 Multi-Dimension Scatterplot	287
16.2.5 Correlation Plot	288
16.2.6 Colourful Correlations	291
16.2.7 Projection Pursuit	293
16.2.8 RADVIZ	294
16.2.9 Parallel Coordinates	295
16.3 Measuring Data Distributions	296
16.3.1 Textual Summaries	296
16.3.2 Boxplot	299
16.3.3 Violin Plot	305
16.3.4 What Distribution	306

16.3.5 Labelling Outliers	306
16.4 Miscellaneous Plots	306
16.4.1 Line and Point Plots	306
16.4.2 Matrix Data	308
16.4.3 Multiple Plots	309
16.4.4 Aligned Plots	310
16.4.5 Probability Scale	311
16.4.6 Network Plot	312
16.4.7 Sunflower Plot	314
16.4.8 Stairs Plot	315
16.4.9 Graphing Means and Error Bars	316
16.4.10 Bar Charts With Segments	319
16.4.11 Bar Plot With Means	322
16.4.12 Multi-Line Title	323
16.4.13 Mathematics	324
16.4.14 Plots for Normality	325
16.4.15 Basic Bar Chart	326
16.4.16 Bar Chart Displays	327
16.4.17 Multiple Dot Plots	329
16.4.18 Alternative Multiple Dot Plots	330
16.4.19 3D Plot	331
16.4.20 Box and Whisker Plot	332
16.4.21 Box and Whisker Plot: With Means	333
16.4.22 Clustered Box Plot	334

16.4.23 Perspective Plots	335
16.4.24 Star Plot	336
16.4.25 Residuals Plot	337
16.5 Dates and Times	338
16.5.1 Simple Time Series	339
16.5.2 Multiple Time Series	340
16.5.3 Plot Time Series	341
16.5.4 Plot Time Series with Axis Labels	342
16.5.5 Grouping Time Series for Box Plot	342
16.6 Using gGobi	343
16.6.1 Quality Plots Using R Data Mining Survival	343
16.7 Textual Summaries	345
16.7.1 Stem and Leaf Plots	347
16.7.2 Histogram	349
16.7.3 Barplot	350
16.7.4 Density Plot	350
16.7.5 Basic Histogram	351
16.7.6 Basic Histogram with Density Curve	352
16.7.7 Practical Histogram	353
16.7.8 Correlation Plot	353
16.7.9 Colourful Correlations	356
16.8 Measuring Data Distributions	357
16.8.1 Textual Summaries	358
16.8.2 Boxplot	361

16.8.3 Box and Whisker Plot	363
16.8.4 Box and Whisker Plot: With Means	364
16.8.5 Clustered Box Plot	365
16.9 Further Resources	366
16.10 Map Displays	367
16.11 Further Resources	368
17 Preparing Data	369
17.1 Data Selection and Extraction	369
17.1.1 Training and Test Datasets	369
17.2 Data Cleaning	370
17.2.1 Variable Manipulations	375
17.2.2 Cleaning the Wine Dataset	377
17.2.3 Cleaning the Cardiac Dataset	377
17.2.4 Cleaning the Survey Dataset	377
17.3 Imputation	378
17.3.1 Nearest Neighbours	378
17.3.2 Multiple Imputation	378
17.4 Data Linking	379
17.4.1 Simple Linking	379
17.4.2 Record Linkage	380
17.5 Data Transformation	380
17.5.1 Aggregation	380
17.5.2 Normalising Data	381

17.5.3 Binning	383
17.5.4 Interpolation	384
17.6 Outlier Detection	384
17.7 Variable Selection	384
18 Descriptive and Predictive Analytics	387
18.1 Building a Model	388
19 Cluster Analysis: K-Means	391
19.1 Summary	391
19.1.1 Clusters	391
19.2 Other Cluster Examples	395
20 Association Analysis: Apriori	397
20.1 Summary	398
20.2 Overview	398
20.3 Algorithm	399
20.4 Usage	401
20.4.1 Read Transactions	402
20.4.2 Summary	402
20.4.3 Apriori	402
20.4.4 Inspect	402
20.5 Examples	402
20.5.1 Video Marketing: Transactions From File	403
20.5.2 Survey Data: Data Preparation	406

20.5.3 Other Examples	413
20.6 Resources and Further Reading	414
21 Classification: Decision Trees	415
21.1 Summary	415
21.2 Overview	415
21.3 Algorithm	415
21.4 Usage	415
21.4.1 Rpart	415
21.5 Examples	417
21.6 Resources and Further Reading	426
22 Classification: Boosting	427
22.1 Summary	428
22.2 Overview	428
22.3 AdaBoost Algorithm	429
22.4 Examples	433
22.4.1 Step by Step	433
22.4.2 Using gbm	436
22.5 Extensions and Variations	438
22.5.1 Alternating Decision Tree	438
22.6 Resources and Further Reading	439
23 Classification: Random Forests	443
23.1 Summary	443

23.2 Overview	443
23.3 Algorithm	443
23.4 Usage	443
23.4.1 Random Forest	444
23.5 Examples	444
23.6 Resources and Further Reading	446
24 Issues	447
24.1 Incremental or Online Modelling	447
24.2 Model Tuning	447
24.2.1 Tuning rpart	449
24.3 Unbalanced Classification	450
24.4 Building Models	451
24.5 Outlier Analysis	451
24.6 Temporal Analysis	453
24.7 Survival Analysis	453
25 Evaluation	455
25.1 Basics	455
25.2 Basic Measures	457
25.3 Cross Validation	458
25.4 Graphical Performance Measures	460
25.4.1 Lift	460
25.4.2 The ROC Curve	462
25.4.3 Other Examples	463

25.5 Calibration Curves	466
26 Cluster Analysis	467
III Text Mining	469
27 Text Mining	471
27.1 Text Mining with R	471
IV Algorithms	475
28 Bagging	479
28.1 Summary	479
28.2 Overview	480
28.3 Example	480
28.4 Algorithm	480
28.5 Resources and Further Reading	480
29 Bayes Classifier	481
29.1 Summary	481
29.2 Example	482
29.3 Algorithm	482
29.4 Resources and Further Reading	484
30 Bootstrapping	485
30.1 Summary	485

30.2 Usage	486
30.3 Further Information	486
31 Cluster Analysis	487
31.1 Discriminant Coordinates Plot	487
31.2 K Means	487
31.2.1 Summary	488
31.2.2 Clusters	488
31.3 Hierarchical Clustering	492
31.4 Summary	492
31.5 Examples	492
31.6 Resources and Further Reading	493
32 Conditional Trees	495
32.1 Summary	495
32.2 Algorithm	496
32.3 Examples	496
32.4 Resources and Further Reading	497
33 Hierarchical Clustering	499
33.1 Summary	499
33.2 Examples	499
33.3 Resources and Further Reading	499
34 K-Nearest Neighbours	501
34.1 Summary	502

34.2 Resources and Further Reading	502
35 Linear Models	503
35.0.1 Linear Model	503
36 Regression: Ordinal Regression	505
37 Logistic Regression	507
37.1 Summary	507
37.1.1 Linear Model	507
37.2 Resources and Further Reading	508
38 Neural Networks	509
38.1 Overview	510
38.2 Algorithm	510
38.2.1 Neural Network	510
38.3 Resources and Further Reading	510
39 SVM	511
39.1 Overview	511
39.2 Examples	512
39.3 Resources and Further Reading	512
39.3.1 Overview	515
39.3.2 Examples	516
39.3.3 Resources and Further Reading	516

V Open Products	517
39.4 Rattle and Other Data Mining Suites	519
40 AlphaMiner	521
41 Borgelt	523
41.1 Summary	524
41.2 Usage	524
42 KNime	525
43 R	527
43.1 Summary	528
43.2 Further Information	528
44 Rapid-I	529
45 Rattle	531
45.1 Summary	532
45.2 Usage	532
46 Weka	535
46.1 Summary	536
46.2 Usage	536
VI Closed Products	541
47 C4.5	543

xxx

47.1	Summary	543
47.2	Overview	544
47.3	Resources and Further Reading	545
48	Clementine	547
48.1	Summary	547
49	Equbits Foresight	549
49.1	Summary	549
50	GhostMiner	551
50.1	Summary	551
50.2	Usage	552
51	InductionEngine	555
51.1	Summary	555
52	Oracle Data Mining	557
52.1	Summary	558
52.2	Usage	558
53	SAS Enterprise Miner	559
53.1	Summary	560
53.2	Usage	560
53.3	Tips and Tricks	561
54	Statistica	563

54.1 Summary	564
54.2 Usage	564
54.3 Sample Applications	566
54.4 Further Information	567
55 TreeNet	569
55.1 Summary	569
56 Virtual Predict	571
56.1 Summary	571
56.2 Usage	572
Togaware Watermark For Data Mining Survival	
VII Appendices	573
A Glossary	575
Bibliography	584
Index	591

List of Figures

2.1	The introductory screen displayed on starting Rattle	15
2.2	Initial steps of the data mining process (Tony Nolan)	17
2.3	The Rattle window showing paradigms	21
2.4	Selecting the Unsupervised paradigm	22
2.5	A sample of plots	23
3.1	Rattle title bar showing the file name	27
3.2	The CSV file chooser	29
3.3	After identifying a file to load	30
3.4	Data tab dataset summary.	31
3.5	Loading an ARFF file	33
3.6	Loading data through an ODBC database connection	35
3.7	Teradata ODBC connection	35
3.8	Netezza ODBC connection	36
3.9	Netezza configuration	36
3.10	Loading an R binary data file.	37
3.11	Loading an already defined R data frame	37

3.12 Selected region of a spreadsheet copied to the clipboard	38
3.14 Data entry spreadsheet	39
3.13 Loading an R data frame originally from the clipboard	40
4.1 Select tab choosing Adjusted as a Risk variable.	43
5.1 Missing value summary for a version of the <i>audit</i> modified to include missing values.	55
5.2 Benford stratified by Marital and Gender.	69
5.3 Mosaic plot of Age by Adjusted.	70
6.1 Transform options.	84
6.2 Selection of normalisations.	86
6.3 Normalisations of Age.	86
6.4 Normalisations of Age.	87
6.5 Selection of imputations.	89
6.6 Imputation using the mode for missing values of Age.	91
6.7 Binning Age.	93
6.8 Distributions of binned Age.	93
6.9 Turning Gender into an Indicator Variable.	94
6.10 Selection of cleanup operations.	96
7.1 Random forest tuning parameters.	114
7.2 Random forests only supports factors with up to 32 levels.	114
7.3 Random forest model of audit data.	115
7.4 Random forest model measure of variable importance.	116
7.5 Random forest risk charts: test and train datasets.	117

7.6 Warning when evaluating a model on the training dataset.	117
7.7 Random forest ROC chart.	118
8.1 KMeans Iteration Interface	127
8.2 KMeans Iteration Plot	127
9.1 Informal dialog when using training set for evaluation	137
13.1 R command line under GNU/Linux	168
13.2 R command line under MS/Windows	168
13.3 R GUI using ESS for Emacs	170
13.4 R Commander GUI	171
16.1 An ordered monthly box plot.	343
18.1 A approximate model of random data.	389
22.1 Reduced example of an alternating decision tree.	440
22.2 Audit risk chart from an alternating decision tree.	440
45.1 Togaware's Rattle Gnome Data Mining interface.	533
46.1 The Weka GUI chooser.	537
46.2 Weka explorer viewing data.	538
46.3 Import CSV data into Weka.	538
46.4 Output from running J48 (C4.5).	539
50.1 Fujitsu GhostMiner interface.	553

52.1 Sample ODMiner interface to ODM.	558
53.1 SAS Enterprise Miner interface (Version 4).	561
54.1 <i>Statistica</i> Data Miner graphical interface.	565

Togaware Watermark For Data Mining Survival

List of Tables

29.1 Contact lens training data.	482
--	-----

Togaware Watermark For Data Mining Survival

Togaware Watermark For Data Mining Survival

List of Listings

3.1 Locate and view the package supplied sample dataset	28
3.2 Sample of a CSV format dataset	28
3.3 Sample of an ARFF format dataset	33
3.4 ARFF ISO-8601 date specification <i>and Mining Survival</i>	34
3.5 Load data from clipboard into R	39
code/get-wine.R	219
code/get-survey.R	221
code/get-australia.R	230
graphics/map-australia-plot.R	231
graphics/rplot-iris-scatter.R	237
graphics/rplot-iris-topbox.R	239
graphics/rplot-iris-arrows.R	240
graphics/rplot-legends.R	241
graphics/rplot-mandelbrot.R	249
graphics/rplot-wine-matplot.R	263
graphics/rplot-wine-zoo.R	264
graphics/rplot-wine-pie.R	265

graphics/rplot-wine-hist.R	270
graphics/rattle-audit-explore-distr-hist-income.R	271
graphics/rplot-wine-barplot.R	272
graphics/rplot-histogram-trellis.R	273
graphics/rplot-iris-density.R	275
graphics/rplot-hist.R	277
graphics/rplot-hist-density.R	278
graphics/rplot-hist-colour.R	280
graphics/rplot-wine-scatter.R	284
graphics/rplot-wine-scatterm.R	287
graphics/rplot-wine-corr.R	289
graphics/rplot-corr-wine.R	292
graphics/rplot-wine-boxplot-single.R	299
graphics/rplot-wine-boxplot-multi.R	300
graphics/rplot-wine-boxplot-type.R	301
graphics/rplot-wine-boxplot-tuning.R	303
graphics/rplot-boxplot-qplot.R	304
graphics/rplot-wine-vioplot.R	305
graphics/rplot-dot.R	306
graphics/rplot-matplot.R	308
graphics/rplot-multi-hist.R	309
graphics/rplot-multi-align.R	310
graphics/rplot-proby.R	311
graphics/rplot-network.R	312

graphics/rplot-line.R	315
graphics/rplot-line-means.R	316
graphics/rplot-bar-complex.R	320
graphics/rplot-bar-means.R	322
graphics/rplot-titles.R	323
graphics/rplot-labels.R	324
graphics/rplot-bar.R	326
graphics/rplot-bar-horizontal.R	327
graphics/rplot-trellis.R	329
graphics/rplot-trellis-shapes.R	330
graphics/rplot-3dbox.R	331
graphics/rplot-boxplot.R	332
graphics/rplot-boxplot-means.R	333
graphics/rplot-bwplot.R	334
graphics/rplot-stars.R	336
graphics/rplot-lm-residuals.R	337
graphics/rplot-date.R	338
graphics/rplot-time-multi.R	340
graphics/rplot-time-basic.R	341
graphics/rplot-time-basic.R	341
graphics/rplot-time-basic-labels.R	342
graphics/rplot-wine-hist.R	349
graphics/rplot-wine-barplot.R	350
graphics/rplot-iris-density.R	350

graphics/rplot-hist.R	351
graphics/rplot-hist-density.R	352
graphics/rplot-hist-colour.R	353
graphics/rplot-wine-corr.R	354
graphics/rplot-corr-wine.R	357
graphics/rplot-wine-boxplot-single.R	361
graphics/rplot-wine-boxplot-multi.R	362
graphics/rplot-wine-boxplot-type.R	363
graphics/rplot-boxplot.R	363
graphics/rplot-boxplot-means.R	364
graphics/rplot-bwplot.R	365
graphics/map-australia-states.R	367
graphics/rplot-cluster.R	392
graphics/rplot-rpart.R	418
graphics/rplot-adaboost.R	432
graphics/rplot-rocr-survey-lift.R	460
graphics/rplot-rocr-survey-tpfp.R	462
graphics/rplot-rocr-4plots.R	463
graphics/rplot-rocr-10xfold.R	466
graphics/rplot-cluster.R	488
graphics/rplot-ctree.R	496

Preface

Knowledge leads to wisdom and better understanding. Data mining builds knowledge from information, adding value to the tremendous stores of data that abound today—stores that are ever increasing in size and availability. Emerging from the database community in the late 1980’s the discipline of data mining grew quickly to encompass researchers and technologies from Machine Learning, High Performance Computing, Visualisation, and Statistics, recognising the growing opportunity to add value to data. Today, this multi-disciplinary and trans-disciplinary effort continues to deliver new techniques and tools for the analysis of very large collections of data. Searching through databases measuring in the gigabytes and terabytes, data mining delivers discoveries that improve the way an organisation does business. It can enable companies to remain competitive in this modern data rich, knowledge hungry, wisdom scarce world. Data mining delivers knowledge to drive the getting of wisdom.

The range of techniques and algorithms used in data mining may appear daunting and overwhelming. In performing data mining for a data rich client many decisions need to be made regarding the choice of methodology, the choice of data, the choice of tools, and the choice of application.

Data Mining with Rattle

In this book we introduce the basic concepts and algorithms of data mining, deploying the Free and Open Source Software package **Rattle**, built on top of the **R** system. As Free Software the source code of **Rattle** and **R** is available to anyone, and anyone is permitted, and indeed encouraged, to extend the software, and to read the source code to learn from it. Indeed, **R** is supported by a world wide network of some of the world’s

leading Statisticians. This book guides you through the various options that Rattle provides and serves as a user guide both to Rattle and to Data Mining. Some extensions into using R itself are presented, where this will help with the migration to using the full capacity of the R system. The companion book, *R for the Data Miner*, extensively covers the use of R for data mining.

R for the Data Miner

In this book we deploy the Free and Open Source Software statistical programming language R to illustrate the deployment of data mining technology. As Free Software the source code of R is available to anyone, and anyone is permitted, and indeed encouraged, to extend the software, and to read the source code to learn from it. Indeed, R is supported by a world wide network of some of the world's leading Statisticians. This book introduces the R language and then guides you through the various R packages that are essential and comprehensive for the Data Miner. A companion book, *Data Mining with Rattle* is a more gentle introduction to Data Mining, and uses the Rattle graphical user interface to introduce data mining (and hence requires very little knowledge of R).

Goals

This book presents a unique and easily accessible single stop resource for the data miner. It provides a practical guide to actually doing data mining. It is accessible to the information technology worker, the software engineer, and the data analyst. It also serves well as a textbook for an applications and techniques oriented course on data mining. While much data analysis and modelling relies on a foundation of statistics, the aim here is to not lose the reader in the statistical details. This presents a challenge! At times the presentation will leave the statistically sophisticated wanting a more solid treatment. In these cases the reader is referred to the excellent statistical expositions in [Dalgaard \(2002\)](#), [Venables and Ripley \(2002\)](#), and [Hastie et al. \(2001\)](#).

Organisation

Part ?? constitutes a complete guide to using Rattle for data mining.

In Chapter 2 we introduce Rattle as a graphical user interface (GUI) developed for making any data mining project a lot simpler. This covers the installation of both R and Rattle, as well as basic interaction with Rattle.

Chapters 3 to 9 then detail the steps of the data mining process, corresponding to the straightforward interface presented through Rattle. We describe how to get data into Rattle, how to select variables, and how to perform sampling in Chapter 3. Chapter 5 then reviews various approaches to exploring the data in order for us to gain some insights about the data we are looking at as well as understanding the distribution of the data and to assess the appropriateness of any modelling.

Chapters 8 to 27 cover modelling, including descriptive and predictive modelling, and text mining. The evaluation of the performance of the models and their deployment is covered in Chapter 9. Chapter 11 provides an introduction to migrating from Rattle to the underlying R system. It does not attempt to cover all aspects of interacting with R but is sufficient for a competent programmer or software engineer to be able to extend and further fine tune the modelling performed in Rattle. Chapter 12 covers troubleshooting within Rattle.

Part II delves much deeper into the use of R for data mining. In particular, R is introduced as a programming language for data mining. Chapter 13 introduces the basic environment of R. Data and data types are covered in Chapter 14 and R's extensive capabilities in producing stunning graphics is introduced in Chapter 15. We then pull together the capabilities of R to help us understand data in Chapter 16. We then move on to preparing our data for data mining in Chapter 17, building models in Chapter 24, and evaluating our models in Chapter 25.

Part IV reviews the algorithms employed in data mining. The encyclopedic type overview covers many tools and techniques deployed within data mining, ranging from decision tree induction and association rules, to multivariate adaptive regression splines and patient rule induction methods. We also cover standards for sharing data and models.

We continue the Desktop Guide with a snapshot of some current alternative open source and then commercial data mining products in Part [V](#), *Open Source Products*, and Part [VI](#), *Commercial Off The Shelf Products*.

Features

A key feature of this book, that differentiates it from many other very good textbooks on data mining, is the focus on the end-to-end process for data mining. That is, we cover in quite some detail the business understanding, data, modelling, evaluation, and practical deployment. In addition to presenting descriptions of approaches and techniques for data mining using modern tools, we provide a very practical resource with actual examples using Rattle. These will be immediately useful in delivering on data mining projects. We have chosen an easy to use yet very powerful open source software for the examples presented in the book. Anyone can obtain the appropriate software for free from the Internet and quickly set up their computer with a very sophisticated data mining environment, whether they use GNU/Linux, Unix, Mac/OSX, or even MS/Windows.

Audience

The book is accessible to many readers and not necessarily just those with strong backgrounds in computer science or statistics. At times we do introduce some statistical, mathematical, and computer science notations, but intentionally keep it simple. Sometimes this means oversimplifying concepts, but only where it does not lose intent of the concept and only where it retains its fundamental accuracy.

Typographical Conventions

We use the R language and the Rattle application to illustrate concepts and modelling in data mining. R is both a programming language and an interpreter. When we illustrate interactive sessions with R we will

generally show the R prompt, which by default is “>”. This allows the output from R to more easily be distinguished. However, we generally do not include the continuation prompt (a “+”) which appears when a single command extends over multiple lines, as this makes it more difficult to cut-and-paste from the examples in the electronic version of the book.

In providing example output from commands, at times we will truncate the listing and indicate missing components with [...]. While most examples will illustrate the output exactly as it appears in R there will be times where the format will be modified slightly to fit with publication limitations. This might involve silently removing or adding blank lines.

In describing the functionality of Rattle we will use a sans serif font to identify a Rattle **widget** (which is a graphical user interface component that we interact with, such as a button or menu). All of the widgets are indexed under Rattle/Widget in the index. The kinds of widgets that are used in Rattle include the **check box** for turning options on and off, and the **radio button** for selecting an option from a list of alternatives,

A Note on Languages

Rattle, as a graphical user interface, has been developed using the Gnome toolkit with the Glade GUI builder. Java was considered for the implementation of the GUI, using the Swing toolkit. But although Java was a very good language in its early days, it continued to grow and lose its attraction as a clean and easy to use and deploy system. Deployment became painful, and the Java system remained very memory hungry. Sun also refused for a long time to free it of restrictive license limitations.

Gnome, on the other hand, is programming language independent. Indeed, the GUI side of Rattle started out as a Python program using Gnome before it moved to R. The Rattle GUI is developed using the Glade GUI builder, which is very simple and easy to use. This tool generates an XML file that describes the interface in a programming language independent way. That file can be loaded into any supported programming language to immediately display the GUI. The actual functionality underlying the application can then be written in any supported language, which includes Java, C, C++, Ada, Python, Ruby, and R! We

have the freedom, though, to rather quickly change languages if the need arose.

Acknowledgements

Many thanks to my many students from the Australian National University over the years who have been the reason for me to collect my thoughts and experiences with data mining to bring together into this book. I have benefited from their insights into how they learn best. They have also contributed in a number of ways with suggestions and example applications. I am also in debt to my colleagues over the years, particularly Peter Milne and Zhexue Huang, for their support and contributions over the years in the development of Data Mining in Australia.

Colleagues in various organisations deploying or developing skills in data mining have also provided significant feedback, as well as the motivation, for this book. These include, but are not limited to my Australian Taxation Office colleagues, Stuart Hamilton, Frank Lu, Anthony Nolan, Peter Ricci, Shawn Wicks, and Robert Williams.

This book has grown from a desire to share experiences in using and deploying data mining tools and techniques. A considerable proportion of the material draws on over ten years of teaching data mining to undergraduate and graduate students and running industry outreach courses. The aim is to provide recipe type material that can be instantly deployed, as well as reference material covering the concepts and terminology a data miner is likely to come across.

Many have contributed to the content of the book, providing insights and comments. Illustrative examples of using R have also come from the R mailing lists and I have used many of these to guide the kinds of examples that are included in the book. Many contributors to that list need to be thanked, and include Gabor Grothendi, Jim Holtman, Domenico Vistocco, and Earl F. Glynn.

Financial support for maintenance of the book is always welcome. Financial support is used to contribute toward the costs of running the web pages and the desktop machine used to make this book available.

I acknowledge the support of many, including: Boudhayan Sen, Nissen Lars, Ravi Vishnu, Graca Gaspar, Dag Petter Svendsen, Danijela Puric-Mladenovic, Barend Bronsvoort, Carlos Martinez, Robert Berry, William West, Hennie Gerber, Victor Urrea Gales, Jose Matias, Rene Koch, David Thomas, Daniel Piret, Idielle Walters, Ambrose Andrews, Katsuhiko Kawai, Ellen Pitt, Jean Coursol, Dorothy Webb, Julian Shaw, Nicholas Barcza, Julien Mazerolle, Douglas Stone, Chris Liles, Takehiko Yasukawa, Thomas Callahan, Timothy Boudreau, Leif Kastdalén, Robert Flagg, Steven King, Robert Azzopardi, Peter Newbigin, Anthony Holmes, Daniel Naiman, Joseph Larimarange, Deborah Champagne, Joshua Rosenthal, James Porzak, Gary Kerns, Alfonso Iodice, Milton Cabral, Gail McEwen, Wade Thunborg, Longbing Cao, Martin Schultz, Danilo Cillario, Toyota Finance, Michael Stigall, Melanie Hilario, Siva Ganesh, Myra O'Regan, Stephen Zelle, Welling Howell, Adam Weisberg, Takaharu Aaki, Caroline Rodriguez, Patrick L Durusau, Menno Bot, Dorene A Gilyard, Henry Walker, Fei Huang, Di Peter Kralicek, Lo Siu Keung, Julian Leslie, Mona Habib, John Chow, Michael Provost, Hamish R Hutchison, Chris Raymond, Keith Lyons, Shawn Swart, Hubert Weikert, and Tom Thomas.

Graham J Williams

Canberra

Togaware Watermark For Data Mining Survival

Part I

Data Mining with Rattle

Togaware Watermark For Data Mining Survival

Chapter 1

Introduction

We are living in a time where data is collected and stored in unprecedented volumes. Large and small enterprises collect data about their businesses, their customers, their human resources, their products, their manufacturing processes, their suppliers, their business partners, their local and international markets and their competitors. Turning this data into information and that information into knowledge has become a key component of the success of a business. Data contains valuable information that can support managers in their business decisions in effectively and efficiently running a business. Information is the basis for identifying new opportunities. Knowledge is the lynchpin of society!

Data mining is about building models from data. We build models to gain insights into the world and how the world works. A data miner, in building models, deploys many different data analysis and model building techniques. Our choices depend on the business problems to be solved. Although data mining is not the only approach it is becoming very widely used because it is well suited to the data environments we find in today's enterprises. This is characterised by the volume of data available, commonly in the gigabytes and fast approaching the terabytes, and the complexity of that data, both in terms of the relationships that are awaiting discovery in the data and the data types available today, including text, image, audio, and video. Also, the business environments are rapidly changing, and analyses need to be regularly performed and models regularly updated to keep up with today's dynamic world.

Modelling is what people often think of when they think of data mining. Modelling is the process of turning data into some structured form or model that reflects that data in some useful way. Overall the aim is to address a specific problem through modelling the world in some way, and from that model to develop a better understanding of the world.

There is a bewildering array of tools and techniques at the disposal of the data miner for gaining insights into data and for building models.

In this chapter we introduce a modelling framework within which we can present the various algorithms that we use in data mining for building models of the world.

1.1 Data Mining

1.2 Types of Analysis

Much of the terminology used in data mining has grown out of that used in both machine learning and statistics. We identify, for example, two very broad categories of analysis as **unsupervised** and **supervised** (as in supervised and unsupervised learning).

1.3 Data Mining Applications

Data mining is finding application in many areas, and indeed, is becoming an all pervasive technology. Here we highlight many of the traditional areas in which data mining has played a successful role.

1.4 A Framework for Modelling

Architects build models. Why? To see how things fit together, to make sure they do fit together, to see how things will work in the real world, and even to sell the idea behind the model they build! Data mining is

about building models that give us insights into the world and how the world works.

Building models is fundamental to understanding our world. When we build a model, whether it be with lego bricks or computer software, we get a new perspective of how things fit together or interact. Once we have some basic models we can start to get ideas about more complex models, building on what has come before.

In understanding new complex ideas we often begin by trying to map the idea into concepts or constructs that we already know, by bringing those constructs together in different ways that reflect how we understand the new complex idea. As we learn more about the new complex idea we change our model to better reflect the idea, until eventually we have a model that matches the idea enough for us to make good effect of our understanding of the idea.

And so it is with model building in computer science. Indeed, writing a computer program is essentially about building a model.

There are three components to building a model: how do we represent the knowledge (the **language** for building models); how do we search through all the possible ways of building the model (**sentences** in the language); and how do we know when we have a good model (**measurement**). In all of the model building that we are going to talk about in this book, we will use this framework to present the approach and to contrast the approach to alternatives.

1.5 Agile Data Mining

It is a curious fact that building models, in the context of the framework we have just presented, is but one task of the data miner, albeit perhaps the most important task. Almost as important, though, are all the other tasks associated with data mining. We must ensure our data mining activities are tackling the right business problem. We must understand the data that is available and turn noisy data into data from which we can build robust models. We must evaluate and demonstrate the performance of our models. And we must ensure the effective deployment of our models.

Whilst we can easily describe these steps, it is important to be aware that data mining is really what we might call an agile activity. The concept of agility comes from the Agile Software Engineering principles which include the evolution or incremental development of the business requirements, the requirement for regular client input or feedback, the testing of our models as they are being developed, and frequent rebuilding of the models to improve their performance. An allied aspect is the concept of peer programming where two data miners work together on the same data, in a friendly, competitive and collaborative approach to building models. The agile approach also emphasises the importance of face-to-face communication over all the effort that is otherwise expended, and often wasted, on written documents. This is not to remove the need to write documents, but to identify what is really required to be documented.

This book provides a practical guide to data mining, showing practitioners how to deliver successful data mining projects. It does this by stepping through the stages of an idealised data mining project. We say “idealised” because every project is different, offering different challenges, and often requiring different approaches to the model building. Nonetheless, we build from the commonality presented here, to form a solid foundation for successful data mining.

We identify the steps in a data mining project and note that the following chapters then walk us through these steps, one step at a time!

As well as the chapters in this book following this step-by-step process of a data mining project, the open source and freely available tool, **Rattle**, that is used here to illustrate data mining, is very much based around these same steps. Using a tab based interface, each tab represents one of the steps, and we proceed through the tabs as we work our way through a data mining project. One noticeable exception to this is the first step of business understanding. That is something that needs study, discussion, thought, and brain power, and practical tools to help in this process are not common.

Chapter 2

Rattle Data Miner

In learning about data mining it is important to learn by example and by doing. Data mining is a very practical activity, often following ones nose as we weave our way through our data. Our aim through this book is to provide hands on practice in data mining. For this we need a tool, and ideally, not one that is expensive and aims to hide how things are done. Instead, we use the open source and freely available data mining tool, called Rattle. It is available for anyone to download from rattle.togaware.com.

Rattle (the R Analytical Tool To Learn Easily) is a graphical data mining application built upon the statistical language R. An understanding of R is not required in order to use Rattle. However, a basic introduction is provided in Chapter 13 with the idea being that this is a springboard into more sophisticated data mining in R itself. Rattle is simple to use, quick to deploy, and allows us to rapidly work through the modelling phase of a data mining project. R, on the other hand, provides a very powerful language for performing data mining, and when we need to fine tune our data mining projects we can migrate from Rattle to R simply by taking Rattle's underlying commands and deploying them within the R console.

Rattle uses the Gnome graphical user interface and runs under various operating systems, including GNU/Linux, Macintosh OS/X, and MS/Windows. Its intuitive user interface takes us through the basic steps of data

mining, as well as illustrating (through a **Log** tab) the actual R code that is used to achieve this. The corresponding R code can be saved to file and used as a script which can be loaded into R (outside of **Rattle**) to repeat any data mining exercise.

While **Rattle** by itself may be sufficient for all of a user's needs, it also provides a stepping stone to more sophisticated processing and modelling in R itself. The user is not limited to how **Rattle** does things. For sophisticated and unconstrained data mining, the experienced user can progress to interacting directly with a powerful language.

In this chapter we present the **Rattle** interface, and its basic environment for interaction, including menus and toolbars, and saving and loading projects. Chapter 3 works through the process of loading data into **Rattle** and Chapter 5 presents the various options within **Rattle** for exploring our data. Chapter 6 considers the options for transforming our data in various ways. We then go through the process of building models and evaluating the models in Chapters 7.1 to 9. Chapter 11 provides an insight into how **Rattle** works under the bonnet. It begins the user on their way to using R itself.

We begin this chapter with the instructions for installing **Rattle**.

2.1 Installing GTK, R, and Rattle

Rattle is distributed as an R package and is freely available from [CRAN](#), the Comprehensive R Archive Network. The latest development version is also available as an R package from [Togaware](#). Whilst this is a development version, it is generally quite stable, and bugs will be fixed in this version very quickly. The source code is freely available from [Google Code](#), where it is also possible to join the **Rattle** users mailing list.

Below we will find a few pages that cover the installation of the system. This might seem quite daunting, but the process is straightforward and we do try to cover various contingencies and operating system differences all in one go!

The first step in installing **Rattle** is to install the GTK+ libraries, which provide the Gnome user interface used by **Rattle**. We need to install the

correct package for our operating system. This installation is independent of the installation of R itself and is emphasised as a preliminary step that is often overlooked when installing Rattle.

If you are new to R there are just a few steps to get up and running with Rattle. If you are running on a Macintosh¹, be sure to run R from inside X11 (off the XCode CD) using the X11 shell to start R. Native Mac GTK+ is not fully supported. (You also need to use gcc 4.0.3, rather than the Mac's 4.0.1.) Be sure to install the glade libraries before installing RGtk2, since RGtk2 will ignore libraries that it can't find at the time of installation (e.g., you may find a “newGladeXML is undefined” error message when starting up Rattle).

2.1.1 Quick Start Install

The **Quick Start** (with Debian GNU/Linux assumed for the example) is:

- Install the GTK+ libraries for your operating system

```
$ wajig install libglade2-0
```

- Install R for your operating system

```
$ wajig install r-base-core
```

- Install Rattle

```
$ R
> install.packages("rattle", dependencies=TRUE)
```

- Start up Rattle

```
> library(rattle)
> rattle()
```

And that is all there is! The details and idiosyncracies follow.

¹Thanks to Daniel Harabor and Antony Unwin for the Mac/OSX information.

2.1.2 Installation Details

We now provide a detailed step-by-step guide for installing Rattle. For your particular operating system (e.g., GNU/Linux), simply ignore those paragraphs beginning with the name of another operating system (i.e., MS/Windows and Mac/OSX). This intermingling of instructions, whilst on the surface appears complicated, simplifies the presentation of the process.

1. Install the GTK+ libraries

GNU/Linux: these libraries will already be installed if you are running Gnome, but make sure you also have libglade installed. If you are not running Gnome you may need to install the GTK+ libraries in your distribution. For example, with the excellent [Debian GNU/Linux](#) distribution you can simply install the Glade package:

```
Debian: wajig install libglade2-0
```

MS/Windows: install the latest version of the Glade package from the [Glade for Windows](#) website. Download the self-installing package (e.g., `gtk-dev-2.10.11-win32-1.exe`) and open it to install the libraries:

```
MS/Windows: run gtk-dev-2.10.11-win32-1.exe
```

An alternative that seems to work quite well (thanks to Andy Liaw for pointing this out) is to run an R script that will install everything required for GTK+ on MS/Windows. This R script installs the `rggobi` package (and other things it depends on). You can start up R (after installing R as in step 2 below) and then type the command:

```
source("http://www.ggobi.org/downloads/install.r")
```

This installs the GTK libraries for MS/Windows and the rggobi package for R. (You need R installed already of course - see the next step.)

Mac/OSX: make sure Apple X11 is installed on your machine as GTK (and anything built with it) is not native to OSX. Using darwinports (from <http://darwinports.opendarwin.org/>) you can install the packages:

```
Mac/OSX: $ sudo port install gtk2      (couple of hours)
Mac/OSX: $ sudo port install libglade2 (about 10 minutes)
```

All Operating Systems: after installing libglade or any of the other libraries, be sure to restart the R console, if you have one running. This will ensure R can find the newly installed libraries.

2. Install R

GNU/Linux: R is packaged for many GNU/Linux distributions. For example, on [Debian GNU/Linux](#) install the packages with:

```
Debian: $ wajig install r-recommended
```

MS/Windows: the binary distribution can be obtained from the [R Project](#) website. Download the self-installing package and open it (and R will be installed—we can generally accept all of the default options that are offered in the installation process).

```
MS/Windows: run R-2.6.1-win32.exe
```

Mac/OSX: download the package from CRAN and install it.

All Operating Systems: to confirm you have R installed, start up a Terminal and enter the command R (that's just the capital letter R). If the response is that the command is not found, then you probably need to install the R application!

```
$ R

R version 2.6.0 (2007-10-03)
Copyright (C) 2007 The R Foundation for Statistical Computing
ISBN 3-900051-07-0

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.

[Previously saved workspace restored]
>
```

3. Install RGtk2

This package is available on CRAN and from the [RGtk2](#) web site. If you have installed all of the dependent R packages in the previous step then you won't need to install it here. Otherwise, from R, use:

```
R: > install.packages("RGtk2")
```

Or to install the most recent release:

```
R: > install.packages("RGtk2", repos="http://www.ggobi.org/r/")
```

GNU/Linux: you can generally just install the appropriate package for your distribution. On Debian this is done with:

```
Debian: $ wajig install r-cran-gtk2
```

Mac/OSX: download the source package from http://www.ggobi.org/rgtk2/RGtk2_2.8.6.tar.gz, and run the command line install:

```
Mac/OSX: $ R CMD INSTALL RGtk2_2.8.6.tar.gz (30 minutes)
```

You may not be able to compile RGtk2 via the R GUI on Mac/OSX as the GTK libraries can not be found when `gcc` is called. Once installed though, R will detect the package; just don't try to load it within the GUI as GTK is not a native OSX application and it will break. On the Mac/OSX be sure to run R from X11.

All Operating Systems: to confirm you have RGtk2 installed enter the R command

```
R: > library(RGtk2)
```

4. Install R Packages

The following additional R packages are used by Rattle, and without them some functionality will be missing. Rattle will gracefully handle them being missing so you can install them when needed,

or else all at once. If we perform an action where Rattle indicates a package is missing, we can then install the package, if we wish. Type `?install.packages` at the R prompt for further help on installing packages. To automatically get all of the packages suggested by Rattle (and suggested by its dependencies in turn), use the *dependencies* option. Otherwise list them all out in the command or list just the ones you want:

```
R: > install.packages("rattle", dependencies=TRUE)
```

or

```
R: > install.packages(c("ada", "amap", "arules", "bitops",
  "cairoDevice", "cba", "combinat", "doBy", "ellipse",
  "fEcofin", "fCalendar", "fBasics", "fpc",
  "gdata", "gtools", "gplots", "Hmisc", "kernlab",
  "mice", "network", "pmlm", "randomForest", "reshape",
  "rggobi", "ROCR", "RODBC", "rpart", "RSvgDevice",
  "XML"))
```

5. Install Rattle

From within R you can install Rattle directly from CRAN with:

```
R: > install.packages("rattle")
```

An alternative is to install the most recent release from Togaware:

```
R: > install.packages("rattle",
  repos="http://rattle.togaware.com")
```

If these don't work for some reason you can also download the latest version of the `rattle` package directly from <http://rattle.togaware.com>. Download either the `.tar.gz` file for GNU/Linux and Mac/OSX, or the `.zip` file for MS/Windows, and then install with, for example:

```
R: > install.packages("rattle_2.2.84.zip", repos=NULL)
```

Alternatively, for example on Mac/OSX, you can do the following:

```
Mac: R CMD INSTALL rattle_2.2.84.tar.gz
```

Use the name of the file as it was downloaded. Some people report that the filename as downloaded actually becomes:

```
rattle_2.2.84.tar.gz.tar
```

You can either correct the name or use this name in the command.

6. Start Rattle

From an X Windows terminal if you are using GNU/Linux, or from an XTerminal if you are using Mac/OSX, or from the Rgui.exe if using MS/Windows (we call all these, generically, the **R Console**), you can load the **rattle** package into R's library:

```
R: > library(rattle)
```

This loads the Rattle functionality (which is also available without running the Rattle GUI). To start the Rattle GUI simply run the command:

```
R: > rattle()
```

We can package up the initiation of the Rattle application so that it can be run by clicking an icon on the GNU/Linux or MS/Windows desktop.

On **GNU/Linux** you can create a shell script to run Rattle. The script might be:

```
#!/bin/sh
wd=${PWD}
if [ ! -d ${HOME}/rattle ]; then mkdir ${HOME}/rattle; fi
echo "library(rattle); setwd(\"${wd}\"); rattle()" \
> ${HOME}/rattle/.Rprofile
gnome-terminal --working-directory=${HOME}/rattle \
--title="Rattle: R Console" \
--hide-menubar --execute "R"
```

This uses a trick of creating a .Rprofile file in a specific location. Such a file is read when R is started up, and the commands there tell R to start Rattle. You can call this script **rattle** and place it in **/usr/local/bin** or in your own bin folder **\${HOME}/bin**.

On **MS/Windows** a similar trick is possible. Simply start R up in a folder containing a .Rprofile with the following lines:

```
library(rattle)
rattle()
```

When starting up Rattle the main window will be displayed. You will see a welcoming message and a hint about using Rattle (see Figure 2.1).

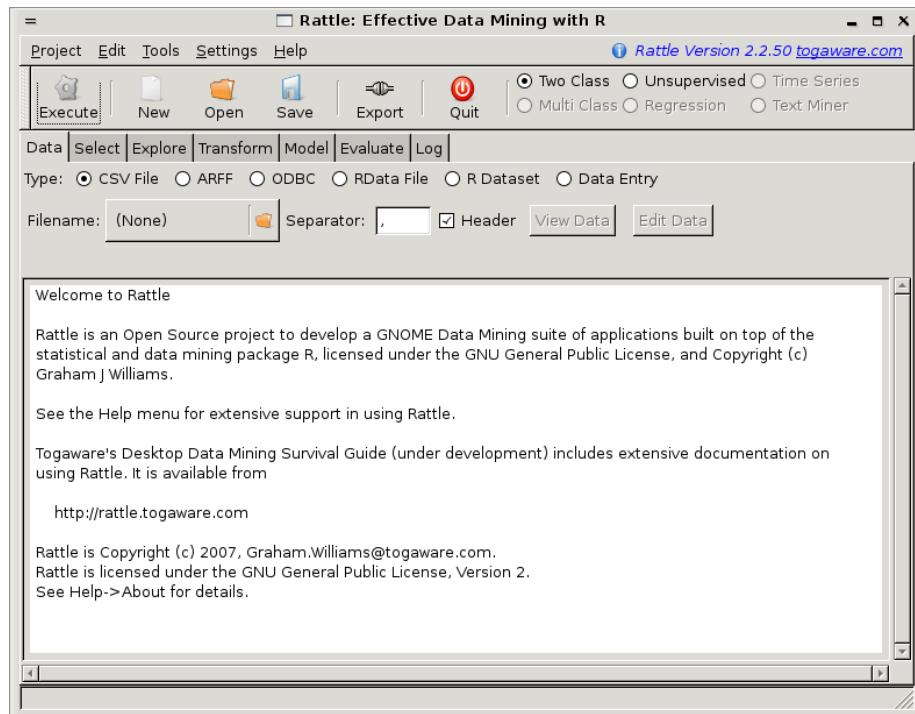


Figure 2.1: The introductory screen displayed on starting Rattle

2.2 The Initial Interface

The user interface for Rattle flows through the data mining process, progressing through the Tabs that form the primary mechanism for operating with Rattle. We essentially work our way from the left most tab (the **Data** tab) to the right most tab (the **Log** tab).

We illustrate the basics of this simple interface for the common case of the **Two Class** paradigm. The work flow process in Rattle can be summarised as:

1. Load a **Dataset**;
2. **Select** variables and entities for exploring and mining;
3. **Explore** the data;
4. **Transform** the data into training and test datasets;

5. Build your **Models**;
6. **Evaluate** the models;
7. Review the **Log** of the data mining process.

Pictorially, we illustrate a typical work flow that is embodied in the Rattle interface in Figure 2.2.

Rattle supports a number of paradigms for data mining. The collection of **Paradigms**, displayed as radio buttons to the right of the buttons on the toolbar, allow a multitude of Rattle functionality to be shared while supporting the variety of different types of tasks associated with the different paradigms. For example, selecting the **Unsupervised** paradigm will expose the **Cluster** and **Associate** tabs, suitable for descriptive data mining, whilst hiding the **Model** and **Evaluation** tabs which are most useful for predictive model building.

We will present more on paradigms in Section 2.5. Before we get there, we need to understand how to interact with Rattle.

2.3 Interacting with Rattle

The Rattle interface is based on this set of tabs through which we progress. For any tab, once we have set up the required information, we will click the Execute button to perform the actions. Take a moment to explore the interface a little. Notice the Help menu and find that the help layout mimics the tab layout.

We will work through the functionality of Rattle with the use of a simple dataset, the *audit* dataset, which is supplied as part of the Rattle package (it is also available for download as a CSV file from <http://rattle.togaware.com/audit.csv>). This is an artificial dataset consisting of 2,000 fictional clients who have been audited, perhaps for tax refund compliance. For each case an outcome is recorded (whether the taxpayer's claims had to be adjusted or not) and any amount of adjustment that resulted is also recorded.

The dataset is only 2,000 entities in order to ensure model building is relatively quick, for illustrative purposes. Typically, our data contains

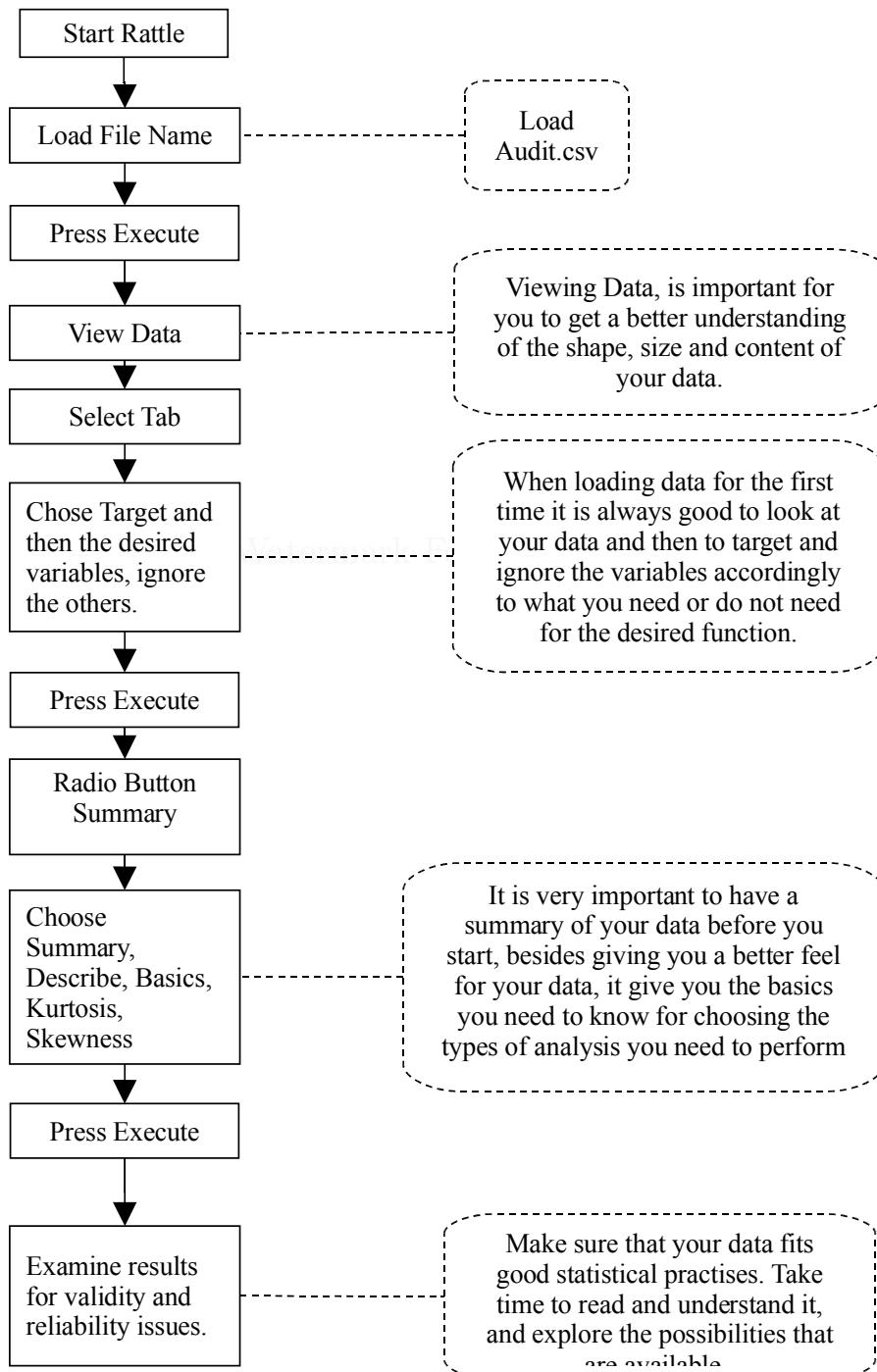


Figure 2.2: Initial steps of the data mining process (Tony Nolan)

Copyright © 2006-2008 Graham Williams

tens of thousands and more (often millions) of entities. The *audit* dataset contains 13 columns (or variables), with the first being a unique client identifier. Again, real data will often have one hundred or more variables.

We proceed through the typical steps of a data mining project, beginning with a data load and selection, then an exploration of the data, some transformations, and finally, modelling and evaluation.

We step through each tab, left to right, performing the corresponding actions. Remember that for any tab configure the options and then click the Execute button (or F5) to perform the appropriate tasks. It is important to note that the tasks are **not** performed until the Execute button (or F5 or the Execute menu item under Tools) is clicked.

The Status Bar at the base of the window will indicate when the action is completed. Messages from R (e.g., error messages, although many R error messages are captured by Rattle and displayed in a popup) will appear in the R console from where Rattle was started. *Survival*

The R Code that is executed underneath will appear in the Log tab. This allows us to review the R commands that perform the corresponding data mining tasks. The R code snippets can be copied as text from the Log tab and pasted into the R Console from which Rattle is running, to be directly executed. This allows us to deploy Rattle for basic tasks, yet still give us the full power of R to be deployed as needed, perhaps through using more command options than exposed through the Rattle interface. This also allows us the opportunity to export the whole session as an R script file as a record of the actions taken, and possibly for running directly and automatically through R itself at a later time. Simply click on the Export button to export the log to a file that will have the .R extension.

2.4 Menus and Buttons

Before we proceed into the major functionality of Rattle, which is covered in the following chapters, we will review the interface functions provided by the menus and toolbar buttons. The Open and Save toolbar functions and the corresponding menu items under the Project menu are discussed in Section 2.4.1. Projects allow the current state of your interaction with

Rattle to be saved to file for continuing later.

2.4.1 Project Menu and Buttons

A project is a packaging of a dataset, variable selections, explorations, clusters and models built from the data. Rattle allows projects to be saved for later resumption of the work or for sharing the data mining project with other users.

A project is typically saved to a file with the `.rattle` extension (although in reality it is just a standard `.RData` file).

At a later time you can load a project into rattle to restore the data, models, and other displayed information relating to the project, and resume your data mining from that point. You can also share these project files with other Rattle users, which is quite useful for data mining teams.

You can rename the files, keeping the `.rattle` extension, without impacting the project file itself—that is, the file name has no formal bearing on the contents, so use it to be descriptive—but best to avoid vacant spaces and unusual characters!

2.4.2 Edit Menu

The Edit menu is currently not implemented.

2.4.3 Tools Menu and Toolbar

Execute

It is important to understand the user interface paradigm used within Rattle. Basically, we specify within each tab what it is we want to happen, and then click the Execute button to have the actions performed. Pressing the F5 function key and selecting the menu item Execute under the Tools menu have the same effect.

Export

The **Export** button is available to export various objects and entities from Rattle. Details are available together with the specific sections in the following Chapters. The nature of the export depends on which tab is active, and within the tab, which option is active. For example, if the **Model** tab is on display then Export will save the current model as PMML.

The **Export** button is not available for all tabs and options.

2.4.4 Settings

The settings menu allows us to turn tool tips on and off and to choose whether to use the Cairo graphics device for displaying plots. The Cairo device used to display graphics is covered in Section 2.6

2.4.5 Help

Extensive help is available through the Help menu. The structure of the menu follows that of the Tabs of the main interface. On selecting a help topic, a brief text popup will display some basic information. Many of the popups then have the option of displaying further information, which will be displayed within a Web browser. This additional documentation is just that which is supplied by the corresponding R package.

2.5 Paradigms

There are many different uses to which data mining can be put. For identifying fraud or assessing the likelihood of a client to take up a particular product, we might think of the task as deciding on one of two outcomes. We might think of this as a two class problem. Or the task may be to decide what type of item from amongst a collection of items a client may have a propensity to purchase. This is then a multi class problem. Perhaps we wish to predict how much someone might overstate an insurance claim or understate their income for taxation purposes or



Figure 2.3: Paradigms as radio buttons to the right of the toolbar icons

overstate their income when seeking approval for a credit card. Here we are predicting a continuous outcome, and refer to this as **regression**.

In all of these situations, or *paradigms*, we might think of it as having a teacher who has supplied us examples of the outcomes—whether examples of fraudulent and non-fraudulent cases, or examples of different types clients, or examples of clients and their declared income and actual income. In such cases we refer to the task as **supervised modelling**.

Perhaps though we know little about the individual, specific targets, but instead have general information about our population or their purchasing patterns. We might think of what we need to do in this case as building a model without the help of a teacher—or **unsupervised modelling**.

Alternatively, our data may have some special characteristics, such as time series and text data. In these cases we have different tasks we wish to perform.

We refer to these different types of tasks in data mining as different paradigms, and Rattle provides different subsets of functionality for the different paradigms.

The paradigms are listed at the right end of the toolbar, as seen in Figure 2.3, and are selectable as radio buttons. The paradigms provided by Rattle are:

- **Two Class** for binary classification predictive modelling;
- **Multi Class** for multi-way classification predictive modelling;
- **Regression** for continuous variable predictive modelling;



Figure 2.4: Changing paradigms changes some of the displayed tabs

- **Unsupervised** for learning without a target or descriptive modelling;
- **Time Series** for temporal data mining; and
- **Text Mining** for mining of unstructured text data.

Selecting a paradigm will change the tabs that are available in the main body of Rattle. For example, the default paradigm is the Two Class paradigm, which displays a Model and Evaluate tab, as well as the other common tabs. The Model tab exposes a collection of techniques for building two class or binary models. The Evaluate tab provides a collection of tools for evaluating the performance of those models.

Selecting the Unsupervised paradigm, as in Figure 2.4, removes the Model and the Evaluate tabs, replacing them with a Cluster and an Associate tab, for cluster analysis and association analysis, respectively.

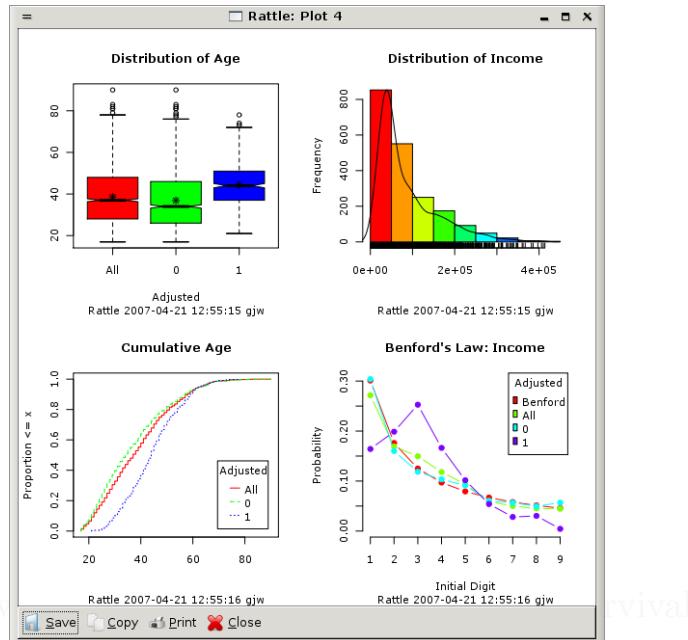


Figure 2.5: A sample of plots

2.6 Interacting with Plots

Rattle uses the Cairo device for displaying graphic plots. If the Cairo device is not available within your installation then Rattle resorts to the default window device for the operating system (*x11* for GNU/Linux and *window* for MS/Windows). The **Settings** menu also allows control of the choice of graphics device (allowing us to use the default, which is *x11* for GNU/Linux and *windows* for MS/Windows). The Cairo device has a number of advantages, one being that the device can be encapsulated within other windows, as is done with Rattle, to provide various operating system independent functionality, and a common interface. If we choose not to use the Cairo device, we will have the default devices, and they work just fine.

At the bottom of the window that embeds the Cairo device is a series of buttons that allow us to **Save** the plot to file, to **Copy** the plot to the clipboard to allow it to be pasted into other applications (e.g., OpenOffice), to **Print** the plot, and to **Close** the plot.

The **Save** button of the plot window (on the Cairo device) allows you to save the graphics to a file in one of the supported formats. The supported formats include **pdf**, **png** (good for vector images and text), **jpg** (good for colourful images), **svg** (for general scalable vector graphics), and, on MS/Windows, **wmf** (for MS/Windows specific vector graphics). A popup will request the filename to save to. The default is to save as PDF format, saving to a file with the filename extension of **.pdf**. You can choose to save in the other formats simply by specifying the appropriate filename extension.

The **Copy** button will save a copy of the plot into the clipboard. This will allow the image to be directly pasted into other applications. Typically we might create a report using OpenOffice writer, and this option can be used to include graphics from the exploration and modelling of our data. The image is captured as a PNG image, and thus as a bitmap image rather than a vector graphics. Generally the resolution will be quite adequate, but it is not scalable.

The **Print** button will send the plot off to a printer. This requires the underlying R application to have been set up properly to access the required printer.

Finally, once we are finished with the plot we can click the **Close** button to shut down that particular plot window.

2.7 Summary

In this Chapter we have covered the installation and setup of the Rattle graphical user interface for data mining with R. The basic elements of the Rattle interface have been covered, providing the basis for our interaction with Rattle.

Chapter 3

Sourcing Data

Data is the starting point for all data mining—without it there is nothing to mine. Today there is certainly no shortage of data—but turning that data into information and knowledge is no simple matter. In this chapter we explore issues relating to data, in particular, loading and selecting the data for data mining.

3.1 Nomenclature

Data miners have a plethora of terminology for many of the same things, due primarily to the history of data mining with its roots in many disciplines. Throughout this book we will use a single, consistent nomenclature, and one that is generally accepted. This nomenclature is introduced here.

We refer to collections of data as **datasets**. This might be a *matrix* or a database *table*. A dataset consists of rows which we might refer to as **entities**, and those entities are described in terms of **variables** which form the columns. Synonyms for *entity* include *record* and *object*, while synonyms for *variable* include *attribute* and *feature*.

Variables can serve one of two roles: as *input variables* or *output variables* (Hastie et al., 2001). **Input variables** are measured or preset data items while **output variables** are those that are perhaps “influenced” by the

input variables. In data mining we often build models to predict the output variables in terms of the input variables. Input variables are also known as *predictors*, *independent variables*, *observed variables* and *descriptive variables*. Output variables are also known as *response* and *dependent variables*.

Variables can be categorical or numeric. A **categorical variable** is one like eye colour and type of motor vehicle. Such variables take on a value from a fixed set of values (e.g., a colour, or `passenger vehicle`, `utility`, etc, or the common categorisations like *low*, *medium*, and *high*). A **numeric variable** has values that are integers or real numbers, such as a persons age or weight, or their income or amount of money in the bank. Synonyms for *categorical variable* include *nominal variable*, *qualitative variable* and *factor*, while synonyms for *numeric variable*, include *quantitative variable* and *continuous variables*.

Thus, we will talk of **datasets** consisting of **entities** described using **variables**, which might consist of a mixture of **input variables** and **output variables**, either of which may be **categorical** or **numeric**.

A **dataset** (or subsets of a dataset) might have different roles. For building classification models, for example, we often partition a dataset into a **training dataset** and a **testing dataset**. Typically, we build our model on the training dataset and evaluate its performance on the testing dataset.

3.2 Loading Data

The **Data** tab is the starting point for Rattle, and is where we load a specific dataset into Rattle.

Rattle is able to load data from various sources. Support is directly included in Rattle for comma separated data files (`.csv` files as might be exported by a spreadsheet), tab separated files (`.txt`, which are also commonly exported from spreadsheets), the common data mining dataset format used by Weka (`.arff` files), and from an ODBC connection (thus allowing connection to an enormous collection of data sources including MS/Excel, MS/Access, SQL Server, Oracle, IBM DB2, Teradata, MySQL, Postgress, and SQLite).



Figure 3.1: Rattle title bar showing the file name

When loading data into Rattle certain special strings are used to identify variable roles. For example, if the variable names starts with ID then the variable is marked as having an ID role See Section 4.2 for details.

Underneath Rattle, R is very flexible in where it obtains its data from, and data from almost any source can be loaded. Consequently, Rattle is able to access this same variety of sources. It does, however, require the loading of the data into the R console and then within Rattle loading it as an R Dataset. All kinds of additional data sources can be loaded directly into R—including loading data directly from SAS, SPSS, Minitab, Oracle, MySQL, and SQLite.

Once a dataset has been identified the name of the dataset will be displayed in the title of the Rattle window, as in Figure 3.1.

The remainder of this Chapter covers the loading of data sources directly supported by Rattle.

3.3 CSV Data

The CSV option of the Data tab is an easy way to load data from many different sources into Rattle. CSV stands for “comma separated value” and is a standard file format often used to exchange data between applications. CSV files can be exported from spreadsheets and databases, including OpenOffice Calc, Gnumeric, MS/Excel, SAS/Enterprise Miner, Teradata’s Warehouse, and many, many, other applications. This is a pretty good option for importing your data into Rattle, although it does lose meta data information (that is, information about the data types of the dataset). Without this meta data R sometimes guesses at the wrong data type for a particular column, but it isn’t usually fatal!

```
> system.file("csv", "audit.csv", package = "rattle")
[1] "/usr/local/lib/R/site-library/rattle/csv/audit.csv"
> file.show(system.file("csv", "audit.csv", package = "rattle"))
```

Listing 3.1: Locate and view the package supplied sample dataset

```
ID, Age, Employment, Education, Marital, Occupation, Income, Gender, ...
1004641, 38, Private, College, Unmarried, Service, 81838, Female, ...
1010229, 35, Private, Associate, Absent, Transport, 72099, Male, ...
1024587, 32, Private, HSgrad, Divorced, Clerical, 154676.74, Male, ...
1038288, 45, Private, Bachelor, Married, Repair, 27743.82, Male, ...
1044221, 60, Private, College, Married, Executive, 7568.23, Male, ...
...
```

Listing 3.2: A sample of the top 6 lines of the CSV file audit.csv

An example CSV file is provided by Rattle and is called `audit.csv`. It will have been installed when we installed Rattle and we would find it's actual location with:

The top of the file will be similar to the following (perhaps with quotes around values, although they are not necessary, and perhaps with some different values):

A CSV file is actually a normal text file that you could load into any text editor to review its contents. A CSV file usually begins with a header row, listing the names of the variables, each separated by a comma. If any name (or indeed, any value in the file) contains an embedded comma, then that name (or value) will be surrounded by quote marks. The remainder of the file after the header is expected to consist of rows of data that record information about the entities, with fields generally separated by commas recording the values of the variables for this entity.

To make a CSV file known to Rattle we click the Filename button. A file chooser dialog will pop up (Figure 3.2). We can use this to browse our file system to find the file we wish to load into Rattle. By default, only files that have a `.csv` extension will be listed (together with folders). The pop up includes a pull down menu near the bottom right, above the Open button, to allow you to select which files are listed. You can list only files that end with a `.csv` or a `.txt` or else to list all files. The `.txt` files are similar to CSV files but tend to use tab to separate columns in the data, rather than commas. The window on the left of the popup

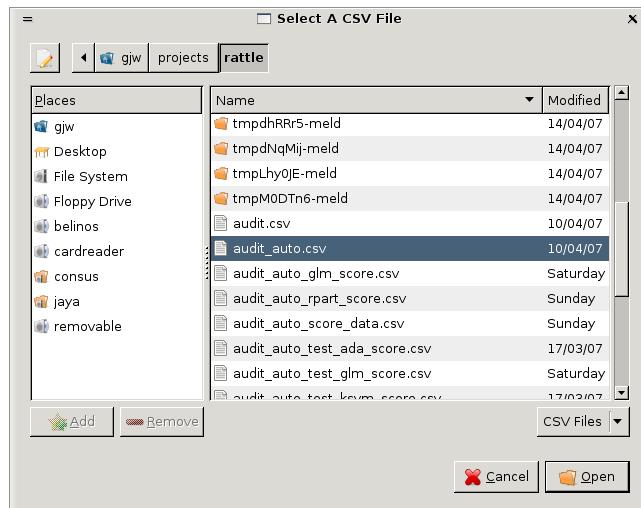


Figure 3.2: The CSV file chooser showing just those files with a .csv extension, although we can also select to display just the .txt files or else all files

allows us to browse to the different file systems available to us, while the series of boxes at the top let us navigate through a series of folders on a single file system. Once we have navigated to the folder on the file system on which we have saved the `audit.csv` file, we can select this file in the main panel of the file chooser dialog. Then click the Open button to tell Rattle that this is the file we are interested in.

Notice in Figure 3.3 that the textview of the Data tab has changed to give a reminder as to what we need to do next. That is, we have not yet told Rattle to actually load the data—we have just identified where the data is. So we now click the Execute button (or press the F5 key) to load the dataset from the `audit.csv` file. Since Rattle is a simple graphical interface sitting on top of R itself, the message in the textview also reminds us that some errors encountered by R on loading the data (and in fact during any operation performed by Rattle) may be displayed in the R Console.

You can choose the field delimiter through the Separator entry. A comma is the default. To load a .txt file which uses a tab as the field separator enter `\t` (that is, two slashes followed by a `t`) as the separator. You can

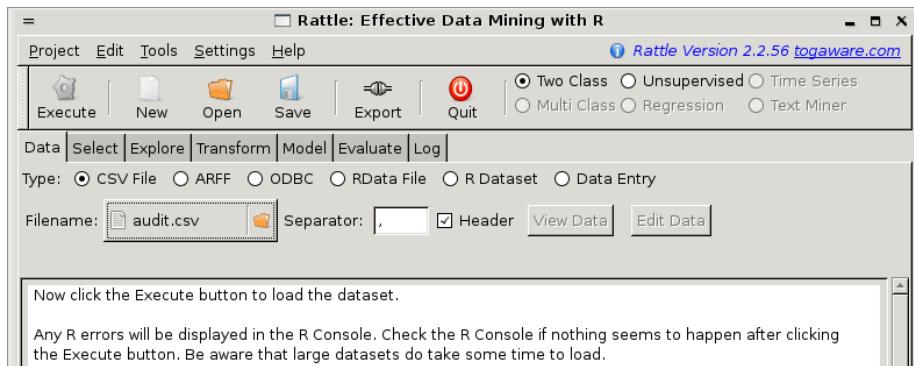


Figure 3.3: The Welcome Message of the startup window is replace by a reminder to Execute the tab before the data is actually loaded

also leave the separator empty and any white space will be used as the separator.

Any data with missing values (i.e., no value between a pair of commas) or having the value “NA” or “.” or “?” is treated as a missing value, which is represented in R as the string NA. Support for the “.” convention allows the importation of CSV data generated by SAS, whilst the usage of “?” is common following its usage in some of the early machine learning applications like C4.5.

The contents of the textview of the Data tab has now changed again, as we see in Figure 3.4. The panel contains a brief summary of the dataset. From the summary we see that Rattle has loaded the file we requested, showing the full path to the file. We then see that Rattle has created something called a ‘`data.frame`’. This is a basic data type in R used to store a table of data, where the columns (the variables) can have a mixture of data types. We then see that Rattle has loaded 2,000 entities (called observations or `obs.` in R), each described by 13 variables. The data type, and the first few values, for each entity are also displayed.

We can start getting an idea of the shape of the data from this simple summary. For example, the first two variables, `ID` and `Age`, are both identified as integers (`int`). The first few values of `ID` are 1004641, 1010229, 1024587, and so on. They all appear to be of the same length (i.e., the same number of digits) and together with having a name like `ID` provides

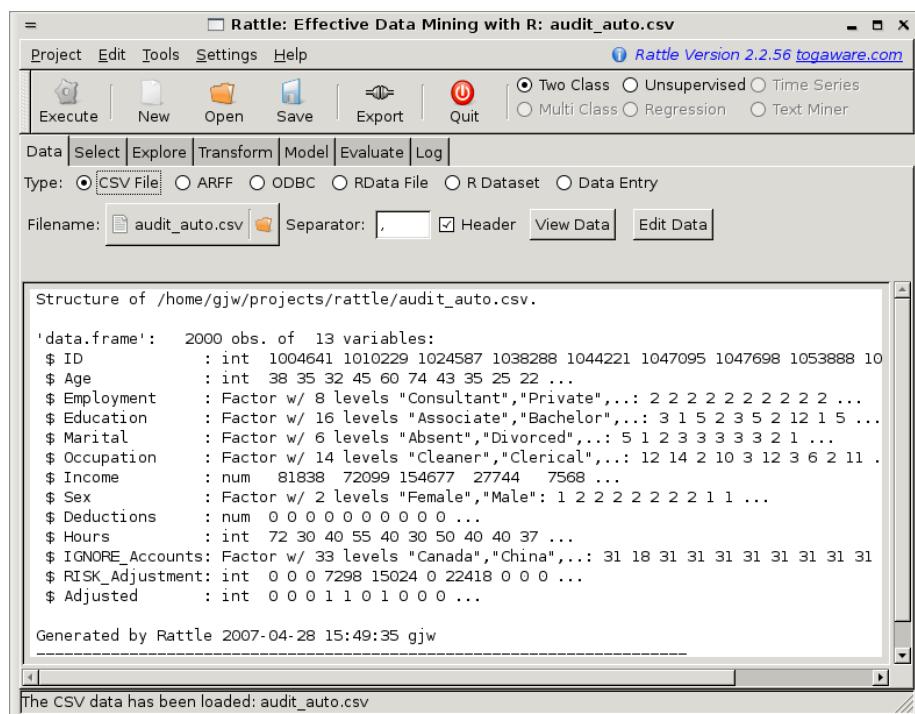


Figure 3.4: Data tab dataset summary.

a very strong indicator that this is some kind of identifier for each entity. The first few values of `Age` are 38, 35, 32, 45, 60, and so on.

The next variable, `Employment`, illustrates how R deals with categorical variables. In R terms it is a `Factor` with 8 levels (i.e., 8 possible values). The levels begin with "Consultant" and "Private". The following sequence of numbers, all of which happen to be 2 for the first 10 entities of this dataset, discloses how R stores categorical data. Effectively, R maintains an integer indexed table, associating the levels with integers, so that "Consultant" is associated with 1, "Private" with 2, and so on. Then only these integers need to be stored for each entity, which is generally more efficient on memory usage. We see this more convincingly for the following categorical variables, `Education`, `Marital`, and `Occupation` (because they have more than just a single level displayed in this summary).

The seventh variable, `Income`, has been identified as a more general numeric rather than specific integer variable. The display of the first few values does not actually give us any insight as to why this might be so, but reviewing the actual CSV data as in Listing 3.2 on page 28, we see that the third entity actually has a value of 154676.74 for `Income`, indicating that these values are real numbers rather than just integers.

We also note that `Adjusted`, for example, looks like it might be a categorical variable, with values 0 and 1, but R identifies it as an integer! That's fine for our purposes here. We can always change this later.

3.4 ARFF Data

The Attribute-Relation File Format (ARFF) is an ASCII text file format that is essentially a CSV file with a header that describes the meta-data. ARFF was developed for use in the Weka machine learning software and there are quite a few datasets in this format now. We can load an ARFF dataset into Rattle through the ARFF option (Figure 3.5).

An example of the ARFF format for our `audit` dataset is shown in Listing 3.3.

A dataset is firstly described, beginning with the name of the dataset (or

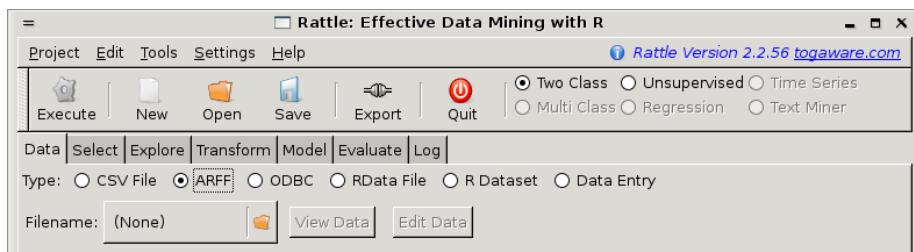


Figure 3.5: Choosing the ARFF radio button to load an ARFF file

Togaware Watermark For Data Mining Survival

```
@relation audit
@attribute ID numeric
@attribute Age numeric
@attribute Employment {Consultant, PSFederal, PSLocal, ...}
@attribute Education {Associate, Bachelor, College, Doctorate, ...}
@attribute Marital {Absent, Civil, Divorced, Married, ...}
@attribute Occupation {Cleaner, Clerical, Executive, Farming, ...}
@attribute Income numeric
@attribute Gender {Female, Male}
@attribute Deductions numeric
@attribute Hours numeric
@attribute Accounts {Canada, China, Columbia, Cuba, Ecuador, ...}
@attribute Adjustment numeric
@attribute Adjusted {0, 1}
@data
1004641,38,Private,College,Separated,Service,71511.95,Female,0,...
1010229,35,Private,Associate,Unmarried,Transport,73603.28,Male,....
1024587,32,Private,HSgrad,Divorced,Clerical,82365.86,Male,0,40,...
1038288,45,Private,?,Civil,Repair,27332.32,Male,0,55,...
1044221,60,Private,College,Civil,Executive,21048.33,Male,0,40,...
...
```

Listing 3.3: Sample of an ARFF format dataset

```
@attribute lodged date "yyyy-MM-dd'T'HH:mm:ss"
```

Listing 3.4: ARFF ISO-8601 date specification

the *relation* in ARFF terminology). Each of the variables (or *attribute* in ARFF terminology) used to describe the entities is then identified, together with their data type, each definition on a single line (we have truncated the lines in the above example). Numeric variables are identified as *numeric*, *real*, or *integer*. For categorical variables we simply see a list the of possible values.

Two other data types recognised by ARFF are *string* and *date*. A *string* data type simple indicates that the variable can have any string as its value. A *date* data type also optionally specifies the format in which the date is presented, with the default being in ISO-8601 format which is equivalent to the specification shown in Listing 3.4.

The actual entities are then listed, each on a single line, with fields separated by commas, much like a CSV file.

Comments can be included in the file, introduced at the beginning of a line with a %, whereby the remainder of the line is ignored.

A significant advantage of the ARFF data file over the CSV data file is the meta data information. This is particularly useful in Rattle where for categorical data the possible values are determined from the data (which may not included every possible value) rather than from a full list of possible values.

Also, the ability to include comments ensure we can record extra information about the data set, including how it was derived, where it came from, and how it might be cited.

Missing values in an ARFF dataset are identified using the question mark ?. These are identified by *read.arff* underneath and we see them as the usual NA in Rattle.

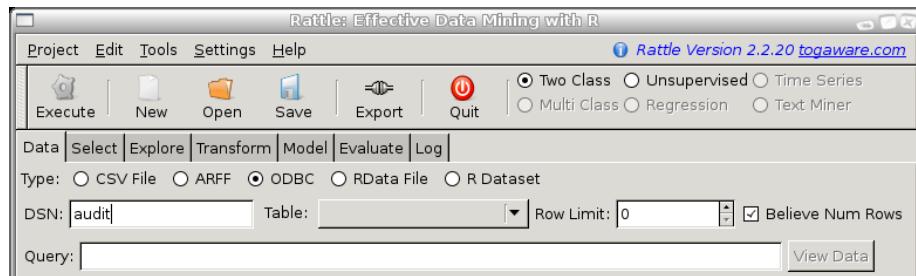


Figure 3.6: Loading data through an ODBC database connection

3.5 ODBC Sourced Data

Rattle supports obtaining a dataset from any database accessible through ODBC (Open Database Connectivity) with the ODBC Option (Figure 3.6).

The key to using ODBC is to know (or to set up) the data source name (DSN) for your databases. The setting up of DSNs is outside the scope of Rattle, being a configuration task through your operating system. Under GNU/Linux, for example, using the `unixodbc` package, the system DSNs are often defined in the file `/etc/odbcinst.ini` and in `/etc/odbc.ini`. Under MS/Windows the control panel provides access to a DSN tool.

Within Rattle we specify a known DSN by typing the name into the text entry. Once that is done, we press the Enter key and Rattle will attempt to connect. This may require



Figure 3.7: Teradata ODBC connection

a username and password to be supplied. For a Teradata Warehouse connection you will be presented with a dialog box like the one on the right (Figure 3.7). For a Netezza ODBC connection we will get a window like that in Figure 3.8.

If the connection is successful we will find a list of available tables in the Table combobox.

We can choose a Table, and also include a limit on the number of rows that we wish to load into Rattle. This allows us to get a smaller sample of the data for testing purposes before loading up a large dataset. If the Row Limit is set to 0 then all of the rows from the table are retrieved. Unfortunately there is now SQL standard for limiting the number of rows returned from a query. For the Teradata and Netezza warehouses the SQL keyword is **LIMIT** and this is what is used by Rattle.

The Believe Num Rows option is an oddity required for some ODBC drivers and appears to be associated with the pre-fetch behaviour of these drivers. The default is to activate the check box (i.e., Believe Num Rows is True). However, if you find that you are not retrieving all rows from the source table, the the ODBC driver may be using a pre-fetch mechanism that does not “correctly” report the number of rows (it is probably only reporting the number of rows limited to the size of the pre-fetch). In these cases deactivate the Believe Num Rows check box. See Section 14.6 for more details. Another solution is to either disable the pre-fetch option of the driver, or to increase its count. For example, in connecting through the Netezza ODBC driver the configuration window is available, where you can change the default Prefetch Count

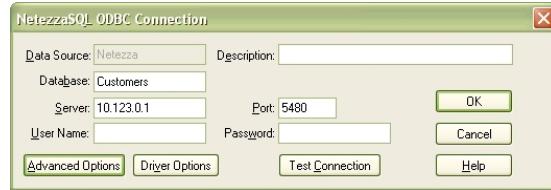


Figure 3.8: Netezza ODBC connection

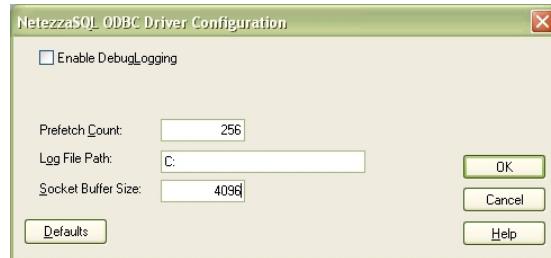


Figure 3.9: Netezza configuration

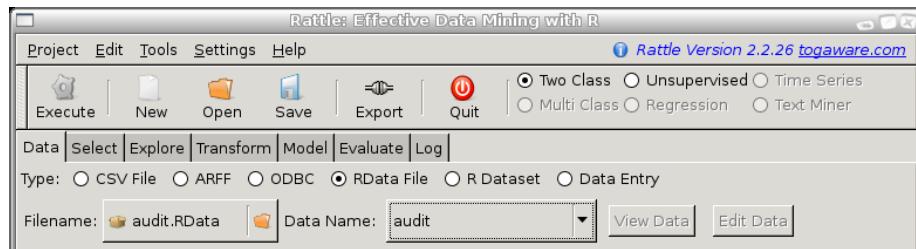


Figure 3.10: Loading an R binary data file.

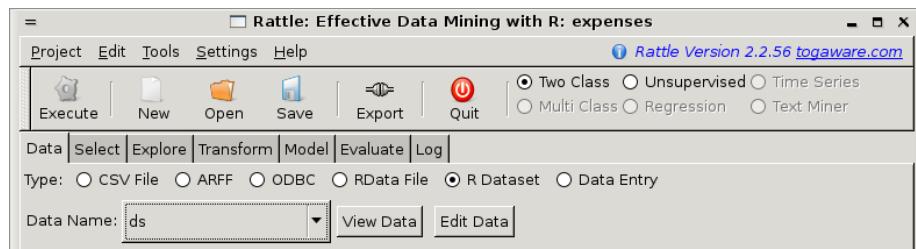


Figure 3.11: Loading an already defined R data frame

value.

3.6 R Data

Using the *RData File* option data can be loaded directly from a native R data file (usually with the .RData or .RData extension). Such files may contain multiple datasets (compressed) and you will be given an option to choose just one of the available datasets from the combo box.

3.7 R Dataset

Rattle can use a dataset that is already loaded into R (although it will take a copy of it, with memory implications). Only data frames are currently supported, and Rattle will list for you the names of all of the available data frames.

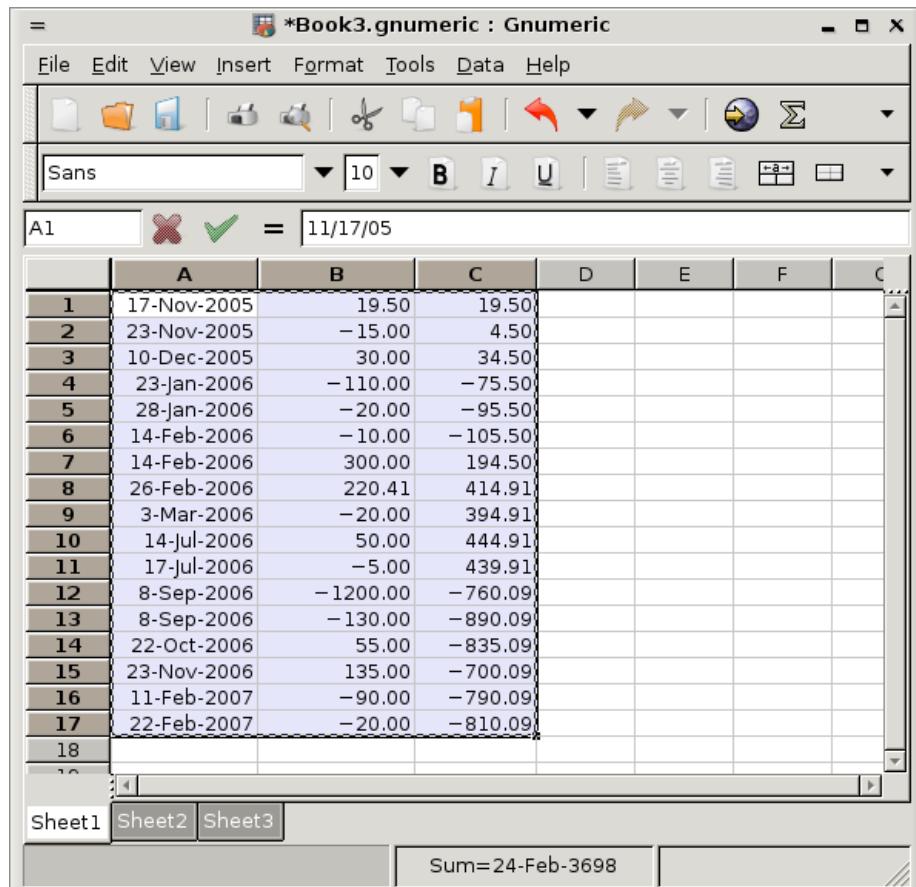


Figure 3.12: Selected region of a spreadsheet copied to the clipboard

The data frames need to be constructed in the same R session that is running Rattle (i.e., the same R Console in which you load the Rattle package). This provides much more flexibility in loading data into Rattle, than is provided directly through the actual Rattle interface.

This is useful if you have data in a spreadsheet (or any application really) and want to directly paste that data from a clipboard into R. Figure 3.12 illustrates a spreadsheet with three columns of data selected and copied to the clipboard (by the right mouse button menu selecting the Copy option). This data can then be loaded into R quite easily as illustrated in Listing 3.5. There, we also convert the date from a string, which is

```
> expenses <- read.table(file("clipboard"), header=TRUE)
> expenses$Date <- as.Date(expenses$Date, format="%d-%b-%Y")
> expenses
   Date Expense Total
1 2005-11-17  19.50  19.50
2 2005-11-23 -15.00   4.50
3 2005-12-10  30.00  34.50
4 2006-01-23 -110.00 -75.50
5 2006-01-28 -20.00 -95.50
6 2006-02-14 -10.00 -105.50
7 2006-02-14  300.00 194.50
8 2006-02-26 220.41 414.91
9 2006-03-03 -20.00 394.91
10 2006-07-14  50.00 444.91
11 2006-07-17 -5.00 439.91
12 2006-09-08 -120.00 319.91
13 2006-09-08 -130.00 189.91
14 2006-10-22  55.00 244.91
15 2006-11-23 135.00 379.91
16 2007-02-11 -90.00 289.91
17 2007-02-22 -20.00 269.91
```

Listing 3.5: Load data from clipboard into R

Togaware Watermark For Data Mining Survival

nothing more than a string, into a actual date data type.

We can then load this into Rattle directly, as in Figure 3.13.

As another example, you may want to load data from an SQLite database directly, and have this available in Rattle.

3.8 Data Entry

The Data Entry option provides a basic mechanism to manually enter data for use in Rattle. Of course, we would not want to do this for anything but a small dataset, but at least the option is there for some very simple exploration of Rattle without the overhead of loading some other dataset

	var1	var2	var3	var4
1	23	1	a	1
2	45	3	b	0
3	15	2	a	0
4	48	1	c	1
5	72	4	b	1
6	67	2	c	0
7				
8				
9				
10				
11				
12				
13				
14				
15				
16				
17				
18				

Figure 3.14: Data entry spreadsheet

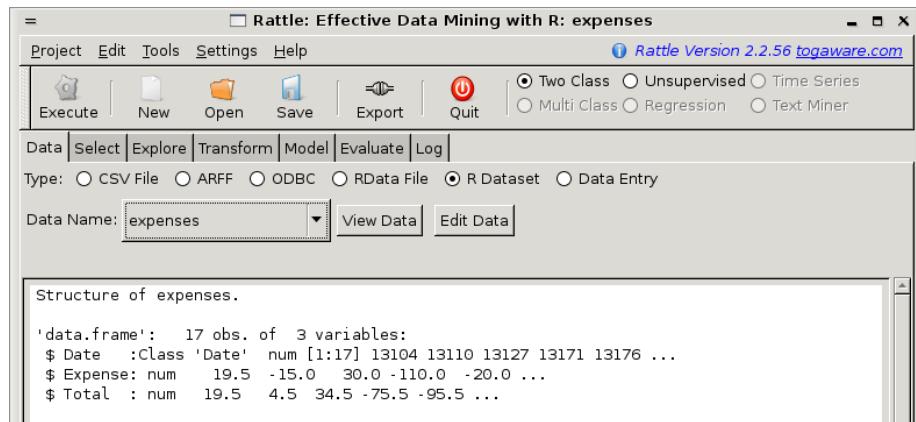


Figure 3.13: Loading an R data frame originally from the clipboard

into Rattle.

To use the Data Entry option,
select the radio button and click on Execute. A window will popup,
having an interface like a spreadsheet. We can now start out data entry.

Chapter 4

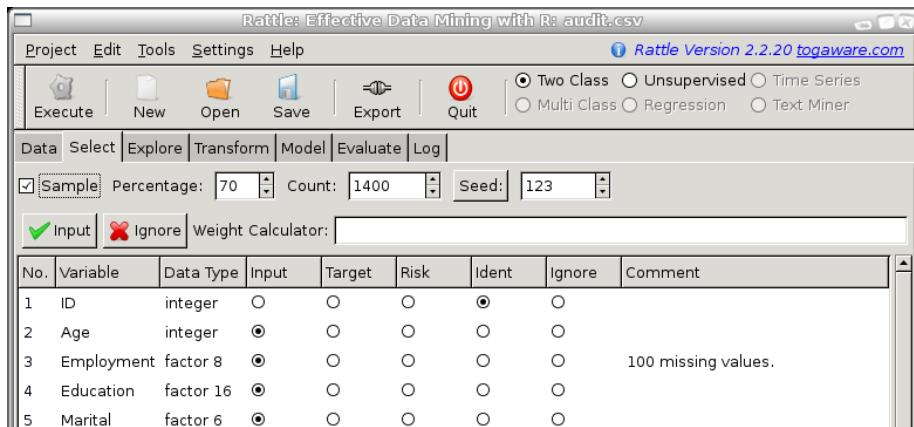
Selecting Data

The **Select** tab is used to select a subset of rows (entities) to include in the data mining, and to identify the role played by each of the variables in the dataset.

Remember, for any changes that we make to the **Select** tab to actually take effect we need to click the **Execute** button (or press F5 or choose the **Execute** menu item from the **Tools** menu.)

4.1 Sampling Data

The **Sample** option allows us to partition our dataset into a training dataset and a testing dataset, and to select different random samples if we wish to explore the sensitivity of our models to different data samples. Sampling is also useful when we have a very large dataset and want to obtain some initial insights quickly, whilst exploring the whole dataset may take hours.



Here we specify how we might partition the dataset for exploratory and modelling purposes. The default for Rattle is to build two subsets of the dataset: one is a training dataset from which to build models, while the other is used for testing the performance of the model. The default for Rattle is to use a 70% training and a 30% testing split, but you are welcome to turn sampling off, or choose other samplings. A very small sampling may be required to perform some explorations of the smaller dataset, or to build models using the more computationally expensive algorithms (like support vector machines).

R uses random numbers to generate samples, which may present a problem with regard repeatable modelling. This presents itself through the fact that each time the *sample* function is called we will get a different random sample. However, R provides the *set.seed* function to set a seed for the next random numbers it generates. Thus, by setting the seed to the same number each time you can be assured of obtaining the same sample.

Rattle allows you to specify the random number generator seed. By default, this is the number 123, but you can change this to get a different random sample on the next Execute. By keep the random number generator seed constant you can guarantee to get the same model, and by changing it you can explore the sensitivity of the modelling to different samples of the dataset.

Often in modelling we build our model on a training dataset and then test its performance on a test dataset.

4.2 Variable Roles

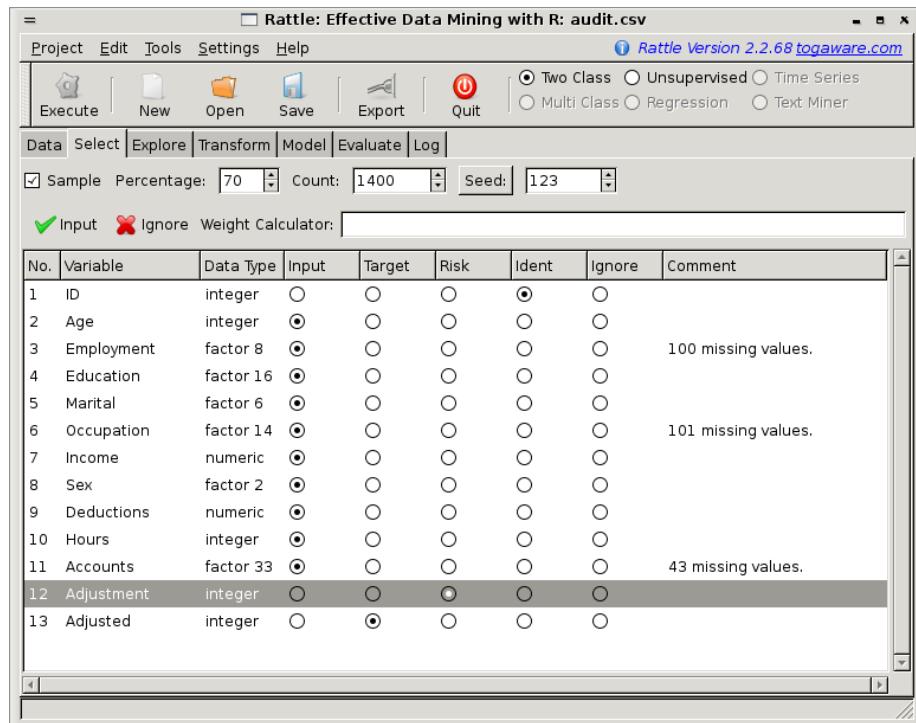


Figure 4.1: Select tab choosing Adjusted as a Risk variable.

Variables can be inputs to modelling, the target variable for modelling, the risk variable, an identifier, or an ignored variable. The default role for most variables is that of an Input (i.e., independent) variable. Generally, these are the variables that will be used to predict the value of a Target (or dependent) variable.

Variables with particular names will have a default role assigned for them. For example, if the variable name begins with `ID` then the default role is set to `Identifier`. Other special strings that are looked for at the beginning of the variable's name include:

<code>ID</code>	Identifier
<code>IGNORE</code>	Ignored
<code>RISK</code>	Risk measure

Further, if a variable name begins with `IMP_` (for imputed) then if the same variable name without the `IMP_` is found in the dataset then it will be set to be ignored by default.

Rattle also uses simple heuristics to guess at a Target role for one of the variables. Here we see that Adjusted has been selected as the target variable. In this instance it is correct. The heuristic involves examining the number of distinct values that a variable has, and if it has less than 5, then it is considered as a candidate. The candidate list is ordered starting with the last variable (often the last variable is the target), and then proceeding from the first onwards to find the first variable that meets the conditions of looking like a target.

Any numeric variables that have a unique value for each entity is automatically identified as an Ident. Any number of variables can be tagged as being an Ident. All Ident variables are ignored when modelling, but are used after scoring a dataset, being written to the resulting score file so that the cases that are scored can be identified.

Sometimes not all variables in your dataset should be used or may not be appropriate for a particular modelling task. For example, the random forest model builder does not handle categorical variables with more than 32 levels, so you may choose to Ignore Accounts. You can change the role of any variable to suit your needs, although you can only have one Target and one Risk.

For an example of the use of the Risk variable, see Section [7.2](#).

4.3 Automatic Role Identification

Special variable names can be used with data imported into Rattle (and in fact for any data used by Rattle) to identify their role. Any variable with a name beginning with `IGNORE_` will have the default role of Ignore. Similarly `RISK_` and `TARGET_`. Any variable beginning with `IMP_` is assumed to be an imputed variable, and if there exists a variable with the same name, but without the `IMP_` prefix, that variable will be marked as Ignore.

4.4 Weights Calculator

Togaware Watermark For Data Mining Survival

Togaware Watermark For Data Mining Survival

Chapter 5

Exploring Data

A key task in any data mining project is **exploratory data analysis** (often abbreviated as EDA), which generally involves getting a basic understanding of a dataset. Statistics, the fundamental tool here, is essentially about uncertainty—to understand it and thereby to make allowance for it. It also provides a framework for understanding the discoveries made in data mining. Discoveries need to be statistically sound and statistically significant—uncertainty associated with modelling needs to be understood.

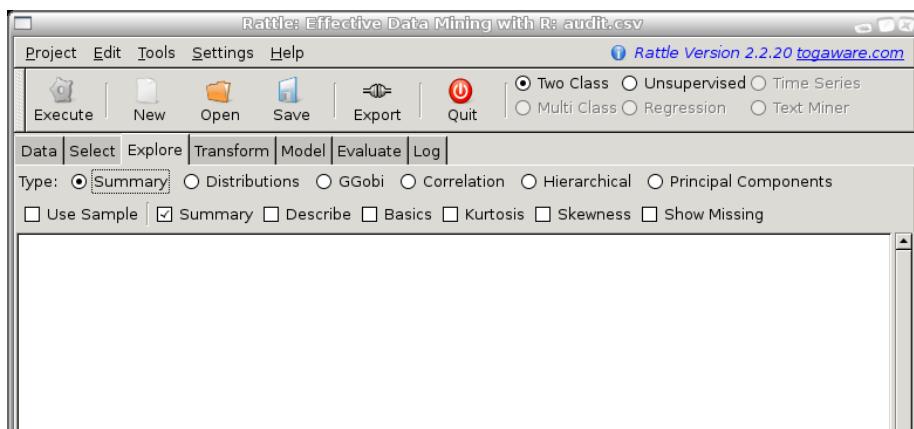
We explore the shape or distribution of our data before we begin mining. Through this exploration we begin to understand the “lay of the land,” just as a miner works to understand the terrain before blindly digging for gold. Through this exploration we may identify problems with the data, including missing values, noise and erroneous data, and skewed distributions. This will then drive our choice of tools for preparing and transforming our data and for mining it.

Rattle provides tools ranging from textual summaries to visually appealing graphical summaries, tools for identifying correlations between variables, and a link to the very sophisticated GGobi tool for visualising data. The Explore tab provides an opportunity to understand our data in various ways.

5.1 Summarising Data

While a picture might tell a thousand stories, textual summaries still play an important roll in our understanding of data. We saw a basic summary of our data after first loading the data into Rattle (page 31). The data types and the first few values for each of the variables are automatically listed. This is the most basic of summaries, and even so, begins to tell a story about the data. It is the beginnings of understanding the data.

Rattle's Summary option of the Explore tab provides a number of more detailed textual summaries of our data.



With the **Use Sample** check box we can choose to summarise the whole dataset, or just the training dataset. We might choose to only summarise the sample when the dataset itself is very large and the summaries take a long time to perform. We would usually not choose the sample option.

The rest of the check boxes of the **Summary** option allows us to fine tune what it is we wish to explore textually. We can choose to display one or many of the summary options. The first three—**Summary**, **Describe**, and **Basic**—are three alternatives that provide overall statistics for each variable (although the **Basics** option only summarises numeric variables). The final two, **Kurtosis** and **Skewness** provide specific measures of the characteristics of the data. These are separated out so that we can compare the kurtosis or skewness directly across a number of variables. These two measures both apply only to numeric data.

5.1.1 Summary

The **Summary** check box provides numerous measures for each variable, including, in the first instance, the minimum, maximum, median, mean, and the first and third quartiles. Generally, if the mean and median are significantly different then we would think that there are some entities with very large values in the data pulling the mean in one direction. It does not seem to be the case for Age but is for Income.

The screenshot shows the Rattle software interface version 2.2.26. The menu bar includes Project, Edit, Tools, Settings, Help, and a Rattle Version link. The toolbar has icons for Execute, New, Open, Save, Export, and Quit. The main window has tabs for Data, Select, Explore, Transform, Model, Evaluate, and Log. The Data tab is selected. The Type dropdown is set to Summary. The summary output for the 'audit.csv' dataset is displayed in the main pane:

```

Summary of the full dataset.

The data contains 141 entities with missing values.

(Hint: 25% of values are below 1st Quartile.)

Age Employment Education Marital
Min. :17.00 Private :1411 HsGrad :660 Absent :669
1st Qu.:28.00 Consultant: 148 College :442 Divorced :266
Median :37.00 PSLocal : 119 Bachelor :345 Married :917
Mean :38.62 SelfEmp : 79 Master :102 Married-spouse-absent: 22
3rd Qu.:48.00 PSSstate : 72 Vocational: 86 Unmarried : 67
Max. :90.00 (Other) : 71 Yr11 : 74 Widowed : 59
NA's : 100 (Other) :291

Occupation Income Sex Deductions
Executive :289 Min. : 146.1 Female: 632 Min. : 0.00
Professional:247 1st Qu.: 35079.4 Male :1368 1st Qu.: 0.00
Clerical :232 Median : 57885.7 Median : 0.00
Repair :225 Mean : 85896.3 Mean : 67.57
Service :210 3rd Qu.:119643.5 3rd Qu.: 0.00
(Other) :696 Max. :421362.7 Max. :2904.00
NA's :101

Hours Accounts Adjustment Adjusted
Min. : 1.00 UnitedStates:1804 Min. : -2194 Min. :0.0000
1st Qu.:38.00 Mexico : 43 1st Qu.: 0 1st Qu.:0.0000
Median :40.00 Philippines : 13 Median : 0 Median :0.0000
Mean :40.07 Vietnam : 8 Mean : 2006 Mean : 0.2315
3rd Qu.:45.00 China : 7 3rd Qu.: 0 3rd Qu.:0.0000
Max. :99.00 (Other) : 82 Max. :102937 Max. :1.0000
NA's : 43

```

Generated by Rattle 2007-03-22 21:22:23 gjw

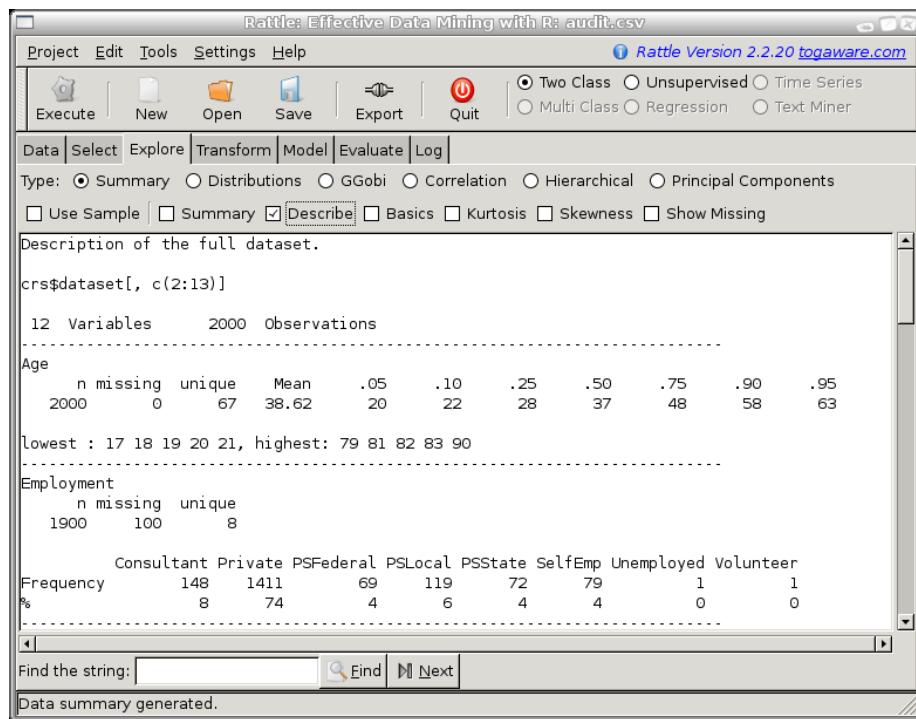
Find: [] Find Next

Data summary generated.

For categorical variables the

5.1.2 Describe

The **Describe** check box



Rattle: Effective Data Mining with R: audit.csv

Project Edit Tools Settings Help Rattle Version 2.2.20 togaware.com

Execute New Open Save Export Quit

Two Class Unsupervised Time Series

Multi Class Regression Text Miner

Data Select Explore Transform Model Evaluate Log

Type: Summary Distributions GGobi Correlation Hierarchical Principal Components

Use Sample Summary Describe Basics Kurtosis Skewness Show Missing

Description of the full dataset.

```
crs$dataset[, c(2:13)]
12 Variables      2000 Observations
-----
Age
  n missing unique   Mean    .05   .10   .25   .50   .75   .90   .95
  2000     0       67  38.62    20    22    28    37    48    58    63

lowest : 17 18 19 20 21, highest: 79 81 82 83 90
-----
Employment
  n missing unique
  1900     100      8

  Consultant Private PSFederal PSLocal PSState SelfEmp Unemployed Volunteer
Frequency      148      1411      69     119      72      79        1        1
%             8        74        4        6        4        4        0        0
-----
```

Find the string: Find Next

Data summary generated.

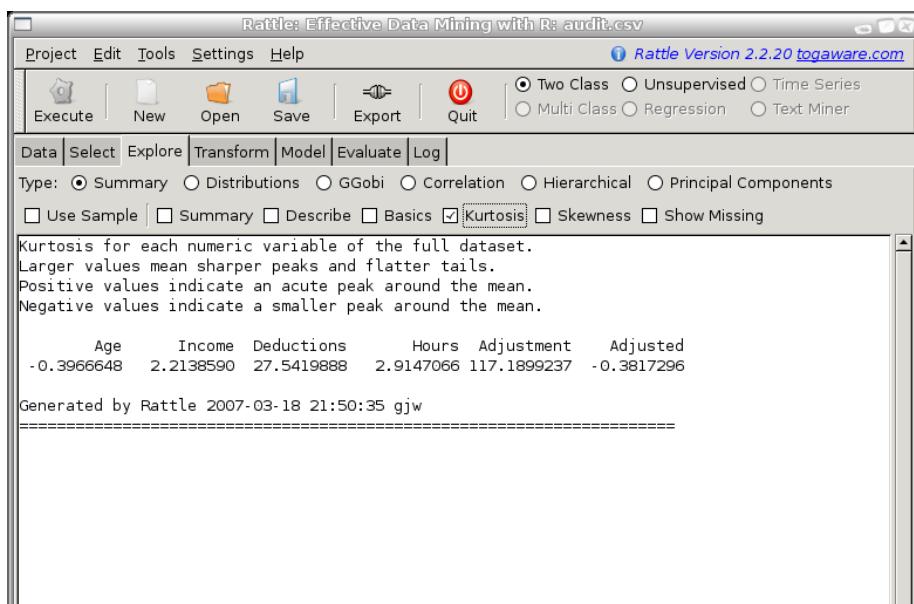
5.1.3 Basics

The Basics check box

The screenshot shows the Rattle interface for data mining with R. The title bar reads "Rattle: Effective Data Mining with R: audit.csv". The menu bar includes Project, Edit, Tools, Settings, Help, and a version notice "Rattle Version 2.2.20 togaware.com". The toolbar contains icons for Execute, New, Open, Save, Export, and Quit. A radio button group for "Type:" is set to "Summary". Below the toolbar, there are several checkboxes: "Use Sample" (unchecked), "Summary" (unchecked), "Describe" (unchecked), "Basics" (checked), "Kurtosis" (unchecked), "Skewness" (unchecked), and "Show Missing" (unchecked). A large text area displays "Basic statistics for each numeric variable of the full dataset." for the variable "\$Age". The statistics listed include: nobs (2000.000000), Nas (0.000000), Minimum (17.000000), Maximum (90.000000), 1. Quartile (28.000000), 3. Quartile (48.000000), Mean (38.622000), Median (37.000000), Sum (77244.000000), SE Mean (0.303764), LCL Mean (38.026272), UCL Mean (39.217728), Variance (184.545389), Stdev (13.584748), Skewness (0.499070), and Kurtosis (-0.396665). At the bottom of the text area, there is a search bar with "Find the string:" and "Find" and "Next" buttons, followed by the message "Data summary generated."

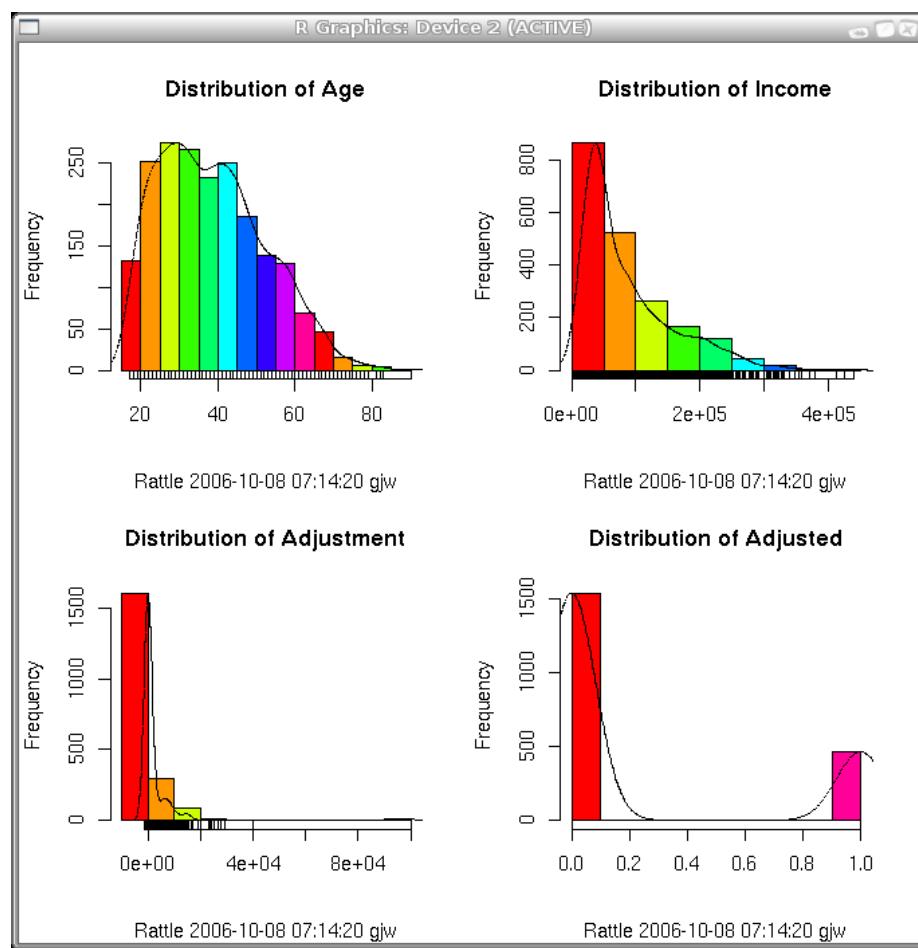
5.1.4 Kurtosis

The **kurtosis** is a measure of the nature of the peaks in the distribution of the data. A larger value for the kurtosis will indicate that the distribution has a sharper peak, as we can see in comparing the distributions of Income and Adjustment. A lower kurtosis indicates a smoother peak.



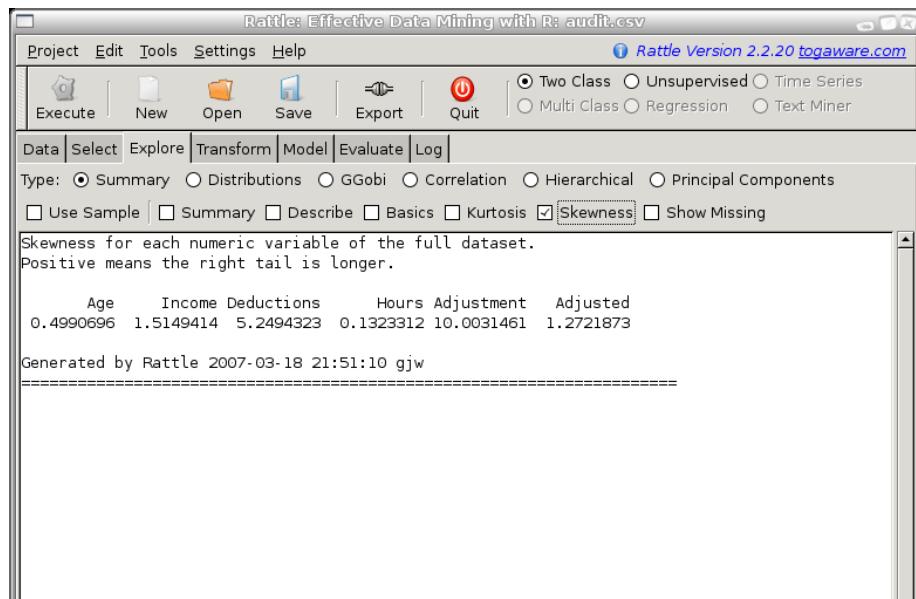
5.1 Summarising Data

53



5.1.5 Skewness

The **skewness** is a measure of how asymmetrical our data is distributed. A positive skew indicates that the tail to the right is longer, and a negative skew that the tail to the left is longer.



5.1.6 Missing

Missing values present challenges to data mining. The Show Missing check button of the Summary option of the Explore tab provides a summary of missing values in our dataset. Figure 5.1 illustrates the missing value summary. Such information is useful in understanding structure in the missing values.

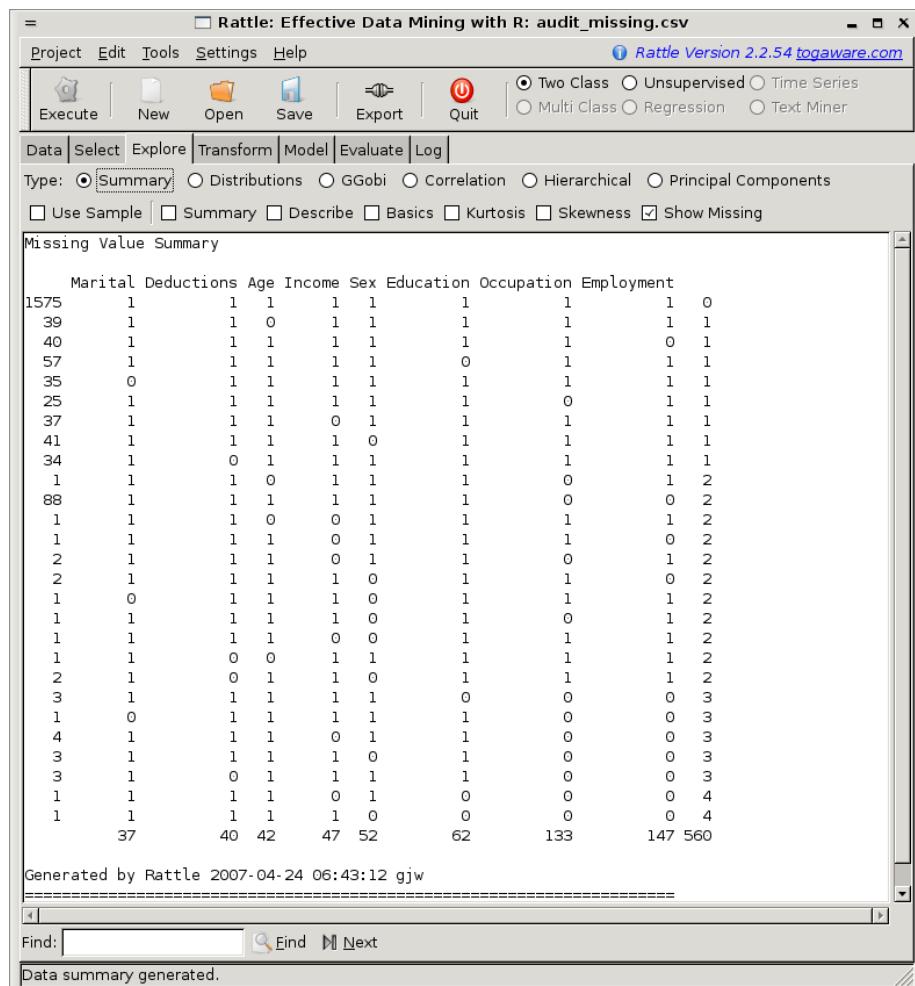


Figure 5.1: Missing value summary for a version of the *audit* modified to include missing values.

The missing value summary table is presented with the variables listed along the top. Each row corresponds to a pattern of missing values. A 1 indicates a value is present, whereas a 0 indicates a value is missing.

The right hand column records the number of entities that exhibit that pattern, so that the sum of this column should be equal to the number of entities in our dataset. The left hand column records the number of variables with missing values for each pattern. The final row records the number of missing values over the whole dataset for each of the variables, with the total number of missing values recorded at the bottom right.

The rows and columns are sorted according to the amount of missing data.

Generally, the first row records the number of entities that have no missing values, as is the case in Figure 5.1, where 1575 rows are complete.

The second row corresponds to a pattern of missing values for the variable `Age`. There are 39 entities that have just `Age` missing (and there are 42 entities that have `Age` missing, overall). This particular row's pattern has just a single variable missing, as indicated by the 1 in the final column.

The final row indicates that there are, for example, 37 missing values for the variable `Marital`, and that there are 560 missing values altogether in this dataset.

See Section 6.2 for dealing with missing values through imputation.

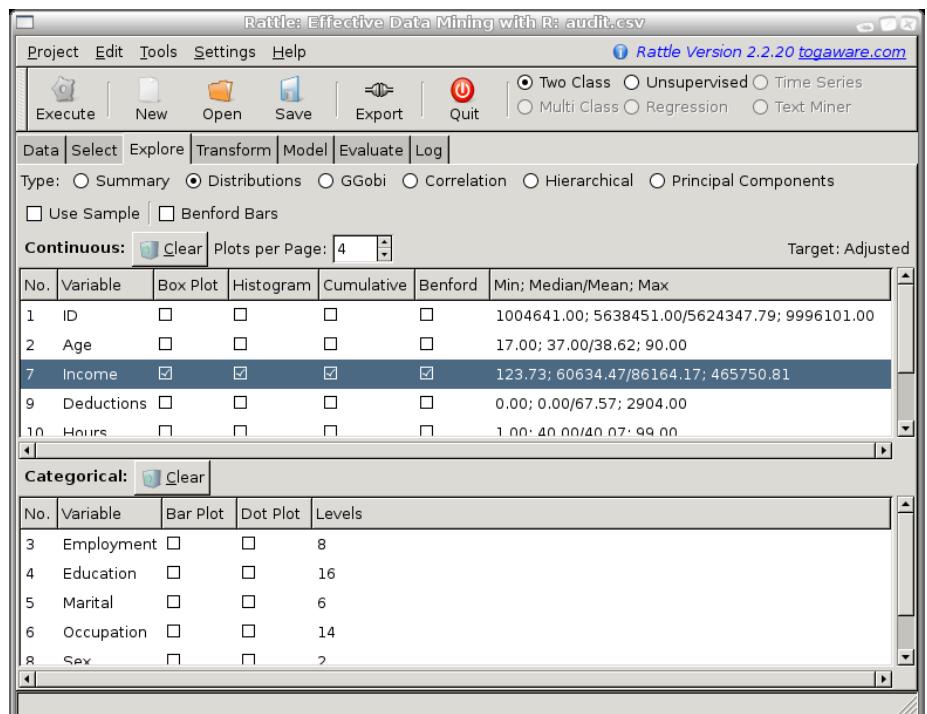
5.2 Exploring Distributions

It is usually a good idea to review the distributions of the values of each of the variables in your dataset. The Distributions option allows you to visually explore the distributions for specific variables.

Using graphical tools to visually investigate the data's characteristics can help our understanding the data, in error correction, and in variable selection and variable transformation.

Graphical presentations are more effective for most people, and Rattle provides a graphical summary of the distribution of the data with the Distribution option of the Explore tab.

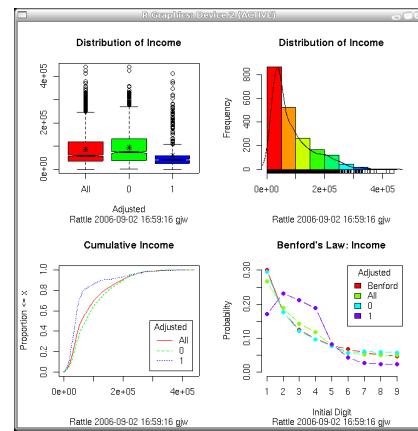
Visualising data has been an area of study within statistics for many years. A vast array of tools are available within R for presenting data visually and the topic is covered in detail in books in their own right, including [Cleveland \(1993\)](#) and Tufte.



By choosing the Distributions radio button you can select specific variables of interest, and display various distribution plots. Selecting many variables will lead to many plots being displayed, and so it may be useful to display multiple plots per page (i.e., per window) by setting the appropriate value in the interface. By default, four plots will be displayed per page or window, but you can change this to anywhere from 1 plot per page to 9 plots per page. If we are actually generating fewer plots than our selected number of plots per page, then we will see that the plots will fill up the window rather than leaving empty places. Thus the Plots per Page is really a maximum number of plots per page.

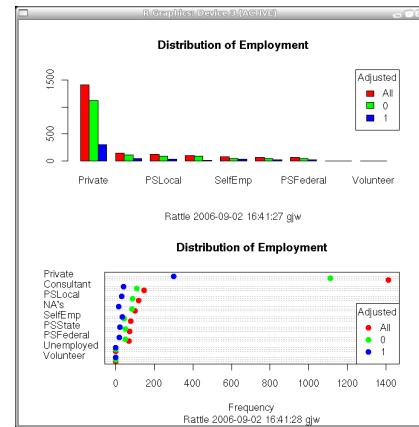
The Annotate check box can be checked to obtain additional annotations on the plots. For a Box Plot, for example, this will add the actual numeric values for the median and quartiles.

Here we illustrate a window with the default four plots. Four plots per page are useful, for example, to display each of the four different types of plots for a single continuous variable. Clockwise, they are the Box Plot, the Histogram, a Cumulative Function Plot, and a Benford's Law Plot. Because we have identified a target variable the plots include the distributions for each subset of entities associated with each value of the target variable, wherever this makes sense to do so (e.g., not for the histogram).



The box and whiskers plot identifies the median and mean of the variable, the spread from the first quartile to the third, and indicates the outliers. The histogram splits the range of values of the variable into segments and shows the number of entities in each segment. The cumulative plot shows the percentage of entities below any particular value of the variable. And the Benford's Law plot compares the distribution of the first digit of the numbers against that which is expected according to Benford's Law. Each of the plots shown here is explained in more detail in the following sections.

For categorical variables three types of plots are supported, more as alternatives than adding extra information: the Bar Plot, the Dot Plot, and the Mosaic Plot. Each plot shows, in its own way, the number of entities that have a particular value for the chosen variable. The bar and dot plots sort the categorical values from the most frequent to the least frequent value. In the example here, we can see that the value **Private** of the variable **Employment** is the most frequent, occurring over 1,400 times in this dataset.



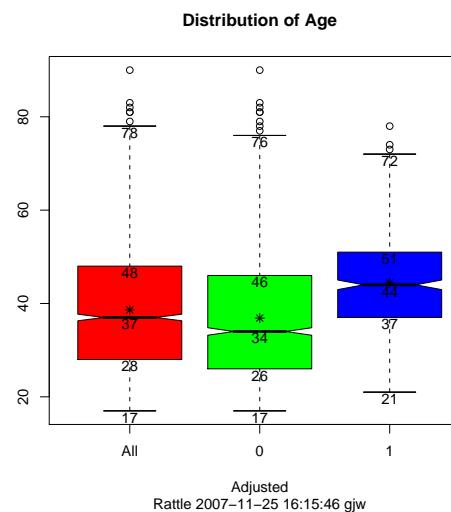
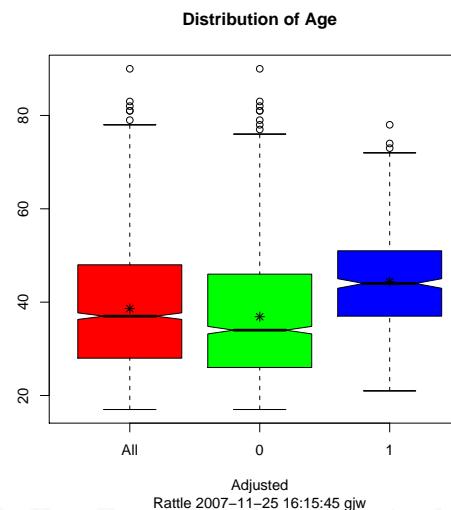
A bar plot uses vertical bars while the dot plot uses dots placed horizontally. The dot plot has more of a chance to list the actual values for the variable, whilst a bar plot will have trouble listing all of the values (as illustrated here). The mosaic plot is similar to the bar plot but each “bar” splits the values between the distinct values of the target variable (if one is chosen).

Each of the plots is explained in more detail in the following sections.

5.2.1 Box Plot

A **boxplot** (Tukey, 1977) (also known as a box-and-whisker plot) provides a graphical overview of how data is distributed over the number line. Rattle's Box Plot displays a graphical representation of the textual *summary* of data. It is useful for quickly ascertaining the skewness of the distribution of the data. If we have identified a Target variable, then the boxplot will also show the distribution of the values of the variable partitioned by values of the target variable, as we illustrate for the variable Age where Adjusted has been chosen as the Target variable.

The boxplot (which here is shown with the Annotate option checked) shows the **median** (which is also called the second **quartile** or the 50th **percentile**) as the thicker line within the box ($Age = 37$ over the whole population, as we can see from the Summary option's **Summary** check box). The top and bottom extents of the box (48 and 28 respectively) identify the upper quartile (the third quartile or the 75th percentile) and the lower quartile (the first quartile and the 25th percentile). The extent of the box is known as the **interquartile range** ($48 - 28 = 20$). The dashed lines extend to the



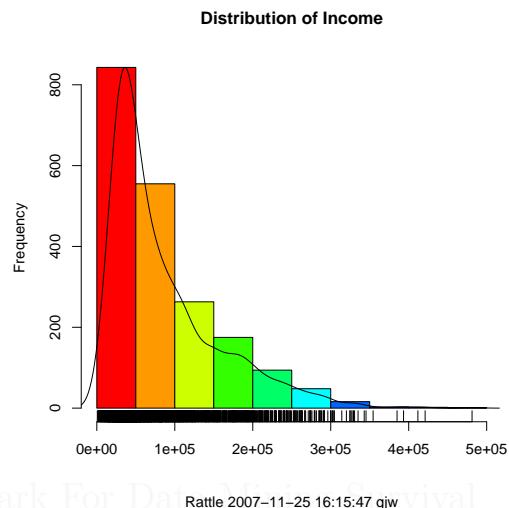
maximum and minimum data points that are no more than 1.5 times the interquartile range from the median. Outliers (points further than 1.5 times the interquartile range from the median) are then individually plotted (at 79, 81, 82, 83, and 90). The **mean** (38.62) is also displayed as the asterisk.

The notches in the box, around the median, indicate a level of confidence about the value of the median for the population in general. It is useful in comparing the distributions, and in this instance it allows us to say that all three distributions being presented here have significantly different means. In particular we can state that the positive cases (where *Adjusted* = 1) are older than the negative cases (where *Adjusted* = 0).

We note that the annotated box plot (as enable by checking the **Annotate** check box) does not attempt to place the annotations in any particularly optimal location, except a little below the point being annotated. They may be a little difficult to read at times. The user is at liberty to correct thus through replicating the plotting steps from the log window, but modifying the offsets in the display of the annotations.

5.2.2 Histogram

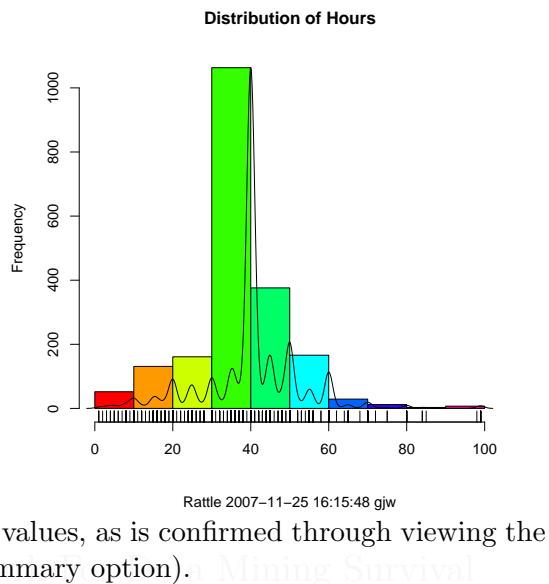
A **histogram** provides a quick and useful graphical view of the spread of the data. A histogram plot in Rattle includes three components. The first of these is obviously the coloured vertical bars. The continuous data in the example here (Distribution of Income) has been partitioned into ranges, and the frequency of each range is displayed as the bar. R is automatically choosing both the partitioning and how the x-axis is labelled here, showing x-axis points at 0, 10,000 (using scientific notation of $1e + 05$ which means 1×10^5 , or 10,000), and so on. Thus, we can see that the most frequent range of values is in the 0 – 5,000 partition. However, each partition spans quite a large range (a range of \$5,000).



The plot also includes a line plot showing the so called **density estimate** and is a more accurate display of the actual (at least estimated true) distribution of the data (the values of **Income**). It allows us to see that rather than values in the range 0 – 5,000 occurring frequently, in fact there is a much smaller range (perhaps 3,000 – 5,000) that occurs very frequently.

The third element of the plot is the so called *rug* along the bottom of the plot. The rug is a single dimension plot of the data along the number line. It is useful in seeing exactly where data points actually lay. For large collections of data with a relatively even spread of values the rug ends up being quite black, as is the case here, up to about \$25,000. Above about \$35,000 we can see that there is only a splattering of entities with such values. In fact, from the Summary option, using the **Describe** check box, we can see that the highest values are actually \$36,1092.60, \$38,0018.10, \$39,1436.70, \$40,4420.70, and \$42,1362.70.

This second plot, showing the distribution for the variable `Hours`, illustrates a more **normal distribution**. It is, roughly speaking, a distribution with a peak in the middle and diminishing on both sides, with regards the frequency. The density plot shows that it is not a very strong normal distribution, and the rug plot indicates that the data take on very distinct values (i.e., one would suggest that they are integer values, as is confirmed through viewing the textual summaries in the Summary option).

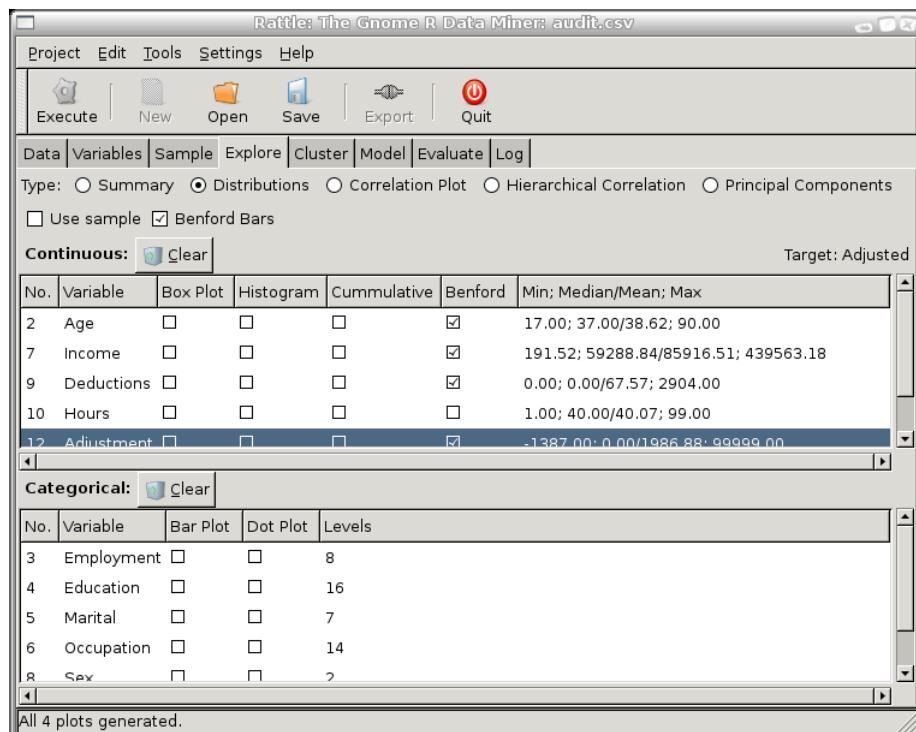


5.2.3 Cumulative Distribution Plot

CUMULATIVE PLOTS OF INCOME AND HOURS TO GO HERE.

Togaware Watermark For Data Mining Survival

5.2.4 Benford's Law



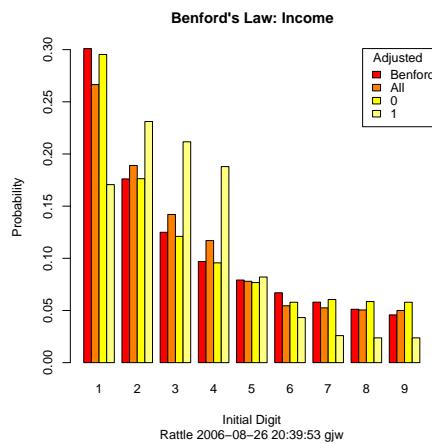
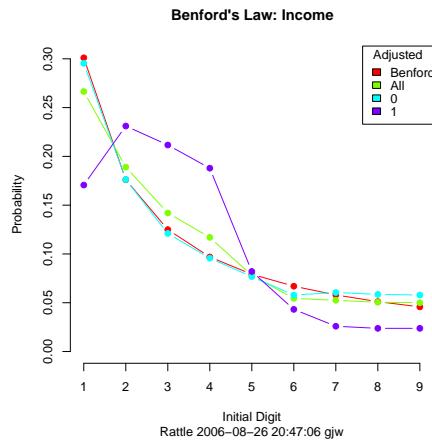
The use of **Benford's Law** has proven to be effective in identifying oddities in data. For example, it has been used for sample selection in fraud detection. Benford's law relates to the frequency of occurrence of the first digit in a collection of numbers. In many cases, the digit '1' appears as the first digit of the numbers in the collection some 30% of the time, whilst the digit '9' appears as the first digit less than 5% of the time. This rather startling observation is certainly found, empirically, to hold in many collections of numbers, such as bank account balances, taxation refunds, stock prices, death rates, lengths of rivers, and process that are described by what are called power laws, which are common in nature. By plotting a collection of numbers against the expectation as based on Benford's law, we are able to quickly see any odd behaviour in the data.

Benford's law is not valid for all collections of numbers. For example, people's ages would not be expected to follow Benford's Law, nor would telephone numbers. So use the observations with care.

You can select any number of continuous variables to be compared with Benford's Law. By default, a line chart is used, with the red line corresponding to the expected frequency for each of the initial digits. In this plot we have requested that Income be compared to Benford's Law. A Target variable has been identified (in the Variables tab) and so not only is the whole population's distribution of initial digits compared to Benford's Law, but so are the distributions of the subsets corresponding to the different values of the target variable. It is interesting to observe here that those cases in this dataset that required an adjustment after investigation (*Adjustment* = 1) conformed much less to Benford's Law than those that were found to require no adjustment (*Adjustment* = 0). In fact, this latter group had a very close conformance to Benford's Law.

By selecting the Benford Bars option a bar chart will be used to display the same information. The expected distribution of the initial digit of the numbers under consideration, according to Benford's Law, is once again shown as the initial red bar in each group. This is followed by the population distribution, and then the distribution for each of the sub-populations corresponding to the value of the Target variable. The bar chart again shows a very clear differentiation between the adjusted and non-adjusted cases.

Some users find the bar chart presentation more readily conveys the information, whilst many prefer the less clutter and increased clarity of



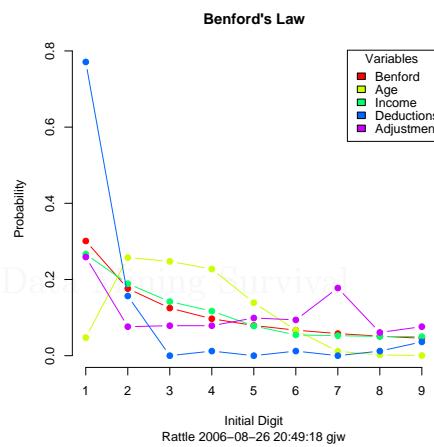
the line chart. However, a bar chart is useful if when you display a line chart you can not see all of the lines because they overlap. The bar chart will show all of the bars.

Regardless of which you prefer, Rattle will generate a single plot for each of the variables that have been selected for comparison with Benford's Law.

In the situation where no target variable has been identified (either because, for the dataset being explored, there is no target variable or because the user has purposely not identified the target variable to Rattle) and where a line chart, rather than a bar chart, is requested, the distribution of all variables will be displayed on the one plot. This is the case here where we have chosen to explore Age, Income, Deductions, and Adjustment.

This particular exploration of Benford's Law leads to a number of interesting observations. In the first instance, the variable *Age* clearly does not conform. As mentioned, age is not expected to conform since it is a number series that is constrained in various ways. In particular, people under the age of 20 are very much under-represented in this dataset, and the proportion of people over 50 diminishes with age.

The variable *Deductions* also looks particularly odd with numbers beginning with '1' being way beyond expectations. In fact, numbers beginning with '3' and beyond are very much under-represented, although, interestingly, there is a small surge at '9'. There are good reasons for this. In this dataset we know that people are claiming deductions of less than \$300, since this is a threshold in the tax law below which less documentation is required to substantiate the claims. The surge at '9' could be something to explore further, thinking perhaps that clients committing fraud may be trying to push their claims as high as possible (although there is really no need, in such circumstances, to limit oneself, it would



seem, to less than \$1000).

By exploring this single plot (i.e., without partitioning the data according to whether the case was adjusted or not) we see that the interesting behaviours we observed with relation to *Income* have disappeared. This highlights a point that the approach of exploring Benford's Law may be of most use in exploring the behaviours of particular sub-populations.

Note that even when no target is identified (in the Variables tab) and the user chooses to produce Benford Bars, a new plot will be generated for each variable, as the bar charts can otherwise become quite full.

Other Digits

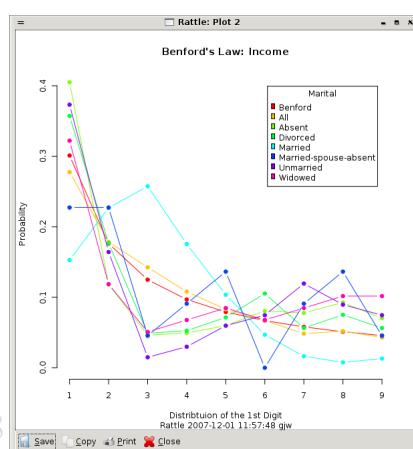
Benford's Law primarily applies to the first digit of the numbers. A similar, but much less strong, law also applies to the second, third and fourth digits. A common mathematical formula has been developed that generalises the distribution for the first, and the distributions converge to a flat distribution the further we go.

For the second digit, for example, the expected frequency of the digit 1 is 11.4%, compared to the digit 2's 10.9%. As we proceed to the third and fourth and so on, each has an expected frequency pretty close to 0.1 (or 10%), indicating they are all generally equally likely.

Even though the distributions become flat, it can still be useful to plot the actual distribution in order to explore for oddities in our data. Rattle allows us to plot up to the ninth digit.

Stratified Benford Plots

We often want to stratify our data (that is, split it up into subgroups in some way). For example, in fraud investigations we might split our data up into groups associated with different geographic regions, or different auditors, etc. Suppose we are con-



sidering accounts payable data where each record is a payment and there are, say, ten individuals who sign off on the invoices. We can choose in the Select tab the variable that identifies the individuals who are signing off as the Target variable.

The plot here illustrates the idea using the *audit* dataset. Here, we have chosen **Marital** to have the role as a Target variable (doing this in the Select tab). Then we have asked for a Benford plot of the **Income** variable, and we can see that the plot is stratified over the possible values for the **Marital** variable.

To stratify on more than two categorical variables requires a little extra work. Rattle does not allow selecting more than a single target! However, under the Transform tab, under the Remap option (Section 6.3.3, page 95), you can "join" two categorical variables into one and then set this combined categorical as your target variable.

This could be useful when, using the accounts payable example again, we have a person signing off the invoices and another person issuing the invoices, and we wish to explore whether there are any patterns through the combination of these two.

That is, the person signing off invoices might only be manipulating those invoices issued by a specific individual. Thus, re-mapping these two categorical variables into a single combined categorical variable will allow us to explore this relationship.

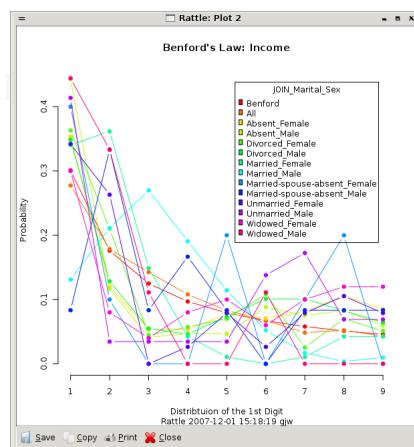


Figure 5.2: Benford stratified by Marital and Gender.

5.2.5 Bar Plot

XXXX

5.2.6 Dot Plot

XXXX

5.2.7 Mosaic Plot

A mosaic plot is an effective way to visualise the distribution of the values of a variable over the different values of another variable. Often, this second variable is the target variable that we are interested in (e.g., `Adjusted` in our *audit*).

The example in Figure 5.3 displays the relationship between the input variable `Age` and target variable `Adjusted`. `Age` is a numeric variable, so for it to make sense in a mosaic plot we need to bin it so that it becomes categorical, which can be accomplished using the `Remap` option of the `Transform` tab. In this case we have transformed `Age` into 6 quantiles (equal sized groups).

The plot tells us that XXXX

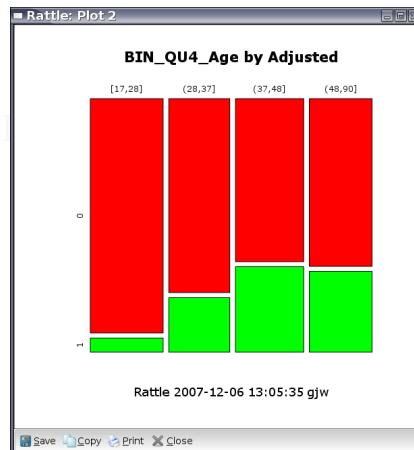
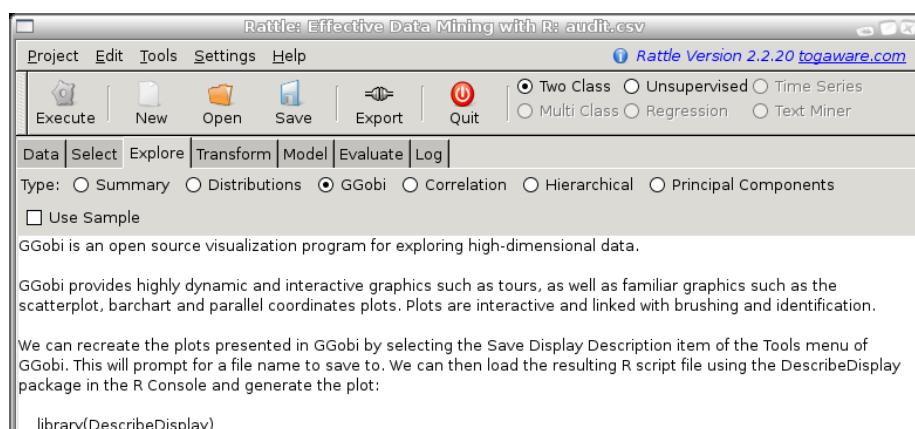


Figure 5.3: Mosaic plot of `Age` by `Adjusted`.

5.3 Sophisticated Exploration with GGobi

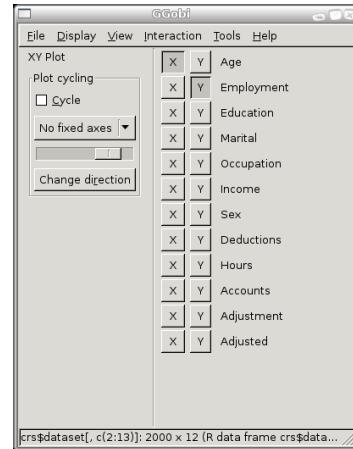
GGobi provides quite sophisticated visualisation tools as an adjunct to Rattle basic visualisations provided. To use the GGobi option the GGobi application will need to be installed on your system. GGobi runs under GNU/Linux, OS/X, and MS/Windows and is available for download from <http://www.ggobi.org/>.



GGobi is very powerful indeed, and here we only cover the basic functionality. With GGobi we are able to explore high-dimensional data through highly dynamic and interactive graphics such as tours, scatterplots, bar-charts and parallel coordinates plots. The plots are interactive and linked with brushing and identification. The available functionality is extensive, but includes being able to review entities with low or high values on particular variables and to view values for other variables through brushing in linked plots. Panning and zooming is supported. Data can be rotated in 3D, and we can tour high dimensional data through 1D, 2D, and 2x1D projections, with manual and automatic control of projection pursuits.

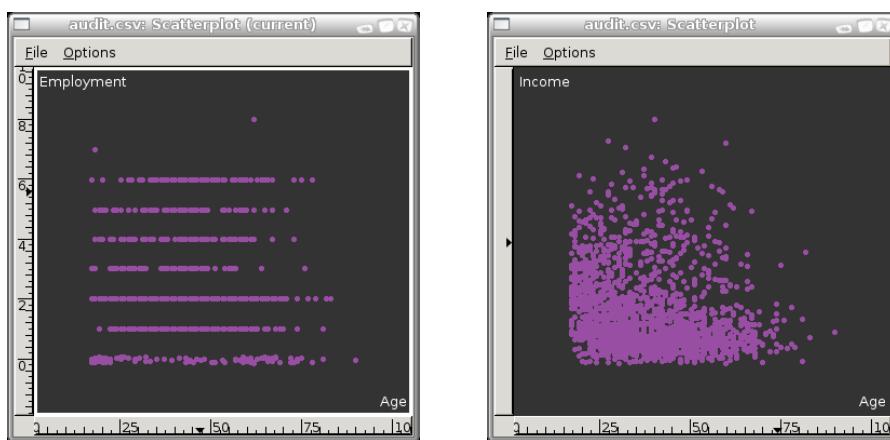
5.3.1 Scatterplot

When you startup GGobi (Execute the GGobi option) two windows will appear: one to control the visualisations and the other to display the default visualisation (a two variable scatterplot). The control window is as displayed to the right. It is a basic window with menus that provide the overall control of the visualisations. Below the menu bar you will see XY Plot which tells us that we are displaying a two variable scatterplot. On the right hand side is a list of the variables from your dataset, together with buttons to choose which variable to plot as the X and the Y.



By default, the first (**Age**) and second (**Employment**) are chosen. You can choose any of your variables to be the X or the Y by clicking the appropriate button. This will change what is displayed in the plot.

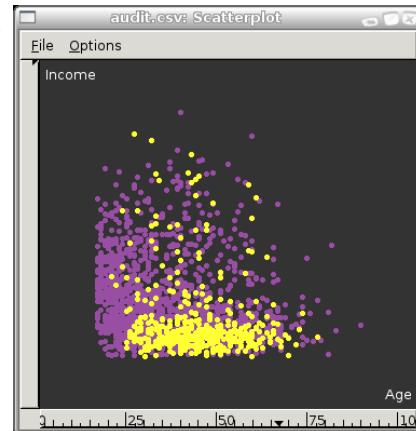
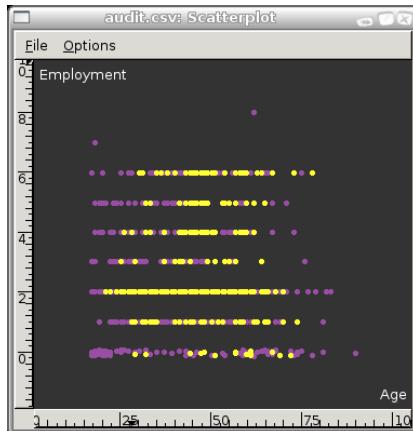
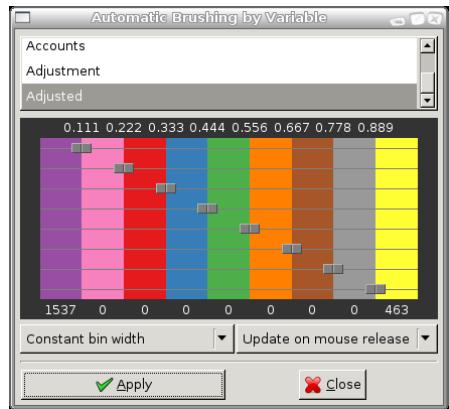
From the Display menu you can choose a New Scatterplot Display so that you can have two (or more) plots displayed at a time. At any one time just one plot is the current plot (as indicated in the title) and you can make a plot current by clicking in it.



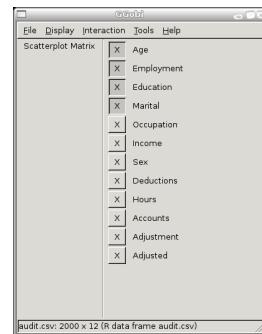
5.3 Sophisticated Exploration with GGobi

73

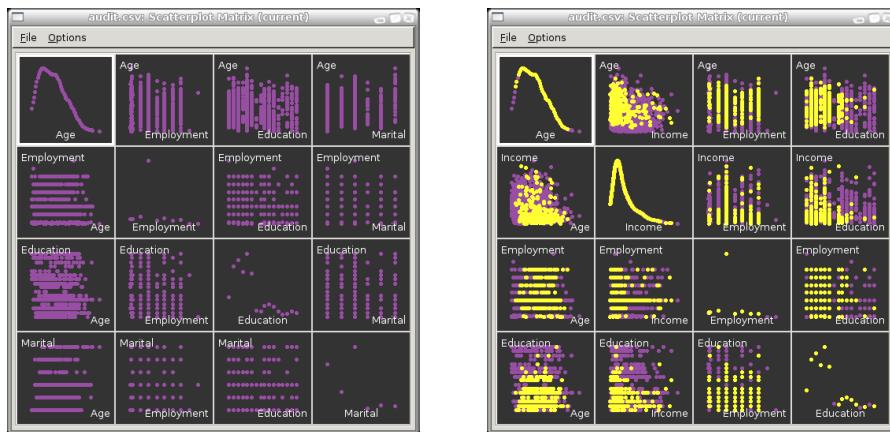
For our purposes we are usually most interested in the relationship between the values of the variables for entities that have an **Adjusted** value of 1 or 0. We can have these highlighted in different colours very easily. From the Tools menu choose Automatic Brushing. From the variables list at the top of the resulting popup window choose **Adjusted**. Now click on the Apply button and you will see that the 1,537 points that have a value of 0 for **Adjusted** remain purple, whilst those 463 entities that have a value of 1 are now yellow. This will apply to all displayed plots.



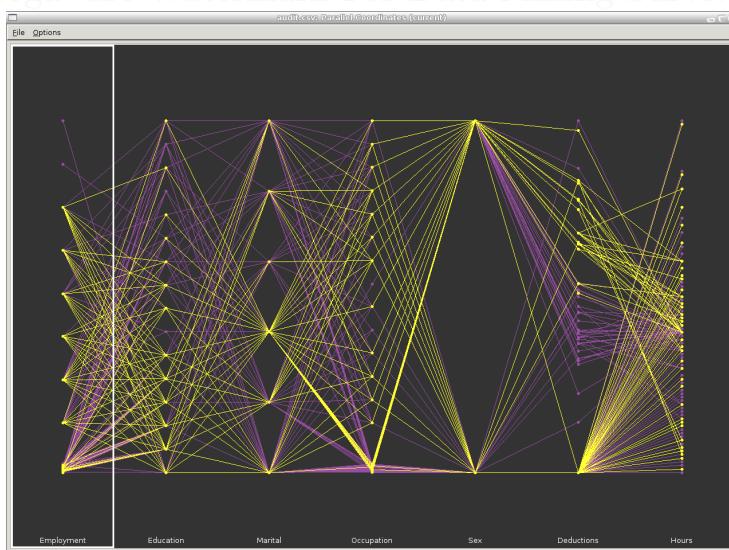
The Display menu provides a number of other options. The Scatterplot Matrix, for example, can be used to display a matrix of scatterplots across many variables at the one time. By default, the first four variables are displayed, as illustrated here, but we can add and remove variables by selecting the appropriate buttons in the control window (which is now displaying only the choice of X variables. You can also use the Automatic Brushing that we illustrated above to highlight



the adjusted cases. Such matrix scatterplots are effective in providing an overview of the distributions of our data.



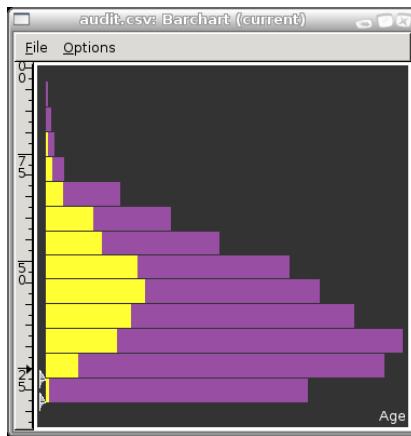
INCLUDE PARALLEL COORDINATES



INCLUDE BAR CHART PLUS THE INTERACTIVE BINNING

The thing to note here is the two arrows down the bottom left side of the plot. Drag these around to get different width bars.

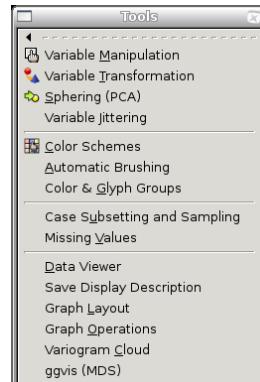
A scatterplot over very many points will sometimes be solid black and



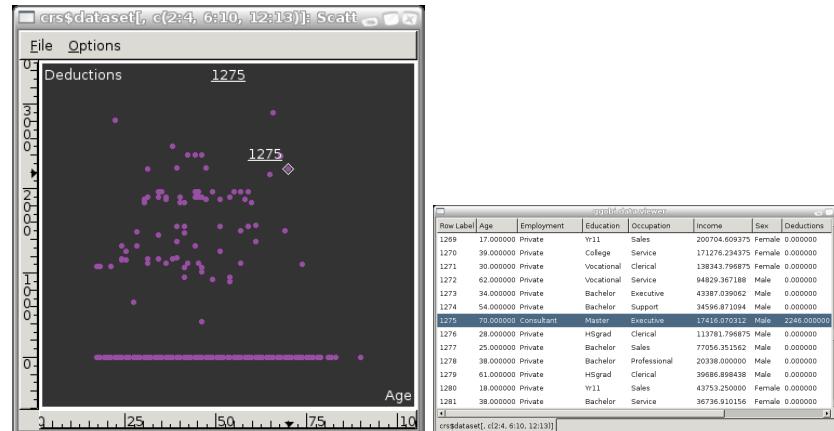
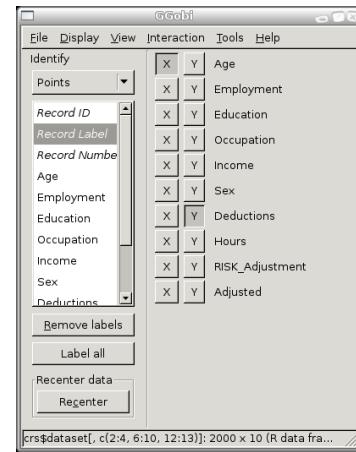
shows little useful information. In these cases a bagplot may be useful.

5.3.2 Data Viewer: Identifying Entities in Plots

Often we are interested in viewing the actual data/entities associated with points in our plots. The Tools menu provides many options and operations for visualising our data. In particular it provides access to a number of GGobiplugins. Some basic information about plugins is available from the Help menu, selecting the About Plugins item. The actual plugin we are interested in is the Data Viewer. Selecting this item from the Tools menu will display a textual view of the data, showing the row numbers and the column names, and allowing us to sort the rows by clicking on particular column names.



To make most use of the Data Viewer we will want to use the Identify option available under the Interaction menu. This will change the Control window to display the Identify controls as shown here. The Data Viewer is then linked to the Plot, allowing us to select a particular row in the Data Viewer to have the corresponding entity identified in the current Plot. Similarly, we can mouse over the Plot to have the individual points identified (with their row number) as well as displaying to the entity within the Data Viewer. Within the Plot display we can also right mouse button a point to have it's row number remain within the plot (useful when printing to highlight particular points). The right mouse button on the same point will remove the row number display.



5.3.3 Other Options

The Variable Manipulation option allows variables to be rescaled, cloned and manipulated and new variables to be created. The Transform Variables option allows us to view various transformations of the variables, including Log transforms, rescaling, and standardisation. Sphering performs a visual principal components analysis. The Color Scheme option

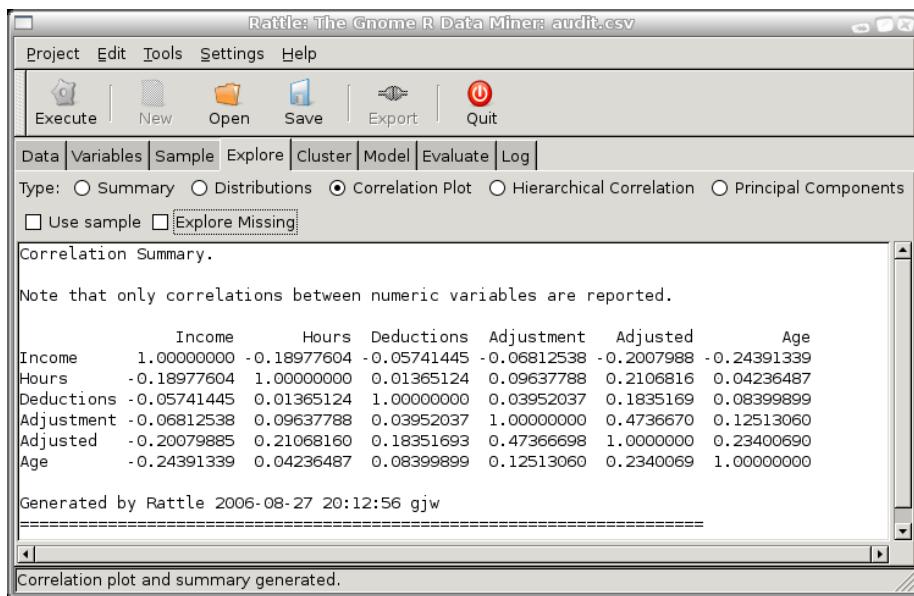
allows us to change the colours used in our plots. Automatic Brushing will colour various ranges of the values of each variable. Subsetting and Sampling allows us to choose subsets of the whole dataset using different methods, including random sampling, selection of blocks of data, every n th entity, and several others. There are also options to specify Graph Layout and Graph Options.

5.3.4 Further GGobi Documentation

We have only really scratched the surface of using GGobi here. There is a lot more functionality available, and whilst the functionality that is likely to be useful for the data miner has been touched on, there is a lot more to explore. So do explore the other features of GGobi as some will surely be useful for new tasks.

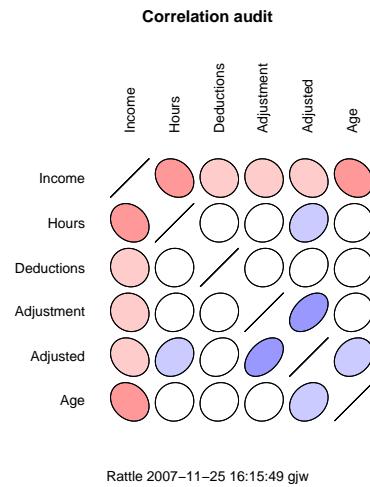
A full suite of documentation for GGobi is available from the GGobi web site at <http://www.ggobi.org/>, including a tutorial introduction and a complete book. These provide a much more complete treatise of the application.

5.4 Correlation Analysis



A correlation plot will display correlations between the values of variables in the dataset. In addition to the usual correlation calculated between values of different variables, the correlation between missing values can be explored by checking the **Explore Missing** check box.

The first thing to notice for this correlation plot is that only the numeric variables appear. Rattle only computes correlations between numeric variables at this time. The second thing to note about the graphic is that it is symmetric about the diagonal. The correlation between two variables is the same, irrespective of the order in which we view the two variables. The



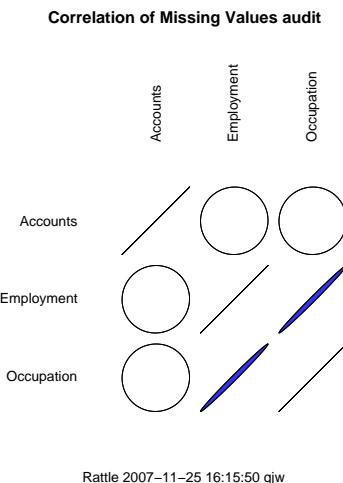
third thing to note is that the order of the variables does not correspond to the order in the dataset, but to the order of the strength of any correlations, from the least to the greatest. This is done simply to achieve a more pleasing graphic which is easier to take in.

We interpret the degree of any correlation by both the shape and colour of the graphic elements. Any variable is, of course, perfectly correlated with itself, and this is reflected as the diagonal lies on the diagonal of the graphic. Where the graphic element is a perfect circle, then there is no correlation between the variables, as is the case in the correlation between Hours and Deductions—although in fact there is a correlation, just a very weak one.

The colours used to shade the circles give another (if perhaps redundant) clue to the strength of the correlation. The intensity of the colour is maximal for a perfect correlation, and minimal (white) if there is no correlation. Shades of red are used for negative correlations and blue for positive correlations.

By selecting the **Explore Missing** check box you can obtain a correlation plot that will show any correlations between the missing values of variables. This is particularly useful to understand how missing values in one variable are related to missing values in another.

We notice immediately that only three variables are included in this correlation plot. Rattle has identified that the other variables in fact have no missing values, and so there is no point including them in the plot. We also notice that a categorical variable, Accounts, is included in the plot even though it was not included in the usual correlation plot. In this case we can obtain a correlation for categorical variables since we only measure missing and presence of a value, which is easily interpreted as numeric.



The graphic shows us that Employment and Occupation are highly correlated in their presence of missing values. That is, when Employment has a missing value, so does Occupation, and vice versa, at least in general. The actual correlation is 0.995 (which can be read from the Rattle text view window), which is very close to 1.

On the other hand, there is no (in fact very little at 0.013) correlation between Accounts and the other two variables, with regard missing values.

It is important to note that the correlations showing missing values may be based on very small samples, and this information is included in the text view of the Rattle window. For example, in this example we can see that there are only 100, 101, and 43 missing values, respectively, for each of the three variables having any missing values. This corresponds to approximately 5%, 5%, and 2% of the entities, respectively, having missing values for these variables.

```

Rattle: The Gnome R Data Miner: audit.csv
Project Edit Tools Settings Help
Execute New Open Save Export Quit
Data Variables Sample Explore Cluster Model Evaluate Log
Type:  Summary  Distributions  Correlation Plot  Hierarchical Correlation  Principal Components
 Use sample  Explore Missing
Missing Values Correlation Summary.
Note that only correlations between numeric variables are reported.
      Accounts Employment Occupation
Accounts 1.0000000 0.01344443 0.01304277
Employment 0.01344443 1.0000000 0.99477530
Occupation 0.01304277 0.99477530 1.0000000

Count of missing values:
Occupation Employment Accounts
      101        100       43

Percent missing values:
Occupation Employment Accounts
      5.05       5.00      2.15

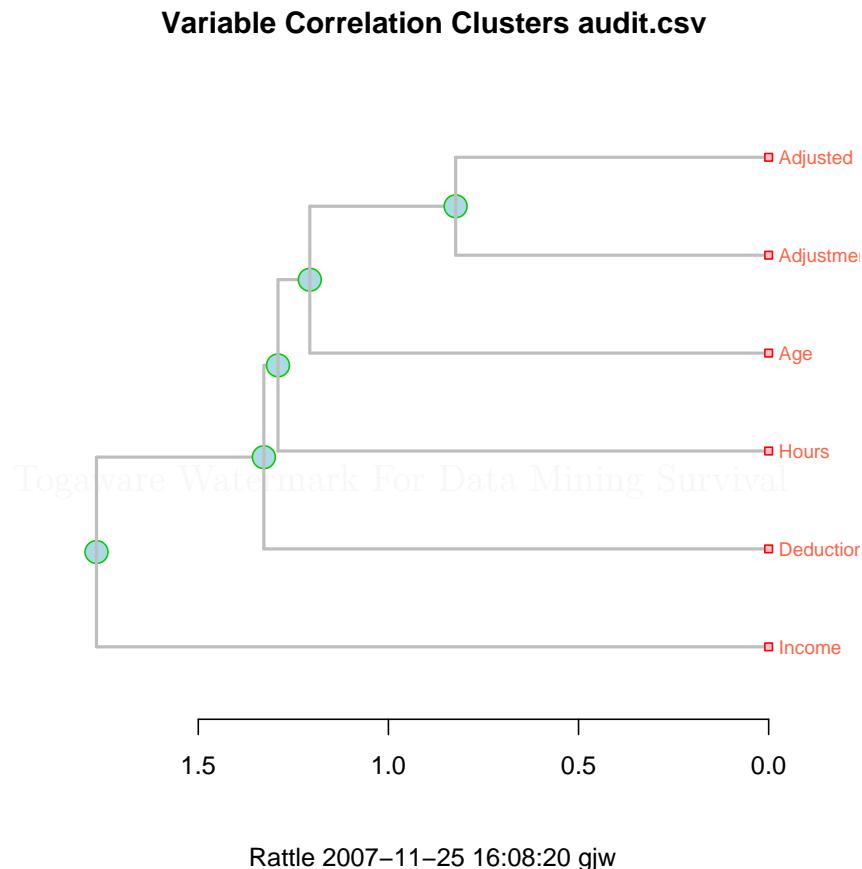
Generated by Rattle 2006-08-27 19:52:39 gjw
=====
Correlation plot and summary generated.

```

Rattle uses the default R correlation calculation known as Pearson's correlation, a common measure of correlation.

Togaware Watermark For Data Mining Survival

5.4.1 Hierarchical Correlation



5.4.2 Principal Components

5.5 Single Variable Overviews

Chapter 6

Transforming Data

The **Transform** tab provides numerous options for transforming our datasets. Cleaning our data and creating new features from the data occupies much of our time as data miners. There is a myriad of approaches, and a programming language like R supports them all. Through the Rattle user interface we can perform some of the more common transformations. This includes normalising our data, filling in missing values, turning numeric variables into categorical variables, and vice versa, dealing with outliers, and removing variables or entities with missing values.

In this chapter we introduce the various transformations supported by Rattle. Transformations are not always appropriate and so we indicate where they might be applicable as well providing warnings about the different approaches, particularly in the context of imputation, which can significantly alter the distribution of our datasets.

In tuning our dataset to suit our needs, we do often transform it in many different ways. Of course, once we have transformed our dataset, we will want to save the new version. After working on our dataset through the **Transform** tab we can save the data through the **Export** button. We will be prompted for a CSV file into which the current transformation of the dataset will be saved. In fact, this is the same save operation as available through the **Export** button on the **Data** and **Select** tabs.

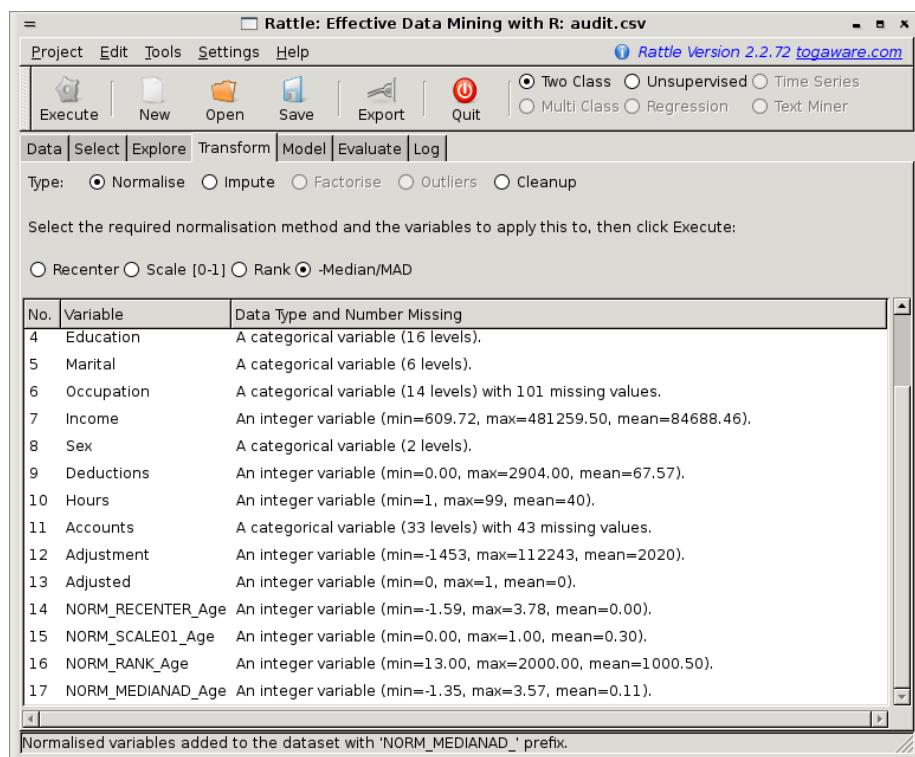


Figure 6.1: Transform options.

6.1 Normalising Data

Different model builders require different characteristics of the data from which the models will be built. For example, when building a clustering using any kind of distance measure, we may need to normalise the data. Otherwise, a variable like Income will overwhelm a variable like Age, when calculating distances. A distance of 10 “years” may be more significant than a distance of \$10,000, yet, 10000 swamps 10 when they are added together, as would be the case by calculating distances.

In these situations we will want to **Normalise** our data. The types of normalisations (available through the **Normalise** option of the **Transform** tab) we may want to perform include re-centering and rescaling our data to be around zero (**Recenter**), rescaling our data to be in the range from 0 to 1 (**Scale [0,1]**), convert the numbers into a rank ordering (**Rank**), and finally, to do a robust rescaling around zero using the median (-**Median/MAD**). Figure 6.2 displays the interface.

We can see in Figure 6.2 the approach we take to normalising (and to transforming) our data. The original data is not modified. Instead, a new variable is created with a prefix added to the variable’s name that indicates the kind of transformation. As we can see in the figure, the prefixes are **NORM_RECENTER_**, **NORM_SCALE01_**, **NORM_RANK_**, and **NORM_MEDIANAD_**.

We can see the effect of the four normalisations in comparing the histogram of the variable, Age, in Figure 6.3, with the four plots in Figure 6.4 for the corresponding four normalisations.

6.1.1 Recenter

A common normalisation is to recenter and rescale our data. The simplest approach to do this is to subtract the mean value of a variable from each entity’s value of the variable (to recenter the variable) and to then divide the values by the standard deviation, which re-scales the variable back to a range of a few integer values around zero.

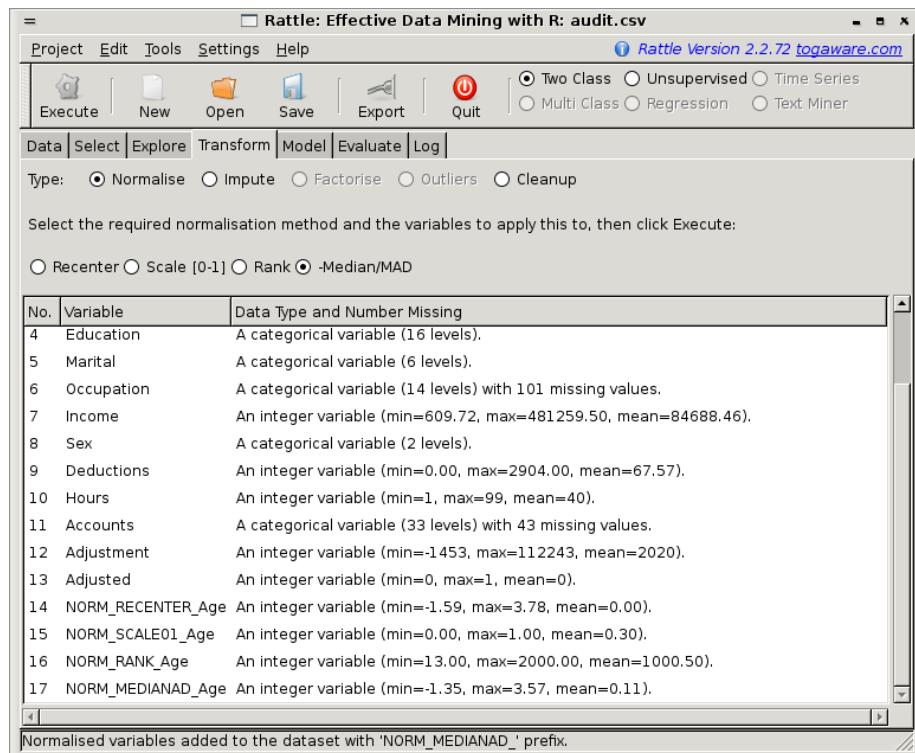


Figure 6.2: Selection of normalisations.

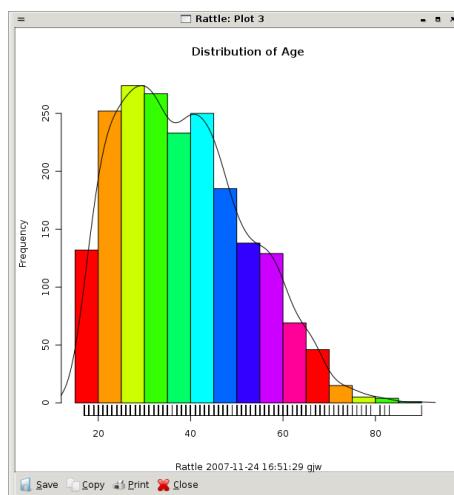


Figure 6.3: Normalisations of Age.

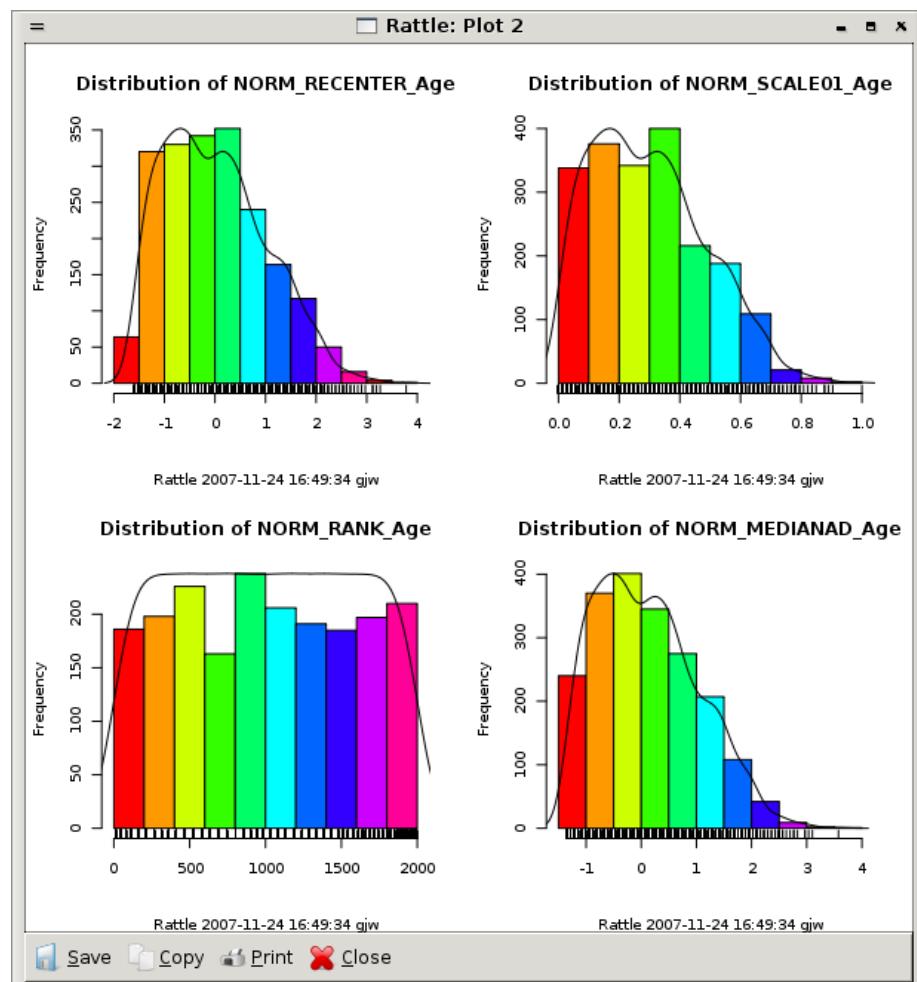


Figure 6.4: Normalisations of Age.

6.1.2 Scale [0,1]

Another common requirement is to remap our data to the $[0, 1]$ range.

6.1.3 Rank

This option will convert the values into a rank.

6.1.4 Median/MAD

This option is regarded as a robust version of the standard Recenter option. Instead of using the mean and standard deviation, we subtract the median and divide by median absolute deviation.

6.2 Impute

Imputation is the process of filling in the gaps (or missing values) in data. Often, data will contain missing values, and this can cause a problem for some modelling algorithms. For example, the random forest option silently removes any entity with any missing value! For datasets with a very large number of variables, and a reasonable number of missing values, this may well result in a small, unrepresentative dataset, or even no dataset at all!

There are many types of imputations available, only some of which are directly available in Rattle. We note thought that there is always discussion about whether imputation is a good idea or not. After all, we end up inventing data to suit the needs of the tool we are using. We won't discuss the pros and cons in much detail, but we provide some observations and concentrate on how we might impute values. Do be aware though that imputation can be problematic.

If the missing data pattern is monotonic, then imputation can be simplified. See [Schafer \(1997\)](#) for details. The pattern of missing values is also useful in suggesting which variables could be candidates for imputing the

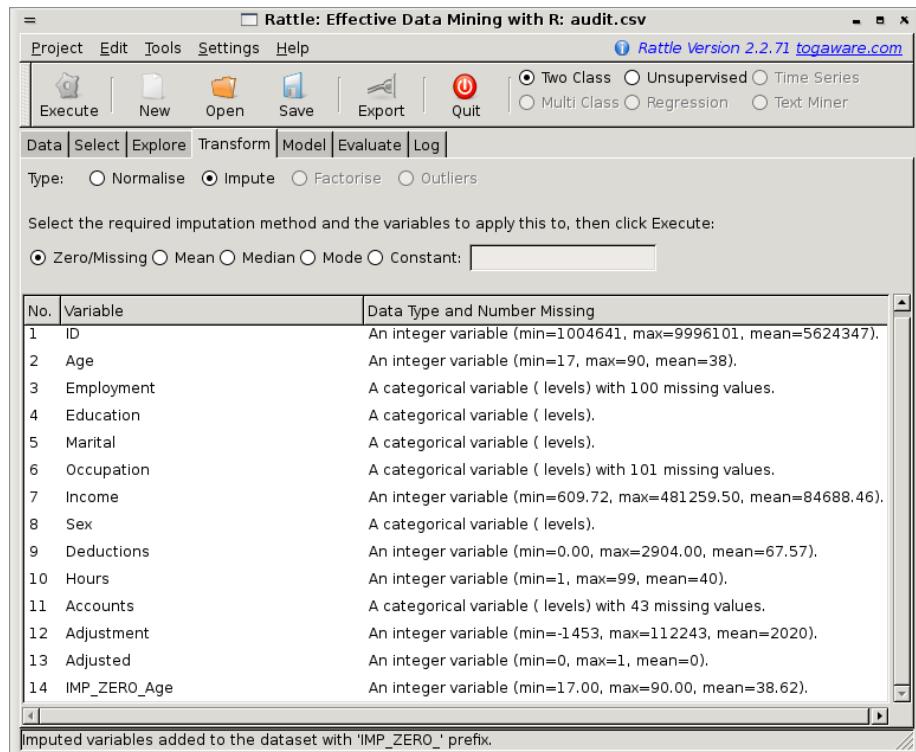


Figure 6.5: Selection of imputations.

missing values of other variables. Refer to the Show Missing check button of the Summary option of the Explore tab for details

(see Section 5.1.6).

When Rattle performs an imputation it will store the results in a variable of the dataset which has the same name as the variable that is imputed, but prefixed with IMP_. Such variables, whether they are imputed by Rattle or already existed in the dataset loaded into Rattle (e.g., a dataset from SAS), will be treated as input variables, and the original variable marked to be ignored.

6.2.1 Zero/Missing

The simplest of imputations involves replacing all missing values for a variable with a single value! This makes most sense when we know that the missing values actually indicate that the value is 0 rather than unknown. For example, in a taxation context, if a tax payer does not provide a value for a specific type of deduction, then we might assume that they intend it to be zero. Similarly, if the number of children in a family is not recorded, it could be a reasonable assumption to assume it is zero.

For categorical data the simplest approach to imputation is to replace missing values with a special value, *Missing*.

6.2.2 Mean/Median/Mode

Often a simple, if not always satisfactory, choice for missing values that are known not to be zero is to use some “central” value of the variable. This is often the mean, median, or mode, and thus usually has limited impact on the distribution. We might choose to use the mean, for example, if the variable is otherwise generally normally distributed (and in particular does not have any skewness). If the data does exhibit some skewness though (e.g., there are a small number of very large values) then the median might be a better choice.

For categorical variables, there is, of course, no mean nor median, and so in such cases we might choose to use the mode (the most frequent value) as the default to fill in for the otherwise missing values. The mode can also be used for numeric variables.

Whilst this is a simple and computationally quick approach, it is a very blunt approach to imputation and can lead to poor performance from the resulting models.

We can see the effect of the imputation of missing values on the variable `Age` using the mode in Figure

Refer to [Data Mining With R](#), from page 42, for more details.

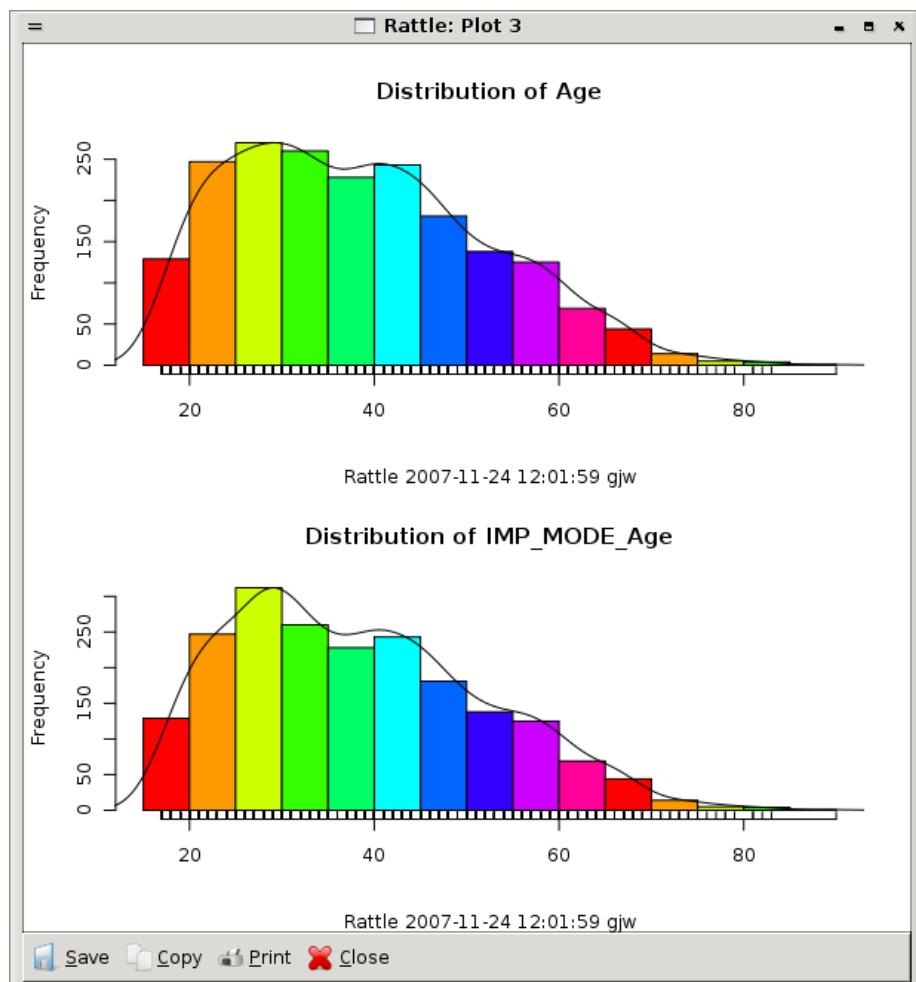


Figure 6.6: Imputation using the mode for missing values of Age.

6.2.3 Constant

This choice allows us to provide our own default value to fill in the gaps. This might be an integer or real number for numeric variables, or else a special marker or the choice of something other than the majority category for Categorical variables.

6.3 Remap

This provides numerous re-mapping operations, including binning, log transforms, ratios, and mapping categorical variables into indicator variables.

6.3.1 Binning

Togaware Watermark For Data Mining Survival

A *binning* function is provided by Rattle, coded by Daniele Medri. The Rattle interface provides an option to choose between Quantile binning, KMeans binning, and Equal Width binning. For each option the default number of bins is 4, and we can change this to suit our needs. The generated variables are prefixed with either `BIN_QUn_`, `BIN_KMn_`, and `BIN_EWn_` respectively, with `n` replaced with the number of bins. Thus, we can create multiple binnings for any variable.

Note that quantile binning is the same as equal count binning.

6.3.2 Indicator Variables

Some model builders do not handle categorical variables. Neural networks and regression are two examples. A simple approach in this case is to turn the categorical variable into some numeric form. If the categorical variable is not an ordered categorical variable, then the usual approach is to turn the single variable into a collection of so called indicator variables. For each value of the categorical variable there will be a new indicator variable which will have the value 1 for any entity that has this categorical value, and 0 otherwise. The result is a collection of numeric variables.

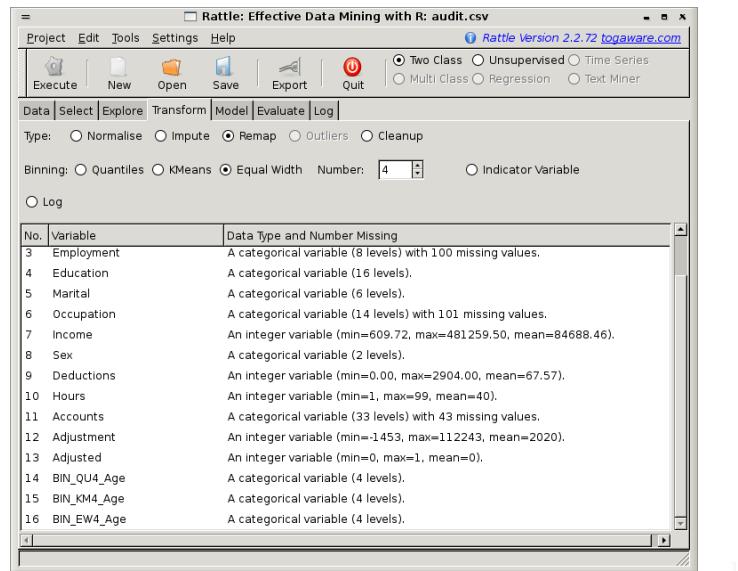


Figure 6.7: Binning Age.

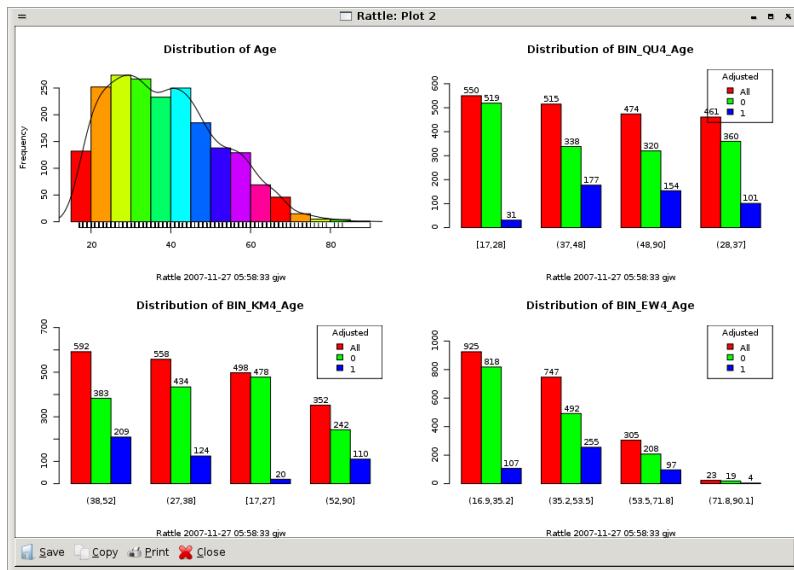


Figure 6.8: Distributions of binned Age.

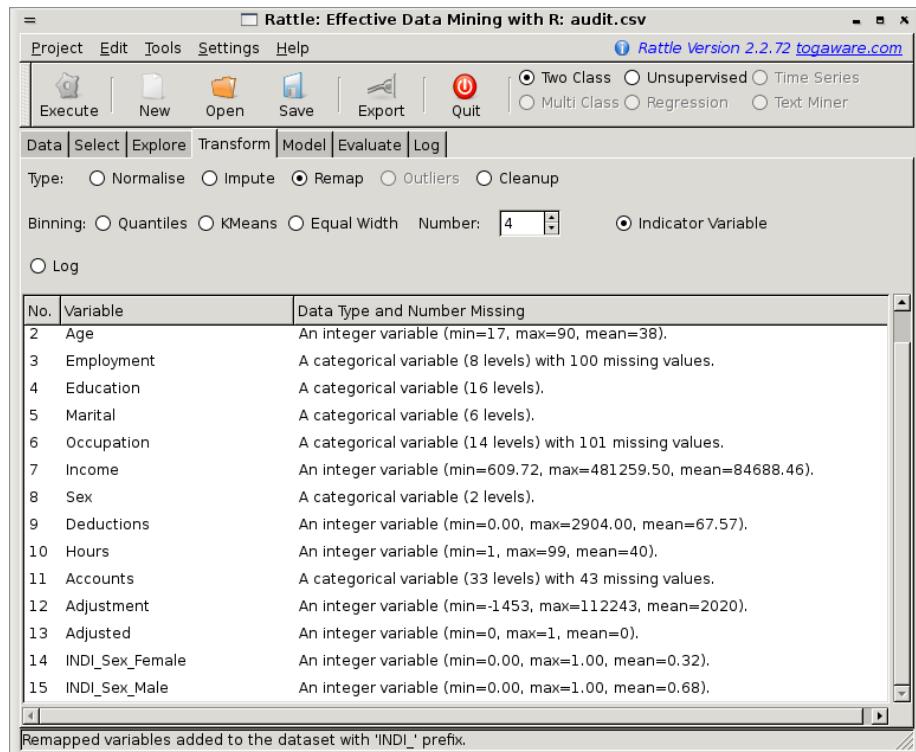


Figure 6.9: Turning Gender into an Indicator Variable.

Rattle's Transform tab provides an option to transform one or more categorical variables into a collection of indicator variables. Each is prefixed by `INDI_` and the remainder is made up of the name of the categorical variable (e.g., `Gender`) and the particular value (e.g., `Female`), to give `INDI_Gender_Female`. Figure 6.9 shows the result of turning the variable `Gender` into two indicator variables.

There is not always a need to transform a categorical variable. Some model builders, like the regressions in Rattle, will do it for us automatically.

6.3.3 Join Categoricals

The **Join Categoricals** option provides a convenient way to stratify the dataset, based on multiple categorical variables. It is a simple mechanism that creates a new variable from the combination of all of the values of the two constituent variables selected in the Rattle interface. The resulting variables are prefixed with `JOIN_` and include the names of both the constituent variables.

A simple example might be to join Gender and Marital, to give a new variable, `JOIN_Marital_Gender`.

We might also want to join a numeric variable and a categorical variable, like the typical `Age` and `Gender` stratification. To do this we first use the **Binning** option within **Remap** to categorise the `Age` variable (Section 6.3.1, page 92).

Togaware Watermark For Data Mining Survival

6.3.4 Math Transforms

A **Log** transform is available. The generated variable is prefixed with `REMAP_LOG_`.

6.4 Outliers

To be implemented.

6.5 Cleanup

It is quite easy to get our dataset variable count up to significant numbers. The **Cleanup** option (not yet available) allows us to tell Rattle to actually delete columns from the dataset. This allows us to perform numerous transformations and then to save the dataset back into a CSV file (by exporting it from the Data tab).

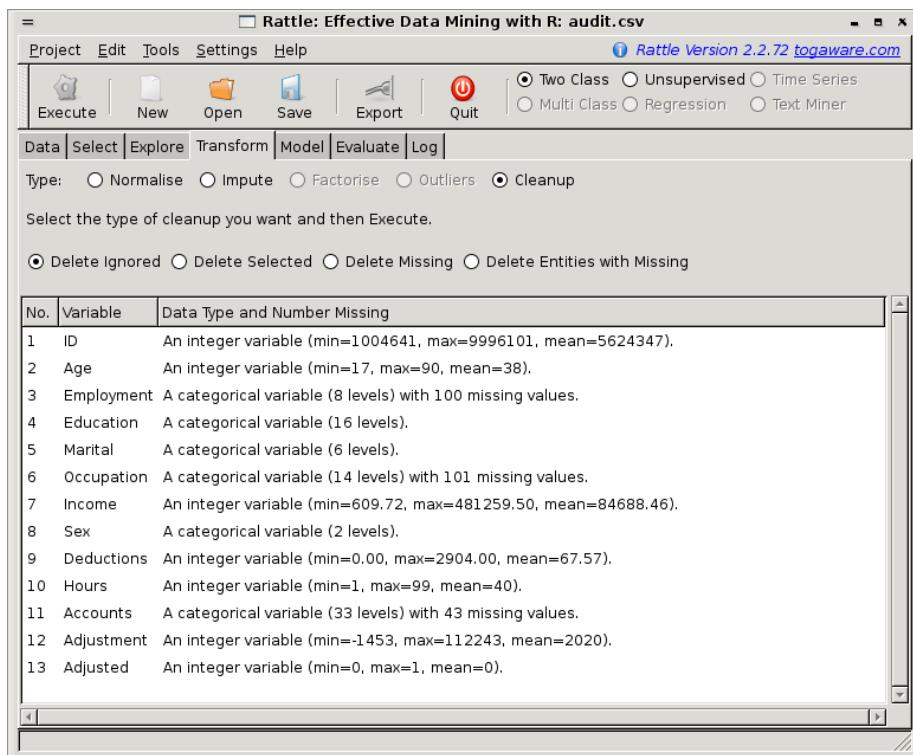


Figure 6.10: Selection of cleanup operations.

6.5.1 Delete Ignored

As an example, suppose we have loaded our familiar *audit* dataset. In the Select tab choose to set the role of Age, Employment, Education, Marital, and Occupation to be Ignore, and then click the Execute button for these new roles to take effect. In the R console window use the *object.size* to determine the current amount of memory the dataset is taking up:

```
> object.size(crs$dataset)
[1] 128904
```

Now, back in the Rattle window, navigate to the Transform tab and choose the **Cleanup** option. The **Delete Ignored** sub-option is the default. Click the **Execute** button to remove the columns that we marked as Ignored. Now in the R console, check how much space the dataset is taking up:

```
> object.size(crs$dataset)
[1] 84216
```

Togaware Watermark For Data Mining Survival

6.5.2 Delete Selected

Delete the variables selected in the textview below the radio buttons.

6.5.3 Delete Missing

Delete all variables that have any missing values. The variables with missing values are indicated in the textview below the radio buttons.

6.5.4 Delete Entities with Missing

Rather than delete variables with missing values, we can delete entities that have any missing values.

Togaware Watermark For Data Mining Survival

Chapter 7

Building Classification Models

Togaware Watermark For Data Mining Survival

The task of classification is at the heart of data mining! Most of what we learn from a traditional data mining course focuses on the algorithms from machine learning and statistics that build classification models. These models can then be used to classify new entities. The actual structure of the model also gives us insight into the relationships between the variables that are important in differentiating the classes.

This chapter focuses on this common data mining task of classification and prediction. We consider binary (or two class) classification, but the concepts also apply to multi-class classification.

The chapter begins with the introduction of a framework in which we understand model building. We then continue with a review of risk charts as a mechanism for evaluating two class models. Whilst a separate chapter (Chapter ??, page ??) covers evaluation in detail we present the concept of risk charts here so that we can explore and compare the performance of the models we build as we introduce the different model builders.

Each of the model builders supported by Rattle is then introduced. The model builders focus on binary (two-class) classification, where the aim is to distinguish between two classes of entities. Such problems abound, and the two classes might, for example, distinguish high risk and low risk

insurance clients, productive and unproductive taxation audits, responsive and non-responsive customers, successful and unsuccessful security breaches, and many other similar examples.

Rattle provides a straight-forward interface to the collection of model builders commonly used in data mining for binary classification. For each, a basic collection of tuning parameters is exposed through the interface for fine tuning the model building process. Where possible, Rattle attempts to present good default values to allow the user to simply build a model with no or little tuning. This may not always be the right approach, but is certainly a good place to start.

The two class model builders provided by Rattle are: Decision Trees, Boosted Decision Trees, Random Forests, Support Vector Machines, and Logistic Regression.

We will consider each of the model builders deployed in Rattle and characterise them through the types of models they generate and how the model building algorithms search for the best model that captures or summarises what the data is indicating.

Whilst a model is being built you will see the cursor image change to indicate the system is busy, and the status bar will report that a model is being built.

7.1 Building Models

In this section we present a framework within which we cast the task of data mining—the task being model building. We refer to an algorithm for building a model as a *model builder*. Rattle supports a number of model builders, including decision tree induction, boosted decision trees, random forests, support vector machines, logistic regression, kmeans, and association rules. In essence, the model builders differ in how they represent the models they build (i.e., the discovered knowledge) and how they find (or search for) the best model within this representation.

We can think of the discovered knowledge, or the *model*, as being expressed as sentences in a language. We are familiar with the fact that we express ourselves using sentences in our own specific human languages

(whether that be English, French, or Chinese, for example). As we know, there is an infinite number of sentences that we can construct in our human languages.

The situation is similar for the “sentences” we construct through using model builders—there is generally an infinite number possible sentences. In human language we are generally very well skilled at choosing sentences from this infinite number of possibilities to best represent what we would like to communicate. And so it is with model building. The skill is to express within the language chosen the best sentences that capture what it is we are attempting to model.

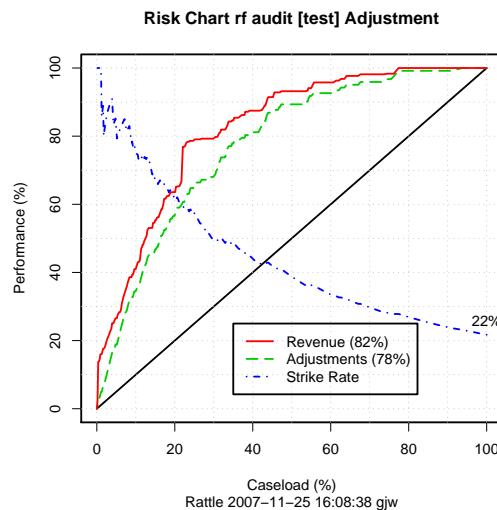
We formally present this general framework. The following sections then present models builders for various tasks in the context of this framework.

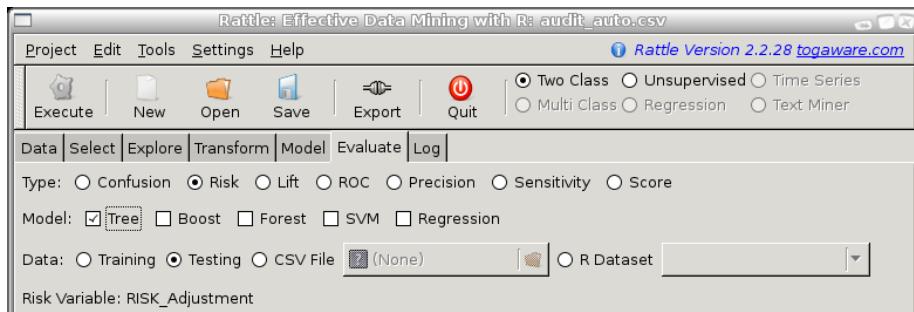
FRAMEWORK GOES HERE

7.2 Risk Charts

We have mentioned in passing, particularly in the previous chapter, the idea of a risk chart for evaluating the performance of our models. A risk chart is somewhat similar in concept to a number of the other evaluation approaches, particularly the ROC curves, which have also been mentioned and will be covered in detail in Chapter 9. We formally introduce the risk chart here

(rather than in Chapter 9) in order to be able to discuss the model builders in practise, and in particular illustrate their performance. The risk charts can be displayed within Rattle, once we have built our models, by choosing the Risk option of the Evaluate tab.





A risk chart is particularly useful in the context of the *audit* dataset, and for risk analysis tasks in general. Already we have noted that this dataset has a two class target variable, `Adjusted`. We have also identified a so called *risk* variable, `Adjustment`, which is a measure of the size of the risk associated with each entity. Entities that have no adjustment following an audit (i.e., they are clients who have supplied the correct information) will of course have no risk associated with them (`Adjustment = 0`). Entities that do have an adjustment will have a risk associated with them, and for convenience we simply identify the value of the adjustment as the magnitude of the risk.

In particular, we can think of revenue (or tax) authorities, where the outcomes of audits include a dollar amount by which the tax obligation of the taxpayer has been changed (which may be a change in favour of the revenue authority or in favour of the taxpayer). For fraud investigations, the outcome might be the dollar amount recovered from the fraudster. In these situations it is often useful to see the tradeoff between the return on investment and the number of cases investigated.

Rattle introduces the idea of a risk chart to evaluate the performance of a model in the context of risk analysis.

A risk chart plots performance against caseload. Suppose we had a population of just 100 entities (audit cases). The case load is the percentage of these cases that we will actually ask our auditors to process. The remainder we will not consider any further, expecting them to be low risk, and hence, with limited resources, not requiring any action. The decision as to what percentage of cases are actually actioned corresponds to the X axis of the risk chart - the caseload. A 100% caseload indicates that we will action all audit cases. A 25% caseload indicates that we will

action just one quarter of all cases.

For a given testing population we know how many cases resulted in adjustments. We also know the magnitude of those adjustments (the risk). For a population of 100 cases, if we were to randomly choose 50% of the cases for actioning, we might expect to recover just 50% of the cases that actually did require an adjustment and 50% of the risk (or in this case, the revenue) recovered from these adjusted cases. Similarly for every caseload value: for a random caseload of 25% of the population we might expect to recover 25% of the adjustments and revenue. The diagonal black line of the risk chart represents this random selection, and can be thought of as the baseline against which to compare the performance of our models.

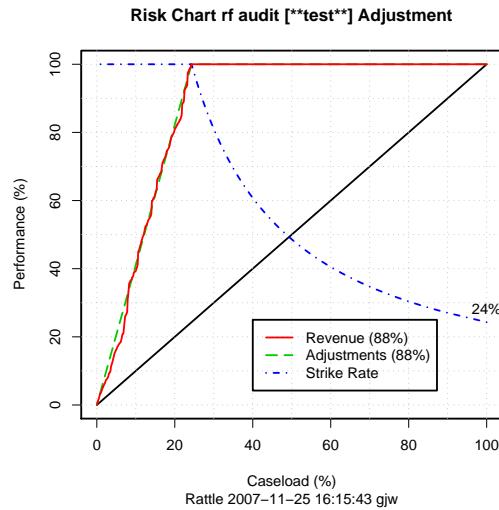
Any model that Rattle builds in the two-class paradigm generates a risk score for each entity. This is generally the probably of the case requiring an adjustment. We can use this score to sort all of the cases in decreasing order of the score. In selecting cases to be actioned, we then start with those cases that have the highest score. Thus, in evaluating the performance of our model, the caseload axis represents the sorted list of cases, from the highest scored cases at the left (starting at 0% of the cases actioned), and the lowest at the right (ending with 100% of the cases actioned).

The green (dashed) line of a risk chart then records the percentage of adjusted cases that we would actually expect to identify for a given percentage of the caseload. In the risk chart above, for example, we can see that if we only actioned the top 20% of the cases, as scored by our model, we recover almost 60% of the cases that actually did require an adjustment. Similarly, for an 80% caseload we can recover essentially all of the cases that required adjustment. Hence our model can save us 20% of our caseload (i.e., 20% of our resources) whilst pretty much recovering all of the adjustments.

The red (solid) line similarly records the percentage of the total revenue (or risk) that is recovered for any particular caseload. In our example above we see that the red and green lines essentially follow each other. This is not always the case.

EXPLAIN THE REVENUE AND STRIKE RATES. EXPLAIN THE AUC.

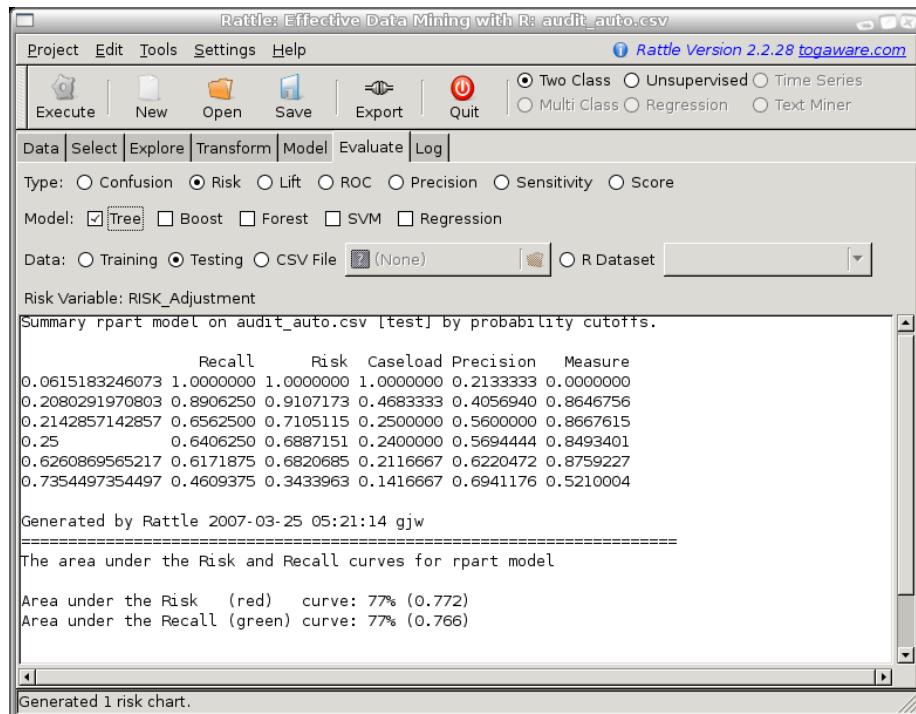
A perfect model performance assessed through a risk chart is then a risk chart that maximises the area under the two curves. We can illustrate such a risk chart by plotting the performance of a random forest model on the training data where a random forest often performs “perfectly,” as illustrated in this risk chart. The strike rate over the whole dataset in this case is 24% (as annotated at the right hand end of the strike rate line). That is, only 24% of all of the cases in this dataset of audit cases actually required an adjustment. The perfect model accurately identifies all 24% of the cases (and 24\$ of the risk) with 24% of the caseload! Thus we see the performance plot having just two components: the essentially linear progression from a caseload of 0% and performance of 0% up to a caseload of 24% and performance of 100%, and then the flat 100% performance as we increase the caseload to 100%.



When a Risk Chart is generated the text window in Rattle will display the aggregated data that is used to construct the plot. This data consists of a row for each level of the probability distribution that is output from the model, ordered from the lowest probability value to a value of 1. For each row we record the model performance in terms of predicting a class of 1 if the probability cutoff was set to the corresponding value.

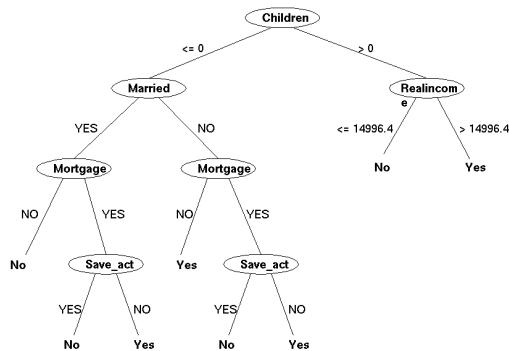
For example, we might choose a cutoff to be a probability of 0.28 so that anything predicted to be in class 1 with a probability of 0.28 or more will be regarded as in class 1. Then the number of predicted positives (or the Caseload) will be 30% (0.301667) of all cases. Amongst this 30% of cases are 69% of all true positives and they account for 79% of the total of the risk scores. The strike rate (number of true positives amongst the positives predicted by the model) is 61%. Finally, the measure reports the sum of the distances of the risk and recall from the baseline (the

diagonal line). This measure can indicate the optimal caseload in terms of maximising both risk recovery and recall.



7.3 Decision Trees

One of the classic machine learning techniques, widely deployed in data mining, is decision tree induction. Using a simple algorithm and a simple knowledge structure, the approach has proven to be very effective. These simple tree structures represent a classification (and regression) model. Starting at the root node, a simple question is asked

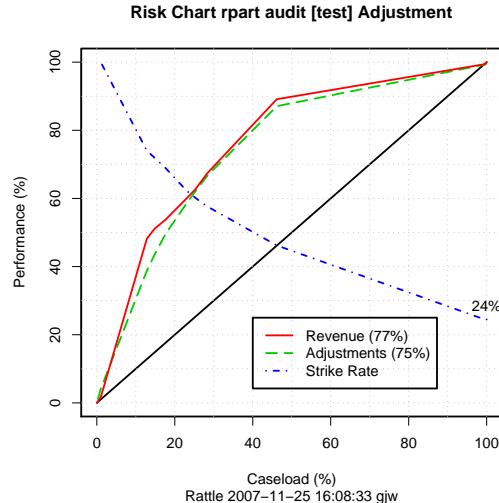


(usually a test on a variable value, like $Age < 35$). The branches emanating from the node correspond to alternative answers. For example, with a test of $Age < 35$ the alternatives would be *Yes* and *No*. Once a leaf node is reached (one from which no branches emanate) we take the decision or classification associated with that node. Some form of probability may also be associated with the nodes, indicating a degree of certainty for the decision. Decision tree algorithms handle mixed types of variables, handle missing values, are robust to outliers and monotonic transformations of the input, and robust to irrelevant inputs. Predictive power tends to be poorer than other techniques.

The model is expressed in the form of a simple decision tree (the knowledge representation). At each node of the tree we test the value of one of the variables, and depending on its value, we follow one of the branches emanating from that node. Thus, each branch can be thought of as having a test associated with it, for example $Age < 35$. This branch then leads to another node where there will be another variable to test, and so on, until we reach a leaf node of the tree. The leaf node represents the decision to be made. For example, it may be a *yes* or *no* for deciding whether an insurance claim appears to be fraudulent.

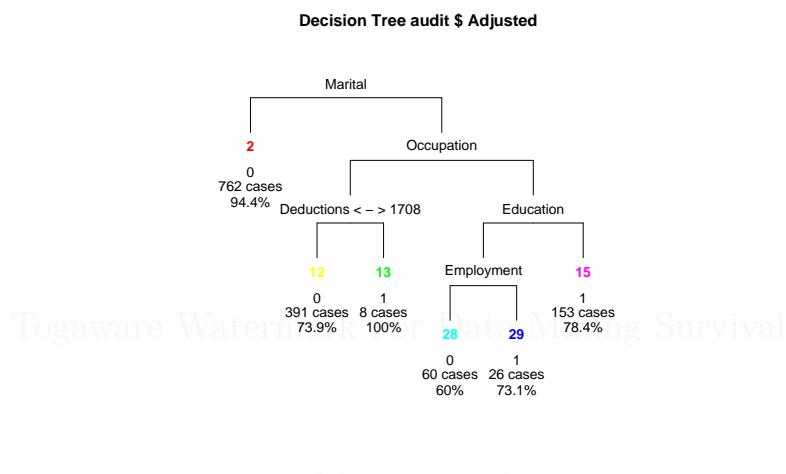
In searching for a decision tree to best model our data, alternative decision trees are considered in a top-down fashion, beginning with the decision of the variable to initially partition the data (at the root node).

Decision trees are the building blocks of data mining. Since their development back in the 1980's they have been the most widely deployed data mining model builder. The attraction lies in the simplicity of the resulting model, where a decision tree (at least one that is not too large) is quite easy to view, to understand, and, indeed, to explain to management! However, deci-



sion trees do not deliver the best performance in terms of the risk charts, and so there is a trade off between performance and simplicity of explanation and deployment.

7.3.1 Tutorial Example



7.3.2 Formalities

7.3.3 Tuning Parameters

Priors

Sometimes the proportions of classes in a training set do not reflect their true proportions in the population. You can inform Rattle of the population proportions and the resulting model will reflect these.

The priors can be used to “boost” a particularly important class, by giving it a higher prior probability, although this might best be done through the Loss Matrix.

In Rattle the priors are expressed as a list of numbers that sum up to 1, and of the same length as the number of classes in the training dataset.

An example for binary classification is 0.5,0.5.

The default priors are set to be the class proportions as found in the training dataset.

Loss Matrix

The loss matrix is used to weight the outcome classes differently (the default is that all outcomes have the same loss of 1). The matrix will be constructed row-wise from the list of numbers we supply, and is of the same dimensions as the number of classes in the training dataset. Thus, for binary classification, four numbers must be supplied. The diagonal should be all zeros.

An example is: 0,10,1,0, which might be interpreted as saying that an actual 1, predicted as 0 (i.e., a false negative) is 10 times more unwelcome than a false positive! © Graham Williams 2006-2008 Data Mining Survival Guide

Rattle uses the loss matrix to alter the priors which will affect the choice of variable to split the dataset on at each node, giving more weight where appropriate.

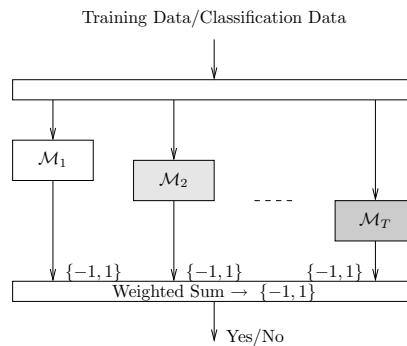
Complexity

The complexity parameter (cp) is used to control the size of the decision tree and to select the optimal tree size. If the cost of adding another variable to the decision tree from the current node is above the value of cp, then tree building does not continue. We could also say that tree construction does not continue unless it would decrease the overall lack of fit by a factor of cp.

Setting this to zero will build a tree to its maximum depth (and perhaps will build a very, very, large tree). This is useful if you want to look at the values for CP for various tree sizes. This information will be in the text view window. You will look for the number of splits where the sum of the xerror (cross validation error, relative to the root node error) and xstd is minimum. This is usually early in the list.

7.4 Boosting

The **Boosting** meta-algorithm is an efficient, simple, and easy to understand model building strategy. The popular variant called **AdaBoost** (an abbreviation for Adaptive Boosting) has been described as the “best off-the-shelf classifier in the world” (attributed to Leo Breiman by [Hastie et al. \(2001, p. 302\)](#)). **Boosting** algorithms build multiple models from a dataset, using some other model builders, such as a decision tree builder, that need not be a particularly good model builder. The basic idea of boosting is to associate a weight with each entity in the dataset. A series of models are built and the weights are increased (boosted) if a model incorrectly classifies the entity. The weights of such entities generally oscillate up and down from one model to the next. The final model is then an additive model constructed from the sequence of models, each model’s output weighted by some score. There is little tuning required and little is assumed about the model builder used, except that it should be a relatively weak model builder! We note that boosting can fail to perform if there is insufficient data or if the weak models are overly complex. Boosting is also susceptible to noise.



Boosting is an example of an ensemble model builder.

Boosting builds a collection of models using a “weak learner” and thereby reduces misclassification error, bias, and variance ([Bauer and Kohavi, 1999](#); [Schapire et al., 1997](#)). Boosting has been implemented in, for example, **C5.0**. The term originates with [Freund and Schapire \(1995\)](#).

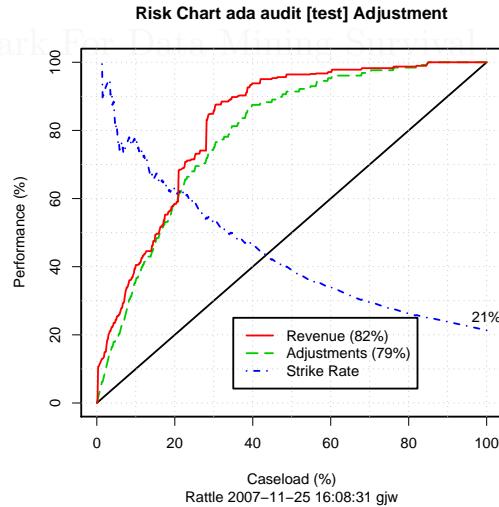
The algorithm is quite simple, beginning by building an initial model from the training dataset. Those entities in the training data which the model was unable to capture (i.e., the model mis-classifies those entities) have their weights boosted. A new model is then built with these boosted entities, which we might think of as the problematic entities in the train-

ing dataset. This model building followed by boosting is repeated until the specific generated model performs no better than random. The result is then a panel of models used to make a decision on new data by combining the “expertise” of each model in such a way that the more accurate experts carry more weight.

As a meta learner Boosting employs some other simple learning algorithm to build the models. The key is the use of a weak learning algorithm—essentially any weak learner can be used. A weak learning algorithm is one that is only somewhat better than random guessing in terms of error rates (i.e., the error rate is just below 50%). An example might be decision trees of depth 1 (i.e., decision stumps).

7.4.1 Tutorial Example

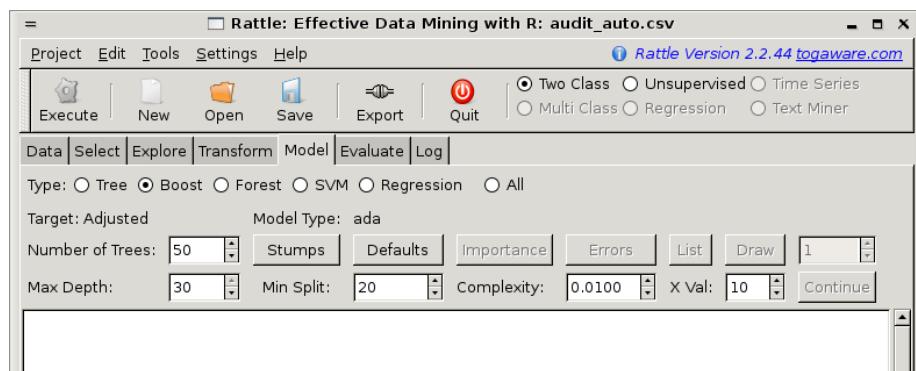
The ensemble approaches build upon the decision tree model builder by building many decision trees through sampling the training dataset in various ways. The *ada* boosting algorithm is deployed by Rattle to provide its boosting model builder. With the default settings a very reasonable model can be built. At a 60% caseload we are recovering 98% of the cases that required adjustment and 98% of the revenue.



7.4.2 Formalities

7.4.3 Tuning Parameters

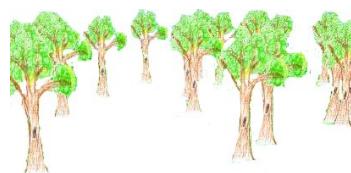
There are quite a few options exposed by Rattle for boosting our decision tree model. We will explore how these can help us to build the best model.



It is not always clear exactly how many trees we should build for our ensemble. The default for ada is to build 50 trees. But is that enough? The Boost functionality in Rattle allows the ensemble of trees to be added to, so that we can easily explore whether more trees will improve the performance of the model. To do so, simply increase the value specified in the Number of Trees text box and click the Continue button. This will pick up the model building from where it was left off and build as many more trees as is needed to get up to the specified number of trees.

7.5 Random Forests

A **random forest** is an ensemble (i.e., a collection) of unpruned decision trees. Random forests are often used when we have very large training datasets and a very large number of input variables (hundreds or even thousands of input variables). A random forest model is typically made up of tens or hundreds



of decision trees.

The generalisation error rate from random forests tends to compare favourably to boosting approaches, yet the approach tends to be more robust to noise in the training dataset, and so tends to be a very stable model builder, not suffering the sensitivity to noise in a dataset that single decision tree induction does. The general observation is that the random forest model builder is very competitive with nonlinear classifiers such as artificial neural nets and support vector machines. However, performance is often dataset dependent and so it remains useful to try a suite of approaches.

Each decision tree is built from a random subset of the training dataset, using what is called replacement (thus it is doing what is known as bagging), in performing this sampling. That is, some entities will be included more than once in the sample, and others won't appear at all. Generally, about two thirds of the entities will be included in the subset of the training dataset, and one third will be left out.

In building each decision tree model based on a different random subset of the training dataset a random subset of the available variables is used to choose how best to partition the dataset at each node. Each decision tree is built to its maximum size, with no pruning performed.

Together, the resulting decision tree models of the forest represent the final ensemble model where each decision tree votes for the result, and the majority wins. (For a regression model the result is the average value over the ensemble of regression trees.)

In building the random forest model we have options to choose the number of trees to build, to choose the training dataset sample size to use for building each decision tree, and to choose the number of variables to randomly select when considering how to partition the training dataset at each node. The random forest model builder can also report on the input variables that are actually most important in determining the values of the output variable.

By building each decision tree to its maximal depth (i.e., by not pruning the decision tree) we can end up with a model that is less biased.

The randomness introduced by the random forest model builder in the

dataset selection and in the variable selection delivers considerable robustness to noise, outliers, and over-fitting, when compared to a single tree classifier.

The randomness also delivers substantial computational efficiencies. In building a single decision tree the model builder may select a random subset of the training dataset. Also, at each node in the process of building the decision tree, only a small fraction of all of the available variables are considered when determining how to best partition the dataset. This substantially reduces the computational requirement.

In summary, a random forest model is a good choice for model building for a number of reasons. First, just like decision trees, very little, if any, pre-processing of the data needs to be performed. The data does not need to be normalised and the approach is resiliant to outliers. Second, if we have many input variables, we generally do not need to do any variable selection before we begin model building. The random forest model builder is able to target the most useful variables. Thirdly, because many trees are built and there are two levels of randomness and each tree is effectively an independent model, the model builder tends not to overfit to the training dataset.

7.5.1 Tutorial Example

Our audit dataset can be used to provide a simple illustration of building a random forest model. To follow this example, load the audit dataset into Rattle using the Data tab (Section 3.3, page 27), and select the appropriate input variables using the Select tab (Section 4.2, page 43) to identify **Adjustment** as the Risk variable (Figure 4.1, page 43).

Now on the Model tab choose the **Forest** radio button as shown in Figure 7.1.

Click the Execute button to build the model. We will see a popup, as in Figure 7.2, explaining one of the limitations of this implementation of random forests. Underneath, Rattle simply employs the R package called *randomForest* (implemented by Andy Liaw) which in turn is based on the original Fortran code from the developers of the algorithm (Leo Breiman and Adele Cutler, who also own the Random Forests trademark

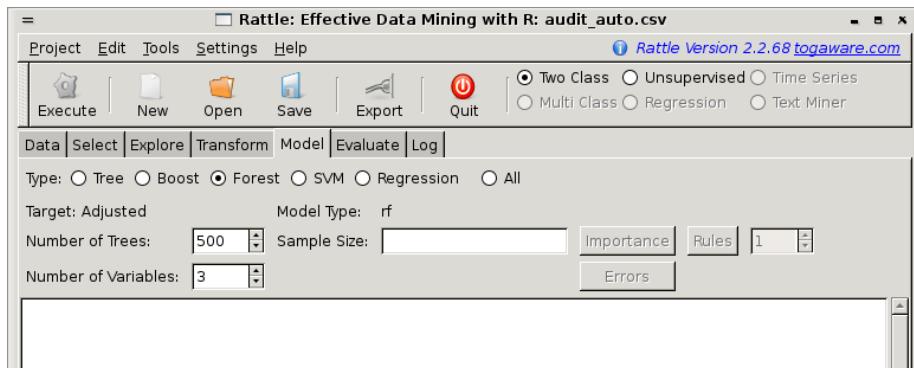


Figure 7.1: Random forest tuning parameters.

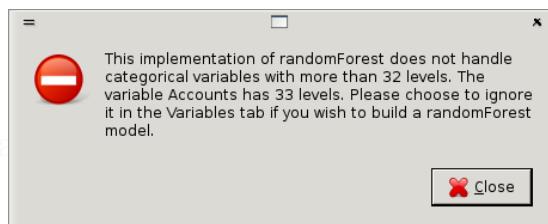


Figure 7.2: Random forests only supports factors with up to 32 levels.

and license it exclusively to Salford Systems). Thus some limitations are carried through to Rattle. This particular limitation is that categorical variables with more than 32 categories are not handled. Statistical concerns also suggest that categorical variables with more than 32 categories don't make a lot of sense.

As an aside, we also note that the Breiman-Cutler implementation of the random forest model builder as used in R appears to produce better results than those produced by the Weka implementation of random forest.

To rectify this problem with too many categorical values, the simplest approach is to change the role of the Accounts variable under the **Select** tab, to be one of **Ignore**, being sure to **Execute** the tab before coming back to the **Model** tab to build a the model. The model building takes about 3 seconds, and the results are displayed in the textview window of

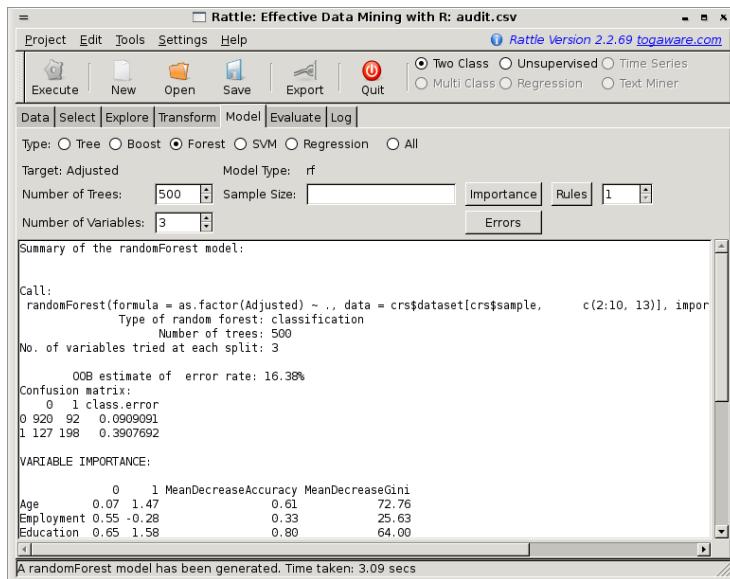


Figure 7.3: Random forest model of audit data.

Rattle, as in Figure 7.3.

We can see that by default the summary begins with a review of the actual underlying R code that was executed. We need to scroll to the right in order to see the full command.

The next bit of text provides a summary of the model builder's parameters, indicating that a classification model was built, consisting of 500 decision trees, and 3 variables to choose from for each time we partition the dataset.

Next we see what is called an **OOB estimate of error rate**. OOB stands for “out of bag.” This estimate is regarded to be an unbiased estimate of the true error of the model. The idea is that in building any particular tree in the ensemble, we use a sample of the training dataset (technically this is called a bootstrap sample and is usually about two thirds of all the available data). The entities that are in the sample are said to be contained within the bag that is used to build the model. Thus those that are not being used to build the model are said to be out-of-bag. These out-of-bag entities are not used in building the model this time round, and hence can be used as a test dataset for this specific model. Any

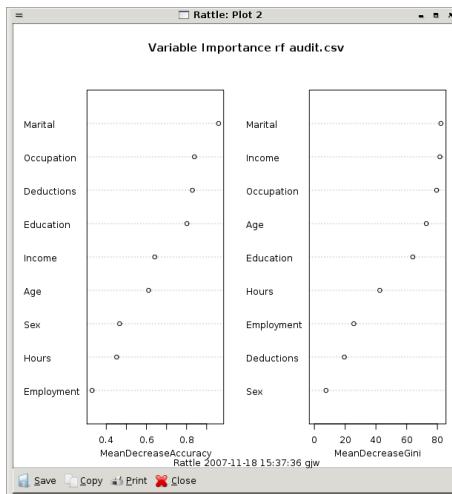


Figure 7.4: Random forest model measure of variable importance.

particular entity will be out-of-bag a reasonable number of times, and so we can obtain the proportion of times this entity is correctly classified and so obtain the estimates of performance for the whole model, which is what is reported by the model builder.

The **Importance** button allows us to graphically view the random forest's perspective on which variables are important in determining the value of the output variable. The plot (Figure 7.4) shows the two different measures of variable importance. The first importance is the scaled average of the prediction accuracy of each variable, and the second is the total decrease in node impurities splitting on the variable over all trees, using the Gini index.

Moving on to the **Evaluate** tab, selecting the **Risk** radio button, and clicking the **Execute** button will generate the risk chart for this model, using the **Adjustment** variable as the measure of the size of the actual risk associated with those cases that required adjustments. The risk chart, Figure 7.5, indicates that we have a reasonable model.

For deployment, as a data miner we can put forward a case for using this model to score our population, and to then only have our auditors review perhaps 70% of the current number of cases. Thus, we would be saving our effort on the remaining 30% where we only recover less than 2% of

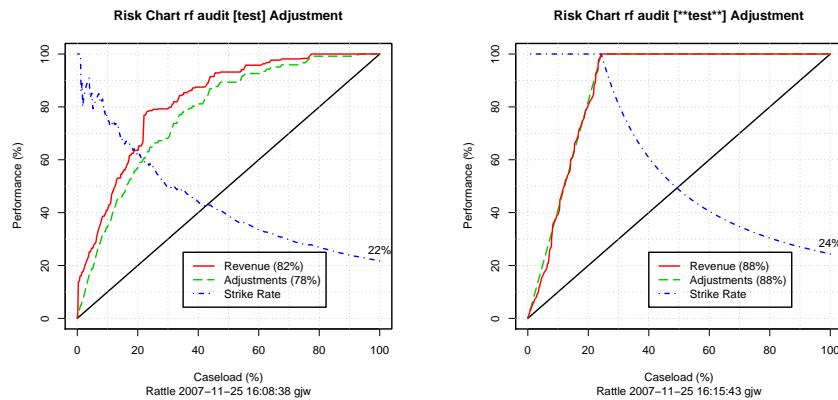


Figure 7.5: Random forest risk charts: test and train datasets.

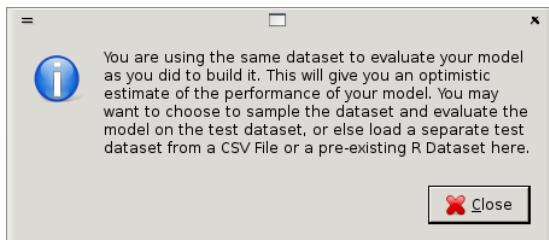


Figure 7.6: Warning when evaluating a model on the training dataset.

the actual risk, and 5% of the actual cases that needed adjustment. This is quite a powerful argument for business, and a saving of even 10% is often worth the effort.

In Rattle the default for the Evaluate tab is to use the testing dataset to show a reasonable estimate of the true performance of the model. It is informative, though to display the performance on the training set. In fact, building a random forest and evaluating it on training dataset gives a “prefect” result. For other models this would tend to tell us that our model has overfit the training data. However, the performance on the testing dataset indicates that we have a generally good model. See Section 10.2 for a discussion of overfitting.

Do note though that when applying the model to the training dataset for evaluation, Rattle will popup a warning indicating that this is not such a good idea (Figure 7.6).

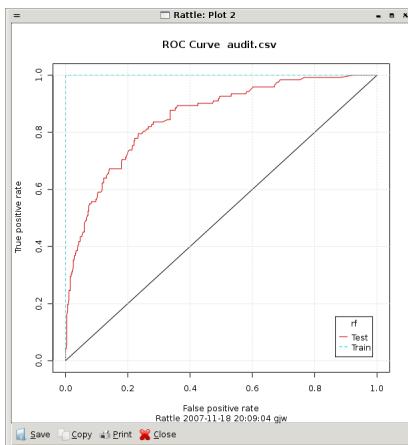


Figure 7.7: Random forest ROC chart.

We can obtain a more traditional ROC chart (Figure 7.7) from the Evaluate tab.

Togaware Watermark For Data Mining Survival

7.5.2 Formalities

The random forest model builder represents its knowledge as an ensemble of decision trees (Section 7.3.2, page 107 for details of decision trees).

7.5.3 Tuning Parameters

For the Two Class paradigm of Rattle, the random forest model build builds a *classification* model. Each tree in the resulting ensemble model is then used to predict the class of an entity, with the proportion of trees predicting the positive class then being the probability of the entity being in the positive class.

Rattle provides access to just three parameters (Figure 7.1) for tuning the models built by the random forest model builder: the number of trees, sample size, and number of variables. As is generally the case with Rattle, the defaults are a very good starting point! The defaults are to build 500 trees, to not do any sampling of the training dataset, and to choose from the square root of the number of variables available. In Figure 7.1 we see that the number of variables has automatically been set to 3 for

the *audit_auto.csv* dataset, which has 9 input variables.

Number of Trees

This specifies how many trees are to be built to populate the random forest. The default value is 500 and a common recommendation is that a minimum of 100 trees be built.

Random forest performance does not degrade as the number of trees increases.

Sample Size

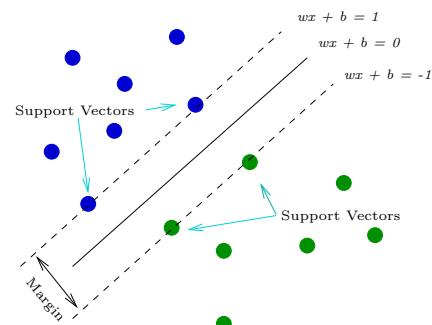
Number of Variables

This option sets the number of variables to randomly select from all of those available, each time we look to partition a dataset in the process of building the decision tree. The general default value is the square root of the total number of variables available.

If there are many noise variables, increase the number of variables considered at each node.

7.6 Support Vector Machine

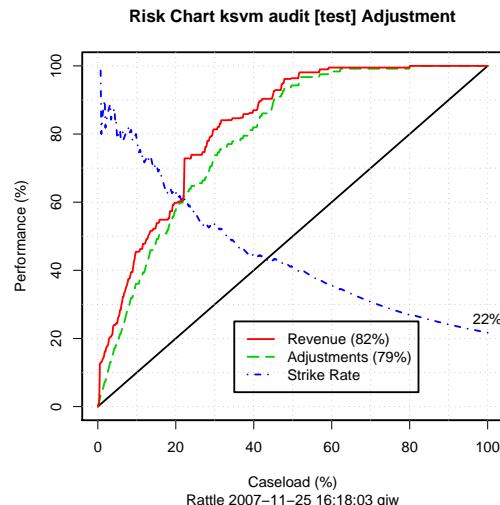
A Support Vector Machine (SVM) searches for so called *support vectors* which are data points that are found to lie at the edge of an area in space which is a boundary from one class of points to another. In the terminology of SVM we talk about the space between regions containing data points in different classes as being the margin between those classes. The sup-



port vectors are used to identify a hyperplane (when we are talking about many dimensions in the data, or a line if we were talking about only two dimensional data) that separates the classes. Essentially, the maximum margin between the separable classes is found. An advantage of the method is that the modelling only deals with these support vectors, rather than the whole training dataset, and so the size of the training set is not usually an issue. If the data is not linearly separable, then kernels are used to map the data into higher dimensions so that the classes are linearly separable. Also, Support Vector Machines have been found to perform well on problems that are non-linear, sparse, and high dimensional. A disadvantage is that the algorithm is sensitive to the choice of variable settings, making it harder to use, and time consuming to identify the best.

Support vector machines do not predict probabilities but rather produce normalised distances to the decision boundary. A common approach to transforming these decisions to probabilities is by passing them through a sigmoid function.

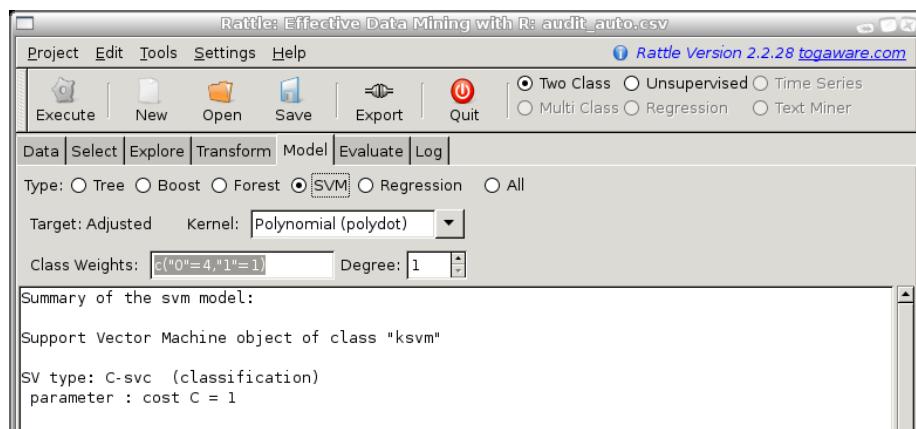
Rattle supports the building of support vector machine (SVM) models using the *kernlab* package for R. This package provides an extensive collection of kernel functions, and a variety of tuning options. The trick with support vector machines is to use the right combination of kernel function and kernel parameters—and this can be quite tricky. Some experimentation with the *audit* dataset, exploring different kernel functions and parameter settings, identified that a polynomial kernel function with the class weights set to `c("0"=4, "1"=1)` resulted in the best risk chart. For a 50% caseload we are recovering 94% of the adjustments and 96% of the rev-



7.7 Logistic Regression

121

enue.



There are other general variables that can be set for ksvm, including the cost of constraint violation (the trade-off between the training error and margin? could be from 1 to 1000, and is 1 by default).

For best results it is often a good idea to scale the numeric variables to have a mean of 0 and a standard deviation of 1.

For polynomial kernels you can choose the degree of the polynomial. The default is 1. For the audit data as you increase the degree the model gets much more accurate on the training data but the model generalises less well, as exhibited by its performance on the test dataset.

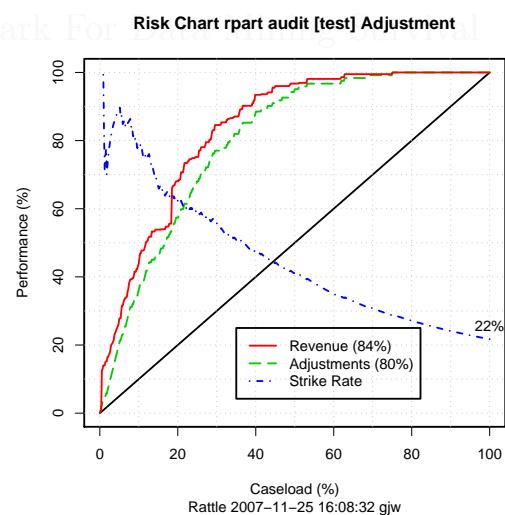
Another variable often needing setting for a radial basis function kernel is the sigma value. Rattle uses automatic sigma estimation (sigest) for this kernel, to find the best sigma, and so the user need not set a sigma value. If we wanted to experiment with various sigma values we can copy the R code from the Log tab and paste it into the R console, add in the additional settings, and run the model. Assuming the results are assigned into the variable `crs$ksvm`, as in the Log, we can then evaluate the performance of this new model using the Evaluate tab.

7.7 Logistic Regression

Logistic regression is a statistical model builder using traditional regression techniques but for predicting a 1/0 outcome. Logistic regression in fact builds a type of generalized linear model, using a so called logit function.

Logistic regression is the traditional statistical approach and indeed it can still produce good models as evidenced in the risk chart here. As noted in Section 10.1 though, logistic regression has not always been found to produce good models. Nonetheless, here we see a very good model that gives us an area under the curve of 80% for both Revenue and Adjustments, and at the 50% caseload we are recovering 94% of the cases requiring adjustment and 95% of the revenue associated with the cases that are adjusted.

For best results it is often a good idea to scale the numeric variables to have a mean of 0 and a standard deviation of 1.



7.8 Bibliographic Notes

Caruana and Niculescu-Mizil (2006) present a comprehensive empirical comparison of many of the modern model builders. An older comparison is known as the Statlog comparison (King et al., 1995).

The *ada* package for boosting was implemented by Mark Culp, Kjell Johnson, and George Michailidis, and is described in Culp et al. (2006).

Random forests were introduced by Breiman (2001), building on the concept of bagging (Breiman, 1996) and the random subspace method for decision forests (Ho, 1998). Breiman observed that “random forests do not overfit.”

Togaware Watermark For Data Mining Survival

Togaware Watermark For Data Mining Survival

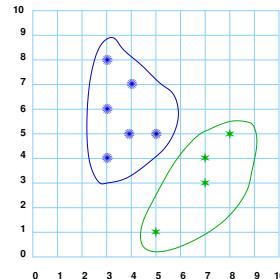
Chapter 8

Unsupervised Modelling

8.1 Cluster Analysis

Togaware Watermark For Data Mining Survival

Clustering is one of the core tools used by the data miner. Clustering allows us to group entities in a generally unguided fashion, according to how similar they are. This is done on the basis of a measure of the distance between entities. The aim of clustering is to identify groups of entities that are close together but as a group are quite separate from other groups.



8.1.1 KMeans

8.1.2 Export KMeans Clusters

We also note the export functionality is implemented for kmeans clusters. We have a choice though in what is exported. To export the actual model as PMML, where the centroids are recorded in the PMML model specification, choose the appropriate radio button before clicking the Export button. Alternatively, to save the association between each entity and the cluster into which it has been placed, choose the Clusters (CSV)

radio button. In either case we are prompted for the name of a file into which we save the data.

8.1.3 Discriminant Coordinates Plot

The discriminant coordinates plot is generated by projecting the original data.

8.1.4 Number of Clusters

Choosing the number of clusters is often quite a tricky exercise. Sometimes it is a matter of just try it and see. Other times you have some heuristics that help you to decide. Rattle provides a iterate approach. There is no definitive statistical answer to this issues

In deciding on a size for a robust cluster we need to note that the larger the number of clusters relative to the size of the sample, then the smaller our clusters will be. Perhaps there is a cluster size below which we don;t want to go.

Different cluster algorithms (and even different random seeds) result in different clusters, and how much they differ is a measure of cluster stability.

One approach to identifying a good cluster number is to iterate through multiple clusters and observe the sum of the within sum of squares. Rattle supports this with the Iterate Clusters option (see Figure 8.1), where a plot is also always generated (see Figure 8.2). A heuristic is to choose the number of clusters where we see the largest drop in the sum of the within sum of squares. In Figure 8.2 we might choose 12, 17 or perhaps even 26.

8.1 Cluster Analysis

127

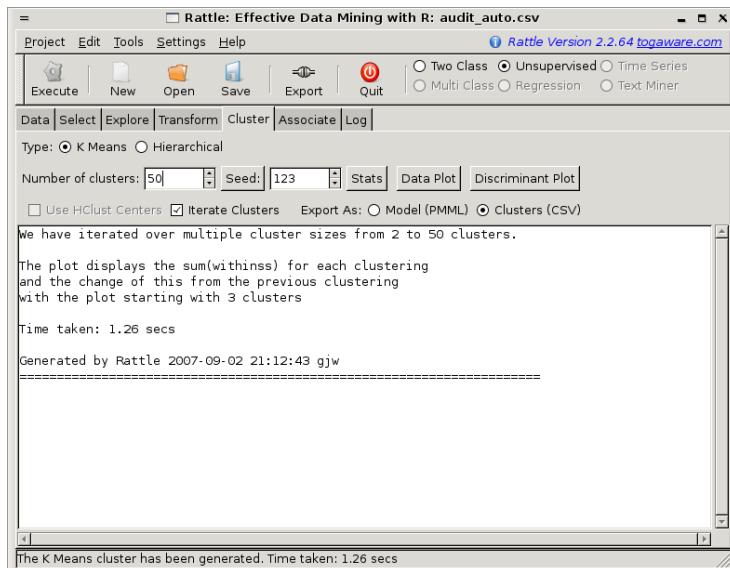


Figure 8.1: KMeans Iteration Interface

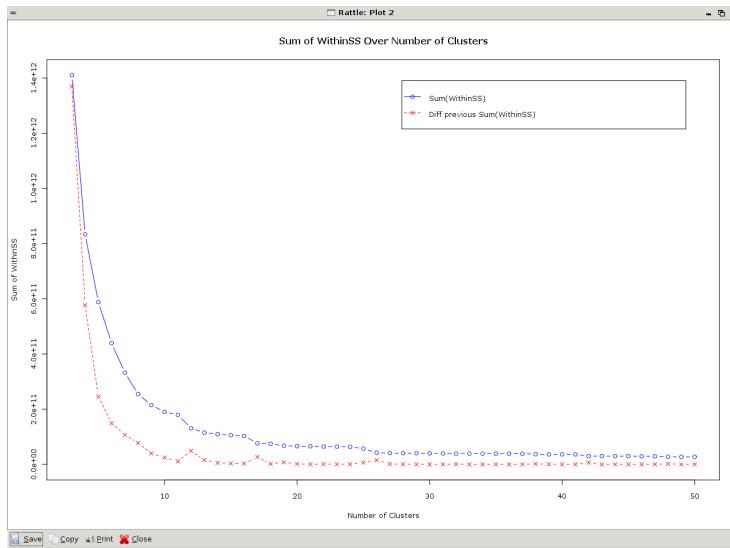


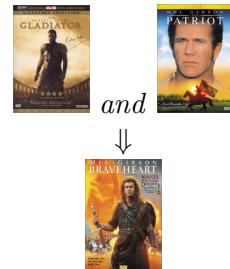
Figure 8.2: KMeans Iteration Plot

8.2 Hierarchical Clusters

Togaware Watermark For Data Mining Survival

8.3 Association Rules

Association analysis identifies relationships or affinities between entities and/or between variables. These relationships are then expressed as a collection of association rules. The approach has been particularly successful in mining very large transaction databases and is one of the core classes of techniques in data mining. A typical example is in the retail business where historic data might identify that customers who purchase the Gladiator DVD and the Patriot DVD also purchase the Braveheart DVD. The historic data might indicate that the first two DVDs are purchased by only 5% of all customers. But 70% of these then also purchase Braveheart. This is an *interesting* group of customers. As a business we may be able to take advantage of this observation by targetting advertising of the Braveheart DVD to those customers who have purchased both Gladiator and Patriot.



DETAILS OF REPRESENTATION AND SEARCH REQUIRED HERE.

Association rules are one of the more common types of techniques most associated with data mining. Rattle supports association rules through the **Associate** tab of the Unsupervised paradigm.

Two types of association rules are supported. Rattle will use either the Ident and Target variables for the analysis if a market basket analysis is requested, or else will use the Input variables for a rules analysis.

8.3.1 Basket Analysis

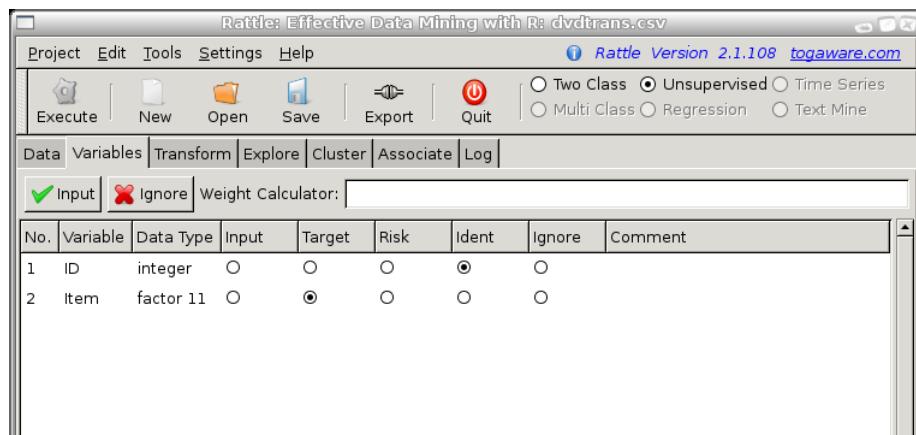
The simplest association analysis is often referred to as market basket analysis. Within Rattle this is enabled when the Baskets button is checked. In this case, the data is thought of as representing shopping baskets (or any other type of collection of items, such as a basket of medical tests, a basket of medicines prescribed to a patient, a basket of stocks held by an investor, and so on). Each basket has a unique identifier, and the variable specified as an Ident variable in the Select tab is taken as the

identifier of a shopping basket. The contents of the basket are then the items contained in the column of data identified as the target variable. For market basket analysis, these are the only two variables used.

To illustrate market basket analysis with Rattle, we will use a very simple dataset consisting of the DVD movies purchased by customers. Suppose the data is stored in the file `dvdtrans.csv` and consists of the following:

```
ID,Item
1,Sixth Sense
1,LOTR1
1,Harry Potter1
1,Green Mile
1,LOTR2
2,Gladiator
2,Patriot
2,Braveheart
3,LOTR1
3,LOTR2
4,Gladiator
4,Patriot
4,Sixth Sense
5,Gladiator
5,Patriot
5,Sixth Sense
6,Gladiator
6,Patriot
6,Sixth Sense
7,Harry Potter1
7,Harry Potter2
8,Gladiator
8,Patriot
9,Gladiator
9,Patriot
9,Sixth Sense
10,Sixth Sense
10,LOTR
10,Gladiator
10,Green Mile
```

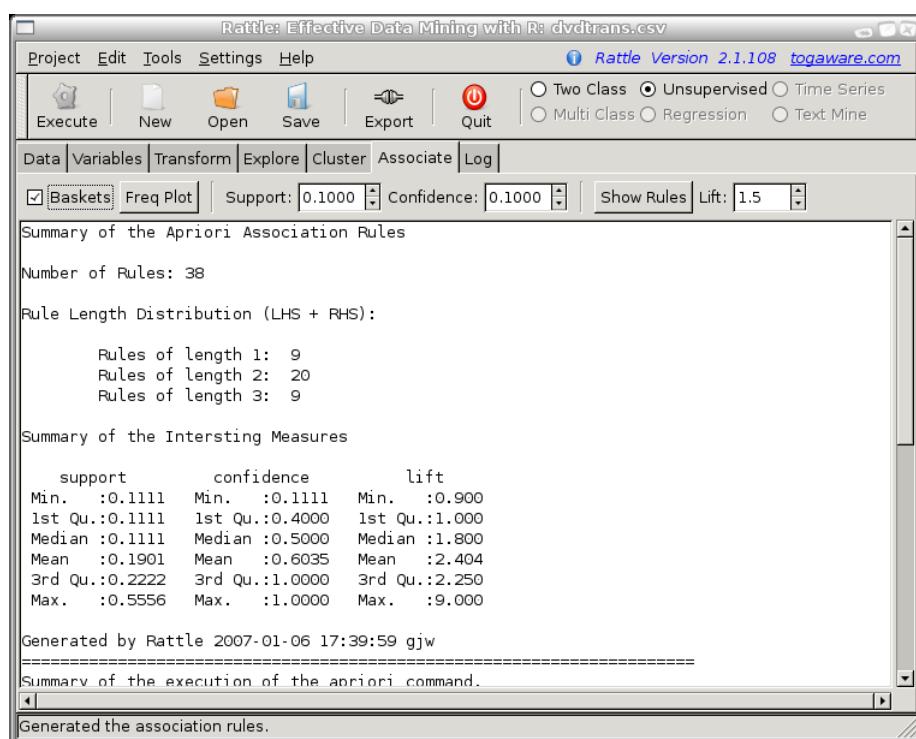
We load this data into Rattle and choose the appropriate variable roles. In this case it is quite simple: On the Associate tab (of the Unsupervised paradigm) ensure the **Baskets** check box is checked. Click the **Execute** button to identify the associations: Here we see a summary of the associations found. There were 38 association rules that met the criteria of having a minimum support of 0.1 and a minimum confidence of 0.1. Of these, 9 were of length 1 (i.e., a single item that has occurred frequently enough in the data), 20 were of length 2 and another 9 of length 3. Across the rules the support ranges from 0.11 up to 0.56. Confidence

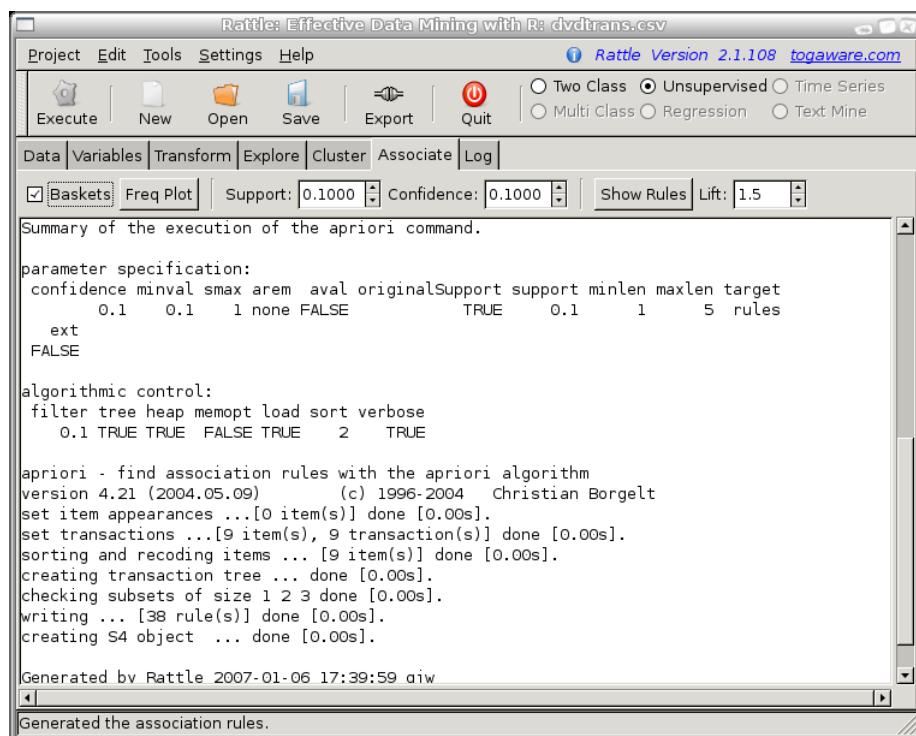


ranges from 0.11 up to 1.0, and lift from 0.9 up to 9.0.

The lower part of the same textview contains information about the running of the algorithm: We can see the variable settings used, noting that Rattle only provides access to a smaller set of settings (support and confidence). The output includes timing information for the various phases of the algorithm. For such a small dataset, the times are of course essentially 0!

8.3.2 General Rules





Togaware Watermark For Data Mining Survival

Chapter 9

Model Evaluation and Deployment

Togaware Watermark For Data Mining Survival

Evaluating the performance of model building is important. We need to measure how any model we build will perform on previously unseen cases. A measure will also allow us to ascertain how well a model performs in comparison to other models we might choose to build, either using the same model builder, or a very different model builder. A common approach is to measure the error rate as the proportional number of cases that the model incorrectly (or equivalently, correctly) classifies. Common methods for presenting and estimating the empirical error rate include confusion matrices and cross-validation.

The various approaches to measuring performance include Lift, area under the ROC curve, the F-score, average precision, precision/recall, squared error, and risk charts.

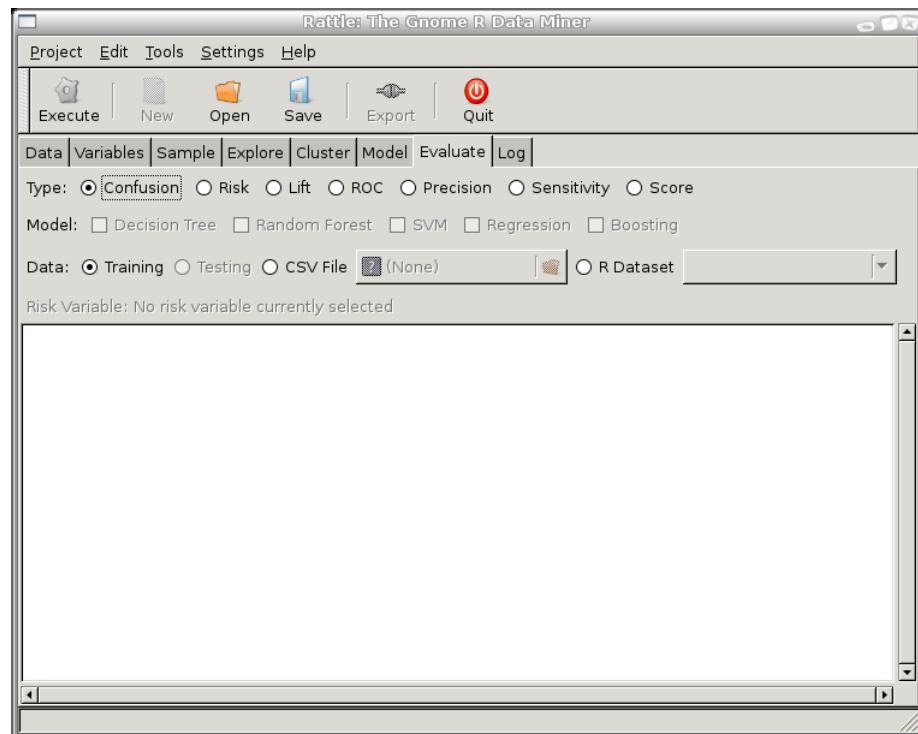
In this chapter we explore Rattle's various tools for reporting the performance of a model and it's various approaches to evaluating the output of data mining. We include the confusion matrix (using underneath the *table* function) for producing confusion matrices, Rattle's new Risk Chart for effectively displaying model performance including a measure of the success of each case, and we explore the use of the *ROCR* package for the graphical presentation of numerous evaluations, including those common approaches included in Rattle. *Moving in to R* illustrates how to fine the

presentations for your own needs.

This chapter also touches on issues around Deployment of our models, and in particular Rattle's Scoring option, which allows us to load a new dataset and apply our model to that dataset, and to save the scores, together with the identity data, to a file for actioning.

9.1 The Evaluate Tab

The Evaluate tab displays all the options available for evaluating the performance of our models, and for deploying the model over new datasets.



The range of different types of evaluations is presented as a series of radio buttons, allowing just a single evaluation type to be chosen at any time. Each type of evaluation is presented in the following sections of this chapter.

Below the row of evaluation types is a row of check boxes to choose the model types we wish to evaluate. The check boxes are only sensitive once a model has been built, and so on a fresh start of Rattle no model check box can be checked. As models are built, they will become sensitive, and as we move from the Model tab to this Evaluate tab the most recently built model will be automatically checked (and any previously checked Model choices will be unselected). This corresponds to a common pattern of behaviour, in that often we will build and tune a model, then want to explore its performance by moving to this Evaluate tab. If the All option has been chosen of the Model tab then all models that were successfully built will automatically be checked on the Evaluate tab.

To evaluate a model we need to identify a dataset on which to perform the evaluation.

The first option (but not the best option) is to evaluate our model on the training dataset. This is generally not a good idea, and the information dialogue shown here will be displayed each time we perform an evaluation on a training dataset. The output of any evaluation on the training dataset will also highlight this fact. The problem is that we have built our model on this training dataset, and it is often the case that the model will perform very well on that dataset! It should, because we've tried hard to make sure it does. But this does not give us a very good idea of how well the model will perform in general, on previously unseen data.

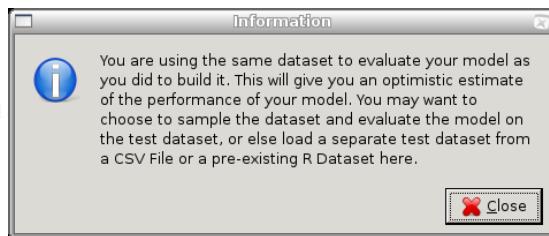


Figure 9.1: Informal dialog when using training set for evaluation

For a better guide to how well the model will perform in general, that is, on new and previously unseen data, we need to apply the model to such data and obtain an error rate. This error rate, and not the error rate from the training dataset, will then be a better estimate of how well the model will perform.

We discussed the concept of a training set in Section 4.1, presenting

the Sample option of the Select tab which provides a simple but effective mechanism for identifying a part of the dataset to be held separately from the training dataset, and to be used explicitly as the testing dataset. As indicated there, the default in Rattle is to use 70% of the dataset for training, and 30% for testing.

The final piece of information displayed in the common area of the Evaluate tab is the Risk Variable. The concept of the Risk Variable has been discussed in Section 4. It is used as a measure of how significant each case is, with a typical example recording the dollar value of the fraud related to the case. The Risk Chart makes use of this variable if there is one, and it is included in the common area of the Evaluate tab for information purposes only.

9.2 Confusion Matrix

Togaware Watermark For Data Mining Survival

9.2.1 Measures

True positives (TPs) are those entities which are correctly classified by a model as positive instances of the concept being modelled (e.g., the model identifies them as a case of fraud, and they indeed are a case of fraud). **False positives** (FPs) are classified as positive instances by the model, but in fact are known not to be. Similarly, **true negatives** (TNs) are those entities correctly classified by the model as not being instances of the concept, and **false negatives** (FNs) are classified as not being instances, but are in fact known to be. These are the basic measures of the performance of a model. These basic measures are often presented in the form of a **confusion matrix**, produced using a **contingency table**.

9.2.2 Graphical Measures

ROC graphs, sensitivity/specificity curves, lift charts, and precision/recall plots are useful in illustrating specific pairs of performance measures for classifiers. The *ROCR* package creates 2D performance curves from any two of over 25 standard performance measures. Curves from different cross-validation or bootstrapping runs can be averaged by different

methods, and standard deviations, standard errors or box plots can be used to visualize the variability across the runs. See `demo(ROCR)` and <http://rocr.bioinf.mpi-sb.mpg.de/> for examples.

9.3 Lift

Togaware Watermark For Data Mining Survival

9.4 ROC Curves

Area Under Curve

9.5 Precision versus Recall

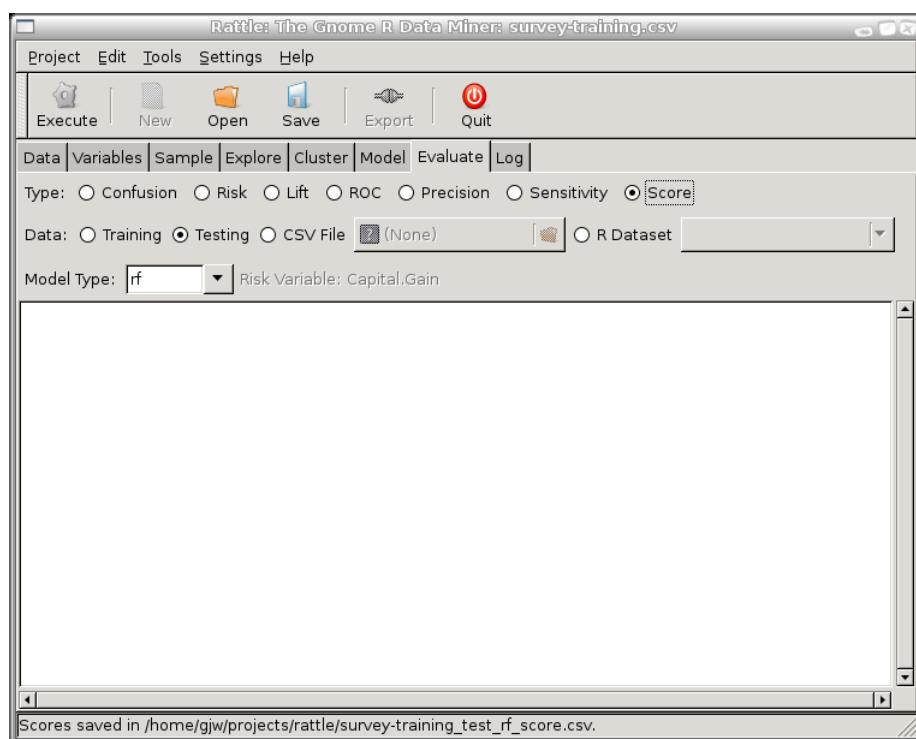
9.6 Sensitivity versus Specificity

Sensitivity is the true positive rate.

Specificity is the true negative rate.

9.7 Scoring

Togaware Watermark For Data Mining Survival



Often you will want to apply a model to a dataset to generate the scores for each entity in the dataset. Such scores may be useful for further exploration in other tools, or for actual deployment.

The Score radio button allows you to score (i.e., to generate probabilities for each entry in) a dataset. The specific dataset which is scored is that which is identified with the Data option. In the above example, the model will be used to score the Testing dataset. You can score the actual Training dataset, a dataset loaded from a CSV data file, or from a dataset already loaded into R.

Rattle will generate a CSV file containing the “scores” for the dataset. Each line of the CSV file will consist of a comma separated list of all of the variables that have been identified as *Idents* in the Variables tab, followed by the score itself. This score will be a number between 0 and 1.

Note the status bar in the sample screenshot has identified that the score file has been saved to the suitably named file. The file name is derived from name of the dataset (perhaps a source data csv filename or the name of an R data frame), whether it is a test or training dataset, the type of model and the type of score.

The output looks like:

```
ID,predict
98953270,0.104
12161980,NA
96316627,0.014
54464140,0.346
57742269,0.648
19307037,0.07
61179245,0.004
36044473,0.338
19156946,0.33
```

9.8 Calibration Curves

Togaware Watermark For Data Mining Survival

Chapter 10

Issues

We consider in this chapter the general issues with model building, such as selecting the best model builder, over-fitting models, and dealing with imbalanced classes.

10.1 Model Selection

The question that obviously now comes to mind is which model builder do we use. That is a question that has challenged us for a long time, and still there remains no definitive answer. It all depends on how well the model builder works on your data, and, in fact, how you measure the performance on the model. We review some of the insights that might help us choice the right model builder and, indeed, the right model, for our task.

Contrary to expectations, there are few comprehensive comparative studies of the performance of various model builders. A notable exception is the study by Caruana and Niculescu-Mizil, who compared most modern model builders across numerous datasets using a variety of performance measures. The key conclusion, they found, was that boosted trees and random forests generally perform the best, and that decision trees, logistic regression and boosted stumps generally perform the worst. Perhaps more importantly though, it often depends on what is being measured

as the performance criteria, and on the characteristics of the data being modelled.

An overall conclusion from such comparative studies, then, is that often it is best to deploy different model builders over the dataset to investigate which performs the best. This is better than a single shot at the bullseye. We also need to be sure to select the appropriate criteria for evaluating the performance. The criteria should match the task at hand. For example, if the task is one of information retrieval then a Precision/Recall measure may be best. If the task is in the area of health, then the area under the ROC curve is an often used measure. For marketing, perhaps it is lift. For risk assessment, the Risk Charts are a good measure.

So, in conclusion, it is good to build multiple models using multiple model builders. The tricky bits are tuning the model builders (requiring an understanding of the sometimes very many and very complex model builder parameters) and selecting the right criteria to assess the performance of the model (a criteria to match the task at hand—noting that raw accuracy is not always, and maybe not often, the right criteria).

10.2 Overfitting

Overfitting is more of a problem when training on smaller datasets.

A characteristic of the random forest algorithm is that it will often overfit the training data. For any model builder this, at first, may be a little disconcerting, with the usual thought that therefore the model will not generalise to new data. However, for random forests, this overfitting is not usually a problem. Applying the model to a test dataset will usually indicate that it does generalise quite well, and that it does not suffer from the usual consequence of a model that has overfit the training dataset.

10.3 Imbalanced Classification

Model accuracy is not such an appropriate measure of performance when the data has a very imbalanced distribution of outcomes. For example, if positive cases account for just 1% of all cases, as might be the situation in

an insurance dataset recording cases of fraud or in medical diagnoses for rare but terminal diseases, then the most accurate, but most useless, of models is one that predicts no fraud or diagnoses no disease in all cases. It will be 99% accurate! In such situations, the usual goal of the model builder, which is to build the most accurate model, does not match the actual goal of the model building.

There are two common approaches to dealing with imbalance: sampling and cost sensitive learning.

Before describing these two approaches to dealing with this issue, it is worth noting that some algorithms have no difficulty with building models from training data with imbalanced classes. Random forests, for example, need no such treatment of the training data in order to build models that capture under-represented classes quite well.

10.3.1 Sampling

A traditional approach to solving class imbalance, and one that works well in many modelling situations, but not all, is to sample your data. Sampling aims to remove or at least redress the balance. It is a data preprocessing step whereby the algorithm used by the model builder does not generally need to be modified. Because of this, the approach is readily applicable (but not necessarily appropriate) to any model builder.

MENTION APPROACHES AND FOR EACH ILLUSTRATE HOW TO DO IT IN R.

Random undersampling will randomly choose a subset of the over represented class (or classes) to approach the same number as the underrepresented class (or classes) for inclusion in the training dataset.

Random oversampling will randomly duplicate records from the underrepresented class (or classes) for inclusion in the training dataset.

The synthetic minority oversampling technique (SMOTE).

Cluster-based oversampling.

One-sided selection.

Wilson's editing.

10.3.2 Cost Based Learning

An alternative is to use cost sensitive learning where the algorithm used by the model builder itself is modified. This approach introduces numeric modifications to any formula used to estimate the error in our model. Mis-classifying a positive example as a negative, a false negative, (e.g., identifying a fraudulent case as not fraudulent) is more "costly" than a false positive. In health, for example, we do not want to miss cases of true cancer, and might find it somewhat more acceptable to momentarily investigate cases that turn out not to be cancer, simply because, missing the cancer may lead to premature death. A model builder will take into account the different costs of the outcome classes and build a model that does not so readily dismiss the very much under represented outcome.

10.4 Model Deployment and Interoperability

10.4.1 SQL

SQL is a structured query language for summarising and reporting on data in a database or data warehouse. Some organisations have attempted to use SQL for the implementation of models. Such attempts have not been successful, in general. SQL is not well suited to efficiently implementing models. Simple models might be okay, but practical models present challenges.

Why would one consider SQL? The data warehouse vendor, Teradata, for example, provides the Teradata Warehouse Miner toolkit which includes some basic, but not sophisticated, data mining tools.

SQL DEPLOYMENT OF MODELS IN TWM

10.4.2 PMML

INTRO PMML.

TWM IMPORTS PMML.

SAS GENERATE PMML.

Rattle EXPORTS PMML. BUT RF PMML IS 250,000 LINES OF SQL

10.5 Bibliographic Notes

Togaware Watermark For Data Mining Survival

Togaware Watermark For Data Mining Survival

Chapter 11

Moving into R

R is a statistical programming language together with an extensive collection of packages (of which Rattle is one) that provide a comprehensive toolkit for, amongst other things, data mining. Indeed, R is extensively deployed in bio-informatics, epidemiology, geophysics, agriculture and crop science, ecology, oceanography, fisheries, risk analysis, process engineering, pharmaceutical research, customer analytics, sociology, political science, psychology, and more.

One of the goals of Rattle is to ease a user's transition into using R directly, and thereby unleashing the full power of the language. R is a relatively simple language to learn, but has its own idiosyncrasies that emerge in any language that has such a long history, as R does.

In this chapter we discuss how we can access the internal data structures used by Rattle. This then allows us to smoothly switch between using Rattle and R in a single session. We significantly augment the functionality encapsulated in Rattle through this direct interaction with R, whilst not losing the ability to quickly explore the results in Rattle.

11.1 The Current Rattle State

Internally, Rattle uses the variable `crs` to store the current rattle state. We can have a look at the structure of this variable at any time using

the *str* function in the R Console.

- crs\$ident** List of variable names treated as identifiers.
- crs\$input** List of variable names for input to modelling.
- crs\$target** The name of the variable used as target for modelling.

Togaware Watermark For Data Mining Survival

11.2 Data

Our example of loading data into Rattle from a CSV file simply uses the R function *read.csv*.

11.3 Samples

Rattle uses a simple approach to generating a partitioning of our dataset into training and testing datasets with the *sample* function.

```
crs$sample <- sample(nrow(crs$dataset), floor(nrow(crs$dataset)*0.7))
```

The first argument to *sample* is the top of the range of integers you wish to choose from, and the second is the number to choose. In this example, corresponding to the *audit* dataset, 1400 (which is 70% of the 2000 entities in the whole dataset) random numbers between 1 and 2000 will be generated. This list of random numbers is saved in the corresponding Rattle variable, `crs$sample` and used throughout Rattle for selecting or excluding these entities, depending on the task.

To use the chosen 1400 entities as a training dataset, we index our dataset with the corresponding Rattle variable:

```
crs$dataset[crs$sample,]
```

This then selects the 1400 rows from `crs$dataset` and all columns.

Similarly, to use the other 600 entities as a testing dataset, we index our dataset using the same Rattle variable, but in the negative!

```
crs$dataset[-crs$sample,]
```

Each call to the *sample* function generates a different random selection. In Rattle, to ensure we get repeatable results, a specific seed is used each time, so that with the same seed, we obtain the same random selection, whilst also providing us with the opportunity to obtain different random selections. The *set.seed* function is called immediately prior to the *sample* call to specify the user chosen seed. The default seed used in Rattle is arbitrarily the number 123:

```
set.seed(123)
crs$sample <- sample(nrow(crs$dataset), floor(nrow(crs$dataset)*0.7))
```

In moving into R we might find the *sample.split* function of the *caTools* package handy. It will split a

vector into two subsets, two thirds in one and one third in the other, maintaining the relative ratio of the different categorical values represented in the vector. Rather than returning a list of indices, it works with a more efficient Boolean representation:

```
> library(caTools)
> mask <- sample.split(crs$dataset$Adjusted)

> head(mask)
[1] TRUE TRUE TRUE FALSE TRUE TRUE

> table(crs$dataset$Adjusted)
 0   1 
1537 463

> table(crs$dataset$Adjusted[mask])
 0   1 
1025 309

> table(crs$dataset$Adjusted[!mask])
 0   1 
512 154
```

Perhaps it will be more convincing to list the proportions in each of the groups of the target variable (rounding these to just two digits):

```
> options(digits=2)

> table(crs$dataset$Adjusted) /
  length(crs$dataset$Adjusted)
 0   1 
0.77 0.23

> table(crs$dataset$Adjusted[mask]) /
  length(crs$dataset$Adjusted[mask])
 0   1 
0.77 0.23

> table(crs$dataset$Adjusted[!mask]) /
  length(crs$dataset$Adjusted[!mask])

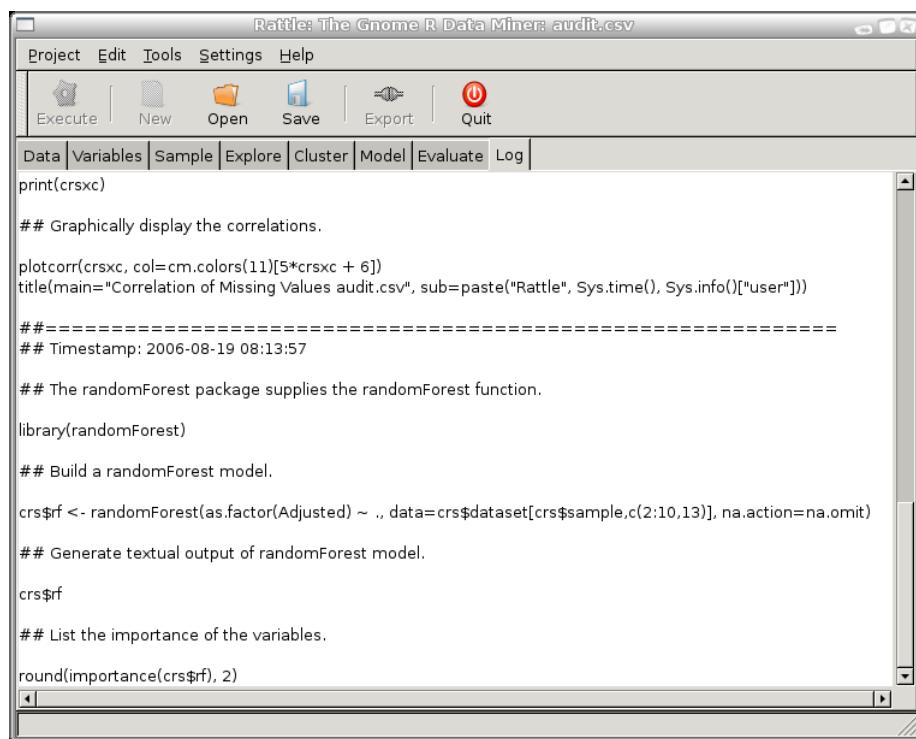
 0   1 
0.77 0.23
```

Thus, using this approach, both the training and the testing datasets will have the same distribution of the target variable.

11.4 Projects

We have illustrated in this chapter how to save Rattle's current state of affairs to a `.rattle` file. This file is in fact a standard R file, and is simply saving a single R object called `crs` (for Current Rattle State). The object has many slots, storing the dataset, the models, and various option settings.

11.5 The Rattle Log



The screenshot shows the Rattle application window titled "Rattle: The Gnome R Data Miner: audit.csv". The window has a menu bar with Project, Edit, Tools, Settings, Help, and a toolbar with Execute, New, Open, Save, Export, and Quit buttons. Below the toolbar is a tab bar with Data, Variables, Sample, Explore, Cluster, Model, Evaluate, and Log tabs. The Log tab is selected and contains the following R code and its output:

```

Rattle: The Gnome R Data Miner: audit.csv
Project Edit Tools Settings Help
Execute New Open Save Export Quit
Data Variables Sample Explore Cluster Model Evaluate Log
print(crs$rc)

## Graphically display the correlations.

plotcorr(crs$rc, col=cm.colors(11)[5*crs$rc + 6])
title(main="Correlation of Missing Values audit.csv", sub=paste("Rattle", Sys.time(), Sys.info()["user"]))

#####
## Timestamp: 2006-08-19 08:13:57

## The randomForest package supplies the randomForest function.

library(randomForest)

## Build a randomForest model.

crs$rf <- randomForest(as.factor(Adjusted) ~ ., data=crs$dataset[crs$sample,c(2:10,13)], na.action=na.omit)

## Generate textual output of randomForest model.

crs$rf

## List the importance of the variables.

round(importance(crs$rf), 2)

```

All R commands that Rattle runs underneath are exposed through the text view of the Log tab. The intention is that the R commands be available for copying into the R console so that where Rattle only exposes a limited number of options, further options can be tuned via the R console.

The **Log** tab aims to be educational as much as possible. Informative comments are included to describe the steps involved.

Also, the whole log can be saved to a script file (with a **R** filename extension) and in principle, loaded into **R** to repeat the exact steps of the **Rattle** interactions. In general, you may want to review the steps and fine tune them to suit your purposes. After pasting the contents of the Log text view into a file, perhaps with a filename of **audit-rf-risk.R**, you can have the file execute as a script in **R** with:

```
> source("audit-rf-risk.R")
```

Internally, **Rattle** uses a variable called **crs** to store its current state, and you can modify this variable directly. Generally, changes you make will be reflected within **Rattle** and vice versa.

We can export the Log to an **R** script file (with the **.R** extension) using the **Export** button.

11.6 Further Tuning Models

One of the goals of **Rattle** is to keep things simple for the user. Consequently, not all options available for many of the functions provided by **R** are exposed through the **Rattle** user interface. This is not meant to be a limitation though, and **Rattle** is quite at ease working with modifications you make to the **crs** data structure within the **R** Console, at least to quite some extent.

Suppose for example that you wish to build an ada model using the **x** and **y** arguments rather than the **formula** argument. First, within **Rattle**, build the normal ada model and go to the **Log** tab to highlight and copy the command used:

```
crs$ada <- ada(Adjusted ~ .,
                 data=crs$dataset[crs$sample,c(2:4,6:10,13)],
                 control=rpart.control(maxdepth=30, cp=0.010000,
                                       minsplit=20, xval=10),
                 iter=50)
```

Now past that into the **R** Console, but modify it appropriately:

```
crs$ada <- ada(crs$dataset[crs$sample,c(2:4,6:10)],
```

```
crs$dataset[crs$sample,c(13)],  
control=rpart.control(maxdepth=30, cp=0.010000,  
minsplit=20, xval=10),  
iter=50)
```

You can now go back to the Rattle window's Evaluate tab and evaluate the performance of this new model. Indeed, you can, if you choose, save the models to different variables in the R Console, and selectively copy them into `crs$ada` and then evaluate then with Rattle. Of course, the alternative is to copy the R commands for the evaluation from the Log tab of Rattle and paste them into the R console and perform the evaluation programmatically.

Togaware Watermark For Data Mining Survival

Togaware Watermark For Data Mining Survival

Chapter 12

Troubleshooting

Rattle is open source, freely available, software, and benefits from user feedback and user contributions. It is also under constant development and improvement. Whilst every effort is made to ensure Rattle is error and bug free, there is always the possibility that we will find a new bug. This is then the opportunity for us to review the code and to improve the code. At the very least though, if you find a problem with Rattle, contact the author to report the problem. At best, feel free to hunt down the source of the problem in the R code and provide the solution to the author, for all to benefit.

In this chapter we record some known issues that you may come across in using Rattle. These aren't bugs as such but rather known issues that a data miner is likely to come across in building models.

12.1 Cairo: cairo_pdf_surface_create could not be located

A number of users have reported (April 2007) the following error when trying to display using the Cairo Device:

```
The procedure entry point cairo_pdf_surface_create could not be  
located in the dynamic link library libcairo-2.dll  
  
Error in dyn.load(x, as.logical(local), as.logical(now)) :
```

```
unable to load shared library  
'C:/PROGRA~1/R/R-24~1.1/library/cairoDevice/libs/cairoDevice.dll':  
LoadLibrary failure: The specified procedure could not be found.
```

The reason for the error is currently unknown. A work-around is to not use the Cairo device for plotting. This can be set in the Settings menu, or else simply remove the cairoDevice package and Rattle will default to using the Windows device.

12.2 A factor has new levels

This occurs when the training dataset does not contain examples of all of the levels of particular factor and the testing set contains examples of these other levels.

Togaware Watermark For Data Mining Survival

Part II

R for the Data Miner

Togaware Watermark For Data Mining Survival

Chapter 13

R: The Language

R is a statistical and data mining package consisting of a programming language and a graphics system. It is used throughout this book to illustrate data mining procedures. It is the programming language used to implement the Rattle graphical user interface for data mining in Chapter 2, page 7. If you are moving to R from SAS or SPSS, then you may find the document <http://oit.utk.edu/scc/RforSAS&SPSSusers.doc> helpful.

In the following sections of this chapter we introduce the basics of R. We will find many examples presented which can be readily copied into an R console to facilitate learning. You will also find many examples on the R-help mailing list at <https://stat.ethz.ch/mailman/listinfo/r-help>.

Learning by example is a powerful learning paradigm. Motivated by the programming paradigm of “programming by example” (Cypher, 1993), the intention is that you will be able to replicate the examples from the book, and then fine tune them to suit your own needs. This, of course, is also one of the underlying principles of Rattle, as described in Chapter 2, page 7, where all of the R commands that are used under the graphical user interface are exposed to the user. This makes it a useful teaching tool in learning R for the specific task of data mining, and also a good memory aid!

So R is a language. The basic modus operandi is to write sentences

expressed in this language. After a while you will want to do more than to issue single, simple, commands (sentences), but to write sentences and paragraphs and full novels in the language! R script files (often with the R filename extension) are the place to write scripts. You can re-run your scripts to transform, at will and automatically, your source data into information and knowledge.

This chapter begins with an overview of some of the key advantages (and disadvantages) of using R and continues with a guide to interacting with R. For data mining purposes the recommended interface is the simple to use Rattle (Chapter 2), although more advanced users will prefer the powerful Emacs editor, augmented with the ESS package. Both run under GNU/Linux, Mac/OSX, and MS/Windows. This is a personal preference and you may prefer some of the alternatives we discuss—this freedom of choice is yours.

Direct interaction with R has a steeper learning curve than using GUI based systems, but once into R, performing operations over the same or similar datasets becomes very easy using its programming language interface. For the R beginner, using a GUI like Rattle, where all underlying R commands are available for your perusal and direct pasting into R itself, may be a good first step.

Let's start with some of the advantages with using R:

- R is licensed under the GNU General Public License, with Copyright held by The R Foundation for Statistical Computing. Thus, it is Free Open Source Software, freely available, so that anyone can freely download and install the software and even freely modify the software, or look at the code behind the software to learn how things can be done. Indeed, anyone is welcome to provide bug fixes, code enhancements, and new packages, and the wealth of quality packages available for R is a testament to this approach to software development and sharing.
- R probably has the most complete collection of statistical functions of any statistical or data mining package. New technology and ideas often appear first in R.
- The graphic capabilities of R are outstanding, providing a fully

programmable graphics language which surpasses most other statistical and graphical packages.

- A very active email list, with some of the worlds leading statisticians actively responding, is available for anyone to join. Questions are quickly answered and the archive provides a wealth of user solutions and examples. Be sure to read the [Posting Guide](#) first.
- Being open source the R source code is peer reviewed, and anyone is welcome to review it and suggest improvements. Bugs are fixed very quickly. Consequently, R is a rock solid product. New packages provided with R do go through a life cycle, often beginning as somewhat less quality tools, but usually quickly evolving into top quality products.
- R plays well with many other tools, importing data, for example, from CSV files, SAS, and SPSS, or directly from MS/Excel, MS/Access, Oracle, MySQL, and SQLite. It can also produce graphics output in PDF, JPG, PNG, and SVG formats, and table output for L^AT_EX and HTML.

Whilst the advantages might flow from the pen with a great deal of enthusiasm, it is useful to note some of the disadvantages or weaknesses of R, even if they are perhaps transitory!

- R is not so easy to use for the novice. There are several simple to use graphical user interfaces (GUIs) for R that encompass point and click interactions, but they generally do not have the polish of the commercial offerings of Clementine (Chapter 48, page 547) and SAS/Enterprise Miner (Chapter 53, page 559).
- Documentation is sometimes patchy. Whilst there are extensive documents on line and available in books and throughout the Internet, it can sometimes be terse and even impenetrable to the non-statistician. On the other hand, for example, SAS has extensive, self-contained, and often well explained, documentation, readily available to the user. Nonetheless, users do comment that the R documentation is to the point and easy to consult.

- The quality of some packages is less than perfect, although if a package is useful to many people, it will quickly evolve into a very robust product through collaborative efforts.
- There is no one to complain to if something doesn't work - at least no one who has a financial interest in keeping you, the user, as a satisfied customer. Organisations are quite happy to pay major premiums for that apparent peace of mind! Nonetheless, problems are usually dealt with quickly on the mailing list, and bugs disappear with lightning speed.

The remaining sections of this chapter can generally be skipped on a reading through the book, particularly if you are using Rattle. They provide a basic reference guide to using R, and in particular some of its programming capabilities. While chapter 3 deals in detail with creating data in R, we introduce some of the basics here. The most basic needs include creating simple datasets, and being familiar with the basic data types and programming concepts, and how to get help.

13.1 Obtaining and Installing R

CRAN is the Comprehensive R Archive Network where you will find the R installation package and all verified R packages. Installation of R is then straightforward, either from source or from a pre-compiled binary package. The installation of R itself is covered in Section 2.1, page 8.

Be aware though that you can even run R without installing it! The Quantian live CD can be used to boot your computer and run GNU/Linux from the CD, from where you can start up R and run it over data on your hard disk. All of this can be done without touching what ever operating system is already installed on your hard disk. If you like what you see (and it will run slower than a proper install) you can then install it properly.

We briefly review the installation of R here.

13.1.1 Installing on Debian GNU/Linux

In short, on a Debian GNU/Linux system, the premier GNU/Linux system, R has been packaged by Dirk Eddelbuettel, and is part of the main Debian distribution. To install it simply install the **r-recommended** package using any of Debian's package management tools. For example, enter the following command into a terminal window:

```
$ wajig install r-recommended
```

This will install a basic system. There are, though, over 95 individual R packages to add to your installation as you need. These include, for example:

```
$ wajig install r-cran-cluster r-cran-hmisc r-cran-rpart
```

To install most of those available:

```
$ wajig listnames ^r- > Rpackages
$ wajig install -file Rpackages
```

Other packages can be installed from the CRAN archives directly using the *install.packages* function. See Section 13.6.2, page 178 for details.

13.1.2 Installing on MS/Windows

To install R under MS/Windows, download the self installing package from <http://cran.us.r-project.org/bin/windows/base/R-2.4.1-win32.exe> (the version number changes each 6 months) or a CRAN site near you. Simply run the installer and R will be appropriately installed. To install individual packages use the *install.packages* function (Section 13.6.2, page 178).

13.1.3 Install MS/Windows Version Under GNU/Linux

Installing the MS/Windows version of R on a GNU/Linux machine seems like a very odd thing to do, but there are times when you may need to share a high quality graphics file in a format that your MS/Windows limited colleagues can use. Only the MS/Windows version of R can generate MS/Windows Metafiles (with filename extension **wmf** and **emf**).

As with many things in GNU/Linux, which is all about freedom, you can install the MS/Windows version of R using the GNU/Linux package called Wine. The steps are (replace <http://cran.au.r-project.org/> with an archive near you—see <http://cran.r-project.org/mirrors>):

```
$ wget http://cran.au.r-project.org/bin/windows/base/R-2.4.1-win32.exe
$ wine R-2.3.1-win32.exe
$ wine ~/wine/fake_windows/Program\ Files/R/2.4.1/bin/Rgui.exe
```

The resulting window will be running the MS/Windows application on your GNU/Linux desktop.

Inside the Rgui you can create a MS/Windows Metafile image in this way:

```
> win.metafile("sample.emf")
> plot(iris$Petal.Length, iris$Petal.Width)
> dev.off()
```

But now we are ahead of ourselves! These three lines illustrate sentences that we write in order to command R to do things for us. The ‘>’ is a prompt which indicates that R is waiting for our instructions. We type in what follows (e.g., *plot*) which instructs R to produce a plot. The information between the brackets tell R what to plot—they are the command arguments.

13.2 Interacting With R

The basic paradigm for interacting with R is that of writing sentences in a language. For this you need an editor, and ideally one that supports: R syntax highlighting in colour; parenthesis checking; command completion; and code evaluation by R. Such an editor, and highly recommended, is Emacs with the ESS package. Which ever editor you prefer, a favoured mode of operation is to write your R sentences, using your editor, into a file, and then ask R to evaluate the instructions you have provided. Such a practise will ensure you work efficiently to capture the results of your data understanding, data cleaning, data transformations, and data mining. It will also ensure your work is repeatable, and as your data changes, you can simply re-run your processes, as expressed in your script files.

While graphical user interfaces (GUIs) provide an easy path into using a tool, you very quickly lose the ability to capture your processes and, by staying within the GUI you quickly find that you have limited functionality and flexibility that a full language would provide. Indeed, you also tend to end up not understanding what it is you are doing, and can easily fall into statistical traps! GUIs are good for helping remember the commands to perform specific tasks but a GUI can often end up getting in the way, rather than helping.

Compare this to writing books which still fundamentally involves putting words into sentences in a document. So it is with writing data mining stories. The tools available provide much help in writing our stories, but still we need to put the sentences together.

And like any story, we will be writing them for others to read, not just for the computer to evaluate. So always write your R code with the intention that others will want to read it. They will!

Togaware Watermark For Data Mining Survival

13.2.1 Basic Command Line

R is essentially a command line tool that is usually initiated by running the command R in a command line window (e.g., a gnome-terminal) on your system. When R is ready to accept your instructions it will issue the > prompt, and then wait for your input. Figure 13.1 shows that a user has invoked the *nrow* function with argument *iris* to find the number of rows in the *iris* dataset. R has responded with an answer of 150

The basic MS/Windows command line GUI provides a menu-based system to access some of the meta-functionality of R, such as to load and install packages. By default it uses a multiple document interface (MDI) so that all R windows open up in a single Rgui frame. To use a Single Document Interface (SDI) choose the appropriate option under *Single or multiple windows*, which you can find under the *GUI preferences* of the *Edit* menu. Then save the configuration in the default *Rconsole* file, and restart R.

To interrupt a running R command simply type **Ctrl-C**. To exit from R use the *q* function:

```
> q()
Save workspace image? [y/n/c]: n
```

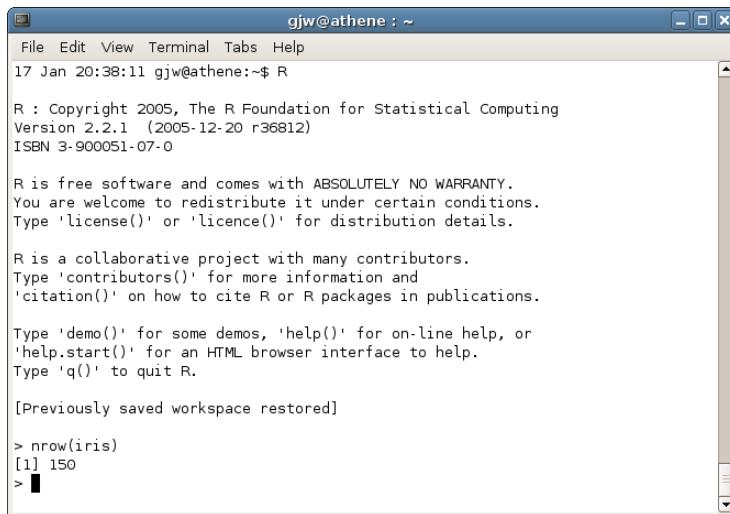


Figure 13.1: The R Command Line is the basic interface to R.

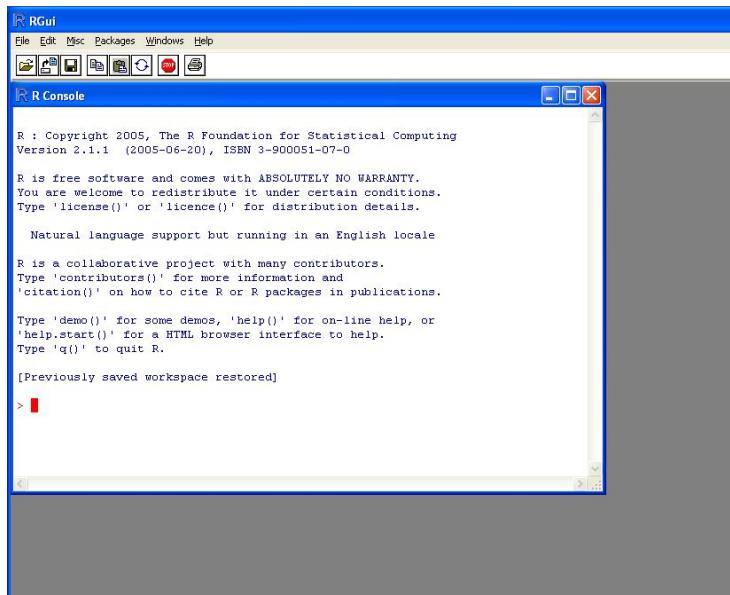


Figure 13.2: The R interface under MS/Windows, with the default MDI display.

You are given the option to save all of the currently defined objects (the workspace image), which will automatically be reloaded next time you start R from this same location (directory). If you choose to do so R will save the workspace image to a file called `.RData`.

13.2.2 Emacs and ESS

We discussed above the philosophy of using an editor to interact with R, and thus fortuitously saving our work to file. Numerous editor based interfaces are available for R, and a good choice is Emacs with the ESS package.

Emacs, as often it does, provides the most sophisticated access to R, through the use of the ESS Emacs package, providing a simple mechanism to type R commands into a file and have them executed by R on request. Figure 13.3 illustrates the basic interface. After starting Emacs, load in a file (or create a new file) with a name ending in R. With the ESS package for Emacs installed (for example, installing the `ess` package on Debian GNU/Linux) you will see an empty window with an R toolbar, similar to Figure 13.3.

Initiate an R subprocess in Emacs by clicking on the R icon. You'll be asked for a folder for R to treat as its default location for storing and reading data. A buffer named `*R*` will display and you can type R functions directly to have them evaluated. Switching back to an R file buffer (or, as in the figure, splitting the window to display both buffers) will allow you to type R functions into the buffer and have them evaluated on request. The series of icons to the right of the R and SPlus icons allow the functions in the file buffer to be executed in the R subprocess buffer. From left to right they are: evaluate the current line (the arrow and single line); evaluate the currently highlighted region; load the file into R; or evaluate the current function. Simply clicking one of these icons will cause the R commands to be evaluated. It is a simple yet effective interface.

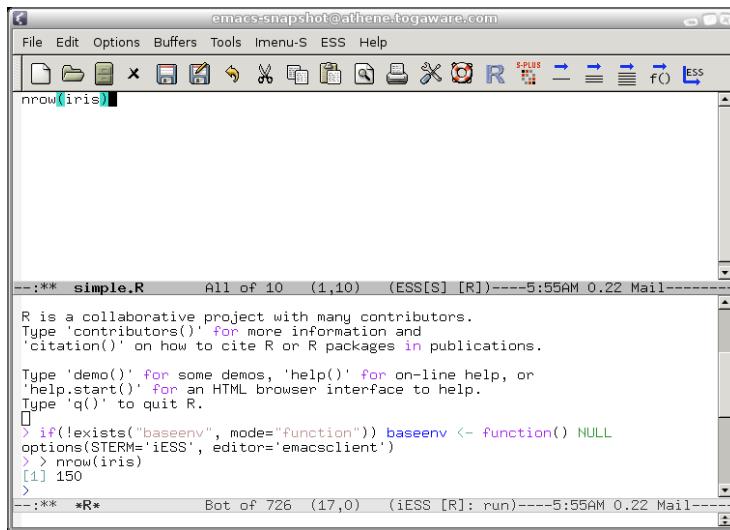


Figure 13.3: The ESS GUI interface to R within Emacs, showing the edit window on top, where R code can be constructed and saved to file and requested to be evaluated. The R window below is where the R code is evaluated and its output is displayed.

13.2.3 Windows, Icons, Mouse, Pointer—WIMP

Graphical interfaces might be good for a while, but soon become restrictive. Nonetheless, people like to get into new tools graphically, so we will introduce a common option within R, using *Rcmdr*, whilst noting that *rattle* (Chapter 2, page 7) is an alternative, specifically targeted for data mining.

Rcmdr is the most feature full R GUI. To start it up, simply load the package:

```
> library(Rcmdr)
```

A basic GUI interface with split screens will appear, as in Figure 13.4.

The top window in the GUI records the commands or functions that are to be sent to R. You can edit the commands here or else use the menus to generate the commands for you. The lower window shows the commands as they are evaluated and their output.

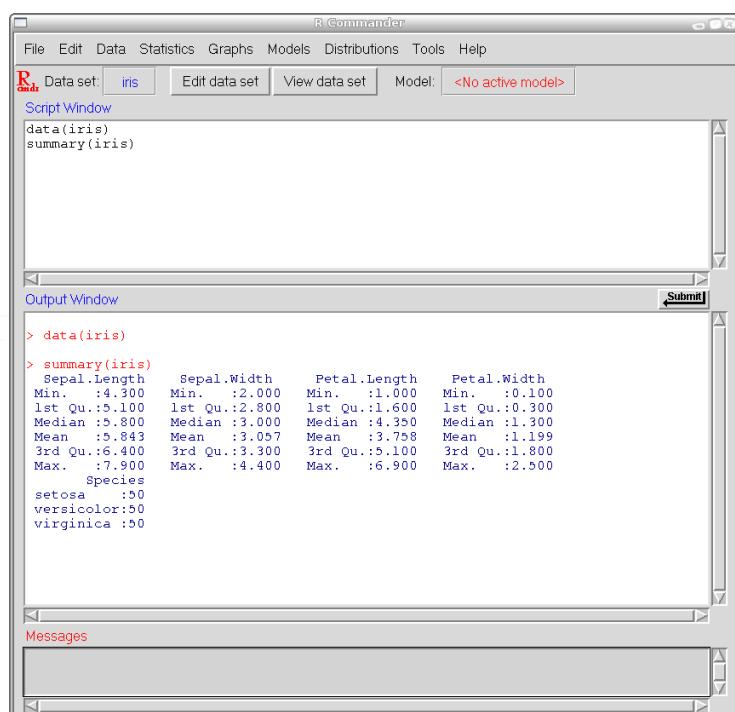


Figure 13.4: The R Commander GUI provides a simplified interface to using R. The interface consists of the top edit window and the lower execution and results window.

The GUI provides an excellent pathway into learning how to use R. Using various menus and resulting pop-up windows to build R functions can be very instructive. For some, the *Rcmdr* may be all that is needed for using R, allowing commands to be entered directly and then saved to script files, either as a record of processing and modelling, or else for later automatic regeneration of results.

Togaware Watermark For Data Mining Survival

13.3 Evaluation

R is an [interpreted language](#) providing procedural, [functional](#), and object oriented paradigms. The basic mode of interacting with R has the user typing commands (function names and arguments) and R evaluating them to return an answer (or an object). The object returned is often a vector (a collection of objects all of the same data type). The elements of the vector are numbered, beginning with 1, as indicated within the square brackets of the output. The returned elements are printed at the beginning of the line following the commands you type, but for brevity we include them as comments in our examples here (a comment is introduced with a # and continues to the end of a line):

```
> 5+2          # [1] 7
> 5-2          # [1] 3
> 5*2          # [1] 10
> 5/2          # [1] 2.5
> 5%2          # [1] 1      Remainder after division
> 5^2          # [1] 25
> 5^2-2*3     # (5^2) - (2*3)      [1] 19
```

All of these examples use in place function names (i.e., operators like + and -) with the arguments on either side and returns a vector of length 1.

Documentation relating to operator precedence is available with:

```
> help(Syntax)
```

R is most typically seen as a [functional language](#), meaning that it works by calling functions, such as *log* (for calculating the logarithm of a number), and returning values from these functions. Functions often require arguments that give additional information to the function. Some arguments are optional, and can be introduced through giving the name of the argument, which can be abbreviated, or missing altogether using the position to indicate the intent of the argument.

```
> log(1000)        # 6.907755
> log(1000, base=10)    # 3
> log(1000, b=10)      # 3 Command arguments can be abbreviated
> log(1000, 10)        # 3 Command arguments can be positional
```

In fact, even the in-place operators we saw above (e.g., 5+2) are just a syntactic abbreviation for a function call:

```
> "+"(5,2)      # In-place operators are a syntactic convenience
```

13.4 Help

R provides extensive on-line documentation, some of which is very good, but there is a degree of variability and often the novice is left with a pointer to a paper in a research journal that requires a degree in statistics to read! But, as is the power of open source software, anyone is welcome to improve the situation by contributing better documentation. Also, the on-line Wikipedia is becoming a great tool for understanding the basic concepts.

You can view documentation either within the R window or else through an external web browser through the call to *help.start*:

```
> help.start()
```

Once this call has been made, all following help requests are displayed in the web browser. You can turn this off with the *htmlhelp* option:

```
> options(htmlhelp=FALSE)
```

Basic documentation is available through the *help* function. Preceding any function name with a question mark (?) will also invoke the *help* function. The *str* and *args* functions are also useful in listing the signature of the function:

```
> help(scale)    # Display the manual page for the scale function.
> ?scale        # Same as help(scale).
> str(scale)    # Show the structure of the function call.
function (x, center = TRUE, scale = TRUE)
> args(scale)
function (x, center = TRUE, scale = TRUE)
NULL
```

To obtain a basic overview of a package you can use the *help* option of the *library* function (see Section 13.6 for information on packages):

```
library(help=maptools)
```

Often, more detailed documentation is provided through vignettes. To list all the available vignettes call the *vignette* function without any arguments. Then to view a specific vignette simply pass it as an argument.

Of particular use is the combination of the R function *edit* with *vignette* which will place you in a text editor with the sample code from the vignette, giving you an opportunity to try the code out.

```
> vignette()
Vignettes in package 'arules':
[...]
> vignette("arules")
> edit(vignette("arules"))
```

Most packages provide examples of their usage with the basic documentation. You can run the examples through R using the *example* function:

```
> example(persp)
```

If you are looking for a specific function, you might want to use the *help.search* function to search through R packages that are installed on your system, or the *RSiteSearch* function to search the CRAN archive:

```
> help.search("lda")
> RSiteSearch("lda", restrict="function")
```

The following rely on the function being identified in the current session (i.e., they are already attached):

```
> find(lda)          #
> apropos("abc")      # List objects matching the string.
> getAnywhere(lda)    # Provides useful description of source.
```

Being an object-oriented language, R provides multiple implementations for numerous functions (or methods). For example, which *plot* function is called is determined according to the type of object being plotted. If it is an *rpart* object, then *plot.rpart* is actually called up to do the plotting. For any such function you can get a list of candidate methods with the *methods* function.

```
> methods(plot)
 [1] plot.Date*           plot.HoltWinters*   plot.POSIXct*
 [4] plot.POSIXlt*        plot.TukeyHSD       plot.acf*
 [7] plot.data.frame*     plot.decomposed.ts* plot.default
[10] plot.dendrogram*    plot.density        plot.ecdf
[13] plot.factor*         plot.formula*      plot.hclust*
[16] plot.histogram*     plot.isoreg*        plot.lm
[19] plot.medpolish*     plot.mlm           plot.ppr*
[22] plot.prcomp*        plot.princomp*    plot.profile.nls*
[25] plot.rpart*          plot.spec          plot.spec.coherency
[28] plot.spec.phase     plot.stepfun      plot.stl*
```

```
[31] plot.table*          plot.ts            plot.tskernel*
Non-visible functions are asterisked
```

Noting that the open square bracket is actually an operator which extracts or replaces parts of an object, we can get help on such operators, and their specific methods, using the same notation:

```
> ?"["
# Help on the [ operator
> ?"[.factor"
# Help on the [ operator when applied to a factor
> ?"[<- .data.frame"
# Help on the data frame replace operator
```

13.5 Assignment

Storing data into variables is a fundamental operation in any language, and R provides the arrow notation for the assignment operator:

```
> x <- 10
```

The assignment can operate in the opposite direction but it is not so common, and is probably best avoided:

```
> 10 -> x
```

Occasionally you might also see the double headed arrow notation:

```
> x <<- 10
```

It is not generally used, should be avoided, and is included here only for completeness. It is used, for example, to maintain information in a function across different function calls to the same function, or to share some information across a number of functions. In effect it assigns a value to a variable that exists outside the function (in fact, it searches through the list of environments up to the Global Environment to find where the variable is defined, and assigns the value to the first it finds, or creates a new one in the Global Environment if none found). This breaks some of the principles of functional programming—in that user variables might be changed unexpectedly, possibly resulting in erroneous computations.

An alternative to `<-` is `=`, and in most situations they are equivalent. There are subtle differences as in:

```
if (x=0) 1 else 2
```

This gives a syntax error, whereas replacing `=` with `<-` is quite legitimate (though unusual). This helps to avoid a common coding bug where this test is not expected to be an assignment, but in fact a comparison (which should really be using `==`).

13.6 Libraries and Packages

Much of the usefulness of R comes from functionality implemented in various packages. R provides an extensive collection of packages available from CRAN. Packages are installed into your own libraries which are directories containing collections of packages. The *library* function requests R to find a package in some library and load that package. The *require* function can be used in packages and functions to also load a library, but only give a warning and return FALSE if the package can not be found.

13.6.1 Searching for Objects

Objects (data, functions, and methods) are searched for through a search path of packages and datasets. If the object is not found an error is raised. The *search* function will list the current search path. We add elements to the search path with the *attach* and *library* functions and can remove them with the *detach* function:

```
> Species
Error: object "Species" not found
> search()
[1] ".GlobalEnv"           "package:Rcmdr"      "package:car"
[4] "package:tcltk"        "package:methods"   "package:stats"
[7] "package:graphics"     "package:grDevices" "package:utils"
[10] "package:datasets"    "RcmdrEnv"         "Autoloads"
[13] "package:base"
> attach(iris)
> search()
[1] ".GlobalEnv"           "iris"            "package:Rcmdr"
[4] "package:car"           "package:tcltk"    "package:methods"
[7] "package:stats"         "package:graphics" "package:grDevices"
[10] "package:utils"         "package:datasets" "RcmdrEnv"
[13] "Autoloads"            "package:base"
> Species
```

```
[1] setosa      setosa      setosa      setosa      setosa      setosa
[...]
[145] virginica  virginica  virginica  virginica  virginica  virginica
Levels: setosa versicolor virginica
> detach(iris)
> search()
[1] ".GlobalEnv"          "package:Rcmdr"       "package:car"
[4] "package:tcltk"        "package:methods"    "package:stats"
[7] "package:graphics"     "package:grDevices" "package:utils"
[10] "package:datasets"     "RcmdrEnv"         "Autoloads"
[13] "package:base"
```

13.6.2 Package Management

Packages can be installed or updated using the *install.packages* function, which will connect to a CRAN repository on the Internet and download the package. If no repository has already been specified, R will request one to use. You can specify one using the *options* function. Your installed packages will be updated using the *update.packages* function, while packages can be removed using *remove.packages*.

```
> options(repos="http://cran.us.r-project.org/")
> install.packages("ellipse")      # Installs the ellipse package from CRAN.
> install.packages("H:/ellipse_2.0-14.zip", repos=NULL)
> update.packages()
> remove.packages([...])
```

The *available.packages* function will list the packages available from the CRAN archives. See also *installed.packages* and *download.packages*.

You can identify the status of the packages you have installed in your R system with *packageStatus*. You will be asked to select a CRAN mirror from which to retrieve information about the available packages.

```
> packageStatus()
--- Please select a CRAN mirror for use in this session ---
Loading Tcl/Tk interface ... done
Number of installed packages:

          ok upgrade unavailable
/usr/local/lib/R/site-library 12      10      0
/usr/lib/R/site-library        46      11      10
/usr/lib/R/library             24      2       0

Number of available packages (each package/bundle counted only once):

                         installed not installed
http://cran.au.r-project.org/src/contrib           89        494
```

If you then request a *summary* of the *packageStatus* you will get details about the packages, including, for example, those available from each of the local libraries, identifying those that have upgrades available.

```
> summary(packageStatus())

Installed packages:
-----
*** Library /usr/local/lib/R/site-library
$ok
[1] "ISwR"          "ROCR"           "XML"            "ash"            "dprep"
[6] "kernlab"        "leaps"          "modeltools"     "mvpart"        "oz"
[11] "sfsmisc"       "vioplot"

$upgrade
[1] "DAAG"          "arules"         "chplot"        "coin"          "ellipse"
[6] "gbm"           "party"          "pixmap"        "plotrix"      "randomForest"

\$unavailable
NULL

*** Library /usr/lib/R/site-library
$ok
[1] "Design"         "Hmisc"          "MCMCpack"      "MNP"           "MatchIt"
[6] "RMySQL"         "RODBC"          "RQuantLib"     "Rcmdr"        "Rmpi"
[11] "Zelig"          "abind"          "acepack"       "coda"          "date"
[16] "effects"        "fBasics"        "fCalendar"    "fExtremes"    "fMultivar"
[21] "fOptions"       "fPortfolio"     "fSeries"       "gtkDevice"    "its"
[26] "lmtest"          "mapdata"         "mapproj"       "maps"          "misc3d"
[31] "multcomp"       "mvtnorm"        "pscl"          "psy"          "qtl"
[36] "quadprog"       "relimp"          "rgl"           "rpvm"         "rsprng"
[41] "sandwich"       "sm"              "snow"          "tkrplot"     "tseries"
[46] "zoo"

$upgrade
[1] "DBI"            "Matrix"         "XML"
[4] "car"             "gregmisc:gdata" "gregmisc:gmodels"
[7] "gregmisc:gplots" "gregmisc:gregmisc" "gregmisc:gtools"
[10] "lme4"            "strucchange"

\$unavailable
[1] "RGtk"           "Rggobi"         "event"         "gnlm"          "growth"
[6] "ordinal"         "repeated"        "reposTools"   "rmutil"        "stable"

*** Library /usr/lib/R/library
$ok
[1] "KernSmooth"     "VR:MASS"        "VR:class"      "VR:nnet"      "VR:spatial"
[6] "base"            "boot"           "cluster"       "datasets"    "foreign"
[11] "grDevices"       "graphics"        "grid"          "lattice"     "methods"
[16] "mgcv"            "rpart"          "splines"       "stats"       "stats4"
[21] "survival"        "tcltk"          "tools"         "utils"
```

```
$upgrade
[1] "StatDataML" "nlme"

$unavailable
NULL

Available packages:
-----
( each package appears only once)

*** Repository http://cran.au.r-project.org/src/contrib
$installed
[1] "DAAG"          "DBI"           "Design"        "Hmisc"         "ISwR"
[6] "KernSmooth"    "MCMCpack"      "MNP"          "MatchIt"       "Matrix"
[11] "RMySQL"        "ROCR"          "RODBC"        "RQuantLib"     "Rcmdr"
[...]
[81] "sfsmisc"       "sm"            "snow"          "strucchange"   "survival"
[86] "tkrplot"        "tseries"       "vioplot"       "zoo"

$"not installed"
[1] "AMORE"          "AlgDesign"      "AnalyzeFMRI"
[4] "BHH2"           "BMA"           "BRugs"
[7] "BSDA"           "Bhat"          "Biodem"
[...]
[487] "verification" "verify"        "waveslim"
[490] "wavethresh"   "wle"          "xgobi"
[493] "xtable"         "zicounts"
```

13.6.3 Information About a Package

Information about a package can be obtained through the *library* function, using the *help* option. The information includes the basic meta-data about the package (including its name, version, author, and dependencies).

```
> library(help=rpart)
      Information on package "rpart"

Description:

Package:      rpart
Priority:     recommended
Version:      3.1-23
Date:         March 2002 version of rpart, R version 2005-04-15
Author:        Terry M Therneau and Beth Atkinson <atkinson@mayo.edu>.
               R port by Brian Ripley <ripley@stats.ox.ac.uk>.
Maintainer:   Brian Ripley <ripley@stats.ox.ac.uk>
Description:   Recursive partitioning and regression trees
Title:        Recursive Partitioning
```

```

Depends:      R (>= 2.0.0)
Suggests:    survival
License:      use under GPL2, or see file LICENCE
LazyData:     yes
URL:          S-PLUS 6.x original at
              http://www.mayo.edu/hsr/Sfunc.html
Packaged:    Tue Apr 19 11:21:21 2005; ripley
Built:        R 2.1.0; i386-pc-linux-gnu; 2005-04-20 03:10:16; unix

Index:

car.test.frame      Automobile Data from 'Consumer Reports' 1990
cu.summary          Automobile Data from 'Consumer Reports' 1990
kyphosis            Data on Children who have had Corrective
                    Spinal Surgery
[...]

```

The package help displayed above also includes an index of what the package contains. Each item here will generally have further help available through the *help* function:

```

> help(car.test.frame)
car.test.frame      package:rpart           R Documentation
Automobile Data from 'Consumer Reports' 1990

Description:
The 'car.test.frame' data frame has 60 rows and 8 columns, giving
data on makes of cars taken from the April, 1990 issue of
Consumer Reports. This is part of a larger dataset, some columns
of which are given in 'cu.summary'.
[...]

```

13.6.4 Testing Package Availability

If a function you write depends on the functionality of some package, such as *gplots*, use the *stopifnot* function to exit if the package is not available:

```

myfun <- function()
{
  stopifnot(require(gplots))
  [...]
}

```

Also, the `.packages` can be used to list all packages that have been installed on your system:

```
> .packages(all=TRUE)
[1] "acepack"           "ada"                 "amap"
[4] "arules"            "bitops"              "butler"
[..]
[73] "tcltk"             "tools"               "utils"
[76] "mapdata"
```

And a call to the `library` function with no arguments will list all packages installed, and a one line description:

```
> library()
Packages in library '/usr/lib/R/library':
base                  The R Base Package
datasets              The R Datasets Package
graphics              The R Graphics Package
[...]
Packages in library '/usr/local/lib/R/site-library':
acepack                ace() and avas() for selecting regression
                        transformations
ada                    Performs boosting algorithms for a binary
                        response
[...]
XML                   Tools for parsing and generating XML within R
                        and S-Plus.
xtable                Export tables to LaTeX or HTML
```

13.6.5 Packages and Namespaces

Detaching a package does not unload its namespace from R. Unloading a namespace does not de-register its methods. As Gabor Grothendieck points out on r-help on 26 Mar 2007:

```
> search()
[1] ".GlobalEnv"          "package:stats"        "package:graphics"
[4] "package:grDevices"   "package:utils"        "package:datasets"
[7] "package:methods"     "Autoloads"           "package:base"
> loadedNamespaces()
[1] "base"                "graphics"           "grDevices"          "methods"
[5] "stats"               "utils"              "grid"              "gridBase"
[9] "gridExtra"            "#> as.Date(1) # error as there is no numeric method for as.Date
Error in as.Date.default(1) : do not know how to convert '1' to class "Date"
```

```

> library(zoo)
> search()
[1] ".GlobalEnv"      "package:zoo"       "package:stats"
[4] "package:graphics" "package:grDevices" "package:utils"
[7] "package:datasets" "package:methods"   "Autoloads"
[10] "package:base"
> loadedNamespaces()
[1] "base"           "graphics"     "grDevices"    "grid"        "lattice"     "methods"
[7] "stats"          "utils"        "zoo"
> as.Date(1) # zoo defines a numeric method for as.Date
[1] "1970-01-02"

> detach()
> unloadNamespace("zoo")
<environment: namespace:zoo>
> search()
[1] ".GlobalEnv"      "package:stats"       "package:graphics"
[4] "package:grDevices" "package:utils"       "package:datasets"
[7] "package:methods"   "Autoloads"        "package:base"
> loadedNamespaces()
[1] "base"           "graphics"     "grDevices"    "grid"        "lattice"     "methods"
[7] "stats"          "utils"        > # zoo is gone from attached package list and loadedNamespaces
> # but numeric method for as.Date from zoo is still registered
> as.Date(1)
[1] "1970-01-02"

```

13.7 Basic Programming in R

13.7.1 Folders and Files

```

> getwd()                      # Identify the current default working directory
> setwd("h:/work/")            # Change the current default working directory
> file.exists("filename")      # Returns TRUE if the file exists
> unlink("filename")           # Deletes the file or directory
> fname <- file.choose()       # An interactive file chooser.
> choose.dir()                 #
> dir <- tclvalue(tkchooseDirectory()) # GUI which requires library(tcltk).

```

MS/Windows paths use the backward slash to separate components. This is a problem since the backslash is used as a standard mechanism for introducing special characters within strings. Thus, R requires a double back slash or will seamlessly allow the use of the forward slash. A useful utility on a MS/Windows environment, where backslashes are used in paths, and R likes to have forward slashes, is the following (Duncan Golicher, 5 Jan 2006, r-help):

```
setwd.clip <- function()
{
  options(warn=-1)
  setwd(gsub("\\\\", "/", readLines("clipboard")))
  options(warn=0)
  getwd()
}
```

Then, simply select a path into your clipboard (Ctrl-C), then in R call *setwd.clip*!

R packages supply files and we can access them in a installation independent way using *system.file*:

```
> system.file("csv", "audit.csv", package = "rattle")
```

We can view the contents of files in R using *file.show*:

```
> file.show(system.file("csv", "audit.csv", package = "rattle"))
```

13.7.2 Flow Control

```
> if (!file.exists("mydata.RData")) { write(myiris, file="mydata.RData") }
```

13.7.3 Functions

```
myfun <- function(arg1, arg2=TRUE, ...)
{
  [...]
```

The results of a function are returned with the *return* function (or else is the result of the last evaluated statement in the function). The returned result will be printed if the result is not assigned to a variable. To avoid the result being printed, use the *invisible* function to return the result.

Anonymous functions can be used:

```
(function(x, y) x^2 + y^2)(0:5, 1)      #  1  2  5 10 17 2
```

Functions are first class objects. You can write a function to return a function:

```
f <- function(y=10)
{
  g <- function(x) seq(x, x+y)
  return(g)
```

```

}
> h <- f(5)
> h(5)
[1] 5 6 7 8 9 10
> h <- f()
> h(5)
[1] 5 6 7 8 9 10 11 12 13 14 15
> get("y",env=environment(h))
[1] 10

```

And you can write a function to apply some given function:

```

chooseFun<-function(dat=1:10,fun=mean,...)fun(dat,...)
chooseFun()
x<-rnorm(100)
chooseFun(x,median)
chooseFun(x,hist)
chooseFun(x,hist,col='gray')

```

13.7.4 Apply

Suppose you have a data frame from which you wish to extract a subset of rows, and you have a matrix to record the start and finish of the sequences of indicies you wish to extract. Whilst a for loop is obvious, *mapply* works nicely.

```

> x <- rbind(c(2,5), c(7,9), c(15,20))
> x
     [,1] [,2]
[1,]    2    5
[2,]    7    9
[3,]   15   20
> unlist(mapply(seq, x[,1], x[,2]))
[1] 2 3 4 5 7 8 9 15 16 17 18 19 20

```

13.7.5 Methods

Sometimes you may want to know how a function is implemented. R is also an object oriented function and so what method is called depends on the type of the argument. This can be illustrated with the *mean* function. Try to determine its implementation:

```

> mean
function (x, ...)
UseMethod("mean")
<environment: namespace:base>

```

The *UseMethod* part indicates that *mean* is a generic function, possibly with many different implementations. A generic function usually has a default method:

```
> mean.default
function (x, trim = 0, na.rm = FALSE, ...)
{
  if (!is.numeric(x) && !is.complex(x) && !is.logical(x)) {
    warning("argument is not numeric or logical: returning NA")
    return(as.numeric(NA))
  }
  if (na.rm)
    x <- x[!is.na(x)]
  trim <- trim[1]
  n <- length(x)
  [...]
  if (is.integer(x))
    sum(as.numeric(x))/n
  else sum(x)/n
}
<environment: namespace:base>
```

Togaware Watermark For Data Mining Survival

13.7.6 Objects

R provides an object oriented interface, with inheritance and objects with attributes. For an object with attributes you can access the attributes with the *attr* function or the *@* operator:

```
> attr(obj, "age")
> obj@age
```

The *get* function is useful to transform a string into the name of an object:

```
> ds <- 1:5
> ds
> ds
[1] 1 2 3 4 5
> "ds"
[1] "ds"
> get("ds")
[1] 1 2 3 4 5
> get(ds)
Error in get(x, envir, mode, inherits) : invalid first argument
```

13.7.7 System

Running System Commands

You can ask the operating system to perform a command with the *system* command:

```
> system("sleep 10")          # Run OS command sleep with argument 10
> system("sleep 10", wait=FALSE) # Run OS command but don't wait (MS/Windows)
> system("sleep 10 &")        # Run OS command but don't wait (GNU/Linux)
> l <- system("ls", intern=TRUE) # Run OS command and collect output
```

System Parameters

At times you may need or want to know about the machine on which you are running. The R *.Platform* variable will give information about the system, and can be used, for example, to conditionally run code depending on which operating system you are on.

```
> .Platform
$OS.type
[1] "unix"

$file.sep
[1] "/"

\$dynlib.ext
[1] ".so"

$GUI
[1] "X11"

$ endian
[1] "little"

$pkgType
[1] "source"

> if (.Platform$OS.type == "unix") system("ls")
```

There are other variables, but *.Platform* is the recommended variable to use for programming. The variable *version* also lists information about the version of R you are running.

```
> version
platform i486-pc-linux-gnu
```

```

arch      i486
os        linux-gnu
system   i486, linux-gnu
status    beta
major     2
minor     2.0
year      2005
month    09
day       28
svn rev  35702
language R

> if (version$os == "linux-gnu") system("ls")
> if (version$os == "mingw32") system("dir")

```

For a summary of the current R session the *sessionInfo* function is useful:

```

> sessionInfo()
R version 2.4.1 (2006-12-18)
i486-pc-linux-gnu

locale:
LC_CTYPE=en_AU;LC_NUMERIC=C;LC_TIME=en_AU;LC_COLLATE=en_AU;
LC_MONETARY=en_AU;LC_MESSAGES=en_AU;LC_PAPER=en_AU;LC_NAME=C;
LC_ADDRESS=C;LC_TELEPHONE=C;LC_MEASUREMENT=en_AU;LC_IDENTIFICATION=C

attached base packages:
[1] "stats"      "graphics"    "grDevices"   "utils"       "datasets"    "methods"
[7] "base"

other attached packages:
ROCR      gplots      gdata      gtools      ada       rpart
"1.0-1"   "2.3.2"    "2.3.1"    "2.3.0"    "2.0-1"   "3.1-34"
ellipse   rggobi     mice      nnet       MASS      Hmisc
"0.3-2"   "2.1.4-4"  "1.15"    "7.2-31"   "7.2-31"  "3.2-1"
RGtk2    rattle     rcompletion
"2.8.7"   "2.1.123"  "0.0-12"

```

Yet another source of system information is *Sys.info* which includes machine name and type, operating system name and version, and username.

```

> Sys.info()
                     sysname          release
                     "Linux"        "2.6.12-1-686-smp"
                     version        nodename
"##1 SMP Tue Sep 6 15:52:07 UTC 2005"      "athene"
                     machine
login           user
"gjw"           "gjw"

> if (Sys.info()["sysname"] == "Linux") system("ls")

```

Information about various limits of the machine (including things like maximum integer) is obtained from the `.Machine` variable.

```
> .Machine
\$double.eps
[1] 2.220446e-16

\$double.neg.eps
[1] 1.110223e-16

[...]

\$integer.max
[1] 2147483647

\$sizeof.long
[1] 4

\$sizeof.longlong
[1] 8

\$sizeof.longdouble
[1] 12

\$sizeof.pointer
[1] 4
```

A call to `capabilities` will list optional features that have been compiled into your version of R:

```
> capabilities()
  jpeg      png     tcltk      X11 http/ftp   sockets   libxml     fifo
  TRUE      TRUE     TRUE      TRUE    TRUE     TRUE     TRUE     TRUE
  cldit    iconv      NLS
  TRUE      TRUE     TRUE
```

The `options` function in R allows numerous characteristics of the running R session to be tuned. The `options` function without arguments returns a list of options and their values. Use `getOption` to get a specific option value, and `options` itself to set values. Options that can be set include the prompt string (`prompt`) and continuation prompt (`continue`), the terminal width (`width`), number of digits to show when printing (`digits`, and many more. In this example we list some of the options then change `width`, storing its old value, evaluate some other functions, then restore the original value:

```
> options()
$OutDec
[1] ". "
```

```
$X11colortype
[1] "true"

[...]

$verbose
[1] FALSE

$warn
[1] 0

$warnings.length
[1] 1000

$width
[1] 80

>getOption("width")
[1] 80
>ow <- options(width=120)
[...]
>options(ow)
```

Other useful functions include:

```
> R.home()           # Location of R installation: /usr/lib/R
> Sys.sleep(5)       # Sleep for 5 seconds.
> proc.time()        # shows how much time is currently consumed
> Rprof("Rprof.out") # Profile the execution of R expressions
> system.time([...]) # Execute a command and report the time taken
> on.exit([...])     # Execute commands on exit or interruption
> username <- as.vector(Sys.info()["login"])
```

13.7.8 Misc

```
> ?plot                  # The ? is a shortcut for help().
> identical(x,y)         # Test if data is identical.
> sample(2:12, 100, replace=TRUE) # Random sample with replacement.

> x <- 1:20
> w <- 1 + sqrt(x)
> q()
```

13.7.9 Internet

You can check whether you have access to an Internet connection using the *nsl* function, checking for a specific hostname:

```
> ns1("www.r-project.org")
[1] "137.208.57.37"
> if(is.null(suppressWarnings(ns1("www.r-project.org")))) print("Connected?")
```

13.8 Memory Management

13.8.1 Memory Usage

Large datasets often present challenges for R on memory limited machines. While you may be able to load a large dataset, processing it and modelling may lead to an error indicating the memory could not be allocated.

To maximise R's capabilities on large datasets, be sure to run a 64bit operating system on a 64 bit platform (e.g., [Debian GNU/Linux](#)) on 64 bit hardware (e.g., [AMD64](#)) with plenty of RAM (e.g., 16GB). Such capable machines are quite affordable.

Selecting and subsetting required datasets off a database (e.g., through the *RODBC* package) or through other means (e.g., using Python) will generally be faster.

On MS/Windows you may need to set the memory size using the command-line flag `--max-mem-size`. The amount of memory currently in use and allocated to the R process, is given by the *memory.size* function.

The example below indicates that some 470MB is in use, altogether about 1GB has been allocated.

```
> memory.size()                      # Current memory in use: 470MB
[1] 477706008
> memory.size(TRUE)                  # Current memory allocated: 1GB
[1] 1050681344
```

The memory limit currently in force in R is reported by the *memory.limit* function which can also be used to set the limit.

```
> memory.limit()                    # Current memory limit: 1GB
[1] 1073741824
> memory.limit(2073741824)        # New memory limit: 2GB
NULL
> memory.limit()
[1] 2684354560
```

A suggested process is to work with a subset of all the data loaded in memory, using a dataset small enough to make this viable. Explore the data, explore for the choice of models, and prototype the final analysis using this smaller dataset. For the final full analyses one may need to allow R to run overnight with enough RAM.

A data frame of 150,000 rows and some 55 columns will be about 500MB of RAM.

Also, note the difference between data frames and arrays/matrices. For example, rbind'ing data frames is much more expensive than rbind'ing arrays/matrices. However, an array/matrix must have all data of the same data type in each column while data frames can have different data types in different columns. A number of functions are written to handle either data frames or matrices (e.g., *rpart*) and it is best, if possible, to use a matrix in these cases. The coercion back to a data frame can always be done afterwards.

Note that to convert a data frame to a matrix you can use *as.matrix*:

```
> m <- as.matrix(dframe)
```

However, if there are any character columns, all the data is converted to character.

To obtain an estimate of the amount of memory being used by an object in R use the *object.size* function:

```
> object.size(ds)                      # Object ds is using 181MB
[1] 181694428
```

The following function can be used to explore memory requirements:

```
sizes <- function(rows, cols=1)
{
  testListLength <- 1000
  cellSize <- object.size(seq(0.5, testListLength/2, 0.5))/testListLength
  cells <- rows * cols
  required <- cells * cellSize
  if (required > 1e12)
    result <- sprintf("%dTB", required %% 1e12)
  else if (required > 1e9)
    result <- sprintf("%dGB", required %% 1e9)
  else if (required > 1e6)
    result <- sprintf("%dMB", required %% 1e6)
  else if (required > 1e3)
    result <- sprintf("%dKB", required %% 1e3)
```

```

    else
      result <- sprintf("%dBytes", required)
    return(result)
}

```

For example, on a 32bit machine, a 1 million row dataset with 400 columns might require about 3GB of memory!

13.8.2 Garbage Collection

When doing timings of commands it is important to know that garbage collection plays a role. R adjusts its garbage collection triggers according to your usage. When you first start using large objects the trigger levels will grow and generally things will speed up.

You can use `gcinfo` to start seeing the adjustments in action:

```
> gcinfo(TRUE)
[1] FALSE # The setting was previously FALSE
```

For the `system.time` function use the `gcFirst`.

The `gc` function will cause a garbage collection to take place, and lists useful information about memory usage (the primary purpose for calling the `gc` function). Ncells is the number of so called cons cells used (each cell is 28 or 56 bytes on 32 or 64 bit systems, and is used for storing fixed sized objects), and this is converted in the function's to Mb for us. Vcells is the number of vector cells used (each cell is 8 bytes, and is used for storing variable sized objects). The final two columns show the maximum amount of memory that has been used since the last call to `gc(reset=TRUE)`.

```
> gc()
      used (Mb) gc trigger (Mb) max used (Mb)
Ncells 177949  4.8     407500 10.9   350000  9.4
Vcells 72431  0.6     786432  6.0   332253  2.6
> survey <- read.csv("survey.csv")
> gc()
      used (Mb) gc trigger (Mb) max used (Mb)
Ncells 212685  5.7     741108 19.8   514436 13.8
Vcells 366127  2.8     1398372 10.7  1387692 10.6
> rm(survey)
> gc()
      used (Mb) gc trigger (Mb) max used (Mb)
Ncells 179940  4.9     741108 19.8   514436 13.8
Vcells 72773  0.6     1118697  8.6   1387692 10.6
```

Here, after reading the datafile survey.csv (which is 4100478 bytes, or 4MB, in size as a text file), XXXX

13.8.3 Errors

When an error occurs the message may not always be as insightful as one would like. The *traceback* function can be used to review the sequence of function calls that lead to the error:

```
> fun1 <- function(v){fun2(v)}
> fun2 <- function(v){x.no.name + y.other.name}
> fun1(10)
Error in fun2(v) : object "x.no.name" not found
> traceback()
2: fun2(v)
1: fun1(10)
```

13.9 Frivolous

R also provides many packages to do other interesting things! Here's a brief collection that are recorded here for the curious!

13.9.1 Sudoku

Solve a sudoku puzzle with the *sudoku* package! Into a file place the following representation of a sudoku puzzle:

```
2--f-c-8----ab7-
e---a-6-f-03--d-
--305b-f--2496--
-7-----b---5-f1
-----e9-d----6
-0fec-89----1-b
6----532--7--9-8
-9--d--0----7a4-
-6dc---3---7--a-
7-0--e--46b----9
1-e-----5d-9fc3-
4----3-ae-----
93-1---c-----e-
--72e4--8-6dbf--
-e--8a-3-1-5---c
-a45----7-9-8--0
```

Then:

```
> install.packages("sudoku")
> library(sudoku)
> library(help=sudoku)
> sud <- readSudoku("sudoku.txt")
> solveSudoku("sudoku.txt", map=c(0:9,letters))
2 [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12] [,13]
[1,] 2 4 6 f 0 c e 8 d 9 5 1
a [2,] e b 5 9 a 1 6 7 f 8 0 3
c [3,] c 1 3 0 5 b d f a 7 2 4
9 [4,] 8 7 a d 3 9 2 4 b e c 6
5 [5,] 5 8 1 7 4 f a e 9 b d 0
3 [6,] a 0 f e c 7 8 9 6 4 3 2
d [7,] 6 d c 4 b 5 3 2 1 a 7 f
e [8,] 3 9 2 b d 6 1 0 c 5 e 8
7 [9,] b 6 d c 9 8 4 1 3 2 f 7
0 [10,] 7 5 0 3 f e c d 4 6 b a
1 [11,] 1 2 e a 7 0 b 6 5 d 8 9
f [12,] 4 f 9 8 2 3 5 a e 0 1 c
6 [13,] 9 3 8 1 6 d 0 c 2 f a b
4 [14,] 0 c 7 2 e 4 9 5 8 3 6 d
b [15,] f e b 6 8 a 7 3 0 1 4 5
2 [16,] d a 4 5 1 2 f b 7 c 9 e
8
     [,14] [,15] [,16]
[1,] b 7 3
[2,] 4 d 2
[3,] 6 8 e
[4,] 0 f 1
[5,] 2 c 6
[6,] 1 5 b
[7,] 9 0 8
[8,] a 4 f
[9,] e a 5
[10,] 8 2 9
[11,] c 3 4
[12,] 7 b d
[13,] 5 e 7
[14,] f 1 a
[15,] d 9 c
```

```
[16,]    3    6    0
```

Another one. The original pattern is:

```
+-----+-----+
| 1 . . | . . 7 | . 9 . |
| . 3 . | . 2 . | . . 8 |
| . . 9 | 6 . . | 5 . . |
+-----+-----+-----+
| . . 5 | 3 . . | 9 . . |
| . 1 . | . 8 . | . . 2 |
| 6 . . | . . 4 | . . . |
+-----+-----+-----+
| 3 . . | . . . | . 1 . |
| . 4 . | . . . | . . 7 |
| . . 7 | . . . | 3 . . |
+-----+-----+-----+
```

As a text file for R:

```
1----7-9-
-3--2---8
--96--5--
--53--9--
-1--8---2
6----4---
3-----1-
-4-----7
--7---3--
```

And the solution!

```
> z <- readSudoku("sudoku2.txt")
> z
 [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
 [1,]    1    0    0    0    0    7    0    9    0
 [2,]    0    3    0    0    2    0    0    0    8
 [3,]    0    0    9    6    0    0    5    0    0
 [4,]    0    0    5    3    0    0    9    0    0
 [5,]    0    1    0    0    8    0    0    0    2
 [6,]    6    0    0    0    0    4    0    0    0
 [7,]    3    0    0    0    0    0    0    1    0
 [8,]    0    4    0    0    0    0    0    0    7
 [9,]    0    0    7    0    0    0    3    0    0
> solveSudoku(z)
 [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
 [1,]    1    6    2    8    5    7    4    9    3
 [2,]    5    3    4    1    2    9    6    7    8
 [3,]    7    8    9    6    4    3    5    2    1
 [4,]    4    7    5    3    1    2    9    8    6
 [5,]    9    1    3    5    8    6    7    4    2
 [6,]    6    2    8    7    9    4    1    3    5
 [7,]    3    5    6    4    7    8    2    1    9
 [8,]    2    4    1    9    3    5    8    6    7
```

 [9,] 8 9 7 2 6 1 3 5 4

13.10 Further Resources

There are extensive resources available to assist in learning and using R.

13.10.1 Using R

13.10.2 Specific Purposes

While we concentrate on data mining in this book, R can serve many purposes and there are specialised packages and web resources available.

Togaware Watermark For Data Mining Survival Survey Analysis

The *survey* package provides facilities in R for analysing data from complex surveys. As reported on its web page, it supports: means, totals, ratios, quantiles, contingency tables, regression models, for the whole sample and for domains; variances by Taylor linearization or by replicate weights (BRR, jackknife, bootstrap, or user-supplied); multi-stage sampling with or without replacement; post-stratification, ranking, GREG estimation; graphics. Further information from <http://faculty.washington.edu/tlumley/survey/>.

Togaware Watermark For Data Mining Survival

Chapter 14

Data

R supports multiple paradigms, including functional and object-oriented programming. Every object in R has a class. Functions in R then operate on objects of particular classes.

14.1 Data Types

The basic types of objects in R include `logical`, `integer`, `double`, `complex`, and `character`. We obtain the type of an object using the `typeof` function:

```
> typeof(TRUE)
[1] "logical"
> typeof(5+4)
[1] "double"
> typeof(letters)
[1] "character"
> typeof(typeof)
[1] "closure"
```

The last example here illustrates the fact that functions in R are also first class objects—that is, they are just like any other data type. Functions have a type called a `closure`, which is a type that wraps up the function’s definition and the environment in which the function was created. The other special types include `NULL`, `symbol`, `environment`, `promise`, `language`, `special`, `builtin`, `expression`, and `list`.

The *mode* of an object is another indication of the type of the object. For example, objects of type **integer** and **double** are of mode **numeric**.

```
> mode(x)
> storage.mode(x)
> class(x)                                # The object class for method dispatch
```

Normally we build up an object to have some structure, like a vector, an array, a matrix, or a data frame. The basic building block of a data structure in R is a vector of objects of some type. Indeed, an object of type **double** will in fact be a vector of numbers.

R will keep track of objects you have created:

```
> ls()                                     # List all objects you have created.
> rm(list=ls(all=TRUE)) # Remove all of your objects.
```

In the following sections we introduce each of the main types of objects and illustrate typical operations for manipulating the objects.

Togaware Watermark For Data Mining Survival

14.1.1 Numbers

Numbers in R can be **integer**, **double**, or **complex**. The actual type is generally determined by R based on context and defaults. Numbers do tend to be **double** unless they are in some way restricted to being integers, or you wish to store **complex numbers**.

```
> typeof(1)          # "double"
> typeof(1.3)        # "double"
> typeof(1:2)         # "integer" - this is a vector of two integers 1 and 2
> typeof((1:2)[1])   # "integer" - this is the first element of the vector
> typeof(2+5i)        # "complex" - complex numbers have real and imaginary parts
```

The basic numeric operators include:

```
\verb|+| Addition           \verb|-| Subtraction     \verb|*| Multiplication
\verb|/| Division            \verb|^| Exponentiation  \verb|:| Sequence
\verb|%%| Remainder          \verb|/%/%| Integer division
```

See the documentation for Arithmetic for details:

```
> ?Arithmetic
```

Typical numeric functions include:

```
> 1:5          # 1 2 3 4 5
> round(1234.56)    # 1235
> trunc(1234.56)   # 1234
```

14.1.2 Strings

R provides the usual string type and string manipulation functions:

```
> chartr("a-m", "m-z", "abcdef")      # "mnopqr"
> grep
> nchar
> sub
> substr
> tolower("ABCdef")                  # "abcdef"
> toupper("ABCdef")                  # "ABCDEF"
> pmatch("png", c("jpeg", "png", "gif")) # 2
```

R provides some useful predefined variables, including:

```
> letters      # a b c d [...] z
> LETTERS      # A B C D [...] Z
```

Building Strings

A typical usage of *paste* is to build a label vector to be used in a plot. The labels may want to include both the variable values being plotted and perhaps the percentage of observations having that value.

```
> labels <- c("A", "B", "C", "D")
> per     <- c(25, 10, 15, 45)
> paste(labels, rep(" ", 4), per, rep("%", 4), sep="")
[1] "A (25%)" "B (10%)" "C (15%)" "D (45%)"
> sprintf("%s (%d%)", labels, per)
[1] "A (25%)" "B (10%)" "C (15%)" "D (45%)"
```

Splitting Strings

strsplit is used to split a string into substrings:

```
> unlist(strsplit("abc def ghi jkl", " "))
[1] "abc" "def" "ghi" "jkl"
```

```
> unlist(strsplit("abc,def:ghi.jkl", "\\.|,|:"))
[1] "abc" "def" "ghi" "jkl"
```

The split pattern is a regular expression (hence the `\.` is required to quote the full stop). For details on regular expressions see `?regexp`

Substitution

Remove all decimal points from a string representing a real number using `sub` with either limiting the replacement to digits, or else substituting any characters:

```
> sub("\\.[[:digit:]]*\\.", "", "12345.67")
[1] "12345"
> sub("[.]*", "", "12345.67")
[1] "12345"
```

In the second example the `"."` does not need to be escaped since it appears in a character class.

Trim Whitespace

A simple function to trim whitespace from the beginning and end of a string, which you could then use to more simply split a string into a vector, uses `gsub`:

```
> trim.ws <- function(text)
+   gsub("^[:blank:]*", "", gsub("[:blank:]*$", "", text))
> s <- " a b      c "
> strsplit(trim.ws(s), " +")
[1] "a" "b" "c"
```

Evaluating Strings

A string can be evaluated as R Code.

```
> s <- "wine$Alcohol<14"
> eval(parse(text=s))
```

14.1.3 Logical

Objects of type `logical` take on the values `TRUE` or `FALSE`. The basic logical operators include:

<code>!</code>	Not	<code><</code>	Less than	<code><=</code>	Less than or equal to
<code>==</code>	Equal to	<code>></code>	Greater than	<code>>=</code>	Greater than or equal to
<code>&</code>	And	<code> </code>	Or	<code>&&</code>	Non vectorised versions

```

> 5 == 4          # FALSE
> 5 != 4          # TRUE
> ! 1 - 1         # TRUE since 1-1 is 0 and is coerced to FALSE
> TRUE * 2        # 2 since TRUE is coerced to 1.

```

The single logical connectives `&` and `|` operates on vectors, whilst the double connective returns a single result, and does a minimal amount of comparison to get the result.

```

> c(TRUE, TRUE, FALSE) & c(TRUE, FALSE, FALSE)
[1] TRUE FALSE FALSE
> c(TRUE, TRUE, FALSE) && c(TRUE, FALSE, FALSE)
[1] TRUE

```

The double form is usually what you want in an *if* statement.

14.1.4 Dates and Times

To calculate the differences between times use *difftime*.

When importing data from a CSV file, for example, dates are simply read as factors. These can easily be converted to date objects using *as.Date*:

```

> ds <- read.csv("authors.csv")
> ds$Notified
[1] 2005/06/05 2005/06/05
> as.Date(ds$Notified, format="%Y/%m/%d")
[1] NA           "2005-06-05" "2005-06-05"

```

The default format is `"%Y-%m-%d"`. See the help for *strftime* for an explanation of the format. Any extra text found in the string after the text has been consumed by the format string will simply be ignored. But if the format is not found at the beginning of the string then a NA is returned.

```
> ds <- c("2005-05-22 12:35:00", "2005-05-23 abc", "abc 2005-05-24")
> ds
[1] "2005-05-22 12:35:00" "2005-05-23 abc"      "abc 2005-05-24"
> class(ds)
[1] "character"
> ds <- as.Date(ds)
> ds
[1] "2005-05-22" "2005-05-23" NA
> class(ds)
[1] "Date"
```

To compare date values use *as.Date*:

```
> ds > as.Date("2005-05-22")
[1] FALSE  TRUE   NA
```

To view the methods associated with the Date class:

```
> methods(class = "Date")
[1] as.character.Date  as.data.frame.Date as.POSIXct.Date    Axis.Date*
[5] c.Date            cut.Date          -.Date           [<-.Date
[9] [.Date            [[.Date          +.Date           diff.Date
[13] format.Date     hist.Date*      is.numeric.Date  julian.Date
[17] Math.Date        mean.Date       months.Date    Ops.Date
[21] plot.Date*      print.Date     quarters.Date   rep.Date
[25] round.Date      seq.Date       summary.Date   Summary.Date
[29] trunc.Date      weekdays.Date

Non-visible functions are asterisked
```

To aggregate by month, some alternatives:

```
> library(chron)
> dts=seq.dates("1/1/01","12/31/03")
> rnum=rnorm(1:length(dts))
> df=data.frame(date=dts,obs=rnum)
> aggregate(df[,2],list(year=years(df[,1]),month=months(df[,1])),sum)
> library(zoo)
> aggregate(zoo(rnum, dts), as.yearmon, sum)
> aggregate(rnum, list(dts = as.yearmon(dts)), sum)
```

Extract the year from a vector of dates:

```
> dates <- c("26 Jan 1974", "April 3, 2002", "23 June, 1999", "2007")
> gsub("[0-9]{1}[0-9]{2} [0-9]{2}[0-9]{2}", "\\\1", dates)
[1] "1974" "2002" "1999" "2007"

> as.POSIXlt('2005-7-1')
[1] "2005-07-01"
> unlist(as.POSIXlt('2005-7-1'))
sec  min  hour  mday  mon  year  wday  yday  isdst
  0    0     0     1     6   105      5    181      0
```

14.1.5 Space

14.2 Data Structures

14.2.1 Vectors

The most basic data structure is a simple vector, a list-like data structure that stores values which are all of the same data type or class. You can either directly create a vector using the R function *c* (for combine), or else have R create a random list of numbers for you, using, for example *runif* (which will generate a sequence of random numbers uniformly distributed between the supplied limits).

```
> v <- c(1, 2, 3, 4, 5)
> v
[1] 1 2 3 4 5
> class(v)
[1] "numeric"
> v <- runif(20, 0, 100)
> v
[1] 69.717291 98.491863 98.541503 72.558488 85.607629 35.441444 59.622427
[8] 40.191194 8.311273 24.215177 77.378846 55.563735 71.554547 97.522348
[15] 2.186403 52.528335 69.281037 44.634309 2.063750 47.125579
```

The *vector* function will create a vector of a specific mode (logical, by default):

```
> vector(length=10)
[1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
> vector(mode="numeric", length=10)
[1] 0 0 0 0 0 0 0 0 0 0
```

Various sequences of numbers can be generated to produce a vector using the *seq* function:

```
> seq(10)                      # [1] 1 2 3 4 5 6 7 8 9 10
> seq(1, 10)                    # [1] 1 2 3 4 5 6 7 8 9 10
> seq(length=10)                # [1] 1 2 3 4 5 6 7 8 9 10
> seq(2, 10, 2)                 # [1] 2 4 6 8 10
> seq(10, 2, -2)                # [1] 10 8 6 4 2
> seq(length = 0)                # numeric(0)
> seq(0)                        # [1] 1 0
> seq(0, 1, by=.1)              # [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9
> seq(0, 1, length=11)           # [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

R will operate on vectors whenever they are given as arguments.

```
> c(2, 4, 6, 8, 10)/2          # [1] 1 2 3 4 5
> c(2, 4, 6, 8, 10)/c(1, 2, 3, 4, 5)    # [1] 2 2 2 2 2
```

```
> log(c(0.1, 1, 10, 100), 10)           # [1] -1 0 1 2
```

In vector operations, short vectors are recycled when additional values are required, but the longer vector's length must be a multiple of the shorter vector's length.

```
> c(1, 2, 3, 4) + c(1, 2)           # [1] 2 4 4 6
> c(1, 2, 3, 4, 5) + c(1, 2)
[1] 2 4 4 6 6
Warning message:
longer object length
  is not a multiple of shorter object length in:
  c(1, 2, 3, 4, 5) + c(1, 2)
```

14.2.2 Arrays

An *array* is a vector that in addition has attributes associated with it. Attributes are used in R to record additional information about an object. Consider the simple example of a sequence of numbers (which is a simple vector by default):

```
> seqv <- 1:20
> is.vector(seqv)
[1] TRUE
> is.array(seqv)
[1] FALSE

> seqa <- as.array(1:20)
> is.array(seqa)
[1] TRUE
> is.vector(seqa)
[1] FALSE

> attributes(seqv)
NULL
> attributes(seqa)
$dim
[1] 20

> seqv@dim
Error: cannot get a slot ("dim") from an object of type "integer"

> seqa@dim
[1] 20
```

14.2.3 Lists

14.2.4 Sets

```
> a <- 1:9
> b <- c(4,5)
> setdiff(a, b)           # [1] 1 2 3 6 7 8 9
> union(a, b)            # [1] 1 2 3 4 5 6 7 8 9
> intersect(a, b)         # [1] 4 5
> setequal(a, b)          # [1] FALSE
```

14.2.5 Matrices

A dataset is usually more complex than a simple vector. Indeed, often we have several vectors making up the dataset, and refer to this as a matrix. A matrix is a data structure containing items all of the same data type. We construct a matrix with the *matrix* and *c* functions. Rows and columns of a matrix can have names, and the functions *colnames* and *rownames* will list the current names. However, you can also assign a new list of names to these functions!

```
> ds <- matrix(c(52, 37, 59, 42, 36, 46, 38, 21, 18, 32, 10, 67),
               nrow=3, byrow=T)
> colnames(ds) <- c("Low", "Medium", "High", "VHigh")
> rownames(ds) <- c("Married", "Prev.Married", "Single")
> ds
      Low Medium High VHigh
Married     52     37    59    42
Prev.Married 36     46    38    21
Single       18     32    10    67
```

Of course, manually creating datasets in this way is only useful for small data collections. A slightly easier approach is to manually modify and add to the dataset using a simple spreadsheet-like interface through the *edit* function or through the *fix* function which will also assign the results of the edit back to the variable being edited. Note that normally the *edit* function returns , and thus prints to the screen if it is not assigned, the datasets. To avoid the dataset being printed to the screen, when you do not assign *edit* to a variable because all you wanted to do was browse the dataset, use the *invisible* function.

```
> ds <- edit(ds)
> fix(ds)
> invisible(edit(ds))
```

The *cbind* function combines each of its arguments, column-wise (the *c* in the name is for *column*), into a single data structure:

```
> age <- c(35, 23, 56, 18)
> gender <- c("m", "m", "f", "f")
> people <- cbind(age, gender)
> people
      age   gender
[1,] "35" "m"
[2,] "23" "m"
[3,] "56" "f"
[4,] "18" "f"
```

Because the resulting matrix must have elements all of the same data type, we see that the variable `age` has been transformed into the character data type (since `gender` could not be so convincingly converted to numeric).

The *rbind* function similarly combines its argument, but in a row-wise manner. The result will be the same as if we transpose the matrix with the *t* function:

```
> t(people)
      [,1] [,2] [,3] [,4]
age     "35" "23" "56" "18"
gender "m"   "m"   "f"   "f"
> people <- rbind(age, gender)
> people
      [,1] [,2] [,3] [,4]
age     "35" "23" "56" "18"
gender "m"   "m"   "f"   "f"
```

14.2.6 Data Frames

A data frame is essentially a list of named vectors, where, unlike a matrix, the different vectors (or columns) need not all be of the same data type. A data frame is analogous to a database table, in that each column has a single data type, but different columns can have different data types. This is distinct from a matrix in which all elements must be of the same data type.

```
> age <- c(35, 23, 56, 18)
> gender <- c("m", "m", "f", "f")
> people <- data.frame(Age=age, Gender=gender)
> people
  Age Gender
1  35      m
```

```

2 23      m
3 56      f
4 18      f

```

The columns of the data frame have names, and the names can be assigned as in the above example. The names can also be changed at any time by assignment to the output of the function call to *colnames*:

```

> colnames(people)
[1] "Age"      "Gender"
> colnames(people)[2] <- "Sex"
> colnames(people)
[1] "Age" "Sex"
> people
  Age Sex
1 35   m
2 23   m
3 56   f
4 18   f

```

If we have the datasets we wish to combine as a single list of datasets, we can use the *do.call* function to apply *rbind* to that list so that each element of the list becomes one argument to the *rbind* function:

```

j <- list()                      # Generate a list of data frames
for (i in letters[1:26])
{
  j[[i]] <- data.frame(rep(i,25),matrix(rnorm(250),nrow=25))
}
j[[1]]
allj <- do.call("rbind", j)       # Combine list of data frames into one.

```

You can reshape data in a data frame using *unstack*:

```

> ds <- data.frame(type=c('x', 'y', 'x', 'x', 'x', 'y', 'y', 'x', 'y', 'y'),
                     value=c(10, 5, 2, 6, 4, 8, 3, 6, 8))
> ds
  type value
1     x    10
2     y     5
3     x     2
4     x     6
5     x     4
6     y     8
7     y     3
8     x     6
9     y     6
10    y     8
> unstack(ds, value ~ type)
  x  y
1 10  5
2  2  8

```

```
3   6  3
4   4  6
5   6  8
```

To even assign the values to variables of the same names as the types you could use *attach*:

```
> attach(unstack(ds, value ~ type))
> x
[1] 10  2  6  4  6
> y
[1] 5  8  3  6  8
```

Accessing Columns

When accessing a column name of a data frame with the \$ notation you can directly use the column name. Using the [notation you will need to quote the name.

```
> people[, "Age"]
> people$Age
> people[["Age"]]
> subset(people, select=Age)
```

This also illustrates a use of *subset*.

Removing Columns

To retain only those columns that begin with INPUT_:

```
> new.audit <- audit[, grep("^INPUT_", names(audit))]
```

14.2.7 General Manipulation

Factors

```
> ds <- data.frame(age=c(34, 56, 23, 72, 48),
                     risk=c("high", "low", "high", "low", "high"))
> ds
  age risk
1  34  high
2  56  low
3  23  high
```

```
4 72 low
5 48 high
> levels(ds$risk)
[1] "high" "low"
```

By default levels within a factor are not ordered:

```
> ds$age[1] < ds$age[2]
[1] TRUE
> ds$risk[1] < ds$risk[2]
[1] NA
Warning message:
< not meaningful for factors in: Ops.factor(ds$risk[1], ds$risk[2])
```

We can order the levels using the *ordered* function:

```
> ds$risk <- ordered(ds$risk)
> levels(ds$risk)
[1] "high" "low"
> ds$risk[1] < ds$risk[2]
[1] TRUE
```

Togaware Watermark For Data Mining Survival

Saying that *high* is less than *low* is probably not what we wanted. The ordering used is the same as what *levels* returns.

You can change the names of the levels by assigning to the *levels* call:

```
> levels(ds$risk) <- c("upper", "lower")
> ds
  age   risk
1 34 upper
2 56 lower
3 23 upper
4 72 lower
5 48 upper
```

Elements

```
> letters          # a b c [...] z
> letters[10]       # "j"
> letters[10:15]    # "j" "k" "l" "m" "n" "o"
> letters[c(1, 2, 4, 8, 16)] # "a" "b" "d" "h" "p"
> letters[-(10:26)] # "a" "b" "c" "d" "e" "f" "g" "h" "i"
```

An operator (or function) can be applied to a vector to return a vector. This is particularly useful for boolean operators, returning a vector of boolean values which can then be used to select specific elements of a vector:

```
> letters > "j"                                # FALSE FALSE FALSE [...] TRUE
> letters[letters > "j"]
> letters[letters > "w" | letters < "e"]      # "k" "l" "m" "n" [...] "y" "z"
                                                # "a" "b" "c" "d" "x" "y" "z"
```

Here's a useful trick to ensure we don't divide by zero, which would otherwise give an infinite answer (*Inf*):

```
> x <- c(0.28, 0.55, 0, 2)
> y <- c(0.53, 1.34, 1.2, 2.07)
> sum(((x-y)^2/x))
[1] Inf
> sum(((x-y)^2/x)[x!=0])                      # Exclude the zeros
[1] 1.360392
```

We could also generate random subsets of our data.

```
> subdataset <- dataset[sample(seq(1, nrow(dataset)), 1000),]
```

We can select elements meeting set inclusion conditions. Here we first select a subset of rows from a data frame having particular colours.

```
> ds[ds$colour %in% c("green", "blue"),]
> ds[ds$colour %in% names(which(table(ds$colour) > 11)),]
```

Rows and Columns

A number of operators are available to extract data from matrices. The single open square bracket [is used to one or more elements from a matrix, while the double open square bracket returns just the specific element specified, requiring all relevant subscripts to be supplied.

```
> ds[1:20,]                                     # Rows 1 to 20.
> ds[,5:10]                                    # Columns 5 to 10.
> ds[,c(3,5,8,9)]                             # Columns 3, 5, 8, and 9.
> lst[[1]]                                      # First element of list lst.
```

The [operator can select multiple elements from an object whilst [[and \$ select just a single element.

Finding Index of Elements

To obtain a list of row indices for a matrix or data frame for those rows meeting some criteria we can use the *which* function:

```
> which(iris$Sepal.Length == 6.2)
[1] 69 98 127 149
```

Conditions can be combined, in which case it becomes useful to use the *with* function, which specifies a dataset with which to evaluate the expression:

```
> with(iris, which(Sepal.Length < 6.2 & Sepal.Width > 4))
[1] 16 33 34
```

Similarly for a matrix:

```
> A <- matrix(c(1:19, 10), 4, 5)
> A
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    5    9   13   17
[2,]    2    6   10   14   18
[3,]    3    7   11   15   19
[4,]    4    8   12   16   10
> which(A == 11, arr.ind = TRUE)
      row col
[1,] 3   3
> which(A == 14, arr.ind = TRUE)
      row col
[1,] 2   4
> which(A == 10, arr.ind = TRUE)
      row col
[1,] 2   3
[2,] 4   5
```

Partitions

We us the *split* function to partition the data into the three groups defined by Type.

Head and Tail

R provides two functions for quickly looking at some subset of the data (whether this be vectors, matrices, data frames, or even functions). These mimic traditional Unix and GNU commands *head* and *tail*. The *head* function returns the top few (6 by default) rows of a data frame.

```
> head(iris)
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1       3.5        1.4       0.2  setosa
2          4.9       3.0        1.4       0.2  setosa
3          4.7       3.2        1.3       0.2  setosa
4          4.6       3.1        1.5       0.2  setosa
5          5.0       3.6        1.4       0.2  setosa
6          5.4       3.9        1.7       0.4  setosa
```

Similarly, *tail* returns the bottom few (6 by default) rows of a data frame:

```
> tail(iris)
   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
145       6.7        3.3       5.7        2.5 virginica
146       6.7        3.0       5.2        2.3 virginica
147       6.3        2.5       5.0        1.9 virginica
148       6.5        3.0       5.2        2.0 virginica
149       6.2        3.4       5.4        2.3 virginica
150       5.9        3.0       5.1        1.8 virginica
```

Reverse a List

Use *rev* to reverse an object:

```
> rev(letters) # "z" "y" "x" ... "c" "b" "a"
```

Sorting

Log-aware Watermark For Data Mining Survival

To sort a matrix on two columns us the *order* function:

```
x <- rep(rbinom(5, size=4, prob=.5), 6)
y <- rnorm(30)
df <- data.frame(x=x, y=y)
df <- df[order(df$x, df$y),]
```

To reverse sort on the second sort column:

```
df <- df[order(df$x, -df$y),]
```

For dealing with missing values in a sort, see <http://www.ats.ucla.edu/STAT/r/faq/sort.htm>.

Unique Values

Use the *unique* function.

```
> x <- c(1, 1, 1, 2, 2, 2, 3, 3, 3, 3)
> unique(x)
[1] 1 2 3
```

14.3 Loading Data

Data can be sourced from many diverse repositories. Data is often stored today in relational databases like Oracle, MySQL, and SQLite, and accessed by the structured query language (SQL). It could be stored in systems like SAS and many other alternative proprietary systems. Many such systems provide access to the data in a number of forms and have particularly well tuned and efficient procedures for accessing the data. Such access is fine for many applications, but the user tends to be limited to the functionality provided by the particular application environment.

R provides a wealth of tools for importing data. In this section we overview the functions used to access common data formats. For a more complete exposition see [R D \(2005\)](#).

Data warehouses

~~Relational Database For Data Mining Survival~~

OLTP systems such as billing systems

Special databases, such as marketing and sales

Excel, Access

Flat files

External data sources, e.g., demographic data, market survey data

Describe output variable and input variables.

```
> library(help="foreign")
```

14.3.1 Interactive Responses

```
> grids <- as.integer(readline("Please enter number of grids: "))
Please enter number of grids: 30
> grids
[1] 30
```

14.3.2 Interactive Data Entry

To interactively load some data into a vector we can use the *scan* function which can, for example, read data you type until it finds an empty line:

```
> ds <- scan()
1: 54 56 57 59
5: 63 64 66 68
9:
Read 8 items
> ds
[1] 54 56 57 59 63 64 66 68
```

Alternatively, we may have a simple list of numbers in one of our windows on the screen and simply wish to load this into R. For example, select the list of numbers below, assuming we are reading this document on the screen (e.g., we hold down the left mouse button while highlighting the list of numbers):

```
54 56 57 59 63 64 66 68 68 72 72 75 76 81 84 88 106
```

In the R console window specify the filename to be the special name, *clipboard*, and the selected numbers will be read. For example:

```
> ds <- scan("clipboard")
Read 17 items
> ds
[1] 54 56 57 59 63 64 66 68 68 72 72 75 76 81 84 88 106
```

All of the above read the data into a vector. An alternative is to load data into a data frame. Similar approaches are possible. Suppose we have this height and weight data for 30 eleven year old girls attending Heaton Middle School, Bradford UK (<http://www.sci.usq.edu.au/staff/dunn/Datasets/Books/Hand/Hand-R/height-R.html>, Hand 96).

Height	Weight
135	26
146	33
153	55
154	50
139	32
131	25
149	44
137	31
143	36
146	35
141	28
136	28
154	36
151	48
155	36
133	31
149	34
141	32
164	47

```

146      37
149      46
147      36
152      47
140      33
143      42
148      32
149      32
141      29
137      34
135      30

```

We can also place the data into a string within R and read the data from directly from the string:

```

lines <- "Height
54
56
57
59
63
64
66
68
68
72
72
75
76
81
84
88
106
"
ds <- read.table(textConnection(lines), header = TRUE)

```

14.3.3 Available Datasets

R provides many datasets that can serve to illustrate much of its functionality. We will use a number of the datasets here for this same purpose. You can get a list of available datasets with the *data* function:

```

> data()
Data sets in package 'datasets':
AirPassengers      Monthly Airline Passenger Numbers 1949-1960
BJsales            Sales Data with Leading Indicator
BJsales.lead (BJsales)   Sales Data with Leading Indicator
BOD                Biochemical Oxygen Demand

```

C02	Carbon Dioxide uptake in grass plants
ChickWeight	Weight versus age of chicks on different diets
[...]	
warpbreaks	The Number of Breaks in Yarn during Weaving
women	Average Heights and Weights for American Women

The Iris Dataset

The *iris* dataset is available in a standard installation of R and is a dataset used in many statistical text books.

14.3.4 CSV Data Used In The Book

A very convenient, simple, and universally recognised data format is the trivial text data file with one record of data per line within the file and each line containing comma separated fields. Such a format is referred to as comma separated values (or CSV for short). Such a simple format overs many advantages over proprietary formats, including the straightforward ability to share the data easily amongst many applications. Also, for many processing tasks where all of the data is touched, access to such a simply format is considerably faster than through, for example database querying. While the sophisticated database administrator can certainly explore and tune and index a database to provide targeted, efficient and fast access for particular queries, simple progression through a CSV file requires much less sophistication, generally without sacrificing performance, and often with improved performance.

Another advantage is that the steps written to process a CSV data file, using R to implement the processing, can simply and freely be transferred from platform to platform, whether it be GNU/Linux or MS/Windows. Thus the investment in processing and delivering results from the data are enhanced.

Comma Separated Value (CSV) files can be read and written in R using *read.csv* and *write.table*. Our first example obtains a small (12K) CSV file from the Internet([Blake and Merz, 1998](#)) using the *download.file* function. The data is then loaded into R, with the appropriate column names added (since the dataset doesn't come with the names). We then save the dataset to a new CSV file (with the right headers) using *write.table*,

as well as to a binary R data file using *save*. The resulting data files will be used in the examples throughout the book.

These datasets will be used throughout the book to illustrate various techniques and approaches to data cleaning, variable selection, and modelling.

The Wine Dataset

The *wine* dataset contains the results of a chemical analysis of wines grown in a specific area of Italy. Three types of wine are represented in the 178 samples, with the results of 13 chemical analyses recorded for each sample. Note that we transform the Type into a categorical variable, but this information is only recovered in the binary R dataset, and not the CSV dataset.

```
UCI <- "ftp://ftp.ics.uci.edu/pub"
REPOS <- "machine-learning-databases"
wine.url <- sprintf("%s/%s/wine/wine.data", UCI, REPOS)
wine <- read.csv(wine.url, header=F)
colnames(wine) <- c('Type', 'Alcohol', 'Malic', 'Ash',
                     'Alcalinity', 'Magnesium', 'Phenols',
                     'Flavanoids', 'Nonflavanoids',
                     'Proanthocyanins', 'Color', 'Hue',
                     'Dilution', 'Proline')
wine$Type <- as.factor(wine$Type)
write.table(wine, "wine.csv", sep=",", row.names=FALSE)
save(wine, file="wine.Rdata", compress=TRUE)
}
```

R code source: [get-wine.R](#).

At a later time you can simply read in the CSV dataset or else load in the R dataset:

```
> wine <- read.csv("wine.csv")
OR
> load("wine.RData")
> dim(wine)
[1] 178 14
> str(wine)
'data.frame': 178 obs. of 14 variables:
 $ Type      : Factor w/ 3 levels "1","2","3": 1 1 1 1 1 1 1 1 1 ...
 $ Alcohol   : num  14.2 13.2 13.2 14.4 13.2 ...
 $ Malic     : num  1.71 1.78 2.36 1.95 2.59 1.76 1.87 2.15 1.64 1.35 ...
 $ Ash       : num  2.43 2.14 2.67 2.5 2.87 2.45 2.45 2.61 2.17 2.27 ...
 $ Alcalinity: num  15.6 11.2 18.6 16.8 21 15.2 14.6 17.6 14 16 ...
 \$ Magnesium: int  127 100 101 113 118 112 96 121 97 98 ...
 $ Phenols   : num  2.8 2.65 2.8 3.85 2.8 3.27 2.5 2.6 2.8 2.98 ...
```

```
$ Flavanoids      : num  3.06 2.76 3.24 3.49 2.69 3.39 2.52 2.51 2.98 3.15 ...
$ Nonflavanoids   : num  0.28 0.26 0.3 0.24 0.39 0.34 0.3 0.31 0.29 0.22 ...
$ Proanthocyanins: num  2.29 1.28 2.81 2.18 1.82 1.97 1.98 1.25 1.98 1.85 ...
$ Color          : num  5.64 4.38 5.68 7.8 4.32 6.75 5.25 5.05 5.2 7.22 ...
$ Hue            : num  1.04 1.05 1.03 0.86 1.04 1.05 1.02 1.06 1.08 1.01 ...
$ Dilution       : num  3.92 3.4 3.17 3.45 2.93 2.85 3.58 3.58 2.85 3.55 ...
$ Proline         : int   1065 1050 1185 1480 735 1450 1290 1295 1045 ...
```

Note that R provides a useful interactive file chooser through the function `file.choose`. This will prompt for a file name, and provides tab completion.

```
> ds <- read.csv(file.choose())
```

The Cardiac Arrhythmia Dataset

The Arrhythmia dataset will be used to illustrate issues with data cleaning.

The dataset is of moderate size (392Kb), with 452 entities. This dataset has 280 variables, one being an output variable with 16 values. Of the input variables some 40 of them are categorical. Although a meta-data file on the repository lists the variables, we may not want to give them all names just now (too many to do by hand). We select a few to give other than the default R names to them. As with other data from the UCI repository ? is used for missing values and we deal with that when we read the downloaded data into R.

```
> UCI <- "ftp://ftp.ics.uci.edu/pub"
> REPOS <- "ml-repos/machine-learning-databases"
> cardiac.url <- sprintf("%s/%s/arrhythmia/arrhythmia.data", UCI, REPOS)
> download.file(cardiac.url, "cardiac.data")
> cardiac <- read.csv("cardiac.data", header=F, na.strings="?")
> summary(cardiac)

      V1          V2          V3          V4
Min.   : 0.00   Min.   :0.0000   Min.   :105.0   Min.   : 6.00
1st Qu.:36.00  1st Qu.:0.0000   1st Qu.:160.0   1st Qu.: 59.00
Median :47.00  Median :1.0000   Median :164.0   Median : 68.00
Mean   :46.47  Mean   :0.5509   Mean   :166.2   Mean   : 68.17
3rd Qu.:58.00  3rd Qu.:1.0000   3rd Qu.:170.0   3rd Qu.: 79.00
Max.   :83.00  Max.   :1.0000   Max.   :780.0   Max.   :176.00
[...]
> str(cardiac)
'data.frame': 452 obs. of 280 variables:
 $ V1 : int  75 56 54 55 75 13 40 49 44 50 ...
 $ V2 : int  0 1 0 0 0 1 1 0 1 ...
 $ V3 : int  190 165 172 175 190 169 160 162 168 167 ...
```

```
$ V4   : int  80 64 95 94 80 51 52 54 56 67 ...
$ V5   : int  91 81 138 100 88 100 77 78 84 89 ...
$ V6   : int  193 174 163 202 181 167 129 0 118 130 ...
$ V7   : int  371 401 386 380 360 321 377 376 354 383 ...
$ V8   : int  174 149 185 179 177 174 133 157 160 156 ...
$ V9   : int  121 39 102 143 103 91 77 70 63 73 ...
$ V10  : int  -16 25 96 28 -16 107 77 67 61 85 ...
[...]
$ V278: num  23.3 20.4 12.3 34.6 25.4 13.5 14.3 15.8 12.5 20.1 ...
$ V279: num  49.4 38.8 49 61.6 62.8 31.1 20.5 19.8 30.9 25.1 ...
$ V280: int   8 6 10 1 7 14 1 1 1 10 ...
```

We will now give a names to a few columns, then save it to a cleaner CSV file and a binary RData file where ? will be NA, and all columns will have names, some that we have given, and the rest as given by R.

```
> colnames(cardiac)[1:4] <- c("Age", "Gender", "Height", "Weight")
> write.table(cardiac, "cardiac.csv", sep=",", row.names=F)
> save(cardiac, file="cardiac.RData", compress=TRUE)
> dim(cardiac)
[1] 452 280
> str(cardiac)
'data.frame': 452 obs. of 280 variables:
 $ Age   : int  75 56 54 55 75 13 40 49 44 50 ...
 $ Gender: int  0 1 0 0 0 1 1 0 1 ...
 \$ Height: int  190 165 172 175 190 169 160 162 168 167 ...
 $ Weight: int  80 64 95 94 80 51 52 54 56 67 ...
 $ V5    : int  91 81 138 100 88 100 77 78 84 89 ...
[...]
```

The Adult Survey Dataset

The *survey* dataset is a little larger (3.8MB) and illustrates many more of the options to the *read.csv* function. The data was extracted from the US Census Bureau database, and is again available from the UCI Machine Learning Repository.

```
UCI <- "ftp://ftp.ics.uci.edu/pub"
REPOS <- "machine-learning-databases"
survey.url <- sprintf("%s/%s/adult/adult.data", UCI, REPOS)
survey <- read.csv(survey.url, header=F, strip.white=TRUE,
na.strings="?", 
col.names=c("Age", "Workclass", "fnlwgt",
"Education", "Education.Num", "Marital.Status",
"Occupation", "Relationship", "Race", "Sex",
"Capital.Gain", "Capital.Loss",
"Hours.Per.Week", "Native.Country",
"Salary.Group"))
write.table(survey, "survey.csv", sep=",", row.names=F)
```

```
save(survey, file="survey.Rdata", compress=TRUE)
}
```

R code source: [get-survey.R](#).

```
> dim(survey)
[1] 32561      15
> str(survey)
'data.frame':   32561 obs. of  15 variables:
 $ Age        : int  39 50 38 53 28 37 49 52 31 42 ...
 $ Workclass   : Factor w/ 8 levels "Federal-gov",...: 7 6 4 4 4 4 4 4 ...
 $ fnlwgt     : int  77516 83311 215646 234721 338409 284582 160187 ...
 $ Education   : Factor w/ 16 levels "10th","11th",...: 10 10 12 2 10 13 7 ...
 $ Education.Num: int  13 13 9 7 13 14 5 9 14 13 ...
 $ Marital.Status: Factor w/ 7 levels "Divorced",...: 5 3 1 3 3 3 4 3 5 3 ...
 $ Occupation   : Factor w/ 14 levels "Adm-clerical",...: 1 4 6 6 10 4 8 ...
 $ Relationship  : Factor w/ 6 levels "Husband","Not-in-family",...: 2 1 2 1 ...
 $ Race         : Factor w/ 5 levels "Amer-Indian-Eskimo",...: 5 5 5 3 3 5 ...
 $ Sex          : Factor w/ 2 levels "Female","Male": 2 2 2 2 1 1 1 2 1 2 ...
 $ Capital.Gain : int  2174 0 0 0 0 0 0 14084 5178 ...
 $ Capital.Loss  : int  0 0 0 0 0 0 0 0 0 0 ...
 $ Hours.Per.Week: int  40 13 40 40 40 16 45 50 40 ...
 $ Native.Country: Factor w/ 41 levels "Cambodia","Canada",...: 39 39 39 39 ...
 $ Salary.Group  : Factor w/ 2 levels "<=50K",>50K": 1 1 1 1 1 1 2 2 2 ...
```

Once again, the dataset can be read in from the CSV file or else loaded as an R dataset:

```
> survey <- read.csv("survey.csv")
OR
> load("survey.RData")
```

14.4 Saving Data

All R objects can be saved using the *save* function and then restored at a later time using the *load* function. The data will be saved into a .RData file. To illustrate this we make use of a standard dataset called *iris*.

We create a random sample of 20 entities from the dataset. This is done by randomly sampling 20 numbers between 1 and the number of rows (*nrow*) in the *iris* dataset, using the *sample* function. The list of numbers generated by *sample* is then used to index the *iris* dataset, to select the sample of rows, by supplying this list of rows as the first argument in the square brackets. The second argument in the square brackets is left blank, indicating that all columns are required in our new dataset. We

then save the dataset to file using the *save* function which compresses the data for storage:

```
> rows <- sample(1:nrow(iris), 20)
> myiris <- iris[rows,]
> dim(myiris)
[1] 20  5
> save(myiris, file="myiris.RData", compress=TRUE)
```

At a later date you can load your dataset back into R with the *load* function:

```
> load("myiris.RData")
> dim(myiris)
[1] 20  5
```

Using the *compress* option will reduce disk space required to store the dataset.

You can save any objects in an R binary file. For example, suppose you have built a model and want to save it for later exploration:

```
> library(rpart)
> iris.rp <- rpart(Species ~ ., data=iris)
> save(iris.rp, file="irisrp.RData", compress=TRUE)
```

At a later stage, perhaps on a fresh start of R, you can load the model:

```
> load("irisrp.RData")
> iris.rp
n= 150

node), split, n, loss, yval, (yprob)
 * denotes terminal node

1) root 150 100 setosa (0.33333333 0.33333333 0.33333333)
  2) Petal.Length< 2.45 50  0 setosa (1.00000000 0.00000000 0.00000000) *
  3) Petal.Length>=2.45 100  50 versicolor (0.00000000 0.50000000 0.50000000)
     6) Petal.Width< 1.75 54   5 versicolor (0.00000000 0.90740741 0.09259259) *
     7) Petal.Width>=1.75 46   1 virginica (0.00000000 0.02173913 0.97826087) *
```

To identify what is saved into an RData file you can *attach* the file and then get a listing of its contents:

```
attach("irisrp.RData")
ls(2)
...
detach(2)
```

Reading Direct from URL

```
> audit <- read.csv("http://rattle.togaware.com/audit.csv")[,c(2:10,13)]
> myrpart <- rpart(Adjusted ~ ., data=audit, method="class")
```

14.4.1 Formatted Output

The *format.df* function of the *Hmisc* package will format a data frame or matrix as a L^AT_EX table. Also see *latex* in *Hmisc*.

Use *format* for sophisticated formatting of numbers, such as inserting commas in thousands, etc. Also for formatting strings.

14.4.2 Automatically Generate Filenames

To generate a series of filenames, all with the same base name but having a sequence of numbers, as might be the case when you generate a sequence of plots and want to save each to a different file, the traditional R approach is to use *formatC*:

```
paste("df", formatC(1:10, digits=0, wid=3, format=d, flag="0"), ".dat", sep="")
[1] "df001.dat" "df002.dat" "df003.dat" "df004.dat" "df005.dat" "df006.dat"
[7] "df007.dat" "df008.dat" "df009.dat" "df010.dat"
```

Even easier though is to use the newer *sprintf*. Here's a couple of alternatives. The first assumes a fixed number of digits for the increasing number, whilst the second uses only the number of digits actually required, by finding out the number of digits using *nchar*:

```
> n <- 10
> sprintf("df%03d.dat", 1:10)
[1] "df001.dat" "df002.dat" "df003.dat" "df004.dat" "df005.dat" "df006.dat"
[7] "df007.dat" "df008.dat" "df009.dat" "df010.dat"
> sprintf("plot%0*d", nchar(n), 1:n)
[1] "plot01" "plot02" "plot03" "plot04" "plot05" "plot06" "plot07" "plot08"
[9] "plot09" "plot10"
> n <- 100
> sprintf("plot%0*d", nchar(n), 1:n)
[1] "plot001" "plot002" "plot003" "plot004" "plot005" "plot006" "plot007"
[...] 
[99] "plot099" "plot100"
```

14.5 Using SQLite

SQLite (from www.sqlite.org) is an open source database package that is well supported in R. It has the advantage that it requires no setup or administration (other than installing the package) and is an embedded system so that there is less of a connection overhead. You are able to manage very large datasets in SQLite without needing to load all the data into memory, unlike R itself, so that you are able to manipulate the data using SQL then load in just the data you need.

For small dataset SQLite is a good choice, but for very large datasets, MySQL still performs very well.

There is also a project under way as part of the [Google Summer of Code](#) project, that aims to create a package that will store data frames and matrices into sqlite databases, initially called sqlite data frames (sdf). These sdf's will behave like ordinary data frames so that existing R functions will work. This will enable R users to work with very large datasets much more readily, with no user effort.

For now, SQLite allows the easy import and export of data to text files.

```
library(RSQLite)
con <- dbConnect(SQLite(), "foo3.db")
dbGetQuery(con, "pragma cache_size")
  cache_size
  1 2000
dbGetQuery(con, "pragma cache_size=2500")
  NULL
dbGetQuery(con, "pragma cache_size")
  cache_size
  1 2500
```

As an example, first create an empty SQLite database (outside of R) and import a CSV (comma separated value) file, tell sqlite to use commas, not '—':

```
$ sqlite3 -separator , audit.db
sqlite> create table audit(ID INTEGER, Age INTEGER, Employment TEXT,
   Education TEXT, Marital TEXT, Occupation TEXT,
   Income REAL, Sex TEXT, Deductions REAL,
   Hours INTEGER, Accounts TEXT,
   Adjustment REAL, Adjusted INTEGER);
sqlite> .tables
audit
sqlite> .import audit.csv audit
sqlite> select count(*) from audit;
```

```

2001
sqlite> delete from audit where ID='ID';
sqlite> select count(*) from audit;
2000

sqlite> .quit

```

Now in R in the same directory:

```

library(DBI)
library(RSQLite)

driver<-dbDriver("SQLite")
connect<-dbConnect(driver, dbname = "audit.db")
dbWriteTable(connect, "audit", audit, overwrite = T, row.names = F)

dbListTables(connect)
[1] "audit"

query01 <- dbSendQuery(connect, statement = "select * from audit");
data01 <- fetch(query01, n = 10)
contents(data01)

sqliteCloseResult(query01)
sqliteCloseConnection(connect)
sqliteCloseDriver(driver)

```

14.6 ODBC Data

The *RODBC* package provides direct access, through ODBC to database entities.

14.6.1 Database Connection

The basic usage of *RODBC* will connect to a known ODBC object using the *odbcConnect* function and query the database for the tables it exports using *sqlTables*:

```

> library(RODBC)
> channel <- odbcConnect("DWH")
# This may pop up a window to enter username and password
> tables <- sqlTables(channel)
> columns <- sqlColumns(channel, "clients")

```

You can then retrieve the full contents of a table with *sqlFetch*:

```
> ds <- sqlFetch(channel, "tablename")
```

Or else you can send a SQL query to the database:

```
> ds <- sqlQuery(channel, "SELECT * FROM clients WHERE age > 35")
```

Some ODBC drivers, such as the Netezza ODBC driver, have a pre-fetch option that interacts poorly with applications connecting through the driver. With a pre-fetch option the driver appears to report fewer rows being available than actually available. It seems that the number of rows reported is in fact the pre-fetch limited number of rows. For the Netezza ODBC driver, for example, the default is 256 rows. This confuses the application connecting to ODBC (in this case, R through the *RODBC*). The symptom is that we only receive 256 rows from the table. Internally, the application is probably using either the SQLExtendedFetch or SQLFetchScroll ODBC functions.

There are a number of solutions to this issue. One from the applications side is to set *believeNRows* to FALSE. This will then retrieve all the rows from the table. Another solution is at the driver configuration level. For example, in connecting through the Netezza ODBC driver a configuration option is available where you can change the default Prefetch Count value.

An example of the issue is illustrated below:

```
> channel <- odbcConnect("netezza")
> orders <- sqlQuery(channel, "SELECT * FROM orders LIMIT 500")
> dim(orders)
[1] 256   9
> orders <- sqlQuery(channel, "select * from orders limit 500",
+                           believeNRows=FALSE)
> dim(orders)
[1] 500   9
> odbcCloseAll()
```

We can reopen the driver and in the resulting GUI configuration set the Prefetch Count to perhaps 10,000. Then:

```
> channel <- odbcConnect("netezza")
> orders <- sqlQuery(channel, "SELECT * FROM orders LIMIT 500")
> dim(orders)
[1] 500   9
> orders <- sqlQuery(channel, "SELECT * FROM orders LIMIT 50000")
> dim(orders)
[1] 10000  9
> orders <- sqlQuery(channel, "SELECT * FROM orders LIMIT 50000",
```

```
> dim(orders)
[1] 50000      9
>
```

Note that we would not want to default believeNRows to FALSE since, for example, with a Teradata query this increase the query time by some 3 times!

For an SQLite database, edit .odbc.ini

```
[audit]
Description=SQLite test audit database for Rattle
Driver=SQLite3
Database=/home/kayon/projects/rattle/audit.db
# optional lock timeout in milliseconds
Timeout=2000
```

14.6.2 Excel

Logware Watermark For Data Mining Survival

The simplest way to transfer data from Excel, or any spreadsheet in fact, is to save the data in CSV (Comma Separated Value) format, usually into a file with extension `.csv`. This is supported in all spreadsheet applications and is effective in that if we are fluent with data manipulation in Excel, then we can get our data into shape using Excel, and then load it into Rattle for data mining.

Alternatively, on MS/Windows Excel spreadsheets can be directly accessed and manipulated through ODBC using `odbcConnectExcel`. Available sheets can be listed with `sqlTables` and individual sheets can be queried through the `sqlQuery` function or else imported with `sqlFetch`. To use a spreadsheet as a database though, the first row of the spreadsheet must be the column names! If not, we will find that we end up reading from the second row of our data.

In this example we open a connection to a spreadsheet and then give a sample query:

```
> library(RODBC)
> channel <- odbcConnectExcel("h:/audit.xls")
> ds <- sqlQuery(channel, "SELECT * FROM 'Sheet1$'
   WHERE Type = 'TOC'
   AND Valve='5010-05'")
> odbcClose(channel)
```

To simply fetch the full contents of a single sheet of a spreadsheet we can use the *sqlFetch* query:

```
library(RODBC)
channel <- odbcConnectExcel("h:/audit.xls")
ds <- sqlFetch(xlsConnect, "Sheet1")
odbcClose(xlsConnect)
```

On MS/Windows you can also use the *xlsReadWrite* package to directly access and manipulate an Excel spreadsheet. For example, to read a spreadsheet we can use *read.xls*:

```
library(xlsReadWrite)
ds <- read.xls("audit.xls", colNames=TRUE, sheet=6,
               colClasses=c("factor", "integer", "double"))
```

14.6.3 Access

MS/Access databases, on MSWindows, can be directly accessed through ODBC with an *odbcConnectAccess*. A database can be queried for a list of all tables available, using *sqlTables*, and imported with *sqlFetch* or *sqlQuery*. We can then directly *save* the table as an R object for simpler loading into R at a later time:

```
> library(RODBC)
> channel <- odbcConnectAccess("h:/sample.mdb")
> sqlTables(channel)$TABLE_NAME
> clients <- sqlFetch(channel, "Clients")
> odbcClose(channel)
> save(clients, file="clients.RData")
```

14.7 Clipboard Data

Suppose you are reviewing a small sample of data on the screen in any application (e.g., browsing a web site with some sample data). You want to load the data into R. This can be easily accomplished by selecting or highlighting the data with the mouse and telling R to read from the clipboard.

As an example, visit one of the UCI Machine Learning Repository datasets, such as: <http://www.ics.uci.edu/~mlearn/databases/autos/imports-85.data>.

Highlight the first few rows of the data and then run the following *read.table* function with the *file* function identifying the *clipboard* to be read from:

```
> autos <- read.csv(file("clipboard"), header=FALSE)
> autos

  V1 V2          V3 V4 V5 V6          V7 V8 V9 V10 V11 V12
V13 V14
1 3 ? alfa-romero gas std two convertible rwd front 88.6 168.8 64.1 48.8 2548
2 3 ? alfa-romero gas std two convertible rwd front 88.6 168.8 64.1 48.8 2548
3 1 ? alfa-romero gas std two hatchback rwd front 94.5 171.2 65.5 52.4 2823

  V15 V16 V17 V18 V19 V20 V21 V22 V23 V24 V25 V26
1 dohc four 130 mpfi 3.47 2.68 9 111 5000 21 27 13495
2 dohc four 130 mpfi 3.47 2.68 9 111 5000 21 27 16500
3 ohcv six 152 mpfi 2.68 3.47 9 154 5000 19 26 16500
```

You can also use *scan*, for example, to read data from the clipboard into a vector or list:

```
> x <- scan("clipboard", what="")
Read 7 items
> x
[1] "Age"    "Gender"   "Salary"   "Home"    "Vehicle"  "Address"  "Married"
```

To try this out yourself, select the list of strings and run the *scan* function.

You can also write to the clipboard:

```
> write.table(ds, "clipboard", sep="\t", row.names=FALSE)
```

14.8 Map Data

R provides a set of tools for reading geographic (map) data, particularly ESRI *shapefiles*, and plotting and manipulating such data. Maps for many countries are available, particularly the US, Europe, and New Zealand. Limited Australian map data is also freely available.

```
download.file("http://www.vdstech.com/mapdata/australia.zip", "australia.zip")
system("unzip australia.zip; rm australia.zip")
```

R code source: [get-australia.R](#).

The data can be read in with *readShapePoly* and displayed with *plot*. Australia has a few outlying islands which we crop from the main focus of the map here using *xlim* and *ylim*.



```
library(maptools)
aus <- readShapePoly("australia.shp")
plot(aus, lwd=2, border="grey", xlim=c(115,155), ylim=c(-35,-20))
dev.off()
```

R code source: [map-australia-plot.R](#).

The class of the resulting object (*aus*) is *SpatialPolygonsDataFrame*. Such an object has a collection of slots. For example, the *data* slot includes meta information about the region recorded in the data frame.

```
> aus@data
```

	FIPS_ADMIN	GMI_ADMIN		ADMIN_NAME	FIPS_CNTRY	CNTRY_NAME
0	AS01	AUS-ACT	Australian Capital Territory	AS	Australia	
1	AS02	AUS-NSW	New South Wales	AS	Australia	
2	AS03	AUS-NTR	Northern Territory	AS	Australia	
3	AS04	AUS-QNS	Queensland	AS	Australia	

	AS05	AUS-SAS	South Australia	AS	Australia	
4	AS06	AUS-TSM	Tasmania	AS	Australia	
5	AS07	AUS-VCT	Victoria	AS	Australia	
6	AS08	AUS-WAS	Western Australia	AS	Australia	
7						
	REGION	CONTINENT	POP_ADMIN	SQKM_ADMIN	SQMI_ADMIN	
TYPE_ENG						
0	Australia/New Zealand	Australia	292475	2342.295	904.36	Territory
1	Australia/New Zealand	Australia	6338919	803110.812	310081.09	State
2	Australia/New Zealand	Australia	161294	1352365.000	522148.09	Territory
3	Australia/New Zealand	Australia	3107362	1733475.000	669294.69	State
4	Australia/New Zealand	Australia	1445153	985308.500	380427.59	State
5	Australia/New Zealand	Australia	472122	68131.477	26305.56	State
6	Australia/New Zealand	Australia	4354611	227781.406	87946.40	State
7	Australia/New Zealand	Australia	1655588	2533628.000	978233.81	State
TYPE_LOC						
0	Territory					
1	State					
2	Territory					
3	State					
4	State					
5	State					
6	State					
7	State					

The bounding box of the plot is available using `bbox`.

```
> bbox(aus)
      min         max
r1 112.90721 159.10190
r2 -54.75389 -10.05139
```

14.9 Other Data Formats

14.9.1 Fixed Width Data

Suppose we have a fixed-width data file with fields of 10 and 5 characters each.

```
> lines <- readLines("mydata.dat")
> dframe <- data.frame(salary = as.numeric(substr(lines, 1, 10)),
   age = as.numeric(substr(lines, 11, 15)))
```

14.9.2 Global Positioning System

In some situations in analysing data you may actually be collecting the data directly for analysis by R from some external device, possibly connected through the computer's serial port. One such example is using a global positioning system. On a GNU/Linux system this might be connected to your serial device and in R (after ensuring you have read access to the serial port `/dev/ttyS0`) you can read your current position.

```
> gps <- scan(file="/dev/ttyS0", n=1, what="character")
Read 1 items
> gps
[1] "@051226122125S0341825E01500808G006+00350E0000N0000D0000"

> columns <- c("tag", "date", "time", "latitude", "longitude",
   +           "quality", "level", "movelong", "movelat", "movevert")
> widths <- c(1, 6, 6, 8, 9, 4, 6, 5, 5, 5)
> gps.data <- read.fwf("/dev/ttyS0", widths, col.names=columns, n=1,
   +                      colClasses="character")
> gps.data
  tag    date     time latitude longitude quality  level movelong movelat movevert
1  @ 050221 122125 S0341825 E01500808      G006 +00350     E0000     N0000
D0000
```

This tells me that the location is 34degrees, 18.25minutes south, 150degrees, 8.08minutes east. The quality is good with a 6meter positional error. This is 35m above sea level, and not moving.

14.10 Documenting a Dataset

R provides support for documenting a dataset through `.Rd` files that R will format. You can create a template based on a particular dataset using the `prompt` function:

```
> load("survey.RData")
> prompt(survey)
Created file named 'survey.Rd'.
Edit the file and move it to the appropriate directory.
```

14.11 Common Data Problems

Data size - too much data, too little relevant.

Data sufficiency - current data warehouses don't contain all necessary customer data.

Data integrity - data in different sources don't always match.

Data accessibility - DBAs control the data access.

Data understandability - data dictionary, metadata.

Data quality - Errors, missing values.

Data reliability - out-of-date data, unreliable data.

Integration of external data sources.

Togaware Watermark For Data Mining Survival

Chapter 15

Graphics in R

As well as being a package of choice for many Statisticians, R is capable of producing excellent graphics in many formats, including PostScript, PDF, PNG (for vector images and text), and JPG (for colorful images).

Example code presented in the following chapters will illustrate the generation of publication quality PDF (portable document format) graphics that can be viewed with many viewers, including Adobe's Acrobat. However, R supports many output formats, including PNG (portable network graphics, supported by many web browsers and importable into many word processors), JPG, and PostScript. Another format supported is XFIG. Such output is editable with the *xfig* graphics editor, allowing further annotations and modifications to be made to the automatically generated plot. The XFIG graphics can then be converted to an even larger collection of graphics formats, including PDF. For the graphics actually presented here in the book R has been used, in fact, to generate XFIG output which is then converted to PDF. Thus the code examples here, generating PDF directly, may give slightly different layouts to the figures that actually appear here.

A highly interoperable approach is to generate graphs in FIG format which can then be loaded into the **xfig** application, for example, for further editing. This allows, for example, minor changes to be made to fine tune the graphics, but at the cost of losing the ability to automatically

regenerate the plot from the original R code. For L^AT_EX processing the **rubber** package (under Debian GNU/Linux) will automatically convert them to the appropriate EPS or PDF format. Of course, xfig can also generate PNG and JPG and many other formats.

The basic concept of R's graphics model is that a plot is built up bit by bit. Each latter component of the plot overlays earlier components. A plot also has two components. The plotting area is identified by through the *usr* parameter, as 4 numbers x_1 , x_2 , y_1 , and y_2 . You can retrieve the current plotting region (which is defined by the first component of a plot) with:

```
> plot(rnorm(10))
> par("usr")
[1] 0.640000 10.360000 -1.390595 1.153828
```

The whole figure itself will encompass the plotting region and the region around the plot used to add axis information and labels. Outside of the figure region is the device region. Normally, adding components to a plot, outside of the plotting region, will have no effect—they will be cropped. To ensure they do not get cropped, set the graphic parameter *xpd* to TRUE:

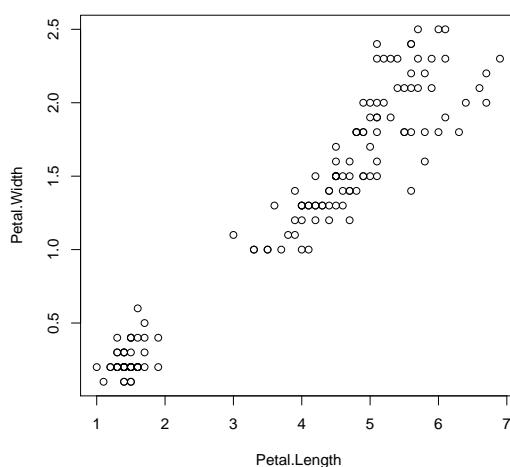
```
> par(xpd=TRUE)
```

15.1 Basic Plot

R's *plot* function provides the basic interface to the sophisticated graphics system. Given any object *plot* will endeavour to determine how to display the object. Other components of the graphic can then be built up using a variety of functions. As new components are added, they lay on top of what is already on the graphic, possibly occluding other components.

Standard types of plots include *scatterplots*, *boxplots*, *barplots*, *histograms*, and *piecharts*. Each can be quite simply generated through high level calls to R functions.

The simplest of plots is the scatterplot which displays the location of points on a two dimensional plot. Here we *attach* the common *iris* dataset and choose two variables to *plot*: *Petal.Length* and *Petal.Width*. The *attach* function allows the column names to be used without the normal *iris\$* prefix (by adding the dataset to the search path). A *detach* removes the object from the search path. The resulting scatterplot illustrates some degree of correlation between these two variables, in that, generally speaking, for larger petal lengths, the petal width is also larger. We can also see two clear groups or clusters in this data: a cluster of entities with a petal length less than 2 and width less than about 0.6, and another group with petal length greater than about 3 and petal width greater than about 0.9.



```
attach(iris)
```

```
plot(Petal.Length, Petal.Width)
detach()
```

R code source: [rplot-iris-scatter.R](#).

If the dataset is only being used by the *plot* function and nowhere else then we could use *with*:

```
with(iris, plot(Petal.Length, Petal.Width))
```

Of course, we could simply use the full path to each column of data to be plotted:

```
plot(iris$Petal.Length, iris$Petal.Width)
```

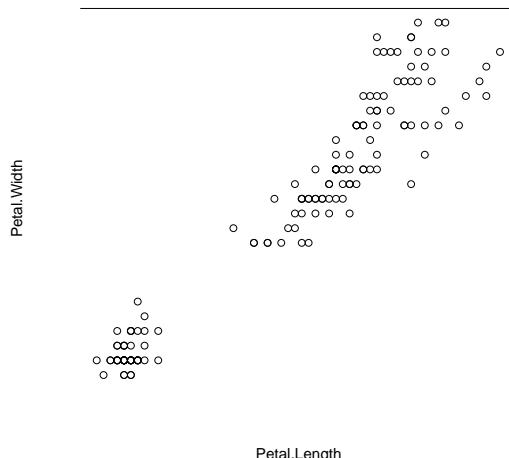
Togaware Watermark For Data Mining Survival

15.2 Controlling Axes

You can build up various features of a plot bit by bit. For example, you can specify how much of a box to add around the plot with the *box* function. In fact, you can use any of the following box line options to draw different extents of the box. The sides are number from 1 for the lower side, then clockwise.

```
"o" gives you all four sides (1:4)
"l" gives you left and lower (2 and 1)
"7" gives you upper and right (3:4)
"c" gives you all except right (1:2, 3)
"u" gives you all except upper (1:2, 4)
"]" gives you all except left (1, 3:4)
```

The plot function here requests that the axes not be drawn, and the chosen box is then drawn on top of the current plot.



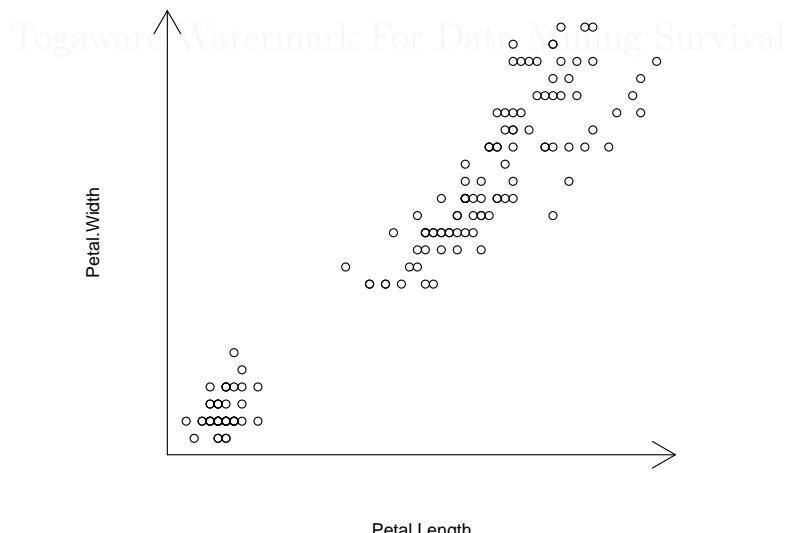
```
attach(iris)
plot(Petal.Length, Petal.Width, axes=FALSE)
box(bty='7')
```

R code source: [rplot-iris-topbox.R](#).

15.3 Arrow Axes

The `par` function in R can be used to fine tune characteristics of a plot. Here we add arrows to the axes. For many more arrow options see the `p.arrows` function in the `sfsmisc` package.

In the example here, `par("usr")` is used to identify the bounds of the plot, and these are then employed to draw axes with arrows. The `arrows` function takes a starting point and an ending point for the line to be drawn. The `code=2` selects the type of arrow head to draw and the `xpd=TRUE` ensures the arrows are not cropped to the default size of the plot (the plot region).

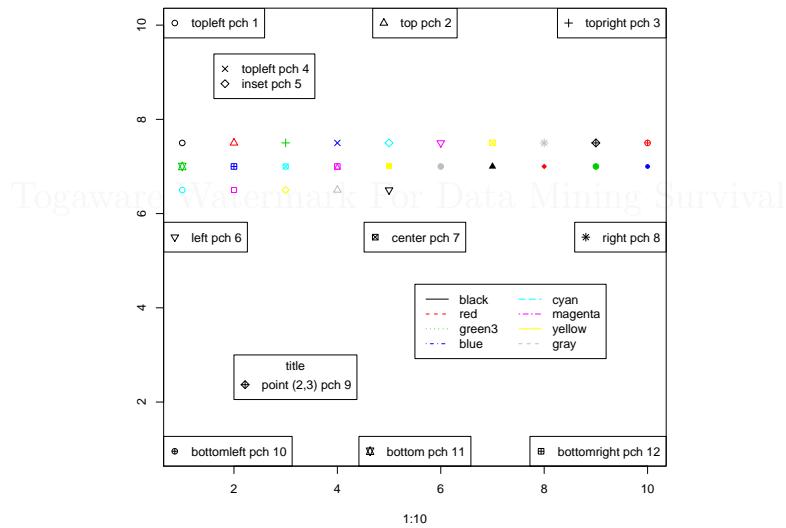


```
attach(iris)
plot(Petal.Length, Petal.Width, axes=FALSE)
u <- par("usr")
arrows(u[1], u[3], u[2], u[3], code=2, xpd=TRUE)
arrows(u[1], u[3], u[1], u[4], code=2, xpd=TRUE)
```

R code source: [rplot-iris-arrows.R](#).

15.4 Legends and Points

The *legend* function is used to add a legend to a plot. A specific coordinate can be given as the first two arguments, or else a symbolic location, such as *topleft* and *center*, can be given. The *pch* option specifies a plotting symbol, and the first 25 (1:25) are shown on the plot below. Other symbols are available in the 32:255 range. The *lty* is used to specify line types. The *col* is used to specify colours.



```
plot(1:10, rep(7.5,10), ylab="", ylim=c(1,10), xlim=c(1,10), pch=1:10, col=1:10)
points(1:10, rep(7,10), pch=11:20, col=11:20)
points(1:5, rep(6.5,5), pch=21:25, col=21:25)
legend("topleft", "topleft pch 1", pch=1)
legend("top", "top pch 2", pch=2)
legend("topright", "topright pch 3", pch=3)
legend("left", "left pch 6", pch=6)
legend("center", "center pch 7", pch=7)
legend("right", "right pch 8", pch=8)
legend("bottomleft", "bottomleft pch 10", pch=10)
legend("bottom", "bottom pch 11", pch=11)
legend("bottomright", "bottomright pch 12", pch=12)
legend <- c("topleft pch 4", "inset pch 5")
legend("topleft", legend, inset=c(0.1, 0.1), pch=c(4,5))
legend(2, 3, legend="point (2,3) pch 9", title="title", pch=9)
plen <- length(palette())
legend(5.5, 4.5, palette(), lty=1:plen, col=1:plen, ncol=2)
```

R code source: [rplot-legends.R](#).

Togaware Watermark For Data Mining Survival

15.4.1 Colour

R provides a collection of predefined colours which you can use to name colours, where appropriate. The list of colours is obtained from the *colour* function:

```
> colours()
[1] "white"          "aliceblue"        "antiquewhite"
[4] "antiquewhite1"  "antiquewhite2"    "antiquewhite3"
[7] "antiquewhite4"  "aquamarine"       "aquamarine1"
[10] "aquamarine2"   "aquamarine3"     "aquamarine4"
[13] "azure"          "azure1"          "azure2"
[16] "azure3"          "azure4"          "beige"
[19] "bisque"          "bisque1"         "bisque2"
[...]
[646] "wheat"          "wheat1"          "wheat2"
[649] "wheat3"          "wheat4"          "whitesmoke"
[652] "yellow"          "yellow1"         "yellow2"
[655] "yellow3"         "yellow4"         "yellowgreen"
```

There's plenty of colours there to choose from!

The *col* option of a plot is used to change any default colours used by a plot. You can supply a list of integers which will index the output of a call to the *palette* function. The default palette is:

```
> palette()
[1] "black"      "red"        "green3"      "blue"        "cyan"        "magenta"    "yellow"
[8] "gray"
```

You can generate a contiguous colour palette using *cm.colors*.

```
> cm.colors(10)
[1] "#80FFFF" "#99FFFF" "#B2FFFF" "#CCFFFF" "#E6FFFF" "#FFE6FF" "#FFCCFF"
[8] "#FFB2FF" "#FF99FF" "#FF80FF"
```

Similarly, to generate a sequence of colours from a rainbow:

```
> rainbow(10)
[1] "#FF0000" "#FF9900" "#CCFF00" "#33FF00" "#00FF66" "#00FFFF" "#0066FF"
[8] "#3300FF" "#CC00FF" "#FF0099"
```

To generate a sequence of six grays you can use the *gray* function:

```
> gray(seq(0.1, 0.9, len=6))
[1] "#1A1A1A" "#424242" "#6B6B6B" "#949494" "#BDBDBD" "#E6E6E6"
```

15.5 Symbols

See `demo(Hershey)`.

Togaware Watermark For Data Mining Survival

15.6 Multiple Plots

Use layout.

Togaware Watermark For Data Mining Survival

15.7 Other Graphic Elements

Place onto a plot *text* in a *rect* box with background shading, using *strwidth* to determine the rectangles coordinates:

```
text <- "Some Text"; x <- 10; y <- 10
xpad <- 0.1; ypad <- 1.0; bg="wheat"
w <- strwidth(text) + xpad*strwidth(text)
h <- strheight(text) + ypad*strheight(text)
rect(x-w/2, y-h/2, x+w/2, y+h/2, col=bg)
text(x, y, text)
```

Togaware Watermark For Data Mining Survival

15.8 Maths in Labels

A large collection of mathematical symbols are available for adding to plots.

Togaware Watermark For Data Mining Survival

15.9 Making an Animation

Create 10 files of jpg

```
frames <- 10
for(i in 1:frames)
{
  jpeg(sprintf("ani_%02d.jpg", i))
  plot(1:10, 1:10, col=i)
  dev.off()
}
```

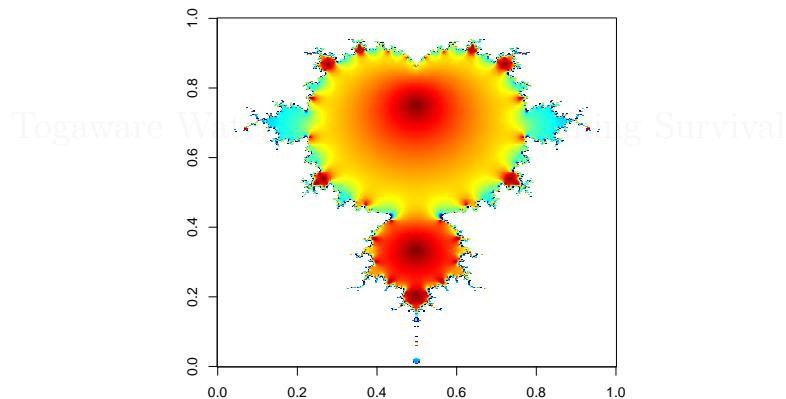
Then use ImageMagick to transform to an animated gif:

```
system("convert -delay 10 ani_???.jpg animation.gif")
```

Togaware Watermark For Data Mining Survival

15.10 Animated Mandelbrot

The following example, suggested by Jaroslaw Tuszynski (author of the *caTools* package), illustrates the use of complex numbers constructed using *complex*, as well as the writing of multiple images to a *gif* file to construct an animated *gif*. Some 160,000 (400 by 400) complex numbers are generated and stored in a matrix. A list of 20 arrays is then built, each being 400 by 400. The magic is then done in the for loop, 20 times, to generate 20 images. All images are written to a single *gif* file using *write.gif*. The final image (*X[, ,k]*) is also displayed using *image*. We use *tim.colors* from *fields* to select a pleasing rainbow colour sequence, but replace the first with *transparent* to achieve a clear background.



```
library(fields) # for tim.colors
library(caTools) # for write.gif
m <- 400 # grid size
C <- complex(real=rep(seq(-1.8,0.6, length.out=m), each=m),
              imag=rep(seq(-1.2,1.2, length.out=m), m))
C <- matrix(C, m, m)
Z <- 0
X <- array(0, c(m, m, 20))
for (k in 1:20)
{
  Z <- Z^2+C
  X[, ,k] <- exp(-abs(Z))
}
col <- tim.colors(256)
col[1] <- "transparent"
write.gif(X, "graphics/rplot-mandelbrot.gif", col=col, delay=100)
image(X[, ,k], col=col) # show final image in R
```

R code source: [rplot-mandelbrot.R](#).

Togaware Watermark For Data Mining Survival

15.11 Adding a Logo to a Graphic

```
library(pixmap)
# From the addlogo example
x <- read.pnm(system.file("pictures/logo.ppm", package="pixmap")[1])
fg <- matrix(c(0,1,0,1,0,.05,0,.05), ncol=4, byrow=TRUE)
split.screen(fg)
screen(1)
plot(rnorm(100))
screen(2)
addlogo(x,c(0,1),c(0,1))
```

15.12 Graphics Devices Setup

R produces its graphics on a graphics device. By default, a screen-based graphics device is usually opened when any new plot is generated. Generally, a user will interactively create the graphic they are interested in, as a series of R function calls, and then either request this to be saved to a particular file format, or else will repeat the same sequence of R function calls, but with a different graphics device (associated with a specific file format and filename). The default graphics device tends to be *x11* on Unix and GNU/Linux, *windows* on MS/Windows, and *quartz* on the Macintosh OS X.

15.12.1 Screen Devices

A new graphics device is created with any one of *x11*, *windows*, and *quartz*. Once a graphics device is opened, all further graphics commands relate to that device, until it is closed with the *dev.off* function. The *graphics.off* function closes all devices.

```
> x11()
> plot(rnorm(50))
> x11()
> plot(rnorm(10))
> dev.off()
X11
 2
> dev.off()
null device
 1
```

On screen devices you can sometimes print the plots directly from the right mouse button menu (depending on the operating system and the

GUI). Once you have a plot drawn on a screen device you can always save the current plot to PDF with:

```
dev.print(pdf, file="rplot.pdf")
```

15.12.2 Multiple Devices

At any one time, any number of graphics devices may be present, but only one is active. To list all open devices use *dev.list*. To identify the active device use *dev.cur*, and to make a device current use *dev.set*. The functions *dev.next* and *dev.prev* makes the next or previous device active.

```
> plot(iris$Sepal.Length)
> x11()
> plot(iris$Sepal.Width)
> dev.list()
X11 X11
 2   3
> dev.cur()
X11
 3
> dev.set(2)
X11
 2
> dev.cur()
X11
 2
```

This allows two plots to be displayed separately.

15.12.3 File Devices

In order to save the graphics to a file the following file devices are provided:

```
> bitmap("sample.png")          # GhostScript to generate any format
> bmp("sample.bmp")           # MS/Windows only
> fig("sample.fig")           # Vector based and editable with xfig
> jpeg("sample.jpg")           # Bitmap
> pdf("sample.pdf")            # Vector based Portable Document Format
> png("sample.png")           # Bitmap
> postscript("sample.eps")      # Vector based
> win.metafile("sample.emf")    # MS/Windows only
```

Once created, you can start building the elements of the graphic you wish to produce (and there are plenty of examples of building a variety

of graphics throughout this book). Once complete, close the specific graphic device with:

```
> dev.off()
```

Each device tends to have a collection of options that fine tune the capabilities of the device. Refer to the on-line documentation for details:

```
> ?pdf
```

To generate a an older tiff format (now largely superseded by the png format for graphics that include vectors and text), perhaps as 24-bit RGB output (8 bits per component), you can use the bitmap device:

```
> bitmap("sample.tiff", type="tiff24nc")
```

Any output device supported by GhostScript is available. GhostScript needs to be installed, but once installed you can get a list of supported output devices:

```
$ gs --help
[...]
Available devices:
alc1900 alc2000 alc4000 alc4100 alc8500 alc8600 alc9100 ap3250 appledmp
atx23 atx24 atx38 bbox bit bitcmyk bitrgb bj10e bj10v bj10vh bj200 bjc600
bjc800 bjc880j bjccmyk bjccolor bjcgray bjcmono bmp16 bmp16m bmp256
bmp32b bmpa16 bmpa16m bmpa256 bmpa32b bmpamono bmpasep1 bmpasep8 bmpgray
bmpmono bmpsep1 bmpsep8 ccr cdeskjet cdj1600 cdj500 cdj550 cdj670 cdj850
cdj880 cdj890 cdj970 cdjcolor cdjmono cfax cgm24 cgm8 cgmono chp2200 cif
cljet5 cljet5c cljet5pr coslw2p coslwxl cp50 cups declj250 deskjet
devicen dfaxhigh dfaxlow dj505j djet500 djet500c dl2100 dnj650c epl2050
epl2050p epl2120 epl2500 epl2750 epl5800 ep15900 epl6100 epl6200 eps9high
eps9mid epson epsonc epswrite escp escpage faxg3 faxg3d faxg4 fmlbp fmpr
fs600 gdi hl1240 hl1250 hl7x0 hpdj1120c hpdj310 hpdj320 hpdj340 hpdj400
hpdj500 hpdj500c hpdj510 hpdj520 hpdj540 hpdj550c hpdj560c hpdj600
hpdj660c hpdj670c hpdj680c hpdj690c hpdj850c hpdj855c hpdj870c hpdj890c
hpdjplus hpdjportable ibmpro ijs imagen inferno iwhi iwlo iwlq jetp3852
jj100 jpeg jpeggray la50 la70 la75 la75plus laserjet lbp310 lbp320 lbp8
lex2050 lex3200 lex5700 lex7000 lips2p lips3 lips4 lips4v lj250 lj3100sw
lj4dith lj4dithp lj5gray lj5mono ljet2p ljet3 ljet3d ljet4 ljet4d
ljet4pj1 ljetplus ln03 lp1800 lp1900 lp2000 lp2200 lp2400 lp2500 lp2563
lp3000c lp7500 lp7700 lp7900 lp8000 lp8000c lp8100 lp8200c lp8300c
lp8300f lp8400f lp8500c lp8600 lp8600f lp8700 lp8800c lp8900 lp9000b
lp9000c lp9100 lp9200b lp9200c lp9300 lp9400 lp9500c lp9600 lp9600s
lp9800c lq850 lx5000 lxm3200 lxm5700m m8510 mag16 mag256 md1xMono md2k
md50Eco md50Mono md5k mgr4 mgr8 mgrgray2 mgrgray4 mgrgray8 mgrmono miff24
mj500c mj6000c mj700v2c mj8000c ml600 necp6 npdl nullpage oce9050 oki182
oki4w okiibm omni oprp opvp paintjet pam pbm pbmraw pcl3 pcx16 pcx24b
pcx256 pcx2up pcxcmyk pcxgray pcxmono pdfwrite pgm pgmraw pgnm pgnmraw
photoex picty180 pj pjetxl pjxl pjx1300 pkm pkmlraw pksm pksmraw plan9bm
png16 png16m png16m png256 png256 pngalpha pngalpha pnggray pnggray
pngmono pngmono pnmm raw ppm ppmraw pr1000 pr1000_4 pr150 pr201 psdcmyk
```

```

psdrgb psgray psmono psrgb pswrite pxlcolor pxlmono r4081 rpd1 samsunggdi
sgirgb sj48 spotcmyk st800 stcolor sunhmono t4693d2 t4693d4 t4693d8
tek4696 tiff12nc tiff24nc tiffcrle tiffg3 tiffg32d tiffg4 tifflzw
tiffpack uniprint x11 x11alpha x11cmyk x11cmyk2 x11cmyk4 x11cmyk8
x11gray2 x11gray4 x11mono xcf xes
[...]

```

That's a pretty impressive collection of output devices!

Note that the *win.metafile* device requires the MS/Windows libraries. On a GNU/Linux system, to generate the MS/Windows Metafile format, you will need to install and run a version of R under, for example, the GNU/Linux *wine* package. This was discussed in Section 13.1.3, page 165.

The graphics included in this book are generated as **pdf** using scalable vector graphics. We could have generated **png** files for inclusion in web pages or even convert **pdf** files to **png** using the Debian GNU/Linux **convert** program:

```
$ convert rplot-basic.pdf rplot-basic.png
```

15.12.4 Multiple Plots

On any device, a sequence of graphics may be produced. For a screen device, for example, each call of the *plot* function will effectively wipe the active screen device and start drawing a new graphic. Multiple plots to the same device, without wiping the screen, can be achieved by setting the *new* option to true for each *plot*:

```

> plot(a)
> par(new=TRUE)
> plot(b)
> par(new=TRUE)
> plot(c)

```

On a MS/Windows device there is the opportunity to interactively cycle through the sequence of graphics, after turning on the Recording option of the History menu of the graphics Window. You can then use the Page Up and Page Down keys (or the menu) to cycle through the sequence of graphics. This is not available on the X11 screen device under Unix and GNU/Linux.

When displaying on a file device, multiple plots equate to multiple pages. Some file devices, such as *postscript* and *pdf* support multiple pages. Many print devices support the production of multiple files for multiple plots, through setting `onelife=FALSE` and the file name to something like `file="plot%03d"`.

15.12.5 Copy and Print Devices

From a MS/Windows screen device you can choose from the right mouse button menu to Copy the graphics to the Clipboard as a MS/Metafile or Bitmap format, or else to save the graphics to a file as a Metafile or PostScript file. You can also directly print the graphic from the same menu. This functionality is not available on the X11 screen device under Unix and GNU/Linux.

Having interactively generated a graphic you can copy the graphic to any other device with the *dev.copy* function. R accomplishes this by keeping a so called *display list* which tracks the graphics operations used to draw the graphic. When a copy is requested, or the graphics needs to be redrawn, these graphics commands are executed.

```
> dev.set(3)
> dev.copy(pdf, file="currentplot.pdf")
> dev.off()
```

The recording of the graphics operations is enabled only for screen devices. The recording to the display list can be turned off, if memory is at a premium:

```
> dev.control("inhibit")
```

15.13 Graphics Parameters

The *par* function in R applies various options to the current graphics device. It can not be set until the device has been created. It is used to change or add to the appearance of a plot, or to obtain information about a plot. For example, the *usr* option returns the coordinates of the plotting region:

```
> plot(rnorm(50))
> par("usr")
[1] -0.960000 51.960000 -3.245599 2.852000
```

This is useful when you want to place additional objects on the plot—it will tell you the extent of the coordinates currently in use.

When using the *lattice* package note that instead of *par* we use *trellis.par.get* and *trellis.par*, as in:

```
> library(lattice)
> mt <- trellis.par.get("par.main.text")
> mt$cex <- 0.7
> trellis.par.set("par.main.text", mt)
```

The output of *trellis.par.get* without arguments lists all the possible options. Calling *show.settings* can also help.

15.13.1 Plotting Region

The *xpd* specifies whether anything plotted outside the plotting region should be displayed (default is FALSE).

```
par("usr")           # Returns coordinates of plot region.
par(usr=c(0,600,0,800)) # Set the plot region.
par(xpd=TRUE)        # Do not clip to the plot region.
```

15.13.2 Locating Points on a Plot

You may sometimes want to know specific locations on a plot, for example to place some text at specific points. To find the locations the interactive function *locator* is most useful. Execute the function and then left mouse click on the plot at the points you wish to locate, followed by a right mouse click to finish. A list of x and y points will be displayed in the R window.

15.13.3 Scientific Notation and Plots

By default, if the numbers labelling the axes of a plot will end up taking more digits than the scientific notation, then scientific notation will be

used. You can add a penalty to the determination of when scientific notation will be used by using the *scipen* option. Setting to a large positive number will ensure scientific notation will not be used, while setting it to a large negative number will ensure scientific notation will always be used:

```
> options(scipen=99)
```

Togaware Watermark For Data Mining Survival

Togaware Watermark For Data Mining Survival

Chapter 16

Understanding Data

A key task in any data mining project is **exploratory data analysis** (often abbreviated as EDA). This task generally involves getting the basic statistics of a dataset and using graphical tools to visually investigate the data's characteristics. Visual data exploration can help in understanding the data, in error correction, and in variable selection and variable transformation.



Statistics is the fundamental tool in understanding data. Statistics is essentially about uncertainty—to understand and thereby to make allowance for it. It also provides a framework for understanding the discoveries made in data mining. Discoveries need to be statistically sound and statistically significant—any uncertainty associated with the modelling needs to be understood.

Visualising data has been an area of study within statistics for many years. A vast array of tools are available for presenting data visually. The whole topic deserves a book in its own right, and indeed there are many, including [Cleveland \(1993\)](#) and Tufte.

In this chapter we introduce some of the basic statistical concepts that a data miner needs to know. We then provide a gallery of graphical approaches to visualise and understand our data. Many of the plots

we present here could have just as easily, or perhaps initially even more easily, been produced using a spreadsheet application. However there are significant advantages in programmatically generating the plots. There could be tens, or even hundreds, of plots you would like to generate. Doing this by hand in a spreadsheet is cumbersome and error prone. Also, any plots produced from the first data extraction are just the start. As the data is refined and new datasets generated, manually regenerating plots is not a productive exercise. Using R to extract and manipulate the data and to plot the data is a cost effective exercise, using open source software (on either GNU/Linux or MSWindows platforms).

After loading data, as discussed in Chapter 3, we can start our exploration of the data itself. In addition to textual summaries, building on the basic graphics capabilities introduced in Section 15, page 235, we provide an overview of R’s extensive graphics capabilities for exploring and understanding the data. Section 16.1 explores the basic characteristics of a dataset, while Section 16.8 begins to provide basic statistical summaries of the data.

16.1 Single Variable Overviews

16.1.1 Textual Summaries

We saw in Chapter ?? some of the R functions that help us get a basic picture of the scope and type of data in any dataset. These include the most basic of information including the number and names of columns and rows (for data frames) and a summary of the data values themselves. We illustrate this again with the *wine* dataset (see Section 14.3.4, page 219):

```
> load("wine.RData")
> dim(wine)
[1] 178   14
> nrow(wine)
[1] 178
> ncol(wine)
[1] 14
> colnames(wine)
[1] "Type"          "Alcohol"        "Malic"          "Ash"
[5] "Alcalinity"     "Magnesium"      "Phenols"        "Flavanoids"
[9] "Nonflavanoids" "Proanthocyanins" "Color"         "Hue"
[13] "Dilution"       "Proline"
```

```
> rownames(wine)
 [1] "1"   "2"   "3"   "4"   "5"   "6"   "7"   "8"   "9"   "10"  "11"
[12]
[13] "13"  "14"  "15"  "16"  "17"  "18"  "19"  "20"  "21"  "22"  "23"
[24]
[...]
[157] "157" "158" "159" "160" "161" "162" "163" "164" "165" "166" "167" "168"
[169] "169" "170" "171" "172" "173" "174" "175" "176" "177" "178"
```

This gives us an idea of the shape of the data. We are dealing with a relatively small dataset of 178 entities and 14 variables.

Next, we'd like to see what the data itself looks like. We can list the first few rows of the data using *head*:

```
> head(wine)
  Type Alcohol Malic  Ash Alkalinity Magnesium Phenols Flavanoids Nonflavanoids
1    1     14.23  1.71 2.43          15.6       127     2.80      3.06
0.28
2    1     13.20  1.78 2.14          11.2       100     2.65      2.76
0.26
3    1     13.16  2.36 2.67          18.6       101     2.80      3.24
0.30
4    1     14.37  1.95 2.50          16.8       113     3.85      3.49
0.24
5    1     13.24  2.59 2.87          21.0       118     2.80      2.69
0.39
6    1     14.20  1.76 2.45          15.2       112     3.27      3.39
0.34
  Proanthocyanins Color  Hue Dilution Proline
1            2.29  5.64 1.04      3.92     1065
2            1.28  4.38 1.05      3.40     1050
3            2.81  5.68 1.03      3.17     1185
4            2.18  7.80 0.86      3.45     1480
5            1.82  4.32 1.04      2.93      735
6            1.97  6.75 1.05      2.85     1450
```

Next we might look at the structure of the data using the *str* (structure) function. This provides a basic overview of both values and their data type:

```
> str(wine)
'data.frame': 178 obs. of 14 variables:
 $ Type        : Factor w/ 3 levels "1","2","3": 1 1 1 1 1 1 1 1 1 ...
 $ Alcohol     : num  14.2 13.2 13.2 14.4 13.2 ...
 $ Malic       : num  1.71 1.78 2.36 1.95 2.59 1.76 1.87 2.15 1.64 1.35 ...
 $ Ash         : num  2.43 2.14 2.67 2.5 2.87 2.45 2.45 2.61 2.17 2.27 ...
 $ Alkalinity  : num  15.6 11.2 18.6 16.8 21 15.2 14.6 17.6 14 16 ...
 $ Magnesium   : int  127 100 101 113 118 112 96 121 97 98 ...
 $ Phenols     : num  2.8 2.65 2.8 3.85 2.8 3.27 2.5 2.6 2.8 2.98 ...
 $ Flavanoids  : num  3.06 2.76 3.24 3.49 2.69 3.39 2.52 2.51 2.98 3.15 ...
 $ Nonflavanoids: num  0.28 0.26 0.3 0.24 0.39 0.34 0.3 0.31 0.29 0.22 ...
```

```
$ Proanthocyanins: num  2.29 1.28 2.81 2.18 1.82 1.97 1.98 1.25 1.98 1.85 ...
$ Color          : num  5.64 4.38 5.68 7.8 4.32 6.75 5.25 5.05 5.2 7.22 ...
$ Hue            : num  1.04 1.05 1.03 0.86 1.04 1.05 1.02 1.06 1.08 1.01 ...
$ Dilution       : num  3.92 3.4 3.17 3.45 2.93 2.85 3.58 3.58 2.85 3.55 ...
$ Proline        : int   1065 1050 1185 1480 735 1450 1290 1295 1045 1045 ...
```

We are now starting to get an idea of what the data itself looks like. The categorical variable `Type` would appear to be something that we might want to model—the output variable. The remaining variables are all numeric variables, a mixture of integers and real numbers.

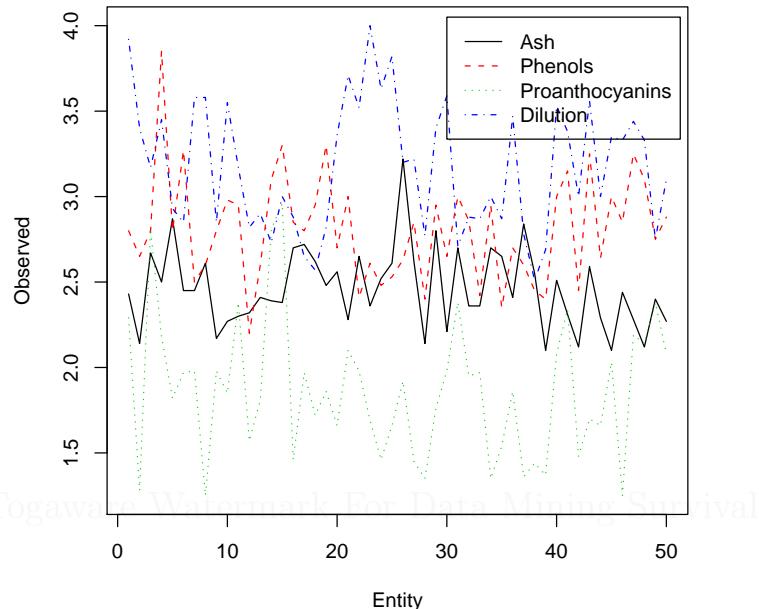
The final step in the first look at the data is to get a summary of each variable using *summary*:

```
> summary(wine)
      Type      Alcohol       Malic       Ash      Alcalinity 
  1:59  Min.   :11.03   Min.   :0.740   Min.   :1.360   Min.   :10.60 
  2:71  1st Qu.:12.36  1st Qu.:1.603  1st Qu.:2.210  1st Qu.:17.20 
  3:48  Median :13.05  Median :1.865  Median :2.360  Median :19.50 
        Mean   :13.00  Mean   :2.336  Mean   :2.367  Mean   :19.49 
        3rd Qu.:13.68  3rd Qu.:3.083  3rd Qu.:2.558  3rd Qu.:21.50 
        Max.   :14.83  Max.   :5.800  Max.   :3.230  Max.   :30.00 
[...]
```

16.1.2 Multiple Line Plots

A line plot displays a line corresponding to one or more variables over some series of data or entities. The *matplot* function will display multiple lines from data in a matrix. such a plot is useful in observing changes in variables over time or across entities.

In our example we plot just the first 50 entities in the *wine* dataset, and choose four variables to plot. These kinds of plots can get very crowded if we attempt to plot too much data. A legend is placed in an appropriate location (at the point $x = 33, y = 4.05$) using the *legend* function, where we also identify the colours, using *col*, to be the first four in the colour pallete, and the line types, using *lty*, as the firsts four line types.

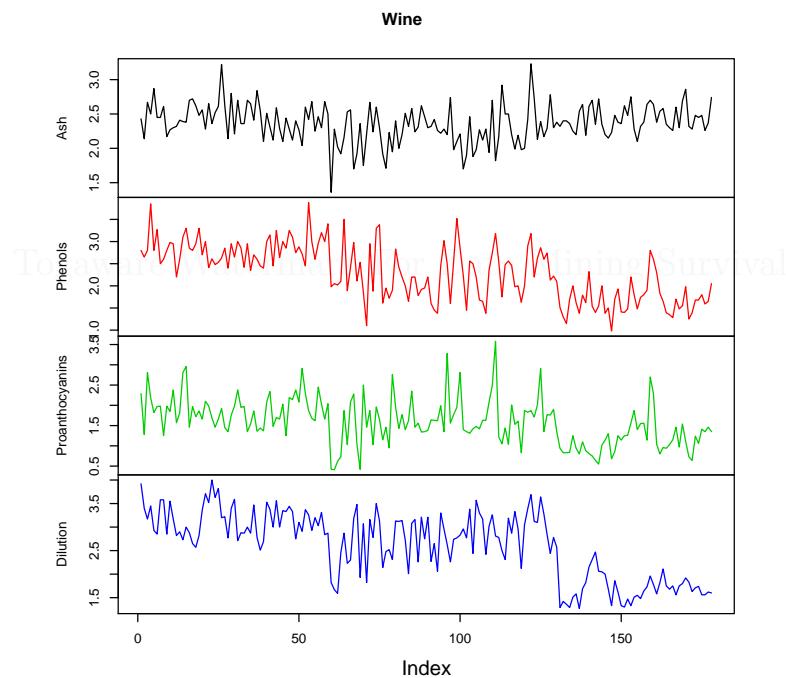


```
load("wine.Rdata")
cols <- c("Ash", "Phenols", "Proanthocyanins", "Dilution")
matplot(1:50, wine[1:50,cols], type="l", ylab="Observed", xlab="Entity")
legend(30, 4.05, cols, col=1:4, lty=1:4)
```

R code source: [rplot-wine-matplot.R](#).

16.1.3 Separate Line Plots

The multiple line plots can be separated into their own plots, stacked vertically, to give a clearer view of the individual plots. We can use the *zoo* package to plot sequences. These plots are especially useful for data that is sequence oriented, such as time series plots, but even here we can get some insights into the data.



```
library(zoo)
load("wine.Rdata")
cols <- c("Ash", "Phenols", "Proanthocyanins", "Dilution")
zdat <- zoo(wine[,cols], 1:nrow(wine))
plot(zdat, main="Wine", col=1:4)
```

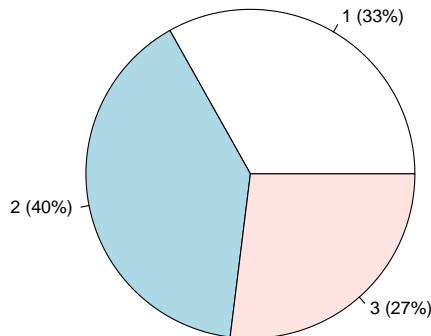
R code source: [rplot-wine-zoo.R](#).

16.1.4 Pie Chart

A **pie chart** partitions a circle into proportions related to some data. The R function *pie* is used to produce a pie chart. A pie chart can be used to display the proportion of entities spread across some partitioning of the dataset. In French, they are referred to as *le camembert* (as in the round cheese), and in Danish, as *Lagkagediagram* (traditional layer cakes).

Pie charts are a perennial favourite even though common wisdom suggests avoiding them. The human eye is not well suited to differentiating angular variations and a bar chart provides a better alternative. However, many people still enjoy the look of a pie chart.

In our example, using the *wine* dataset, the data is partitioned on categorical variable *Type*. The default plot produced by *pie* will produce quite a respectable looking pie chart. We add in to the basic plot the percentage of entities in each category, including this with the labels of the pie chart. This helps in communicating the distribution of the data over *Type*.



```
load("wine.Rdata")
attach(wine)
percent <- round(summary(Type) * 100 / nrow(wine))
labels <- sprintf("%s (%d%)", levels(Type), percent)
pie(summary(Type), lab=labels)
```

R code source: [rplot-wine-pie.R](#).

Togaware Watermark For Data Mining Survival

16.1.5 Fan Plot

```

fan.plot<-function(x,edges=200,radius=1,col=NULL,centerpos=pi/2,
                     labels=NULL,...) {
  if (!is.numeric(x) || any(is.na(x) | x<=0))
    stop("fan.plot: x values must be positive.")
  # scale the values to a half circle
  x<-pi*x/sum(x)
  xorder<-order(x,decreasing=TRUE)
  nx <- length(x)
  if (is.null(col)) col<-rainbow(nx)
  else if(length(col) < nx) col<-rep(col,nx)
  oldpar<-par(no.readonly=TRUE)
  par(mar=c(0,0,4,0))
  plot(0,xlim=c(-1,1),ylim=c(-0.6,1),xlab="",ylab="",type="n",axes=FALSE)
  lside<-0.8
  for(i in 1:nx) {
    n<-edges*x[xorder[i]]/pi
    t2p<-seq(centerpos-x[xorder[i]],centerpos+x[xorder[i]],length=n)
    xc<-c(cos(t2p)*radius,0)
    yc<-c(sin(t2p)*radius,0)
    polygon(xc,yc,col=col[xorder[i]],...)
    if(!is.null(labels)) {
      xpos<-lside*sin(x[xorder[i]])*radius
      ypos<-i/10
      text(xpos,ypos,labels[xorder[i]])
      ytop<-cos(x[xorder[i]])*radius*radius
      segments(xpos,ypos+1/20,xpos,ytop)
      lside<-lside
    }
    radius<-radius-0.02
  }
}
fan.plot(c(20,38,3,17))

```

16.1.6 Stem and Leaf Plots

A **Stem-and-leaf** plot is a simple textual plot of numeric data that is useful to get an idea of the shape of a distribution. It is similar to the graphic histograms that we will see next, but a useful quick place to start for smaller datasets. A stem-and-leaf plot has the advantage of showing actual data values in the plot rather than just a bar indicating frequency.

In reviewing a stem-and-leaf plot we might look to see if there is a clear central value, or whether the data is very spread out. We look at the spread to see if it might be symmetric about the central value or whether

there is a skew in one particular direction. We might also look for any data values that are a long way from the general values in the rest of the population.

```
> stem(wine$Magnesium)

The decimal point is 1 digit(s) to the right of the |

 7 | 0
 7 | 888
 8 | 0000012444
 8 | 55555566666666667778888888888899999
 9 | 0000112222233444444
 9 | 555666666667777788888888889
10 | 000111111111222222233333444
10 | 55666677778888
11 | 00011122222233
11 | 5566678889
12 | 0001234
12 | 678
13 | 24
13 | 69
14 |
14 |
15 | 1
15 |
16 | 2
```

The stem is to the left of the bar and the leaves are to the right.

Note the change in where the decimal point is.

```
> stem(wine$Alcohol)

The decimal point is 1 digit(s) to the left of the |

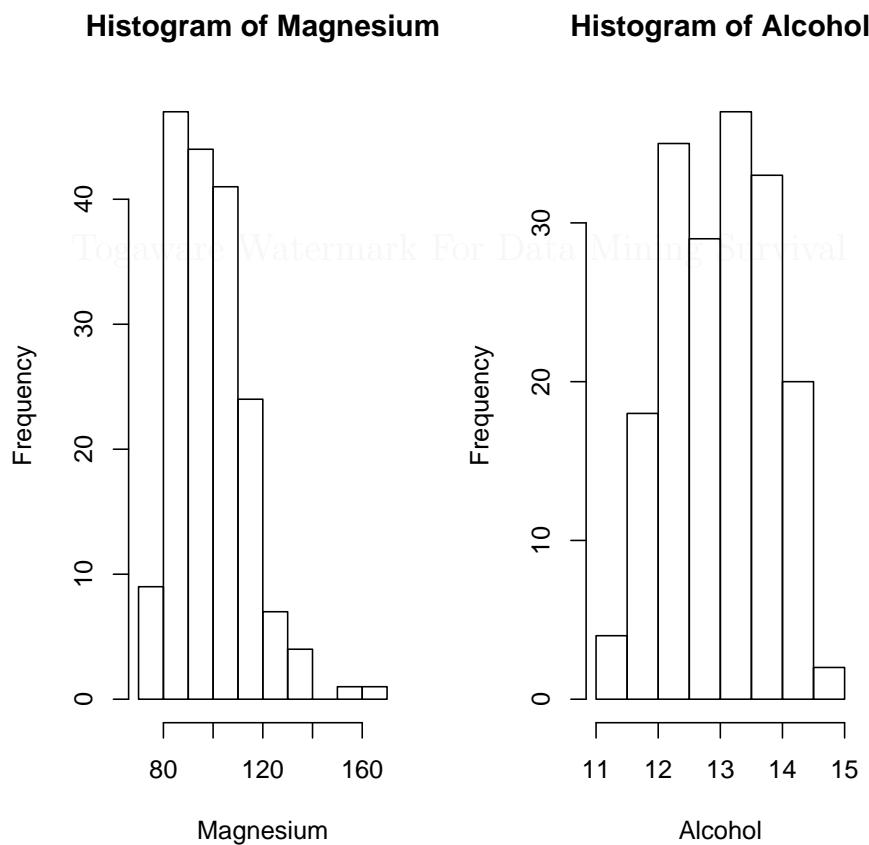
110 | 3
112 |
114 | 1566
116 | 1245669
118 | 1224476
120 | 000478888867
122 | 01255599993346777777
124 | 22235711238
126 | 004790022779
128 | 124556783369
130 | 355555578116677
132 | 034478902469
134 | 0015889900126688
136 | 2347891123345678
138 | 23346678804
140 | 266002369
142 | 01223047889
144 |
```

146 | 5
148 | 3

Togaware Watermark For Data Mining Survival

16.1.7 Histogram

A histogram allows the basic distribution of the data to be viewed. Here we plot the histogram for magnesium and alcohol content of various wines, and we might compare it with the previous stem-and-leaf plot which summarises the same data. The shape is basically the same, although in detail they go up and down at different points!



```
attach(wine)
par(mfrow=c(1, 2))
hist(Magnesium)
hist(Alcohol)
```

R code source: [rplot-wine-hist.R](#).

Also, from Rattle, we have:

```
library(rattle)
data(audit)
hs <- hist(audit$Income, main="", xlab="", col=rainbow(10))
dens <- density(audit$Income)
rs <- max(hs$counts)/max(dens$y)
lines(dens$x, dens$y*rs, type="l")
rug(audit$Income)
title(main="Distribution of Income",
      sub=paste("Rattle", Sys.time(), Sys.info()["user"]))
```

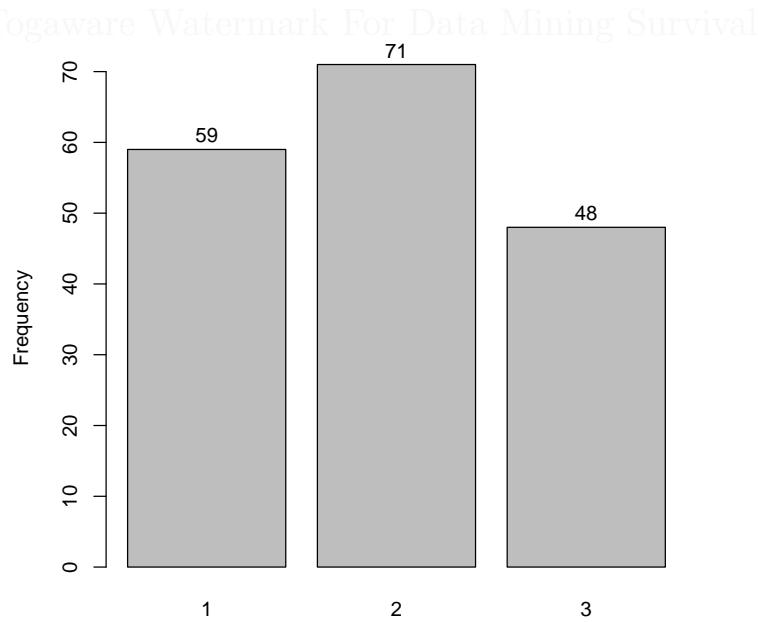
R code source: [rattle-audit-explore-distr-hist-income.R](#).

Togaware Watermark For Data Mining Survival

16.1.8 Barplot

A barplot displays data as bars, each bar being proportional to the data being plotted. In R a barplot is built using the *barplot* function. We can use a barplot, for example, to illustrate the distribution of entities in a dataset across some variable. With the *wine* dataset *Type* is a categorical variable with three levels: 1, 2, and 3. A simple bar plot illustrates the distribution of the entities across the three *Type*s. The *summary* function is used to obtain the data we wish to plot (59, 71, and 48).

We place the actual counts on the plot with the *text* function. The trick here is that the *barplot* function returns the bar midpoints, and these can be used to place the actual values. We add 2 to the *y* values to place the numbers above the bars. Also note that *xpd* is set to **TRUE** to avoid the highest number being chopped (because it, 71, is actually outside the plot region).

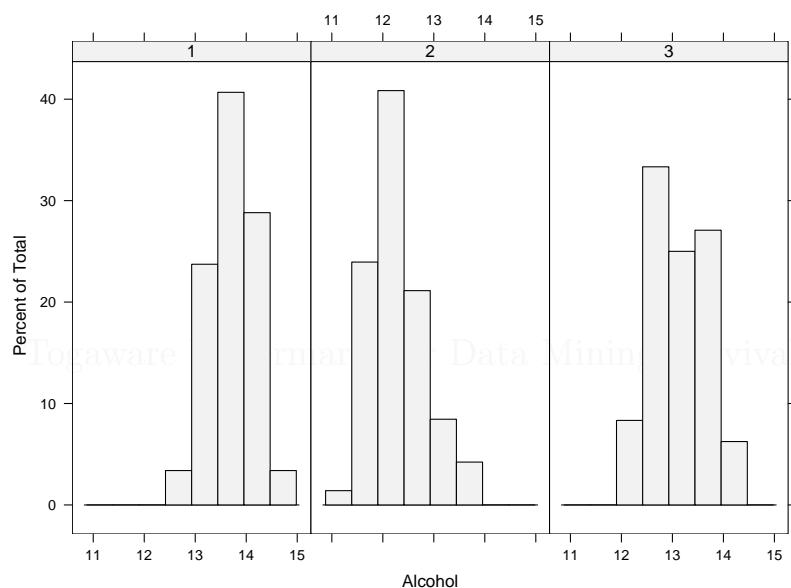


```
load("wine.Rdata")
attach(wine)
par(xpd=TRUE)
bp <- barplot(summary(Type), xlab="Type", ylab="Frequency")
text(bp, summary(Type)+2, summary(Type))
```

R code source: [rplot-wine-barplot.R](#).

16.1.9 Trellis Histogram

Multiple plots can be placed into a single plot using the *lattice* package. Here we also illustrate the use of the *color* option of the *histogram* function, setting it to FALSE to obtain a transparent background.



```
library(lattice)
load("wine.Rdata")
trellis.device(width=7, height=5, new=FALSE, color=FALSE)
with(wine, histogram(~ Alcohol | Type))
```

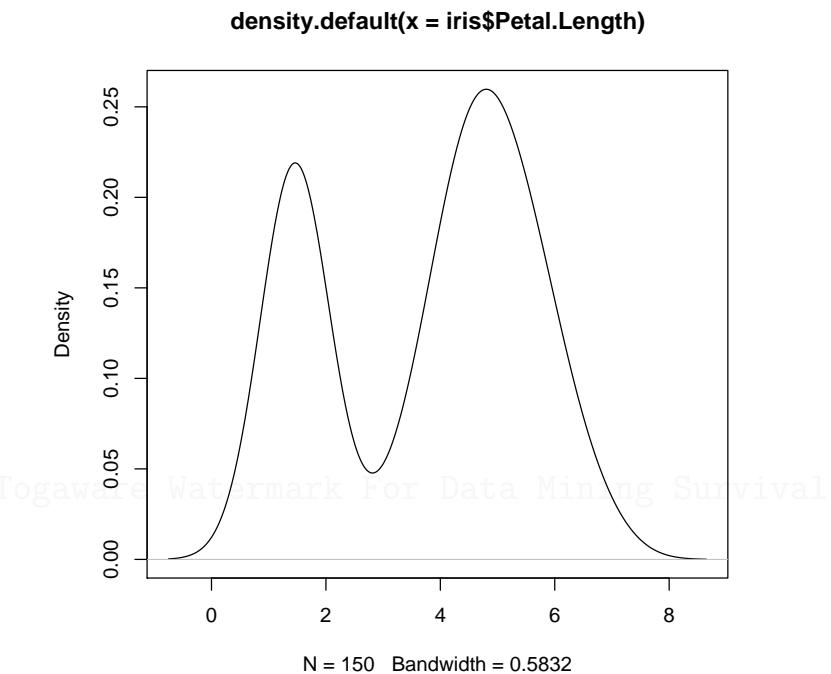
R code source: [rplot-histogram-trellis.R](#).

16.1.10 Histogram Uneven Distribution

Suppose your data has a number of outliers. The *breaks* option of *hist* allows you to specify where the splits occur.

Togaware Watermark For Data Mining Survival

16.1.11 Density Plot



```
plot(density(iris$Petal.Length))
```

R code source: [rplot-iris-density.R](#).

Here's an example that illustrates uniformity. The histogram shows a lot of variance in the uniform random sample, at least for small samples, whereas the quantile plots are more effective in showing the uniformity (or density).

```
> hist(runif(100))
> hist(runif(1000))
> hist(runif(10000))
> hist(runif(100000))
> hist(runif(1000000))
> hist(runif(10000000))
> hist(runif(100000000))

> par(mfrow=c(2,2))
> for(i in c(10, 100, 1000, 10000)) {
  qqplot(runif(i), qunif(seq(1/i, 1, length=i)), main=i,
         xlim=c(0,1), ylim=c(0,1),
         xlab="runif", ylab="Uniform distribution quantiles")
```

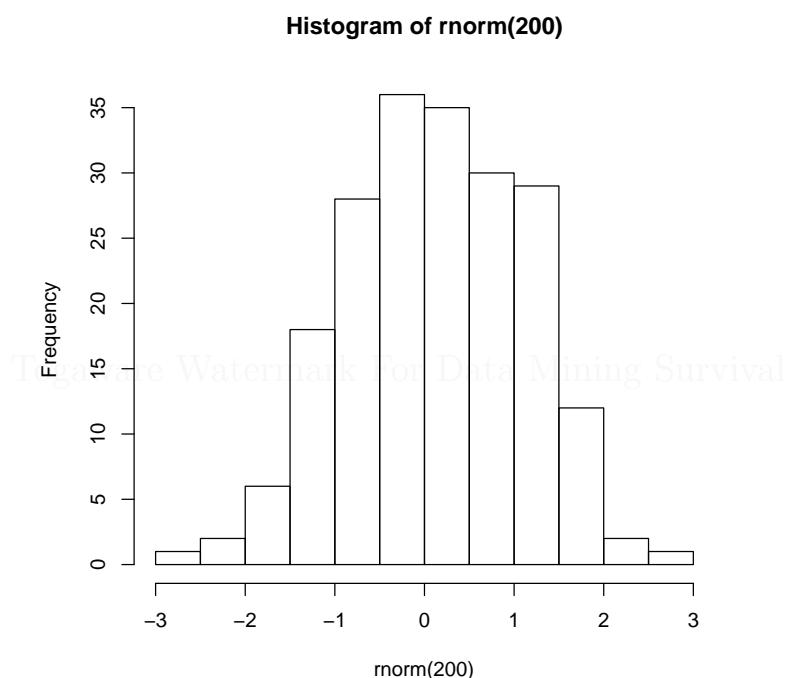
```
    abline(0,1,col="lightgray")
}
```

Histograms are not particularly good as density estimators. However, most of the time histograms are used as an exploratory tool useful in assisting in understanding our data. Using small bin widths helps find unexpected gaps and patterns in our data, and gives an initial view of the distribution.

Togaware Watermark For Data Mining Survival

16.1.12 Basic Histogram

A histogram illustrates the distribution of values. The following example is the most basic of histograms.

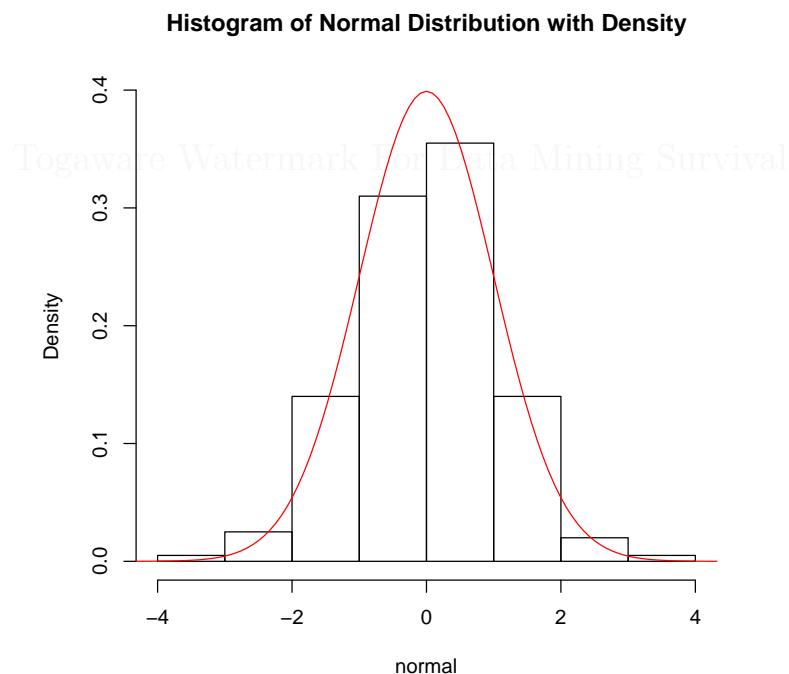


```
pdf("graphics/rplot-hist.pdf")
hist(rnorm(200))
dev.off()
```

R code source: [rplot-hist.R](#).

16.1.13 Basic Histogram with Density Curve

R allows plots to be built up—this example shows a density histogram of a set of random numbers extracted from a normal distribution with the density curve of the same normal distribution also displayed. In the R code we build the histogram at first without plotting it, so as to determine the y limits (*range* selects the minimum and maximum values, while *h\$density* is the list of density values being plotted and *dnorm(0)* is the maximum possible value of the density), since otherwise the curve might push up into the title!



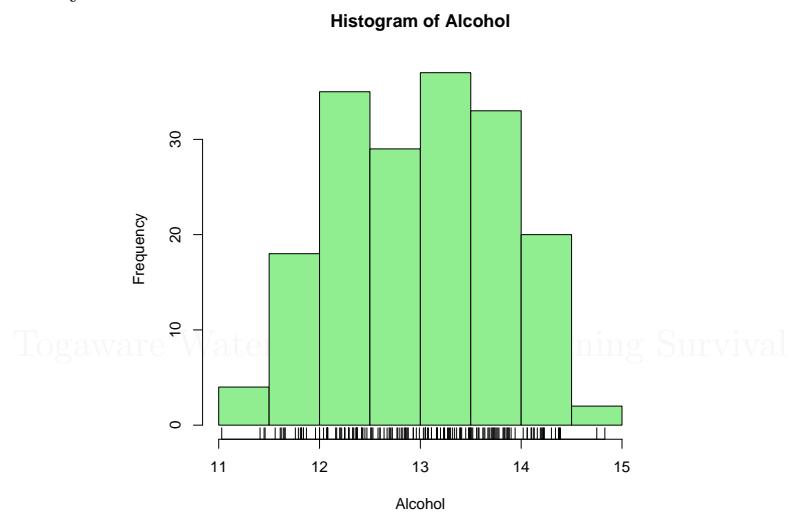
```
ds <- rnorm(200)
pdf("graphics/rplot-hist-density.pdf")
par(xpd=T)
h <- hist(ds, plot=F)
ylim <- range(0, h$density, dnorm(0))
hist(ds, xlab="normal", ylim=ylim, freq=F,
      main="Histogram of Normal Distribution with Density")
curve(dnorm, col=2, add=T)
dev.off()
```

R code source: [rplot-hist-density.R](#).

Togaware Watermark For Data Mining Survival

16.1.14 Practical Histogram

Suppose we are interested in the distribution of the Alcohol content in the *wine* dataset. The numeric values are grouped by *hist* into intervals and the bars represent the frequency of occurrence of each interval as a height. A *rug* is added to the plot, just above the x-axis, to illustrate the density of values.



```
pdf('graphics/rplot-hist-colour.pdf')
load("wine.Rdata")
attach(wine)
hist(Alcohol, col='lightgreen')
rug(Alcohol)
dev.off()
```

R code source: [rplot-hist-colour.R](#).

16.2 Multiple Variable Overviews

16.2.1 Pivot Tables

The *reshape* package was inspired by pivot tables. The package works on homogeneous data only, so your data needs to be all numeric or all character, and not a mixture of the.

The aim is to generate various aggregate summaries of the data. For example, with the *wine* dataset we may like to look at the average values of a number of input variables for each Type. The first step, using the *reshape* package, is to *melt* the data frame, which expands the non-identifying variables across the identifying variables:

```
> wine.molten <- melt(wine, id="Type")
> head(wine.molten)
  Type variable value
1     1   Alcohol 14.23
2     1   Alcohol 13.20
3     1   Alcohol 13.16
4     1   Alcohol 14.37
5     1   Alcohol 13.24
6     1   Alcohol 14.20
> tail(wine.molten)
  Type variable value
17312     3 Proline  660
17412     3 Proline  740
17512     3 Proline  750
17612     3 Proline  835
17712     3 Proline  840
17812     3 Proline  560
```

Now we can use *cast* to recast the data into the shape we want. Here we reshape it by Type and list the *mean* of each input variable across the values of Type:

```
> cast(wine.molten, Type ~ variable, mean)

  Type Alcalinity Alcohol      Ash    Color Dilution Flavanoids
Hue
1     1 17.03729 13.74475 2.455593 5.528305 3.157797 2.9823729 1.0620339
2     2 20.23803 12.27873 2.244789 3.086620 2.785352 2.0808451 1.0562817
3     3 21.41667 13.15375 2.437083 7.396250 1.683542 0.7814583 0.6827083

  Magnesium    Malic Nonflavanoids Phenols Proanthocyanins    Proline
106.3390 2.010678       0.290000 2.840169       1.899322 1115.7119
  94.5493 1.932676       0.363662 2.258873       1.630282 519.5070
  99.3125 3.333750       0.447500 1.678750       1.153542 629.8958
```

We can also include the column and row totals. We will illustrate this with a subset of the *wine* dataset:

```
> measure <- c("Alcohol", "Malic", "Ash")
> wine.molten <- melt(wine, id="Type", measure=measure)
> cast(wine.molten, Type ~ variable, mean, margins=c("grand_row", "grand_col"))
   Type Alcohol Malic Ash .
1 13.74475 2.010678 2.455593 6.070339
2 12.27873 1.932676 2.244789 5.485399
3 13.15375 3.333750 2.437083 6.308194
. 13.00062 2.336348 2.366517 5.901161
```

In this case the row totals have no meaning but the column totals do.

Also see *aggregate*:

```
> aggregate(wine[,-1], list(Type=wine$Type), mean)
   Type Alcohol Malic Ash Alkalinity Magnesium Phenols Flavanoids
1    1 13.74475 2.010678 2.455593 17.03729 106.3390 2.840169 2.9823729
2    2 12.27873 1.932676 2.244789 20.23803 94.5493 2.258873 2.0808451
3    3 13.15375 3.333750 2.437083 21.41667 99.3125 1.678750 0.7814583
   Nonflavanoids Proanthocyanins Color Hue Dilution Proline
1      0.290000      1.899322 5.528305 1.0620339 3.157797 1115.7119
2      0.363662      1.630282 3.086620 1.0562817 2.785352 519.5070
3      0.447500      1.153542 7.396250 0.6827083 1.683542 629.8958
```

Another example using reshape.

```
> dat <- read.table("clipboard", header=TRUE)
> dat
   Q S C
1 1 A 5
2 1 B 10
3 1 C 50
4 1 D 10
5 2 A 20
6 2 E 10
7 2 C 40
8 3 D 5
9 3 F 1
10 3 G 5
11 3 B 75
> res <- reshape(dat, direction = "wide", idvar = "Q", timevar = "S")
> res
   Q C.A C.B C.C C.D C.E C.F C.G
1 1 5 10 50 10 NA NA NA
5 2 20 NA 40 NA 10 NA NA
8 3 NA 75 NA 5 NA 1 5
> res[is.na(res)] <- 0
> names(
> res
   Q C.A C.B C.C C.D C.E C.F C.G
1 1 5 10 50 10 0 0 0
5 2 20 0 40 0 10 0 0
```

```
8 3 0 75 0 5 0 1 5
```

Or the same, but using the reshape package:

```
> library(reshape)
> datm <- melt(dat, id=1:2)
> cast(datm, Q ~ S)

S   A   B   C   D   E   F   G
Q   A   B   C   D   E   F   G
1   5  10  50  10  NA  NA  NA
2  20  NA  40  NA  10  NA  NA
3  NA  75  NA   5  NA   1   5
```

Togaware Watermark For Data Mining Survival

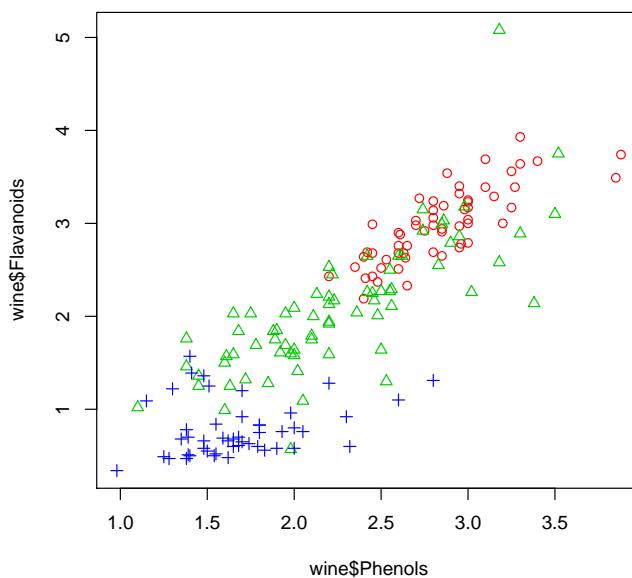
With the basics in hand we can now explore the data in a more graphical fashion, beginning with plots that help understand individual variables (barplot, piechart, and line plots), followed by a number of plots that explore relationships between variables (scatterplot and correlation plot).

16.2.2 Scatterplot

A **scatterplot** presents points in 2-dimensional space corresponding to a pair of chosen variables. R's *plot* function defaults to a scatterplot. Relationships between pairs of variables can be seen through the use of a scatterplot and clusters and outliers can begin to be identified.

Using the *wine* dataset a plot is created to display **Phenols** versus **Flavanoids**. To add a little more interest to the plot, a different symbol (and for colour devices, a different colour) is used to display the three different values of **Type** for each point. The symbols are set using **Type** as the argument to *pch*, but after converting it to integers with *as.integer*. In a similar fashion, the colours are chosen to replace numbers in a transformation of the **Type** vector by indexing into the output of *palette*, achieved using *lapply*, and turning the result into a flat list, rather than a list of lists, using *unlist*.

We can start to understand that there is somewhat of a linear relationship between these two variables, and even more interesting is the clustering of **Types**.



```
iType <- as.integer(wine$type)
colours <- unlist(lapply(iType, function(x){palette()[x+1]}))
plot(wine$Phenols, wine$Flavanoids, col=colours, pch=iType)
dev.off()
```

R code source: [rplot-wine-scatter.R](#).

Togaware Watermark For Data Mining Survival

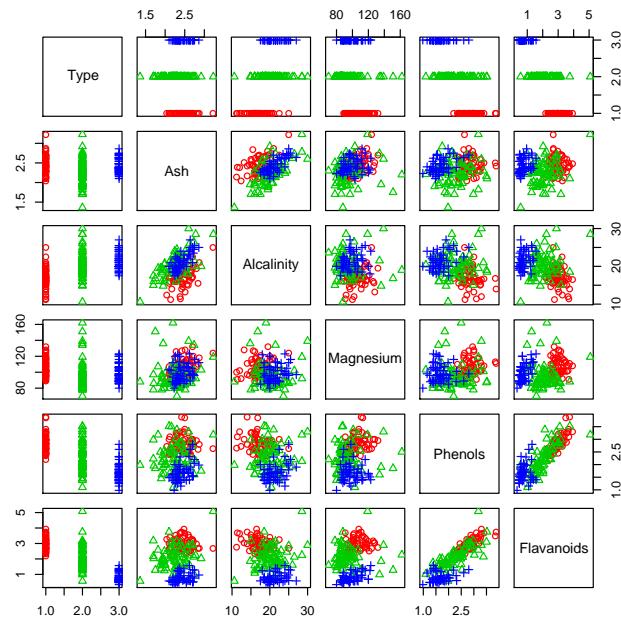
16.2.3 Scatterplot with Marginal Histograms

Togaware Watermark For Data Mining Survival

16.2.4 Multi-Dimension Scatterplot

For data with multiple dimensions, *plot* will decide on a multi-dimensional scatterplot. This produces a scatterplot for each pair of variables, with the variable names identified in the diagonal. Each plot is a scatterplot of the data for the variable in the column by the variable in the row. The upper right triangle of the scatterplot is the mirror image of the lower left triangle of the scatterplot—with the axes swapped. Although this results in repeated information, it is visually effective since it is possible to scan all plots for one variable either vertically or horizontally, rather than having to turn corners.

Once again, we use different symbols (and colour) to highlight the distribution of Type across each plot, borrowing (but not showing) the appropriate code from the scatterplot example of Section 16.2.2, page 284. The plot is also limited to just the first six variables, to avoid too much clutter. Note that the scatterplot of Section 16.2.2 is also included.



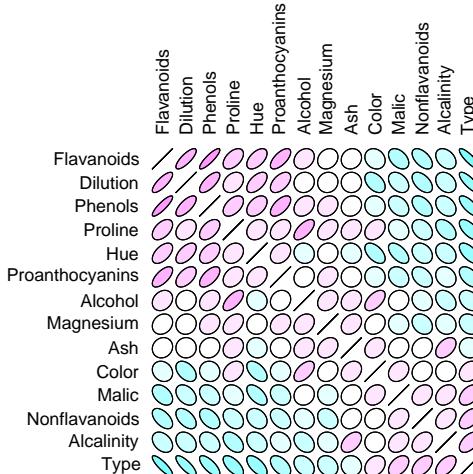
```
plot(wine[,c(1, 4:8)], col=colours, pch=iType)
```

R code source: [rplot-wine-scatterm.R](#).

16.2.5 Correlation Plot

A **correlation** measures how two variables are related and is useful for measuring the association between the two variables. A correlation plot shows the strength of any linear relationship between a pair of variables. The *ellipse* package provides the *plotcorr* function for this purpose. Linear relationships between variables indicate that as the value of one variable changes, so does the value of another. The degree of correlation is measured between $[-1, 1]$ with 1 being perfect correlation and 0 being no correlation. The Pearson correlation coefficient is the common statistic and R also supports Kendall's tau and Spearman's rho statistics for rank-based measures of association, which are regarded as being more robust and recommended other than for a bivariate normal distribution. The *cor* function is used to calculate the correlation matrix between variables in a numeric vector, matrix or data frame. A matrix is always symmetric about the diagonal, and the diagonal consists of 1s (each variable is perfectly correlated with itself!).

The sample R code here generates the correlations for variables in the *wine* dataset (*cor*) and then orders the variables according to their correlation with the first variable (Type: `[1,]`). This is sorted and ellipses are printed with colour fill using *cm.colors*.



```
library(ellipse)
wine.corr <- cor(wine)
ord <- order(wine.corr[1,])
xc <- wine.corr[ord, ord]
plotcorr(xc, col=cm.colors(11)[5*xc + 6])
```

R code source: [rplot-wine-corr.R](#).

The correlation matrix is:

```
> wine.corr
          Type      Alcohol      Malic       Ash Alcalinity
Type 1.00000000 -0.32822194 0.43777620 -0.049643221 0.51785911
Alcohol -0.32822194 1.00000000 0.09439694 0.211544596 -0.31023514
Malic 0.43777620 0.09439694 1.00000000 0.164045470 0.28850040
Ash -0.04964322 0.21154460 0.16404547 1.000000000 0.44336719
Alcalinity 0.51785911 -0.31023514 0.28850040 0.443367187 1.00000000
Magnesium -0.20917939 0.27079823 -0.05457510 0.286586691 -0.08333309
Phenols -0.71916334 0.28910112 -0.33516700 0.128979538 -0.32111332
Flavanoids -0.84749754 0.23681493 -0.41100659 0.115077279 -0.35136986
Nonflavanoids 0.48910916 -0.15592947 0.29297713 0.186230446 0.36192172
Proanthocyanins -0.49912982 0.13669791 -0.22074619 0.009651935 -0.19732684
Color 0.26566757 0.54636420 0.24898534 0.258887259 0.01873198
Hue -0.61736921 -0.07174720 -0.56129569 -0.074666889 -0.27395522
Dilution -0.78822959 0.07234319 -0.36871043 0.003911231 -0.27676855
Proline -0.63371678 0.64372004 -0.19201056 0.223626264 -0.44059693

          Magnesium      Phenols Flavanoids Nonflavanoids
Type -0.20917939 -0.71916334 -0.8474975 0.4891092
Alcohol 0.27079823 0.28910112 0.2368149 -0.1559295
Malic -0.05457510 -0.33516700 -0.4110066 0.2929771
Ash 0.28658669 0.12897954 0.1150773 0.1862304
Alcalinity -0.08333309 -0.32111332 -0.3513699 0.3619217
Magnesium 1.00000000 0.21440123 0.1957838 -0.2562940
Phenols 0.21440123 1.00000000 0.8645635 -0.4499353
Flavanoids 0.19578377 0.86456350 1.0000000 -0.5378996
Nonflavanoids -0.25629405 -0.44993530 -0.5378996 1.0000000
Proanthocyanins 0.23644061 0.61241308 0.6526918 -0.3658451
Color 0.19995001 -0.05513642 -0.1723794 0.1390570
Hue 0.05539820 0.43368134 0.5434786 -0.2626396
Dilution 0.06600394 0.69994936 0.7871939 -0.5032696
Proline 0.39335085 0.49811488 0.4941931 -0.3113852

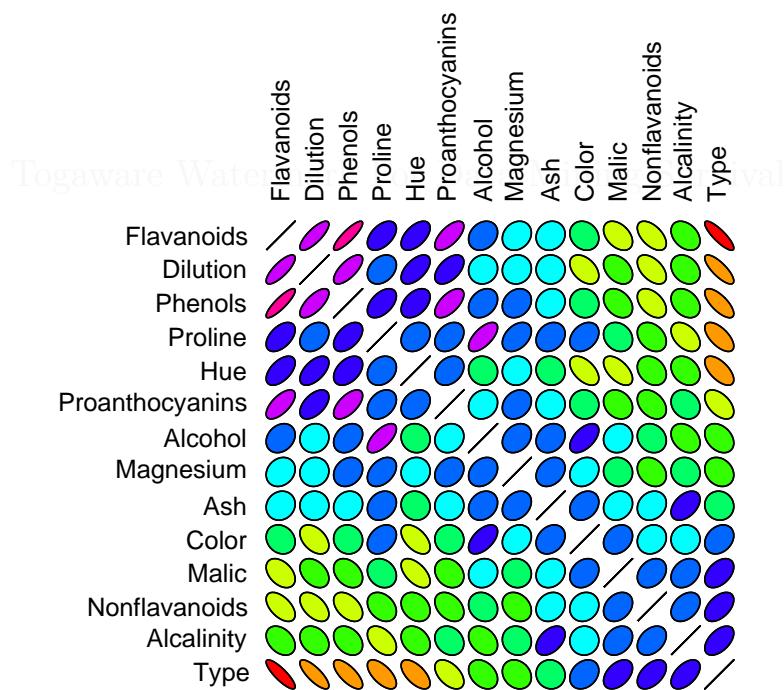
          Proanthocyanins      Color      Hue      Dilution
Proline -0.499129824 0.26566757 -0.61736921 -0.788229589 -0.6337168
Type 0.136697912 0.54636420 -0.07174720 0.072343187
Alcohol 0.6437200
Malic -0.220746187 0.24898534 -0.56129569 -0.368710428 -0.1920106
Ash 0.009651935 0.25888726 -0.074666889 0.003911231
Alcalinity 0.2236263
Magnesium 0.236440610 0.19995001 0.05539820 0.066003936
Phenols 0.612413084 -0.05513642 0.43368134 0.699949365
Proline 0.4981149
```

Flavanoids	0.652691769	-0.17237940	0.54347857	0.787193902	
0.4941931					
Nonflavanoids	-0.365845099	0.13905701	-0.26263963	-0.503269596	-0.3113852
Proanthocyanins	1.000000000	-0.02524993	0.29554425	0.519067096	
0.3304167					
Color	-0.025249931	1.000000000	-0.52181319	-0.428814942	
0.3161001					
Hue	0.295544253	-0.52181319	1.00000000	0.565468293	
0.2361834					
Dilution	0.519067096	-0.42881494	0.56546829	1.000000000	
0.3127611					
Proline	0.330416700	0.31610011	0.23618345	0.312761075	
1.0000000					

Togaware Watermark For Data Mining Survival

16.2.6 Colourful Correlations

You could write your own path.colors as below and obtain a more colourful correlation plot. The colours are quite garish but it gives an idea of what is possible—The reds and purples give a good indication of high correlation (negative and positive), while the blues and greens identify less correlation.



```

# Suggested by Duncan Murdoch
path.colors <- function(n, path=c('cyan', 'white', 'magenta'),
                        interp=c('rgb', 'hsv'))
{
  interp <- match.arg(interp)
  path <- col2rgb(path)
  nin <- ncol(path)
  if (interp == 'hsv')
  {
    path <- rgb2hsv(path)
    # Modify the interpolation so that the circular nature of hue
    for (i in 2:nin)
      path[1,i] <- path[1,i] + round(path[1,i-1]-path[1,i])
    result <- apply(path, 1, function(x) approx(seq(0, 1,
                                                len=nin), x, seq(0, 1, len=nin))$y)
    return(hsv(result[,1] %% 1, result[,2], result[,3]))
  }
  else
  {
    result <- apply(path, 1, function(x) approx(seq(0, 1,
                                                len=nin), x, seq(0, 1, len=nin))$y)
    return(rgb(result[,1]/255, result[,2]/255, result[,3]/255))
  }
}

pdf('graphics/rplot-corr-wine.pdf')
library(ellipse)
load('wine.Rdata')
corr.wine <- cor(wine)
ord <- order(corr.wine[1,])
xc <- corr.wine[ord, ord]
plotcorr(xc, col=path.colors(11,
                            c("red", "green", "blue", "red"),
                            interp="hsv")[5*xc + 6])
dev.off()

```

R code source: [rplot-corr-wine.R](#).

16.2.7 Projection Pursuit

Togaware Watermark For Data Mining Survival

16.2.8 RADVIZ

Togaware Watermark For Data Mining Survival

16.2.9 Parallel Coordinates

Togaware Watermark For Data Mining Survival

16.3 Measuring Data Distributions

We now start to explore how the data in each of the variables is distributed. This might be as simple as looking at the spread of the numeric values, or the number of entities having a specific value for a variable. Another aspect involves measuring the central tendency of data, or determining the [mean](#) and [median](#). Yet another is a measure of the spread or [variance](#) of the data from this central tendency. We again begin with textual presentations of the distributions, and then graphical presentations.

16.3.1 Textual Summaries

The [summary](#) function provides the first insight into how the values for each variable are distributed:

```
> summary(wine)

Type      Alcohol        Malic        Ash       Alcalinity
1:59    Min.   :11.03    Min.   :0.740    Min.   :1.360    Min.   :10.60
2:71    1st Qu.:12.36   1st Qu.:1.603   1st Qu.:2.210   1st Qu.:17.20
3:48    Median :13.05   Median :1.865   Median :2.360   Median :19.50
        Mean   :13.00   Mean   :2.336   Mean   :2.367   Mean   :19.49
        3rd Qu.:13.68   3rd Qu.:3.083   3rd Qu.:2.558   3rd Qu.:21.50
        Max.   :14.83   Max.   :5.800   Max.   :3.230   Max.   :30.00

Magnesium      Phenols      Flavanoids      Nonflavanoids
Min.   : 70.00    Min.   :0.980    Min.   :0.340    Min.   :0.1300
1st Qu.: 88.00   1st Qu.:1.742   1st Qu.:1.205   1st Qu.:0.2700
Median  : 98.00   Median :2.355   Median :2.135   Median :0.3400
Mean   : 99.74   Mean   :2.295   Mean   :2.029   Mean   :0.3619
3rd Qu.:107.00   3rd Qu.:2.800   3rd Qu.:2.875   3rd Qu.:0.4375
Max.   :162.00   Max.   :3.880   Max.   :5.080   Max.   :0.6600

Proanthocyanins      Color          Hue          Dilution
Min.   : 0.410    Min.   : 1.280    Min.   :0.4800    Min.   :1.270
1st Qu.: 1.250   1st Qu.: 3.220    1st Qu.:0.7825   1st Qu.:1.938
Median  : 1.555   Median : 4.690    Median :0.9650   Median :2.780
Mean   : 1.591   Mean   : 5.058    Mean   :0.9574   Mean   :2.612
3rd Qu.: 1.950   3rd Qu.: 6.200    3rd Qu.:1.1200   3rd Qu.:3.170
Max.   : 3.580   Max.   :13.000    Max.   :1.7100   Max.   :4.000

Proline
Min.   : 278.0
1st Qu.: 500.5
Median  : 673.5
Mean   : 746.9
3rd Qu.: 985.0
```

```
Max. : 1680.0
```

Next, we would like to know how the data is distributed. For categorical variables this will be how many of each level there are. For numeric variables this will be the mean and median, the minimum and maximum values, and an idea of the spread of the values of the variable.

We would also like to know about missing values (referred to in R as NAs—short for Not Available), and the *summary* function will also report this:

```
> load("survey.RData")
> summary(survey)
[...]
  Native.Country  Salary.Group
United-States:29170  <=50K:24720
Mexico       : 643   >50K : 7841
Philippines  : 198
Germany      : 137
Canada        : 121
(Other)       : 1709
NA's          : 583
```

We also see here that the categorical variable `Native.Country` has more than five levels, and there are 1,709 entities with values for this variable other than the five listed here. The five listed are the most frequently occurring.

The *mean* provides a measure of the average or central tendency of the data. It is denoted as μ if x_1, \dots, x_n is the whole population (*population mean*), and \bar{X} if it is a sample of the population (*sample mean*).

In calculating the *mean* of a sample from a population we generally need at least 30 observations in the sample before it makes sense. This is based on the central limit theorem that indicates that for $n = 30$ the shape of a distribution approaches normal.

R provides the *mean* function to calculate the mean. The mean is also reported as part of the output from *summary*. The *summary* function in fact will use the method associated with the data type of the object passed. For example, if it is a data frame the function *summary.data.frame* will be called upon. To see the actual function definition, simply type the function name at the command line (without brackets). The actual code will be printed out. A user can then fine

tune the function, if desired.

A quick trick to roughly get the mode of a dataset is to use the denisity.

```
mode <- function (n)
{
  n <- as.numeric(n)
  n.density <- density(n)
  round(n.density$x[which(n.density$y==max(n.density$y))])
}
```

You can then simply write your own functions to summarise the data using *sapply*:

```
> sapply(wine,
         function(x)
{
  x <- as.numeric(x)
  res <- c(mean(x), median(x), mode(x), mad(x), sd(x))
  names(res) <- c("mean", "median", "mode", "mad", "sd")
  res
})

      Type    Alcohol    Malic     Ash Alcalinity Magnesium Phenols
mean  1.938202 13.0006180 2.336348 2.366517 19.494944 99.74157 2.295112
median 2.000000 13.0500000 1.865000 2.360000 19.500000 98.00000 2.355000
mode   2.000000 14.0000000 2.000000 2.000000 19.000000 90.00000 3.000000
mad    1.482600 1.0081680 0.770952 0.237216 3.039330 14.82600 0.748713
sd     0.775035 0.8118265 1.117146 0.274344 3.339564 14.28248 0.625851

      Flavanoids Nonflavanoids Proanthocyanins Color       Hue
Dilution
mean   2.0292697      0.3618539      1.5908989 5.058090 0.9574494 2.6116854
median  2.1350000      0.3400000      1.5550000 4.690000 0.9650000 2.7800000
mode   3.0000000      0.0000000      1.0000000 3.000000 1.0000000 3.0000000
mad    1.2379710      0.1260210      0.5633880 2.238726 0.2446290 0.7709520
sd     0.9988587      0.1244533      0.5723589 2.318286 0.2285716 0.7099904

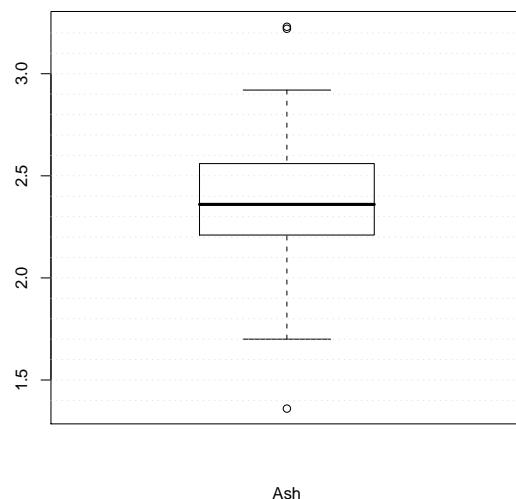
      Proline
mean   746.8933
median 673.5000
mode   553.0000
mad   300.2265
sd    314.9075
```

In the following sections we provide graphic presentations of the mean and standard variation.

16.3.2 Boxplot

A **boxplot** (Tukey, 1977) (also known as a box-and-whisker plot) provides a graphical overview of how data is distributed over the number line. R's *boxplot* function displays a graphical representation of the textual *summary* of data. The skewness of the distribution of the data becomes clear.

A boxplot shows the **median** (the second **quartile** or the 50th **percentile**) as the thicker line within the box ($Ash = 2.36$). The top and bottom extents of the box (2.558 and 2.210 respectively) identify the upper quartile (the third quartile or the 75th percentile) and the lower quartile (the first quartile and the 25th percentile). The extent of the box is known as the **interquartile range** ($2.558 - 2.210 = 0.348$). The dashed lines extend to the maximum and minimum data points that are no more than 1.5 times the interquartile range from the median. Outliers (points further than 1.5 times the interquartile range from the median) are then individually plotted (at 3.23, 3.22, and 1.36). Our plot here adds faint horizontal lines to more easily read off the various values.



```
load("wine.Rdata")
attach(wine)
boxplot(Ash, xlab="Ash")
abline(h=seq(1.4, 3.2, 0.1), col="lightgray", lty="dotted")
```

R code source: [rplot-wine-boxplot-single.R](#).

Multiple Boxplots

The default *boxplot* function in fact will plot multiple boxplots.

By comparing a number of variables we can see that some have quite a bit more spread than others, and their medians have different relative positions within the box.

We include the code here to generate a PDF version of the plot primarily to demonstrate how we can increase the width of the plot for a more pleasing presentation.

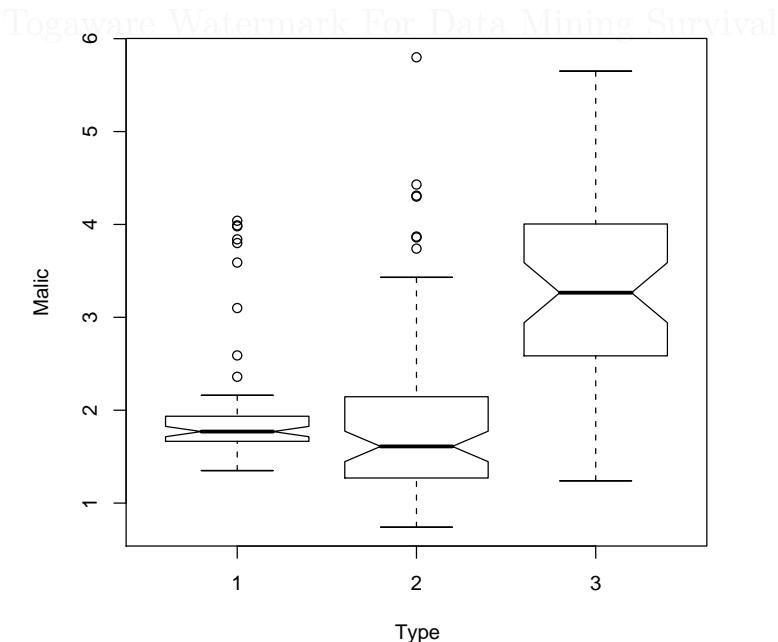
We could have presented the plot horizontally by setting the *horizontal* option to TRUE.

```
Togaware Watermark For Data Mining Survival  
pdf("graphics/rplot-wine-boxplot-multi.pdf", width=9)  
load("wine.Rdata")  
boxplot(wine[,c(3,4,7,8,10,13)])  
dev.off()
```

R code source: [rplot-wine-boxplot-multi.R](#).

Boxplot by Class

With a *boxplot* it is often useful to display the distribution of one variable as it relates to some other variable. An example in the wine data would be to partition the data according to the `Type`, and then to explore the resulting distribution of, for example, `Malic`. This is achieved with the formula notation `Malic ~ Type`. The boxplot then allows us to understand any potential relationship between the input variable and the output variable. For such plots we enable the notch display, which indicates whether there is a significant difference between the medians. In the case here the median for `Type 3` is significantly different from the other two, but the other two are not significantly different from each other.



```
load("wine.Rdata")
attach(wine)
boxplot(Malic ~ Type, notch=TRUE, xlab="Type", ylab="Malic")
```

R code source: [rplot-wine-boxplot-type.R](#).

Tuning a Boxplot

Here we illustrate how we can refine exactly what we want to draw in the box plot. Three boxplots are produced on the single plot using `par` to set `mfrow` to one row and three columns. We then collect the output from the `boxplot` function which we might look at to determine information about what is being plotted. In this case we might decide to set the limits of the boxplot to be 0 and 5.2, and we note the other statistics in the `stats` attribute of the output.

```
> boxplot.info
$stats
[ ,1]
[1,] 0.340
[2,] 1.200
[3,] 2.135
[4,] 2.880
[5,] 5.080

$n
[1] 178

$conf
[ ,1]
[1,] 1.936044
[2,] 2.333956

$out
numeric(0)

$group
numeric(0)

$names
[1] "1"
```

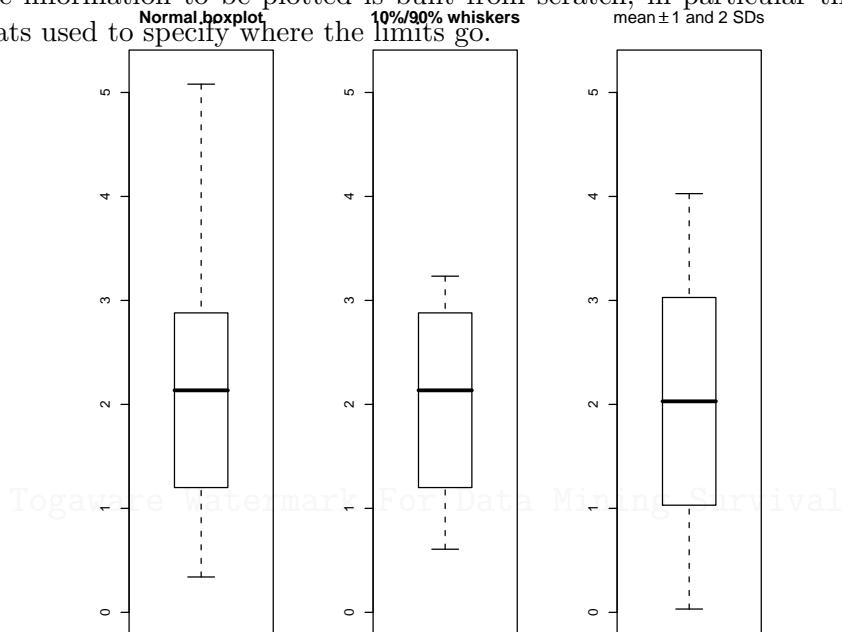
The `bxp` function (used internally by `boxplot`) is used to plot the boxplot.

We now modify the boxplot information (the `stats` attribute) to use 10% and 90% deciles (obtained using `quantile`) instead of the default 0% and 100% deciles.

```
> deciles
 0%   10%   20%   30%   40%   50%   60%   70%   80%   90%  100%
0.340 0.607 0.872 1.324 1.738 2.135 2.460 2.689 2.980 3.233 5.080
```

This generates the second boxplot.

Finally, a completely different boxplot showing the mean +/- one and two standard deviations, is produced. The structure used by *bxp* for recording the information to be plotted is built from scratch, in particular the 5 stats used to specify where the limits go.



```
oldpar <- par(mfrow=c(1,3))

x <- wine$Flavanoids
boxplot.info <- boxplot(x, plot=FALSE)
bxp(boxplot.info, main="Normal boxplot", ylim=c(0,5.2))

deciles <- quantile(x, probs=seq(0,1,0.1))
boxplot.info$stats[1] <- deciles["10%"]
boxplot.info$stats[5] <- deciles["90%"]
bxp(boxplot.info, main="10%/90% whiskers", ylim=c(0,5.2))

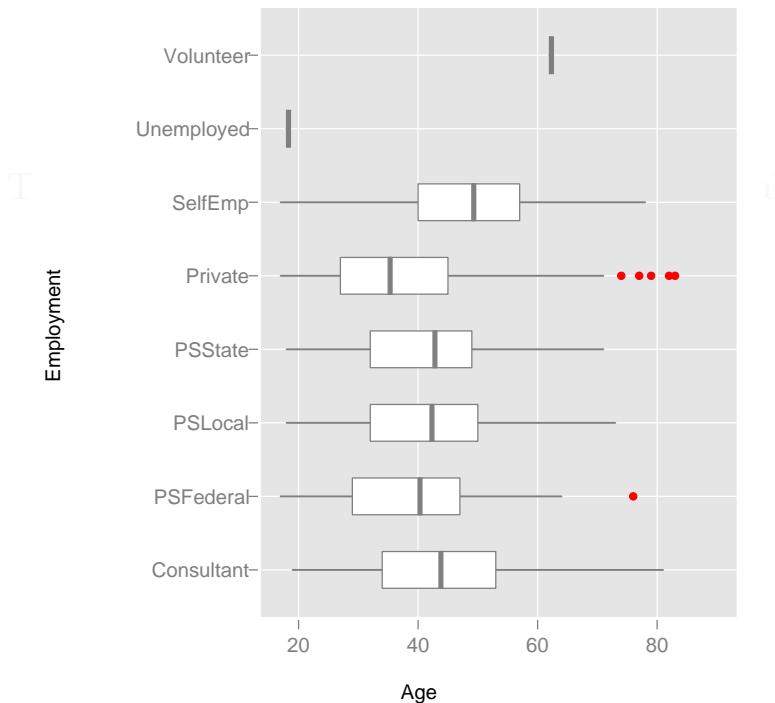
boxplot.limits <- as.matrix(c(mean(x) - 2*sd(x),
                               mean(x) - sd(x),
                               mean(x),
                               mean(x) + sd(x),
                               mean(x) + 2*sd(x)))
boxplot.meansd <- list(stats = boxplot.limits,
                        n = length(x),
                        conf = NULL,
                        out = numeric(0))
bxp(boxplot.meansd, main=expression("mean" %+-% "1 and 2 SDs"), ylim=c(0,5.2))
```

```
par(oldpar)
```

R code source: [rplot-wine-boxplot-tuning.R](#).

Boxplot From ggplot

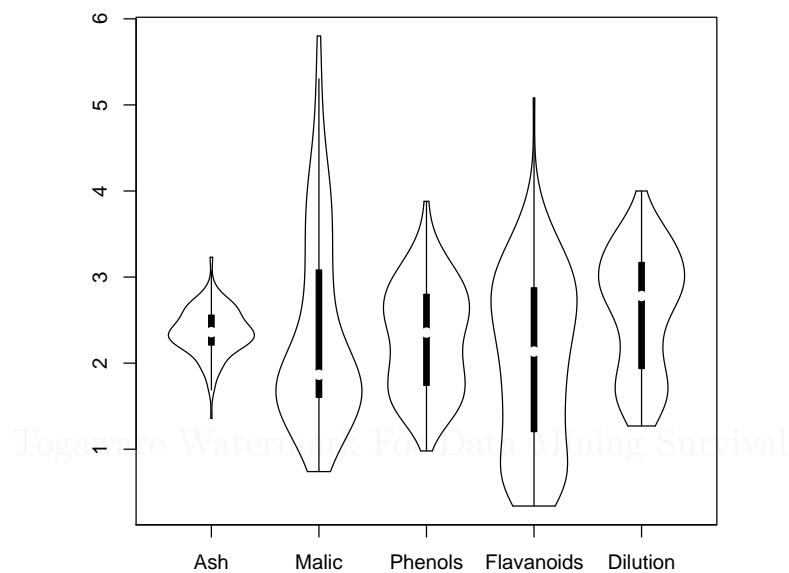
The *ggplot* also provide functionality to display boxplots. The *qplot* function is a simple interface to generate one. Here we use the *audit* dataset to explore the distribution of Age against Education.



```
library(rattle)
data(audit)
attach(audit)
```

R code source: [rplot-boxplot-qplot.R](#).

16.3.3 Violin Plot



```
load("wine.Rdata")
library(vioplot)
attach(wine)
vioplot(Ash, Malic, Phenols, Flavanoids, Dilution,
        names=c("Ash", "Malic", "Phenols", "Flavanoids", "Dilution"))
```

R code source: [rplot-wine-vioplot.R](#).

See also

```
> library(lattice)
> example(panel.violin)
```

16.3.4 What Distribution

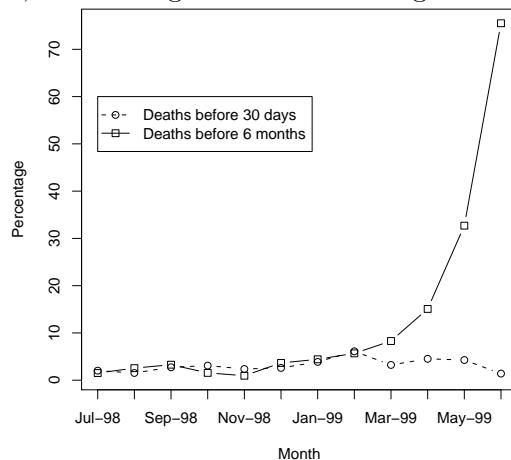
16.3.5 Labelling Outliers

```
set.seed(10)
x <- rexp(100)
out <- boxplot(x)$out
text(rep(1.1, length(out)), out, sprintf("%0.2f", out))
```

16.4 Miscellaneous Plots

16.4.1 Line and Point Plots

A dot plot can be extended to draw lines between the dots! We use a simple dot plot to compare two sets of data. The points record (fictional) percentages of patients recorded as dying within 30 days and within 6 months of some procedure. Once again, a plot is first created for `death6m`. In this case we have both points and lines (`type="b"`), solid lines are used (`lty=1`), and a small square is used to plot points (`pch=0`). The rest of the plot is then constructed by adding a plot for `death30`, adding a box around the plot, and adding two axes and a legend.



```
pdf('graphics/rplot-dot.pdf')
dates <- c('Jul-98', 'Aug-98', 'Sep-98', 'Oct-98', 'Nov-98', 'Dec-98',
          'Jan-99', 'Feb-99', 'Mar-99', 'Apr-99', 'May-99', 'Jun-99')
death30 <- c(2.02, 1.53, 2.73, 3.09, 2.37, 2.60,
            3.87, 6.11, 3.23, 4.52, 4.27, 1.40)
death6m <- c(1.52, 2.55, 3.28, 1.55, 0.95, 3.65,
            4.42, 5.68, 8.29, 15.08, 32.70, 75.52)
```

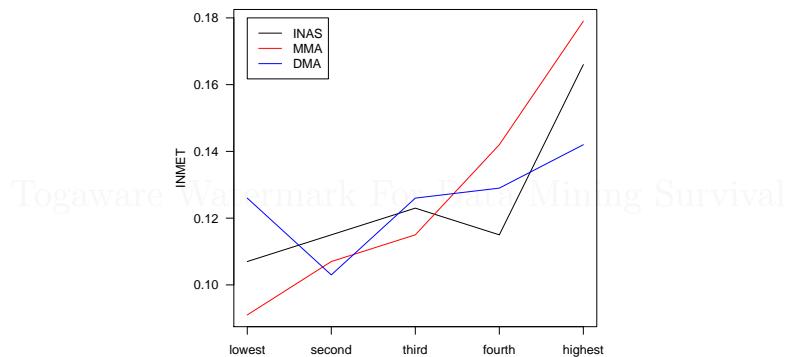
```
plot(death6m, type='b', xlab='Month', ylab='Percentage',
      lty=1, pch=0, axes=FALSE)
lines(death30, type='b', lty=2, pch=1)
box()
axis(1, at=seq(1, length(dates)), labels=dates)
axis(2, at=seq(0, 100, 10))
legend(1, 60, c('Deaths before 30 days', 'Deaths before 6 months'),
       lty=c(2, 1), pch=c(1, 0))
dev.off()
```

R code source: [rplot-dot.R](#).

Togaware Watermark For Data Mining Survival

16.4.2 Matrix Data

The following example illustrates the use of a matrix as the source of the data to be plotted. The matrix has three columns, one for each of the items of interest. The rows correspond to observations. We want to plot the values for each item across the observations, joining the points with a line. Key features to note include the axis labels always being horizontal (`las=1`) and each line being a solid line (`lty=1`).

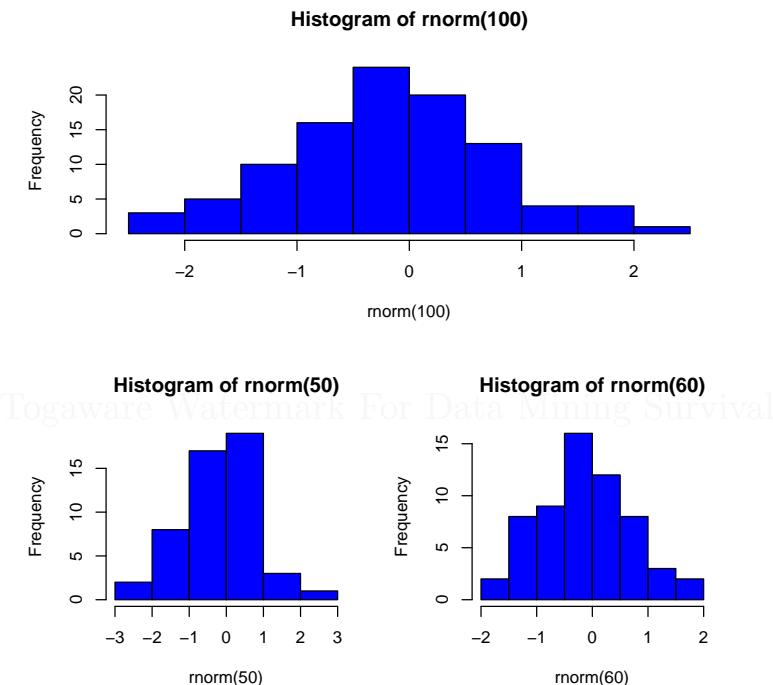


```
# Suggested by Chuck Cleland
pdf("graphics/rplot-matplot.pdf")
myFrame <- data.frame(lowest = c(0.107, 0.091, 0.126),
                      second = c(0.115, 0.107, 0.103),
                      third = c(0.123, 0.115, 0.126),
                      fourth = c(0.115, 0.142, 0.129),
                      highest = c(0.166, 0.179, 0.142),
                      sig = c(0.000, 0.000, 0.031))
rownames(myFrame) <- c("INAS", "MMA", "DMA")
par(las=1)
matplot(t(myFrame[,-6]), type="l", xaxt="n", ylab="INMET",
        col=c("black", "red", "blue"), lty=c(1,1,1))
axis(side=1, at=1:5, names(myFrame)[1:5])
legend(1, 0.18, rownames(myFrame), lty=c(1,1,1),
       col=c("black", "red", "blue"))
dev.off()
```

R code source: [rplot-matplot.R](#).

16.4.3 Multiple Plots

Place three plots on a single plot with *layout*

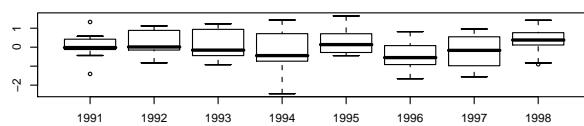


```
pdf("graphics/rplot-multi-hist.pdf")
layout(matrix(c(1, 1, 2, 3), 2, 2, byrow = TRUE))
hist(rnorm(100), col='blue')
hist(rnorm(50), col='blue')
hist(rnorm(60), col='blue')
dev.off()
```

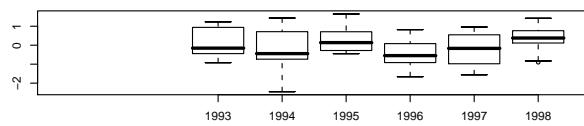
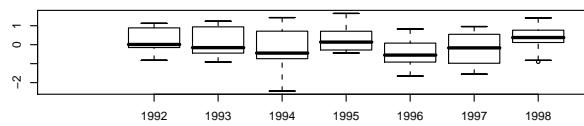
R code source: [rplot-multi-hist.R](#).

16.4.4 Aligned Plots

The next example illustrates how to make use of information in the *usr* parameter (the x and y extent of the current plot) to align columns in three separate plots. Here we create some random data and arrange it into a data frame for 8 years of observations. Using the *mfrow* option we indicate that we want 3 rows and 1 column of plots. We plot the first boxplot and save the value of *usr*. A new plot canvas is created and its *usr* values is set to the same as the first plot so that we can specify locations to place the following plots (using *at*). We also illustrate the use of the *subset* function.



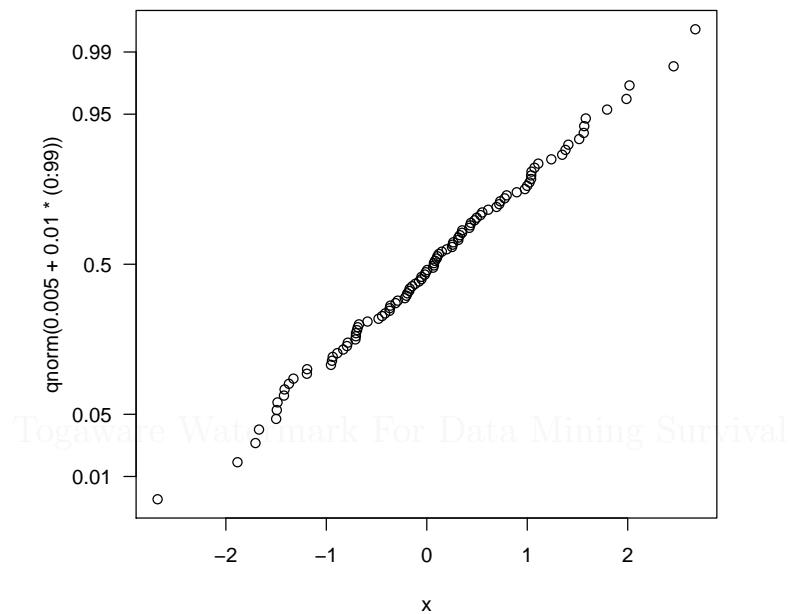
Togaware Watermark For Data Mining Survival



```
dat <- rnorm(80)
years <- rep(1991:1998, each=10)
ds <- cbind(dat, years)
par(mfrow=c(3, 1))
boxplot(dat ~ years, ds)
usr <- par("usr")
plot.new()
par(usr=usr)
boxplot(dat ~ years, subset(ds, years %in% 1992:1998), at=2:8, add=TRUE)
plot.new()
par(usr=usr)
boxplot(dat ~ years, subset(ds, years %in% 1993:1998), at=3:8, add=TRUE)
```

R code source: [rplot-multi-align.R](#).

16.4.5 Probability Scale



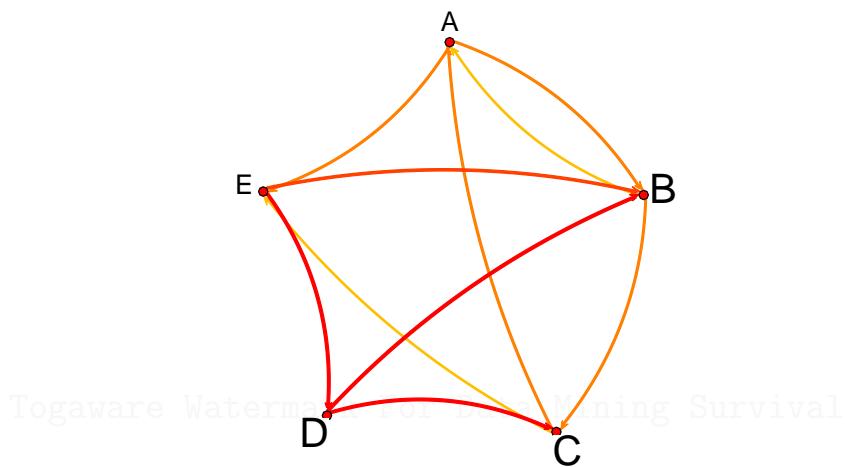
```

pdf("graphics/rplot-proby.pdf")
x <- sort(rnorm(100))
y <- 0.5+(0:99)
p <- c(0.01, 0.05, 0.5, 0.95, 0.99)
qqplot(x, qnorm(0.005+0.01*(0:99)), yaxt="n")
axis(2, at=qnorm(p), label=p, las=1)
dev.off()

```

R code source: [rplot-proby.R](#).

16.4.6 Network Plot



Rattle provides *plotNetwork* to do most of the following.

```
pdf("graphics/rplot-network.pdf")
library(network)
cash <- matrix(c(0, 10000, 0, 0, 20000,
                 1000, 0, 10000, 0, 0,
                 5000, 0, 0, 0, 3000,
                 0, 1000000, 600000, 0, 0,
                 0, 50000, 0, 500000, 0),
                 nrow=5, byrow=TRUE)

## Label the entities

rownames(cash) <- colnames(cash) <- c("A", "B", "C", "D", "E")

## Create a network

cash.net <- network(cash)

## We can change the line widths to represent the magnitude of the
## cash flow. We used a log transform to get integers for the line
## widths.
```

```

cash.log <- log10(cash) # Log 10 to get magnitude
cash.log[cash.log==Inf] <- 0 # Set resulting -Infinity (log10(0)) values to 0
cash.mag <- round(cash.log) # Round them to

## We can also add color to indicate the magnitude. Use heat colours
## to indicate the magnitude of the cash flow, from yellow to red.

heat <- rev(heat.colors(max(cash.mag)))
cash.col <- cash.mag
for (i in 1:length(heat)) cash.col[cash.col==i] <- heat[i]

## Record the magnitude of cash coming into any label and use this to
## scale the entity labels.

entity.sizes <- round(log10(apply(cash, 2, sum)))
entity.sizes <- 1 + entity.sizes-min(entity.sizes)
entity.sizes <- 1 + entity.sizes/max(entity.sizes)

plot(cash.net, displaylabels=TRUE, usecurve=TRUE, mode="circle",
      edge.lwd=cash.mag, edge.col=cash.col,
      label.cex=entity.sizes, label.border=0)

dev.off()

```

R code source: [rplot-network.R](#).

16.4.7 Sunflower Plot

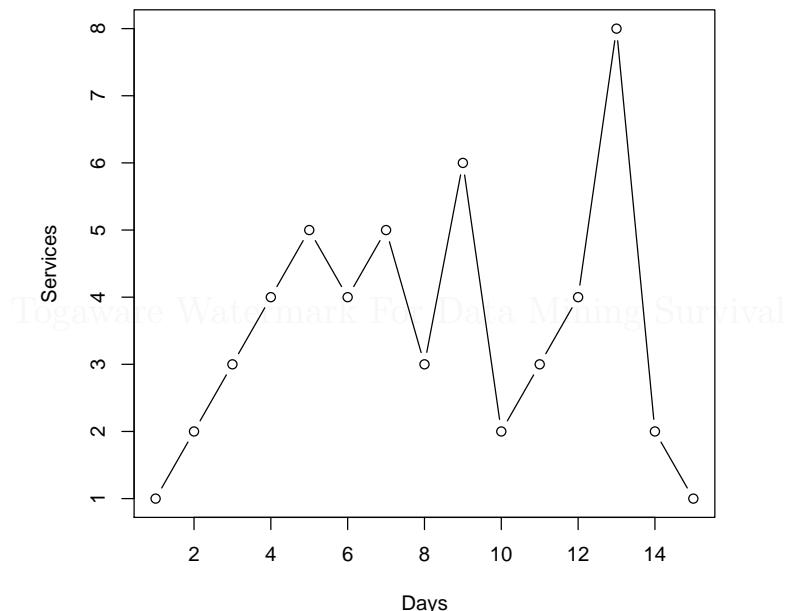
A sunflower plot will plot multiple points at the same location “sunflowers” with multiple leaves or petals. The *sunflower* function is provided to generate such plots. Thus, overplotting is visualised instead of it simply disappearing.

```
# Better example of overplotting points.  
> sunflowerplot(wine$Phenols, wine$Flavanoids)
```

Togaware Watermark For Data Mining Survival

16.4.8 Stairs Plot

Another simple example plotting a sequence of numbers uses stairs (`type="s"`) to give a city landscape type of plot.

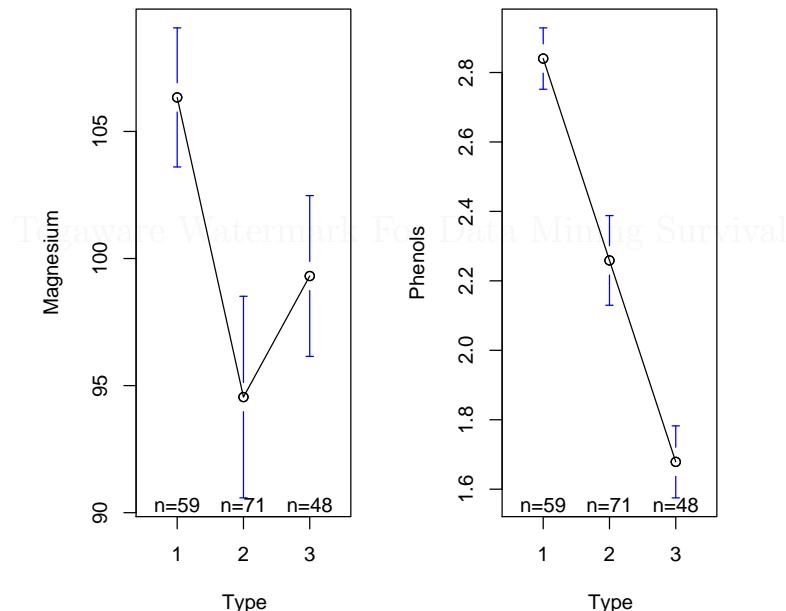


```
pdf("graphics/rplot-line.pdf")
l = c(1, 2, 3, 4, 5, 4, 5, 3, 6, 2, 3, 4, 8, 2, 1)
plot(l, type="b", xlab="Days", ylab="Services")
dev.off()
```

R code source: [rplot-line.R](#).

16.4.9 Graphing Means and Error Bars

The simplest plot of means is achieved using the *plotmeans* function of the *gplots* package. The example uses the wine dataset, aggregating the data into the three classes defined by **Type** and plotting the mean of the value for Phenols and Magnesium for each class.



```
library("gplots")
load("wine.Rdata")
attach(wine)

pdf("graphics/rplot-line-means.pdf")
par(mfrow=c(1,2))
plotmeans(Magnesium ~ Type)
plotmeans(Phenols ~ Type)
dev.off()
```

R code source: [rplot-line-means.R](#).

Both plots are placed onto the one plotting canvas (using `par(mfrow=c(1,2))`). They are placed side-by-side which exaggerates the bars around the means. A visual inspection indicates that the three groups have quite different means for Magnesium and for Phenols, but it is more significant for Phenols.

We can evaluate this statistically using R. Comparing the means between different subsets of a dataset is called **analysis of variance** or **ANOVA**. Here we compare the means of Magnesium, and, separately, the means of Phenols across the Types.

```
> anova(lm(Phenols ~ Type))

Analysis of Variance Table

Response: Phenols
            Df  Sum Sq Mean Sq F value    Pr(>F)
Type          2 35.857  17.928  93.733 < 2.2e-16 ***
Residuals   175 33.472    0.191
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

> anova(lm(Magnesium ~ Type))

Analysis of Variance Table

Response: Magnesium
            Df  Sum Sq Mean Sq F value    Pr(>F)
Type          2 4491.0  2245.5  12.430 8.963e-06 ***
Residuals   175 31615.1    180.7
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

The `Pr(>F)` value is clearly smaller than 0.05, thus with 95% confidence we see that the means are different.

If however we look at just Types 2 and 3, and compare the means of the two groups:

```
> wine23 <- wine[Type!=1,]
> attach(wine23)
> anova(lm(Magnesium ~ Type))

Analysis of Variance Table

Response: Magnesium
            Df  Sum Sq Mean Sq F value    Pr(>F)
Type          1  649.8   649.8  3.0141  0.08518 .
Residuals   117 25221.9    215.6
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

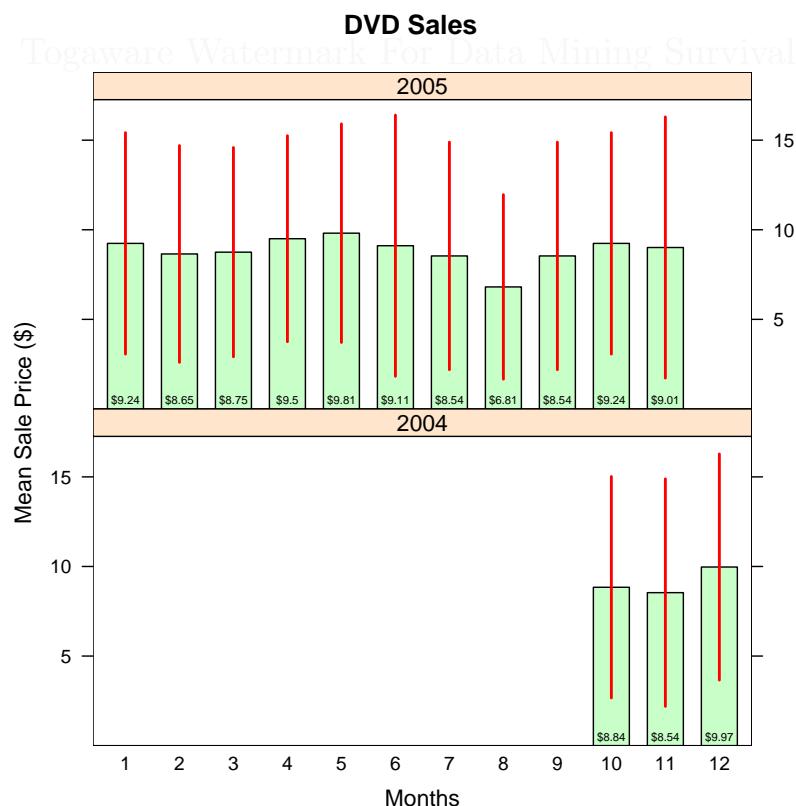
With a $\Pr(>F)$ of 0.08518, which is larger than 0.05, the means for Magnesium across these two groups is not significantly different (at the 95% level). However, it is significant at the 90% level of confidence (indicated by the period following the number in the output, and the legend below associating this with 0.1 - 10%).

Togaware Watermark For Data Mining Survival

16.4.10 Bar Charts With Segments

In this example we illustrate one of the powerful features of the *barchart* function (and others) in the *lattice* package. In addition to drawing the two basic barcharts (one corresponding to each year of the data) we also want to include standard deviation bars. This is achieved by drawing the normal barcharts but adding, through the use of the *panel* argument, the standard deviation plots.

First we create a data frame recording the mean sale price of an item each month, its standard deviation, and the number of items sold. We then *attach* the data frame so we can more easily refer to the column names within the *barchart* call without having to prefix them with the data frame name. We also choose a white background using *trellis.par.set*.



A *barchart* is then constructed to plot the Mean across Month, by Year. The layout specifies a single column, and as many years as in the data. The ylim ensures we have enough space to draw the standard deviation bars. Next the StdDev is assigned to **sd**, and the **panel** function is defined to make use of **sd**. This draws the actual barchart and adds panel segments corresponding to the values in **sd** for the specific panel (as identified in the **subscripts** argument). We also include on the graphic the actual mean dollar amount.

```
# Suggested by Sandeep Ghosh
library(lattice)

prices <- matrix(c(10, 2004, 8.84, 6.18, 524,
                  11, 2004, 8.54, 6.35, 579,
                  12, 2004, 9.97, 6.31, 614,
                  1, 2005, 9.24, 6.18, 634,
                  2, 2005, 8.65, 6.05, 96,
                  3, 2005, 8.75, 5.84, 32,
                  4, 2005, 9.50, 5.75, 96,
                  5, 2005, 9.81, 6.10, 165,
                  6, 2005, 9.11, 7.29, 8,
                  7, 2005, 8.54, 6.35, 579,
                  8, 2005, 6.81, 5.15, 16,
                  9, 2005, 8.54, 6.35, 579,
                  10, 2005, 9.24, 6.18, 634,
                  11, 2005, 9.01, 7.29, 8),
                  ncol=5, byrow=TRUE)
prices <- as.data.frame(prices)
colnames(prices) <- c("Month", "Year", "Mean", "StdDev", "Count")
prices$Month <- factor(prices$Month)      # Turn Month into a categorical
prices$Year <- factor(prices$Year)        # Turn Year into a categorical

attach(prices)

pdf("graphics/rplot-bar-complex.pdf")

trellis.par.set(theme = col.whitebg())
barchart(Mean ~ Month | Year, data=prices,
         layout=c(1, length(levels(Year))),
         ylim=c(0, max(Mean) + max(StdDev)),
         main="DVD Sales",
         xlab="Months",
         ylab="Mean Sale Price ($)",
         sd=StdDev,
         panel=function(x, y, ..., sd, subscripts)
{
  panel.barchart(x, y, ...)
  sd <- sd[subscripts]
  panel.segments(as.numeric(x), y-sd, as.numeric(x), y+sd,
                 col="red", lwd=2)
  means <- Mean[subscripts]
  panel.text(as.numeric(x), rep(0.5, length(subscripts)),
             paste("$", means, sep=""), cex=0.5)
```

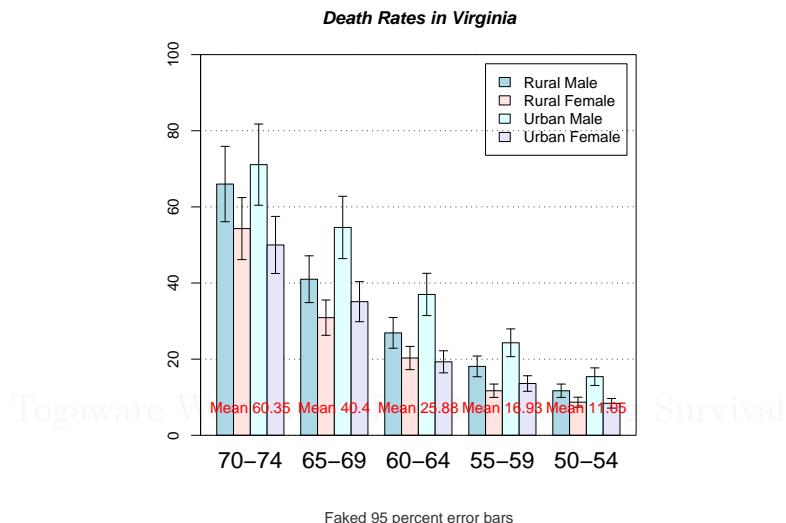
```
    })
dev.off()
```

R code source: [rplot-bar-complex.R](#).

Togaware Watermark For Data Mining Survival

16.4.11 Bar Plot With Means

Plot means and error bars by a group factor.

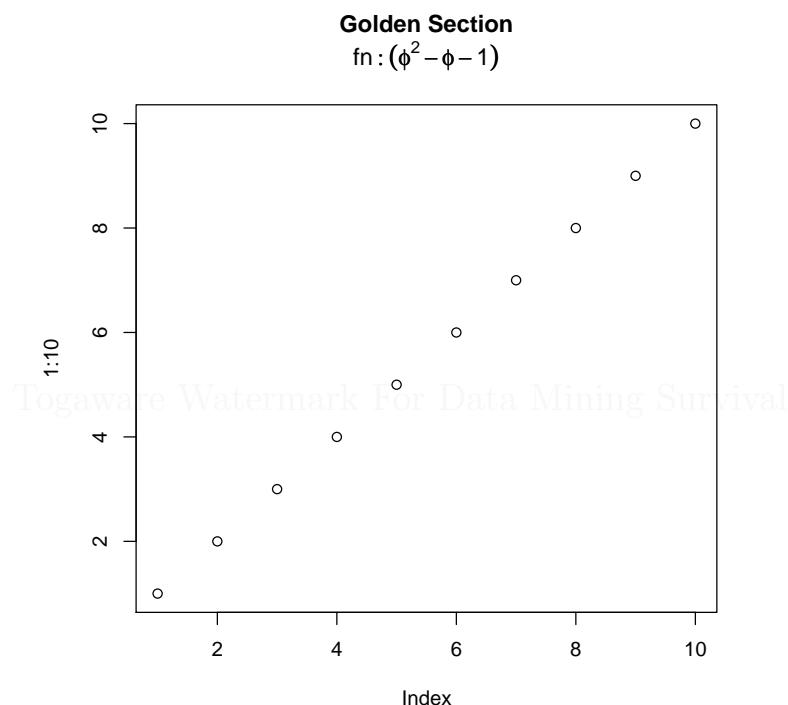


```
# Suggested by Sander Dom
library("gplots")
pdf("graphics/rplot-bar-means.pdf")
hh <- t(VADeaths)[, 5:1]
mybarcol <- "gray20"
ci.l <- hh * 0.85
ci.u <- hh * 1.15
mp <- barplot2(hh, beside = TRUE,
                col = c("lightblue", "mistyrose",
                       "lightcyan", "lavender"),
                legend = colnames(VADeaths), ylim = c(0, 100),
                main = "Death Rates in Virginia", font.main = 4,
                sub = "Faked 95 percent error bars", col.sub = mybarcol,
                cex.names = 1.5, plot.ci = TRUE, ci.l = ci.l, ci.u = ci.u,
                plot.grid = TRUE)
mtext(side = 1, at = colMeans(mp), line = -2,
      text = paste("Mean", formatC(colMeans(hh))), col = "red")
box()
dev.off()
```

R code source: [rplot-bar-means.R](#).

16.4.12 Multi-Line Title

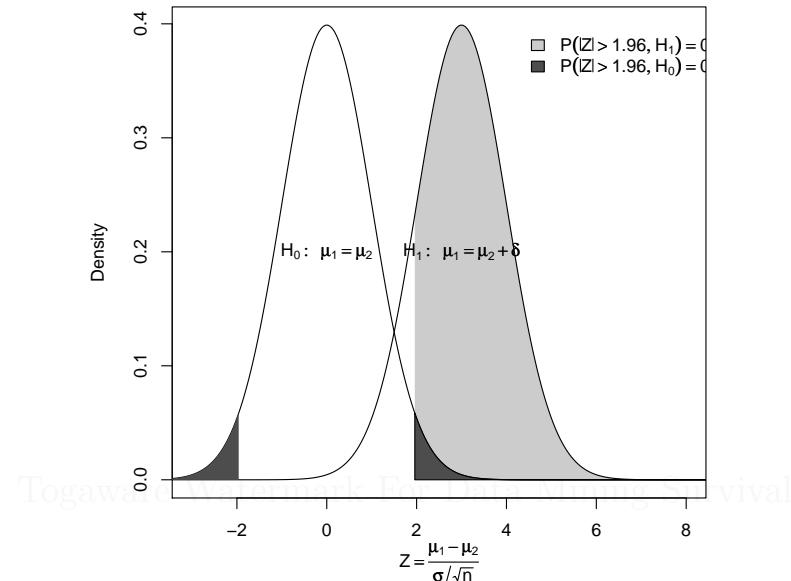
R does not support multi-line expressions but you can get the same effect by adding them separately through calls to `title`.



```
pdf("graphics/rplot-titles.pdf")
plot(1:10)
title("Golden Section", line=3)
title(expression(fn:(phi^2-phi-1)), line=2)
dev.off()
```

R code source: [rplot-titles.R](#).

16.4.13 Mathematics



```
# Posted by Thomas Lumley 20 Aug 2005
pdf("graphics/rplot-labels.pdf")
x<-seq(-10,10,length=400)
y1<-dnorm(x)
y2<-dnorm(x,m=3)
par(mar=c(5,4,2,1))
plot(x, y2, xlim=c(-3,8), type="n", xlab=quote(Z==frac(mu[1]-mu[2],
sigma/sqrt(n))), ylab="Density")
polygon(c(1.96,1.96,x[240:400],10), c(0,dnorm(1.96,m=3),y2[240:400],0),
col="grey80", lty=0)
lines(x, y2)
lines(x, y1)
polygon(c(-1.96,-1.96,x[161:1],-10), c(0,dnorm(-1.96,m=0), y1[161:1],0),
col="grey30", lty=0)
polygon(c(1.96, 1.96, x[240:400], 10), c(0,dnorm(1.96,m=0),
y1[240:400],0), col="grey30")
legend(4.2, .4, fill=c("grey80","grey30"),
legend=expression(P(abs(Z)>1.96, H[1])==0.85,
P(abs(Z)>1.96,H[0])==0.05), bty="n")
text(0, .2, quote(H[0]:~~mu[1]==mu[2]))
text(3, .2, quote(H[1]:~~mu[1]==mu[2]+delta))
dev.off()
```

R code source: [rplot-labels.R](#).

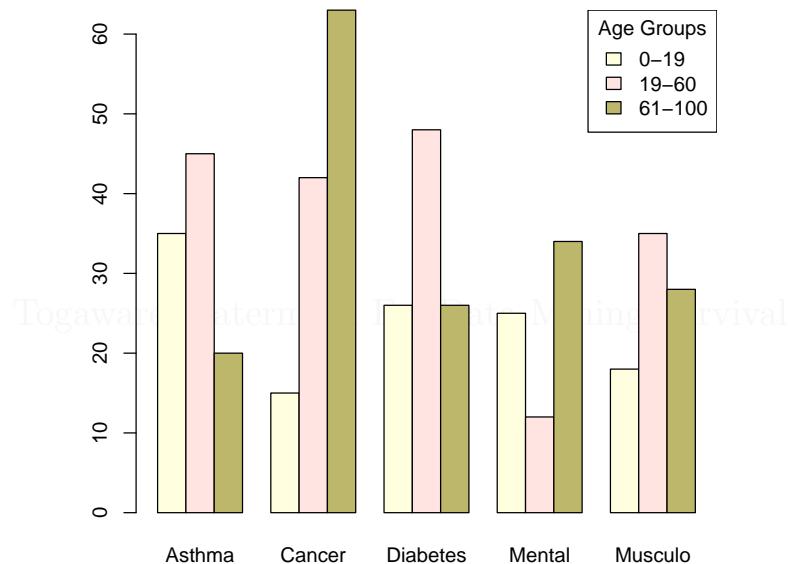
16.4.14 Plots for Normality

Q-Q Plot

Togaware Watermark For Data Mining Survival

16.4.15 Basic Bar Chart

A bar chart is useful to illustrate distributions of the population. In the following bar chart five classes of patient disease groups are identified, and a distribution of three age groups is shown for each disease group.



```

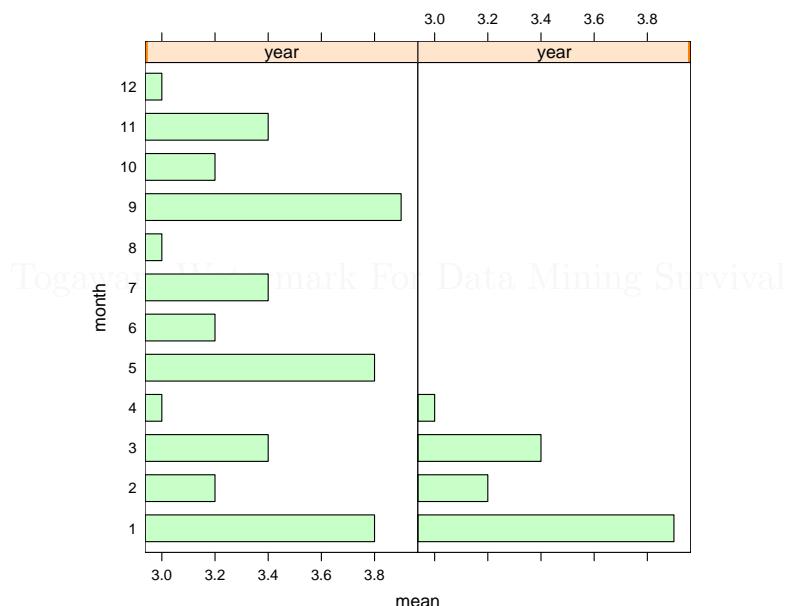
bars <- matrix(c(35, 15, 26, 25, 18,
                 45, 42, 48, 12, 35,
                 20, 63, 26, 34, 28), nrow=3, byrow=T)
rownames(bars) <- c("0-19", "19-60", "61-100")
colnames(bars) <- c("Asthma", "Cancer", "Diabetes", "Mental", "Musculo")
col <- c("lightyellow", "mistyrose", "darkkhaki")
barplot(bars, beside=TRUE, col=col)
legend("topright", rownames(bars), title="Age Groups", fill=col)

```

R code source: [rplot-bar.R](#).

16.4.16 Bar Chart Displays

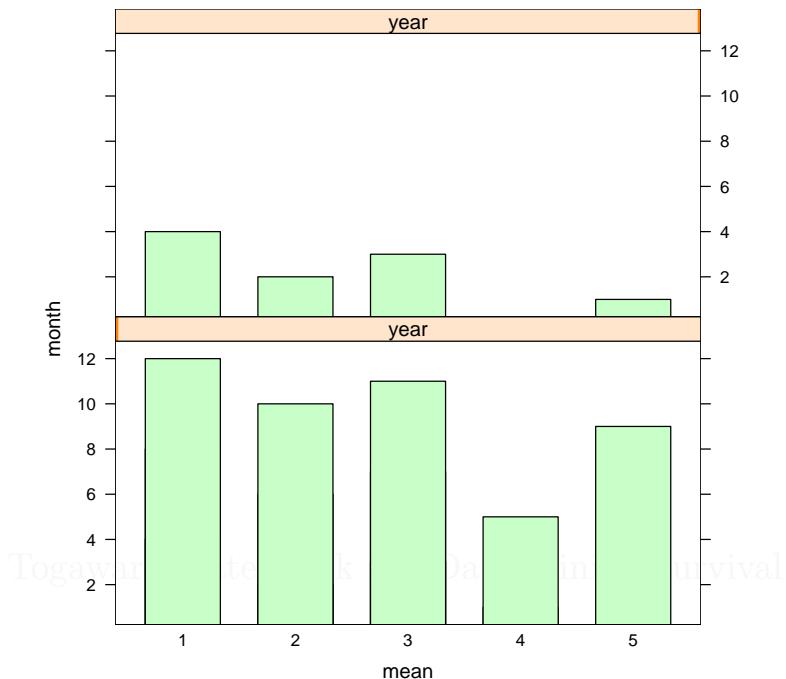
We illustrate some options in drawing bar charts from the *lattice* package. First a simple plot. We use `NO_CONVERSION` for `r.data.frame` so that the data frame retains the column names and for `r.barchart` so that the barchart can be printed to the device (otherwise it simply returns a data structure of the information).



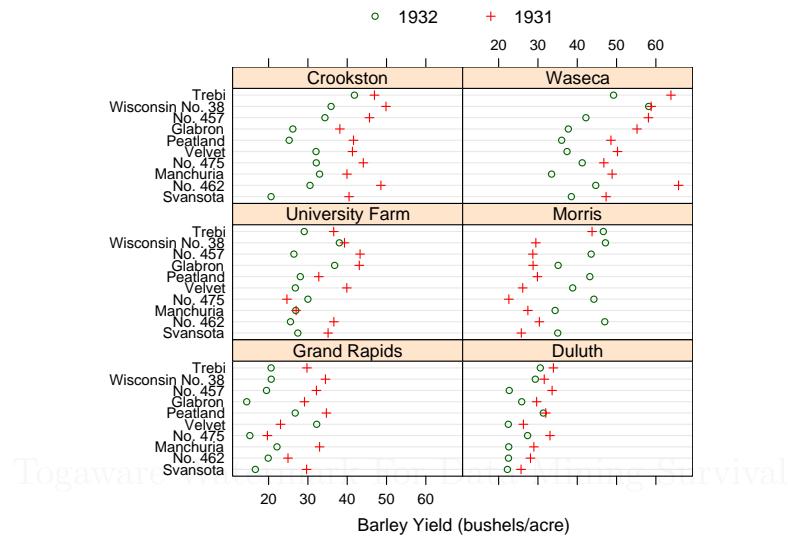
```
library("lattice")
pdf('graphics/rplot-bar-horizontal.pdf')
dataset <- data.frame(month=c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,
                           1, 2, 3, 4),
                       year=c(2004, 2004, 2004, 2004, 2004, 2004, 2004, 2004,
                             2004, 2004, 2004, 2005, 2005, 2005, 2005),
                       mean=c(3.8, 3.2, 3.4, 3.0, 3.8, 3.2, 3.4, 3.0,
                             3.9, 3.2, 3.4, 3.0, 3.9, 3.2, 3.4, 3.0))
trellis.par.set(theme=col.whitebg())
barchart(month ~ mean | year, data=dataset)
dev.off()
```

R code source: [rplot-bar-horizontal.R](#).

If we add `horizontal = False`, `layout = (1,2)` to the call to the `r.barchart` function, we can rotate the graphics:



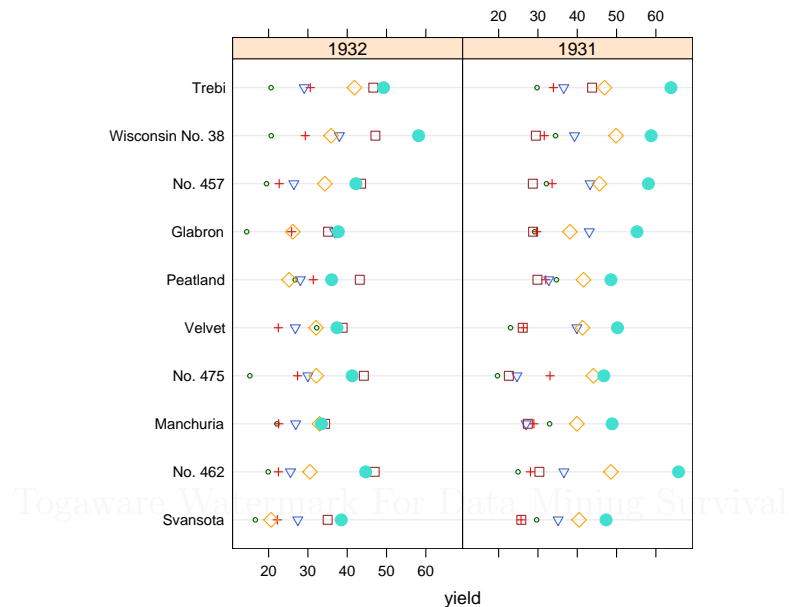
16.4.17 Multiple Dot Plots



```
# Suggested by Michael Friendly
pdf("graphics/rplot-trellis.pdf")
library(lattice)
data(barley)
n <- length(levels(barley$year))
trellis.device(new = FALSE, theme = col.whitebg())
dotplot(variety ~ yield | site,
        data = barley, groups = year,
        layout = c(2, 3), aspect = .5,
        xlab = "Barley Yield (bushels/acre)",
        key = list(points = Rows(trellis.par.get("superpose.symbol")), 1:n),
        text = list(levels(barley$year)), columns = n)
dev.off()
```

R code source: [rplot-trellis.R](#).

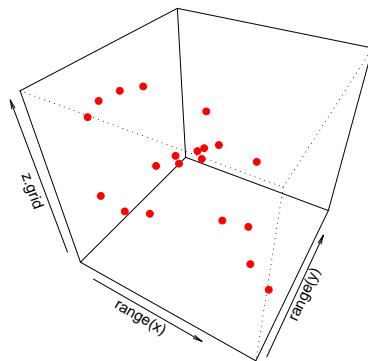
16.4.18 Alternative Multiple Dot Plots



```
# Suggested by Sundar Dorai-Raj
library("lattice")
data(barley)
pdf("graphics/rplot-trellis-shapes.pdf")
new.theme <- function()
{
  theme <- col.whitebg()
  symb <- theme$superpose.symbol
  symb$cex <- seq(0.5, 1.5, length = length(symb$cex))
  theme$superpose.symbol <- symb
  theme
}
trellis.par.set(theme = new.theme())
dotplot(variety ~ yield | year, data=barley, groups=site)
dev.off()
```

R code source: [rplot-trellis-shapes.R](#).

16.4.19 3D Plot



Togaware Watermark For Data Mining Survival

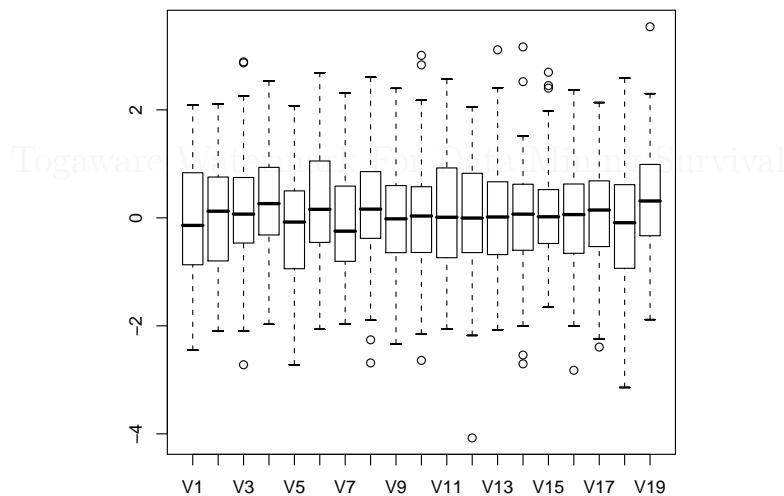
```
# Suggested by Robin Hankin
pdf("graphics/rplot-3dbox.pdf")
points3d <- function(x, y, z, jj.colour="black", ...)
{
  if(is.matrix(x))
  {
    z <- x[,3]
    y <- x[,2]
    x <- x[,1]
  }
  z.grid <- matrix(range(z), 2, 2)
  persp(range(x), range(y), z.grid,
        col=NA, border=NA, ...) -> res
  trans3d <- function(x,y,z, pmat)
  {
    tr <- cbind(x,y,z,1) %*% pmat
    list(x = tr[,1]/tr[,4], y= tr[,2]/tr[,4])
  }
  points(trans3d(x,y,z,pmat), col=jj.colour, ...)
}
O <- matrix(rnorm(60), 20, 3)
options(warn=-1) # Ignore two warnings from the following.
points3d(O, jj.colour="red", pch=16, theta=30, phi=40)
dev.off()
```

R code source: [rplot-3dbox.R](#).

16.4.20 Box and Whisker Plot

A simple box plot of randomly generated data. The box in each plot shows the median (as the line within the box) and one standard deviation from the mean (the extremes of the box). The whiskers show the second standard deviation, and the circles show outliers.

In this code we use R's *rnorm* function to generate a standard_normal random matrix of the given shape (rows by cols). The dataset is transformed to an R data frame. The x axis is labelled with an appropriate number of letters from the alphabet.

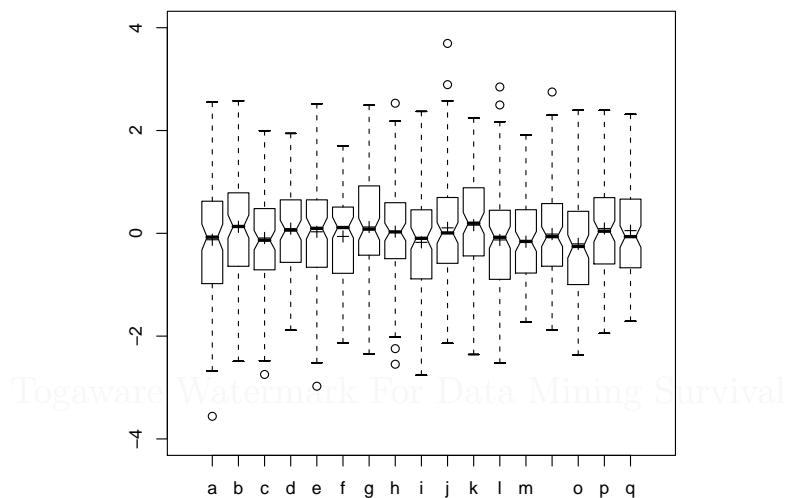


```
set.seed(2)
ds <- matrix(rnorm(19 * 100), ncol = 19)
pdf("graphics/rplot-boxplot.pdf")
plot.new()
plot.window(xlim = c(0, 20), ylim = range(ds), xaxs = "i")
boxplot(as.data.frame(ds), add = TRUE, at = 1:19)
dev.off()
```

R code source: [rplot-boxplot.R](#).

16.4.21 Box and Whisker Plot: With Means

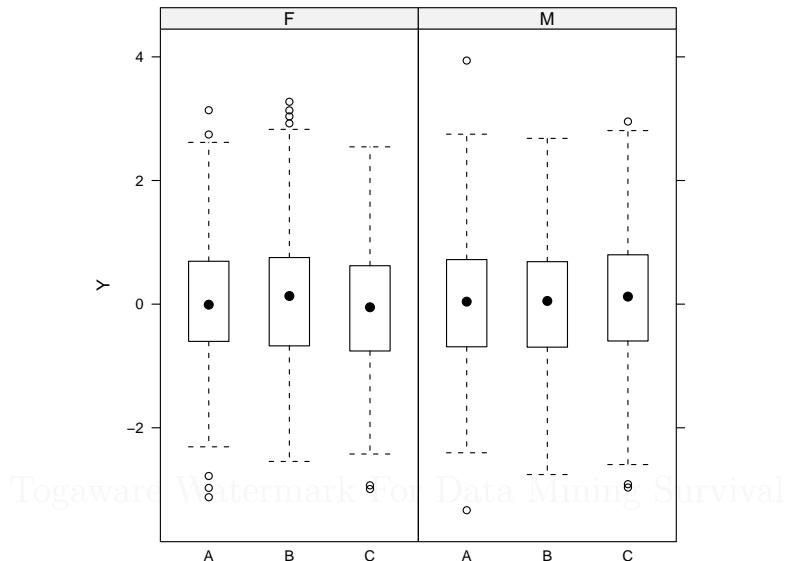
The simple box plot is extended to include the mean of each column. The means are calculated using `scipy`, and plotted as a + (`pch=3`).



```
rows <- 100
cols <- 17
pdf("graphics/rplot-boxplot-means.pdf")
dataset <- matrix(rnorm(rows*cols), nrow=rows, ncol=cols)
means <- mean(as.data.frame(dataset))
boxplot(as.data.frame(dataset), notch=TRUE, at=1:17,
        xlim=c(0, cols+1), ylim=c(-4,4), xaxt='n')
axis(side=1, at=1:17, labels=letters[1:17])
points(1:17, means, pch=3)
dev.off()
```

R code source: [rplot-boxplot-means.R](#).

16.4.22 Clustered Box Plot



```

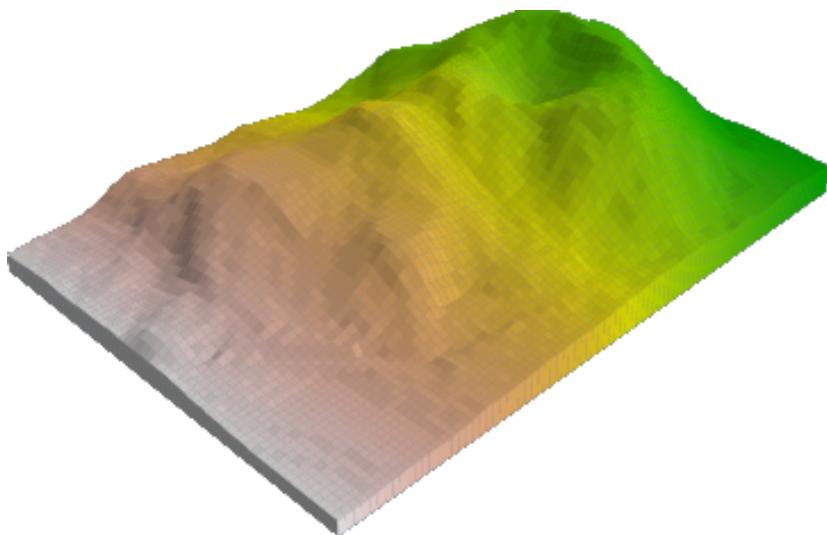
pdf("graphics/rplot-bwplot.pdf")
mydata <- data.frame(Y = rnorm(3*1000),
                      INDFACT = rep(c("A", "B", "C"), each=1000),
                      CLUSFACT=factor(rep(c("M", "F"), 1500)))
library(lattice)
trellis.device(new=FALSE, col=FALSE)
bwplot(Y ~ INDFACT | CLUSFACT, data=mydata, layout=c(2,1))
dev.off()

```

R code source: [rplot-bwplot.R](#).

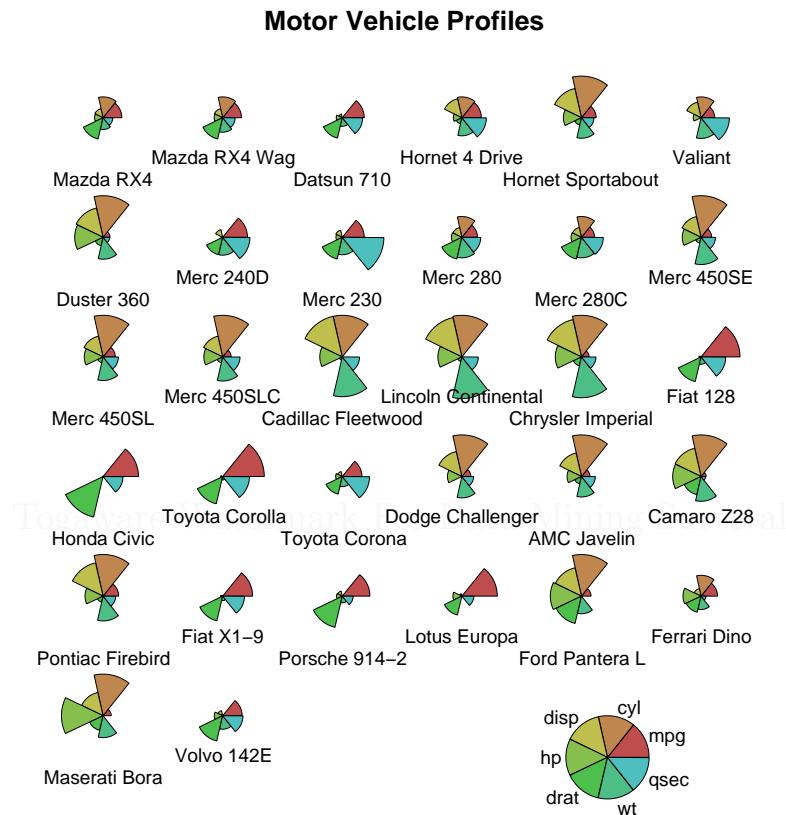
16.4.23 Perspective Plots

See the *persp* package.



```
demo(persp)
```

16.4.24 Star Plot

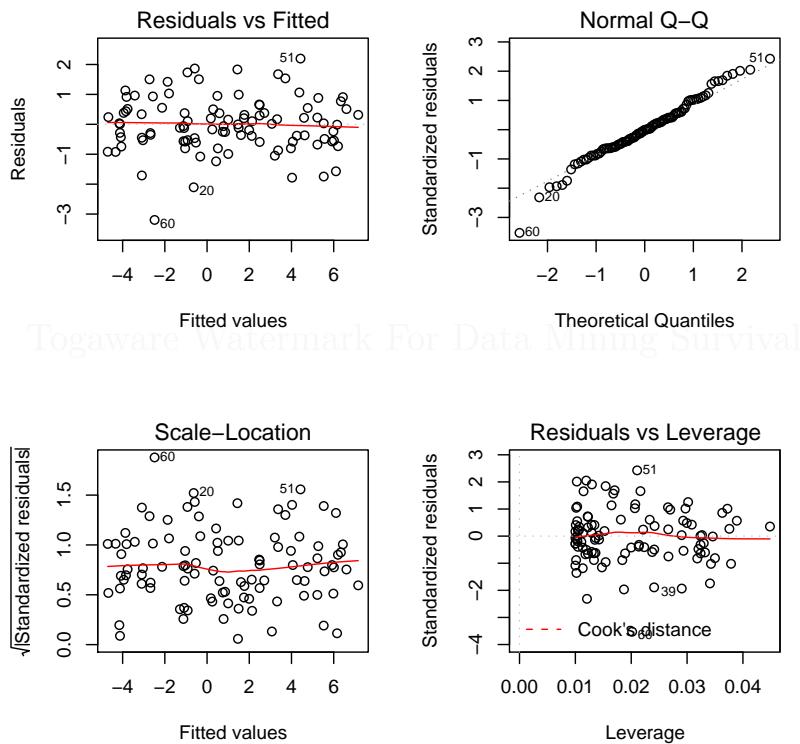


```
# See http://zonnek2.free.fr/UNIX/48_R/04.html
pdf('graphics/rplot-stars.pdf')
data(mtcars)
palette(rainbow(12, s = 0.6, v = 0.75))
stars(mtcars[, 1:7], len = 0.8, key.loc = c(12, 1.5),
      main = "Motor Vehicle Profiles", draw.segments = TRUE)
dev.off()
```

R code source: [rplot-stars.R](#).

16.4.25 Residuals Plot

A plot of an lm (linear model) object. This provides a clear picture of any strange behaviour from the residuals.

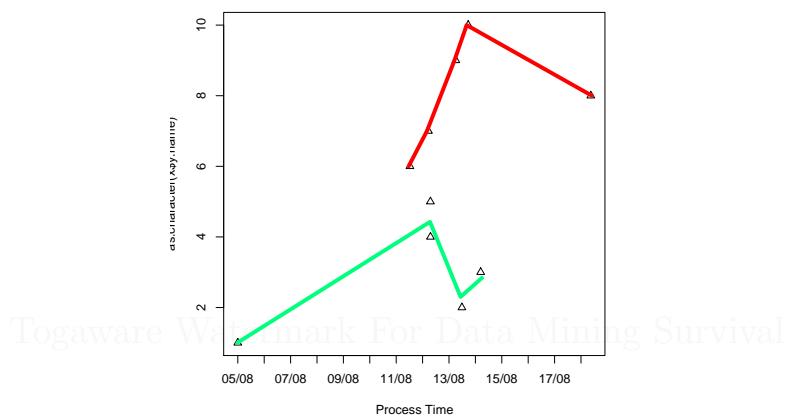


```
pdf("graphics/rplot-lm-residuals.pdf")
x <- runif(100, -3, 3)
y <- 1 + 2*x + rnorm(100)
m <- lm(y~x)
par(mfrow=c(2,2))
plot(m)
dev.off()
```

R code source: [rplot-lm-residuals.R](#).

16.5 Dates and Times

R has full support for dealing with dates and times. The `date()` function returns the current date as a string: `Wed Oct 20 06:48:06 2004`. The following example illustrates some of the basic functionality for date handling.



```

pdf("graphics/rplot-date.pdf")
Time <- c("2004-08-05 09:08:48", "2004-08-13 20:53:38",
         "2004-08-14 13:57:23", "2004-08-12 16:17:41",
         "2004-08-12 16:15:27", "2004-08-11 21:38:24",
         "2004-08-12 14:28:41", "2004-08-18 18:04:47",
         "2004-08-13 15:23:14", "2004-08-14 02:36:33")
Time <- as.POSIXct(Time)
x <- data.frame(main.name=="AAA", fname=rep(c("Apple","Watermelon"),
                                             each=5), x.name=Time, y.name=(1:10))
par(mai=c(1, .7, .5, .3))
plot(x$x.name, as.character(x$y.name), xlab="Process Time", xaxt='n',
      pch=2)
axis.POSIXct(1, at=seq(min(x$x.name), max(x$x.name), "days"),
             format="%d/%m")
fruit.class <- table(x$fname)
fcolor <- c(611,552,656,121,451,481,28,652,32,550,90,401,150,12,520,8)
for(j in 1:length(fruit.class))
{
  fruit <- names(fruit.class)[j]
  lines(smooth.spline(x[x$fname==fruit, "x.name"],
                       x[x$fname==fruit, "y.name"]),
        col=colors()[fcolor[j]], cex = 0.5, lwd=5)
}
dev.off()

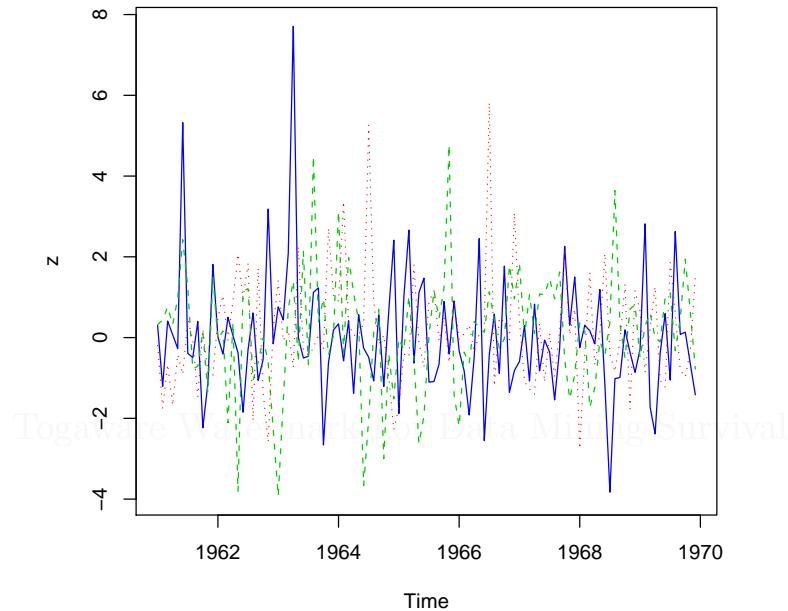
```

R code source: [rplot-date.R](#).

16.5.1 Simple Time Series

Togaware Watermark For Data Mining Survival

16.5.2 Multiple Time Series

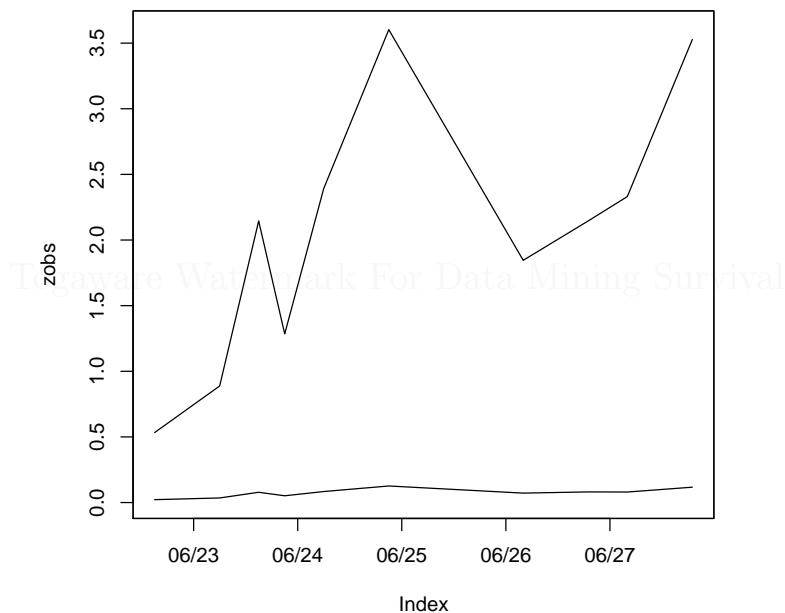


```
z <- ts(matrix(rt(200*8, df=3), 200, 8), start=c(1961,1), frequency=12)
z <- window(z[, 1:3], end=c(1969, 12))
plot(z, plot.type="single", lty=1:3, col=4:2)
```

R code source: [rplot-time-multi.R](#).

16.5.3 Plot Time Series

This example creates a time series dataset recording two observations at each time step. The date and times are converted to *chron* objects, and then a *zoo* series is created, which is then plotted.



```

year  <- c(rep(2005,10))
doy   <- c(rep(173,5), rep(174,5))
time  <- c(15,30,45,100,115,15,30,45,100,115)
obs1  <- c(0.022128,0.035036,0.051632,0.071916,0.081136,
          0.07837,0.083902,0.126314,0.080214,0.117094)
obs2  <- c(0.533074667,0.887982667,1.284938,1.845450333,2.145839333,
          2.145126667,2.392422,3.60253,2.330776333,3.5277)
obs   <- cbind(year, doy, time, obs1, obs2)

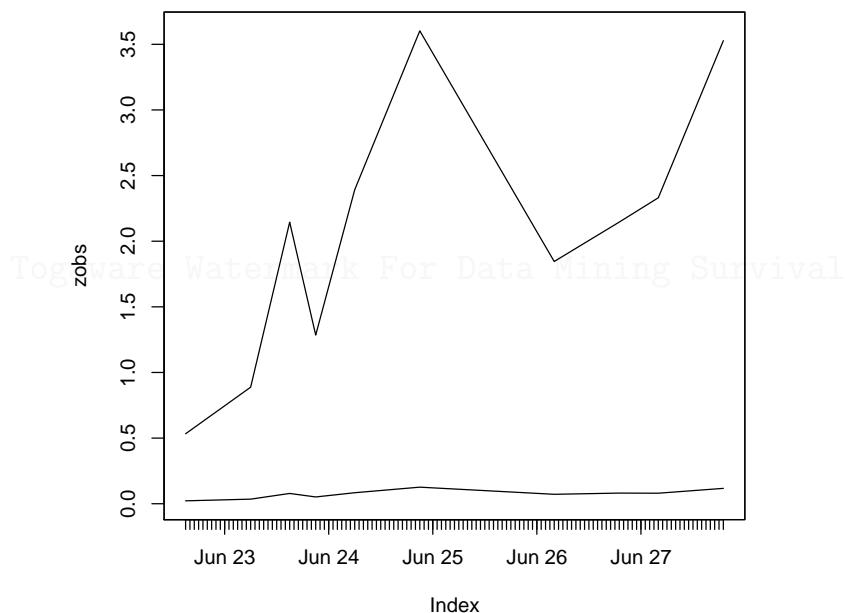
library(chron)
library(zoo)
datetimes <- chron(paste(1, 1, obs[,1], sep="/"), obs[,3]/24) + obs[,2] - 1
zobs <- zoo(obs[,4:5], datetimes)
plot(zobs, plot.type = "single")

```

R code source: [rplot-time-basic.R](#).

16.5.4 Plot Time Series with Axis Labels

With the same data as previously, here we control the axis labels. We tell the plot not to include axes (`xaxt="n"`). Then we add days to the x axis, and then we add tick marks for the hours within the day.



```
plot(zobs, plot.type="single", xaxt="n")
days <- seq(min(floor(datetimes)), max(floor(datetimes)))
axis(1, days, format(as.Date(days), "%b %d"), tcl=-0.6)
hours <- seq(min(datetimes), max(datetimes), by=1/24)
axis(1, hours, FALSE, tcl=-0.4)
```

R code source: [rplot-time-basic-labels.R](#).

16.5.5 Grouping Time Series for Box Plot

```
> observer.time <- strptime(1:365, format="%j") # Generate dates
> observer.measure <- rnorm(365)
> ds <- data.frame(time = observer.time, measure = observer.measure)
```

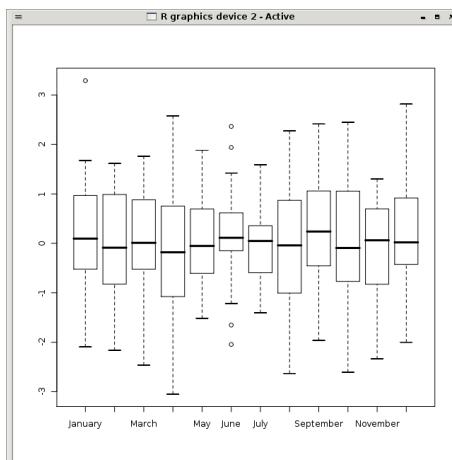


Figure 16.1: An ordered monthly box plot.

```
> observer.months <- ordered(months(ds$time),
  levels = c("January", "February", "March", "April", "May", "June",
            "July", "August", "September", "October", "November",
            "December"))
> boxplot(ds$measure ~ observer.months)
```

16.6 Using gGobi

Gobi is an excellent free and open source tool for visualising data, supporting brushing.

Run the `ggobi` function. Load a CSV file (select the Reader Type in the File→Open dialogue). Then under the Display menu choose a variety of displays.

16.6.1 Quality Plots Using R

We can save the plots generated by GGobi into an R script file and then have R generate the plots for you. This allows the plots to be regenerated as publication quality graphics.

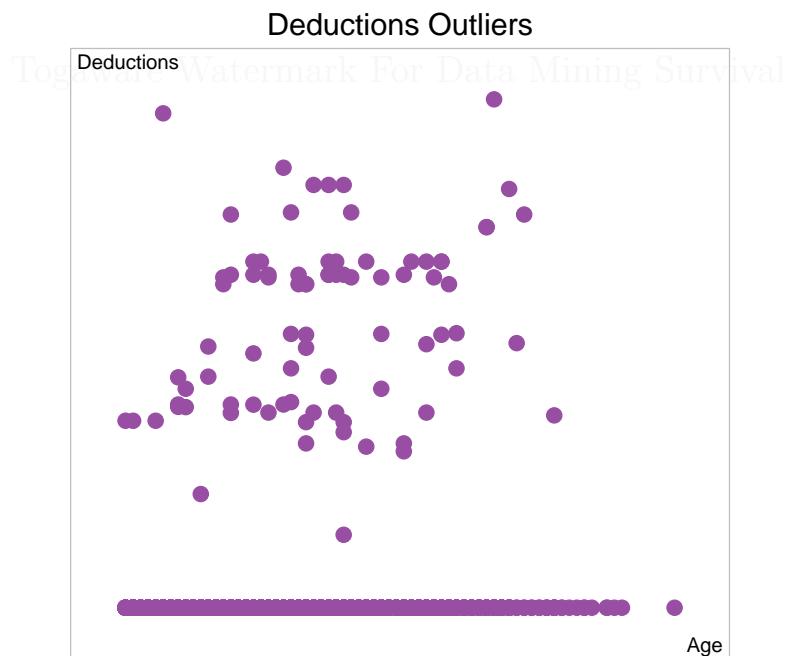
The `DescribeDisplay` package is required for this:

```
> install.packages("DescribeDisplay")
> library(DescribeDisplay)
```

Then, within GGobi choose from the Tools menu to Save Display Description. This will prompt you for a filename into which GGobi will write an R script to recreate the current graphic. We can load this script into R with the *dd_load* function and generate a plot in the usual way:

```
> pd <- dd_load("ggobi-saved-display-description.R")
> pdf("ggobi-rplot-deductions-outliers")
> plot(pd)
> dev.off()
> ggplot(pd)
```

The resulting plot saved to a pdf file is:



16.7 Textual Summaries

We saw in Chapter 3 some of the R functions that help us get a basic picture of the scope and type of data in any dataset. These include the most basic of information including the number and names of columns and rows (for data frames) and a summary of the data values themselves. We illustrate this again with the *wine* dataset (see Section 14.3.4, page 219):

```
> load("wine.RData")
> dim(wine)
[1] 178 14
> nrow(wine)
[1] 178
> ncol(wine)
[1] 14
> colnames(wine)
[1] "Type"          "Alcohol"        "Malic"          "Ash"
[5] "Alcalinity"     "Magnesium"      "Phenols"        "Flavanoids"
[9] "Nonflavanoids" "Proanthocyanins" "Color"         "Hue"
[13] "Dilution"       "Proline"
> rownames(wine)
[1] "1"   "2"   "3"   "4"   "5"   "6"   "7"   "8"   "9"   "10"  "11"
"12"
[13] "13"  "14"  "15"  "16"  "17"  "18"  "19"  "20"  "21"  "22"  "23"
"24"
[...]
[157] "157" "158" "159" "160" "161" "162" "163" "164" "165" "166" "167" "168"
[169] "169" "170" "171" "172" "173" "174" "175" "176" "177" "178"
```

This gives us an idea of the shape of the data. We are dealing with a relatively small dataset of 178 entities and 14 variables.

Next, we'd like to see what the data itself looks like. We can list the first few rows of the data using *head*:

```
> head(wine)
  Type Alcohol Malic  Ash Alkalinity Magnesium Phenols Flavanoids Nonflavanoids
1    1    14.23  1.71 2.43           15.6      127    2.80      3.06
0.28
2    1    13.20  1.78 2.14           11.2      100    2.65      2.76
0.26
3    1    13.16  2.36 2.67           18.6      101    2.80      3.24
0.30
4    1    14.37  1.95 2.50           16.8      113    3.85      3.49
0.24
5    1    13.24  2.59 2.87           21.0      118    2.80      2.69
0.39
6    1    14.20  1.76 2.45           15.2      112    3.27      3.39
0.34
  Proanthocyanins Color  Hue Dilution Proline
```

1	2.29	5.64	1.04	3.92	1065
2	1.28	4.38	1.05	3.40	1050
3	2.81	5.68	1.03	3.17	1185
4	2.18	7.80	0.86	3.45	1480
5	1.82	4.32	1.04	2.93	735
6	1.97	6.75	1.05	2.85	1450

Next we might look at the structure of the data using the *str* (structure) function. This provides a basic overview of both values and their data type:

```
> str(wine)
'data.frame': 178 obs. of 14 variables:
 $ Type      : Factor w/ 3 levels "1","2","3": 1 1 1 1 1 1 1 1 1 1 ...
 $ Alcohol   : num  14.2 13.2 13.2 14.4 13.2 ...
 $ Malic     : num  1.71 1.78 2.36 1.95 2.59 1.76 1.87 2.15 1.64 1.35 ...
 $ Ash       : num  2.43 2.14 2.67 2.5 2.87 2.45 2.45 2.61 2.17 2.27 ...
 $ Alcalinity: num  15.6 11.2 18.6 16.8 21 15.2 14.6 17.6 14 16 ...
 $ Magnesium: int  127 100 101 113 118 112 96 121 97 98 ...
 $ Phenols   : num  2.8 2.65 2.8 3.85 2.8 3.27 2.5 2.6 2.8 2.98 ...
 $ Flavanoids: num  3.06 2.76 3.24 3.49 2.69 3.39 2.52 2.51 2.98 3.15 ...
 $ Nonflavanoids: num  0.28 0.26 0.3 0.24 0.39 0.34 0.3 0.31 0.29 0.22 ...
 $ Proanthocyanins: num  2.29 1.28 2.81 2.18 1.82 1.97 1.98 1.25 1.98 1.85 ...
 $ Color     : num  5.64 4.38 5.68 7.8 4.32 6.75 5.25 5.05 5.2 7.22 ...
 $ Hue       : num  1.04 1.05 1.03 0.86 1.04 1.05 1.02 1.06 1.08 1.01 ...
 $ Dilution  : num  3.92 3.4 3.17 3.45 2.93 2.85 3.58 3.58 2.85 3.55 ...
 $ Proline   : int  1065 1050 1185 1480 735 1450 1290 1295 1045 1045 ...
```

We are now starting to get an idea of what the data itself looks like. The categorical variable *Type* would appear to be something that we might want to model—the output variable. The remaining variables are all numeric variables, a mixture of integers and real numbers.

The final step in the first look at the data is to get a summary of each variable using *summary*:

```
> summary(wine)
    Type      Alcohol      Malic      Ash      Alcalinity 
 1:59   Min.   :11.03   Min.   :0.740   Min.   :1.360   Min.   :10.60 
 2:71   1st Qu.:12.36   1st Qu.:1.603   1st Qu.:2.210   1st Qu.:17.20 
 3:48   Median :13.05   Median :1.865   Median :2.360   Median :19.50 
          Mean   :13.00   Mean   :2.336   Mean   :2.367   Mean   :19.49 
          3rd Qu.:13.68   3rd Qu.:3.083   3rd Qu.:2.558   3rd Qu.:21.50 
          Max.   :14.83   Max.   :5.800   Max.   :3.230   Max.   :30.00 
 [...]
```

16.7.1 Stem and Leaf Plots

A **Stem-and-leaf** plot is a simple textual plot of numeric data that is useful to get an idea of the shape of a distribution. It is similar to the graphic histograms that we will see next, but a useful quick place to start for smaller datasets. A stem-and-leaf plot has the advantage of showing actual data values in the plot rather than just a bar indicating frequency.

In reviewing a stem-and-leaf plot we might look to see if there is a clear central value, or whether the data is very spread out. We look at the spread to see if it might be symmetric about the central value or whether there is a skew in one particular direction. We might also look for any data values that are a long way from the general values in the rest of the population.

```
> stem(wine$Magnesium)

The decimal point is 1 digit(s) to the right of the |

 7 | 0
 7 | 888
 8 | 0000012444
 8 | 55555566666666667778888888888899999
 9 | 0000112222233444444
 9 | 55566666666677778888888889
10 | 0001111111122222233333444
10 | 55666677778888
11 | 0001112222233
11 | 5566678889
12 | 0001234
12 | 678
13 | 24
13 | 69
14 |
14 |
15 | 1
15 |
16 | 2
```

The stem is to the left of the bar and the leaves are to the right.

Note the change in where the decimal point is.

```
> stem(wine$Alcohol)

The decimal point is 1 digit(s) to the left of the |

110 | 3
112 |
114 | 1566
```

```
116 | 1245669
118 | 1224476
120 | 000478888867
122 | 01255599993346777777
124 | 22235711238
126 | 004790022779
128 | 124556783369
130 | 355555578116677
132 | 034478902469
134 | 0015889900126688
136 | 2347891123345678
138 | 23346678804
140 | 266002369
142 | 01223047889
144 |
146 | 5
148 | 3
```

Togaware Watermark For Data Mining Survival

16.7.2 Histogram

A histogram allows the basic distribution of the data to be viewed. Here we plot the histogram for magnesium and alcohol content of various wines, and we might compare it with the previous stem-and-leaf plot which summarises the same data. The shape is basically the same, although in detail they go up and down at different points!

```
attach(wine)
par(mfrow=c(1, 2))
hist(Magnesium)
hist(Alcohol)
```

R code source: [rplot-wine-hist.R](#).

Togaware Watermark For Data Mining Survival

16.7.3 Barplot

A barplot displays data as bars, each bar being proportional to the data being plotted. In R a barplot is built using the *barplot* function. We can use a barplot, for example, to illustrate the distribution of entities in a dataset across some variable. With the *wine* dataset Type is a categorical variable with three levels: 1, 2, and 3. A simple bar plot illustrates the distribution of the entities across the three Types. The *summary* function is used to obtain the data we wish to plot (59, 71, and 48).

We place the actual counts on the plot with the *text* function. The trick here is that the *barplot* function returns the bar midpoints, and these can be used to place the actual values. We add 2 to the *y* values to place the numbers above the bars. Also note that *xpd* is set to TRUE to avoid the highest number being chopped (because it, 71, is actually outside the plot region)

```
plot(wine)
load("wine.Rdata")
attach(wine)
par(xpd=TRUE)
bp <- barplot(summary(Type), xlab="Type", ylab="Frequency")
text(bp, summary(Type)+2, summary(Type))
```

R code source: [rplot-wine-barplot.R](#).

16.7.4 Density Plot

```
plot(density(iris$Petal.Length))
```

R code source: [rplot-iris-density.R](#).

16.7.5 Basic Histogram

A histogram illustrates the distribution of values. The following example is the most basic of histograms.

```
pdf("graphics/rplot-hist.pdf")
hist(rnorm(200))
dev.off()
```

R code source: [rplot-hist.R](#).

Togaware Watermark For Data Mining Survival

16.7.6 Basic Histogram with Density Curve

R allows plots to be built up—this example shows a density histogram of a set of random numbers extracted from a normal distribution with the density curve of the same normal distribution also displayed. In the R code we build the histogram at first without plotting it, so as to determine the y limits (*range* selects the minimum and maximum values, while *h\$density* is the list of density values being plotted and *dnorm(0)* is the maximum possible value of the density), since otherwise the curve might push up into the title!

```
ds <- rnorm(200)
pdf("graphics/rplot-hist-density.pdf")
par(xpd=T)
h <- hist(ds, plot=F)
ylim <- range(0, h$density, dnorm(0))
hist(ds, xlab="normal", ylim=ylim, freq=F,
      main="Histogram of Normal Distribution with Density")
curve(dnorm, col=2, add=T)
dev.off()
```

R code source: [rplot-hist-density.R](#).

16.7.7 Practical Histogram

Suppose we are interested in the distribution of the Alcohol content in the *wine* dataset. The numeric values are grouped by *hist* into intervals and the bars represent the frequency of occurrence of each interval as a height. A *rug* is added to the plot, just above the x-axis, to illustrate the density of values.

```
pdf('graphics/rplot-hist-colour.pdf')
load("wine.Rdata")
attach(wine)
hist(Alcohol, col='lightgreen')
rug(Alcohol)
dev.off()
```

R code source: [rplot-hist-colour.R](#).

16.7.8 Correlation Plot

A [correlation](#) measures how two variables are related and is useful for measuring the association between the two variables. A correlation plot shows the strength of any linear relationship between a pair of variables. The *ellipse* package provides the *plotcorr* function for this purpose. Linear relationships between variables indicate that as the value of one variable changes, so does the value of another. The degree of correlation is measured between $[-1, 1]$ with 1 being perfect correlation and 0 being no correlation. The Pearson correlation coefficient is the common statistic and R also supports Kendall's tau and Spearman's rho statistics for rank-based measures of association, which are regarded as being more robust and recommended other than for a bivariate normal distribution. The *cor* function is used to calculate the correlation matrix between variables in a numeric vector, matrix or data frame. A matrix is always symmetric about the diagonal, and the diagonal consists of 1s (each variable is perfectly correlated with itself!).

The sample R code here generates the correlations for variables in the *wine* dataset (*cor*) and then orders the variables according to their correlation with the first variable (Type: `[1,]`). This is sorted and ellipses are printed with colour fill using *cm.colors*.

```
library(ellipse)
wine.corr <- cor(wine)
ord <- order(wine.corr[1,])
xc <- wine.corr[ord, ord]
plotcorr(xc, col=cm.colors(11)[5*xc + 6])
```

R code source: [rplot-wine-corr.R](#).

The correlation matrix is:

```
> wine.corr
          Type      Alcohol       Malic        Ash    Alcalinity
Type 1.000000000 -0.32822194  0.43777620 -0.049643221  0.51785911
Alcohol -0.32822194  1.000000000  0.09439694  0.211544596 -0.31023514
Malic   0.43777620  0.09439694  1.000000000  0.164045470  0.28850040
Ash    -0.049643221  0.21154460  0.164045471  1.000000000  0.44336719
Alcalinity  0.51785911 -0.31023514  0.28850040  0.443367187  1.000000000
Magnesium -0.20917939  0.27079823 -0.05457510  0.286586691 -0.08333309
Phenols   -0.71916334  0.28910112 -0.33516700  0.128979538 -0.32111332
Flavanoids -0.84749754  0.23681493 -0.41100659  0.115077279 -0.35136986
Nonflavanoids  0.48910916 -0.15592947  0.29297713  0.186230446  0.36192172
Proanthocyanins -0.49912982  0.13669791 -0.22074619  0.009651935 -0.19732684
Color     0.26566757  0.54636420  0.24898534  0.258887259  0.01873198
Hue      -0.61736921 -0.07174720 -0.56129569 -0.074666889 -0.27395522
Dilution  -0.78822959  0.07234319 -0.36871043  0.003911231 -0.27676855
Proline   -0.63371678  0.64372004 -0.19201056  0.223626264 -0.44059693

          Magnesium      Phenols  Flavanoids Nonflavanoids
Type -0.20917939 -0.71916334 -0.8474975  0.4891092
Alcohol 0.27079823 0.28910112 0.2368149 -0.1559295
Malic  -0.05457510 -0.33516700 -0.4110066  0.2929771
Ash   0.28658669  0.12897954  0.1150773  0.1862304
Alcalinity -0.08333309 -0.32111332 -0.3513699  0.3619217
Magnesium 1.000000000 0.21440123 0.1957838 -0.2562940
Phenols   0.21440123 1.000000000 0.8645635 -0.4499353
Flavanoids 0.19578377 0.86456350 1.0000000 -0.5378996
Nonflavanoids -0.25629405 -0.44993530 -0.5378996 1.0000000
Proanthocyanins 0.23644061 0.61241308 0.6526918 -0.3658451
Color    0.19995001 -0.05513642 -0.1723794 0.1390570
Hue     0.05539820  0.43368134  0.5434786 -0.2626396
Dilution 0.06600394  0.69994936  0.7871939 -0.5032696
Proline  0.39335085  0.49811488  0.4941931 -0.3113852

          Proanthocyanins      Color        Hue      Dilution
Proline -0.499129824  0.26566757 -0.61736921 -0.788229589 -0.6337168
Type    0.136697912  0.54636420 -0.07174720  0.072343187
Alcohol 0.6437200   0.24898534 -0.56129569 -0.368710428 -0.1920106
Malic   -0.220746187  0.236440610 0.19995001 0.05539820 0.066003936
Ash    0.009651935  0.25888726 -0.074666889 0.003911231
Alcalinity 0.236440610 0.19995001 0.05539820 0.066003936
Magnesium 0.39335085  0.612413084 -0.05513642 0.43368134 0.699949365
Phenols  0.49811488  0.4941931  0.43368134 0.699949365
0.39335085
```

Flavanoids	0.652691769	-0.17237940	0.54347857	0.787193902	
0.4941931					
Nonflavanoids	-0.365845099	0.13905701	-0.26263963	-0.503269596	-0.3113852
Proanthocyanins	1.000000000	-0.02524993	0.29554425	0.519067096	
0.3304167					
Color	-0.025249931	1.000000000	-0.52181319	-0.428814942	
0.3161001					
Hue	0.295544253	-0.52181319	1.00000000	0.565468293	
0.2361834					
Dilution	0.519067096	-0.42881494	0.56546829	1.000000000	
0.3127611					
Proline	0.330416700	0.31610011	0.23618345	0.312761075	
1.0000000					

Togaware Watermark For Data Mining Survival

16.7.9 Colourful Correlations

You could write your own path.colors as below and obtain a more colourful correlation plot. The colours are quite garish but it gives an idea of what is possible—The reds and purples give a good indication of high correlation (negative and positive), while the blues and greens identify less correlation.

Togaware Watermark For Data Mining Survival

```

# Suggested by Duncan Murdoch
path.colors <- function(n, path=c('cyan', 'white', 'magenta'),
                        interp=c('rgb', 'hsv'))
{
  interp <- match.arg(interp)
  path <- col2rgb(path)
  nin <- ncol(path)
  if (interp == 'hsv')
  {
    path <- rgb2hsv(path)
    # Modify the interpolation so that the circular nature of hue
    for (i in 2:nin)
      path[1,i] <- path[1,i] + round(path[1,i-1]-path[1,i])
    result <- apply(path, 1, function(x) approx(seq(0, 1,
                                                len=nin), x, seq(0, 1, len=n))$y)
    return(hsv(result[,1] %% 1, result[,2], result[,3]))
  }
  else
  {
    result <- apply(path, 1, function(x) approx(seq(0, 1,
                                                len=nin), x, seq(0, 1, len=n))$y)
    return(rgb(result[,1]/255, result[,2]/255, result[,3]/255))
  }
}

pdf('graphics/rplot-corr-wine.pdf')
library(ellipse)
load('wine.Rdata')
corr.wine <- cor(wine)
ord <- order(corr.wine[1,])
xc <- corr.wine[ord, ord]
plotcorr(xc, col=path.colors(11,
                            c("red", "green", "blue", "red"),
                            interp="hsv")[5*xc + 6])
dev.off()

```

R code source: [rplot-corr-wine.R](#).

16.8 Measuring Data Distributions

We now start to explore how the data in each of the variables is distributed. This might be as simple as looking at the spread of the numeric values, or the number of entities having a specific value for a variable. Another aspect involves measuring the central tendency of data, or determining the [mean](#) and [median](#). Yet another is a measure of the spread or [variance](#) of the data from this central tendency. We again begin with textual presentations of the distributions, and then graphical presentations.

16.8.1 Textual Summaries

The *summary* function provides the first insight into how the values for each variable are distributed:

```
> summary(wine)

  Type      Alcohol        Malic        Ash       Alcalinity
1:59   Min. :11.03   Min. :0.740   Min. :1.360   Min. :10.60
2:71   1st Qu.:12.36  1st Qu.:1.603  1st Qu.:2.210  1st Qu.:17.20
3:48   Median :13.05  Median :1.865  Median :2.360  Median :19.50
        Mean   :13.00  Mean   :2.336  Mean   :2.367  Mean   :19.49
        3rd Qu.:13.68  3rd Qu.:3.083  3rd Qu.:2.558  3rd Qu.:21.50
        Max.   :14.83  Max.   :5.800  Max.   :3.230  Max.   :30.00

  Magnesium      Phenols      Flavanoids Nonflavanoids
Min.   : 70.00  Min.   :0.980  Min.   :0.340  Min.   :0.1300
1st Qu.: 88.00  1st Qu.:1.742  1st Qu.:1.205  1st Qu.:0.2700
Median  : 98.00  Median :2.355  Median :2.135  Median :0.3400
Mean    : 99.74  Mean   :2.295  Mean   :2.029  Mean   :0.3619
3rd Qu.:107.00  3rd Qu.:2.800  3rd Qu.:2.875  3rd Qu.:0.4375
Max.   :162.00  Max.   :3.880  Max.   :5.080  Max.   :0.6600

  Proanthocyanins      Color          Hue      Dilution
Min.   : 0.410  Min.   : 1.280  Min.   :0.4800  Min.   :1.270
1st Qu.:1.250  1st Qu.: 3.220  1st Qu.:0.7825  1st Qu.:1.938
Median  :1.555  Median : 4.690  Median :0.9650  Median :2.780
Mean    :1.591  Mean   : 5.058  Mean   :0.9574  Mean   :2.612
3rd Qu.:1.950  3rd Qu.: 6.200  3rd Qu.:1.1200  3rd Qu.:3.170
Max.   :3.580  Max.   :13.000  Max.   :1.7100  Max.   :4.000

  Proline
Min.   : 278.0
1st Qu.: 500.5
Median  : 673.5
Mean    : 746.9
3rd Qu.: 985.0
Max.   :1680.0
```

Next, we would like to know how the data is distributed. For categorical variables this will be how many of each level there are. For numeric variables this will be the mean and median, the minimum and maximum values, and an idea of the spread of the values of the variable.

We would also like to know about missing values (referred to in R as NAs—short for Not Available), and the *summary* function will also report this:

```
> load("survey.RData")
> summary(survey)
[...]
```

```

Native.Country  Salary.Group
United-States:29170  <=50K:24720
Mexico        : 643    >50K : 7841
Philippines   : 198
Germany       : 137
Canada         : 121
(Other)        : 1709
NA's          : 583

```

We also see here that the categorical variable `Native.Country` has more than five levels, and there are 1,709 entities with values for this variable other than the five listed here. The five listed are the most frequently occurring.

The `mean` provides a measure of the average or central tendency of the data. It is denoted as μ if x_1, \dots, x_n is the whole population (*population mean*), and \bar{X} if it is a sample of the population (*sample mean*).

In calculating the `mean` of a sample from a population we generally need at least 30 observations in the sample before it makes sense. This is based on the central limit theorem that indicates that for $n = 30$ the shape of a distribution approaches normal.

R provides the `mean` function to calculate the mean. The mean is also reported as part of the output from `summary`. The `summary` function in fact will use the method associated with the data type of the object passed. For example, if it is a data frame the function `summary.data.frame` will be called upon. To see the actual function definition, simply type the function name at the command line (without brackets). The actual code will be printed out. A user can then fine tune the function, if desired.

A quick trick to roughly get the mode of a dataset is to use the density.

```

mode <- function (n)
{
  n <- as.numeric(n)
  n.density <- density(n)
  round(n.density$x[which(n.density$y==max(n.density$y))])
}

```

You can then simply write your own functions to summarise the data:

```

> sapply(wine,
         function(x)
         {
           x <- as.numeric(x)

```

```

res <- c(mean(x), median(x), mode(x), mad(x), sd(x))
names(res) <- c("mean", "median", "mode", "mad", "sd")
res
}

      Type    Alcohol     Malic      Ash Alkalinity Magnesium Phenols
mean   1.938202 13.0006180 2.336348 2.366517 19.494944 99.74157 2.295112
median 2.000000 13.0500000 1.865000 2.360000 19.500000 98.00000 2.355000
mode   2.000000 14.0000000 2.000000 2.000000 19.000000 90.00000 3.000000
mad    1.482600 1.0081680 0.770952 0.237216 3.039330 14.82600 0.748713
sd     0.775035 0.8118265 1.117146 0.274344 3.339564 14.28248 0.625851

      Flavanoids Nonflavanoids Proanthocyanins Color      Hue
Dilution
mean   2.0292697      0.3618539      1.5908989 5.058090 0.9574494 2.6116854
median 2.1350000      0.3400000      1.5550000 4.690000 0.9650000 2.7800000
mode   3.0000000      0.0000000      1.0000000 3.000000 1.0000000 3.0000000
mad    1.2379710      0.1260210      0.5633880 2.238726 0.2446290 0.7709520
sd     0.9988587      0.1244533      0.5723589 2.318286 0.2285716 0.7099904

      Proline
mean   746.8933
median 673.5000
mode   553.0000
mad    300.2265
sd     314.9075

```

In the following sections we provide graphic presentations of the mean and standard variation.

16.8.2 Boxplot

A **boxplot** (Tukey, 1977) (also known as a box-and-whisker plot) provides a graphical overview of how data is distributed over the number line. R's *boxplot* function displays a graphical representation of the textual *summary* of data. The skewness of the distribution of the data becomes clear.

A boxplot shows the **median** (the second **quartile** or the 50th **percentile**) as the thicker line within the box ($Ash = 2.36$). The top and bottom extents of the box (2.558 and 2.210 respectively) identify the upper quartile (the third quartile or the 75th percentile) and the lower quartile (the first quartile and the 25th percentile). The extent of the box is known as the **interquartile range** ($2.558 - 2.210 = 0.348$). The dashed lines extend to the maximum and minimum data points that are no more than 1.5 times the interquartile range from the median. Outliers (points further than 1.5 times the interquartile range from the median) are then individually plotted (at 3.23, 3.22, and 1.36). Our plot here adds faint horizontal lines to more easily read off the various values.

```
load("wine.Rdata")
attach(wine)
boxplot(Ash, xlab="Ash")
abline(h=seq(1.4, 3.2, 0.1), col="lightgray", lty="dotted")
```

R code source: [rplot-wine-boxplot-single.R](#).

Multiple Boxplots

The default *boxplot* function in fact will plot multiple boxplots.

By comparing a number of variables we can see that some have quite a bit more spread than others, and their medians have different relative positions within the box.

We include the code here to generate a PDF version of the plot primarily to demonstrate how we can increase the width of the plot for a more pleasing presentation.

We could have presented the plot horizontally by setting the *horizontal* option to TRUE.

```
Togaware Watermark For Data Mining Survival
pdf("graphics/rplot-wine-boxplot-multi.pdf", width=9)
load("wine.Rdata")
boxplot(wine[,c(3,4,7,8,10,13)])
dev.off()
```

R code source: [rplot-wine-boxplot-multi.R](#).

Boxplot by Class

With a *boxplot* it is often useful to display the distribution of one variable as it relates to some other variable. An example in the wine data would be to partition the data according to the `Type`, and then to explore the resulting distribution of, for example, `Malic`. This is achieved with the formula notation `Malic ~ Type`. The boxplot then allows us to understand any potential relationship between the input variable and the output variable. For such plots we enable the notch display, which indicates whether there is a significant difference between the medians. In the case here the median for `Type 3` is significantly different from the other two, but the other two are not significantly different from each other.

```
Togaware Watermark For Data Mining Survival
load("wine.Rdata")
attach(wine)
boxplot(Malic ~ Type, notch=TRUE, xlab="Type", ylab="Malic")
```

R code source: [rplot-wine-boxplot-type.R](#).

16.8.3 Box and Whisker Plot

A simple box plot of randomly generated data. The box in each plot shows the median (as the line within the box) and one standard deviation from the mean (the extremes of the box). The whiskers show the second standard deviation, and the circles show outliers.

In this code we use R's *rnorm* function to generate a standard_normal random matrix of the given shape (rows by cols). The dataset is transformed to an R data frame. The x axis is labelled with an appropriate

```
set.seed(2)
ds <- matrix(rnorm(19 * 100), ncol = 19)
pdf("graphics/rplot-boxplot.pdf")
plot.new()
plot.window(xlim = c(0, 20), ylim = range(ds), xaxs = "i")
boxplot(as.data.frame(ds), add = TRUE, at = 1:19)
dev.off()
```

R code source: [rplot-boxplot.R](#).

16.8.4 Box and Whisker Plot: With Means

The simple box plot is extended to include the mean of each column. The means are calculated using `scipy`, and plotted as a + (`pch=3`).

```
rows <- 100
cols <- 17
pdf("graphics/rplot-boxplot-means.pdf")
dataset <- matrix(rnorm(rows*cols), nrow=rows, ncol=cols)
means <- mean(as.data.frame(dataset))
boxplot(as.data.frame(dataset), notch=TRUE, at=1:17,
        xlim=c(0, cols+1), ylim=c(-4,4), xaxt='n')
axis(side=1, at=1:17, labels=letters[1:17])
points(1:17, means, pch=3)
dev.off()
```

R code source: [rplot-boxplot-means.R](#).

Togaware Watermark For Data Mining Survival

16.8.5 Clustered Box Plot

```
pdf("graphics/rplot-bwplot.pdf")
mydata <- data.frame(Y = rnorm(3*1000),
                      INDFACT = rep(c("A", "B", "C"), each=1000),
                      CLUSFACT=factor(rep(c("M","F"), 1500)))
library(lattice)
trellis.device(new=FALSE, col=FALSE)
bwplot(Y ~ INDFACT | CLUSFACT, data=mydata, layout=c(2,1))
dev.off()
```

R code source: [rplot-bwplot.R](#).

Togaware Watermark For Data Mining Survival

16.9 Further Resources

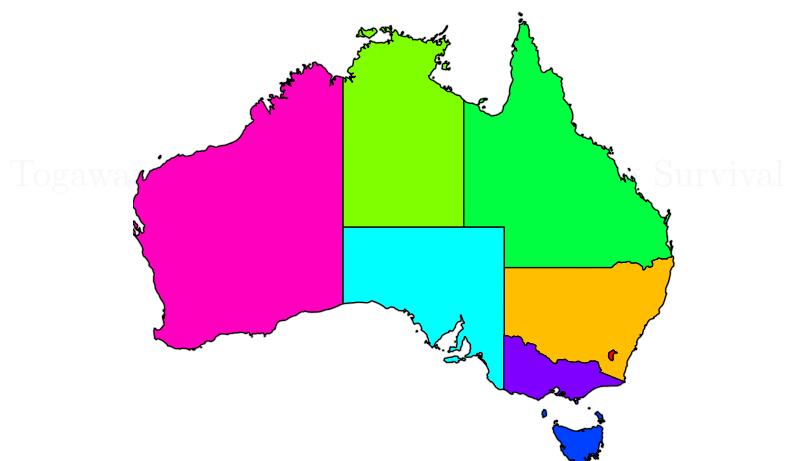
Useful on-line statistical resources include:

- Wikipedia has a growing collection of statistical entries.
- NIST
- StatSoft
- Tufts
- R Graph Gallery

Togaware Watermark For Data Mining Survival

16.10 Map Displays

Map displays often provide further insights into patterns that vary according to region. In this plot we illustrate some basic colouring of a map. This can be used to translate a variable into a colour which is then plotted for the state. We use the *rainbow* function to generate the bright colours to use to colour the states.



```
library(maptools)
aus <- readShapePoly("australia.shp")
plot(aus, lwd=2, border="grey", xlim=c(115,155), ylim=c(-35,-20))
colours <- rainbow(8)
# Must be a better way than this....
nsw <- aus; nsw@plotOrder <- as.integer(c(2)); plot(nsw,col=colours[2],add=TRUE)
act <- aus; act@plotOrder <- as.integer(c(1)); plot(act,col=colours[1],add=TRUE)
nt <- aus; nt@plotOrder <- as.integer(c(3)); plot(nt, col=colours[3],add=TRUE)
qld <- aus; qld@plotOrder <- as.integer(c(4)); plot(qld,col=colours[4],add=TRUE)
sa <- aus; sa@plotOrder <- as.integer(c(5)); plot(sa, col=colours[5],add=TRUE)
tas <- aus; tas@plotOrder <- as.integer(c(6)); plot(tas,col=colours[6],add=TRUE)
vic <- aus; vic@plotOrder <- as.integer(c(7)); plot(vic,col=colours[7],add=TRUE)
```

R code source: [map-australia-states.R](#).

16.11 Further Resources

Useful on-line statistical resources include:

- [Wikipedia](#) has a growing collection of statistical entries.
- [NIST](#)
- [StatSoft](#)
- [Tufts](#)
- [R Graph Gallery](#)

Togaware Watermark For Data Mining Survival

Chapter 17

Preparing Data

Data is fundamental to data mining, but quality data is fundamental to quality data mining.^{17.1} The data preparation step in a data mining project involves assessing and improving the data quality—transforming and cleaning and subsetting the data to suit to requirements of the data mining task. In this chapter we explore the process of transforming a data source into a dataset ready for mining.

17.1 Data Selection and Extraction

17.1.1 Training and Test Datasets

Often in modelling we build our model on a training set and then test its performance on a test set. The simplest approach to generating a partitioning of your dataset into a training and test set is with the *sample* function:

```
> sub <- sample(nrow(iris), floor(nrow(iris) * 0.8))
> iris.train <- iris[sub, ]
> iris.test <- iris[-sub, ]
```

The first argument to *sample* is the top of the range of integers you wish to choose from, and the second is the number to choose.

The `sample.split` function of the `caTools` package also comes in handy here. It will split a vector into two subsets, two thirds in one and one third in the other, maintaining the relative ratio of the different categorical values represented in the vector:

```
> mask <- sample.split(iris$Species)
> mask
[1] FALSE  TRUE  TRUE FALSE  TRUE  TRUE FALSE FALSE FALSE  TRUE  TRUE FALSE
[...]
[145]  TRUE  TRUE  TRUE  TRUE FALSE  TRUE
> table(iris$Species)

  setosa versicolor  virginica
  50          50          50
> table(iris$Species[mask])

  setosa versicolor  virginica
  33          33          33
> table(iris$Species[!mask])

  setosa versicolor  virginica
  17          17          17
```

Togaware Watermark For Data Mining Survival

17.2 Data Cleaning

Data cleaning deals with issues of removing errant transactions, updating transactions to account for reversals, elimination of missing data, and so on.

The aim of data cleaning is to raise the data quality to a level suitable for the selected analyses.

The data cleaning to be performed depends on purpose to which the data is to be put. Some activities will require a selection of data cleaning and data transformation modules to be applied to the data.

Data cleaning occurs early in the process and then continually throughout the process as we learn more about the data.

Field selection

Sampling

Data correction

Missing values treatment

Data transformation, e.g., birth date to age.

Derive new fields

Useful steps:

Understand the business problem.

Collect the materials about the data sources and study them to understand what data is available.

Identify the data items relevant to the business problem, e.g., tables and attributes.

Make a data extraction plan and arrange the data extraction (with DBAs).

Calculate the summary statistics of the extracted data. *Survival*

Review Data

Often we will find ourselves loading data from a CSV file which is readily supported by R (Section 14.3.4, page 218). On the first loading of the data we generally want to get a quick summary, using R's *summary* function. It is here that we might note that some numeric columns have become factors!

Consider the example of the cardiac dataset (Section 14.3.4, page 220).

```
> cardiac <- read.csv("cardiac.data", header=F)
> summary(cardiac)
[...]
      V10          V11          V12          V13          V14
Min.   :-172.00    52   : 13   60   : 23   49   :  9   ?
:376
1st Qu.:  3.75    36   : 10   ?    : 22   55   :  9   84
: 3
Median  : 40.00    42   :  9   61   : 16   59   :  9   -157
: 2
Mean    : 33.68    10   :  8   56   : 14   62   :  9   -164
: 2
3rd Qu.: 66.00    33   :  8   58   : 13   26   :  8   -93
: 2
Max.   : 169.00   41   :  8   68   : 12   33   :  8   103
: 2
```

```
(Other):396    (Other):352    (Other):400    (Other): 65
[...]
```

Our understanding of the data might be that we expect these variables to be numeric. Indeed, the telltale sign is V14 having a ? as one of its values. A little more exploration to show the frequency of each value will indicate that the apparently nominal variables only have a single non-numeric value, the ?. When we read the data from the CSV file we need to tell R that the ? is used to indicate missing values

```
> cardiac <- read.csv("cardiac.data", header=F, na.string="?")
> summary(cardiac)
[...]
   V11          V12          V13          V14
Min. :-177.00  Min. :-170.00  Min. :-135.00  Min. :-179.00
1st Qu.: 14.00  1st Qu.: 41.00  1st Qu.: 12.00  1st Qu.:-124.50
Median : 41.00  Median : 56.00  Median : 40.00  Median : -50.50
Mean   : 36.15  Mean   : 48.91  Mean   : 36.72  Mean   : -13.59
3rd Qu.: 63.25  3rd Qu.: 65.00  3rd Qu.: 62.00  3rd Qu.: 117.25
Max.   : 179.00  Max.   : 176.00  Max.   : 166.00  Max.   : 178.00
NA's   : 8.00    NA's   : 22.00  NA's   : 1.00   NA's   : 376.00
[...]
```

That's looking better. Note that the NAs are reported and that V14 has 376 of them, in accord with the previous observation of 376 ?'s.

Removing Duplicates

The function *duplicated* identifies elements of a data structure that are duplicated:

```
> x <- c(1, 1, 1, 2, 2, 2, 3, 3, 3, 3)
> duplicated(x)
[1] FALSE  TRUE  FALSE  TRUE  TRUE  FALSE  TRUE  TRUE  TRUE
> x <- x[!duplicated(x)]
> x
[1] 1 2 3
>
```

This is a simple example, but works just as well to remove duplicated rows from a matrix or data frame.

Selectively Changing Vector Values

The next example changes the values in one vector (`weights`) according to some conditions on the values in another vector (`data`). The `data` vector is randomly sampled from the *letters* of the alphabet. Both vectors are the same length. Where `data` is larger than `m`, the weight is set to 2. Where it is between `d` and `m`, the weight is set to 3.

```
> weights <- rep(1, 10)
> data <- letters[sample(seq(1,length(letters)), 10)]
> data
[1] "y" "b" "j" "m" "c" "q" "o" "a" "i" "p"
> weights[data > "m"] <- 2
> weights
[1] 2 1 1 1 1 2 2 1 1 2
> weights[data <= "m" & data >= "d"] <- 3
> weights
[1] 2 1 3 3 1 2 2 1 3 2
```

An example of where this might be useful is in data mining pre-processing where we wish to selectively change the weights associated with entities in a modelling exercise. The weights might indicate the relative important of the specific entities. An example of this transformation is included in the usage of `rpart` in Chapter ??, page ??.

Replace Indices By Names

```
> city.name <- c("Munich", "Paris", "Tokyo", "London", "Boston")
> X <- cbind(c(2, 5, 5), c(4, 1, 3))
> X
     [,1] [,2]
[1,]    2    4
[2,]    5    1
[3,]    5    3
> matrix(city.name[X], ncol = 2)
     [,1]      [,2]
[1,] "Paris"   "London"
[2,] "Boston"  "Munich"
[3,] "Boston"  "Tokyo"
```

Missing Values

Missing data can affect modelling, particularly if the data is not randomly missing, but missing because of some underlying systematic reason (e.g., censoring). If data is missing at random (often abbreviated as MAR)

then it is more likely that the missing values will have little affect on the modelling.

An excellent reference on dealing with missing data is Schafer (1997).

Missing values are specially recorded in R as `NA`. Various functions can be used to check for a missing value (`is.na`), to remove any entities with missing values (`na.omit` and to identify those entities that are complete (`complete.cases`). The `apply` function also comes in handy here.

```
> ds <- ds[!apply(is.na(ds),1,all),]      # Remove all rows of all NA's.
> ds <- na.omit(ds)                      # Remove all rows that have any NA's.
> ds <- ds[complete.cases(ds),]          # Remove all rows that have any NA's.
```

In some very simple (i.e., not rigorous) timing experiments the second of these using `complete.cases` is faster.

Remove Levels from a Factor

Copyright © 2006-2008 Graham Williams

Some tools will attempt to model all levels in an output variable. If there are no entities in the dataset with a value for one of the levels of an output variable, the tool will fail (e.g., `randomForest`). To remove unused levels from a factor:

```
> dataset$Target <- dataset$Target[,drop=TRUE]
```

Removing Outliers

Tests for outliers have primarily been superseded by the use of robust methods. Outlier tests are poor in that outliers tend to damage results long before they are detected. Robust methods attempt to compensate rather than reject outliers. RandomForrest modelling helps avoid the issue of outliers.

You can get a list of what the `boxplot` function thinks are outliers:

```
> load("wine.RData")
> bp <- boxplot(wine$Ash, plot=FALSE)
> bp$out
[1] 3.22 1.36 3.23
```

17.2.1 Variable Manipulations

Remove Columns

```
> names(ds)                                # List the column names
> ds$fred <- NULL                         # Removes the column
> ds <- subset(ds, select=-c(tom, jerry)) # Remove multiple columns
```

Reorder Columns

```
> ds <- ds[,c(1,2,5,6,3,4,7,8)]          # Columns will be reordered
```

Remove Non-Numeric Columns

We might only be interested in the numeric data, so we remove all columns that are not numeric from a dataset. We can use the `survey` dataset to illustrate this. First load the dataset and have a look at the column names and their types. We use the *lapply* function to apply the `class` function to each column of the data frame.

```
> load("survey.RData")
> colnames(survey)
[1] "Age"           "Workclass"      "fnlwgt"        "Education"
[5] "Education.Num" "Marital.Status" "Occupation"    "Relationship"
[9] "Race"          "Sex"          "Capital.Gain" "Capital.Loss"
[13] "Hours.Per.Week" "Native.Country" "Salary.Group"
> lapply(survey, class)
$Age
[1] "integer"

$Workclass
[1] "factor"

$fnlwgt
[1] "integer"

$Education
[1] "factor"

$Education.Num
[1] "integer"

$Marital.Status
[1] "factor"

$Occupation
[1] "factor"

\$Relationship
```

```
[1] "factor"
$Race
[1] "factor"
$Sex
[1] "factor"
$Capital.Gain
[1] "integer"
$Capital.Loss
[1] "integer"
$Hours.Per.Week
[1] "integer"
$Native.Country
[1] "factor"
$Salary.Group
[1] "factor"
```

We can now simply use *is.numeric* to select the numeric columns and store the result in a new dataset, using *sapply* to extract the list of numeric columns:

```
> survey.numeric <- survey[, sapply(survey, is.numeric)]
```

You could instead build a list of the columns to remove and then explicitly remove them from the dataset in place, so that you don't create a need for extra data storage.

First build a numeric list of columns to remove, and reverse it since after we remove a column, all the remaining columns are shifted left and their index is then one less! We use *sapply* to extract the list of numeric columns (those for which *is.numeric* is true).

```
> rmcols <- rev(seq(1, ncol(survey)))[!as.logical(sapply(survey, is.numeric))]
> rmcols
[1] 15 14 10  9  8  7  6  4  2
```

Now remove the columns from the dataset simply by setting the column to NULL.

```
> for (i in rmcols) survey[[i]] <- NULL
> colnames(survey)
[1] "Age"          "fnlwgt"        "Education.Num"  "Capital.Gain"
[5] "Capital.Loss" "Hours.Per.Week"
```

This same process can be used to remove or retain columns of any type, simply by using the appropriate R function: e.g., *is.factor*, *is.logical*, *is.integer*, or *is.numeric*.

Remove Variables with no Variance

We also only want columns where there is some variance in the values, so also remove those columns with a minimum value equal to the maximum. Again, use is made of *lapply* to apply a function (in this case *max* and *min*) to the data.

```
> rmcols <- as.numeric(lapply(dat, min, na.rm=T)) ==
  as.numeric(lapply(dat, max, na.rm=T))
> rmcols <- rev(seq(1,ncol(dat))[rmcols])
> for (i in rmcols) dat[[i]] <- NULL
> ncol(dat)
[1] 59
```

Togaware Watermark For Data Mining Survival

17.2.2 Cleaning the Wine Dataset

17.2.3 Cleaning the Cardiac Dataset

17.2.4 Cleaning the Survey Dataset

We summarise a number of cleaning operations that might be performed on the *survey* dataset.

Remove entities with null values:

```
> load("survey.RData")
> survey <- na.omit(survey)
> dim(survey)
[1] 30162      15
```

Remove non-numeric data:

```
> load("survey.RData")
> rmcols <- rev(seq(1,ncol(survey))[as.logical(lapply(survey, is.factor))])
> for (i in rmcols) survey[[i]] <- NULL
> dim(survey)
[1] 32561      6
> colnames(survey)
[1] "Age"          "fnlwgt"        "Education.Num"  "Capital.Gain"
[5] "Capital.Loss" "Hours.Per.Week"
```

17.3 Imputation

Multiple imputation (MI) is a general purpose method for handling of missing data. The basic idea is: Impute missing values using an appropriate model that incorporates random variation; Do this m times (often 3-5 times) to obtain m datasets, all with no missing values; Do the intended analysis on each of these datasets; Get the average values of the parameter estimates across the m samples to have a single point estimate; Calculate standard errors by firstly averaging the squared standard errors of the m estimates and calculating the variance of the m parameter estimates across samples, and then combine these in some way.

There are a number of R packages for imputation.

17.3.1 Nearest Neighbours

We might, more reasonably, be more sophisticated and use the average value of the k nearest neighbours, where the neighbours are determined by looking at the other variables (not yet implemented in Rattle).

Another approach to filling in the missing values is to look at the entities that are closest to the entity with a missing value, and to use the values for the missing variable of these nearby neighbours to fill in the missing value for this entity. Refer to [Data Mining With R](#), page 48 and following for example R code to do this.

17.3.2 Multiple Imputation

This is the most accurate method, but is computationally expensive. Worth doing though if you don't want to lose any data, but is not supported directly in Rattle. The R package *mitools* is useful for multiple imputation.

17.4 Data Linking

Linking involves bringing together multiple sources of data and connecting related pieces of information across the multiple sources. Often the linking is performed for a particular summarising or analytic task.

17.4.1 Simple Linking

The *merge* function can be used to join several datasets on common fields. the default behaviour is to join on any columns that the data frames have in common. This is what we demonstrate below.

```
> ds1 <- read.table(file("clipboard"), header=T)
> ds1
  id age gender
1  1   32      M
2  2   45      F
3  3   29      F
> ds2 <- read.table(file("clipboard"), header=T)
> ds2
  id day   x1
1  1    1  0.52
2  1    2  0.72
3  1    3  0.29
4  2    1  0.51
5  2    2  0.18
6  3    2  0.22
7  3    3  0.54
> ds3 <- read.table(file("clipboard"), header=T)
> ds3
  id day   x2
1  1    1  0.34
2  1    2  0.55
3  1    3  0.79
4  2    1  0.12
5  2    2  0.23
6  3    2  0.45
7  3    3  0.56
> merge(ds1, ds2)
  id age gender day   x1
1  1   32      M    1  0.52
2  1   32      M    2  0.72
3  1   32      M    3  0.29
4  2   45      F    1  0.51
5  2   45      F    2  0.18
6  3   29      F    2  0.22
7  3   29      F    3  0.54
> merge(merge(ds1, ds2), ds3)
  id day age gender   x1   x2
```

```

1 1 1 32      M 0.52 0.34
2 1 2 32      M 0.72 0.55
3 1 3 32      M 0.29 0.79
4 2 1 45      F 0.51 0.12
5 2 2 45      F 0.18 0.23
6 3 2 29      F 0.22 0.45
7 3 3 29      F 0.54 0.56

```

17.4.2 Record Linkage

Often data linkage is not so straightforward as linking on common columns. Indeed, the data sources may store data in very different ways and the linking may need to probabilistically match entries that appear to relate to the same entity. This is typified by attempting to match names and addresses from different data sources. The entities we are attempting to match could be businesses, patients, and clients.

A very useful tool to help out in this process is the open source Febrl.

17.5 Data Transformation

17.5.1 Aggregation

Sum of Columns

The *colSums* function is an optimised function for calculating the sums of columns in an array. The example here also illustrates how to format output to make large numbers much easier to read, by including commas. The times can be obtained using the *system.time* function.

```

> A <- matrix(runif(10000000, 0, 100), ncol=10)
> colSums(A)                                # Optimised: 0.1 seconds
> rep(1, nrow(A)) %*% A                     # Slower: 0.3 seconds
> apply(A, 2, sum)                            # Slowest: 0.7 seconds
> format(colSums(A), big.mark=",")          

[1] "49,966,626" "49,968,075" "49,977,689" "50,010,843" "50,038,271"
[6] "49,936,119" "50,027,467" "49,985,741" "50,065,027" "49,985,044"

> colSums(A)

[1] 49966626 49968075 49977689 50010843 50038271 49936119 50027467 49985741

```

```
[9] 50065027 49985044
```

17.5.2 Normalising Data

R's *scale* is used to re-center and re-scale data in a numeric matrix. The re-centering involves subtracting a column's mean from each value in the column. The re-scaling then divides each value by the root-mean-square.

```
> ds <- wine[1:20,c(2,9,14)]
> summary(ds)
  Alcohol      Nonflavanoids      Proline
Min.   :13.16    Min.   :0.1700    Min.   : 735
1st Qu.:13.72   1st Qu.:0.2600   1st Qu.:1061
Median :14.11    Median :0.2950   Median :1280
Mean   :14.01    Mean   :0.2970   Mean   :1235
3rd Qu.:14.32   3rd Qu.:0.3225   3rd Qu.:1352
Max.   :14.83    Max.   :0.4300   Max.   :1680
> ds
  Alcohol Nonflavanoids Proline
1     14.23        0.28    1065
2     13.20        0.26    1050
3     13.16        0.30    1185
4     14.37        0.24    1480
5     13.24        0.39     735
6     14.20        0.34    1450
7     14.39        0.30    1290
8     14.06        0.31    1295
9     14.83        0.29    1045
10    13.86        0.22    1045
11    14.10        0.22    1510
12    14.12        0.26    1280
13    13.75        0.29    1320
14    14.75        0.43    1150
15    14.38        0.29    1547
16    13.63        0.30    1310
17    14.30        0.33    1280
18    13.83        0.40    1130
19    14.19        0.32    1680
20    13.64        0.17    845
> scale(ds)
  Alcohol Nonflavanoids Proline
1     0.4630901 -0.27054355 -0.7184008
2    -1.7198976 -0.58883009 -0.7819386
3    -1.8046738  0.04774298 -0.2100983
4     0.7598069 -0.90711662  1.0394785
5    -1.6351214  1.48003239 -2.1162325
6     0.3995079  0.68431605  0.9124029
7     0.8021950  0.04774298  0.2346663
8     0.1027912  0.20688625  0.2558456
```

```

9   1.7347334  -0.11140029 -0.8031179
10  -0.3210899  -1.22540316 -0.8031179
11   0.1875674  -1.22540316  1.1665541
12   0.2299555  -0.58883009  0.1923078
13  -0.5542245  -0.11140029  0.3617419
14   1.5651810   2.11660546 -0.3583532
15   0.7810009  -0.11140029  1.3232807
16  -0.8085532   0.04774298  0.3193834
17   0.6114485   0.52517278  0.1923078
18  -0.3846721   1.63917565 -0.4430703
19   0.3783139   0.36602952  1.8866493
20  -0.7873591  -2.02111950 -1.6502886
attr(,"scaled:center")
  Alcohol Nonflavanoids      Proline
  14.0115          0.2970     1234.6000
attr(,"scaled:scale")
  Alcohol Nonflavanoids      Proline
  0.47183042      0.06283646  236.07991510
> ds
  Alcohol Nonflavanoids Proline
1   14.23          0.28      1065
2   13.20          0.26      1050
3   13.16          0.30      1185
4   14.37          0.24      1480
5   13.24          0.39      735
6   14.20          0.34      1450
7   14.39          0.30      1290
8   14.06          0.31      1295
9   14.83          0.29      1045
10  13.86          0.22      1045
11  14.10          0.22      1510
12  14.12          0.26      1280
13  13.75          0.29      1320
14  14.75          0.43      1150
15  14.38          0.29      1547
16  13.63          0.30      1310
17  14.30          0.33      1280
18  13.83          0.40      1130
19  14.19          0.32      1680
20  13.64          0.17      845
> summary(scale(ds))
    Alcohol           Nonflavanoids        Proline
Min.   :-1.805e+000  Min.   :-2.021e+000  Min.   :-2.116e+000
1st Qu.:-6.125e-01  1st Qu.:-5.888e-01  1st Qu.:-7.343e-01
Median : 2.088e-01  Median : -3.183e-02  Median : 1.923e-01
Mean   :-3.381e-15  Mean   :-6.217e-16  Mean   : 3.886e-16
3rd Qu.: 6.485e-01  3rd Qu.: 4.058e-01  3rd Qu.: 4.994e-01
Max.   : 1.735e+000 Max.   : 2.117e+000  Max.   : 1.887e+000

```

The function *rescaler* from Hadley Wickham's *reshape* package supports five methods for rescaling/standardising data: rescale to [0, 1]; subtract mean and divide by the standard deviation; subtract median and divide by median absolute deviation; convert values to a rank; and do nothing.

17.5.3 Binning

Many algorithms for data mining and before that for machine learning have been developed to deal only with categorial variables. Thus, the ability to turn numeric variables into categorical variables is important.

To divide the range of a numeric variable into intervals, and to then code the values of the variable according to which interval they fall into, thus transforming the variable into a categorical variable, we can use the *cut* function. In the example below the values are cut into three ranges, and given appropriate labels. As a bonus, the percentage distribution across the three ranges is also given!

```
> v <- c(1, 1.4, 3, 1.1, 0.3, 0.6, 4,5)
> v.cuts <- cut(v, breaks=c(-Inf, 1, 2, Inf), labels=c("Low", "Med", "High"))
> v.cuts

[1] Low  Med  High Med  Low  Low  High  High
Levels: Low Med High

> table(v.cuts)/length(v.cuts)*100

v.cuts
  Low   Med  High 
37.5  25.0 37.5
```

An example of this kind of transformation in practise is given in Chapter 20, page 397, where the *apriori* function requires categorical variables.

Binning is in fact a common concept and tools exist to automatically bin data using different strategies. The *binning* function of the *sm* package provides basic binning functionality.

```
> library(sm)
> x <- rnorm(100)
> y <- cut(x, breaks=binning(x, nbins=3)$breaks, labels=c("Lo", "Med", "Hi"))
> y
 [1] Lo  Lo  Med Med Med Lo  Med Med Med Med Lo  Lo  Med Hi  Hi
Med Lo
 [19] Lo  Med Lo  Hi  Lo  Med Hi  Lo  Med Lo  Med Lo  Med Med Lo  Med Med Med
 [37] Lo  Med Med Lo  Lo  Lo  Med Med Med Lo  Med Med Med Lo  Med Lo
Med
 [55] Med Lo  Med Med Med Med Lo  Med Med Med Lo  Med Lo  Med Med Med Med Lo
Med
 [73] Med Med Med Med Lo  Med Lo  Med Med Med Lo  Med Med Lo  Lo  Med Lo
Lo
 [91] Lo  Med Med Lo  Lo  Med Med Med Lo  Med
Levels: Lo Med Hi
```

17.5.4 Interpolation

17.6 Outlier Detection

17.7 Variable Selection

Variable selection (also known as **feature selection**) will identify a good subset of the data from which to perform modelling. In many cases, using a good subset of all available variables will lead to better models, expressed in the simplest of forms. This may include removing redundant input variables. Indeed, the principle of Occam's Razor indicates, and the need to communicate and understand models requires, that it is best to choose the simplest model from among the models that explain the data. This also avoids unnecessary variables confusing the modelling process with noise, and reduces the likelihood of having input variables that are dependent.

Variable selection is important in classification and is the process of selecting key features from the collection of variables (sometimes from thousands of variables) available. In such cases, most of the variables might be unlikely to be useful for classification purposes. Decision tree algorithms perform automatic feature selection and so they are relatively insensitive to variable selection. However, nearest neighbour classifiers do not perform feature selection and instead all variables, whether they are relevant or not, are used in building the classifier!

In decision tree algorithms, variables are selected at each step based on a selection criteria, and the number of variables that are used in the final model is determined by pruning the tree using cross-validation.

We present an example here of removing columns from data in R.

The *dprep* package in R provides support for variable selection, including *fincos* and *relief* selection methods.

Associated with variable selection is variable weighting. The aim here

is to essentially score variables according to their relevance or predictive power with respect to an output variable. Algorithms for variable weighting come in two flavours: those that use feedback from the modelling and those that don't. So called wrapper methods score variables by using subsets of variables to model and rating the variables according to how well the model performs. Filter algorithms, on the other hand, explore relationships within the data. The wrapper based approaches tend to produce better results but are computationally more expensive.

Togaware Watermark For Data Mining Survival

Togaware Watermark For Data Mining Survival

Chapter 18

Descriptive and Predictive Analytics

Togaware Watermark For Data Mining Survival

In this chapter we introduce the concepts of descriptive and predictive Analytics

The following chapters are then devoted to particular algorithms.

Modelling is what people often think of when they think of data mining. Modelling is the process of taking some data (usually) and building a model that reflects that data. Usually the aim is to address a specific problem through modelling the world in some way and from the model develop a better understanding of the world.

There is a bewildering array of tools and techniques at the disposal of the data miner. We can get a better understanding of what is available through categorising the algorithms according to the types of analysis performed. In this chapter we introduce and summarise the broader categories of data mining analysis. Part IV then presents, in a systematic manner, many algorithms that are used in data mining and available either freely or else implemented in commercial toolkits.

Much of the terminology used in data mining has grown out of that used in both machine learning and statistics. We identify, for example, two very broad categories of analysis as **unsupervised** and **supervised** (as in supervised and unsupervised learning).

We introduce such an ordering to the world of data mining techniques in this chapter. In summary:

18.1 Building a Model

Let's have a look at the simplest of problems. Suppose we want to model one variable (e.g., a person's height) in terms of another variable (e.g., a person's age).

We can create a collection of people's ages and heights, using some totally random data:

```
> set.seed(123)                      # To ensure repeatability.
> ages <- runif(10, 1, 20)          # Random ages between 1 and 20
> heights <- 30 + rnorm(10, 1, as.integer(ages)) + ages*5
> plot(ages, heights)
```

We can now build a model (in fact, a linear interpolation) that approximates this data using R's *approxfun*:

```
> my.model <- approxfun(ages, heights)
> my.model(15)
[1] 85.38172
> plot(my.model, add=TRUE, col=2, ylim=c(20,200), xlim=c(1,20))
```

The resulting plot is show in Figure 18.1. We can see it is only an approximate model and indeed, not a very good model. The data is pretty deficient, and we also know that generally height does not decrease for any age group in this range. It illustrates the modelling task though.

```
> my.spline <- splinefun(ages, heights)
```

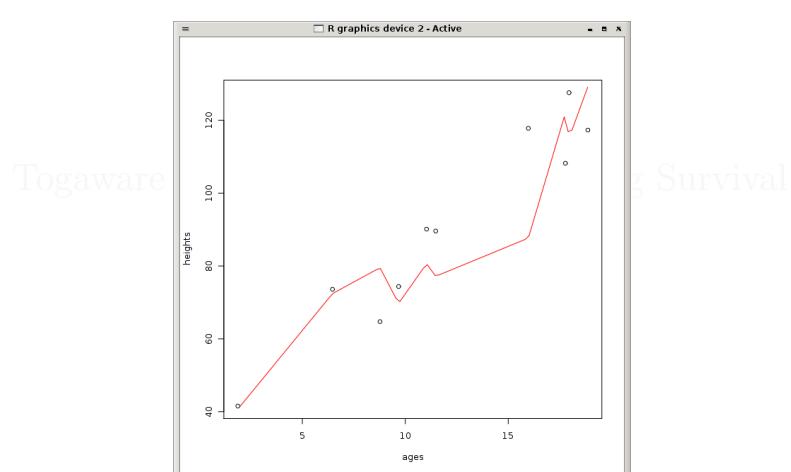


Figure 18.1: A approximate model of random data.

Togaware Watermark For Data Mining Survival

Chapter 19

Cluster Analysis: K-Means

Togaware Watermark For Data Mining Survival

The *amap* package includes k-means with a choice of distances like Euclidean and Spearman.

- . We optimize implementation (with a parallelized hierarchical clustering) and allow the possibility of using different distances like Euclidean or Spearman (rank-based metric).

19.1 Summary

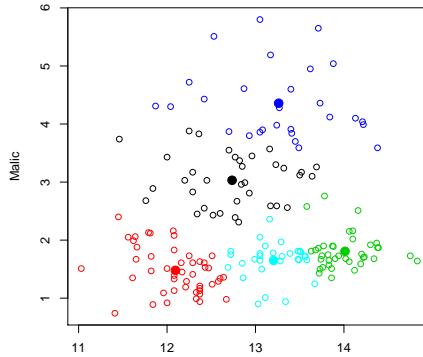
Complexity Clustering is usually expensive and K-Means is $O(n^2)$.

19.1.1 Clusters

Basic Clustering

We illustrate very simple clustering through a complete example where the task is to read data from a file (Section 14.3.4, page 219), extract the numeric fields, and then use k-means (Chapter 31.2, page 487) to cluster on just two columns. A plot of the clusters over the two columns

shows the points and the cluster centroids. Normally, the clusters would be built over more than just two columns. Also note that each time the code is run a different clustering is likely to be generated!



Togaware Watermark For Data Mining Survival

```
clusters <- 5
load("wine.Rdata")
pdf("graphics/rplot-cluster.pdf")
wine.cl = kmeans(wine[,2:3], clusters)
plot(wine[,2:3], col=wine.cl$cluster)
points(wine.cl$centers, pch=19, cex=1.5, col=1:clusters)
dev.off()
```

R code source: [rplot-cluster.R](#).

The resulting cluster entity has the following entries:

- cluster:** The cluster that each row belongs to.
- centers:** The medoid of each cluster.
- withinss:** The within cluster sum of squares.
- size:** The size of each cluster.

Hot Spots

Cluster analysis can be used to find clusters that are *most interesting* according to some criteria. For example, we might cluster the spam7 data of the DAAG package (without using yesno in the clustering) and then score the clusters depending on the proportion of yes cases within

the cluster. The following R code will build K clusters (user specified) and return a score for each cluster.

```
# Some ideas here from Felix Andrews
kmeans.scores <- function(x, centers, cases)
{
  clust <- kmeans(x, centers)
  # Iterate over each cluster to generate the scores
  scores <- c()
  for (i in 1:centers)
  {
    # Count number of TRUE cases in the cluster
    # as the proportion of the cluster size
    scores[i] <- sum( cases[clust$cluster == i] == TRUE ) / clust$size[i]
  }
  # Add the scores as another element to the kmeans list
  clust$scores <- scores
  return(clust)
}
```

We can now run this on our data with:

```
> require(DAAG)
> data(spam7)
> clust <- kmeans.scores(spam7[,1:6], centers=10, spam7["yesno"]=="y")
> clust[c("scores","size")]
$scores
[1] 0.7037037 0.1970109 0.5995763 0.7656250 0.8043478 1.0000000 0.4911628
[8] 0.7446809 0.6086957 0.6043956

$size
[1] 162 2208 472 128 46 5 1075 47 276 182
```

Thus, cluster 5 with 46 members has a high proportion of positive cases and may be a cluster we are interested in exploring further. Clusters 4, 8, and 1 are also probably worth exploring.

Now that we have built some clusters we can generate some rules that describe the clusters:

```
hotspots <- function(x, cluster, cases)
{
  require(rpart)
  overall = sum(cases) / nrow(cases)
  x.clusters <- cbind(x, cluster)
  tree = rpart(cluster ~ ., data = x.clusters, method = "class")
  # tree = prune(tree, cp = 0.06)
  nodes <- rownames(tree$frame)
  paths = path.rpart(tree, nodes = nodes)

  TO BE CONTINUED

  return(tree)
```

```
}
```

And to use it:

```
> h <- hotspots(spam7[,1:6], clust$cluster, spam7["yesno"]=="y")
```

Alternative Clustering

For model-based clustering see the BIC algorithm in the **mclust** package. This estimates density with a mixture of Gaussians.

For density-based clustering the following implementation of DBSCAN may be useful. It follows the notation of the original KDD-96 DBSCAN paper. For large datasets, it may be slow.

```
#Christian Hennig
distvector <- function(x,data)
{
  ddata <- t(data)-x
  dv <- apply(ddata^2,2,sum)
}
# data may be npx or distance matrix
# eps is the dbscan distance cutoff parameter
# MinPts is the minimum size of a cluster
# scale: Should the data be scaled?
# distances: has to be TRUE if data is a distance matrix
# showplot: Should the computation process be visualized?
# countmode: dbscan gives messages when processing point no. (countmode)
dbscan <- function(data,eps,MinPts=5, scale=FALSE, distances=FALSE,
                     showplot=FALSE,
                     countmode=c(1,2,3,5,10,100,1000,5000,10000,50000)){
  data <- as.matrix(data)
  n <- nrow(data)
  if (scale) data <- scale(data)
  unregpoints <- rep(0,n)
  e2 <- eps^2
  cv <- rep(0,n)
  cn <- 0
  i <- 1
  for (i in 1:n){
    if (i %in% countmode) cat("Processing point ", i, " of ", n, ".\n")
    unclass <- cv<1
    if (cv[i]==0){
      if (distances) seeds <- data[i,]<=eps
      else{
        seeds <- rep(FALSE,n)
        seeds[unclass] <- distvector(data[i,],data[unclass,])<=e2
      }
      if (sum(seeds)+unregpoints[i]<MinPts) cv[i] <- (-1)
      else{
        cn <- cn+1
        cv[i] <- cn
        for (j in 1:n)
          if (cv[j]==cv[i] & j!=i) seeds[j] <- TRUE
        if (sum(seeds)<MinPts) cv[i] <- (-1)
        else{
          cn <- cn+1
          cv[i] <- cn
          for (j in 1:n)
            if (cv[j]==cv[i] & j!=i) seeds[j] <- TRUE
        }
      }
    }
  }
}
```

```

cn <- cn+1
cv[i] <- cn
seeds[i] <- unclass[i] <- FALSE
unregpoints[seeds] <- unregpoints[seeds]+1
while (sum(seeds)>0){
  if (showplot) plot(data,col=1+cv)
  unclass[seeds] <- FALSE
  cv[seeds] <- cn
  ap <- (1:n)[seeds]
  print(ap)
  seeds <- rep(FALSE,n)
  for (j in ap){
    if (showplot) plot(data,col=1+cv)
    jseeds <- rep(FALSE,n)
    if (distances) jseeds[unclass] <- data[j,unclass]<=eps
    else{
      jseeds[unclass] <- distvector(data[j,],data[unclass,])<=e2
    }
    unregpoints[jseeds] <- unregpoints[jseeds]+1
    if (cn==1)
    #      cat(j," sum seeds=",sum(seeds)," unreg=",unregpoints[j],
    #           " newseeds=",sum(cv[jseeds]==0),"\n")
    if (sum(jseeds)+unregpoints[j]>=MinPts){
      seeds[jseeds] <- cv[jseeds]==0
      cv[jseeds & cv<0] <- cn
    }
    } # for j
  } # while sum seeds>0
} # else (sum seeds + ... >= MinPts)
} # if cv==0
} # for i
if (sum(cv==(-1))>0){
  noisenumber <- cn+1
  cv[cv==(-1)] <- noisenumber
}
else
  noisenumber <- FALSE
out <- list(classification=cv, noisenumber=noisenumber,
            eps=eps, MinPts=MinPts, unregpoints=unregpoints)
out
} # dbscan
# classification: classification vector
# noisenumber: number in the classification vector indicating noise points
# unregpoints: ignore...

```

19.2 Other Cluster Examples

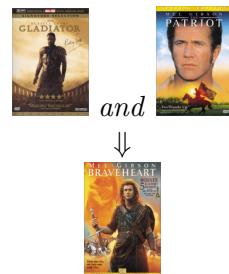
Togaware Watermark For Data Mining Survival

Chapter 20

Association Analysis: Apriori

Togaware Watermark For Data Mining Survival

Association analysis identifies relationships or affinities between entities and/or between variables. These relationships are then expressed as a collection of association rules. The approach has been particularly successful in mining very large transaction databases and is one of the core classes of techniques in data mining. A typical example is in the retail business where historic data might identify that customers who purchase the Gladiator DVD and the Patriot DVD also purchase the Braveheart DVD. The historic data might indicate that the first two DVDs are purchased by only 5% of all customers. But 70% of these then also purchase Braveheart. This is an *interesting* group of customers. As a business we may be able to take advantage of this observation by targetting advertising of the Braveheart DVD to those customers who have purchased both Gladiator and Patriot.



20.1 Summary

<i>Usage</i>	Market basket analysis, customer relationship management.
<i>Input</i>	Transactional data consisting of items whose co-occurrence within a transaction is of interest.
<i>Output</i>	Association rules identifying relationships between items within each transaction.
<i>Complexity</i>	Adversely affected as the support threshold decreases, particularly for large datasets. Exponential.
<i>Availability</i>	Freely available as part of the Borgelt Suite (Chapter 41, page 523) and integrated within R as part of the <i>arules</i> package (Chapter 43, page 527). Most commercial data mining toolkits include an implementation of apriori. Examples include SPSS (Chapter 48, page 547), SAS Enterprise Miner (Chapter 53, page 559), and Statistica (Chapter 54, page 563).

20.2 Overview

The Apriori algorithm is the original association rule algorithm. Each transaction is thought of as a basket of items (which we might represent as $\{A, B, C, D, E, F\}$). The algorithm searches for collections of items that appear together in multiple baskets (e.g., $\{A, C, F\}$). From these so called *itemsets* it identifies rules like $A, F \Rightarrow C$ which we read as indicating that A and F appearing in a transaction typically entails that C will also appear in the transaction.

The basis of an association analysis algorithm is the generation of frequent itemsets. However, naïve approaches will be quite expensive in computational time with even moderately sized databases. The Apriori algorithm takes advantage of the simple *apriori* observation that all subsets of a frequent itemset must also be frequent. That is, if $\{\text{milk}, \text{bread}, \text{cheese}\}$ is a frequent itemset then so must each of the smaller itemsets $\{\text{milk}, \text{bread}\}$, $\{\text{milk}, \text{cheese}\}$, $\{\text{bread}, \text{cheese}\}$, $\{\text{milk}\}$, $\{\text{bread}\}$, and $\{\text{cheese}\}$. This observation allows the algorithm to consider a significantly reduced search space by starting with frequent individual

items (eliminating rare items). We can then combine these into itemsets containing just two items and retain only those that are frequent enough. Similarly for itemsets containing three items, and so on.

Suppose we have a rule of the form $\mathcal{A} \Rightarrow \mathcal{C}$. We call \mathcal{A} the *antecedent* and \mathcal{C} the *consequent*, and both are non-empty sets of items.

The concept of *frequent enough* is a parameter of the algorithm, used to control the number of association rules discovered. This *support* specifies how frequently the items must appear in the whole dataset before the items can be considered as a candidate association rule. For example, the user may choose to consider only sets of items that occur in at least 5% of all transactions. Formally we define *support* for a collection of items \mathcal{I} as the proportion of all baskets in which all items in \mathcal{I} appear. Then we can define the support for an association rule as:

$$\text{support}(\mathcal{A} \Rightarrow \mathcal{C}) = \text{support}(\mathcal{A} \cup \mathcal{C})$$

A second parameter, the *confidence*, calculates the proportion of transactions containing \mathcal{A} that also contain \mathcal{C} . The confidence specifies a minimal probability for the association rule. For example, the user may choose to only generate rules which are true at least 90% of the time (that is, when \mathcal{A} appears in the basket, \mathcal{C} also appears in the same basket at least 90% of the time). Formally:

$$\text{confidence}(\mathcal{A} \Rightarrow \mathcal{C}) = \text{support}(\mathcal{A} \Rightarrow \mathcal{C}) / \text{support}(\mathcal{A})$$

20.3 Algorithm

The Apriori algorithm is a breadth-first or generate-and-test type of search algorithm. Only after exploring all possibilities of associations containing k items does it then consider those containing $k + 1$ items. For each k , all candidates are tested to determine whether they have enough support.

The algorithm uses a simple two step generate and merge process: generate frequent itemsets of size k then combine them to generate candidate frequent itemsets of size $k + 1$.

The algorithm is generally simple to implement and is reasonably efficient

even though the number of possible items is generally large and the baskets are generally small.

The input *data* to the algorithm consists of entities or transactions, each transaction representing a basket of items.

The two primary tuning parameters are *minsup* (**minimum support**) expressed as a percentage of the total number of transactions in *data*) and *mincon* (**minimum confidence**) also expressed as a percentage of the total number of transactions in *data*). Typically they have quite small values because of the size of the databases we are dealing with. Thus a support of 0.1% or smaller is not unusual.

Procedure APRIORI returns a set of association rules, each consisting of a left hand side, right hand side and a support and confidence tuple.

```

APRIORI(data, minsup, mincon):
1   tcount  $\leftarrow$  length(data)
2   items  $\leftarrow$  LISTUNIQUEITEMS(data)
3   icount  $\leftarrow$  length(items)
4   scount  $\leftarrow$  max(tcount * minsup / 100, 1)

5   f1  $\leftarrow$  FREQUENCIES(items, data)
6   REMOVEINFREQUENT(f1, scount)
7   if not f1: return NIL

8   for k  $\leftarrow$  2 to icount:
9       candidates  $\leftarrow$  GENERATECANDIDATES(fk-1)
10      if not candidates: break
11      fk  $\leftarrow$  FREQUENCIES(candidates, data)
12      REMOVEINFREQUENT(fk, scount)
13      if not fk: break
14   return BUILDASSOCIATIONS(f, mincon)

```

```

GENERATECANDIDATES( $f_k$ ):
1  $candidates \leftarrow NIL$ 
2 for  $u \in f_k, v \in f_k, u < v:$ 
3   if  $u_{1:-1} = v_{1:-1}:$ 
4      $c \leftarrow u \cup v_{-1}$ 
5      $s \leftarrow SUBSETS(c)$ 
6     if length(filter( $\lambda x : x \in f_k, s$ )) = length( $s$ ):
7        $candidates .append(c)$ 
8 return  $candidates$ 

```

```

BUILDASSOCIATIONS( $f, mincon$ ):
1  $rules \leftarrow NIL$ 
2 for  $k \in fis, itemset \in f_k, i \leftarrow 1$  to length( $itemset$ ):
3   for  $c \in COMBINATIONS(itemset, i):$ 
4      $lhs \leftarrow c$ 
5      $rhs \leftarrow NIL$ 
6     for  $k \in itemset:$  if  $k \notin c:$   $rhs .append(k)$ 
7      $confidence \leftarrow 100 \times \frac{SUPPORT(itemset, f)}{SUPPORT(lhs, f)}$ 
8     if  $confidence > mincon:$ 
9        $support \leftarrow SUPPORT(itemset, f)$ 
10       $rules .append(lhs \rightarrow rhs(support,$ 
11         $confidence))$ 
12 return  $rules$ 

```

20.4 Usage

The *arules* package in R provides the apriori functionality for R. As is the power of R, the packages is actually simply an interface to the widely used, and freely available, aprior software from Borgelt. This software was, for example, commercially licensed for use in the Clementine data mining package.

20.4.1 Read Transactions

file

format

sep

cols

rm.duplicates

20.4.2 Summary

20.4.3 Apriori

Togaware Watermark For Data Mining Survival

data

parameter

appearance

control

20.4.4 Inspect

20.5 Examples

The R function *apriori* from the *arules* package provides the apriori functionality using Borgelt's excellent implementation (Chapter 41, page 523). We use the *arules* package here to illustrate the discovery of apriori rules.

20.5.1 Video Marketing: Transactions From File

A simple example from e-commerce is that of an on-line retailer of DVDs, maintaining a database of all purchases made by each customer. (They will also, of course, have web log data about what the customers browsed.) The retailer might be interested to know what DVDs appear regularly together and to then use this information to make recommendations to other customers.

The input data consists of “transactions” like the following, which record on each line the purchase history of a customer, with each purchase separated by a comma (i.e., CSV format as discussed in Section 14.3.4, page 218):

```
Sixth Sense,LOTR1,Harry Potter1,Green Mile,LOTR2
Gladiator,Patriot,Braveheart
LOTR1,LOTR2
Gladiator,Patriot,Sixth Sense
Gladiator,Patriot,Sixth Sense
Gladiator,Patriot,Sixth Sense
Harry Potter1,Harry Potter2
Gladiator,Patriot
Gladiator,Patriot,Sixth Sense
Sixth Sense,LOTR,Gladiator,Green Mile
```

This data might be stored in the file *DVD.csv* which can be directly loaded into R using the *read.transactions* function of the *arules* package:

```
> library(arules)
> dvd.transactions <- read.transactions("DVD.csv", sep=",")
> dvd.transactions

transactions in sparse format with
10 transactions (rows) and
11 items (columns)
```

This tells us that there are, in total, 11 items that appear in the basket. The *read.transactions* function can also read data from a file with transaction ID and a single item per line (using the *format="single"* option).

For example, if the data consists of:

```
1,Sixth Sense
1,LOTR1
1,Harry Potter1
1,Green Mile
1,LOTR2
```

```

2,Gladiator
2,Patriot
2,Braveheart
3,LOTR1
3,LOTR2
4,Gladiator
4,Patriot
4,Sixth Sense
5,Gladiator
5,Patriot
5,Sixth Sense
6,Gladiator
6,Patriot
6,Sixth Sense
7,Harry Potter1
7,Harry Potter2
8,Gladiator
8,Patriot
9,Gladiator
9,Patriot
9,Sixth Sense
10,Sixth Sense
10,LOTR
10,Gladiator
10,Green Mile

```

we read the data with:

```

> dvd.transactions <- read.transactions("DVD.csv", format="single",
                                         sep=",", cols=c(1,2))
> dvd.transactions

transactions in sparse format with
 10 transactions (rows) and
 11 items (columns)

```

A *summary* of the dataset is obtained in the usual way:

```

> summary(dvd.transactions)

transactions as itemMatrix in sparse format with
 10 rows (elements/itemsets/transactions) and
 11 columns (items)

most frequent items:
  Gladiator      Patriot      Sixth Sense      Green Mile
       6             6              6                  2
Harry Potter1      (Other)
                   2                 8

element (itemset/transaction) length distribution:
2 3 4 5
3 5 1 1

```

```

Min. 1st Qu. Median   Mean 3rd Qu. Max.
2.00    2.25    3.00    3.00    3.00    5.00

includes extended transaction information - examples:
transactionIDs
1          1
2          2
3          3

```

The dataset is identified as a sparse matrix consisting of 10 rows (transactions in this case) and 11 columns or items. In fact, this corresponds to the total number of distinct items in the dataset, which internally are represented as a binary matrix, one column for each item. A distribution across the most frequent items (`Gladiator` appears in 6 “baskets”) is followed by a distribution over the length of each transaction (one transaction has 5 items in the “basket”). The final extended transaction information can be ignored in this simple example, but is explained for the more complex example that follows.

Association rules can now be built from the dataset:

```

> dvd.apriori <- apriori(dvd.transactions)

parameter specification:
  confidence minval smax arem aval originalSupport support minlen
    0.8      0.1     1 none FALSE           TRUE      0.1      1
maxlen target ext
  5   rules FALSE

algorithmic control:
  filter tree heap memopt load sort verbose
  0.1 TRUE TRUE FALSE TRUE    2    TRUE

apriori - find association rules with the apriori algorithm
version 4.21 (2004.05.09)      (c) 1996-2004 Christian Borgelt
set item appearances ...[0 item(s)] done [0.00s].
set transactions ...[11 item(s), 10 transaction(s)] done [0.00s].
sorting and recoding items ... [7 item(s)] done [0.00s].
creating transaction tree ... done [0.00s].
checking subsets of size 1 2 3 done [0.00s].
writing ... [7 rule(s)] done [0.00s].
creating S4 object ... done [0.01s].

```

The output here begins with a summary of the parameters chosen for the algorithm. The default values of confidence (0.8) and support (0.1) are noted, in addition to the minimum and maximum number of items in an itemset (`minlen=1` and `maxlen=5`). The default target is *rules*, but you could instead target *itemsets* or *hyperedges*. These can be set in the call

to *apriori* with the *parameter* argument which takes a list of keyword arguments.

We view the actual results of the modelling with the *inspect* function:

```
> inspect(dvd.apriori)

      lhs          rhs      support  confidence      lift
1 {LOTR1}    => {LOTR2}      0.2        1 5.000000
2 {LOTR2}    => {LOTR1}      0.2        1 5.000000
3 {Green Mile} => {Sixth Sense} 0.2        1 1.666667
4 {Gladiator}  => {Patriot}   0.6        1 1.666667
5 {Patriot}    => {Gladiator} 0.6        1 1.666667
6 {Sixth Sense,
   Gladiator} => {Patriot}   0.4        1 1.666667
7 {Sixth Sense,
   Patriot}    => {Gladiator} 0.4        1 1.666667
```

The rules are listed in order of decreasing lift.

We can change the parameters to get other association rules. For example we might reduce the support and deliver many more rules (81 rules):

```
> dvd.apriori <- apriori(dvd.transactions, par=list(supp=0.01))
```

Or else we might maintain support but reduce confidence (20 rules):

```
> dvd.apriori <- apriori(dvd.transactions, par=list(conf=0.1))
```

20.5.2 Survey Data: Data Preparation

For this example we will use the survey dataset (see Section 14.3.4, page 221). This dataset is a reasonable size and has some common real world issues. The *vignette* for *arules*, by the authors of the package (Hahsler et al., 2005), also use a similar dataset, available within the package through *data(Survey)*. We borrow some of their data transformations here.

We first review the dataset: there are 32,561 entities and 15 variables.

```
> load("survey.RData")
> dim(survey)
[1] 32561     15
> summary(survey)

      Age          Workclass      fnlwgt
Min. :17.00  Private       :22696  Min. : 12285

```

```

1st Qu.:28.00  Self-emp-not-inc: 2541   1st Qu.: 117827
Median :37.00  Local-gov       : 2093   Median : 178356
Mean   :38.58   State-gov       : 1298   Mean   : 189778
3rd Qu.:48.00  Self-emp-inc    : 1116   3rd Qu.: 237051
Max.   :90.00   (Other)        :  981   Max.   :1484705
NA's    :         NA's            : 1836

      Education      Education.Num          Marital.Status
HS-grad      :10501   Min.   : 1.00   Divorced      : 4443
Some-college: 7291   1st Qu.: 9.00   Married-AF-spouse : 23
Bachelors    : 5355   Median  :10.00   Married-civ-spouse :14976
Masters      : 1723   Mean    :10.08   Married-spouse-absent: 418
Assoc-voc    : 1382   3rd Qu.:12.00   Never-married   :10683
11th        : 1175   Max.    :16.00   Separated     : 1025
(Other)      : 5134

      Occupation           Relationship
Prof-specialty : 4140   Husband      :13193   Amer-Indian-Eskimo:
311
Craft-repair   : 4099   Not-in-family : 8305   Asian-Pac-Islander: 1039
Exec-managerial: 4066   Other-relative: 981   Black          : 3124
Adm-clerical   : 3770   Own-child    : 5068   Other          :
271
Sales          : 3650   Unmarried    : 3446   White          :27816
(Other)        :10993   Wife         : 1568
NA's           : 1843

      Sex      Capital.Gain   Capital.Loss   Hours.Per.Week
Female:10771  Min.   : 0       Min.   : 0.0   Min.   : 1.00
Male  :21790   1st Qu.: 0       1st Qu.: 0.0   1st Qu.:40.00
               Median : 0       Median : 0.0   Median :40.00
               Mean   : 1078   Mean   : 87.3   Mean   :40.44
               3rd Qu.: 0       3rd Qu.: 0.0   3rd Qu.:45.00
               Max.   : 99999  Max.   :4356.0  Max.   :99.00

      Native.Country  Salary.Group
United-States:29170  <=50K:24720
Mexico       : 643    >50K : 7841
Philippines  : 198
Germany     : 137
Canada      : 121
(Other)      : 1709
NA's        : 583

```

The first 5 rows of the dataset give some idea of the type of data:

```

> survey[1:5,]

      Age      Workclass fnlwgt Education Education.Num   Marital.Status
1  39      State-gov  77516  Bachelors      13  Never-married
2  50  Self-emp-not-inc  83311  Bachelors      13  Married-civ-spouse
3  38      Private   215646  HS-grad        9   Divorced
4  53      Private   234721    11th        7  Married-civ-spouse
5  28      Private   338409  Bachelors      13  Married-civ-spouse

```

	Occupation	Relationship	Race	Sex	Capital.Gain	Capital.Loss
1	Adm-clerical	Not-in-family	White	Male	2174	
0						
2	Exec-managerial	Husband	White	Male	0	
0						
3	Handlers-cleaners	Not-in-family	White	Male	0	
0						
4	Handlers-cleaners	Husband	Black	Male	0	
0						
5	Prof-specialty		Wife Black	Female	0	
0						
	Hours.Per.Week	Native.Country	Salary.Group			
1	40	United-States	<=50K			
2	13	United-States	<=50K			
3	40	United-States	<=50K			
4	40	United-States	<=50K			
5	40	Cuba	<=50K			

The dataset contains a mixture of categorical and numeric variables while the apriori algorithm works just with categorical variables (or factors). We note that the variable `fnlwgt` is a calculated value and not of interest to us so we can remove it from the dataset. The variable `Education.Num` is redundant since is it simply a numeric mapping of `Education`. We can remove these from the data frame simply by assigning NULL to them:

```
> survey$fnlwgt <- NULL
> survey$Education.Num <- NULL
```

This still leaves `Age`, `Capital.Gain`, `Capital.Loss`, and `Hours.Per.Week`. Following Hahsler et al. (2005), we will partition `Age` and `Hours.Per.Week` into fours segments each:

```
> survey$Age <- ordered(cut(survey$Age, c(15, 25, 45, 65, 100)),
   labels = c("Young", "Middle-aged", "Senior", "Old"))

> survey$Hours.Per.Week <- ordered(cut(survey$Hours.Per.Week,
   c(0, 25, 40, 60, 168)),
   labels = c("Part-time", "Full-time", "Over-time", "Workaholic"))
```

Again following Hahsler et al. (2005) we map `Capital.Gain` and `Capital.Loss` to `None`, and `Low` and `High` according to the *median*:

```
> survey$Capital.Gain <- ordered(cut(survey$Capital.Gain,
   c(-Inf, 0, median(survey$Capital.Gain[survey$Capital.Gain >0]), 1e+06)),
   labels = c("None", "Low", "High"))

> survey$Capital.Loss <- ordered(cut(survey$Capital.Loss,
```

```
c(-Inf, 0, median(survey$Capital.Loss[survey$Capital.Loss >0]), 1e+06),
labels = c("None", "Low", "High"))
```

That is pretty much it in terms of preparing the data for *apriori*:

```
> survey[1:5,]

      Age       Workclass Education Marital.Status Occupation
1 Middle-aged State-gov Bachelor Never-married Adm-clerical
2 Senior Self-emp-not-inc Bachelor Married-civ-spouse Exec-managerial
3 Middle-aged Private HS-grad Divorced Handlers-cleaners
4 Senior Private 11th Married-civ-spouse Handlers-cleaners
5 Middle-aged Private Bachelor Married-civ-spouse Prof-specialty

  Relationship Race Sex Capital.Gain Capital.Loss Hours.Per.Week
1 Not-in-family White Male Low None Full-time
2 Husband White Male None None Part-time
3 Not-in-family White Male None None Full-time
4 Husband Black Male None None Full-time
5 Wife Black Female None None Full-time

Native.Country Salary.Group
1 United-States <=50K
2 United-States <=50K
3 United-States <=50K
4 United-States <=50K
5 Cuba <=50K
```

The *apriori* function will coerce the data into the *transactions* data type, and this can also be done prior to calling *apriori* using the *as* function to view the data as a transaction dataset:

```
> library(arules)
> survey.transactions <- as(survey, "transactions")
> survey.transactions
transactions in sparse format with
 32561 transactions (rows) and
 115 items (columns)
```

This illustrates how the *transactions* data type represents variables in a binary form, one binary variable for each level of each categorical variable. There are 115 distinct levels (values for the categorical variables) across all 13 of the categorical variables.

The *summary* function provides more details:

```
> summary(survey.transactions)
transactions as itemMatrix in sparse format with
 32561 rows (elements/itemsets/transactions) and
```

```

115 columns (items)

most frequent items:
    Capital.Loss = None          Capital.Gain = None
                           31042           29849
Native.Country = United-States          Race = White
                           29170           27816
    Salary.Group = <=50K          (Other)
                           24720           276434

element (itemset/transaction) length distribution:
   10     11     12     13
  27   1809    563  30162

   Min. 1st Qu. Median   Mean 3rd Qu.   Max.
 10.00   13.00   13.00  12.87   13.00   13.00

includes extended item information - examples:
      labels variables      levels
1     Age = Young        Age       Young
2 Age = Middle-aged     Age Middle-aged

```

The summary begins with a description of the dataset sizes. This is followed by a list of the most frequent items occurring in the dataset. A `Capital.Loss` of `None` is the single most frequent item, occurring 31,042 times (i.e., pretty much no transaction has any capital loss recorded). The length distribution of the transactions is then given, indicating that some transactions have NA's for some of the variables. Looking at the summary of the original dataset you'll see that the variables `Workclass`, `Occupation`, and `Native.Country` have NA's, and so the distribution ranges from 10 to 13 items in a transaction.

The final piece of information in the `summary` output indicates the mapping that has been used to map the categorical variables to the binary variables, so that `Age = Young` is one binary variable, and `Age = Middle-aged` is another.

Now it is time to find all association rules using *apriori*. After a little experimenting we have chosen a support of 0.05 and a confidence of 0.95. This gives us 4,236 rules.

```

> survey.rules <- apriori(survey.transactions,
                           parameter = list(support=0.05, confidence=0.95))

parameter specification:
  confidence minval smax arem  aval originalSupport support minlen maxlen target
            0.95     0.1    1 none FALSE                  TRUE     0.05      1
5  rules
  ext

```

```

FALSE

algorithmic control:
  filter tree heap memopt load sort verbose
    0.1 TRUE TRUE FALSE TRUE 2 TRUE

apriori - find association rules with the apriori algorithm
version 4.21 (2004.05.09)          (c) 1996-2004 Christian Borgelt
set item appearances ...[0 item(s)] done [0.00s].
set transactions ...[115 item(s), 32561 transaction(s)] done [0.07s].
sorting and recoding items ... [36 item(s)] done [0.01s].
creating transaction tree ... done [0.08s].
checking subsets of size 1 2 3 4 5 done [0.23s].
writing ... [4236 rule(s)] done [0.00s].
creating S4 object ... done [0.04s].
```

```

> survey.rules
set of 4236 rules
```

```

> summary(survey.rules)
set of 4236 rules
```

```

rule length distribution (lhs + rhs):
  1   2   3   4   5
  1   34  328 1282 2591

  Min. 1st Qu. Median     Mean 3rd Qu. Max.
  1.000  4.000  5.000  4.517  5.000  5.000
```

```

summary of quality measures:
  support      confidence      lift
  Min. :0.05003  Min. :0.9500  Min. :0.9965
  1st Qu.:0.06469  1st Qu.:0.9617  1st Qu.:1.0186
  Median :0.08435  Median :0.9715  Median :1.0505
  Mean   :0.11418  Mean   :0.9745  Mean   :1.2701
  3rd Qu.:0.13267  3rd Qu.:0.9883  3rd Qu.:1.3098
  Max.   :0.95335  Max.   :1.0000  Max.   :2.9725
```

We can inspect the first 5 rules (slightly edited to suit publication):

```

> inspect(survey.rules[1:5])
      lhs                                rhs            support
conf lift
1 {}           => {Capital.Loss = None}  0.953
0.953 1.00
2 {Occupation = Machine-op-inspct} => {Workclass = Private}  0.058
0.955 1.37
3 {Occupation = Machine-op-inspct} => {Capital.Loss = None}  0.059
0.966 1.01
4 {Race = Black}           => {Capital.Loss = None}  0.093
0.967 1.01
5 {Occupation = Other-service} => {Salary.Group = <=50K}  0.097
0.958 1.26
```

Or we can list the first 5 rules which have a lift greater than 2.5

```
> subset(survey.rules, subset=lift>2.5)
set of 40 rules

> inspect(subset(survey.rules, subset=lift>2.5)[1:5])
      lhs                               rhs           support  conf  lift
1 {Age = Young,
  Hours.Per.Week = Part-time} => {Marital.Status = Never-married} 0.06 0.95 2.9
2 {Age = Young,
  Relationship = Own-child}    => {Marital.Status = Never-married} 0.10 0.97 2.9
3 {Age = Young,
  Hours.Per.Week = Part-time,
  Salary.Group = <=50K}          => {Marital.Status = Never-married} 0.06 0.96 2.9
4 {Age = Young,
  Hours.Per.Week = Part-time,
  Native.Country = United-States}=>{Marital.Status=Never-married} 0.05 0.95 2.9
5 {Age = Young,
  Capital.Gain = None,
  Hours.Per.Week = Part-time} => {Marital.Status = Never-married} 0.05 0.96 2.9
```

Here we build quite a few more rules and then view the rule with highest lift:

```
> survey.rules <- apriori(survey.transactions,
                           parameter = list(support = 0.05, confidence = 0.8))

parameter specification:
  confidence minval smax arem  aval originalSupport support minlen maxlen target
            0.8     0.1    1 none FALSE             TRUE     0.05      1

5  rules
  ext
  FALSE

algorithmic control:
  filter tree heap memopt load sort verbose
  0.1 TRUE TRUE FALSE TRUE    2     TRUE

apriori - find association rules with the apriori algorithm
version 4.21 (2004.05.09)      (c) 1996-2004 Christian Borgelt
set item appearances ...[0 item(s)] done [0.00s].
set transactions ...[115 item(s), 32561 transaction(s)] done [0.09s].
sorting and recoding items ... [36 item(s)] done [0.02s].
creating transaction tree ... done [0.10s].
checking subsets of size 1 2 3 4 5 done [0.35s].
writing ... [13344 rule(s)] done [0.00s].
creating S4 object ... done [0.08s].
```

```
> inspect(SORT(subset(survey.rules, subset=rhs %in% "Salary.Group"),
               by="lift")[1:3])

      lhs                               rhs           support  conf  lift
1 {Occupation = Exec-managerial,
  Relationship = Husband,
```

```

    Capital.Gain = High}          => {Salary.Group = >50K} 0.007
1 4.15
2 {Age = Middle-aged,
   Occupation = Exec-managerial,
   Capital.Gain = High}          => {Salary.Group = >50K} 0.005
1 4.15
3 {Age = Middle-aged,
   Education = Bachelors,
   Capital.Gain = High}          => {Salary.Group = >50K} 0.006
1 4.15

```

20.5.3 Other Examples

Health data is another example where association analysis can be effectively employed. Suppose a patient is obtaining a series of pathology and diagnostic imaging tests as part of an investigation to determine the cause of some symptoms. The “shopping basket” here is the collection of tests performed. Are there items in the basket that don’t belong together? Or are there some patients who don’t seem to be getting the appropriate selection of tests? The Australian Health Insurance Commission discovered an unexpected correlation between two pathology tests performed by pathology laboratories and paid for by insurance (Viveros et al., 1999). It turned out that only one of the tests was actually necessary, yet regularly both were being performed. The insurance organisation was able to reduce over-payment by disallowing payment for both tests, resulting in a saving of some half a million dollars per year.

In a very different application, IBM’s Advance Scout was developed to identify different strategies employed by basketball players in the US NBA. Discoveries include the observation that *Scottie Pippen’s favorite move on the left block is a right-handed hook to the middle*. And *when guard Ron Harper penetrates the lane, he shoots the ball 83% of the time*. Also it was noticed that *17% of Michael Jordan’s offence comes on isolation plays, during which he tends to take two or three dribbles before pulling up for a jumper* (Bhandari et al., 1997).

There are many more examples of unexpected associations having been discovered between items and, importantly, found to be particularly useful for improving business (and other) processes.

20.6 Resources and Further Reading

The original apriori algorithm is due to [Agrawal and Srikant \(1994\)](#). Borgelt (Chapter 41, page 523) provides a freely available, and very efficient, implementation of association analysis, with an extensive collection of measures of interestingness. This same library is directly accessible in R (Chapter 43, page 527) through the *arules* package described in this chapter.

Togaware Watermark For Data Mining Survival

Chapter 21

Classification: Decision Trees

Togaware Watermark For Data Mining Survival

21.1 Summary

<i>Usage</i>	
<i>Input</i>	
<i>Output</i>	
<i>Complexity</i>	
<i>Alternatives</i>	SAS provides

21.2 Overview

21.3 Algorithm

21.4 Usage

21.4.1 Rpart

Rpart generates decision trees by partitioning the data based on the largest amount of information to be gained.

There are a number of tuning variables for *rpart*.

minsplit

The *minsplit* specifies the minimum number of observations that must exist at a node in the tree before any further splitting will be attempted.

minbucket

The *minbucket* is the minimum number of observations in any terminal leaf node.

The two variables *minbucket* and *minsplit* are closely related. In *rpart* if either is not specified then by default the other is calculated as $\text{minsplit} = 3 * \text{minbucket}$.

Togaware Watermark For Data Mining Survival

cp

The variable *cp* governs the minimum complexity benefit that must be gained at each step in order to make a split worthwhile. The default is 0.01.

surrogatestyle

The variable *surrogatestyle* sets how the selection process should select the best variable. If set to 1 variables with a large number of missing values are essentially penalised. Defaults to 0.

maxdepth

The *maxdepth* variable specifies the maximum depth for the tree.

21.5 Examples

Simple Example

```
> sub <- c(sample(1:150, 75)) # Random sampling
> fit <- rpart(Species ~ ., data=iris, subset=sub)
> fit
n= 75

node), split, n, loss, yval, (yprob)
* denotes terminal node

1) root 75 47 virginica (0.2800000 0.3466667 0.3733333)
   2) Petal.Length< 2.5 21 0 setosa (1.0000000 0.0000000 0.0000000) *
   3) Petal.Length>=2.5 54 26 virginica (0.0000000 0.4814815 0.5185185)
      6) Petal.Length< 5.05 29 3 versicolor (0.0000000 0.8965517 0.1034483) *
      7) Petal.Length>=5.05 25 0 virginica (0.0000000 0.0000000 1.0000000) *

> table(predict(fit, iris[-sub,], type="class"), iris[-sub, "Species"])

            setosa versicolor virginica
setosa          29         0         0
versicolor       0        23         6
virginica        0         1        16
```

Convert Tree to Rules

```
list.rules.rpart <- function(model)
{
  if (!inherits(model, "rpart")) stop("Not a legitimate rpart tree")
  #
  # Get some information.
  #
  frm      <- model$frame
  names    <- row.names(frm)
  ylevels <- attr(model, "ylevels")
  ds.size <- model$frame[1,]$n
  #
  # Print each leaf node as a rule.
  #
  for (i in 1:nrow(frm))
  {
    if (frm[i,1] == "<leaf>")
    {
      # The following [,5] is hardwired - needs work!
      cat("\n")
      cat(sprintf(" Rule number: %s ", names[i]))
      cat(sprintf("[yval=%s cover=%d (%.0f%%) prob=%0.2f]\n",
                  ylevels[frm[i,]$yval], frm[i,]$n,
                  round(100*frm[i,]$n/ds.size), frm[i,]$yval2[,5]))
      pth <- path.rpart(model, nodes=as.numeric(names[i]), print.it=FALSE)
      cat(sprintf("    %s\n", unlist(pth)[-1]), sep="")
    }
  }
}
```

```

    }
}
}
```

Predicting Wine Type

A traditional decision tree can be built from the Wine data (Section 14.3.4, page 219) using the *rpart* (recursive partitioning) function. Also see *mvpart* in the *mvpart* package.

```

library("rpart")
load("wine.Rdata")
wine.rpart <- rpart(Type ~ ., data=wine)
par(xpd = TRUE)
par(mar = rep(1.1, 4))
plot(wine.rpart)
text(wine.rpart, use.n=TRUE)
```

R code source: [rplot-rpart.R](#).

```

> wine.rpart
n= 178

node), split, n, loss, yval, (yprob)
 * denotes terminal node

1) root 178 107 2 (0.33146067 0.39887640 0.26966292)
  2) Proline>=755 67 10 1 (0.85074627 0.05970149 0.08955224)
     4) Flavanoids>=2.165 59 2 1 (0.96610169 0.03389831 0.00000000) *
     5) Flavanoids< 2.165 8 2 3 (0.00000000 0.25000000 0.75000000) *
  3) Proline< 755 111 44 2 (0.01801802 0.60360360 0.37837838)
     6) Dilution>=2.115 65 4 2 (0.03076923 0.93846154 0.03076923) *
     7) Dilution< 2.115 46 6 3 (0.00000000 0.13043478 0.86956522)
        14) Hue>=0.9 7 2 2 (0.00000000 0.71428571 0.28571429) *
        15) Hue< 0.9 39 1 3 (0.00000000 0.02564103 0.97435897) *
```

The tree is displayed with the *plot* function.

You can even browse the plot with:

```
> path.rpart(fit)
```

Click on a node in the tree to display the path to that node. Exit with the right mouse button.

Use *printcp* to view the performance of the model.

```
> printcp(wine.rpart)

Classification tree:
rpart(formula = Type ~ ., data = wine)

Variables actually used in tree construction:
[1] Dilution   Flavanoids  Hue          Proline

Root node error: 107/178 = 0.60112

n= 178

      CP nsplit rel_error xerror     xstd
1 0.495327      0    1.00000 1.00000 0.061056
2 0.317757      1    0.50467 0.47664 0.056376
3 0.056075      2    0.18692 0.28037 0.046676
4 0.028037      3    0.13084 0.23364 0.043323
5 0.010000      4    0.10280 0.21495 0.041825

> formula(wine.rpart)
Type ~ Alcohol + Malic + Ash + Alcalinity + Magnesium + Phenols +
  Flavanoids + Nonflavanoids + Proanthocyanins + Color + Hue +
  Dilution + Proline
attr(,"variables")
list(Type, Alcohol, Malic, Ash, Alcalinity, Magnesium, Phenols,
  Flavanoids, Nonflavanoids, Proanthocyanins, Color, Hue, Dilution,
  Proline)
attr(,"factors")
      Alcohol  Malic  Ash  Alcalinity  Magnesium  Phenols  Flavanoids
Type           0       0     0           0           0       0       0
0             1       0     0           0           0       0       0
Alcohol        0       1     0           0           0       0       0
0             0       0     1           0           0       0       0
Malic          0       0     0           0           0       0       0
0             0       1     0           0           0       0       0
Ash            0       0     1           0           0       0       0
0             0       0     0           1           0       0       0
Alcalinity     0       0     0           0           1       0       0
0             0       0     0           0           0       1       0
Magnesium      0       0     0           0           0       1       0
0             0       0     0           0           0       0       1
Phenols         0       0     0           0           0       0       1
0             0       0     0           0           0       1       0
Flavanoids     0       0     0           0           0       0       0
1             0       0     0           0           0       0       0
Nonflavanoids  0       0     0           0           0       0       0
0             0       0     0           0           0       0       0
Proanthocyanins 0       0     0           0           0       0       0
0             0       0     0           0           0       0       0
Color          0       0     0           0           0       0       0
0             0       0     0           0           0       0       0
Hue            0       0     0           0           0       0       0
0             0       0     0           0           0       0       0
Dilution       0       0     0           0           0       0       0
0             0       0     0           0           0       0       0
```

```

Proline          0   0   0          0   0   0
0
Nonflavanoids  Proanthocyanins Color Hue Dilution Proline
Type           0          0   0   0   0
0
Alcohol         0          0   0   0   0
0
Malic           0          0   0   0   0
0
Ash             0          0   0   0   0
0
Alkalinity      0          0   0   0   0
0
Magnesium       0          0   0   0   0
0
Phenols          0          0   0   0   0
0
Flavanoids       0          0   0   0   0
0
Nonflavanoids    1          0   0   0   0
0
Proanthocyanins 0          1   0   0   0
0
Color            0          0   1   0   0
0
Hue              0          0   0   1   0
0
Dilution         0          0   0   0   1
0
Proline          0          0   0   0   0
1
attr(,"term.labels")
[1] "Alcohol"        "Malic"          "Ash"           "Alkalinity"
[5] "Magnesium"      "Phenols"        "Flavanoids"     "Nonflavanoids"
[9] "Proanthocyanins" "Color"          "Hue"           "Dilution"
[13] "Proline"
attr(,"order")
[1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1
attr(,"intercept")
[1] 1
attr(,"response")
[1] 1
attr(,"predvars")
list(Type, Alcohol, Malic, Ash, Alkalinity, Magnesium, Phenols,
     Flavanoids, Nonflavanoids, Proanthocyanins, Color, Hue, Dilution,
     Proline)
attr(,"dataClasses")
      Type      Alcohol      Malic      Ash
Alkalinity "factor" "numeric" "numeric" "numeric"
"numeric"
      Magnesium  Phenols  Flavanoids Nonflavanoids Proanthocyanins
"numeric" "numeric" "numeric" "numeric" "numeric"
"numeric"
      Color      Hue      Dilution      Proline

```

```
"numeric"      "numeric"      "numeric"      "numeric"
```

You can find which terminal branch each entity in the training dataset ends up in with the *where* component of the object.

```
> wine.rpart$where
   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16  17
18  19  20
3   3   3   3   6   3   3   3   3   3   3   3   3   3   3   3   3
3   3   3
21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37
38  39  40
3   3   3   3   3   3   3   3   3   3   3   3   3   3   3   3   3
3   3   3
[...]
161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178
9   8   9   9   9   9   9   9   9   9   9   9   9   9   9   4   4
9
```

Togaware Watermark For Data Mining Survival

The *predict* function will apply the model to data. The data must contain the same variable on which the model was built. If not an error is generated. This is a common problem when wanting to apply the model to a new dataset that does not contain all the same variables, but does contain the variables you are interested in.

```
> cols <- c("Type", "Dilution", "Flavanoids", "Hue", "Proline")
> predict(wine.rpart, wine[,cols])
Error in eval(expr, envir, enclos) : Object "Alcohol" not found
```

Fix this up with

```
> wine.rpart <- rpart(Type ~ Dilution + Flavanoids + Hue + Proline, data=wine)
> predict(wine.rpart, wine[,cols])
   1   2   3
1  0.96610169 0.03389831 0.00000000
2  0.96610169 0.03389831 0.00000000
[...]
70  0.03076923 0.93846154 0.03076923
71  0.00000000 0.25000000 0.75000000
[...]
177 0.00000000 0.25000000 0.75000000
178 0.00000000 0.02564103 0.97435897
```

Display a confusion matrix.

```
> table(predict(wine.rpart, wine, type="class"), wine$Type)
   1   2   3
```

```
1 57 2 0
2 2 66 4
3 0 3 44
```

Predicting Salary Group

A little more complex is the survey data.

```
> survey.rp <- rpart(Salary.Group ~ ., data=survey)
> survey.rp
n= 32561

node), split, n, loss, yval, (yprob)
* denotes terminal node

1) root 32561 7841 <=50K (0.75919044 0.24080956)
   2) Relationship=Not-in-family,Other-relative,Own-child,Unmarried
      17800 1178 <=50K (0.93382022 0.06617978)
      4) Capital.Gain< 7073.5 17482 872 <=50K (0.95012012 0.04987988) *
      5) Capital.Gain>=7073.5 318 12 >50K (0.03773585 0.96226415) *
   3) Relationship=Husband,Wife 14761 6663 <=50K (0.54860782 0.45139218)
      6) Education=10th,11th,12th,1st-4th,5th-6th,7th-8th,9th,Assoc-acdm,
         Assoc-voc,HS-grad,Preschool,Some-college
         10329 3456 <=50K (0.66540807 0.33459193)
      12) Capital.Gain< 5095.5 9807 2944 <=50K (0.69980626 0.30019374) *
      13) Capital.Gain>=5095.5 522 10 >50K (0.01915709 0.98084291) *
   7) Education=Bachelors,Doctorate,Masters,Prof-school 4432 1225 >50K
      (0.27639892 0.72360108) *

> table(survey$Salary.Group)

<=50K  >50K
24720  7841
```

Predicting Fraud: Underrepresented Classes

Consider the problem of fraud investigation, perhaps in insurance claims. Suppose some 10,000 cases have been investigated and of those just 5% (or 500) were found to be fraudulent. This is a typical scenario for numerous organisations. With modelling we wish to improve the deployment of our resources so that the 95% of the cases that were not fraudulent need not all be investigated, yet the 5% still need to be identified. Each case

of actual fraud also has a dollar value associated with it, representing the risk (actually, the magnitude of the risk) associated with the case.

The advantage of a decision tree approach is that the resulting tree (and particularly if we traverse each path through the tree to obtain a set of rules) can be easily understood and explained, allowing decision makers the opportunity to understand the changes being suggested.

An aim of modelling here is to present a model which will allow us to identify a caseload tradeoff with coverage whilst maximising the recovery of dollars represented as the risk.

The first step is to build a decision tree. Because of the skewness of the outcome we might “trick” rpart into working harder to identify the frauds. As Breiman et al. 1984 indicate, different costs for misclassification can be modelled either by modifying the loss matrix or by using different prior probabilities for the classes, or by using different weights for the response classes. These can be achieved using *rpart* with the *parms* option which will record the options we want for the tree building. The variables *loss* and *prior* can be set within the *parms* list of variables. Another approach is to use the *weights* (to weight each case) and *cost* (the relative cost of obtaining the variable value, thus can tune the choice of variables in the model) options of *rpart*.

In using *prior* the relative prior probability assigned to each class can be used to adjust the importance of misclassifications for each class. Thus, priors may be interpreted as case weights, although case weights are treated as case multipliers.

In fraud it is desirable not to misclassify cases of fraud, thus a more accurate classification is desired for some classes over others. This will not be exhibited through the relative class sizes. However, if the criterion for predictive accuracy is misclassification costs, as it often is, then minimising costs amounts to minimising the proportion of misclassified cases when priors are considered proportional to the class sizes and misclassification costs are taken to be equal for every class. A *loss* matrix elaborates the loss incurred if an entity of one decision class (say 1) is erroneously classified by our model as another decision class (say 0).

For the following examples we use the *audit* dataset from the *rattle* package. This dataset consists of a bunch of input variables with the

target (Adjusted) being in the last column. The outcome is binary (0/1) with the positive case (1) being under-represented. The measure of the risk (Adjustment) is assumed to be in the second last column. The risk is the dollar amount recovered from the review of the fraud and should not be used as an input variable in the modelling (thus we use `audit[,-2]` to remove this column from the data). The examples here show the R code behind Rattle as presented in Chapter 2.

Using *prior* to over-emphasize the under-represented outcome:

```
library(rpart)
library(rattle)
data(audit)
audit.rpart <- rpart(Adjusted ~ ., data=audit[,-12], parms=list(prior=c(.5,.5)))
```

Using *loss*:

```
library(rpart)
library(rattle)
data(audit)
loss <- matrix(c(0, 2, 1, 0), byrow=TRUE, ncol=2)
audit.rpart <- rpart(Adjusted ~ ., data=audit[,-12], parms=list(loss=loss))
```

Using *weights* based on the value of the risk:

```
library(rpart)
library(rattle)
data(audit)
weight <- abs(audit$Adjustment)/max(audit$Adjustment)*10+1
audit.rpart <- rpart(Adjusted ~ ., data=audit[,-12], weights=weight)
```

Now we apply the model to the data to obtain probabilistic predictions (note that we are applying the model to the same training set and this is will give us an optimistic estimate - Rattle uses training/test sets to give a better estimate). The result is the probability, for each case, of it being a fraud:

```
audit.predict <- predict(audit.rpart, audit)
```

Now, using Rattle (see Chapter 2) we can produce a Risk Chart that presents the cases ordered by the probability of being a fraud, and plotting the coverage and risk (percentage of dollars) recovered for each choice of caseload.

```
library(rattle)
eval <- evaluateRisk(audit.predict, audit$Adjusted, audit$Adjustment)
plotRisk(eval$Caseload, eval$Precision, eval$Recall, eval$Risk)
title(main="Risk Chart using rpart on the audit dataset",
      sub=paste("Rattle", Sys.time(), Sys.info()["user"]))
```

To produce:

The plot can be easily used to tell a story about the tradeoff between recovering all risk cases and the amount of effort expended. The solid black diagonal line can be thought of as the baseline. The so called optimal line (the caseload where the sum of the distances of the Revenue and Adjustments from the baseline is maximal) might be an interesting point to consider. The story says that if our investigators actually only investigated 25% of the cases that they are currently investigating, then they would recover 64% of the cases that were found to be fraudulent, and 72% of the dollars that were recovered. The other 75% of the investigative resources could be better deployed, perhaps to target higher risk populations where the returns are greater. Note that the Strike Rate has increased from 26% in the original dataset to 67% at this optimal point.

Perhaps an even better story is that with half of the resources currently deployed on investigations (i.e., a caseload of 50%), with our model we could recover almost 90% of the frauds and marginally more than 90% of the dollars known to be recoverable.

We do note that here we are assuming the caseload directly reflects the actual workload (i.e., every case takes the same amount of effort).

Such Risk Charts are used to compare the performance of alternative models, where the aim is often to extend the red (Revenue) and green (Recall) lines toward the top left corner of the plot, or to maximise the area under these curves.

Alternatives and Enhancements

An alternative is provided by the *tree* package, although *rpart* is the generally preferred function.

For multivariate use *mpart*.

Visualise trees with *maptree* and *pinktoe*.

21.6 Resources and Further Reading

The traditional decision tree algorithms suffer from overfitting and a bias toward selecting variables with many possible splits. The algorithms do not use any statistical significance concepts and thus, as noted by [Mingers \(1989\)](#), cannot distinguish between significant and insignificant improvements in the information measure.

Conditional Trees (Chapter [32](#), page [495](#)) take into account distributional properties.

Can library(rgl) be used to visualise a decision tree?

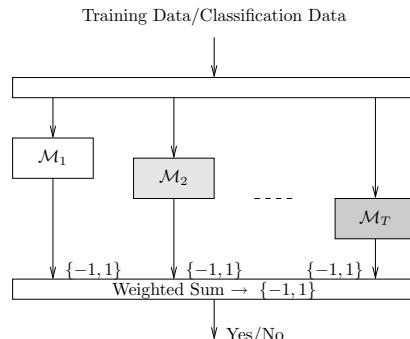
[Ye \(1998\)](#) discussed generalised degrees of freedom and shows that to get an unbiased estimate of R^2 from recursive partitioning (decision tree building) you have to use the formula for adjusted R^2 with the number of parameters far exceeding the number of final splits. He showed how to estimate the degrees of freedom. Decision tree building can result in simple predictive models but this can be an illusion.

Chapter 22

Classification: Boosting

Togaware Watermark For Data Mining Survival

The **Boosting** meta-algorithm is an efficient, simple, and easy to program learning strategy. The popular variant called **AdaBoost** (an abbreviation for Adaptive Boosting) has been described as the “best off-the-shelf classifier in the world” (attributed to Leo Breiman by [Hastie et al. \(2001, p. 302\)](#)). **Boosting** algorithms build multiple models from a dataset, using some other learning algorithm that need not be a particularly good learner. Boosting associates weights with entities in the dataset, and increases (boosts) the weights for those entities that are hard to accurately model. A sequence of models is constructed and after each model is constructed the weights are modified to give more weight to those entities that are harder to classify. In fact, the weights of such entities generally oscillate up and down from one model to the next. The final model is then an additive model constructed from the sequence of models, each model’s output



weighted by some score. There is little tuning required and little is assumed about the learner used, except that it should be a weak learner! We note that boosting can fail to perform if there is insufficient data or if the weak models are overly complex. Boosting is also susceptible to noise.

22.1 Summary

<i>Usage</i>	Classification tasks, regression and other modelling.
<i>Input</i>	Training data consisting of entities expressed as attribute-value pairs, with a class associated with each entity.
<i>Output</i>	An ensemble of models which are to be deployed together with their decisions being combined to give a joint decision.
<i>Complexity</i>	Depends on complexity of the weak learner employed, but generally the weak learner is quite simple (e.g., OneR or Decision Stumps) hence scalability is generally good.
<i>Availability</i>	Freely available in Weka (Chapter 46, page 535) and in R (Chapter 43, page 527). Commercial data mining toolkits implementing AdaBoost include TreeNet (Chapter 55, page 569), Statistica (Chapter 54, page 563), and Virtual Predict (Chapter 56, page 571).

22.2 Overview

Boosting builds a collection of models using a “weak learner” and thereby reduces misclassification error, bias, and variance (Bauer and Kohavi, 1999; Schapire et al., 1997). Boosting has been implemented in, for example, C5.0. The term originates with Freund and Schapire (1995).

The algorithm is quite simple, beginning by building an initial model from the training dataset. Those entities in the training data which the

model was unable to capture (i.e., the model mis-classifies those entities) have their weights boosted. A new model is then built with these boosted entities, which we might think of as the problematic entities in the training dataset. This model building followed by boosting is repeated until the specific generated model performs no better than random. The result is then a panel of models used to make a decision on new data by combining the “expertise” of each model in such a way that the more accurate experts carry more weight.

As a meta learner Boosting employs some other simple learning algorithm to build the models. The key is the use of a weak learning algorithm—essentially any weak learner can be used. A weak learning algorithm is one that is only somewhat better than random guessing in terms of error rates (i.e., the error rate is just below 50%). An example might be decision trees of depth 1 (i.e., decision stumps).

22.3 AdaBoost Algorithm

Boosting employs a weak learning algorithm (which we identify as the *learner*). Suppose the dataset (*data*) consists of N entities described using M variables (lines 1 and 2 of the meta-code below). The M th variable (i.e., the last variable of each entity) is assumed to be the classification of the entity. In the algorithm presented here we denote the training data (an N by $M - 1$ matrix) as x (line 3) and the class associated with each entity in the training data (a vector of length M) as y (line 4). Without loss of generality we can restrict the class to be either 1 (perhaps representing *yes*) or -1 (representing *no*). This will simplify the mathematics. Each entity in the training data is initially assigned the same weight: $w_i = \frac{1}{N}$ (line 5).

The weak learner will need to use the weights associated with each entity. This may be handled directly by the learner (e.g., *rpart* takes an option to specify the *weights*) or else by generating a modified dataset by sampling the original dataset based on the weights.

The first model, \mathcal{M}_1 , is built by applying the weak *learner* to the *data* with weights w (line 7). \mathcal{M}_1 , predicting either 1 or -1 , is then used to identify the set of indicies of misclassified entities (i.e., where $\mathcal{M}_1(x_p) \neq$

y_p), denoted as ms (line 8). For a completely accurate model we would have $\mathcal{M}_1(x_i) = y_i$. Of course the model is expected to be only slightly better than random so ms is unlikely to be empty.

A relative error ϵ_1 for \mathcal{M}_1 is calculated as the relative sum of the weights of the misclassified entities (line 9). This is used to calculate α_1 (line 10), used, in turn, to adjust the weights (line 11). All weights could be either decreased or increased depending on whether the model correctly classifies the corresponding entity, as proposed by [Freund and Schapire \(1995\)](#). However, this can be simplified to only increasing the weights of the misclassified entities, as proposed by [Hastie et al. \(2001\)](#). These entities thus become more important.

The learning algorithm is then applied to the new weighted *data* with the *learner* expected to give more focus on the difficult entities whilst building this next model, \mathcal{M}_2 . The weights are then modified again using the errors from \mathcal{M}_2 . The model building and weight modification is then repeated until the new model performs no better than random (i.e., the error is 50% or more: $\epsilon_i \geq 0.5$), or is perfect (i.e., the error rate is 0% and ms is empty), or perhaps after a fixed number of iterations.

```
ADABoost(data, learner):
```

```

1  N  $\leftarrow$  nrow(data)
2  M  $\leftarrow$  ncol(data)
3  x  $\leftarrow$  data[, 1 : M - 1]
4  y  $\leftarrow$  data[, M]
5  for i  $\leftarrow$  1 to N: wi =  $\frac{1}{N}$ 

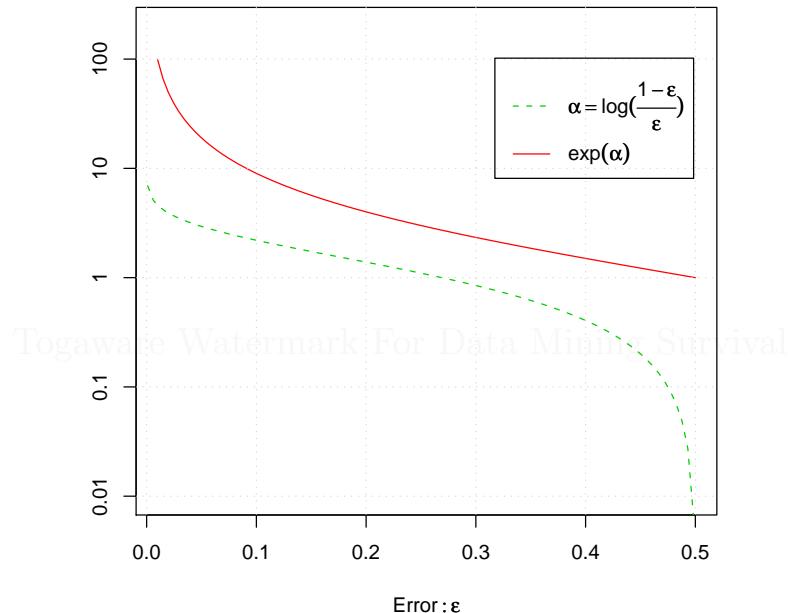
6  repeat i  $\leftarrow$  1, i  $\leftarrow$  i + 1:
7       $\mathcal{M}_i \leftarrow \text{learner}(\text{data}, w)$ 
8      ms = {p |  $\mathcal{M}_i(x_p) \neq y_p$ }
9       $\epsilon_i = \frac{\sum_{j \in ms} w_j}{\sum_{j=1}^n w_j}$ 
10      $\alpha_i = \log((1 - \epsilon_i)/\epsilon_i)$ 
11     for j  $\in$  ms: wj = wj  $\times e^{\alpha_i}$ 
12     for i  $\leftarrow$  1 to N: wi =  $\frac{w_i}{\sum_{j=1}^n w_j}$ 
13  until  $\epsilon_i \geq 0.5$  or ms =  $\emptyset$ 

14 return [ $\mathcal{M}(x) = \text{sign}(\sum_{j=1}^T \alpha_j \mathcal{M}_j(x))$ ]
```

The final model \mathcal{M} (line 14) combines the other models using a weighted sum of the outputs of these other models. The weights, α_j , reflect the accuracy of each of the constituent models.

A simple example can illustrate the process. Suppose the number of training entities, N , is 10. Each weight, w_j , is thus initially 0.1 (line 5). Imagine the first model, \mathcal{M}_1 , correctly classifies the first 6 of the 10 entities (e.g., $ms = \{7, 8, 9, 10\}$), so that $\epsilon_1 = 0.1 + 0.1 + 0.1 + 0.1 / 4 = 0.4$. Then $\alpha_1 = \log(0.6 / 0.4) = 0.405$, and is the weight that will be used to multiply the results from this model to be added into the overall model score. The weights w_7, \dots, w_{10} then become $0.1 \times e^{0.405} = 0.1 \times 1.5 = 0.15$. That is, they now have more importance for the next model build. Suppose now that \mathcal{M}_2 correctly classifies 8 of the entities (with $ms = \{1, 8\}$), so that $\epsilon_2 = (0.1 + 0.15) / 1.2 = 0.208$ and $\alpha_2 = \log(0.792 / 0.208) = 1.337$. Thus $w_1 = 0.1 \times e^{1.337} \approx 0.381$ and $w_8 = 0.15 \times e^{1.337} \approx 0.571$. Note how record 8 is proving particularly troublesome and so its weight is now the highest.

We can understand the behaviour of the function used to weight the models ($\log((1-\epsilon_i)/\epsilon_i)$) and to adjust the entity weights (e^{α_i}) by plotting both of these for the range of errors we expect (from 0 to 0.5 or 50%).



```
plot(function(x) exp(log((1-x)/x)), xlim=c(0.01, 0.5), ylim=c(0.01, 200),
      xlab=expression(Error: epsilon), ylab="", log="y", lty=1, col=2, yaxt="n")
plot(function(x) log((1-x)/x), add=TRUE, lty=2, col=3)
axis(side=2, at=c(0.01, 0.1, 1, 10, 100), c(0.01, 0.1, 1, 10, 100))
grid()
exp.leg <- expression(alpha == log(over(1-epsilon, epsilon)), exp(alpha))
legend("topright", exp.leg, lty=2:1, col=3:2, inset=c(0.04, 0.1))
```

R code source: [rplot-adaboost.R](#).

First, looking at the value of the α 's ($\log((1 - \epsilon_i)/\epsilon_i)$), we see that for errors close to zero, that is, for very accurate models, the α_i (i.e., the weight that is used to multiply the results from the model) is very high. For an error of about 5% the multiplier is almost 3, and for a 1% error the multiplier is about 4.6. With an error of approximately 27% (i.e., $\epsilon = 0.26894$) the multiplier is close to 1. For errors greater than this the model gets weights less than 1, heading down to a weight of 0 at $\epsilon = 0.5$.

In terms of building the models, the entity weights are multiplied by e^{α_i} . If the model we have just built (\mathcal{M}_i) is quite accurate (ϵ close to 0) then fewer entities are being misclassified, and their weights are increased significantly (e.g., for a 5% error the weight is multiplied by 19). For inaccurate models (as ϵ approaches 0.5) the multiplier for the weights approaches 1. Note that if $\epsilon_i = 0.5$ the multiplier is 1, and thus no change is made to the weights of the entities. Of course, building a model on the same dataset with the same weights will build the same model, thus the criteria for continuing to build a model tests that $\epsilon < 0.5$.

22.4 Examples

A number of R packages implement boosting. The *caTools* package provides the *LogitBoost* function which is perhaps the simplest to use, and is an efficient implementation for large datasets. The *boost* package provides the *adaboost* function as well as *logitboost*, and relies on *rpart* for building the models, and is less efficient. The *gbm* package is the more sophisticated of the packages and implements the more general Generalise Boosted Regression Models. We will illustrate boosting with the *gbm* package.

We start our examples though with a step through of the process using just *rpart*.

22.4.1 Step by Step

The *learner* deployed in the AdaBoost algorithm is typically a decision tree learner that builds no more than a single split decision tree (also called a decision stump). Such a decision tree can be built in R using *rpart* and we illustrate this in the following code segments.

First we load the *wine* dataset and extract the input variables (x) and the output variable (y). For a simple application of the algorithm, we'll have only a binary output (predicting $Type == 1$), and again for mathematical convenience we'll predict 1 or -1:

```
> library(rpart)
> load("wine.RData")
> N <- nrow(wine) # 178
```

```
> M <- ncol(wine) # 14
> x <- as.matrix(wine[,2:M])
> y <- as.integer(wine[,1])
> y[y>1] <- -1
> y
[1]  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
1  1  1  1
[...]
[176] -1 -1 -1
```

Now we'll initialise the entity weights and build the first model. We first set the weights to all be the same, and then set up an *rpart.control* structure for building a decision tree stump. The control simply includes the *maxdepth* option set to 1 so that a single level tree is built:

```
> w <- rep(1/N, N)
> w
[1] 0.005617978 0.005617978 0.005617978 0.005617978 0.005617978 0.005617978
[...]
[175] 0.005617978 0.005617978 0.005617978 0.005617978
> control <- rpart.control(maxdepth=1)
> M1 <- rpart(y ~ x, weights=w/mean(w), control=control, method="class")
> M1
n= 178

node), split, n, loss, yval, (yprob)
 * denotes terminal node

1) root 178 59 -1 (0.66853933 0.33146067)
  2) x.Proline< 755 111  2 -1 (0.98198198 0.01801802) *
  3) x.Proline>=755 67 10  1 (0.14925373 0.85074627) *
```

We see that the decision tree algorithm has chosen Proline to split the data on, at a split point of 755. For Proline less than 755 the decision is -1 with probability 0.98, and for Proline greater than or equal to 755 the decision is 1 with probability 0.85.

We now need to find those entities which are incorrectly classified by the model. The R code here calls *predict* to apply the model M1 to the dataset it was built from. From this result we get the second column which is the list of probabilities for each entity being in class 1. If this probability is above 0.5 then the result is 1, otherwise it is -1 (multiplying the logical value by 2 and then subtracting 1 achieves this since TRUE is regarded as 1 and FALSE as 0). The resulting class is then compared to the y's and *which* returns the index of those entities for which the prediction differs from the actual class.

```
> ms <- which(((predict(M1)[,2]>0.5)*2)-1 != y)
```

22.4 Examples

435

```
> names(ms) <- NULL
> ms
[1] 5 44 71 74 75 96 142 145 146 158 176 177
```

We can now calculate the model weight and update the entity weights, dividing by the resulting sum of weights to get a normalised value (so that $\text{sum}(w)$ is 1):

```
> e1 <- sum(w[ms])/sum(w) # 0.06741573
> a1 <- log((1-e1)/e1) # 2.627081
> w[ms] <- w[ms]*exp(a1)
> w[ms]
[1] 0.07771536 0.07771536 0.07771536 0.07771536 0.07771536 0.07771536
[7] 0.07771536 0.07771536 0.07771536 0.07771536 0.07771536 0.07771536
```

We build our second model:

```
> M2 <- rpart(y ~ x, weights=w/mean(w), control=control, method="class")
> M2

n= 178

node), split, n, loss, yval, (yprob)
  * denotes terminal node

1) root 178 45.3935700 -1 (0.744979920 0.255020080)
  2) x.Flavanoids< 2.31 101 0.5361446 -1 (0.995381062 0.004618938) *
  3) x.Flavanoids>=2.31 77 17.0672700 1 (0.275613276 0.724386724) *
> ms <- which(((predict(M2)[,2]>0.5)*2)-1 != y)
> names(ms) <- NULL
> ms
[1] 28 64 66 67 72 74 80 82 98 99 100 110 111 121 122 124 125 126 127
[20] 129
> e2 <- sum(w[ms])/sum(w) # 0.09889558
> a2 <- log((1-e2)/e2) # 2.209557
> w[ms] <- w[ms]*exp(a2)
> w[ms]
[1] 0.05118919 0.05118919 0.05118919 0.05118919 0.05118919 0.70811707
[7] 0.05118919 0.05118919 0.05118919 0.05118919 0.05118919 0.05118919
[13] 0.05118919 0.05118919 0.05118919 0.05118919 0.05118919 0.05118919
[19] 0.05118919 0.05118919
```

And then our third model:

```
> M3 <- rpart(y ~ x, weights=w/mean(w), control=control, method="class")
> M3

n= 178

node), split, n, loss, yval, (yprob)
  * denotes terminal node

1) root 178 27.60091 -1 (0.84493870 0.15506130)
  2) x.Proline< 987.5 134 12.09805 -1 (0.92554915 0.07445085) *
```

```

3) x.Proline >= 987.5 44 0.00000 1 (0.000000000 1.000000000) *
> ms <- which(((predict(M3)[,2]>0.5)*2)-1 != y)
> names(ms) <- NULL
> ms
[1] 5 20 21 22 25 26 29 36 37 40 41 44 45 48 57
> e3 <- sum(w[ms])/sum(w)
> e3
[1] 0.06796657
> a3 <- log((1-e3)/e3)
> a3
[1] 2.618353

```

The final model, if we chose to stop here, is then:

$$\mathcal{M}(x) = 2.627081 * \mathcal{M}_1(x) + 2.209557 * \mathcal{M}_2(x) + 2.618353 * \mathcal{M}_3(x)$$

22.4.2 Using gbm

Generalised boosted models, as proposed by Friedman (2001) and extended by Friedman (2002), has been implemented for R as the *gbm* package by Greg Ridgeway. This is a much more extensive package for boosting than the *boost* package.

We illustrate AdaBoost using the *distribution* option of the *gbm* function.

```

      10      0.9349      nan  0.0010  0.0004
      100     0.8750     nan  0.0010  0.0007

> summary(ds.gbm)
      var   rel.inf
1    Proline  91.82978
2    Flavanoids  8.17022
3    Alcohol  0.00000
4    Malic  0.00000
5    Ash  0.00000
6    Alkalinity  0.00000
7    Magnesium  0.00000
8    Phenols  0.00000
9    Nonflavanoids  0.00000
10 Proanthocyanins  0.00000
11   Color  0.00000
12   Hue  0.00000
13   Dilution  0.00000
> pretty.gbm.tree(ds.gbm)
      SplitVar SplitCodePred LeftNode RightNode MissingNode ErrorReduction Weight
0        12  8.675000e+02       1         2          3      65.36408
89
1       -1 -8.139656e-04      -1        -1        -1      0.00000
62
2       -1  9.236987e-04      -1        -1        -1      0.00000
27
3       -1 -2.868090e-04      -1        -1        -1      0.00000
89
      Prediction
0 -0.0002868090
1 -0.0008139656
2  0.0009236987
3 -0.0002868090
> gbm.show.rules(ds.gbm)
Number of models: 100

Tree 1: Weight XXXX
  Proline < 867.50 : 0 (XXXX/XXXX)
  Proline >= 867.50 : 1 (XXXX/XXXX)
  Proline missing : 0 (XXXX/XXXX)
[...]
Tree 100: Weight XXXX
  Proline < 755.00 : 0 (XXXX/XXXX)
  Proline >= 755.00 : 1 (XXXX/XXXX)
  Proline missing : 0 (XXXX/XXXX)

```

22.5 Extensions and Variations

22.5.1 Alternating Decision Tree

An alternating decision tree [Freund and Mason \(1999\)](#), combines the simplicity of a single decision tree with the effectiveness of boosting. The knowledge representation combines tree stumps, a common model deployed in boosting, into a decision tree type structure. The different branches are no longer mutually exclusive. The root node is a prediction node, and has just a numeric score. The next layer of nodes are decision nodes, and are essentially a collection of decision tree stumps. The next layer then consists of prediction nodes, and so on, alternating between prediction nodes and decision nodes.

A model is deployed by identifying the possibly multiple paths from the root node to the leaves through the alternating decision tree that correspond to the values for the variables of an entity to be classified. The entity's classification score (or measure of confidence) is the sum of the prediction values along the corresponding paths. A simple example involving the variables Income and Deduction (with values \$56,378, and \$1,429, respectively), will result in a score of $0.15 - 0.25 + 0.5 - 0.3 = 0.1$. This is a positive number so we will place this entity into the positive class, with a confidence of only 0.1. The corresponding paths in Figure is highlighted.

We can build an alternating decision tree in R using the *RWeka* package:

```
# Load the sample dataset
> data(audit, package="rattle")
# Load the RWeka library
> library(RWeka)
# Create interface to Weka's ADTree and print some documentation
> ADT <- make_Weka_classifier("weka/classifiers/trees/ADTree")
> ADT
> WOW(ADT)
# Create a training subset
> set.seed(123)
> trainset <- sample(nrow(audit), 1400)
# Build the model
> audit.adt <- ADT(as.factor(Adjusted) ~ .,
                      data=audit[trainset, c(2:11,13)])
> audit.adt
Alternating decision tree:

: -0.568
```

```

| (1) Marital = Married: 0.476
| | (10) Occupation = Executive: 0.481
| | (10) Occupation != Executive: -0.074
| (1) Marital != Married: -0.764
| | (2) Age < 26.5: -1.312
| | | (6) Marital = Unmarried: 1.235
| | | (6) Marital != Unmarried: -1.366
| | (2) Age >= 26.5: 0.213
| (3) Deductions < 1561.667: -0.055
| (3) Deductions >= 1561.667: 1.774
| (4) Education = Bachelor: 0.455
| (4) Education != Bachelor: -0.126
| (5) Occupation = Service: -0.953
| (5) Occupation != Service: 0.052
| | (7) Hours < 49.5: -0.138
| | | (9) Education = Master: 0.878
| | | (9) Education != Master: -0.075
| | (7) Hours >= 49.5: 0.339
| (8) Age < 36.5: -0.298
| (8) Age >= 36.5: 0.153
Legend: -ve = 0, +ve = 1
Tree size (total number of nodes): 31
Leaves (number of predictor nodes): 21

```

Togaware watermark for Data Mining Survival

We can pictorially present the resulting model as in Figure 22.1, which shows a cut down version of the actual ADTree built above. We can explore exactly how the model works using the simpler model in Figure 22.1. We begin with a new instance and a starting score of -0.568 . Suppose the person is married and aged 32. Considering the right branch we add -0.298 . We also consider the left branch, we add 0.476 . Supposing that they are not an Executive, we add another -0.074 . The final score is then $-0.568 - 0.298 + 0.476 - 0.074 = -0.464$.

```

# Explore the results
> predict(audit.adt, audit[-trainset, -12])
> predict(audit.adt, audit[-trainset, -12], type="prob")
# Plot the results
> pr <- predict(audit.adt, audit[-trainset, c(2:11,13)], type="prob")[,2]
> eval <- evaluateRisk(pr, audit[-trainset, c(2:11,13)]$Adjusted,
+                         audit[-trainset, c(2:11,13,12)]$Adjustment)
> title(main="Risk Chart ADT audit [test] Adjustment",
+       sub=paste("Rattle", Sys.time(), Sys.info()["user"]))

```

22.6 Resources and Further Reading

Freund and Schapire (1995) introduced AdaBoost, popularising the idea of ensemble learning where a committee of models cooperate to deliver

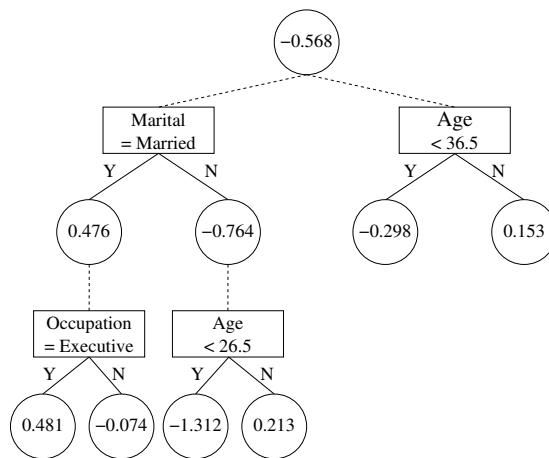


Figure 22.1: Reduced example of an alternating decision tree.

Togaware Watermark For Data Mining Survival

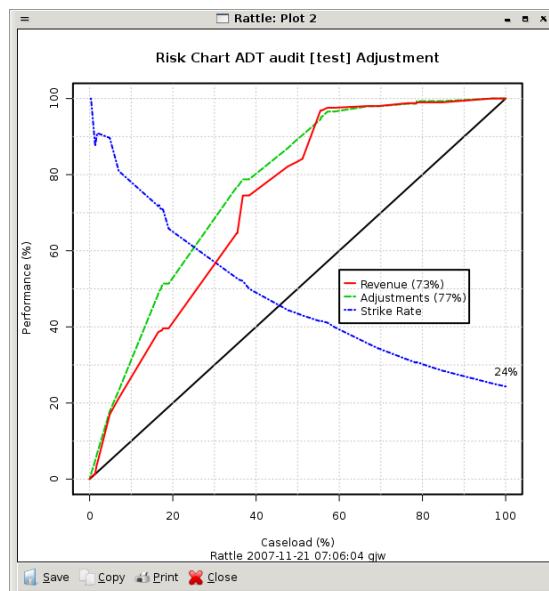


Figure 22.2: Audit risk chart from an alternating decision tree.

a better outcome. For a demonstration of AdaBoost visit <http://www1.cs.columbia.edu/~freund/adaboost/>.

The original formulation of the algorithm, as in [Freund and Schapire \(1995\)](#), adjusts all weights each iteration. Weights are increased if the corresponding record is misclassified by \mathcal{M}_i or decreased if it is correctly classified by \mathcal{M}_i . The weights are then further normalised each iteration to ensure they continue to represent a distribution (so that $\sum_{j=1}^n w_j = 1$). This can be simplified, as by [Hastie et al. \(2001\)](#), to only increase the weights of the misclassified entities. We use this simpler formulation in the above description of the algorithm. Consequently, the calculation of ϵ_i (line 9) includes the sum of the weights as a denominator (which is 1 in the original formulation). Only the weights associated with the misclassified entities are modified in line 11. The original algorithm modified all weights by $e^{-\alpha_i y_i \mathcal{M}_i(x_j)}$ which equates to e^{α_i} for misclassified entities (since either y_i or $\mathcal{M}_i(x_j)$ is -1, but not both) and to $e^{-\alpha_i}$ for correctly classified entities (since both y_i or $\mathcal{M}_i(x_j)$ are either 1 or -1). For each iteration the new weights in the original algorithm are normalised by dividing each weight by a calculated factor Z_i .

Cost functions other than the exponential loss criterion e^{-m} have been proposed. These include the logistic log-likelihood criterion $\log(1 + \exp(-m))$ used in LogitBoost), $1 - \tanh(m)$ (Doom II) and $(1-m)I(m > 1)$ (Support Vector Machines).

BrownBoost addresses the issue of the sensitivity of AdaBoost to noise.

We note that if each weak classifier is always better than chance, then AdaBoost can be proven to converge to a perfectly accurate model (no training error). Also note that even after achieving an ensemble model with no error, as we add in new models to the ensemble the generalisation error continues to improve (the margin continues to grow). Although it was thought, at first, that AdaBoost does not overfit the data, it has since been shown that it can. However, it generally does not, even for large numbers of iterations.

Extensions to AdaBoost include multi-class classification, application to regression (by transforming the problem into a binary classification task), and localised boosting which is similar to mixtures of experts.

Some early practical work on boosting was undertaken with the Aus-

Australian Taxation Office using boosted stumps. Multiple, simple models of tax compliance were produced. The models were easily and independently interpretable. Effectively, the models identified a collection of factors that in combination were useful in predicting compliance.

Togaware Watermark For Data Mining Survival

Chapter 23

Classification: Random Forests

Togaware Watermark For Data Mining Survival

23.1 Summary

23.2 Overview

A key factor about a random forest being a collection of many decision trees is that each decision tree is not influenced by the other decision trees when constructed.

23.3 Algorithm

23.4 Usage

The random forest model builder, as implemented in R:

```
> library(randomForest)  
>
```

23.4.1 Random Forest

importance

The importance option allows us to review the importance of each variable in determining the outcome. The first importance is the scaled average of the prediction accuracy of each variable, and the second is the total decrease in node impurities splitting on the variable over all trees, using the Gini index.

classwt

The *classwt* option in the current *randomForrest* package does not fully work and should be avoided. The *sampsize* and *strata* options can be used together. Note that if *strata* is not specified, the class labels will be used.

23.5 Examples

Here's an example using the iris data:

```
> iris.rf <- randomForest(Species ~ ., iris, sampsize=c(10, 20, 10))
```

This will randomly sample 10, 20 and 10 entities from the three classes of species (with replacement) to grow each tree.

You can also name the classes in the *sampsize* specification:

```
> samples <- c(setosa=10, versicolor=20, virginica=10)
> iris.rf <- randomForest(Species ~ ., iris, sampsize=samples)
```

You can do a stratified sampling using a different variable than the class labels so that you even up the distribution of the class. Andy Liaw gives an example of the multi-centered clinical trial data where you want to draw the same number of patients per center to grow each tree where you can do something like:

```
> randomForest(..., strata=center,
               sampsize=rep(min(table(center))), nlevels(center)))
```

This samples the same number of patients (minimum at any center) from each center to grow each tree.

To be confident that the random forest score is simply the proportion of positive examples, we can try building one tree, then multiple trees, and see what we get. We can start with a single tree (note that we use the Rattle generated commands, as listed in the Log tab, and thus we use the Rattle internal variables.

First build a single tree:

```
> set.seed(123)
> crs$rf <- randomForest(as.factor(Adjusted) ~ .,
  data=crs$dataset[crs$sample,c(2:10,13)],
  ntree=1, importance=TRUE, na.action=na.omit)
> crs$pr <- predict(crs$rf,
  crs$dataset[-crs$sample, c(2:10,13)],
  type="prob")[,2]
> summary(as.factor(crs$pr))
 0     1 NA's
423   139   38
```

Now build two trees and rerun the code:

```
> set.seed(123)
> crs$rf <- randomForest(as.factor(Adjusted) ~ .,
  data=crs$dataset[crs$sample,c(2:10,13)],
  ntree=2, importance=TRUE, na.action=na.omit)
> crs$pr <- predict(crs$rf,
  crs$dataset[-crs$sample, c(2:10,13)],
  type="prob")[,2]
> summary(as.factor(crs$pr))
 0  0.5     1 NA's
353 124    85   38
```

And then four trees:

```
> set.seed(123)
> crs$rf <- randomForest(as.factor(Adjusted) ~ .,
  data=crs$dataset[crs$sample,c(2:10,13)],
  ntree=4, importance=TRUE, na.action=na.omit)
> crs$pr <- predict(crs$rf,
  crs$dataset[-crs$sample, c(2:10,13)],
  type="prob")[,2]
> summary(as.factor(crs$pr))
 0  0.25  0.5  0.75     1 NA's
293   98    68   62    41   38
```

Thus, we can see that when we have four trees voting, the score will be either 0 (no tree voted in favour of the case), 0.25 (one tree in favour),

0.5 (two trees in favour). 0.75 (three trees in favour), and 1 (all trees in favour).

23.6 Resources and Further Reading

Random forests can also be used in an unsupervised mode for clustering. See Unsupervised Learning with Random Forest Predictors at <http://www.genetics.ucla.edu/labs/horvath/RFclustering/RFclustering.htm>.

Togaware Watermark For Data Mining Survival

Chapter 24

Issues

In this chapter we cover a number of topics around building models in
data mining.

24.1 Incremental or Online Modelling

The modelling approaches we have discussed here are what we might think of as batch learners. The model builders take a training dataset to build a model that might then be deployed. If we want to update the model then the process begins again from a new training dataset, in batch mode, to build a new model.

An alternative paradigm is to incrementally build a model. Such approaches are also often referred to as online model building.

24.2 Model Tuning

What is the right value to use for each of the variables of the model building algorithms that we use in data mining? The variable settings can make the difference between a good and a poor model.

The package *caret*, as well as providing a unified interface to many of

the model builders we have covered in this book, provides a parameter tuning approach. Here's a couple of examples:

```
> library(rattle)
> library(caret)
> data(audit)
> mysample <- sample(nrow(audit), 1400)
> myrpart <- train(audit[mysample, c(2,4:5,7:10)],
+                   as.factor(audit[mysample, c(13)]), "rpart")
Model 1: maxdepth=6
  collapsing over other values of maxdepth
> myrpart
Call:
train.default(x = audit[mysample, c(2, 4:5, 7:10)], y = as.factor(audit[mysample,
  c(13)]), method = "rpart")

1400 samples, 7 predictors

largest class: 77.71% (0)

summary of bootstrap (25 reps) sample sizes:
  1400, 1400, 1400, 1400, 1400, ...

boot resampled training results across tuning parameters:

  maxdepth Accuracy Kappa Accuracy SD Kappa SD Optimal
    2       0.817   0.423  0.0142     0.0386
    3       0.818   0.413  0.0171     0.0617      *
    6       0.814   0.412  0.019      0.0488

Accuracy was used to select the optimal model
> myrpart$finalModel
n= 1400

node), split, n, loss, yval, (yprob)
  * denotes terminal node

  1) root 1400 312 0 (0.77714286 0.22285714)
     2) Marital=Absent,Divorced,Married-spouse-absent,Unmarried,Widowed 773
38 0 (0.95084088 0.04915912) *
     3) Marital=Married 627 274 0 (0.56299841 0.43700159)
        6) Education=College,HSgrad,Preschool,Vocational,Yr10,Yr11,Yr12,Yr1t4,Yr5t6,Yr7t8,Yr9 409
           12) Deductions< 1708 400 120 0 (0.70000000 0.30000000) *
           13) Deductions>=1708 9 0 1 (0.00000000 1.00000000) *
        7) Education=Associate,Bachelor,Doctorate,Master,Professional 218
73 1 (0.33486239 0.66513761) *
```

Similarly we can replace rpart with rf.

The *tune* function from the *e1071* package provides a simple, if sometimes computationally expensive, approach to find a good value for a collection of tuning variables. We explore the use of this function here.

The *tune* function provides a number of global tuning variables that affect how the tuning happens. The *nrepeat* variable (number of repeats) specifies how often the training should be repeated. The *repeat.aggregate* variable identifies a function that specifies how to combine the training results over the repeated training. The *sampling* identifies the sampling scheme to use, allowing for cross-validation, bootstrapping or a simple train/test split. For each type of sample, further variables are supplied, including, for example, *cross* = 10 to set the cross validation to be 10-fold. The *sampling.aggregate* variable specifies a function to combine the training results over the various training samples. A good default (provided by *tune*) is to train once with 10-fold cross validation.

24.2.1 Tuning rpart

To keep the examples simple we use the audit dataset and remove entities with missing values and also ignore the Adjustment column.

```
library(e1071)
audit <- na.omit(read.csv("audit.csv"))
audit$Adjustment <- NULL
fm <- formula(Adjusted ~ ID+Age+Employment+Education+
               Marital+Occupation+Income+Sex+
               Deductions+Hours+Accounts)
# Explore minsplit
audit.rpart <- tune.rpart(fm, data=audit, minsplit=seq(10,100,10))
plot(audit.rpart, main="Tune rpart on minsplit")

# cp
audit.rpart <- tune.rpart(fm, data = audit, cp = c(0.002,0.005,0.01,0.015,0.02,0.03))
plot(audit.rpart,main="Performance of rpart vs. cp")
readline()

# maxdepth
audit.rpart <- tune.rpart(fm, data = audit, maxdepth = 1:5)
plot(audit.rpart,main="Performance of rpart vs. cp")
readline()
```

24.3 Unbalanced Classification

This is a common problem that we find in areas such as fraud, rare disease diagnosis, network intrusion, and others. The problem is that one class is very much underrepresented in the data. For example, cases of fraud in a very large medical insurance dataset are perhaps less than 1%. In compliance work where claims are being reviewed for compliance, often the number of claims that require adjustment is perhaps only 10%. In such circumstances, if we build a model in the usual way, where the aim is to minimise error rates, we can build the most accurate model to say that there is no fraud, and the model is up to 99% accurate, but of very little use.

Data mining of unbalanced datasets will often involve adjustments to the modelling in some way. One approach is to down sample the majority case to even up the classes. Alternatively, we might over sample entities from the rare class and by so doing increase the weight of the minorities! Such approaches can work, but it is not always clear that they will. Under-sampling can lead to a loss of information, whilst over-sampling may lead to over-fitting. Although, adaptive under-sampling can lead to a reduced loss of information, producing better results than over-sampling, and is more efficient.

An important thing to know when we have an unequal distribution of negative and positive cases is the misclassification cost—that is, what is the cost of incorrectly classifying a positive case as a negative (a false negative) and of incorrectly classifying a negative as a positive (a false positive). Often these will be different. In fraud for example, it is important to ensure we identify all cases of fraud, and we might be willing to accept that we will have some false positives. Thus false negatives have a very high cost. If the misclassification cost is equal for both false positives and false negatives then a reasonable strategy is simply to minimise the number of misclassified examples (regardless of whether they belong to the majority class or the minority class).

We illustrate two approaches to dealing with unbalanced datasets in Chapter ??, page ?. There, one approach is to modify the weights, and the second is to down sample to balance up the classes. Both have been found to be very effective approaches when coupled with random forests.

24.4 Building Models

To prevent overfitting and under estimating the generalisation error during training, use a training/test set paradigm.

24.5 Outlier Analysis

Rare, unusual, or just plain infrequent events are of interest in data mining in many contexts including fraud in income tax, insurance, and online banking, as well as for marketing. We classify analyses that focus on the discovery of such data items as *outlier analysis*. [Hawkins \(1980\)](#) captures the concept of an outlier as:

an observation that deviates so much from other observations as to arouse suspicion that it was generated by a different mechanism.

Outlier detection algorithms often fall into one of the categories of distance-based methods, density-based methods, projection-based methods, and distribution-based methods.

A general approach to identifying outliers is to assume a known distribution for the data and to examine the deviation of individuals from the distribution. Such approaches are common in statistics ([Barnett and Lewis, 1994](#)) but such approaches do not scale well.

Distance based methods are common in data mining where the measure of an entities outliedness is based on its distance to nearby entities. The number of nearby entities and the minimum distance are two parameters. (see knorr and ng 1998 vldb24)

Density based approaches from breuning kriegel ng and sander 2000 sigmod LOF: local outlier factor. See also jin tung and han kdd2001.

The early work on outliers was carried out from a statistical view point where outliers are data points that deviate significantly from the identified underlying distribution of the data. [Hawkins \(1980\)](#) is a good

textbook on outliers. Barnett and Lewis (1994) is another overview of statistical outliers.

Distance based approaches have been developed by Knorr and Ng (1998), Ramaswamy et al. (2000) and Knorr and Ng (1999). Such approaches usually explore some neighbourhood and do not rely on underlying distributions.

Knorr and Ng (1998) identify outliers by counting the number of neighbours within a specified radius of a data point. The radius q and the threshold number of points n are the only two parameters of the approach. The approach is simple but is inadequate for data that is distributed with uneven density where q and n might need to vary to cope with the changes. Ramaswamy et al. (2000) have a similar approach whereby data points are ranked by the sum of their distance to their nearest k neighbours.

Breunig et al. (1999) and then Breunig et al. (2000) introduce a density based approach to score data points with a local outlier factor (LOF). Jin et al. (2001) introduce a heuristic to more efficiently identify the top outliers using the LOF.

Yamanishi et al. (2000) build mixture models as data becomes available and identifies outliers as those data items causing the most perturbation to the model.

Aggarwal and Yu (2001) explore the issue of outliers in high dimensional space where data tends to be sparse and consequently all data points tend to be equidistant to other points (Beyer et al., 1999) and suggest an algorithm where the high dimensional space is projected to a lower dimensional space having unusually low density. An evolutionary algorithm is proposed to generate candidate subspaces in which outliers are to be searched for.

SmartSifter

24.6 Temporal Analysis

A common data mining task with temporal data is to find repeating patterns in the data - see frequent closed itemsets.

SNN Clustering [Lin et al. \(2000\)](#) and [Lin et al. \(2001\)](#).

A common question is how likely an event will occur at a give point in time. Suppose we had some simple churn data recording how long a customer has been with a telecoms provider before they churned.

ID	Gender	Months	Churn
1	M	12	1
2	M	5	0
3	M	32	1
4	M	4	0
5	M	10	1
6	F	12	0
7	F	5	1
8	F	15	0
9	F	5	1
10	F	12	0

We may be tempted in the first instance to us a logistic regression including `Gender` and `Months` to predict `Churn` with:

$$\text{logit}(Churn = 1) = b_0 + b_1 * \text{Gender} + b_2 * \text{Tenure}$$

24.7 Survival Analysis

We note though that those who have not churned in fact have not *yet* churned! They may churn in the future. We don't know. In such a situation we have what is called censored data and so survival analysis is more appropriate. Survival analysis is analysis of the time to an event and the methods used for survival analysis take in to account the fact that we only have partial information available to us. The partial information for customer 2, for example, is that we know they have been with us for 5 months, but we don't know whether they might be just about to churn or not.

Time to event modelling often uses Survival Analysis ([Klein and Moeschberger,](#)

2003, Second Edition, Survival Analysis: Techniques for Censored and Truncated Data , Springer). Survival analysis models the time to the occurrence of an event (e.g., time to death, time to failure, time to lodgment, time to churn, etc.). It is particularly useful when we have censored observations. The general idea approach introduces a survival function $S(t)$ and a hazard rate function $\lambda(t)$. These describe the status of an entities survival during the period of observation. The survival function gives the probability of surviving beyond a certain point t . The hazard rate function gives the instantaneous risk of non-survival (i.e., death, churn, lodgment, failure) at time t given survival to time t .

The Cox PH model is survival analysis.

Togaware Watermark For Data Mining Survival

Chapter 25

Evaluating Models

Evaluating the outcomes of data mining is important. We need to understand how well any model we build will be expected to perform, and how well it performs in comparison to other models we might choose to build.

A common approach is to compute an error rate which simply reports the number of cases that the model correctly classifies. Common methods for estimating the empirical error rate are, for example, cross-validation (CV), the Bayesian evidence framework, and the PAC framework.

In this chapter we introduce several measures used to report on the performance of a model and review various approaches to evaluating the output of data mining. This will cover *printcp*, *table* for producing confusion matrices, *ROCR* for the graphical presentation of evaluations, as well as how to tune the presentations for your own needs.

25.1 Basics

Use *printcp* to view the performance of the model.

```
> printcp(wine.rpart)
Classification tree:
rpart(formula = Type ~ ., data = wine)
```

```
Variables actually used in tree construction:
[1] Dilution   Flavanoids Hue         Proline

Root node error: 107/178 = 0.60112

n= 178

      CP nsplit rel error xerror     xstd
1 0.495327      0    1.00000 1.00000 0.061056
2 0.317757      1    0.50467 0.47664 0.056376
3 0.056075      2    0.18692 0.28037 0.046676
4 0.028037      3    0.13084 0.23364 0.043323
5 0.010000      4    0.10280 0.21495 0.041825
```

The *predict* function will apply the model to data. The data must contain the same variable on which the model was built. If not an error is generated. This is a common problem when wanting to apply the model to a new dataset that does not contain all the same variables, but does contain the variables you are interested in.

```
> cols <- c("Type", "Dilution", "Flavanoids", "Hue", "Proline")
> predict(wine.rpart, wine[,cols])
Error in eval(expr, envir, enclos) : Object "Alcohol" not found
```

Fix this up with

```
> wine.rpart <- rpart(Type ~ Dilution + Flavanoids + Hue + Proline,
  data=wine)
> predict(wine.rpart, wine[,cols])
      1          2          3
1  0.96610169 0.03389831 0.00000000
2  0.96610169 0.03389831 0.00000000
[...]
70  0.03076923 0.93846154 0.03076923
71  0.00000000 0.25000000 0.75000000
[...]
177 0.00000000 0.25000000 0.75000000
178 0.00000000 0.02564103 0.97435897
```

Display a confusion matrix.

```
> table(predict(wine.rpart, wine, type="class"), wine$Type)
      1   2   3
1 57   2   0
2  2 66   4
3  0   3 44
```

25.2 Basic Measures

True positives (TPs) are those records which are correctly classified by a model as positive instances of the concept being modelled (e.g., the model identifies them as a case of fraud, and they indeed are a case of fraud). **False positives** (FPs) are classified as positive instances by the model, but in fact are known not to be. Similarly, **true negatives** (TNs) are those records correctly classified by the model as not being instances of the concept, and **false negatives** (FNs) are classified as not being instances, but are in fact known to be. These are the basic measures of the performance of a model. These basic measures are often presented in the form of a [confusion matrix](#), produced using a [contingency table](#).

In the following example a simple decision tree model, using *rpart*, is built using the *survey* dataset to predict *Salary.Group*. The model is then applied to the full dataset using *predict*, to predict the class of each entity (using the *type* option to specify **class** rather than the default probabilities for each class). A confusion matrix is then constructed using *table* to build a contingency table. Note the use of named parameters (**Actual** and **Predicted**) to have these names appear in the table.

```
> load("survey.RData")
> survey.rp <- rpart(Salary.Group ~ ., data=survey)
> survey.pred <- predict(survey.rp, data=survey, type="class")
> head(survey.pred)
[1] <=50K >50K <=50K <=50K >50K >50K
Levels: <=50K >50K
> table(Actual=survey$Salary.Group, Predicted=survey.pred)
   Predicted
Actual    <=50K >50K
  <=50K 23473 1247
  >50K   3816 4025
```

From this confusion matrix, interpreting the class $\leq 50K$ as the positive class (essentially arbitrarily), we see that there are 23,473 true positives, 4,025 true negatives, 3,816 false positives, and 1,237 false negatives.

Rather than the raw numbers we usually prefer to express these in terms of percentages or rates. The accuracy of a model can, for example, be calculated as the number of entities correctly classified over the total number of entities classified:

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

The **recall** or **true positive rate** is the proportion of positive entities which are classified as positive by the model:

$$\text{Recall} = \frac{TP}{TP + FN}$$

The recall is a measure of how much of the positive class was actually recovered by the model.

25.3 Cross Validation

In R see the `errorest()` function in the `ipred` package.

Cross validation is a method for estimating the true error of a model. When a model is built from training data, the error on the training data is a rather optimistic estimate of the error rates the model will achieve on unseen data. The aim of building a model is usually to apply the model to new, unseen data—we expect the model to generalise to data other than the training data on which it was built. Thus, we would like to have some method for better approximating the error that might occur in general. Cross validation provides such a method.

Cross validation is also used to evaluate a model in deciding which algorithm to deploy for learning, when choosing from amongst a number of learning algorithms. It can also provide a guide as to the effect of parameter tuning in building a model from a specific algorithm.

Test sample cross-validation is often a preferred method when there is plenty of data available. A model is built from a training set and its predictive accuracy is measured by applying the model a test set. A good rule of thumb is that a dataset is partitioned into a training set (66%) and a test set (33%).

To measure error rates you might build multiple models with the one algorithm, using variations of the same training data for each model. The average performance is then the measure of how well this algorithm works in building models from the data.

The basic idea is to use, say, 90% of the dataset to build a model. The data that was removed (the 10%) is then used to test the performance

of the model on “new” data (usually by calculating the **mean squared error**). This simplest of cross validation approaches is referred to as the **holdout method**.

For the **holdout method** the two datasets are referred to as the **training set** and the **test set**. With just a single evaluation though there can be a high variance since the evaluation is dependent on the data points which happen to end up in the training set and the test set. Different partitions might lead to different results.

A solution to this problem is to have multiple subsets, and each time build the model based on all but one of these subsets. This is repeated for all possible combinations and the result is reported as the average error over all models.

This approach is referred to as **k-fold cross validation** where k is the number of subsets (and also will be the number of models built). Research indicates that there is little to gain by using more than 10 partitions, so usually $k = 10$. That is, the available data is partitioned into 10 subsets (each contains 10% of the available data). The holdout method is then replicated k times, each time combining $k - 1$ (i.e., 9) subsets to form the training set (consisting of 90% of the original data), and the remaining subset (10%) is the test set.

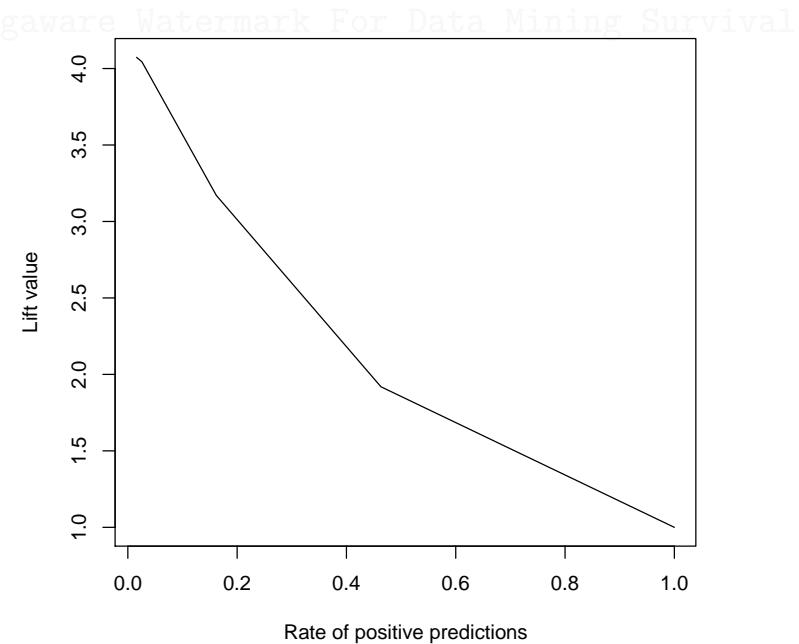
Some prefer test sample cross-validation where a classification tree is built from a training dataset and the predictive accuracy is tested by predicting on a test dataset. The costs for the test dataset are compared to those for training dataset (cost is the proportion of misclassified cases when priors are estimated and misclassification costs are equal). Poor cross-validation when test costs are hight.

k-fold cross-validation is useful when no test dataset is available (e.g., the available dataset is too small). k is the number of nearly equal sized random subsamples. Build model k times leaving out one of the subsamples each time. The remaining subsample is used as a test dataset for cross-validation. The cross validation costs computed for each of the k test samples are then averaged to give the k-fold estimate of the cross validation costs.

25.4 Graphical Performance Measures

ROC graphs, sensitivity/specificity curves, lift charts, and precision/recall plots are useful in illustrating specific pairs of performance measures for classifiers. The *ROCR* package creates 2D performance curves from any two of over 25 standard performance measures. Curves from different cross-validation or bootstrapping runs can be averaged by different methods, and standard deviations, standard errors or box plots can be used to visualize the variability across the runs. See `demo(ROCR)` and <http://rocr.bioinf.mpi-sb.mpg.de/> for examples.

25.4.1 Lift



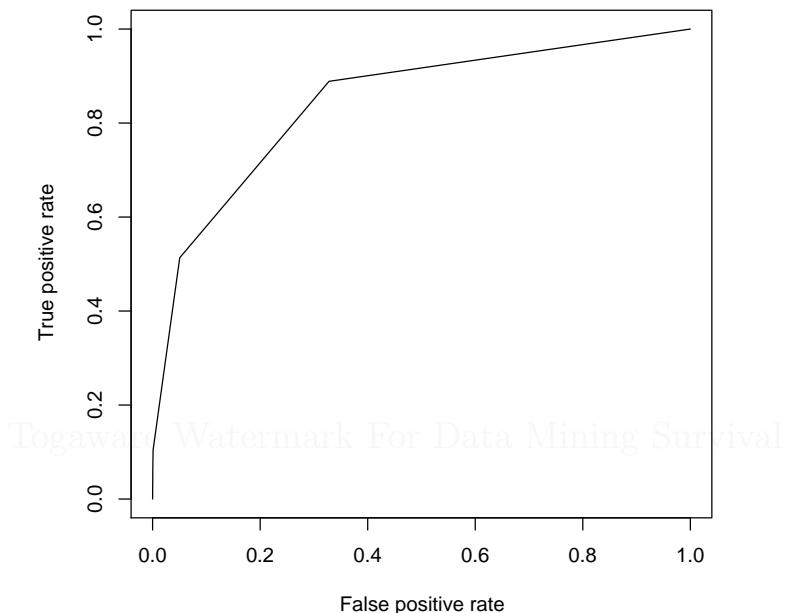
```
pdf("graphics/rplot-rocr-survey-lift.pdf")
library(rpart)
library(ROCR)
load("survey.Rdata")
```

```
survey.rp <- rpart(Salary.Group ~ ., data=survey)
survey.pred <- predict(survey.rp, data=survey)
pred <- prediction(survey.pred[,2], survey$Salary.Group)
lift <- performance(pred, "lift", "rpp")
plot(lift)
dev.off()
```

R code source: [rplot-rocr-survey-lift.R](#).

Togaware Watermark For Data Mining Survival

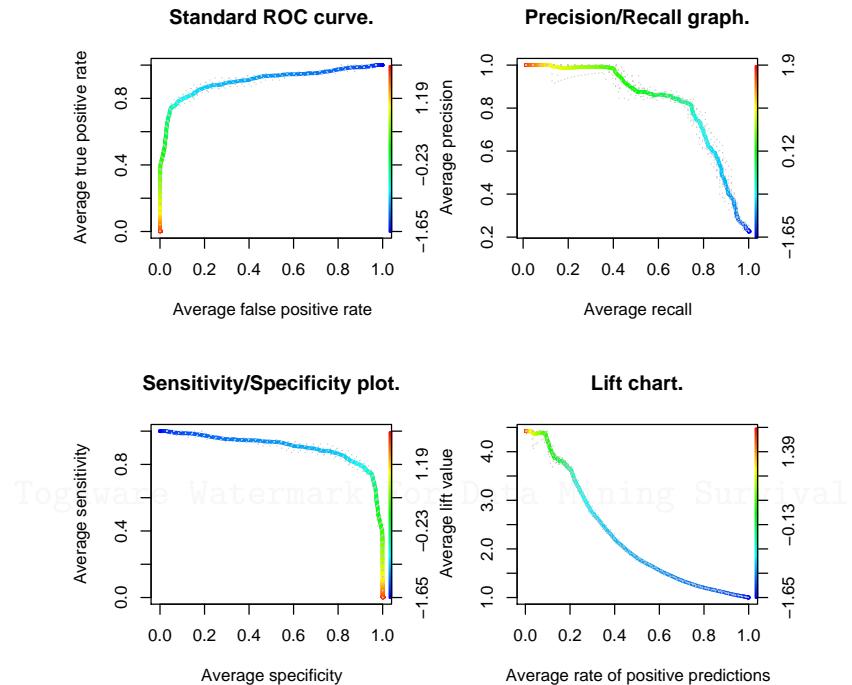
25.4.2 The ROC Curve



```
pdf("graphics/rplot-rocr-survey-tpfp.pdf")
library(rpart)
library(ROCR)
load("survey.Rdata")
survey.rp <- rpart(Salary.Group ~ ., data=survey)
survey.pred <- predict(survey.rp, data=survey)
pred <- prediction(survey.pred[,2], survey$Salary.Group)
perf <- performance(pred, "tpr", "fpr")
plot(perf)
dev.off()
```

R code source: [rplot-rocr-survey-tpfp.R](#).

25.4.3 Other Examples



```
# Based on code from demo(ROCR)
library(ROCR)
data(ROCR.hiv)
pp <- ROCR.hiv$hiv.svm$predictions
ll <- ROCR.hiv$hiv.svm$labels
pred <- prediction(pp, ll)
perf <- performance(pred, "tpr", "fpr")
pdf("graphics/rplot-rocr-4plots.pdf")
par(mfrow = c(2, 2))
plot(perf, avg = "threshold", colorize = T, lwd = 3,
     main = "Standard ROC curve.")

plot(perf, lty = 3, col = "grey78", add = T)
perf <- performance(pred, "prec", "rec")
plot(perf, avg = "threshold", colorize = T, lwd = 3,
     main = "Precision/Recall graph.")

plot(perf, lty = 3, col = "grey78", add = T)
perf <- performance(pred, "sens", "spec")
plot(perf, avg = "threshold", colorize = T, lwd = 3,
     main = "Sensitivity/Specificity plot.")
```

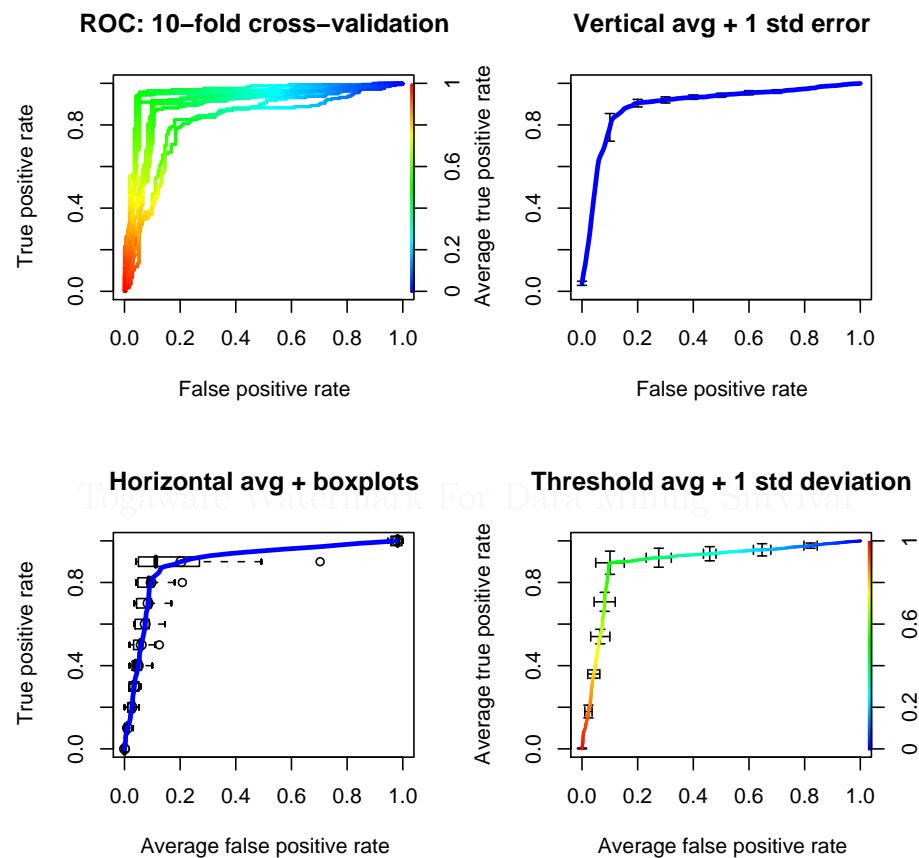
```
plot(perf, lty = 3, col = "grey78", add = T)
perf <- performance(pred, "lift", "rpp")
plot(perf, avg = "threshold", colorize = T, lwd = 3,
     main = "Lift chart.")

plot(perf, lty = 3, col = "grey78", add = T)
dev.off()
```

R code source: [rplot-rocr-4plots.R](#).

Togaware Watermark For Data Mining Survival

10 Fold Cross Validation



```

library(ROCR)
data(ROCR.xval)
pp <- ROCR.xval$predictions
ll <- ROCR.xval$labels
pred <- prediction(pp, ll)
perf <- performance(pred, "tpr", "fpr")
pdf("graphics/rplot-rocr-10xfold.pdf")
par(mfrow = c(2, 2))
plot(perf, colorize = T, lwd = 2,
     main = "ROC: 10-fold cross-validation")

plot(perf, avg = "vertical", spread.estimate = "stderror",
      lwd = 3, main = "Vertical avg + 1 std error",
      col = "blue")

plot(perf, avg = "horizontal", spread.estimate = "boxplot",
      lwd = 3, main = "Horizontal avg + boxplots",
      col = "blue")

plot(perf, avg = "threshold", spread.estimate = "stddev",
      lwd = 2, main = "Threshold avg + 1 std deviation",
      colorize = T)
dev.off()

```

R code source: [rplot-rocr-10xfold.R](#).

Area Under Curve

The area under a curve examples:

```

> library(Hmisc)
> set.seed(1)
> predicted <- runif(200)
> dead <- sample(0:1, 200, TRUE)
> roc.area <- somers2(predicted, dead)()["C"]

```

25.5 Calibration Curves

Chapter 26

Cluster Analysis

Togaware Watermark For Data Mining Survival

Togaware Watermark For Data Mining Survival

Part III

Text Mining

Togaware Watermark For Data Mining Survival

Chapter 27

Text Mining

A common procedure for text mining is to ‘score’ each document by a vector that records the frequency of occurrence of commonly used and subject matter specific words and phrases. Assuming the documents are themselves classified into a number of classes already (perhaps those that are relevant versus those that are not) you can use this “training set” with any of the many supervised learning or classification tools in R (e.g., trees, logistic regression, boosting, Random Forests, support vector machines, linear discriminant analysis, etc.).

27.1 Text Mining with R

See *ttda* and *tm*.

Text mining begins with feature extraction. Techniques include:

- Keyword extraction
- Bag of words
- Term weighting
- Co-occurrence of words

Using *tm*, here is a simple example. The *crude* dataset contains 20 news articles dealing with crude oil. The data type of the dataset is identified as a text document collection (TextDocCol). We can create our own text document collections using functions provided by the *tm* package which will read a collection of source documents from a specified directory, and process them into a TextDocCol. We can then take the TextDocCol and using *TermDocMatrix* generate a weighted count of terms in the documents (remove the weight argument if you just want to use term counting).

The actual data is :

```
> library(tm)
> vignette("tm")
> data(crude)
> class(crude)
[1] "TextDocCol"
attr(,"package")
[1] "tm"
> crude
A text document collection with 20 text documents
> crude@.Data
[[1]]
[1] "Diamond Shamrock Corp said that \neffective [...]"  

[[2]]
[1] "OPEC may be forced to meet before a \nscheduled [...]"  

[...]  

[[20]]
[1] "Argentine crude oil production was \ndown 10.8 pct [...]"  

> tdm <- TermDocMatrix(crude, weighting = "tf-idf", stopwords = TRUE)
An object of class "TermDocMatrix"
Slot "Data":
20 x 859 sparse Matrix of class "dgCMatrix"
 [[ suppressing 859 column names 'barrel', 'brings', 'citing' ... ]]  

127 2 2.321928 4.321928 2.736966 2 4.643856 4.321928 2.736966
144 . . . . . . . .  

[...]  

> tdm <- TermDocMatrix(crude, stopwords = TRUE)  

> tdm
An object of class "TermDocMatrix"
Slot "Data":
20 x 859 sparse Matrix of class "dgCMatrix"
 [[ suppressing 859 column names 'barrel', 'brings', 'citing' ... ]]
```

```
127 2 1 1 1 1 2 1 1 2 2 1 2 2 1 1 1 1 1 1 1 5 2 2 3 1 2  
144 . . . 1 . . . . . . . . . . . . . 4 1 12 . 1 5 . .  
191 1 1 . . 1 1 . . 2 . . . 1 1 . . 1 . . . 2 1 2 . . .  
194 1 1 . . 1 1 . . 3 . . . 2 1 . 1 . . . 1 1 2 . . .  
[...]
```

To transform `tdm` into a simple matrix to save the word counts or to compute various measures, such as to calculate the Euclidian distance:

```
> x <- as.matrix(tdm@Data)  
> write.csv(x, "crude_words.csv")  
> dist(x, method = "euclidean")
```

Togaware Watermark For Data Mining Survival

Togaware Watermark For Data Mining Survival

Part IV

Algorithms

Togaware Watermark For Data Mining Survival

THESE ARE BEING MOVED INTO THE R FOR THE DATA MINER
BOOK.

Togaware Watermark For Data Mining Survival

Togaware Watermark For Data Mining Survival

Chapter 28

Bagging: Meta Algorithm

Togaware Watermark For Data Mining Survival

Bagging is a variance reduction method for model building. That is, through building multiple models from samples of the training data, the aim is to reduce the variance. Bagging is a technique generating multiple training sets by sampling with replacement from the available training data. In an ideal world we can eliminate variance due to a particular choice of training set by combining models that are built from each training set of size N . In practise only one training set is available. By sampling with replacement from the training set to form new training sets, bagging simulates the ideal situation. Bagging is also known as bootstrap aggregating. See Chapter 28, page 479



28.1 Summary

A good introduction is available from http://www.idiap.ch/~bengio/lectures/tex_ensemble.pdf

Bagging is bootstrap aggregation. The underlying idea is that part of the error due to variance in building a model comes from the specific choice of the training dataset. So create many similar training data sets, and for each of them train a new function. The final function will then be the average of each functions output.

28.2 Overview

28.3 Example

28.4 Algorithm

28.5 Resources and Further Reading

Togaware Watermark For Data Mining Survival

The term originates with Breiman (1996).

Chapter 29

Bayes Classifier: Classification

Togaware Watermark For Data Mining Survival

Bayes classifiers came in two varieties: naïve and full. Naïve Bayes $\mathcal{P}(\text{MoulinRouge}|\dots) \approx$ is a technique for estimating probabilities of individual variable values, given a class, from training data and to then allow the use of these probabilities to classify new entities. Naïve Bayes has been demonstrated usefully on moderate and large datasets. It can be used for diagnosis and classification tasks. Despite the fact that the assumption of conditional independence is often violated the approach continues to work well. Full Bayes ...

29.1 Summary

Output Probabilities of outcomes.

Complexity Computationally efficient, but at the expense of accuracy.

29.2 Example

Suppose we have a database of patients for whom we have, in the past, identified suitable types of contact lenses. Four categorical variables might describe each patient: *Age* (having possible values *young*, *presbyopic*, and *hypermetropic*), type of *Prescription* (*myope* and *hypermetropic*), whether the patient is *Astigmatic* (boolean), and *TearRate* (*reduced* or *normal*). The patients are already classified into one of three classes indicating the type of contact lens to fit: *hard*, *soft*, *none*.

Sample data is presented in Table 29.1.

Table 29.1: Contact lens training data.

We can make use of this historic data to talk about the probabilities of requiring *soft*, *hard*, or no lenses, according to a patient's *Age*, *Prescription*, *Astigmatic* condition and *TearRate*. Having a probabilistic model we could then make predictions using the model about the suitability of particular lens types for new clients. Thus, if we know historically that the probability of a patient requiring a soft contact lens, given that they were *young* and had a *normal* rate of tear production, was 0.9 (i.e., 90%) then we would supply soft lens to most such patients in the future. Symbolically we write this probability as:

$$P(\text{soft}|\text{young}, \text{normal}) = 0.9$$

Generally the real problem is to determine something like:

$$P(\text{soft}|\text{young}, \text{myope}, \text{no}, \text{normal})$$

To do this we would need to have a sufficient number of examples of $(\text{young}, \text{myope}, \text{no}, \text{normal})$ in the database. For databases with many variables, and even for very large databases, it is not likely that we will have sufficient (or even any) examples of all possible combinations of all variable values. This is where naïve Bayes helps.

29.3 Algorithm

The basis of naïve Bayes is *Bayes theorem*. Essentially we want to classify a new record based on probabilities estimated from the training data.

That is, we want to determine the probability of a hypothesis h (or specifically class membership) given the training data D (i.e., $P(h|D)$) Bayes theorem gives us the clue to calculating this value:

$$P(h|D) = \frac{P(D|h)P(h)}{P(D)}$$

Ignoring the detail we can work with $P(D|h)P(h)$ to identify a hypothesis that best matches our data (since $P(D)$ is constant for a particular D). Both of these quantities we can estimate from the training data D . $P(h)$ is simply the proportion of the database consistent with hypothesis h (i.e., the proportion of entities with class c_i — c_i is regarded as the hypothesis).

Calculation of $P(D|h)$ still poses some challenges and this is where the naïve part of naïve Bayes comes in. The naïve assumption is that the variables (each record D is described by variables a_1, a_2, \dots, a_n) are *conditionally independent* given the class (i.e., the hypothesis that the classification is c_j , which could be the class *soft* in our example data) of the record. That is, given that a patient has a *soft* contact lens, to use our example, the probability of the patient being all of *young*, *myope*, *non-astigmatic* and with a *normal-tear-rate* is the same as the product of the individual probabilities. The naïve assumption, more concretely, is that being *astigmatic* or not, for example, does not affect the relationship between being *young* given the use of *soft* lenses. Mathematically we write this as:

$$P(a_1, a_2, \dots, a_n | c_j) = \prod_i P(a_i | c_j)$$

Empirically determining the values of the joint probability on the left of this equation is a problem. To estimate it from the data we need to have available in the data every possible combination of values and examples of their classification so we can then use these frequencies to estimate the probabilities. This is usually not feasible. However, the collection of probabilities on the right poses little difficulty. We can easily obtain the estimates of these probabilities by counting their occurrence in the database.

For example, we can count the number of patients with *Age* being *young*

and belonging to class *soft* (perhaps there are only two) and divide this by the number of patients overall belonging to class *soft* (perhaps there are five). The resulting estimate of the probability (i.e., $P(young|soft)$) is 0.4.

The Naïve Bayes algorithm is then quite simple. From the training data we estimate the probability of each class, $P(c_j)$, by the proportions exhibited in the database. Similarly the probabilities of each variable's value, given a particular class ($P(a_i|c_j)$), is simply the proportion of those training entities with that class having the particular variable value.

Now to place a new record (d) into a class we simply assign it to the class with the highest probability. That is, we choose the c_j which maximises:

$$P(c_j) \prod_{a_i \in d} P(a_i|c_j)$$

29.4 Resources and Further Reading

The example data for contact lenses comes from Cendrowska (1987) and is available from the machine learning repository at <ftp://ftp.ics.uci.edu/pub/machine-learning-databases/lenses/lenses.data>.

A problem with naïve Bayes arises when the training database has no examples of a particular value of a variable for a particular class.

Bayesian networks relax the conditional independence assumption by identifying conditional independence among subsets of variables.

Kohavi (1996) addressed the problem of independence by combining naïve Bayes with decision trees. The decision tree is used to partition a database and for each resulting partition (corresponding to separate paths through the decision tree) a naïve Bayes classifier is built using variables not included in the corresponding path through the decision tree. Whilst some improvement in accuracy can result, the final knowledge structures tend to be less compact (with replicated structures). Nonetheless this may be a useful approach for very large databases.

Chapter 30

Bootstrapping: Meta Algorithm

Togaware Watermark For Data Mining Survival

Booststrapping, introduced by Efron in 1979, brings together ideas of resampling and simulation-based statistical analysis. The aim is to understand bias, variance, and other measures of uncertainty through computer simulations.



30.1 Summary

<i>Market</i>	FOR WHICH MARKET IS THIS PRODUCT AIMED.
<i>Techniques</i>	LIST TECHNIQUES IMPLEMENTED
<i>Platforms</i>	GNU/Linux, Unix, MacOS, MSWindows.
<i>Website</i>	http://
<i>Pricing</i>	Available from vendor. Freely available—GNU General Public License.
<i>Vendor</i>	NAME OF VENDOR, LOCATION.

30.2 Usage

PROVIDE AN OVERVIEW OF DEPLOYMENT OF THE PRODUCT, INCLUDING SAMPLE APPLICATIONS AND SAMPLE CUSTOMERS.

30.3 Further Information

POINTERS TO FURTHER INFORMATION INCLUDING PERHAPS: OVERVIEW OF COMPANY/DEVELOPER; ADDRESS; EMAIL ADDRESS.

Some material in this section was provided by the vendor and has been used in this book with their express permission.

Bootstrap Methods and Their Applications by *Anthony C. Davison and David V. Hinkley*. 256 pages published by Cambridge University Press, 1994, ISBN 0-521-57391-2. The R package `boot` is based on the methods discussed in this book.



Chapter 31

Cluster Analysis

Illustrate:

```
hc <- hclust(dist(crs$dataset[1:17,c(2,7,9:10,12)]), "ave")
plot(hc)
This looks like 4 clusters.
km <- kmeans(crs$dataset[1:17,c(2,7,9:10,12)], 4)
km$cluster
```

This gives the same clusters!

31.1 Discriminant Coordinates Plot

Discriminant coordinates displays the primary differences between clusters, and is similar to principal components analysis.

31.2 K Means

The aim of clustering is to identify groups of data points that are close together but as a group are separate from other groups.

The *amap* package includes k-means with a choice of distances like Eulidean and

Spearman.

- . We optimize implementation (with a parallelized hierarchical clustering) and allow the possibility of using different distances like Euclidean or Spearman (rank-based metric).

31.2.1 Summary

Complexity Clustering is usually expensive and K-Means is $O(n^2)$.

31.2.2 Clusters

Clustering is a core tool for the data miner, allowing data to be grouped according to how similar they are, based on some measure of distance.

Basic Clustering

We illustrate very simple clustering through a complete example where the task is to read data from a file (Section 14.3.4, page 219), extract the numeric fields, and then use k-means (Chapter 31.2, page 487) to cluster on just two columns. A plot of the clusters over the two columns shows the points and the cluster centroids. Normally, the clusters would be built over more than just two columns. Also note that each time the code is run a different clustering is likely to be generated!

```
clusters <- 5
load("wine.Rdata")
pdf("graphics/rplot-cluster.pdf")
wine.cl = kmeans(wine[,2:3], clusters)
plot(wine[,2:3], col=wine.cl$cluster)
points(wine.cl$centers, pch=19, cex=1.5, col=1:clusters)
dev.off()
```

R code source: [rplot-cluster.R](#).

The resulting cluster entity has the following entries:

- cluster:** The cluster that each row belongs to.
- centers:** The medoid of each cluster.
- withinss:** The within cluster sum of squares.
- size:** The size of each cluster.

Hot Spots

Cluster analysis can be used to find clusters that are *most interesting* according to some criteria. For example, we might cluster the `spam7` data of the DAAG package (without using `yesno` in the clustering) and then score the clusters depending on the proportion of yes cases within the cluster. The following R code will build K clusters (user specified) and return a score for each cluster.

```
# Some ideas here from Felix Andrews
kmeans.scores <- function(x, centers, cases)
{
  clust <- kmeans(x, centers)
  # Iterate over each cluster to generate the scores
  scores <- c()
  for (i in 1:centers)
  {
    # Count number of TRUE cases in the cluster
    # as the proportion of the cluster size
    scores[i] <- sum( cases[clust$cluster == i] == TRUE ) / clust$size[i]
  }
  # Add the scores as another element to the kmeans list
  clust$scores <- scores
  return(clust)
}
```

We can now run this on our data with:

```
> require(DAAG)
> data(spam7)
> clust <- kmeans.scores(spam7[,1:6], centers=10, spam7["yesno"]=="y")
> clust[c("scores","size")]
$scores
[1] 0.7037037 0.1970109 0.5995763 0.7656250 0.8043478 1.0000000 0.4911628
[8] 0.7446809 0.6086957 0.6043956

$size
[1] 162 2208 472 128 46 5 1075 47 276 182
```

Thus, cluster 5 with 46 members has a high proportion of positive cases and may be a cluster we are interested in exploring further. Clusters 4,

8, and 1 are also probably worth exploring.

Now that we have built some clusters we can generate some rules that describe the clusters:

```
hotspots <- function(x, cluster, cases)
{
  require(rpart)
  overall = sum(cases) / nrow(cases)
  x.clusters <- cbind(x, cluster)
  tree = rpart(cluster ~ ., data = x.clusters, method = "class")
  # tree = prune(tree, cp = 0.06)
  nodes <- rownames(tree$frame)
  paths = path.rpart(tree, nodes = nodes)

  TO BE CONTINUED

  return(tree)
}
```

And to use it:

```
> h <- hotspots(spam7[,1:6], clust$cluster, spam7["yesno"]=="y")
```

Alternative Clustering

For model-based clustering see the BIC algorithm in the **mclust** package. This estimates density with a mixture of Gaussians.

For density-based clustering the following implementation of DBSCAN may be useful. It follows the notation of the original KDD-96 DBSCAN paper. For large datasets, it may be slow.

```
#Christian Hennig
distvector <- function(x,data)
{
  ddata <- t(data)-x
  dv <- apply(ddata^2,2,sum)
}

# data may be nxp or distance matrix
# eps is the dbscan distance cutoff parameter
# MinPts is the minimum size of a cluster
# scale: Should the data be scaled?
# distances: has to be TRUE if data is a distance matrix
# showplot: Should the computation process be visualized?
# countmode: dbscan gives messages when processing point no. (countmode)
dbscan <- function(data,eps,MinPts=5, scale=FALSE, distances=FALSE,
                    showplot=FALSE,
                    countmode=c(1,2,3,5,10,100,1000,5000,10000,50000)){
  data <- as.matrix(data)
```

```

n <- nrow(data)
if (scale) data <- scale(data)
unregpoints <- rep(0,n)
e2 <- eps^2
cv <- rep(0,n)
cn <- 0
i <- 1
for (i in 1:n){
  if (i %in% countmode) cat("Processing point ", i, " of ", n, ".\n")
  unclass <- cv<1
  if (cv[i]==0){
    if (distances) seeds <- data[i,]<=eps
    else{
      seeds <- rep(FALSE,n)
      seeds[unclass] <- distvector(data[i,],data[unclass,])<=e2
    }
    if ((sum(seeds)+unregpoints[i]<MinPts) cv[i] <- (-1)
    else{
      cn <- cn+1
      cv[i] <- cn
      seeds[i] <- unclass[i] <- FALSE
      unregpoints[seeds] <- unregpoints[seeds]+1
      while (sum(seeds)>0){
        if (showplot) plot(data,col=1+cv)
        unclass[seeds] <- FALSE
        cv[seeds] <- cn
        ap <- (1:n)[seeds]
        # print(ap)
        seeds <- rep(FALSE,n)
        for (j in ap){
          if (showplot) plot(data,col=1+cv)
          jseeds <- rep(FALSE,n)
          if (distances) jseeds[unclass] <- data[j,unclass]<=eps
          else{
            jseeds[unclass] <- distvector(data[j,],data[unclass,])<=e2
          }
          unregpoints[jseeds] <- unregpoints[jseeds]+1
          if (cn==1)
          # cat(j, " sum seeds=",sum(seeds), " unreg=",unregpoints[j],
          # " newseeds=",sum(cv[jseeds]==0), "\n")
          if ((sum(jseeds)+unregpoints[j]>=MinPts){
            seeds[jseeds] <- cv[jseeds]==0
            cv[jseeds & cv<0] <- cn
          }
        } # for j
      } # while sum seeds>0
    } # else (sum seeds + ... >= MinPts)
  } # if cv==0
} # for i
if (sum(cv==(-1))>0){
  noisenumber <- cn+1
  cv[cv==(-1)] <- noisenumber
}
else
  noisenumber <- FALSE

```

```

out <- list(classification=cv, noisenumber=noisenumber,
            eps=eps, MinPts=MinPts, unregpoints=unregpoints)
out
} # dbscan
# classification: classification vector
# noisenumber: number in the classification vector indicating noise points
# unregpoints: ignore...

```

31.3 Hierarchical Clustering

Agglomerative clustering is used to build a hierarchical cluster. A complete hierarchical cluster is built on the click of the Execute button. You do not need to re-execute on changing the Number of Clusters. This simply needs to obtain the relevant information from the fully built hclust. But users will automatically go to re-execute after changing this (because this is how everything else in the interface works). An alternative is being considered to make it more obvious not to re-execute.

Once a cluster has been built, have a look at the dendrogram to visually get an idea of the “natural” number of clusters, and then set the number appropriately, then have a look at the stats and the plot.

The *amap* package includes standard hierarchical clustering with a choice of distances like Eulidean and Spearman, and a parallel implementation.

31.4 Summary

- Usage*
- Input*
- Output*
- Complexity*
- Availability*

31.5 Examples

```
> iris.hc <- hclust(dist(iris), "ave")
```

31.6 Resources and Further Reading

Togaware Watermark For Data Mining Survival

Togaware Watermark For Data Mining Survival

Chapter 32

Conditional Trees: Classification

Togaware Watermark For Data Mining Survival

Traditional decision tree induction, as epitomised by CART and ID3/C4.5, do not employ any test of statistical significance in deciding on which variables to choose when partitioning the data. Conditional trees have been introduced to address this by using a conditional distribution, measuring the association between the output and the input variables.



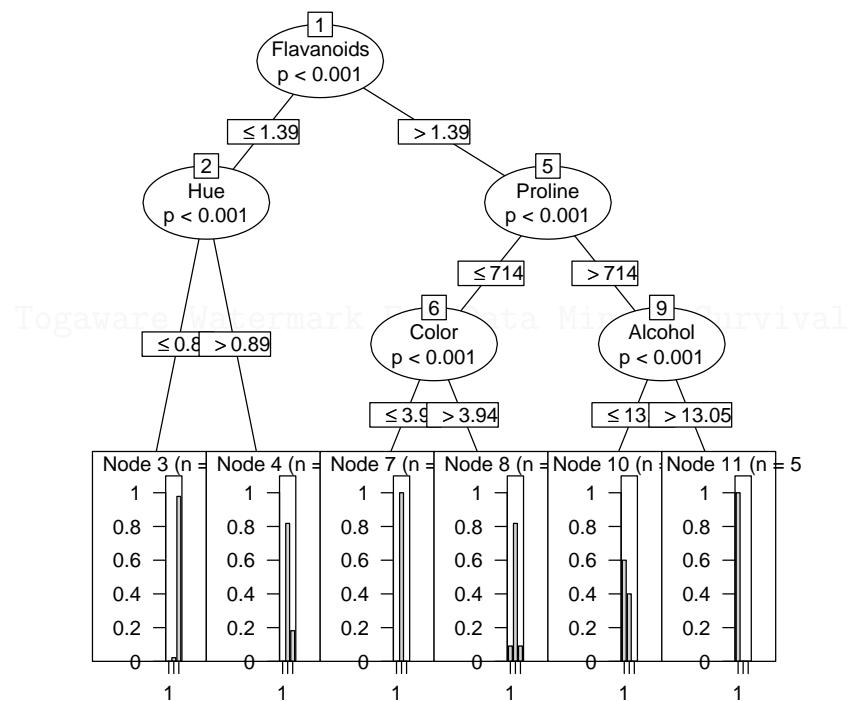
32.1 Summary

*Usage
Input
Output
Complexity
Availability*

32.2 Algorithm

32.3 Examples

You can build a conditional tree using *ctree* from the *party* package:



```
library("party")
load("wine.Rdata")
wine.ctree <- ctree(Type ~ ., data=wine)
pdf("graphics/rplot-ctree.pdf")
plot(wine.ctree)
dev.off()
```

R code source: [rplot-ctree.R](#).

```
> wine.ctree
Conditional tree with 5 terminal nodes

Response: Type
Inputs: Alcohol, Malic, Ash, Alcalinity, Magnesium, Phenols,
```

```
Flavanoids, Nonflavanoids, Proanthocyanins, Color,
Hue, Dilution, Proline
Number of observations: 178

1) Flavanoids <= 1.57; criterion = 1, statistic = 127.131
   2) Hue <= 0.89; criterion = 1, statistic = 36.136
      3)* weights = 47
   2) Hue > 0.89
      4)* weights = 15
1) Flavanoids > 1.57
   5) Proline <= 714; criterion = 1, statistic = 82.158
      6)* weights = 54
   5) Proline > 714
      7) Alcohol <= 13.05; criterion = 1, statistic = 20.638
         8)* weights = 10
      7) Alcohol > 13.05
         9)* weights = 52
```

32.4 Resources and Further Reading

Togaware Watermark For Data Mining Survival

Togaware Watermark For Data Mining Survival

Chapter 33

Hierarchical Clustering: Clustering

Togaware Watermark For Data Mining Survival

The *amap* package includes standard hierarchical clustering with a choice of distances like Eulidean and Spearman, and a parallel implementation.



33.1 Summary

Usage

Input

Output

Complexity

Availability

33.2 Examples

```
> iris.hc <- hclust(dist(iris), "ave")
```

33.3 Resources and Further Reading

Copyright © 2006-2008 Graham Williams

Togaware Watermark For Data Mining Survival

Chapter 34

K-Nearest Neighbours: Classification

Togaware Watermark For Data Mining Survival

The **K-Nearest Neighbour** algorithm.

K-nearest neighbour algorithms handle missing values, are robust to outliers, and can be good predictors. They tend to only handle numeric variables, are sensitive to monotonic transformations, are not robust to irrelevant inputs, and provide models that are not easy to interpret.

K-nearest neighbour classifier, relying on a distance function, is sensitive to noise and irrelevant features, because such features have the same influence on the classification as do good and highly predictive features. A solution to this is to pre-process the data to weight features so that irrelevant and redundant features have a lower weight.



34.1 Summary

Usage
Input
Output
Complexity
Availability

34.2 Resources and Further Reading

Togaware Watermark For Data Mining Survival

Chapter 35

Linear Models

35.0.1 Linear Model

Togaware Watermark For Data Mining Survival

Compare two linear models

```
> wine = read.csv("wine.csv")
> lm1 = lm(Type ~ ., data=wine)
> lm1

Call:
lm(formula = Type ~ ., data = wine)

Coefficients:
(Intercept)      Alcohol       Malic        Ash 
        4.4732853   -0.1170038    0.0301710   -0.1485522
Alcalinity     Magnesium     Phenols     Flavanoids
        0.0398543   -0.0004898    0.1443201   -0.3723914
Nonflavanoids Proanthocyanins Color        Hue  
        -0.3034743    0.0393565    0.0756239   -0.1492451
Dilution       Proline      -0.0007011

> lm2 = lm(Type ~ Alcalinity + Magnesium, data=wine)
> lm2

Call:
lm(formula = Type ~ Alcalinity + Magnesium, data = wine)

Coefficients:
(Intercept)  Alcalinity  Magnesium
        0.563157   0.116950  -0.009072

> anova(lm1, lm2)
Analysis of Variance Table
```

```
Model 1: Type ~ Alcohol + Malic + Ash + Alcalinity + Magnesium + Phenols +
          Flavanoids + Nonflavanoids + Proanthocyanins + Color + Hue +
          Dilution + Proline
Model 2: Type ~ Alcalinity + Magnesium
Res.Df      RSS   Df Sum of Sq    F    Pr(>F)
1       164  10.623
2       175  74.856 -11   -64.234 90.154 < 2.2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Togaware Watermark For Data Mining Survival

Chapter 36

Regression: Ordinal Regression

Togaware Watermark For Data Mining Survival

The output variable is an ordinal - ordered categorical, so that the categorical values have a natural ordering.

Togaware Watermark For Data Mining Survival

Chapter 37

Regression: Logistic Regression

Togaware Watermark For Data Mining Survival

37.1 Summary

Usage

Input

Output

Complexity

Availability

To represent a logistic regression model use effect plots, odds ratio charts, and nomograms are better. See the *Design* package for details.



37.1.1 Linear Model

Compare two linear models

```
> wine = read.csv("wine.csv")
> lm1 = lm(Type ~ ., data=wine)
> lm1

Call:
lm(formula = Type ~ ., data = wine)
```

```

Coefficients:
              (Intercept)          Alcohol
Malic           4.4732853      -0.1170038
               0.0301710      -0.1485522
Alkalinity      Magnesium
Phenols         Flavanoids
               0.0398543      -0.0004898
               0.1443201      -0.3723914
Nonflavanoids   Proanthocyanins
Color           Hue
               -0.3034743      0.0393565
               0.0756239      -0.1492451
Dilution        Proline
               -0.2700542      -0.0007011

> lm2 = lm(Type ~ Alkalinity + Magnesium, data=wine)
> lm2

Call:
lm(formula = Type ~ Alkalinity + Magnesium, data = wine)

Coefficients:
(Intercept)  Alkalinity  Magnesium
0.563157    0.116950   -0.009072

> anova(lm1, lm2)
Analysis of Variance Table

Model 1: Type ~ Alcohol + Malic + Ash + Alkalinity + Magnesium + Phenols +
          Flavanoids + Nonflavanoids + Proanthocyanins + Color + Hue +
          Dilution + Proline
Model 2: Type ~ Alkalinity + Magnesium
Res.Df   RSS   Df Sum of Sq    F
Pr(>F)
1     164  10.623
2     175  74.856 -11    -64.234 90.154 < 2.2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ',' 1

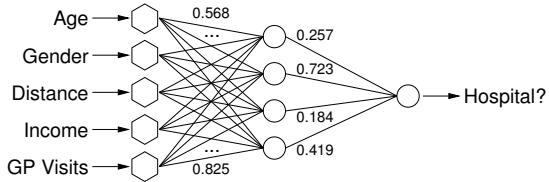
```

37.2 Resources and Further Reading

Chapter 38

Neural Networks: Classification and Regression

Neural networks (often called artificial neural networks to distinguish them from the natural kind found in humans) are a data and processing structure inspired by natural neural networks. The basic idea is to connect a collection of simple neurons into a network. Some of these nodes are identified as input nodes while others are output nodes. The input data is always numeric, perhaps requiring some transformation. The numbers are propagated through the nodes of the network, being modified as they go (multiplied by link weights, and combined with other numbers at nodes), until they pop out at the output nodes. As a classification model the variable values are provided to the input nodes and the “answer” pops out at the output node.



38.1 Overview

Topics include: Neural Networks; Multilayer, Feed-Forward, Neural Network; Neuron Activation functions; Learning through backpropogation.

38.2 Algorithm

Each node in a neural network is an independent processing unit. It accepts multiple numeric inputs that we might consider as signals, and if the combined signal is strong enough it passes the signal on as its single output.

The R package provides a neural network package to fit a neural network with a single hidden layer. See the nnet package in R.

Togaware Watermark For Data Mining Survival

38.2.1 Neural Network

```
> library(nnet)
> ?nnet          # See example there
```

Consider a two-class problem. Build a neural network with

```
>
```

The average Matthew Correlation Coeffience can be used to gauge the performance of the neural network. The highr the value the better.

For an unbalanced class, for example where the ratio of class A to class B is about 3:100, we might decide to weight the under-represented class:

```
> sample.nn <- nnet(..... weights=ifelse(ds$class=="A", 100/3, 1)
```

38.3 Resources and Further Reading

See <http://www.idiap.ch/~bengio/lectures/>

Chapter 39

Support Vector Machines: Classification

Togaware Watermark For Data Mining Survival

39.1 Overview

Support vector machines were introduced by Vapnik (1979, 1998). Their use has become widespread because of their sound theoretical foundations and demonstrated good results in practise. SVMs are based on the idea of structural risk minimisation (SRM).

We can understand the idea best as a binary classification problem, predicting two classes as -1 and 1. The idea is to find the best hyperplane separating the two classes in the training dataset. The best hyperplane is the one that maximises the margin between the two classes—it is the sum of the distances from the hyperplane to the closest positive and negative correctly classified samples. The number of miss-classifications is used to penalise the measure.

The hyperplane can be found in the original dataset (and this is referred to as linear SVMs) or it can be found in a higher-dimensional space by transforming the dataset into a representation having more dimensions (input variables) than the original dataset (referred to as nonlinear SVMs). Mapping the dataset, in this way, into a higher dimensional space, and then reducing the problem to a linear problem, provides a

simple solution.

Computational requirements for SVM are significant.

A kernel is a function $k(x_i, x_j)$ which takes two entities (x_i and x_j) and computes a scalar.

In choosing a kernel: the Gaussian kernel is a good choice when only the smoothness of the data can be assumed.

The main choice then is the γ , the kernel width for SVMs with Gaussian kernel.

The Gaussian kernel is $k(x_i, x_j) = e^{\frac{\|x_i \cdot x_j\|^2}{2\gamma^2}}$. Here, the numerator is the squared 2-norm of the two vectors.

39.2 Examples

R provides the *svm* in *e1071* as an interface to LIBSVM, which provides a very efficient and fast implementation.

```
library(e1071)
iris.svm <- svm(Species ~ ., data=iris, probability=TRUE)
plot(iris.svm, iris, Petal.Width ~ Petal.Length,
     slice = list(Sepal.Width = 3, Sepal.Length = 4))
pred <- predict(iris.svm, iris, probability = TRUE)
attr(pred, "prob") # to get the probabilities
```

kernlab for kernel learning provides *ksvm* and is more integrated into R so that different kernels can easily be explored. A new class called *kernel* is introduced, and kernel functions are objects of this class.

39.3 Resources and Further Reading

An issue with support vector machines is that parameter tuning is not altogether easy for users new to them. One computationally expensive approach is to build multiple models with different parameters and choose the one with lowest expected error. This can lead to suboptimal results though, unless quite extensive search is performed. Research has

explored using previous performance of different parameter settings to predict their relative performance on new tasks Soares et al. (2004).

It has been recognised that the task of learning is often difficult from the specific collection of available variables. Artificial intelligence research, and knowledge representation research in particular, often notes how changing a representation can significantly impact on the ability to reason and learn. Kernel learning projects entities into a higher dimensional space where it is found that learning is actually easier. Kernel learning does this by computing the dot product of the data. Different kernels result in different projections, which result in different distances between entities in the higher dimensional space. A support vector machine is a kernel learner.

Kernel methods are like k nearest neighbours, except all data points contribute through a weighted sum where the kernel measures the distance between points. Kernel methods have demonstrated excellent performance on many machine learning and pattern recognition tasks. However, they are sensitive to the choice of kernel, may be intolerant to noise, and can not deal with missing data and data of mixed types.

A kernel is a function $k(x_i, x_j)$ which takes two entities (x_i and x_j) and computes a scalar.

kernlab for kernel learning provides `ksvm` and is more integrated into R so that different kernels can easily be explored. A new class called `kernel` is introduced, and kernel functions are objects of this class.

A Support Vector Machine (SMV) searches for so called *support vectors* which are data points that are found to lie at the edge of an area in space which is a boundary from one class of points to another. In the terminology of SVM we talk about the space between regions containing data points in different classes as being the margin between those classes. The support vectors are used to identify a hyperplane (when we are talk-

ing about many dimensions in the data, or a line if we were talking about only two dimensional data) that separates the classes. Essentially, the maximum margin between the separable classes is found. An advantage of the method is that the modelling only deals with these support vectors, rather than the whole training dataset, and so the size of the training set is not usually an issue. If the data is not linearly separable, then kernels are used to map the data into higher dimensions so that the classes are linearly separable. Also, Support Vector Machines have been found to perform well on problems that are non-linear, sparse, and high dimensional. A disadvantage is that the algorithm is sensitive to the choice of parameter settings, making it harder to use, and time consuming to identify the best.

It has been recognised that the task of learning is often difficult from the specific collection of available variables. Artificial intelligence research, and knowledge representation research in particular, often notes how changing a representation can significantly impact on the ability to reason and learn. Kernel learning projects entities into a higher dimensional space where it is found that learning is actually easier. Kernel learning does this by computing the dot product of the data. Different kernels result in different projections, which result in different distances between entities in the higher dimensional space. A support vector machine is a kernel learner.

Kernel methods are like k nearest neighbours, except all data points contribute through a weighted sum where the kernel measures the distance between points. Kernel methods have demonstrated excellent performance on many machine learning and pattern recognition tasks. However, they are sensitive to the choice of kernel, may be intolerant to noise, and can not deal with missing data and data of mixed types.

A kernel is a function $k(x_i, x_j)$ which takes two entities (x_i and x_j) and computes a scalar.

kernlab for kernel learning provides `ksvm` and is more integrated into R so that different kernels can easily be explored. A new class called `kernel` is introduced, and kernel functions are objects of this class.

39.3.1 Overview

Support vector machines were introduced by Vapnik (1979, 1998). Their use has become widespread because of their sound theoretical foundations and demonstrated good results in practise. SVMs are based on the idea of structural risk minimisation (SRM).

We can understand the idea best as a binary classification problem, predicting two classes as -1 and 1. The idea is to find the best hyperplane separating the two classes in the training dataset. The best hyperplane is the one that maximises the margin between the two classes—it is the sum of the distances from the hyperplane to the closest positive and negative correctly classified samples. The number of miss-classifications is used to penalise the measure.

The hyperplane can be found in the original dataset (and this is referred to as linear SVMs) or it can be found in a higher-dimensional space by transforming the dataset into a representation having more dimensions (input variables) than the original dataset (referred to as nonlinear SVMs). Mapping the dataset, in this way, into a higher dimensional space, and then reducing the problem to a linear problem, provides a simple solution.

Computational requirements for SVM are significant.

A kernel is a function $k(x_i, x_j)$ which takes two entities (x_i and x_j) and computes a scalar.

In choosing a kernel: the Gaussian kernel is a good choice when only the smoothness of the data can be assumed.

The main choice then is the γ , the kernel width for SVMs with Gaussian kernel.

The Gaussian kernel is $k(x_i, x_j) = e^{\frac{||x_i \cdot x_j||^2}{2\gamma^2}}$. Here, the numerator is the squared 2-norm of the two vectors.

39.3.2 Examples

R provides the *svm* in *e1071* as an interface to LIBSVM, which provides a very efficient and fast implementation.

```
library(e1071)
iris.svm <- svm(Species ~ ., data=iris, probability=TRUE)
plot(iris.svm, iris, Petal.Width ~ Petal.Length,
     slice = list(Sepal.Width = 3, Sepal.Length = 4))
pred <- predict(iris.svm, iris, probability = TRUE)
attr(pred, "prob") # to get the probabilities
```

kernlab for kernel learning provides *ksvm* and is more integrated into R so that different kernels can easily be explored. A new class called *kernel* is introduced, and kernel functions are objects of this class.

39.3.3 Resources and Further Reading

An issue with support vector machines is that parameter tuning is not altogether easy for users new to them. One computationally expensive approach is to build multiple models with different parameters and choose the one with lowest expected error. This can lead to suboptimal results though, unless quite extensive search is performed. Research has explored using previous performance of different parameter settings to predict their relative performance on new tasks Soares et al. (2004).

Part V

Open Products

Togaware Watermark For Data Mining Survival

39.4 Rattle and Other Data Mining Suites

A variety of open source data mining suites are available. Generally, Rattle aims to be quick and easy to deploy for teaching and in business, uncomplicated to use, and to support the delivery of both evidence of the performance of the models, and code for deploying the models in production. However, the choice is always up to you, and in an open source world it is easy to explore the choices yourself. Here we review some of the major data mining suites available and offer very brief comments on their strengths and weaknesses.

Togaware Watermark For Data Mining Survival

520

Togaware Watermark For Data Mining Survival

Copyright © 2006-2008 Graham Williams

Chapter 40

AlphaMiner

Togaware Watermark For Data Mining Survival

Togaware Watermark For Data Mining Survival

Chapter 41

Borgelt Data Mining Suite: From University of Magdeburg

Togaware Watermark For Data Mining Survival

Christian Borgelt of the Working Group on Neural Networks and Fuzzy Systems, Otto-von-Guericke-University of Magdeburg, Germany, makes available a collection of data mining algorithms. These are licensed under the GNU General Public License and available freely for download from the Internet for both Unix/Linux and for MSWindows. Both binary versions and source code are available. This is a tremendous resource for anyone getting started in data mining and indeed provide a suite of tools that are as good as, if not better than, many commercial offerings. Indeed, the apriori algorithm implemented by Borgelt is considered to be one of the fastest available and is also available as part of the Clementine package from SPSS (Chapter 48, page 547).



41.1 Summary

<i>Market Techniques</i>	General data mining application developers. Associations using apriori (<i>apriori</i>). Associations using eclat (<i>eclat</i>). Classification using Bayesian network (<i>ines</i>). Classification using decision trees (<i>dtree</i>). Classification using naive Bayes (<i>bayes</i>). Regression using neural network (<i>nn</i>). Clustering using self-organising maps (<i>xsom</i>).
<i>Platforms</i>	Linux, Unix, MSWindows.
<i>Website</i>	http://fuzzy.cs.uni-magdeburg.de/~borgelt/software.html
<i>Pricing</i>	Freely available—GNU Lesser General Public License.
<i>Vendor</i>	Borgelt, Germany.

41.2 Usage

The Borgelt data mining suite is a collection of command line tools, providing very many options. Below we review the basic command line interfaces for each of the applications.

apriori

Association rules are discovered using the apriori algorithm. The input data file contains transactions, one per line, each transaction containing items, separated by space (or optionally commas, or any other character). A sample input file, `shopping.tab`, is:

```
milk bread
```

To generate the association rules:

```
% apriori shopping.tab shopping.rules
```

Chapter 42

KNime

Togaware Watermark For Data Mining Survival

Togaware Watermark For Data Mining Survival

Chapter 43

R: From the R Foundation

Togaware Watermark For Data Mining Survival

R is both a programming language and an environment for statistical computing and graphics. It is a GNU project providing compatibility with the S language and environment developed by John Chambers and others at Bell Laboratories (formerly AT&T, now Lucent Technologies), from 1976 onwards. It provides a wide variety of statistical (linear and nonlinear modelling, classical statistical tests, time-series analysis, classification, clustering, ...) and graphical techniques. While it is often the vehicle of choice for research in statistical methodology, it provides a powerful environment for data mining, with a comprehensive collection of analytic tools.



43.1 Summary

<i>Market</i>	Statistical research.
<i>Techniques</i>	Many are available as packages, including boosting (<i>gbm</i>), and random forests (<i>randomForests</i>).
<i>Platforms</i>	Linux, Unix, MacOS, MSWindows.
<i>Vendor</i>	The R Foundation for Statistical Computing.
<i>Website</i>	http://www.r-project.org
<i>Pricing</i>	Freely available—GNU General Public License.

43.2 Further Information

A reference card is available from <http://www.rpad.org/Rpad/Rpad-refcard.pdf>

Useful books include:



Bootstrap Methods and Their Applications by *Anthony C. Davison and David V. Hinkley*. 256 pages published by Cambridge University Press, 1994, ISBN 0-521-57391-2. The R package `boot` is based on the methods discussed in this book.

ToDo: Add Hastie, Tibshirani, Friedman. Add Venables, Ripley. Add Daalgaard.

Chapter 44

RapidMiner: From Rapid-I

Togaware Watermark For Data Mining Survival

Heavily based on WEKA, formerly called Yale, provides a good graphical interface and addition data processing tools.

Togaware Watermark For Data Mining Survival

Chapter 45

Rattle: From Togaware

Togaware Watermark For Data Mining Survival

Rattle is an R based data mining tool using the Gnome graphical user interface, available on Debian, GNU/Linux, Unix, MS/Windows, and Macintosh/OSX. It supports a growing collection of algorithms that can be used in general data mining projects. The software is released under the GNU General Public License and freely available. In general the tool provides a consistent interface for interacting with the many of the different packages available in R.



45.1 Summary

<i>Market</i>	General data mining applications, ease of use, ideal for new data miners.
<i>Techniques</i>	Associations using apriori (arules), Classification using decision trees (rpart), generalised linear models (glm), boosting (gbm), and Random Forests (randomForest).
<i>Platforms</i>	GNU/Linux, Unix, MS/Windows, Macintosh/OSX
<i>Website</i>	http://www.togaware.com/datamining/
<i>Pricing</i>	Freely available—GNU General Public License.
<i>Vendor</i>	Togaware, Canberra, Australia.

45.2 Usage

Rattle provides a graphical interface to a variety of data mining applications.

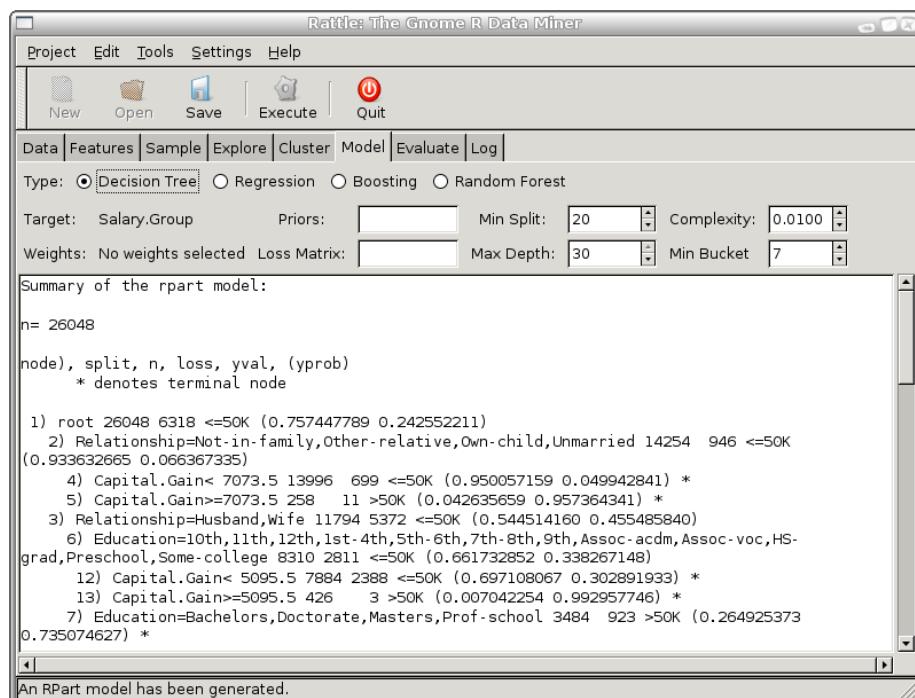


Figure 45.1: Togaware's Rattle Gnome Data Mining interface.

Togaware Watermark For Data Mining Survival

Chapter 46

Weka: From University of Waikato

Togaware Watermark For Data Mining Survival

WEKA (the Waikato Environment for Knowledge Analysis) is an open source and freely available workbench for applying machine learning techniques to practical problems, integrating many different machine learning tools within a common framework and a uniform, if basic but functional, user interface. WEKA incorporates over 60 machine learning techniques, ranging from traditional decision trees, association rules, clustering, through to modern random forests and support vector machines. A WEKA user is able to use machine learning techniques to derive useful knowledge from quite large databases. Typical users include both researchers and industrial scientists.



WEKA is a great suite of data mining algorithms that allow us to quickly explore alternatives approaches to data mining. However, perhaps because it is written in Java, it is well known that the WEKA user interface is very heavy in its use of memory. Staying with the command line in WEKA (see the excellent guide written by Alexander K. Seewald) is a good option with reports of, for example, using the rather efficient NaiveBayesNominal algorithm to process a large ham/spam dataset with

500,000 samples and 1.3 million attributes on the commandline “in a few minutes.” (See [KDD Nuggets, 2007, n24](#)).

46.1 Summary

<i>Market</i>	General data mining applications.
<i>Techniques</i>	<i>Classification</i> using decision trees, support vector machines, conjunctive rules. <i>Regression</i> . <i>Rule induction</i> using FOIL.
<i>Platforms</i>	Java (GNU/Linux, Unix, MSWindows).
<i>Website</i>	http://www.cs.waikato.ac.nz/ml/
<i>Pricing</i>	Open source—GNU General Public License.
<i>Vendor</i>	University of Waikato, Hamilton, New Zealand.

46.2 Usage

Togaware Watermark For Data Mining Survival

Weka runs as a Java application. Thus, a user can simply obtain the appropriate Java archive file `weka.jar` and then start up the application with Java. On GNU/Linux and Unix this is usually:

```
java -jar weka.jar
```

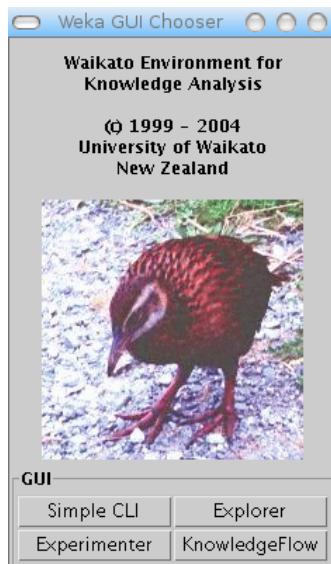
On MSWindows this is usually:

```
javaw -jar weka.jar
```

On start-up you will see the *Weka GUI Chooser* (Figure 46.1). From here you can either run the system from a simple command line interface (**Simple CLI**), or else start up an interactive data explorer and modeller with the **Explorer** button.

The Weka Explorer (Figure 46.2) can be used to interactively load data, pre-process the data, and run the modelling tools. Figure 46.2 shows a dataset having been loaded, with a list of the variables found in the CSV file in the left pane, with a plot of the distribution of the output variable (`yexno`) shown in the right pane.

To load a CSV file, for example, click on the *Open file...* button. This will bring up the Weka Open dialogue (Figure 46.3). Click in the button



Togaware, Watermark and File-Delimited Mining Survival
Figure 46.1: The Weka GUI chooser.

labelled *Arff data files* to change this to *CSV data files*. Then browse to the CSV file you wish to load. In our example this is `wine-nominal.csv`. Double click the name, and then the *Open* button to import the data.

To start building models, go to the Classify tab (Figure 46.4). The default model builder is ZeroR, a very basic model builder indeed! Click the Choose button to select from over 60 model builders. For example, under Trees you could choose J48, which is an implementation of C4.5. You will also find support vector machines (SMO under Functions) and random forests (under Trees) and AdaBoost (under Meta). Once you have chosen your model builder the corresponding command is shown in the text box to the right of the button. Click in here to change any of the parameters, or to read some documentation about the chosen method. From the drop-down menu above the Start button, choose the output variable (in this case we have chosen Class). When you are ready to build your model, click on the Start button.

A tree built this way will list, for each branch, the number of training instances and the number of these that are misclassified.

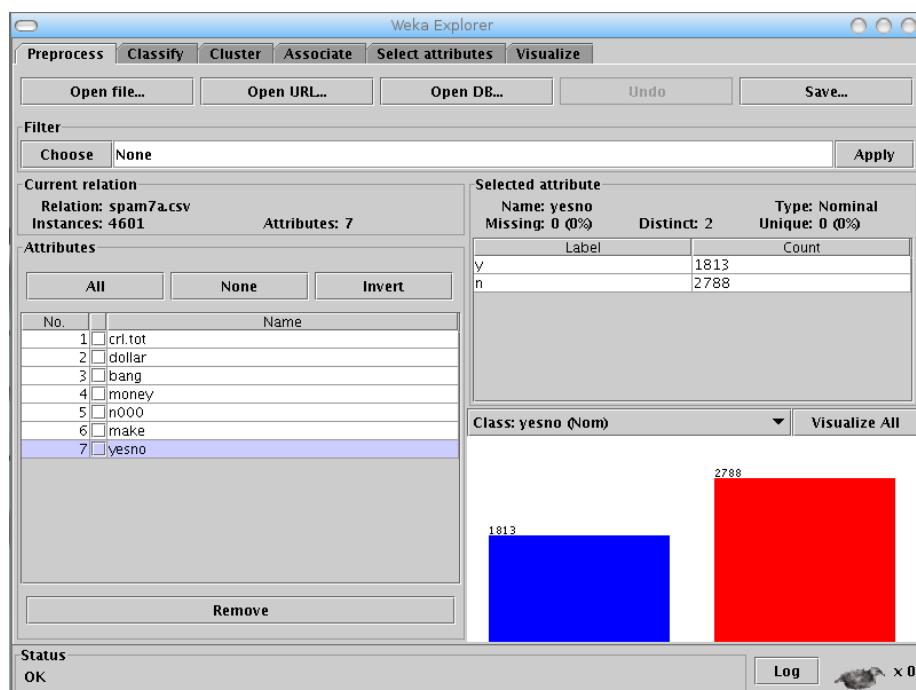


Figure 46.2: Weka explorer viewing data.

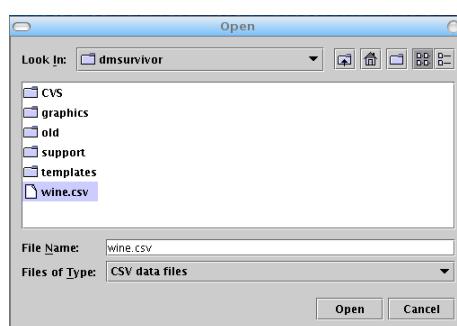


Figure 46.3: Import CSV data into Weka.

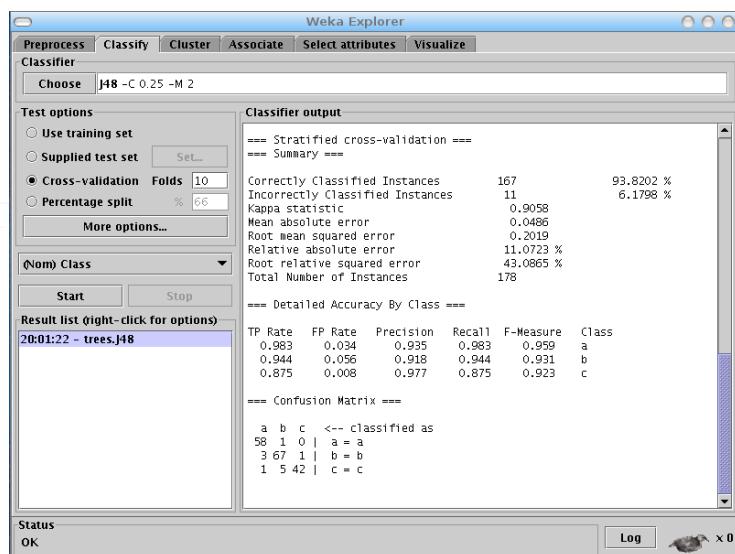


Figure 46.4: Output from running J48 (C4.5).

Togaware Watermark For Data Mining Survival

Part VI

Closed Products

Togaware Watermark For Data Mining Survival

Chapter 47

C4.5: Classification

Togaware Watermark For Data Mining Survival

C4.5 builds decision trees.

47.1 Summary



Complexity

Generally C4.5 is quite efficient as the number of training instances increases, and for specific datasets has been found empirically to be between $O(n^{1.22})$ and $O(n^{1.38})$. With rule generation the algorithm is somewhat more expensive at $O(n^4)$.

Availability

The Borgelt collection (Chapter 41, page 523) contains *dtree*, a generic implementation of the decision tree divide and conqueror algorithm. Weka (Chapter 46, page 535) also provides a freely available implementation of a decision tree induction algorithm (J48) within its Java-based framework. Decision tree induction is a fundamental data mining tool and implementations of C4.5 or its variations are available in most commercial data mining toolkits, including Clementine (Chapter 48, page 547) and STATISTICA (Chapter 54, page 563).

47.2 Overview

Decision tree analysis is a technique for building classification models for predicting classes for unseen entities.

Decision trees have been around for a long time as a mechanism for structuring a series of questions and choosing the next question to ask on the basis of the answer to the previous question. In data mining we commonly identify decision trees as the knowledge representation scheme targeted by the family of techniques originating from ID3 in 1979.

As a classification technique decision tree analysis generates decision trees for discrete value classification. The variables can be either continuous or categorical.

A *decision tree* has the traditional computer science structure of a tree, having a single root node and multiple arcs emanating from the node to connect to other nodes.

Example

You are sitting with the finance manager of the car sales yard applying, on-line, for a loan to purchase a new people mover. The finance manager is asking you a series of questions. You think it odd that just a little while ago you overheard the same finance manager using the same on-line system, asking someone else different questions, but not to worry, your circumstances are different. You supply the details requested and walk out with a new set of wheels.

Over the coming year as you pay back the loan with monthly repayments the financial institution who actually lent you the money is monitoring your account. Are you paying back on time? Or are you sometimes late? Or do you never pay unless you are visited by the debt collector? In the end, were you a good or a bad customer for the financial institution?

The financial institution can use their experience with you and many other customers to develop a prediction system to indicate for each new customer their likelihood of being a good or bad customer. It is this system that decides which are the more important variables to take into account in making that decision. For some combination of variables more details and more questions, and different questions need to be answered.

Algorithm

Decision tree induction algorithms are generally what are called *greedy* algorithms. They are greedy in that they decide on a question to ask then don't consider any more alternatives later on. The algorithms are also *divide and conquer* because they partition the database into smaller sets. In fact, the general algorithm continually partitions the database into smaller sets until the sets all have the same value for the output variable.

Note that pruning is a mechanism for reducing the variance of the resulting models. However, for large datasets the reduction of variance is not usually useful thus unpruned trees may actually be better.

47.3 Resources and Further Reading

C4.5 was made available together with a book (Quinlan, 1993) that served as a guide to using the code, which was printed as half of the book, and supplied on electronically.

The similar technique of classification trees was independently developed at the same time.

The complexity figure comes from Provost et al. (1999).

Togaware Watermark For Data Mining Survival

Chapter 48

Clementine: From SPSS

Togaware Watermark For Data Mining Survival

Clementine packages a number of tools with a GUI which simplifies the process of performing a data mining project. In particular the Clementine workbench supports a number of data mining algorithms through a simple linked node interface supporting the entire business process of data mining using the CRISP-DM model.



48.1 Summary

<i>Market Techniques</i>	General data mining applications. <i>Association analysis</i> using apriori. <i>Classification</i> using C5.0. <i>Clustering</i> using k-means (Chapter 31.2, page 487) and Kohonen self organising maps. <i>Multivariate</i> using <i>principle component analysis</i> and factor analysis. <i>Regression</i> using neural networks. Visualisation (Web Graph).
<i>Platforms</i>	Linux, Unix, MSWindows.
<i>Website</i>	http://www.spss.com/spssbi/clementine/
<i>Vendor</i>	SPSS, Chicago, Illinois.

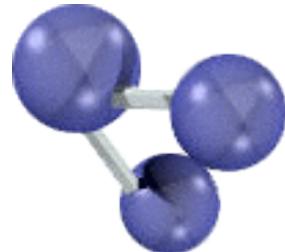
Togaware Watermark For Data Mining Survival

Chapter 49

Equbits Foresight: Tool from Equbits

Togaware Watermark For Data Mining Survival

Equbits provides SVM based predictive modeling solutions to Drug Discovery Professionals. Equbits Foresight, a software application, provides classification and regression modeling that is automated, accurate, and interpretable.



49.1 Summary

<i>Market</i>	Drug Discovery, OEM.
<i>Techniques</i>	Support Vector Machines (SVM's)
<i>Website</i>	http://www.Equbits.com
<i>Vendor</i>	Equbits, CA.

Togaware Watermark For Data Mining Survival

Chapter 50

GhostMiner: From Fujitsu

Togaware Watermark For Data Mining Survival

GhostMiner is an analytical data mining package supported by visualisations for multidimensional data. While supporting the traditional analysis techniques, GhostMiner also provides variable selection facilities that include automated approaches for variable selection and variable ranking.



50.1 Summary

<i>Techniques</i>	Classification using Support Vector Machine. Clustering using Dendograms and Support Vector techniques. Visualisation using Principal Component Analysis.
<i>Platforms</i>	MSWindows NT, MSWindows 2000, MSWindows XP.
<i>Website</i>	http://www.fqsp1.com.pl/ghostminer/
<i>Pricing</i>	From vendor.
<i>Vendor</i>	Fujitsu, Poland.

50.2 Usage

GhostMiner supports the extraction of knowledge from data through the following steps:

- Data selection
- Data preprocessing
- Variable selection
- Model learning
- Model analysis
- Selection of the final model

The final output of each GhostMiner project is a model of the knowledge inherent in your data. This model can then be used for decision support.

GhostMiner includes an intuitive interface for managing projects of arbitrary complexity through a project window and a project tree. Projects store all results of experiments performed so far, automatically adding models created during cross-validation.

Statistical information and charting options are provided to view the data, allowing for the quick detection of outliers. Data may be viewed in its original form or in a standardized/normalized form.

The data can be filtered and ordered in various ways, including.....

Numeric and graphical views display variable statistics such as the average, standard deviation, minimum/maximum values and the number of missing values. A facility is provided to show for each class and each variable the distribution of variable values in 2 and 3-dimensional charts, allowing for the identification of potentially useful variables. Two-dimensional projections allow the viewing of data points for particular combination of variables.

A collection of adaptive analytics, including neural networks, neurofuzzy systems, decision trees, and the k-nearest neighbour algorithm are pro-

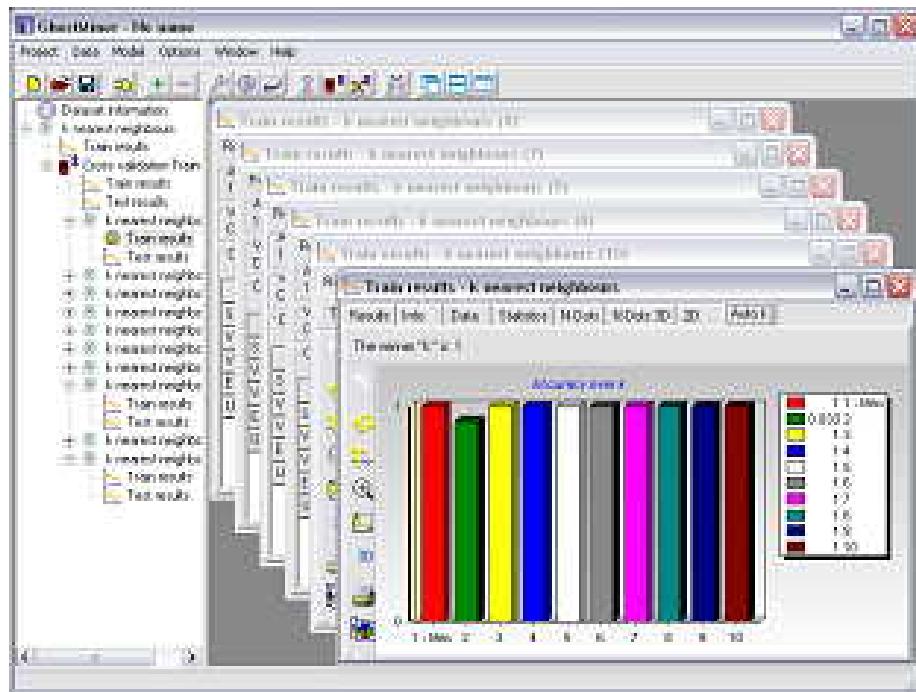


Figure 50.1: Fujitsu GhostMiner interface.

vided. The models can be refined through an ensemble where committees can be composed of arbitrary models or subcommittees of models. GhostMiner introduces K-classifiers which are composed of single models or committees, one for each class.

Various methods for testing the accuracy and efficiency of trained models (cross-validation and -xtest) along with the confusion matrix summarising the model quality are provided.

All these variables are accessible in GhostMiner Developer. After the best model has been selected GhostMiner Analyzer may be used to read it and evaluate new cases. Analyzer displays information about the model and about the data the model has been trained for, without needing to access the tools available in GhostMiner Developer.

Togaware Watermark For Data Mining Survival

Chapter 51

InductionEngine: Tool from PredictionWorks

Togaware Watermark For Data Mining Survival

PredictionWorks' InductionEngine creates real-time predictive models and analytical data marts to support ad hoc analysis. Past systems include customer lifetime value scoring, mortgage prepayment scoring, delinquency scoring, churn scoring, warehouse perishables demand prediction, coupon optimisation, and real-time web personalisation.



51.1 Summary

<i>Techniques</i>	Decision Tree, k-Nearest Neighbor, Logistic Regression, Naive-Bayes,+Visualisation, Auto Sampling, Cost Matrix
<i>Platforms</i>	GNU/Linux, MSWindows.
<i>Website</i>	http://www.predictionworks.com
<i>Pricing</i>	Available from vendor.
<i>Vendor</i>	PredictionWorks, USA.

Togaware Watermark For Data Mining Survival

Chapter 52

ODM: Oracle Data Mining

Togaware Watermark For Data Mining Survival

Having purchased one of the original data mining products (Darwin from Thinking Machines) Oracle Data Miner (ODMiner) has been built from the ground-up, and the data mining engine is integrated into the Oracle server. The Oracle Data Miner is a client-side tool that layers on this server engine. It provides a graphical user interface for data mining using wizards to guide a data miner through the data preparation, data mining, model evaluation, and model scoring process. As the data analyst transforms the data, builds models, and interprets results, Oracle Data Miner can automatically generate code needed to transform the data mining steps into an integrated data mining/BI application. All Oracle Data Mining functions are accessible by PL/SQL and/or Java APIs so you can develop enterprise BI applications on top of your Oracle Database. ODMiner generates Java code from Oracle Data Miner models and results.



52.1 Summary

<i>Market Techniques</i>	Oracle database users.
	Classification: Decision Trees, Support Vector Machines, Naive Bayes, Adaptive Bayes Network. Regression: Support Vector Machines. Clustering: Hierarchical K-Means, Orthogonal partitioned cluster (O-Cluster) Feature Extraction: Non-negative matrix factorization
<i>Platforms</i>	All Oracle10g platforms, including Linux, Unix, MacOS, MSWindows.
<i>Website Vendor</i>	http://www.oracle.com/technology/products/bi/odm/ Oracle, California.

52.2 Usage

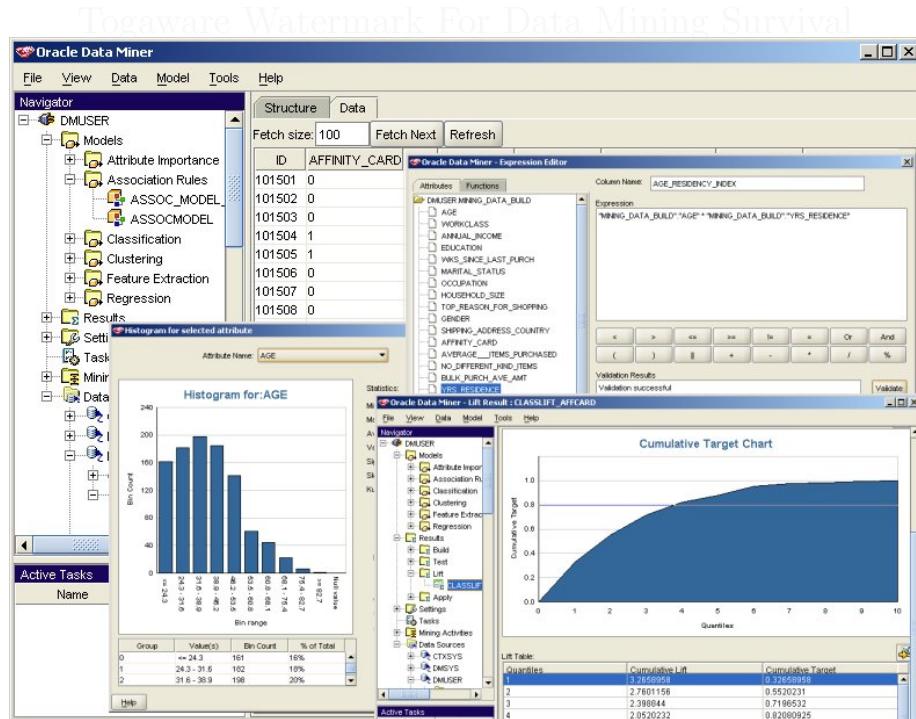


Figure 52.1: Sample ODMiner interface to ODM.

Chapter 53

Enterprise Miner: From SAS

Togaware Watermark For Data Mining Survival

Enterprise Miner is a data mining package with integrated linkage to the popular commercial SAS statistical application. It provides a range of techniques accessed through a graphical user interface, using the node (representing data processing and modelling steps) and link paradigm to build process flows. The product is priced at the higher end of the market with a license restricting its usage to the owner's own data, only—the product is certainly targeted to large enterprises. For smaller projects, SAS/JMP is a good choice, with student pricing starting from about \$100.



53.1 Summary

<i>Market Techniques</i>	Internal data mining applications with statistical backing. <i>Associations.</i> <i>Classification</i> using decision trees and logistic regression. <i>Clustering</i> using k-means. <i>Regression</i> using neural networks and generalised linear models. Ensembles and model assessment. Statistical tools.
<i>Platforms</i>	GNU/Linux, Unix, MSWindows.
<i>Website</i>	http://www.sas.com .
<i>Pricing</i>	High end, annual fees, tens or hundreds of thousand dollars.
<i>Vendor</i>	SAS Institute, Cary, North Carolina.

53.2 Usage

SAS Enterprise Miner is a full scale commercial product with pricing to match. Version 4 provided many more modelling options than the rewritten Version 5 (which now uses a Java thin client to access the Enterprise Miner server). The missing functionality is working its way back into the product.

Enterprise Miner works with the idea of projects. The first task is to make a New Project. Supply a Name (this will be the name of a directory) and a Path (which is where the directory will be created). If you have SAS datasets somewhere, you might want to click across to the Startup Code tab and add something like:

```
libname dmsource "/home/share/data/sas";
```

Then select OK to initialise the project.

Now select the new project, and right click the Data Source node. Create a New Data Source, and Browse to the libname you supplied, and choose a SAS dataset from those available. Click on through to either perform a Basic load (automatic processing of variables) or an Advanced load (to allow tuning of how the variables are treated).

Once the Data Source has been defined, create a New Drawing, and then drag the Data Source on to the new drawing. From the Data tab drag the Partition icon onto the diagram and connect the Data Source to the

53.3 Tips and Tricks

561

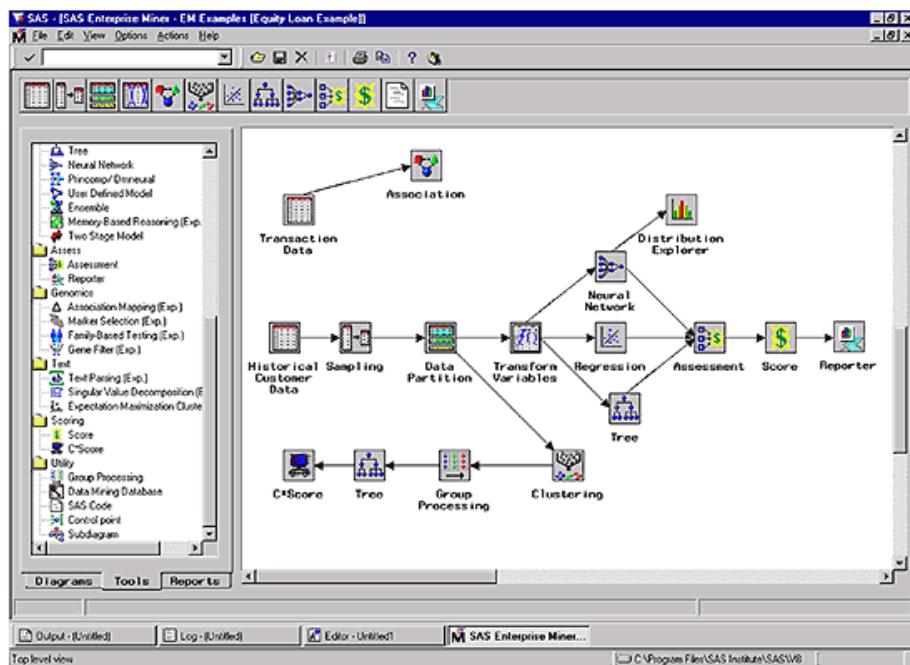


Figure 53.1: SAS Enterprise Miner interface (Version 4).

Partition node. Then from the Classify tab drag a decision tree node onto the diagram and connect the Partition node to the Decision Tree node. No right click the Decision Tree node to select Run, to build the decision tree. Right click again, once it has completed, to view the Results.

53.3 Tips and Tricks

SAS Enterprise Miner is a large product (including the SAS statistical package), and sometimes cumbersome and quaintly old fashioned. It is being migrated into a modern world, slowly, and its functionality is improving regularly. Below we highlight some issues and tricks that may be helpful.

- **Log Transform** of a dataset seems somewhat of a trap. A log transform essentially transforms the data items d with $\frac{\ln(1+(d-d_{min})}{d_{max}-d_{min}}$.

The d_{min} and d_{max} are determined from the current dataset. However, if you log transform your training dataset, and separately log transform your test dataset, then the transforms will end up being different! Hence, the model you build will not be applicable to any data unless it is transformed with the same values of d_{min} and d_{max} ! You may want to avoid such transforms.

Togaware Watermark For Data Mining Survival

Chapter 54

Statistica Data Miner: From StatSoft

Togaware Watermark For Data Mining Survival

STATISTICA Data Miner combines a comprehensive selection of data mining tools with an icon-based interface. It targets integrated data mining solutions in a variety of business domains such as finance, marketing, insurance, and pharmaceuticals. The included data mining tools integrate with the *STATISTICA* line of products which includes a web delivery application. The provided algorithms cover the whole spectrum of data mining technologies including alternative neural networks architectures, classification/regression trees, multivariate modelling, and many other classification and regression techniques. User implemented nodes can easily be added to the interface and many models generate industry standard PMML (XML).

STATISTICA
Data Miner

 **StatSoft®**

54.1 Summary

<i>Market Techniques</i>	General data mining applications. Associations using apriori. Classification using decision trees. Clustering using k-means (Chapter 31.2, page 487) and EM. Regression using regression trees, neural networks, MARS, generalised linear models and CHAID. Miscellaneous techniques including boosting and survival analysis.
<i>Platforms</i>	MSWindows with potential for clients running in web browsers.
<i>Website</i>	http://www.statsoft.com
<i>Pricing</i>	Available from vendor.
<i>Vendor</i>	StatSoft, Tulsa, Oklahoma.

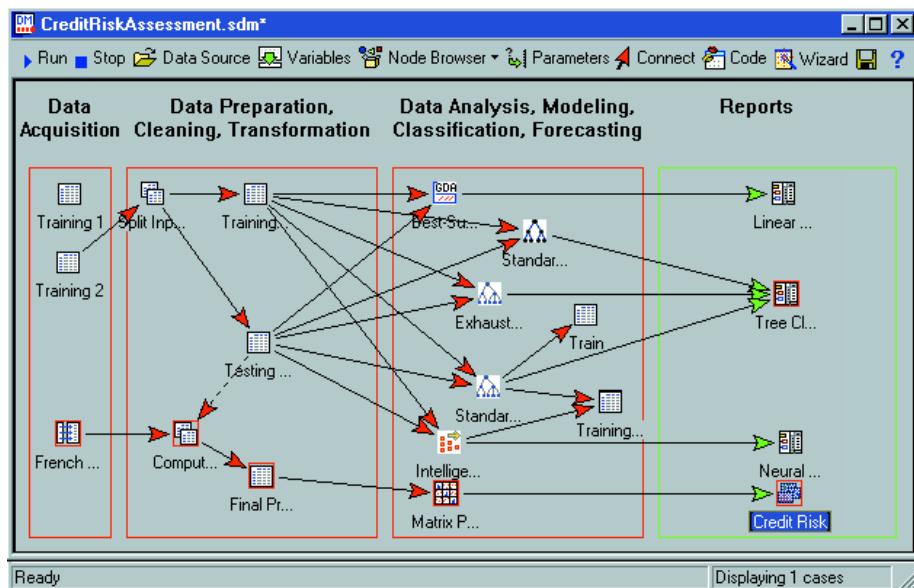
54.2 Usage

STATISTICA Data Miner implements a large number of algorithms for analysing data. They are seamlessly integrated with other analytic and graphics options within an icon based user interface.

The graphical user interface allows analysis nodes to be dragged into the data mining workspace and connected to other nodes. Results can be reviewed, analysed, and saved, or used to perform additional interactive analyses. The analysis nodes can handle multiple data streams, enabling processing of lists of data sources. *STATISTICA* includes options for defining connections to databases on remote servers.

A particularly significant feature of *STATISTICA* is its open architecture which allows users to add their own custom nodes implemented in any language accessible from Visual Basic programs. User supplied nodes can have their own user interface for accepting user parameters, choices, etc. The resulting node then becomes part of the toolkit available to your users.

Multi-threading and distributed processing allows multiple servers to work in parallel for computationally intensive projects. *STATISTICA* Data Miner is also a COM object allowing full integration with other applications or analysis macros under MSWindows.

Figure 54.1: *Statistica* Data Miner graphical interface.

A comprehensive collection of algorithms is implemented in *STATISTICA* Data Miner. These include statistical, exploratory, and visualisation techniques. In addition to the techniques listed in the summary above, graphical and interactive exploration/visualisation tools work with descriptive statistics and exploratory data analysis to give initial insights into the data being explored. General data mining operations, including slice and dice and drill-down, for interactively exploring data on selected variables and categories are also supported.

A goodness of fit module computes various goodness of fit statistics for continuous and categorical output variables (for regression and classification problems). The module provides a competitive evaluation of models, as a tool to choose the best solution.

The Predictive Modelling Markup Language (PMML) is supported as the description language for generated models from many of the included predictive data mining tools. PMML, based on XML, allows both sharing of models and loading of other, possibly externally generated, models into *STATISTICA*.

54.3 Sample Applications

The tool is suitable for general application, including finance and quality management.

The typical finance application is loan application processing. Consider a bank receiving an average of over 900 loan applications per day, requiring considerable manpower to process in a timely and accurate manner. A decision support system for loan application screening was developed using *STATISTICA* Data Miner, where applicant data are processed using a variety of analytic methods, ranging from simple IF-THEN-ELSE decision rules to advanced classification and regression tree models designed to resolve ambiguities. Such preprocessing of each application takes only seconds, and, in some cases, includes verification through automatic queries to a large data warehouse. A risk rating is generated indicating: very low risk, low risk, acceptable risk, high risk, and very high risk. Since the system was deployed the rate of error in the loan approval process has reportedly decreased “significantly”.

For quality management a semiconductor manufacturer employed a totally automated system for “Intelligent Quality Monitoring” (IQM). IQM collects over 50 thousand characteristics of the production process in real-time (ranging from simple temperature variations to scores from multi-perspective laser scanning detection systems). All these data are pre-screened in real-time and fed into an intelligent data processor built on predictive data mining technologies. The risk of producing a defective wafer is calculated and monitored in real-time and the decisions to make costly adjustments or to terminate a wafer burning cycle entirely are made automatically. Moreover, this decision process is integrated with the corporate Enterprise Resource Planning system, and thus linked to data on fluctuating cost considerations. Consistently higher yields per cycle, compared to its competitors, are now achieved, and the cost per unit has been reduced by 27%.

54.4 Further Information

StatSoft was founded in 1984 and produces enterprise and single-user software for data analysis, data mining, quality control/six sigma, and web-based analytics. Training and consulting services are provided through StatSoft's subsidiaries and distributors, worldwide.

Togaware Watermark For Data Mining Survival

Togaware Watermark For Data Mining Survival

Chapter 55

TreeNet: From Salford Systems

Togaware Watermark For Data Mining Survival

Salford Systems has implemented and commercialised several tools from two of the key researchers in data mining (Jerome Friedman and Leo Breiman of Stanford University). TreeNet® uses stochastic gradient boosting to improve modelling capabilities.



55.1 Summary

<i>Market</i>	General data mining projects.
<i>Techniques</i>	Classification.
<i>Platforms</i>	MSWindows.
<i>Website</i>	http://www.salford-systems.com
<i>Vendor</i>	Salford Systems, San Diego, California.

Togaware Watermark For Data Mining Survival

Chapter 56

Virtual Predict: From Virtual Genetics

Togaware Watermark For Data Mining Survival

Virtual Predict is a system for the discovery of classification and regression rules. It provides more than the standard decision tree and rule induction in that it allows for more expressive hypotheses to be generated and more expressive background knowledge to be incorporated in the induction process.



56.1 Summary

<i>Market</i>	Life sciences research consultancies.
<i>Techniques</i>	Rule induction, inductive logic programming.
<i>Platforms</i>	MSWindows.
<i>Website</i>	http://www.vglab.com
<i>Pricing</i>	Available from vendor.
<i>Vendor</i>	Virtual Genetics Laboratory, Stockholm, Sweden.

56.2 Usage

Virtual Predict was deployed by the department of Medicinal Chemistry at AstraZeneca together with consultants from Virtual Genetics. Known molecular properties from about 1,000 substances were used to create a model to predict water solubility in new untested drug leads. Molecular properties included the number of atoms, bonds and rings, graph radius and diameter, Wiener index, carbon, nitrogen and oxygen counts, molecular weight and volume, and polarizability. By including known solubility values Virtual Predict identified important criteria for predicting solubility of unknown substances with a precision between 87%–93%.

Virtual Predict handles both numerical and categorical data. Structured data in the form of is also handled.

The user can choose from a range of data mining techniques including decision trees (divide-and-conquer, or DAC, in Virtual Predict terminology) for classification and regression. Minimum Description Length (MDL) is employed for Options for choosing variables include Information Gain. Pruning, boosting, bagging, boosted stumps, and naive Bayes are all options. And SAC?

Building a model is straight-forward with the simple and uncluttered MSWindows graphical user interface. Data in comma separated format (CSV) can easily be imported. However, visualisations of the resulting models are not provided.

Building multiple models with different approaches is straight forward and test protocols are provided to empirically compare the performance of the various methods for a particular dataset.

Virtual Predict employs a representation language based on the common logic programming language Prolog.

Part VII

Appendices

Togaware Watermark For Data Mining Survival

Appendix A

Glossary

With the rather unruly explosion of interest in data mining and its well documented commercial successes, and the fact that data mining is the fusion of many disciplines, each with their own heritage, the terminology used in the data mining community is at times quite confusing and often redundant. The beginnings of a Glossary began here but has ceased. [Wikipedia](#) is now the canonical source.



Bias: The error in a model due to systematic inadequacies in the learning process. That is, those instances consistently incorrectly classified by models built by the learning algorithm. Modelling error due to bias can be reduced using [Boosting](#). Compare with [variance](#).

C

C5.0: A commercial decision tree and rule induction product from [Rule-Quest](#) developed by Ross Quinlan as the successor to his very successful and widely used ID3 and C4.5 systems.

Churn: The process of customer turnover, as occurs in many consumer service providers such as telecommunications and credit providers.

Classification:

Coevolution: An **Evolutionary Computation** paradigm where both the measure of fitness and the solution evolve separately and affect each other.

Computational Intelligence: Describes numerically based Artificial Intelligence systems distinguished from Symbolic Intelligence. Generally covers **Evolutionary Computation**, Neural Networks and Fuzzy Logic.

Cross Validation: A method of comparing or confirming performance of predictive models by estimating the error that would be produced by a model. Cross validation is also used to compare one method of inducing a predictive model (called an inducer) with another inducer. Cross validation can also be used to refine the parameters of a particular inducer. In this case the inducer's parameters are refined to minimise the error in the model produced. A k -fold cross validation will build k models from k datasets.

Confidence:

Cubist: A commercial piecewise-linear regression product from **Rule-Quest** developed by Ross Quinlan. Uses decision tree induction to identify subsets of the data on which to perform regression.

D

Data Cube: A block of data, often extracted from a **data warehouse**, offering fast access/views of data in any number of dimensions.

Data Mining: A technology concerned with using a variety of techniques, including **associations**, **classification**, and **segmentation**, on problem domains that require analysing extremely large collections of data, including domains such as fraud and **churn**. Typically draws on tools and techniques from machine learning, Parallel Computation, **OLAP**, visualisation, Mathematical Computation, and **statistics**. Fayyad defines data mining as *a single step in the **KDD** process that under ac-*

ceptable computational efficiency limitations, enumerates structures (patterns or models) over the data.

Data Warehouse: Provides a multi-dimensional view of an organisation's data for efficient analysis as distinct from typical transactional relational database systems which manage the day-to-day operations of the organisation. Data warehouses particularly lend themselves to quick ad-hoc queries on large volumes of data on a read-only basis. Inmon defined a data warehouse as *a subject-oriented, integrated, time variant and non-volatile collection of data in support of management's decision-making process*.

Decision Tree:

Demographic Clustering: Clustering performed on the characteristics of population groups in terms of size, distribution, and other vital statistics, rather than on the individuals of the population. Related to **Data Mining** in **Data Cubes** in that Data Cubes allow varying degrees of aggregation over any of the variables.

Dependent Variable: The classical term for an Output Variable.

E

E-Commerce: Conduct of financial transactions by electronic means, usually in the form of on-line business and trade, and often involving the buying and selling of goods over the Internet.

Ensemble Learning: A generic description for a learning system that builds multiple knowledge structures and combines the outcomes in some way. For example, a voting strategy might be used where we have generated multiple **Decision Trees** for the same task. Each decision tree will be applied to an unseen record and the decisions from each tree will be combined in some way. One of the earliest examples was developed by Williams (1987) (also see Williams (1988) and Williams (1991)) where multiple decision trees were combined using a majority rules voting scheme. The idea of a committee of models arose from earlier work in expert systems and then in machine learning with MIL (Multiple Inductive Learning).

Evolutionary Computation: Motivated by Darwinian Evolution heuristic rules are employed to modify a population of solutions in such a way that each generation of the population tends to be on average better than its predecessor. These algorithms generally have three components (analogous to the theory of evolution): A fitness function which is used to guide in the selection of the fittest individual; Reproduction operators to generate new structures from other structures; Genetic operators to guide the combination of the structures in reproduction.

F

Fuzzy Expert System: Uses **Fuzzy Logic** instead of Boolean Logic for reasoning in an Expert Systems.

Fuzzy Logic: Introduced by Dr. Lotfi Zadeh of UCB around 1960. A superset of conventional Boolean logic that allows truth values in the interval $[0, 1]$, instead of just the set $\{0, 1\}$ with a rigorous set of mathematical operations that define how the logic system works. The most common definitions for the basic operations are:

$\text{truth}(x \text{AND} y) = \min(\text{truth}(x), \text{truth}(y)); \text{truth}(x \text{OR} y) = \max(\text{truth}(x), \text{truth}(y)); \text{truth}(\text{NOT} x) = 1 - \text{truth}(x)$
where x and y are truth values (degree of belief, degree of membership).

G

Genetic Algorithms: A form of **Evolutionary Computation** using probabilistic search procedures to search large spaces involving states represented by strings.

I

Independent Variable: The classical term for an **input variable**.

Input Variable: The **variables** of a dataset that are generally measurable or preset, and used as the input to the modelling and mining. It is

synonymous with the statistical term **predictor** and the classical term independent variable. See also Output Variable.

Interestingness: A measure used to rank or filter discoveries, particularly in the context of the often overwhelming number of discoveries that typical data mining tools present the user, with example measures including **support**, **confidence**, **lift**, and **leverage**. See also **subjective interestingness**, **objective interestingness**.

Invariant Clustering:

K

Knowledge Discovery in Databases: Fayyad defines it as *the process of identifying valid, novel, potentially useful, and ultimately understandable structure in data.*

L

Lazy Learning: While traditional learning algorithms compile data into abstractions in the process of inducing concept descriptions, lazy learning algorithms, also known as instance-based, memory-based, exemplar-based, experience-based, and case-based, learning delay this process and represent concept descriptions with the data itself. Lazy learning has its roots in disciplines such as pattern recognition, cognitive psychology, statistics, cognitive science, robotics, and information retrieval. It has received increasing attention in several AI disciplines as researchers have explored issues on massively parallel approaches, cost sensitivity, matching algorithms for use with symbolic and structured data representations, formal analyses, rule extraction, variable selection, interaction with knowledge-based systems, integration with other learning/reasoning approaches, and numerous application-specific issues. Many reasons exist for this level of activity: these algorithms are relatively easy to present and analyse, are easily applied, have promising performance on some measures, and are the basis for today's commercially popular case-based reasoning systems.

Learning Curve: A plot of accuracy against training set size, often used to determine the smallest training set size for which adding extra instances leads to little, if any, improvement of accuracy.

Leverage: A measure of **interestingness** in terms of the difference between the observed frequency of occurrence of items and the expected frequency if the items were independent.

Lift: A measure of **interestingness** capturing the increase in the likelihood of an item occurring, for example, within a defined sub population, compared to the full population.

Link Analysis: Explores associations among entities. For example, a law enforcement application might examine familial relationships among suspects and victims, the addresses at which those persons reside, and the telephone numbers that they called during a specified period. The ability of link analysis to represent relationships and associations among entities of different types has proven crucial in assisting human investigators to comprehend complex webs of evidence and draw conclusions that are not apparent from any single piece of information. Computer-based link analysis is increasingly used in law enforcement investigations, fraud detection, telecommunications network analysis, pharmaceuticals research, epidemiology, and many other specialised applications. Also referred to as Social Network Analysis. *David Jensen jensen@cs.umass.edu, 30 Jan 98, datamine-l@nautilus-sys.com*

Logistic Regression:

M

Mean: A measure of the central tendency (i.e., average) of a set of data, calculated as $\frac{\sum_{i=1}^n x_i}{n}$

Mixture Modelling: models a statistical distribution by a mixture (or weighted sum) of other distributions.

O

Objective Interestingness: A measure of **interestingness** often specified in terms of the knowledge structures discovered and the associated data, independent of the application.

Ontology: The vocabulary and set of constraints on the combination of terms of the vocabulary.

Orthogonal Persistence: A form of **persistence** where the persistence of data is a property of the data orthogonal to all other properties of the value and operations on the value.

P

Parametric Techniques: Analysis techniques which assume a model for the data to be analysed. The task of analysis is to select an appropriate model and then to estimate the parameters of the model.

Parallel Coordinates: A multidimensional visualisation technique where the dimensions are represented as a series of parallel lines and individuals are represented as paths through the parallel coordinates. The idea was first presented by [Alfred Inselberg](#) at the University of Illinois in 1959 and he has been developing the idea since. Wegman has also written on Inselberg's work. See the Journal of Computational Statistics, January 1999 for example, for recent significant developments in Statistics.

Persistence: A general term for mechanisms that save values from a program's execution space so that they can be used in a later execution; i.e. by making the values "persist" from one execution to the next. See also [Orthogonal Persistence](#).

Predictor: The statistical term for an [input variable](#).

p-value: The probability that a "discovery" is pure chance. A p-value of 0.05 indicates that there is a 5% chance that the discovery is purely by chance, or conversely, that the discovery is 95% likely to be valid.

Q

Quartile: A point in a distribution of numeric data which identifies one quarter of the data. The first quartile, for example, splits the lowest valued quarter of the dataset from the upper three quarters of the dataset.

R

Reflection: The ability of a program to manipulate as data something representing the state of the program during its own execution. Reflection can be introspective or intercessory, or both. Introspection is the ability to observe and analyse one's own state; intercession is the ability to modify one's own execution state or to alter one's own meaning.

Reinforcement Learning: An approach to learning what to do so as to maximise a numerical reward by discovering the most rewarding actions by trying them out (trial-and-error), which itself may affect the immediate reward and subsequent rewards (delayed reward).

Residual: The residual is essentially what is left after you have accounted for in your modelling. It is the difference between the modelled and the actual.

r^2 : A measure of the closeness of fit of a regression line.

RuleQuest: Ross Quinlan's commercial venture selling the decision tree and rule induction systems **C5.0**, **See5**, and **Cubist**. Refer to the [RuleQuest Home Page](#).

S

Sample: In statistics a large sample is anything 30 or more!

See5: A commercial Windows95/NT decision tree and rule induction product from **RuleQuest** developed by Ross Quinlan as the successor to his very successful and widely used ID3 and C4.5 systems.

Segmentation:

Similarity-Based Learning: Involves comparing several examples of a concept, searching for common variables, in order to define the concept being learned.

Statistics:

Subjective Interestingness: A measure of **interestingness** incorporating the user's point of view.

Supervised Learning: Training data is labelled by class membership.

Support:

T

Temporal Difference Learning: Learning based on the difference between temporally successive predictions, rather than error between predicted and actual values, so that the learner's current prediction for the current input data more closely matches the next prediction at the next time step, thus learning occurring as predictions change over time. Forms the basis for **Reinforcement Learning**.

Terminological Logics: Formalise the notion of Frames as structured types, often called Concepts. These logics include a set of syntactic constructs that form concepts, and other, related, notions such as roles, and are based on formal model-theoretic semantics which provide firm definitions for the syntactic constructs of the logic. Synonyms include Description Logic and Concept Language.

Test Set: A portion of a dataset used to test the performance of a model. See also **Training Set** and **Validation Set**.

Training Set: A portion of a dataset that is used to build a model. See also **Test Set** and **Validation Set**.

U

Unsupervised Learning: Classes are not defined a priori (essentially Cluster Analysis).

V

Validation Set: A portion of a dataset that is used to test the performance of a model as it is being built. See also Training Set and **Test Set**.

Variance: The error in a model due to the fact that the model is built from only a sample of the data taken to represent the whole population. That is, those instances occasionally incorrectly classified by models built by the learning algorithm. Modelling error due to variance can be reduced using Bagging. Compare with **bias**.

Bibliography

- Aggarwal, C. C. and Yu, P. S. (2001), Outlier detection for high dimensional data, in *Proceedings of the 27th ACM SIGMOD International Conference on Management of Data (SIGMOD01)*, pp. 37–46. [452](#)
- Agrawal, R. and Srikant, R. (1994), Fast algorithms for mining association rules in large databases, in J. B. Bocca, M. Jarke and C. Zaniolo, eds, *Proceedings of the 20th International Conference on Very Large Databases (VLDB94)*, Morgan Kaufmann, pp. 487–499. <http://citeseer.ist.psu.edu/agrawal94fast.html>. [414](#)
- Barnett, V. and Lewis, T. (1994), *Outliers in Statistical Data*, John Wiley. [451](#), [452](#)
- Bauer, E. and Kohavi, R. (1999), ‘An empirical comparison of voting classification algorithms: Bagging, boosting, and variants’, *Machine Learning* **36**(1-2), 105–139. <http://citeseer.ist.psu.edu/bauer99empirical.html>. [109](#), [428](#)
- Beyer, K. S., Goldstein, J., Ramakrishnan, R. and Shaft, U. (1999), When is “nearest neighbor” meaningful?, in *Proceedings of the 7th International Conference on Database Theory (ICDT99)*, Jerusalem, Israel, pp. 217–235. <http://citeseer.ist.psu.edu/beyer99when.html>. [452](#)
- Bhandari, I., Colet, E., Parker, J., Pines, Z., Pratap, R. and Ramanujam, K. (1997), ‘Advance scout: data mining and knowledge discovery in nba data’, *Data Mining and Knowledge Discovery* **1**(1), 121–125. [413](#)
- Blake, C. and Merz, C. (1998), ‘UCI repository of machine learning databases’. <http://www.ics.uci.edu/~mlearn/MLRepository.html>. [218](#)

- Breiman, L. (1996), ‘Bagging predictors’, *Machine Learning* **24**(2), 123–140. <http://citeseer.ist.psu.edu/breiman96bagging.html>. 123, 480
- Breiman, L. (2001), ‘Random forests’, *Machine Learning* **45**(1), 5–32. 123
- Breunig, M. M., Kriegel, H., Ng, R. and Sander, J. (1999), OPTICS-OF: Identifying local outliers, in *Proceedings of the XXXXth Conference on Principles of Data Mining and Knowledge Discovery (PKDD99)*, Springer-Verlag, pp. 262–270. 452
- Breunig, M. M., Kriegel, H., Ng, R. and Sander, J. (2000), LOF: Identifying density based local outliers, in *Proceedings of the 26th ACM SIGMOD International Conference on Management of Data (SIGMOD00)* *Proceedings of the 26th ACM SIGMOD International Conference on Management of Data (SIGMOD00)* (2000). 452
- Caruana, R. and Niculescu-Mizil, A. (2006), An empirical comparison of supervised learning algorithms, in *Proceedings of the 23rd International Conference on Machine Learning*, Pittsburgh, PA. 123
- Cendrowska, J. (1987), ‘An algorithm for inducing modular rules’, *International Journal of Man-Machine Studies* **27**(4), 349–370. 484
- Cleveland, W. S. (1993), *Visualizing Data*, Hobart Press, Summit, New Jersey. 57, 259
- Culp, M., Johnson, K. and Michailidis, G. (2006), ‘ada: An r package for stochastic boosting’, *Journal of Statistical Software* **17**(2). <http://www.jstatsoft.org/v17/i02/v17i02.pdf>. 123
- Cypher, A., ed. (1993), *Watch What I Do: Programming by Demonstration*, The MIT Press, Cambridge, Massachusetts. <http://www.acypher.com/wwid/WWIDToC.html>. 161
- Dalgaard, P. (2002), *Introductory Statistics with R*, Statistics and Computing, Springer, New York. xliv
- Freund, Y. and Mason, L. (1999), The alternating decision tree algorithm, in *Proceedings of the 16th International Conference on Machine Learning*, pp. 124–133. 438

BIBLIOGRAPHY587

- Freund, Y. and Schapire, R. E. (1995), A decision-theoretic generalization of on-line learning and an application to boosting, in *Proceedings of the 2nd European Conference on Computational Learning Theory (Eurocolt95)*, Barcelona, Spain, pp. 23–37. <http://citeseer.ist.psu.edu/freund95decisiontheoretic.html>. 109, 428, 430, 439, 441
- Friedman, J. H. (2001), ‘Greedy function approximation: A gradient boosting machine’, *Annals of Statistics* **29**(5), 1189–1232. <http://citeseer.ist.psu.edu/46840.html>. 436
- Friedman, J. H. (2002), ‘Stochastic gradient boosting’, *Computational Statistics and Data Analysis* **38**(4), 367–378. <http://citeseer.ist.psu.edu/friedman99stochastic.html>. 436
- Hahsler, M., Grün, B. and Hornik, K. (2005), *A Computational Environment for Mining Association Rules and Frequent Item Sets*, R Package, Version 0.2-1. 406, 408
- Hastie, T., Tibshirani, R. and Friedman, J. (2001), *The elements of statistical learning: Data mining, inference, and prediction*, Springer Series in Statistics, Springer-Verlag, New York. xliv, 25, 109, 427, 430, 441
- Hawkins, D. (1980), *Identification of Outliers*, Chapman and Hall, London. 451
- Ho, T. K. (1998), ‘The random subspace method for constructing decision forests’, *IEEE Transactions on Pattern Analysis and Machine Intelligence* **20**(8), 832–844. 123
- Jin, W., Tung, A. K. H. and Han, J. (2001), Mining top-n local outliers in large databases, in *Proceedings of the 7th International Conference on Knowledge Discovery and Data Mining (KDD01)*. 452
- King, R. D., Feng, C. and Sutherland, A. (1995), ‘Statlog: Comparison of classification algorithms on large real-world problems’, *Applied Artificial Intelligence* **9**(3), 289–333. 123
- Knorr, E. and Ng, R. (1998), Algorithms for mining distance based outliers in large databases, in *Proceedings of the 24th International Conference on Very Large Databases (VLDB98)*, pp. 392–403. 452

- Knorr, E. and Ng, R. (1999), Finding intensional knowledge of distance-based outliers, in *Proceedings of the 25th International Conference on Very Large Databases (VLDB99)* *Proceedings of the 25th International Conference on Very Large Databases (VLDB99)* (1999), pp. 211–222. 452
- Kohavi, R. (1996), Scaling up the accuracy of naive-Bayes classifiers: A decision tree hybrid, in *Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining (KDD96)*, Portland, OR, pp. 202–207. <http://citeseer.ist.psu.edu/kohavi96scaling.html>. 484
- Lin, W., Orgun, M. A. and Williams, G. J. (2000), Temporal data mining using multilevel-local polynominal models, in *Proceedings of the 2nd International Conference on Intelligent Data Engineering and Automated Learning (IDEAL 2000)*, Hong Kong, Vol. 1983 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 180–186. 453
- Lin, W., Orgun, M. A. and Williams, G. J. (2001), Temporal data mining using hidden markov-local polynomial models, in D. W.-L. Cheung, G. J. Williams and Q. Li, eds, *Proceedings of the 5th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD01)*, Hong Kong, Vol. 2035 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 324–335. 453
- Mingers, J. (1989), ‘An empirical comparison of selection measures for decision-tree induction’, *Machne Learning* 3(4), 319–342. 426
- Proceedings of the 25th International Conference on Very Large Databases (VLDB99)* (1999). 587, 589
- Proceedings of the 26th ACM SIGMOD International Conference on Management of Data (SIGMOD00)* (2000), ACM Press. 586, 589
- Provost, F. J., Jensen, D. and Oates, T. (1999), Efficient progressive sampling, in *Proceedings of the 5th International Conference on Knowledge Discovery and Data Mining (KDD99)*, San Diego, CA, ACM Press, pp. 23–32. <http://citeseer.ist.psu.edu/provost99efficient.html>. 545
- Quinlan, J. R. (1993), *C4.5: Programs for machine learning*, Morgan Kaufmann. 545

BIBLIOGRAPHY

589

- R D (2005), *R Data Import/Export*, version 2.1.1 edn. 215
- Ramaswamy, S., Rastogi, R. and Kyuseok, S. (2000), Efficient algorithms for mining outliers from large data sets, in *Proceedings of the 26th ACM SIGMOD International Conference on Management of Data (SIGMOD00) Proceedings of the 26th ACM SIGMOD International Conference on Management of Data (SIGMOD00)* (2000), pp. 427–438. 452
- Schafer, J. L. (1997), *Analysis of Incomplete Multivariate Data*, Chapman and Hall, London. 88, 374
- Schapire, R. E., Freund, Y., Bartlett, P. and Lee, W. S. (1997), Boosting the margin: a new explanation for the effectiveness of voting methods, in *Proceedings of the 14th International Conference on Machine Learning (ICML97)*, Morgan Kaufmann, pp. 322–330. <http://citeseer.ist.psu.edu/schapire97boosting.html>. 109, 428
- Soares, C., Brazdil, P. B. and Kuba, P. (2004), ‘Meta-learning method to select the kernel width in support vector regression’, *Machine Learning* 54(3), 195–209. 513, 516
- Tukey, J. W. (1977), *Exploratory data analysis*, Addison-Wesley. 60, 299, 361
- Venables, W. N. and Ripley, B. D. (2002), *Modern Applied Statistics with S*, Statistics and Computing, 4th edn, Springer, New York. xliv
- Viveros, M. S., Nearhos, J. P. and Rothman, M. J. (1999), Applying data mining techniques to a health insurance information system., in *Proceedings of the 25th International Conference on Very Large Databases (VLDB99) Proceedings of the 25th International Conference on Very Large Databases (VLDB99)* (1999), pp. 286–294. <http://www.informatik.uni-trier.de/~ley/vldb/ViverosNR96/Article.PS>. 413
- Williams, G. J. (1987), ‘Some experiments in decision tree induction.’, *Australian Computer Journal* 19(2), 84–91. 577
- Williams, G. J. (1988), Combining decision trees: Initial results from the MIL algorithm, in J. S. Gero and R. B. Stanton, eds, *Artificial Intelligence Developments and Applications*, Elsevier Science Publishers B.V. (North-Holland), pp. 273–289. 577

- Williams, G. J. (1991), Inducing and combining decision structures for expert systems, PhD thesis, Australian National University. <http://togaware.redirectme.net/papers/gjwthesis.pdf>. 577
- Yamanishi, K., ichi Takeuchi, J., Williams, G. J. and Milne, P. (2000), On-line unsupervised outlier detection using finite mixtures with discounting learning algorithms, in *Proceedings of the 6th International Conference on Knowledge Discovery and Data Mining (KDD00)*, pp. 320–324. <http://citeseer.ist.psu.edu/446936.html>. 452
- Ye, J. (1998), ‘On measuring and correcting the effects of data mining and model selection’, *Journal of the American Statistical Association* **93**(441), 120–131. 426

Togaware Watermark For Data Mining Survival

Index

- Median/MAD, 85
- .Machine, 189
- .Platform, 187
- .packages (R function), 182
- R Console, 14
- R, 7
- Access
 - Import data into R, 229
 - ada (R package), 110, 123
 - AdaBoost, 109, 427–442
 - adaboost (R function), 433
 - Adjusted, 70, 73, 102
 - Adjustment, 102, 113, 116
 - Advance Scout, 413
 - Age, 30, 32, 56, 70, 72, 90, 95
 - aggregate (R function), 204, 282
 - amap (R package), 391, 487, 492, 499
 - AMD64, 191
 - analysis of variance, 317
 - analysis of variance, 317
 - Annotate, 58, 61
 - ANOVA, 317, *see* analysis of variance
 - apply, *see* lapply, mapply, sapply
 - apply (R function), 374, 381
 - approxfun (R function), 388
 - Apriori, 414
 - apriori, 397
 - apriori (R function), 402, 406, 409, 410
 - array (R function), 206
 - arrows (R function), 240
 - Artificial neural networks, *see* Neural networks
 - arules (R package), 398, 401–403, 406
 - as (R function), 409
 - as.Date (R function), 203, 204
 - as.integer (R function), 284
 - as.logical (R function), 377
 - as.matrix (R function), 192
 - as.yearmon (R function), 204
 - Associate, 22, 129, 130
 - association analysis
 - Apriori, 397–414
 - associations, 576
 - at, 310
 - attach (R function), 210, 223, 237, 319
 - attr (R function), 186
 - attribute, *see* variable
 - audit (Dataset), xxxiv, 16, 18, 32, 55, 69, 70, 97, 102, 120, 151
 - available.packages (R function), 178
 - Bagging, 479
 - bagging, 479–480
 - barchart, 328
 - barchart (R function), 319, 320

- barplot (R function), 272, 350
 Basics, 51
 Baskets, 130
 batch model building, 447
 bayesian analysis
 Bayes theorem, 482
 bbox (R function), 232
 Believe Num Rows, 36
 believeNRows, 227
 Benford, 69
 Benford's Law, 65
 Binning, 95
 binning (R function), 92, 383, 384
 bitmap (R function), 252
 bmp (R function), 252
 Boost, 111
 boost (R package), 433, 436
 Boosting, 109, 427–442, 575
 boosting, 109–111
 bootstrap aggregating, 479
 Bootstrapping, 485–486
 Borgelt, 523–524
 box and whisker plot, *see* box plot
 boxplot, 60, 299, 361
 boxplot (R function), 237, 299–302, 343, 361–363, 374
 breaks, 274
 bxp (R function), 302, 303
 c (R function), 205, 207, 456
 C4.5, 543–545
 capabilities (R function), 189
 caret (R package), 447
 cast (R function), 281
 caTools (R package), 152, 249, 370, 433
 censored data, 453
 check box, xlvi
 chron (R package), 204, 341
 Churn, 453
 class (R function), 375
 Classification
 C4.5, 543–545
 Conditional trees, 495–497
 Decision trees, 105–108, 415–426, 543–545
 K-nearest neighbour, 501–502
 Kernel methods, 513–514
 Neural networks, 509–510
 Support vector machine (SVM), 513–516
 classification
 Naïve Bayes, 481, 484
 classwt, 444
 Cleanup, 95, 97
 Clementine, 547
 clipboard, 184, 216, 229, 230, 379
 Close, 23, 24
 closure, 199
 Cluster, 22
 Clustering
 Hierarchical, 492–493, 499
 K-means, 391–395, 487–492
 cm.colors (R function), 243, 288, 353
 col, 241, 243, 262
 colnames (R function), 207, 209, 261, 345, 377
 color, 273
 colour (R function), 243
 colSums (R function), 380, 381
 comment, 173
 complete.cases (R function), 374
 complex (R function), 249
 complex numbers, 200
 compress, 223
 Conditional trees, 495–497
 confidence, 399

- confusion matrix, 138, 457
- confusion matrix, 138, 457
- contingency table, 138, 457
- contingency table, 138, 457
- continue, 189
- Copy, 23, 24
- cor (R function), 288, 353
- correlation, 288, 353
- cost, 423
- cp, 416
- crude (Dataset), 472
- CSV, 27
- cmtree (R function), 496
- cut (R function), 383, 384, 408, 409
- Data, 15, 26, 27, 29, 30, 83, 113, 199
- data, 385
 - loading, 26
- data import
 - csv, 228
 - txt, 29
- data sources, *see* data import
- data (R function), 217
- data cleaning, 370–377
- Data Entry, 40
- data frame, 208–210
- data import
 - Access, 26
 - arff, 26, 32–34
 - csv, 26–32
 - DB2, 26
 - Excel, 26, 228
 - missing values, *see* missing values
 - MySQL, 26
 - ODBC, 26, 35–37, 228
 - Oracle, 26
 - SQL Server, 26
- SQLite, 26
- Teradata, 26
- txt, 26
- data linking, 379–380
- data transformation, 380–384
 - aggregation, 380
 - Sum of columns, 380
- data types, 199
 - Data frame, 208–210
 - date, 203–204
 - Matrix, 207–208
 - String, 201–202
 - Vector, 205–206
- dataset, 25
 - testing, 26
 - training, 26
- Datasets
 - audit, xxxiv, 16, 18, 32, 55, 69, 70, 97, 102, 120, 151
 - crude, 472
 - iris, 167, 222, 237
 - survey, 221, 377, 457
 - wine, 219, 260, 262, 265, 272, 281, 282, 284, 288, 345, 350, 353, 433
- date, 203–204
- dd.load (R function), 344
- Debian, 165
- decision tree, *see* random forest
- Decision trees, 105–108, 415–426
- Delete Ignored, 97
- density estimate, 62
- dependencies, 13
- Describe, 50, 62
- DescribeDisplay (R package), 343
- Design (R package), 507
- detach (R function), 177, 237
- dev.copy (R function), 255
- dev.cur (R function), 252

- dev.list (R function), 252
- dev.next (R function), 252
- dev.off (R function), 251
- dev.prev (R function), 252
- dev.set (R function), 252
- difftime (R function), 203
- digits, 152, 189
- dim (R function), 261, 345, 377
- distribution, 436
- distributions
 - normal, 63
- divide by zero, 212
- do.call (R function), 209
- download.file (R function), 218
- download.packages (R function), 178
- dprep (R package), 384
- duplicated (R function), 372
- e1071 (R package), 448, 512, 516
- EDA, *see* Exploratory data analysis, *see also* exploratory data analysis
- Eddelbuettel, Dirk, 165
- edit (R function), 175, 207
- Education, 32
- ellipse (R package), 288, 353
- Employment, 32, 72
- ensemble model builder, 109
- Enterprise Miner, 559–562
- entity, 25
- Equal Width, 92
- Equbits Foresight, 549
- Evaluate, 22, 116–118, 136–138, 155
- evaluation
 - risk chart, 101
- example (R function), 175
- Excel, *see* data import
- Execute, 40, 41, 113, 114, 116, 130
- exploratory data analysis, 47, 259
- Explore, 47, 48, 55, 89
- Explore Missing, 78, 79
- Export, 18, 20, 83, 154
- false negative, 138, 457
- false positive, 138, 457
- feature, *see* variable
- feature selection, 384
- fields (R package), 249
- fig (R function), 252
- file (R function), 230
- file.choose (R function), 220
- file.show (R function), 28, 184
- finco (R function), 384
- fix (R function), 207
- Flavanoids, 284
- floor (R function), 151
- for (R function), 377
- Forest, 113
- format (R function), 224, 381
- format.df (R function), 224
- formatC (R function), 224
- Fujitsu, 551–553
- functional, 173
- functional language, 173
- gbm (R function), 436
- gbm (R package), 433, 436
- gc (R function), 193
- gclinfo (R function), 193
- Gender, 95, 453
- get (R function), 186
- getOption (R function), 189
- GGobi, 71, 72
- ggplot (R package), 304
- GhostMiner, 551–553
- GNU/Linux, 165
- gplots (R package), 181, 316
- graphics

INDEX

595

- barchart, 328
- graphics.off (R function), 251
- gray (R function), 243
- grep (R function), 201
- gsub (R function), 202
- head (R function), 213, 261, 345
- Health Insurance Commission, 413
- help, 174, 180
- help (R function), 174, 181
- help.search (R function), 175
- help.start (R function), 174
- Hierarchical clustering, 492–493, 499
- hist (R function), 237, 274
- histogram, 62
- histogram (R function), 273
- Hmisc (R package), 224
- holdout method, 459
- horizontal, 300, 362
- htmlhelp, 174
- hyperedges, 405
- IBM
 - Advance Scout, 413
- ID, 30
- if (R function), 203
- image (R function), 249
- Importance, 116
- imputation, 88
 - multiple imputation, 378
- Income, 32, 69
- incremental model building, 447
- InductionEngine, 555
- inspect (R function), 406
- install
 - GTK+, 10
 - R, 8
 - R packages, 12
 - Rattle, 13
- RGtk2, 12
- install.packages (R function), 165, 178
- installed.packages (R function), 178
- interpreted language, 173
- interquartile range, 60, 299, 361
- invisible (R function), 184, 207
- iris (Dataset), 167, 222, 237
- is.factor (R function), 377
- is.integer (R function), 377
- is.logical (R function), 377
- is.na (R function), 374
- is.numeric (R function), 376, 377
- itemsets, 405
- join, *see* merge
- Join Categoricals, 95
- jpeg (R function), 252
- JPG, 235
- K-means, 391–395, 487–492
- K-Nearest Neighbour, 501
- K-nearest neighbour, 501–502
- Kernel Methods, 513, 514
- Kernel methods, 513
- kernlab (R package), 120, 512–514, 516
- KMeans, 92
- kurtosis, 52
- lapply (R function), 284, 375, 377
- latex (R function), 224
- lattice (R package), 256, 273, 319, 327
- layout (R function), 309
- legend (R function), 241, 262
- length (R function), 152
- levels (R function), 211
- library (R function), 174, 177, 180, 182

- linear interpolation, 388
- load (R function), 222, 223
- locator (R function), 256
- Log, 15, 18, 95, 121, 154, 155, 445
- log (R function), 173
- Logistic regression, 507–508
- LogitBoost (R function), 433
- logitboost (R function), 433
- loss, 423, 424
- lty, 241, 262
- mapply (R function), 185
- maptree (R package), 425
- Marital, 32, 56, 69
- matplot (R function), 262
- matrix, *see* dataset, 207–208
- matrix (R function), 207, 381
- matrix scatterplot, 287
- max (R function), 377
- maxdepth, 416, 434
- mean, 61, 296, 297, 357, 359
- mean (R function), 185, 186, 281, 297, 359
- median, 60, 296, 299, 357, 361
- median (R function), 408
- merge (R function), 379
- Meta algorithms
 - AdaBoost, 427–442
 - Boosting, 427–442
 - Bootstrapping, 485–486
- meta algorithms
 - bagging, 479–480
 - boosting, 109–111
- methods (R function), 175
- mfrow, 302, 310
- min (R function), 377
- minbucket, 416
- minsplit, 416
- missing values
 - in csv files, 30
- mitools (R package), 378
- mode (R function), 200
- Model, 20, 22, 113, 114, 137
- model
 - linear interpolation, 388
- model builders
 - random forest, 111–119
- modelling
 - supervised, 21
 - unsupervised, 21
- Months, 453
- months (R function), 204, 343
- mvpart (R function), 418
- mvpart (R package), 418, 425
- na.omit (R function), 374
- naïve Bayes classifier, 481–484
- nchar (R function), 201, 224
- ncol (R function), 261, 345, 377
- Neural networks, 509–510
- new, 254
- nomenclature, 25–26
- normal distribution, 63
- Normalise, 85
- nrepeat, 449
- nrow (R function), 151, 167, 222, 261, 345, 381
- nsl (R function), 190
- Number of Trees, 111
- object, *see* entity
- object.size (R function), 97, 192
- Occupation, 32
- ODBC, *see* data import
- odbcClose (R function), 229
- odbcConnect (R function), 226
- odbcConnectAccess (R function), 229
- odbcConnectExcel (R function), 228

- ODM, 557–558
 ODMiner, 557–558
 OLAP, 576
 on.exit (R function), 190
 online model building, 447
 options (R function), 152, 178, 189
 Oracle, 557–558
 order (R function), 214
 ordered (R function), 211, 343, 408,
 409
 out of bag, 115
 outlier analysis, 451–452
 packageStatus (R function), 178, 179
 palette (R function), 243, 284
 par (R function), 255, 256, 302
 parms, 423
 party (R package), 496
 paste (R function), 201
 pch, 241, 284
 PDF, 235
 pdf (R function), 252, 255
 percentile, 60, 299, 361
 Phenols, 284
 pie (R function), 237, 265
 pie chart, 265
 pinktoe (R package), 425
 pivot table
 reshape, 281
 plot (R function), 166, 175, 231,
 237, 238, 254, 284, 287, 418
 plot.rpart (R function), 175
 plotcorr (R function), 288, 353
 plotmeans (R function), 316
 plotNetwork (R function), 312
 plots
 matrix scatterplot, 287
 Scatterplot, 237
 scatterplot, 284, 287
 pmatch (R function), 201
 PNG, 235
 png (R function), 252
 PostScript, 235
 postscript (R function), 252, 255
 predict (R function), 421, 456, 457
 PredictionWorks, 555
 predictor
 seeinput variable, 26
 Print, 23, 24
 printcp (R function), 418, 455
 prior, 423, 424
 proc.time (R function), 190
 prompt, 189
 prompt (R function), 233
 q (R function), 167
 qplot (R function), 304
 Quantian live CD, 164
 Quantile, 92
 quantile (R function), 302
 quartile, 60, 299, 361
 quartz (R function), 251
 R, 527–528
 R functions, 186
 .packages, 182
 adaboost, 433
 aggregate, 204, 282
 apply, 374, 381
 approxfun, 388
 apriori, 402, 406, 409, 410
 array, 206
 arrows, 240
 as, 409
 as.Date, 203, 204
 as.integer, 284
 as.logical, 377
 as.matrix, 192

as.yearmon, 204
 attach, 210, 223, 237, 319
 attr, 186
 available.packages, 178
 barchart, 319, 320
 barplot, 272, 350
 bbox, 232
 binning, 92, 383, 384
 bitmap, 252
 bmp, 252
 boxplot, 237, 299–302, 343, 361–
 363, 374
 bxp, 302, 303
 c, 205, 207, 456
 capabilities, 189
 cast, 281
 class, 375
 cm.colors, 243, 288, 353
 colnames, 207, 209, 261, 345,
 377
 colour, 243
 colSums, 380, 381
 complete.cases, 374
 complex, 249
 cor, 288, 353
 ctree, 496
 cut, 383, 384, 408, 409
 data, 217
 dd.load, 344
 detach, 177, 237
 dev.copy, 255
 dev.cur, 252
 dev.list, 252
 dev.next, 252
 dev.off, 251
 dev.prev, 252
 dev.set, 252
 difftime, 203
 dim, 261, 345, 377
 do.call, 209
 download.file, 218
 download.packages, 178
 duplicated, 372
 edit, 175, 207
 example, 175
 fig, 252
 file, 230
 file.choose, 220
 file.show, 28, 184
 finco, 384
 fix, 207
 floor, 151
 for, 377
 format, 224, 381
 format.df, 224
 formatC, 224
 gbm, 436
 gc, 193
 gcinfo, 193
 get, 186
 getOption, 189
 graphics.off, 251
 gray, 243
 grep, 201
 gsub, 202
 head, 213, 261, 345
 help, 174, 181
 help.search, 175
 help.start, 174
 hist, 237, 274
 histogram, 273
 if, 203
 image, 249
 inspect, 406
 install.packages, 165, 178
 installed.packages, 178
 invisible, 184, 207
 is.factor, 377

INDEX

599

is.integer, 377
 is.logical, 377
 is.na, 374
 is.numeric, 376, 377
 jpeg, 252
 lapply, 284, 375, 377
 latex, 224
 layout, 309
 legend, 241, 262
 length, 152
 levels, 211
 library, 174, 177, 180, 182
 load, 222, 223
 locator, 256
 log, 173
 LogitBoost, 433
 logitboost, 433
 mapply, 185
 matplot, 262
 matrix, 207, 381
 max, 377
 mean, 185, 186, 281, 297, 359
 median, 408
 merge, 379
 methods, 175
 min, 377
 mode, 200
 months, 204, 343
 mvpart, 418
 na.omit, 374
 nchar, 201, 224
 ncol, 261, 345, 377
 nrow, 151, 167, 222, 261, 345,
 381
 nsl, 190
 object.size, 97, 192
 odbcClose, 229
 odbcConnect, 226
 odbcConnectAccess, 229
 odbcConnectExcel, 228
 on.exit, 190
 options, 152, 178, 189
 order, 214
 ordered, 211, 343, 408, 409
 packageStatus, 178, 179
 palette, 243, 284
 par, 255, 256, 302
 paste, 201
 pdf, 252, 255
 pie, 237, 265
 plot, 166, 175, 231, 237, 238,
 254, 284, 287, 418
 plot.rpart, 175
 plotcorr, 288, 353
 plotmeans, 316
 plotNetwork, 312
 pmatch, 201
 png, 252
 postscript, 252, 255
 predict, 421, 456, 457
 printcp, 418, 455
 proc.time, 190
 prompt, 233
 q, 167
 qplot, 304
 quantile, 302
 quartz, 251
 rainbow, 367
 randomForest, 374
 rbind, 208, 209
 read.arff, 34
 read.csv, 151, 203, 218, 221
 read.transactions, 403
 read.xls, 229
 readShapePoly, 231
 rect, 246
 relief, 384
 remove.packages, 178

rep, 381
 require, 177
 rescaler, 382
 return, 184
 rev, 214, 377
 rnorm, 332, 343, 363, 384, 388
 rownames, 207, 261, 345
 rpart, 192, 373, 415, 416, 418,
 423, 429, 433, 457
 rpart.control, 434
 Rprof, 190
 RSiteSearch, 175
 runif, 205, 388
 sample, 42, 151, 222, 369
 sample.split, 152, 370
 sapply, 298, 376
 save, 219, 222, 223, 229
 scale, 381
 scan, 215, 230
 search, 177
 seq, 205, 377
 seq.dates, 204
 sessionInfo, 188
 set.seed, 42
 show.settings, 256
 sprintf, 224
 sqlFetch, 226, 228, 229
 sqlQuery, 228, 229
 sqlTables, 226, 228, 229
 stopifnot, 181
 str, 150, 174, 261, 346
 strftime, 203
 strptime, 343
 strsplit, 201
 strwidth, 246
 sub, 201, 202
 subset, 210, 310
 substr, 201
 sum, 381
 summary, 60, 179, 262, 296, 297,
 299, 346, 358, 359, 361, 371,
 404, 409, 410
 sunflower, 314
 svm, 512, 516
 Sys.info, 188
 Sys.sleep, 190
 system, 187
 system.file, 28, 184
 system.time, 190, 193, 380
 t, 208
 table, 135, 152, 455, 457
 tail, 213, 214
 TermDocMatrix, 472
 text, 246, 272, 350
 tim.colors, 249
 traceback, 194
 trellis.par, 256
 trellis.par.get, 256
 trellis.par.set, 319
 tune, 448, 449
 typeof, 199
 unique, 214
 unlist, 284
 unstack, 209
 update.packages, 178
 UseMethod, 186
 vector, 205
 vignette, 174, 175, 406
 which, 212, 434
 win.metafile, 252, 254
 window, 23
 windows, 23, 251
 with, 213, 238
 write.gif, 249
 write.table, 218
 x11, 23, 251
 years, 204
 zoo, 204

- R options
 - at, 310
 - believeNRows, 227
 - breaks, 274
 - classwt, 444
 - col, 241, 243, 262
 - color, 273
 - compress, 223
 - continue, 189
 - cost, 423
 - dependencies, 13
 - digits, 152, 189
 - distribution, 436
 - help, 174, 180
 - horizontal, 300, 362
 - htmlhelp, 174
 - hyperedges, 405
 - itemsets, 405
 - loss, 423, 424
 - lty, 241, 262
 - maxdepth, 434
 - mfrow, 302, 310
 - new, 254
 - parms, 423
 - pch, 241, 284
 - prior, 423, 424
 - prompt, 189
 - rules, 405
 - sampszie, 444
 - scipen, 257
 - strata, 444
 - type, 457
 - usr, 236, 255, 310
 - weight, 429
 - weights, 423, 424
 - where, 421
 - width, 189
 - xlim, 231
 - xpd, 236, 256, 272, 350
 - y, 272, 350
 - ylim, 231
- R packages
 - ada, 110, 123
 - amap, 391, 487, 492, 499
 - arules, 398, 401–403, 406
 - boost, 433, 436
 - caret, 447
 - caTools, 152, 249, 370, 433
 - chron, 204, 341
 - DescribeDisplay, 343
 - Design, 507
 - dprep, 384
 - e1071, 448, 512, 516
 - ellipse, 288, 353
 - fields, 249
 - gbm, 433, 436
 - ggplot, 304
 - gplots, 181, 316
 - Hmisc, 224
 - kernlab, 120, 512–514, 516
 - lattice, 256, 273, 319, 327
 - maptree, 425
 - mitools, 378
 - mpart, 418, 425
 - party, 496
 - pinktoe, 425
 - randomForest, 113
 - randomForrest, 444
 - rattle, 170, 423
 - Rcmdr, 170, 172
 - reshape, 281, 382
 - rggobi, 10
 - ROCR, 135, 138, 455, 460
 - RODBC, 191, 226, 227
 - rpart, 425, 433
 - RWeka, 438
 - sudoku, 194
 - survey, 197

- tm, 471, 472
- tree, 425
- ttda, 471
- xlsReadWrite, 229
- zoo, 204, 264, 341
- R variables
 - .Machine, 189
 - .Platform, 187
 - cp, 416
 - maxdepth, 416
 - minbucket, 416
 - minsplit, 416
 - nrepeat, 449
 - repeat.aggregate, 449
 - sampling, 449
 - sampling.aggregate, 449
 - surrogatestyle, 416
 - version, 187
- radio button, xlvi
- rainbow (R function), 367
- random forest, 111–119
- randomForest (R function), 374
- randomForest (R package), 113
- randomForrest (R package), 444
- Rank, 85
- Rattle, 531–532
 - install, 13
 - start up, 14
- Variables
 - Adjusted, 70, 73, 102
 - Adjustment, 102, 113, 116
 - Age, 30, 32, 56, 70, 72, 90, 95
 - Churn, 453
 - Education, 32
 - Employment, 32, 72
 - Gender, 95, 453
 - ID, 30
 - Income, 32, 69
- Marital, 32, 56, 69
- Months, 453
- Occupation, 32
- Widgets
 - Median/MAD, 85
 - Annotate, 58, 61
 - Associate, 22, 129, 130
 - Basics, 51
 - Baskets, 130
 - Believe Num Rows, 36
 - Benford, 69
 - Binning, 95
 - Boost, 111
 - Cleanup, 95, 97
 - Close, 23, 24
 - Cluster, 22
 - Copy, 23, 24
 - CSV, 27
 - Data, 15, 26, 27, 29, 30, 83, 113
 - Data Entry, 40
 - Delete Ignored, 97
 - Describe, 50, 62
 - Equal Width, 92
 - Evaluate, 22, 116–118, 136–138, 155
 - Execute, 40, 41, 113, 114, 116, 130
 - Explore, 47, 48, 55, 89
 - Explore Missing, 78, 79
 - Export, 18, 20, 83, 154
 - Forest, 113
 - GGobi, 71, 72
 - Importance, 116
 - Join Categoricals, 95
 - KMeans, 92
 - Log, 15, 18, 95, 121, 154, 155, 445
 - Model, 20, 22, 113, 114, 137

- Normalise, 85
 Number of Trees, 111
 Print, 23, 24
 Quantile, 92
 Rank, 85
 Recenter, 85, 88
 Remap, 69, 70, 95
 Risk, 116
 Sample, 41, 138
 Save, 23, 24
 Scale [0,1], 85
 Select, 41, 69, 83, 97, 113, 114, 129, 138
 Settings, 23
 Show Missing, 55, 89
 Summary, 48, 49, 55, 60, 89
 Tools, 41
 Transform, 69, 70, 83, 85, 94, 97
 Two Class, 118
 Unsupervised, 16
 Use Sample, 48
 rattle (R package), 170, 423
 rbind (R function), 208, 209
 Rcmdr (R package), 170, 172
 read.arff (R function), 34
 read.csv (R function), 151, 203, 218, 221
 read.transactions (R function), 403
 read.xls (R function), 229
 readShapePoly (R function), 231
 recall, 458
 Recenter, 85, 88
 record, *see* entity
 rect (R function), 246
 Regression
 Logistic regression, 507–508
 Neural networks, 509–510
 Support vector machine (SVM), 513–516
 regular expressions, 202
 relief (R function), 384
 Remap, 69, 70, 95
 remove.packages (R function), 178
 rep (R function), 381
 repeat.aggregate, 449
 require (R function), 177
 rescaler (R function), 382
 reshape (R package), 281, 382
 return (R function), 184
 rev (R function), 214, 377
 rggobi (R package), 10
 Ridgeway, Greg, 436
 Risk, 116
 risk analysis, 102
 rnorm (R function), 332, 343, 363, 384, 388
 ROOCR (R package), 135, 138, 455, 460
 RODBC (R package), 191, 226, 227
 rownames (R function), 207, 261, 345
 rpart (R function), 192, 373, 415, 416, 418, 423, 429, 433, 457
 rpart (R package), 425, 433
 rpart.control (R function), 434
 Rprof (R function), 190
 RSiteSearch (R function), 175
 rug, 62
 rules, 405
 runif (R function), 205, 388
 RWeka (R package), 438
 Salford Systems, 569
 Sample, 41, 138
 sample (R function), 42, 151, 222, 369
 sample.split (R function), 152, 370

- sampling, 449
 sampling.aggregate, 449
 sampsize, 444
 sapply (R function), 298, 376
 SAS, 559–562
 Save, 23, 24
 save (R function), 219, 222, 223, 229
 scale (R function), 381
 Scale [0,1], 85
 scan (R function), 215, 230
 scatterplot, 237, 284, 287
 scipen, 257
 script file, 162, 166
 search (R function), 177
 Select, 41, 69, 83, 97, 113, 114, 129, 138
 selection, *see* clipboard
 seq (R function), 205, 377
 seq.dates (R function), 204
 sessionInfo (R function), 188
 set.seed (R function), 42
 Settings, 23
 shapefiles, 230
 Show Missing, 55, 89
 show.settings (R function), 256
 skewness, 54
 sprintf (R function), 224
 SPSS, 547
 sqlFetch (R function), 226, 228, 229
 sqlQuery (R function), 228, 229
 sqlTables (R function), 226, 228, 229
 Statistica, 563–567
 StatSoft, 563–567
 Stem-and-leaf, 267, 347
 stopifnot (R function), 181
 str (R function), 150, 174, 261, 346
 strata, 444
 strftime (R function), 203
 string, 201–202
 strptime (R function), 343
 strsplit (R function), 201
 strwidth (R function), 246
 sub (R function), 201, 202
 subset (R function), 210, 310
 substr (R function), 201
 sudoku (R package), 194
 sum (R function), 381
 Summary, 48, 49, 55, 60, 89
 summary (R function), 60, 179, 262, 296, 297, 299, 346, 358, 359, 361, 371, 404, 409, 410
 sunflower (R function), 314
 support, 399
 Support vector machine (SVM), 513–516
 surrogatestyle, 416
 survey (Dataset), 221, 377, 457
 survey (R package), 197
 survival analysis, 453
 SVM, 513–516
 svm (R function), 512, 516
 Sys.info (R function), 188
 Sys.sleep (R function), 190
 system (R function), 187
 system.file (R function), 28, 184
 system.time (R function), 190, 193, 380
 t (R function), 208
 table, *see* dataset
 table (R function), 135, 152, 455, 457
 tail (R function), 213, 214
 temporal analysis, 453–454
 TermDocMatrix (R function), 472
 test set, 459
 text (R function), 246, 272, 350

INDEX

605

- tim.colors (R function), 249
 tm (R package), 471, 472
 Togaware, 531–532
 Tools, 41
 - Borgelt, 523–524
 - Clementine, 547
 - Enterprise Miner, 559–562
 - Equbits Foresight, 549
 - Ghostminer, 551–553
 - InductionEngine, 555
 - ODM, 557–558
 - ODMiner, 557–558
 - R, 527–528
 - Rattle, 531–532
 - SAS Enterprise Miner, 559–562
 - Statistica, 563–567
 - TreeNet, 569
 - Virtual Predict, 571–572
 - Weka, 535–537
 traceback (R function), 194
 training set, 459
 Transform, 69, 70, 83, 85, 94, 97
 tree (R package), 425
 TreeNet, 569
 trellis.par (R function), 256
 trellis.par.get (R function), 256
 trellis.par.set (R function), 319
 true negative, 138, 457
 true positive, 138, 457
 true positive rate, 458
 ttда (R package), 471
 tune (R function), 448, 449
 Two Class, 118
 type, 457
 typeof (R function), 199
 unique (R function), 214
 University of Magdeburg, 523–524
 University of Waikato, 535–537
- unlist (R function), 284
 unstack (R function), 209
 Unsupervised, 16
 update.packages (R function), 178
 Use Sample, 48
 UseMethod (R function), 186
 usr, 236, 255, 310
- variable, 25
 - descriptive
 - seeinput variable, 26
 - independent
 - seeinput variable, 26
 - input, 25
 - output, 25
 variable selection, 384–385
 variance, 296, 357, 575
 vector, 205–206
 vector (R function), 205
 Vendors
 - Equbits, 549
 - Fujitsu, 551–553
 - Oracle, 557–558
 - PredictionWorks, 555
 - Salford Systems, 569
 - SAS, 559–562
 - SPSS, 547
 - StatSoft, 563–567
 - Togaware, 531–532
 - University of Magdeburg, 523–524
 - University of Waikato, 535–537
 - Virtual Genetics, 571–572
 version, 187
 vignette (R function), 174, 175, 406
 Virtual Genetics, 571–572
 Virtual Predict, 571–572
 weight, 429

weights, 423, 424
Weka, 535–537
where, 421
which (R function), 212, 434
Wickham, Hadley, 382
widget, xvii
width, 189
win.metafile (R function), 252, 254
window (R function), 23
windows (R function), 23, 251
wine (Dataset), 219, 260, 262, 265,
 272, 281, 282, 284, 288, 345,
 350, 353, 433
with (R function), 213, 238
write.gif (R function), 249
write.table (R function), 218
x11 (R function), 23, 251
xlim, 231
xlsReadWrite (R package), 229
xpd, 236, 256, 272, 350

y, 272, 350
years (R function), 204
ylim, 231

zoo (R function), 204
zoo (R package), 204, 264, 341