



RevoScaleR™ 7.0

User's Guide

The correct bibliographic citation for this manual is as follows: Revolution Analytics, Inc. 2013. *RevoScaleR 7.0 User's Guide*. Revolution Analytics, Inc., Mountain View, CA.

RevoScaleR 7.0 User's Guide

Copyright © 2013 Revolution Analytics, Inc. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of Revolution Analytics.

U.S. Government Restricted Rights Notice: Use, duplication, or disclosure of this software and related documentation by the Government is subject to restrictions as set forth in subdivision (c) (1) (ii) of The Rights in Technical Data and Computer Software clause at 52.227-7013.

Revolution R, Revolution R Enterprise, RPE, RevoScaleR, RevoDeployR, RevoTreeView, and Revolution Analytics are trademarks of Revolution Analytics.

Other product names mentioned herein are used for identification purposes only and may be trademarks of their respective owners.

Revolution Analytics
2570 W. El Camino Real
Suite 222
Mountain View, CA 94040
USA.

Revised on December 2, 2013

We want our documentation to be useful, and we want it to address your needs. If you have comments on this or any Revolution document, send e-mail to doc@revolutionanalytics.com. We'd love to hear from you.

Contents

Chapter 1. Introduction	1
1.1 Why RevoScaleR?	1
1.1.1 Accessing External Data Sets	2
1.1.2 Efficiently Storing and Retrieving Data	2
1.1.3 Data Cleaning, Exploration, and Manipulation	2
1.1.4 Statistical Analysis	2
1.1.5 Writing Your Own Analyses for Large Data Sets	3
1.2 Getting Started	3
1.2.1 Accessing External Data Sets	3
1.2.2 Data Cleaning, Exploration, and Transformations	4
1.2.3 Statistical Analysis	6
1.2.4 Writing Your Own Analyses for Large Data Sets	7
1.3 Sample Data for Use with RevoScaleR	9
1.4 Managing Threads	10
1.5 Generating Random Numbers	11
1.6 Using RevoScaleR with Rscript	11
1.7 Getting Help	12
Chapter 2. Importing Data	13
2.1 Data Compression in .xdf Files	13
2.2 Importing Delimited Text Data	14
2.2.1 Specifying Delimiters	15
2.2.2 Specifying a Missing Value String	15
2.3 Importing Fixed-Format Data	16
2.4 Importing SAS Data	17
2.5 Importing SPSS Data	18
2.6 Specifying Variable Data Types	18
2.7 Specifying Additional Variable Information	21
2.8 Appending to an Existing File	21

2.9	Transforming Data on Import	22
2.10	Converting Dates Stored As Character Strings	22
2.11	Splitting Data Files	23
2.12	Reading Data from an .xdf File into a Data Frame	24
2.13	Data Sources	25
2.14	Using Data from the Hadoop Distributed File System	27
2.14.1	Note on Using RevoScaleR with rhdfs.....	28
Chapter 3.	Transforming and Subsetting Data.....	29
3.1	Creating a Subset of Rows and Columns.....	29
3.2	Transforming Data with rxDataStep	31
3.2.1	Creating and Transforming Variables	31
3.2.2	Subsetting and Transforming Variables.....	34
3.3	Using the Data Step to Create an .xdf File from a Data Frame	36
3.4	Converting .xdf Files to Text.....	36
3.5	Re-Blocking an .xdf File	37
3.6	Modifying Variable Information.....	37
3.7	Sorting Data.....	38
3.7.1	Removing Duplicates While Sorting.....	39
3.7.2	The rxQuantile Function and the Five-Number Summary.....	42
3.8	Merging Data.....	42
3.8.1	Inner Merge	43
3.8.2	Outer Merge	44
3.8.3	One-to-one Merge	45
3.8.4	Union Merge	46
3.8.5	Using rxMerge with .xdf files	46
3.9	Recoding Factors	47
3.9.1	Recoding Factors to Ensure Variable Compatibility	50
Chapter 4.	Models in RevoScaleR.....	51
4.1	External Memory Algorithms	51

4.2	Formulas in RevoScaleR	52
4.3	Letting the Data Speak For Itself	53
4.4	Cubes and Cube Regression	53
Chapter 5.	Data Summaries	54
5.1	Using Variable Information	54
5.2	Formulas for rxSummary.....	56
5.3	Computation of Summary Statistics by Group Using Interactions	57
5.4	On-the-Fly Factors in Formulas	57
5.5	Writing By-Group Summary Statistics to an .xdf File	60
5.6	Transforming Data in rxSummary	61
Chapter 6.	Crosstabs	63
6.1	Letting the Data Speak Example 1: Analyzing U.S. 2000 Census Data	65
6.2	Transforming Data.....	67
6.3	Cross-Tabulation with rxCrossTabs	69
6.4	A Large Data Example.....	72
6.5	Tests of Independence on Cross-Tabulated Data	73
6.6	Odds Ratios and Risk Ratios	75
Chapter 7.	Fitting Linear Models.....	77
7.1	Obtaining a Summary of the Model.....	78
7.2	Using Probability Weights	79
7.3	Using Frequency Weights.....	80
7.4	Using rxLinMod with R Data Frames	82
7.5	Using the Cube Option for Conditional Predictions.....	82
7.6	Fitted Values, Residuals, and Prediction	84
7.7	Prediction Standard Errors, Confidence Intervals, and Prediction Intervals	85
7.8	Stepwise Variable Selection	89
7.8.1	Methods of Variable Selection.....	91
7.8.2	Specifying Model Scope	92
7.8.3	Specifying the Selection Criterion.....	92

7.9	Fixed-Effects Models	93
7.10	Least Squares Dummy Variable (LSDV) Models	94
7.10.1	A Quick Review of Interacting Factors	94
7.10.2	Using Dummy Variables in rxLinMod: Letting the Data Speak Example 2	97
7.11	Intercept-Only Models.....	103
Chapter 8.	Fitting Logistic Regression Models	104
8.1	Stepwise Logistic Regression.....	105
8.2	Prediction	106
8.3	Prediction Standard Errors and Confidence Intervals.....	107
Chapter 9.	Fitting Generalized Linear Models.....	109
9.1	A Simple Example Using the Poisson Family	110
9.2	An Example Using the Gamma Family	113
9.3	An Example Using the Tweedie Family	114
9.4	Stepwise Generalized Linear Models	120
Chapter 10.	Estimating Decision Tree Models.....	122
10.1	The rxDTree Algorithm	122
10.2	A Simple Classification Tree.....	123
10.3	A Simple Regression Tree	124
10.4	A Larger Regression Tree Model	125
10.5	Controlling the Model Fit	126
10.6	Large Data Tree Models.....	127
10.7	Handling Missing Values.....	131
10.8	Prediction.....	132
10.9	Visualizing Trees	132
Chapter 11.	Estimating Decision Forest Models	136
11.1	A Simple Classification Forest.....	137
11.2	A Simple Regression Forest	137
11.3	A Larger Regression Forest Model.....	138
11.4	Large Data Decision Forest Models	138

Chapter 12.	Estimating Correlation and Variance/Covariance Matrices.....	142
12.1	Computing Cross-Product Matrices	142
12.2	Computing a Correlation Matrix for Use in Factor Analysis	143
12.3	Computing A Covariance Matrix for Principal Components Analysis	147
12.4	A Large Data Principal Components Analysis	149
Chapter 13.	Clustering	152
13.1	K-means Clustering	152
13.1.1	Clustering the Airline Data	152
13.1.2	Using the Cluster Membership Information	154
Chapter 14.	Converting RevoScaleR Model Objects for Use in PMML.....	156
Chapter 15.	Data Sources.....	159
15.1	Using Data Sources with Analysis Functions	160
15.2	Working with an Xdf Data Source	161
15.3	Using an Xdf Data Source with biglm	162
Chapter 16.	Transform Functions	163
16.1	Creating Variables.....	163
16.2	Using Additional Objects or Data in a Transform Function	165
16.3	Creating a Row Selection Variable.....	166
16.4	Using Internal Variables in a Transformation Function: An Example Computing Moving Averages.....	167
16.5	How Transforms Are Evaluated	171
16.6	Transformations to Avoid	171
Chapter 17.	Visualizing Huge Data Sets: An Example from the U.S. Census	172
17.1	Examining the Data.....	172
17.2	Extending the Analysis.....	177
Bibliography.....		184

Chapter 1.

Introduction

1.1 Why RevoScaleR?

RevoScaleR™ provides tools for both High Performance Computing (HPC) and High Performance Analytics (HPA) with R. HPC capabilities allow you to distribute the execution of essentially any R function across cores and nodes, and deliver the results back to the user. HPA adds ‘big data’ to the challenge. This guide focuses on these HPA ‘big data’ capabilities. [See the *RevoScaleR Distributed Computing Guide* for information on both HPC and HPA capabilities in a distributed context.]

RevoScaleR provides tools for scalable data management and analysis. These tools can be used with data sets in memory, and applied the same way to huge data sets stored on disk. It includes functionality for:

- 1) Accessing external data sets (SAS, SPSS, ODBC, Teradata, and delimited and fixed format text) for analysis in R
- 2) Efficiently storing and retrieving data in a high performance data file
- 3) Cleaning, exploring, and manipulating data
- 4) Fast, basic statistical analyses

RevoScaleR also includes an extensible framework for writing your own analyses for big data sets.

2 Why RevoScaleR?

With RevoScaleR, you can use R to analyze data sets far larger than can be kept in memory. This is possible because RevoScaleR uses external memory algorithms that allow it to work on one chunk of data at a time (that is, a subset of the rows and perhaps the variables in the data set), update the results, and proceed through all available data.

1.1.1 Accessing External Data Sets

Data can be stored in a wide-variety of formats. Typically, the first step in any RevoScaleR analysis is to make the data accessible. With RevoScaleR's data import capability, you can access data from a SAS file, SPSS file, fixed format or delimited text file, an ODBC connection, or a Teradata data base, bringing it into a data frame in memory, or storing it for fast access in chunks on disk.

1.1.2 Efficiently Storing and Retrieving Data

A key component of RevoScaleR is a data file format (.xdf) that is extremely efficient for both reading and writing data. You can create .xdf files by importing data files or from R data frames, and add rows or variables to an existing .xdf file. Once your data is in this file format you can use it directly with analysis functions provided with RevoScaleR, or quickly extract a subsample and read it into a data frame in memory for use in other R functions.

1.1.3 Data Cleaning, Exploration, and Manipulation

When working with a new data set, the first step is to clean and explore. With RevoScaleR, you can quickly obtain information about your data set (e.g., how many rows and variables) and the variables in your data set (e.g., name, data type, value labels). With RevoScaleR's summary statistics and cube functionality, you can examine summary information about your data and quickly plot histograms or relationships between variables.

RevoScaleR also provides all of the power of R to use in data transformations and manipulations. In RevoScaleR's data step functionality, you can specify R expressions to transform specific variables and have them automatically applied to a single data frame or to each block of data as it is read in from an .xdf file. You can create new variables, recode variables, and set missing values with all the flexibility of the R language.

1.1.4 Statistical Analysis

In addition to descriptive statistics and crosstabs, RevoScaleR provides functions for fitting linear and binary logistic regression models, generalized linear models, k-means models, and decision trees and forests. These functions access .xdf files or other data sources directly or operate on data frames in memory. Because these functions are so efficient and do not require that all of the data be in memory at one time, you can analyze huge data sets without requiring huge computing power. In particular, you can relax assumptions previously required. For example, instead of assuming a linear or polynomial functional form in a model, you can break

independent variables into many categories providing a completely flexible functional form. The many degrees of freedom provided by large data sets, combined with RevoScaleR's efficiency, make this approach feasible.

1.1.5 Writing Your Own Analyses for Large Data Sets

All of the main analysis functions in RevoScaleR use updating or external memory algorithms, that is, they analyze a chunk of data, update the results, then move on to the next chunk of data and repeat the process. When all the data has been processed (sometimes multiple times), final results can be calculated and the analysis is complete. You can write your own functions to process a chunk of data and update results, and use RevoScaleR functionality to provide you with access to your data file chunk by chunk.

1.2 Getting Started

In this initial chapter, we take a quick walk through the five feature areas described above using a sample data set. More detailed information about the functions used is available in the other chapters of this manual. We then briefly describe other sample data available for use with RevoScaleR. Finally, we describe the online help available. For additional introductory material, see the *RevoScaleR Getting Started Guide*. For information on using RevoScaleR for distributing computations over more than one computer, see the *RevoScaleR Distributed Computing Guide* and the various Getting Started guides for particular distributed computing platforms: Microsoft HPC Server, IBM Platform LSF, and Hadoop. For more information on accessing databases via ODBC, see the *RevoScaleR ODBC Import Guide*. The *RevoScaleR Teradata Data Source Getting Started Guide* will help with accessing data from a Teradata database.

1.2.1 Accessing External Data Sets

The most common way to store data is in a text file. For example, a comma-delimited, text data file containing a subsample of information on airline departures and arrivals in the United States is available in the RevoScaleR sample data directory. (More examples using this data file are available in the *Getting Started Guide*.) The sample code below will import it using the *rxImport* function. There are a total of 600,000 rows in the data file. By specifying the argument *rowsPerRead*, we read and write the data in 3 blocks of 200,000 rows each.

```
inFile <- file.path(rxGetOption("sampleDataDir"), "AirlineDemoSmall.csv")
airData <- rxImport(inData=inFile, outFile = "airExample.xdf",
  stringsAsFactors = TRUE, missingValueString = "M", rowsPerRead = 200000)
```

If the *outFile* argument had been omitted, the returned *airData* object would be a data frame containing the data. Since the imported data is being stored in an .xdf file, *rxImport* returns an .xdf data source object. This object represents the .xdf data file, but doesn't take up

4 Getting Started

much memory. It can be used in many other RevoScaleR objects interchangeably with a data frame.

To get basic information about the data set and variables, we use *rxGetInfo*:

```
rxGetInfo(airData, getVarInfo = TRUE)
```

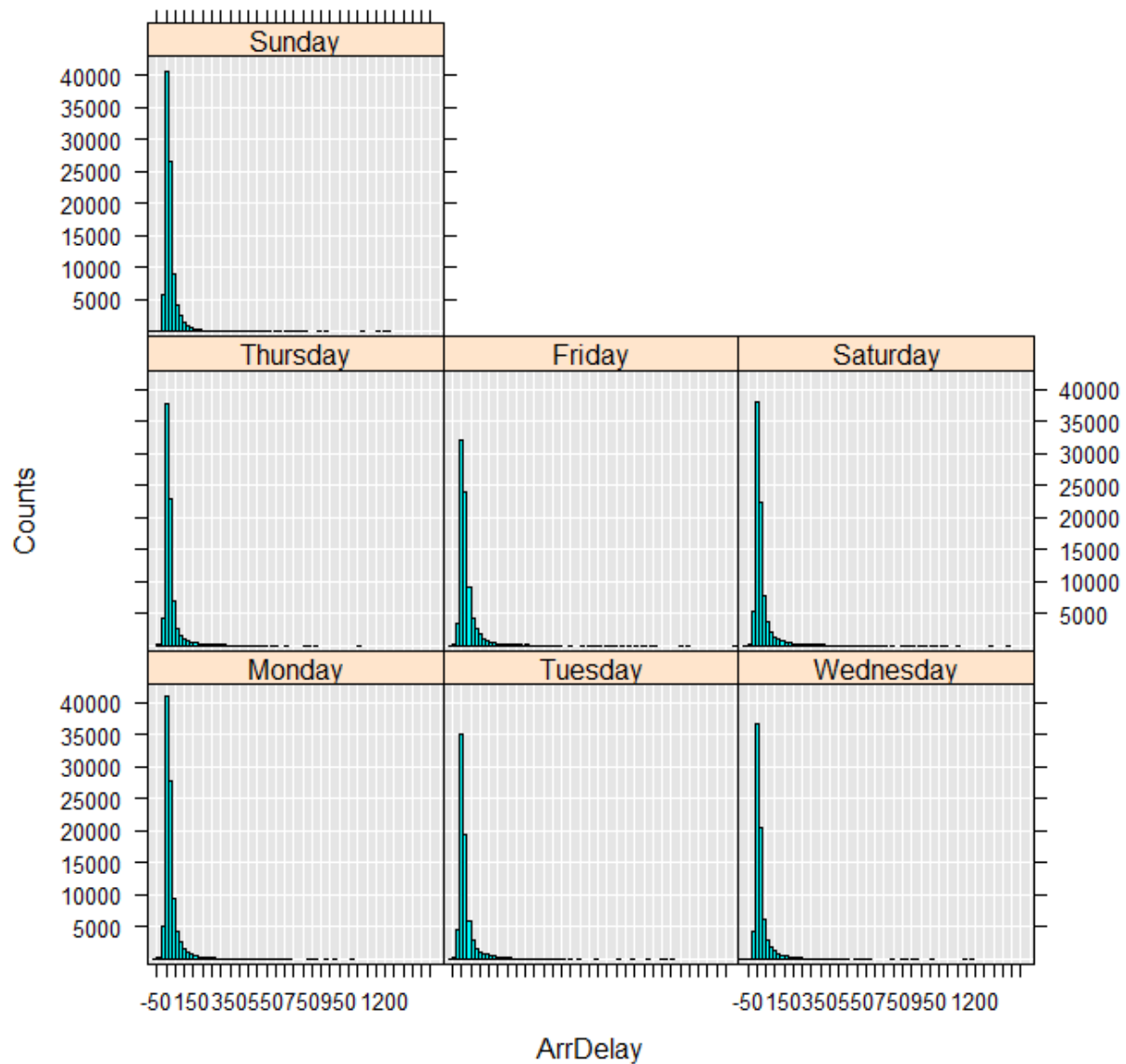
which results in the following output:

```
File name: C:\YourOutputPath\airExample.xdf
Number of observations: 6e+05
Number of variables: 3
Number of blocks: 3
Compression type: zlib
Variable information:
Var 1: ArrDelay, Type: integer, Low/High: (-86, 1490)
Var 2: CRSDepTime, Type: numeric, Storage: float32, Low/High: (0.0167, 23.9833)
Var 3: DayOfWeek
      7 factor levels: Monday Tuesday Wednesday Thursday Friday Saturday
      Sunday
```

1.2.2 Data Cleaning, Exploration, and Transformations

Let's take a quick look at the data. We can use the example *rxHistogram* function to show the distribution in arrival delay by day of week. It uses the *rxCube* function (described later in this manual) to calculate information for a histogram:

```
rxHistogram(~ArrDelay|DayOfWeek, data = airData)
```



We can also compute descriptive statistics for the variable:

```
rxSummary(~ ArrDelay, data = airData )
```

Call:

```
rxSummary(formula = ~ArrDelay, data = airData)
```

Summary Statistics Results for: ~ArrDelay

File name:

C:\YourOutputPath\airExample.xdf

Number of valid observations: 6e+05

Name	Mean	StdDev	Min	Max	ValidObs	MissingObs
ArrDelay	11.31794	40.68854	-86	1490	582628	17372

6 Getting Started

Our first look tells us two important things: arrival delay has a long “tail” for every day of the week, with a few flights delayed for well over two hours, and there are quite a few missing values (17,372). Presumably the missing values for arrival delay represent flights that did not arrive, that is, were cancelled.

We can use RevoScaleR’s data step functionality to create a new variable, *VeryLate*, that represents flights that were either over two hours late or canceled. Since we have our original data safely stored in a text file, we will simply add this variable to our existing *airline.xdf* file using the *transforms* argument to *rxDataStep*. The *transforms* argument takes a list of one or more R expressions, typically in the form of assignment statements. In this case, we use the following expression: `list(VeryLate = (ArrDelay > 120 | is.na(ArrDelay)))`

The full RevoScaleR data step then consists of the following steps:

- 1) Read in the *data* a block (200,000 rows) at a time.
- 2) For each block, pass the *ArrDelay* data to the R interpreter for processing the transformation to create *VeryLate*.
- 3) Write the data out to the data set a block at a time. The argument *overwrite=TRUE* allows us to overwrite the data file.

```
airData <- rxDataStep(inData = airData, outFile = "airExample.xdf",
  transforms=list(VeryLate = (ArrDelay > 120 | is.na(ArrDelay))),
  overwrite = TRUE)
```

1.2.3 Statistical Analysis

We can perform statistical analysis using a data frame or .xdf file. For example, we can estimate a logistic regression on whether or not a flight is “very late” depending on the day of week using *rxLogit*:

```
logitResults <- rxLogit(VeryLate ~ DayOfWeek, data = airData )
summary(logitResults)
```

Call:
rxLogit(formula = VeryLate ~ DayOfWeek, data = airData)

Logistic Regression Results for: VeryLate ~ DayOfWeek
File name:
C:\YourOutputPath\airExample.xdf
Dependent variable(s): VeryLate
Total independent variables: 8 (Including number dropped: 1)
Number of valid observations: 6e+05
Number of missing observations: 0
-2*LogLikelihood: 251244.7201 (Residual deviance on 599993 degrees of freedom)

Coefficients:

	Estimate	Std. Error	z value	Pr(> z)	
(Intercept)	-3.29095	0.01745	-188.64	2.22e-16	***
DayOfWeek=Monday	0.40086	0.02256	17.77	2.22e-16	***
DayOfWeek=Tuesday	0.84018	0.02192	38.33	2.22e-16	***
DayOfWeek=Wednesday	0.36982	0.02378	15.55	2.22e-16	***

```

DayOfWeek=Thursday    0.29396    0.02400    12.25 2.22e-16 ***
DayOfWeek=Friday      0.54427    0.02274    23.93 2.22e-16 ***
DayOfWeek=Saturday    0.48319    0.02282    21.18 2.22e-16 ***
DayOfWeek=Sunday      Dropped    Dropped Dropped Dropped
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Condition number of final variance-covariance matrix: 16.7804
Number of iterations: 3

```

The results show that in this sample, a flight on Tuesday is most likely to be very late or cancelled, followed by flights departing on Friday. In this model, Sunday is the control group, so that coefficient estimates for other days of the week are relative to Sunday. The intercept shown is the same as the coefficient you would get for Sunday if you omitted the intercept term:

```

logitResults2 <- rxLogit(VeryLate ~ DayOfWeek - 1, data = airData )
summary(logitResults2)

Call:
rxLogit(formula = VeryLate ~ DayOfWeek - 1, data = airData)

Logistic Regression Results for: VeryLate ~ DayOfWeek - 1
File name:
  C:\YourOutputPath\airExample.xdf
Dependent variable(s): VeryLate
Total independent variables: 7
Number of valid observations: 6e+05
Number of missing observations: 0
-2*LogLikelihood: 251244.7201 (Residual deviance on 599993 degrees of freedom)

Coefficients:
              Estimate Std. Error z value Pr(>|z|)
DayOfWeek=Monday   -2.89008    0.01431  -202.0 2.22e-16 ***
DayOfWeek=Tuesday   -2.45077    0.01327  -184.7 2.22e-16 ***
DayOfWeek=Wednesday -2.92113    0.01617  -180.7 2.22e-16 ***
DayOfWeek=Thursday  -2.99699    0.01648  -181.9 2.22e-16 ***
DayOfWeek=Friday    -2.74668    0.01459  -188.3 2.22e-16 ***
DayOfWeek=Saturday  -2.80776    0.01471  -190.9 2.22e-16 ***
DayOfWeek=Sunday    -3.29095    0.01745  -188.6 2.22e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Condition number of final variance-covariance matrix: 1
Number of iterations: 7

```

1.2.4 Writing Your Own Analyses for Large Data Sets

We can have RevoScaleR provide data to us, one chunk at a time, and perform our own custom analysis. There are four basic steps in the analysis:

- 1) Initialize results
- 2) Process data, a chunk at a time
- 3) Update results, after processing each chunk
- 4) Final processing of results

8 Getting Started

For illustrative purposes, suppose that we want to compute the average arrival delay for flights that leave in the morning, afternoon, and evening. For each chunk of data, we need to compute the sum of arrival delay for each of the three time intervals, as well as the counts for each interval. We will accumulate these results in a list of “transformObjects” containing the six values. At the end after processing all the data, we will divide the accumulated totals by the accumulated counts to compute the averages.

Most of the work takes place within a transformation function, which processes the data and updates the results for each chunk of data that is read in. We use the `.rxGet` and `.rxSet` functions to store information from one pass of the data to the next. Because we are processing data and not creating newly transformed variables, we return NULL from the function:

```
ProcessAndUpdateData <- function( data )
{
  # Process Data
  notMissing <- !is.na(data$ArrDelay)
  morning <- data$CRSDepTime >= 6 & data$CRSDepTime < 12 & notMissing
  afternoon <- data$CRSDepTime >= 12 & data$CRSDepTime < 17 & notMissing
  evening <- data$CRSDepTime >= 17 & data$CRSDepTime < 23 & notMissing
  mornArr <- sum(data$ArrDelay[morning], na.rm = TRUE)
  mornCounts <- sum(morning, na.rm = TRUE)
  afterArr <- sum(data$ArrDelay[afternoon], na.rm = TRUE)
  afterCounts <- sum(afternoon, na.rm = TRUE)
  evenArr <- sum(data$ArrDelay[evening], na.rm = TRUE)
  evenCounts <- sum(evening, na.rm = TRUE)

  # Update Results
  .rxSet("toMornArr", mornArr + .rxGet("toMornArr"))
  .rxSet("toMornCounts", mornCounts + .rxGet("toMornCounts"))
  .rxSet("toAfterArr", afterArr + .rxGet("toAfterArr"))
  .rxSet("toAfterCounts", afterCounts + .rxGet("toAfterCounts"))
  .rxSet("toEvenArr", evenArr + .rxGet("toEvenArr"))
  .rxSet("toEvenCounts", evenCounts + .rxGet("toEvenCounts"))

  return( NULL )
}
```

Our transformation object values are initialized in a list passed into `rxDataStep`. We also use the argument `returnTransformObjects` to indicate that we want updated values of the `transformObjects` returned rather than a transformed data set:

```
totalRes <- rxDataStep( inData = airData , returnTransformObjects = TRUE,
  transformObjects =
    list(toMornArr = 0, toAfterArr = 0, toEvenArr = 0,
        toMornCounts = 0, toAfterCounts = 0, toEvenCounts = 0),
  transformFunc = ProcessAndUpdateData,
  transformVars = c("ArrDelay", "CRSDepTime"))
```


The `rxDataStep` function will automatically chunk through the data for us. All we need to do is process the final results:

```
FinalizeResults <- function(totalRes)
{
  return(data.frame(
    AveMorningDelay = totalRes$toMornArr / totalRes$toMornCounts,
    AveAfternoonDelay = totalRes$toAfterArr / totalRes$toAfterCounts,
    AveEveningDelay = totalRes$toEvenArr / totalRes$toEvenCounts))
}
FinalizeResults(totalRes)
```

The calculated results are:

	AveMorningDelay	AveAfternoonDelay	AveEveningDelay
1	6.146039	13.66912	16.71271

In this case we can check our results by using the `rowSelection` argument in `rxSummary`:

```
rxSummary(~ArrDelay, data = "airExample.xdf",
  rowSelection = CRSDepTime >= 6 & CRSDepTime < 12 & !is.na(ArrDelay))

Call:
rxSummary(formula = ~ArrDelay, data = "airExample.xdf",
  rowSelection = CRSDepTime >= 6 & CRSDepTime < 12 & !is.na(ArrDelay))

Summary Statistics Results for: ~ArrDelay
File name:
  C:\YourOutputPath\airExample.xdf
Number of valid observations: 234403

Name      Mean      StdDev  Min Max  ValidObs MissingObs
ArrDelay  6.146039  35.4734  -85 1490  234403     0
```

1.3 Sample Data for Use with RevoScaleR

Sample data is available both within the `RevoScaleR` package and from the Revolution Analytics ftp site. To view the files available within the package, use the following command:

```
list.files(system.file("SampleData", package = "RevoScaleR"))
```

This should yield the following list:

```
[1] "AirlineDemo1kNoMissing.csv" "AirlineDemoSmall.csv"
[3] "AirlineDemoSmall.xdf"      "CensusWorkers.xdf"
[5] "claims.dat"                "claims.sas7bdat"
[7] "claims.sav"                "claims.sd7"
[9] "claims.sqlite"             "claims.sts"
[11] "claims.txt"                "claims.xdf"
[13] "claimsExtra.txt"           "claimsQuote.txt"
[15] "claimsTab.txt"             "CustomerSurvey.xdf"
[17] "DJIAdaily.xdf"             "fourthgraders.xdf"
[19] "Kyphosis.xdf"              "mortDefaultSmall.xdf"
[21] "mortDefaultSmall2000.csv"   "mortDefaultSmall2001.csv"
[23] "mortDefaultSmall2002.csv"   "mortDefaultSmall2003.csv"
[25] "mortDefaultSmall2004.csv"   "mortDefaultSmall2005.csv"
[27] "mortDefaultSmall2006.csv"   "mortDefaultSmall2007.csv"
```

10 Managing Threads

```
[29] "mortDefaultSmall2008.csv" "mortDefaultSmall2009.csv" "
```

The location of the sample data directory is stored as an option in RevoScaleR, and you can access it with the following command:

```
rxGetOption("sampleDataDir")
```

Larger data sets containing the full airline, census, and mortgage default data sets are available for download at <http://packages.revolutionanalytics.com/datasets>. We make extensive use of the sample data sets both in this *User's Guide* and the companion documents *RevoScaleR Getting Started Guide* and *RevoScaleR Distributed Computing Guide*.

1.4 Managing Threads

A process is an instance of a software program, running under an operating system. In order to perform parallel computations, a process must provide blocks of executable code to the operating system that can be run in parallel.

There are two primary approaches to this problem. One approach is to use threads. Threads can be thought of as sub-programs that are still bound to the original process that creates them. They communicate with the creating process, while still being autonomous enough to allow the operating system to schedule their operation independently, thus allowing parallelization.

By way of contrast, forking a process (a technique only available on Unix-based systems) creates a complete copy of one process that differs only by a flag available within each of the resulting processes.

Unfortunately, these two methods of parallelization are not completely compatible unless steps are taken to ensure that they do not interfere with each other and cause the software to malfunction. Specifically, program failures can be caused by forking a Unix process while the objects used to control communication and synchronization between threads are in use.

In order to optimize performance, RevoScaleR is capable of establishing and maintaining a pool of threads. This way, there is no overhead in creating these threads to parallelize a computation. However, given the above issues, this functionality is, by default, disabled on Linux (this is not an issue on Windows, as Windows does not use fork; hence, the thread pool is always instantiated and ready for use on Windows).

If you know that you will not be forking your R process (such as spawning it using `nohup` or creating multiple R processes with the `multicore` package), or if you know exactly when you

might fork a process, you can turn the thread pool on and off. RevoScaleR provides an interface to activate the thread pool:

```
rxSetEnableThreadPool(TRUE)
```

Similarly, the thread pool may be disabled as follows:

```
rxSetEnableThreadPool(FALSE)
```

Enabling the thread pool will improve performance, but should not be done if there is any chance you will fork your Revolution R process. If you want to ensure that the thread pool is always enabled, you can add the above command to a *.First* function defined in either your own Rprofile startup file, or the system Rprofile.site file. For example, you can add the following lines after the closing right parenthesis of the existing Rprofile.site file:

```
.First <- function()
{
  .First.sys()
  invisible(rxSetEnableThreadPool(TRUE))
}
```

The *.First.sys* function is normally run after all other initialization is complete, including the evaluation of the *.First* function. We need the call to *rxSetEnableThreadPool* to occur after RevoScaleR is loaded, and that is done by *.First.sys*, so we call *.First.sys* first.

1.5 Generating Random Numbers

R has extensive facilities for managing random number generation, and these are all fully supported in RevoScaleR. In addition, RevoScaleR provides an interface to random number generators supplied by the Vector Statistical Library that is part of Intel’s Math Kernel Library. To use one of these generators, call the RevoScaleR function *rxRngNewStream*, specifying the desired generator (the default is a version of the Mersenne-Twister, MT-2203), the desired substream (if applicable), and a seed (if desired). See the help file for a complete list of the available generators and examples of their use.

The VSL random number generators are most useful in a distributed setting, where they allow parallel processes to generate uncorrelated random number streams. See the *RevoScaleR Distributed Computing Guide* for complete details.

1.6 Using RevoScaleR with Rscript

RevoScaleR depends on a number of packages that are specified as “default packages” in Revolution R Enterprise. To ensure these packages are loaded when using Rscript, you must include the flag *--default-packages=* (with nothing on the right-hand side), and not use the

12 Getting Help

flag `--vanilla` when invoking Rscript. In particular, the methods package is required for correct operation of RevoScaleR.

1.7 Getting Help

The RevoScaleR package includes comprehensive help for all of its functions and data sets, along with an overview topic describing the package as a whole. To view this overview topic, use the R `?` operator at the R prompt as follows:

```
?RevoScaleR
```

To obtain help on a RevoScaleR function, use the R `?` operator at the R prompt together with the function name. For example, the following command displays help about `rxLinMod`:

```
?rxLinMod
```

We will often refer to help topic pages as a function's help file, as in "see the `rxLinMod` help file for details." This always means to type the `?` operator followed by the function name.

Chapter 2.

Importing Data

The main function for importing data is *rxImport*. The *rxImport* function supports delimited text data, fixed-format text data, SAS data, SPSS data, and database data, provided you have a suitable ODBC driver for your database.

Smaller data sets can be imported into data frames in memory. In addition, the RevoScaleR package provides a data file format (.xdf) designed to be very efficient for reading arbitrary rows and columns. In this chapter, we explain how to import data in various formats, how to create a data frame in memory from a subset of an .xdf file, and how to write an .xdf file to a text file.

2.1 Data Compression in .xdf Files

Newer versions of the .xdf format support data compression via the *xdfCompressionLevel* argument to *rxImport* (and most other RevoScaleR functions that write .xdf files). You specify this as an integer in the range -1 to 9. The value -1 tells *rxImport* to use the current default compression value; the integers 1 through 9 specify increasing levels of compression—higher

14 Importing Delimited Text Data

numbers perform more compression, but take more time. The value 0 specifies no compression.

The .xdf format allows different blocks to have different compression levels (but not within a single call to *rxImport*). This can be useful, for example, when appending to an existing data set of unknown compression level—you can specify the compression for the new data without affecting the compression of the existing data.

You can specify a standard compression level for all future .xdf file writes by setting *xdfCompressionLevel* using *rxOptions*. For example, to specify a compression level of 3, use *rxOptions* as follows:

```
rxOptions(xdfCompressionLevel = 3)
```

In Revolution R Enterprise 7.0 and later, the default value of this option is 1.

If you have one or more existing .xdf files and would like to compress them, you can use the function *rxCompressXdf*. You can specify a single file or xdf data source, a character vector of files and data sources, or the path to a directory containing .xdf files. For example, to compress all the .xdf files in the C:\data directory, you would call *rxCompressXdf* as follows:

```
rxCompressXdf("C:/data", xdfCompressionLevel = 1, overwrite = TRUE)
```

2.2 Importing Delimited Text Data

Use the *rxImport* function to convert a text file to a data frame or .xdf file. In the simplest case, just provide the path to the input file. If your text format data set is delimited by commas or tabs, this is all that is required for simple data conversion. For example, the SampleData folder of the RevoScaleR package contains a file claims.txt containing information on car insurance claims. The following sequence of commands imports the data and returns a data frame, which we save as the object *claimsDF*:

```
readPath <- rxGetOption("sampleDataDir")
infile <- file.path(readPath, "claims.txt")
claimsDF <- rxImport(infile)
```

The *rxGetInfo* function then shows us a useful summary of what's contained in the data frame. If we include the argument *getVarInfo=TRUE*, this summary includes the names of the variables and their types:

```
rxGetInfo(claimsDF, getVarInfo = TRUE)

Data frame: claimsDF
Number of observations: 128
Number of variables: 6
Number of blocks: 1
Variable information:
```

```

Var 1: RowNum, Type: integer, Low/High: (1, 128)
Var 2: age, Type: character
Var 3: car.age, Type: character
Var 4: type, Type: character
Var 5: cost, Type: numeric, Storage: float32, Low/High: (11.0000, 850.0000)
Var 6: number, Type: numeric, Storage: float32, Low/High: (0.0000, 434.0000)

```

If we want just the names of the variables in the data file, we can use the `names` function:

```

names(claimsDF)

[1] "RowNum" "age" "car.age" "type" "cost" "number"

```

If instead of importing the data into a data frame in memory, we want to store the data in an .xdf file, we just specify the path as the `outFile`. By default, the new data set is created in our current working directory:

```

claimsDS <- rxImport(inData = inFile, outFile = "claims.xdf")

```

An .xdf file is created, and instead of returning a data frame, the `rxImport` function returns a data source object. This is a small R object that contains information about a data source, in this case the name and path of the .xdf file it represents. This data source object can be used as the input data in most RevoScaleR functions. For example:

```

rxGetInfo(claimsDS, getVarInfo = TRUE)
names(claimsDS)

```

2.2.1 Specifying Delimiters

If your text data is not separated by commas or tabs, you must specify the delimiter using the `columnDelimiters` argument. (This is not actually an argument to `rxImport`, but to the underlying `RxTextData` data source object.) In normal usage, this argument is a single character, such as `columnDelimiters="\t"` for tab-delimited data or `columnDelimiters=","` for comma-delimited data. However, each column may be delimited by a different character; all the delimiters must be concatenated together into a single character string. For example, if you have one column delimited by a comma, a second by a plus sign, and a third by a new line, you would use the argument `columnDelimiters=" ,+ \n"`.

Tip: While `rxImport` can automatically detect comma and tab delimiters, doing so requires extra processing. For fastest processing, use the `columnDelimiters` argument.

2.2.2 Specifying a Missing Value String

If your text data file uses a string other than NA to identify missing values, you must use the `missingValueString` argument. Only one missing value string is allowed per file. We used this briefly in Chapter 1 when we imported the AirlineDemoSmall data:

```

inFile <- file.path(rxGetOption("sampleDataDir"), "AirlineDemoSmall.csv")
airData <- rxImport(inData = inFile, outFile="airExample.xdf",
  stringsAsFactors = TRUE, missingValueString = "M",

```

16 Importing Fixed-Format Data

```
rowsPerRead = 200000, overwrite = TRUE)
```

2.3 Importing Fixed-Format Data

Fixed-format data is text data in which each variable occupies a fixed-width column in the input data file. The format is completely specified by the width of each column and each column's data type. You can import fixed-format data using the `rxImport` function.

Fixed-format data may be accompanied by a *schema file* with an `.sts` extension describing the width and type of each column. For complete details on creating a schema file, see page 93 of the Stat/Transfer PDF Manual (<http://www.stattransfer.com/stman10.pdf>). If you have a schema file, you can create the input data source very simply by specifying the schema file name as the input data file. For example, the RevoScaleR SampleData directory contains a fixed-format version of the claims data as the file `claims.dat` and a schema file named `claims.sts`. To import the data using this schema file, we use `RxImport` as follows:

```
inFile <- file.path(readPath, "claims.sts")
claimsFF <- rxImport(inData = inFile, outFile = "claimsFF.xdf")
```

You can get information about the file using `rxGetInfo` as usual:

```
rxGetInfo(claimsFF, getVarInfo=TRUE)
```

This gives the following output:

```
File name: C:\YourOutputPath\claims.xdf
Number of observations: 128
Number of variables: 6
Number of blocks: 1
Compression type: zlib
Variable information:
Var 1: RowNum, Type: numeric, Storage: float32, Low/High: (1.0000, 128.0000)
Var 2: age, Type: character
Var 3: car.age, Type: character
Var 4: type, Type: character
Var 5: cost, Type: numeric, Storage: float32, Low/High: (11.0000, 850.0000)
Var 6: number, Type: numeric, Storage: float32, Low/High: (0.0000, 434.0000)
```

To read all string data as factors, set the `stringsAsFactors` argument to `TRUE` in your call to `rxImport`

```
claimsFF2 <- rxImport(inFile, outFile = "claimsFF2.xdf", stringsAsFactors=TRUE)
rxGetInfo(claimsFF2, getVarInfo=TRUE)
```

If you have fixed-format data without a schema file, you can specify the start, width, and type information for each variable using the `colInfo` argument to the `RxTextData` data source constructor, and then read in the data using `rxImport`:

```
inFileNS <- file.path(readPath, "claims.dat")
```



```

outFileNS <- "claimsNS.xdf"
colInfo=list("rownum" = list(start = 1, width = 3, type = "integer"),
             "age" = list(start = 4, width = 5, type = "factor"),
             "car.age" = list(start = 9, width = 3, type = "factor"),
             "type" = list(start = 12, width = 1, type = "factor"),
             "cost" = list(start = 13, width = 6, type = "numeric"),
             "number" = list(start = 19, width = 3, type = "numeric"))
claimsNS <- rxImport(inFileNS, outFile = outFileNS, colInfo = colInfo)
rxGetInfo(claimsNS, getVarInfo=TRUE)

```

If you have a schema file, you can still use the `colInfo` argument to specify type information or factor level information. However, in this case, all the variables will be read according to the schema file, and then the `colInfo` data specifications will be applied. This means that you cannot use your `colInfo` list to omit variables, as you can when there is no schema file.

Fixed-width character data is treated as a special type by RevoScaleR for efficiency purposes. You can use this same type for character data in delimited data by specifying a `colInfo` argument with a width argument for the character column. (Typically, you will need to find the longest string in the column and specify a width sufficient to include it.)

2.4 Importing SAS Data

The `rxImport` function can also be used to read data from SAS files having a `.sas7bdat` or `.sd7` extension. You do not need to have SAS installed on your computer.

The RevoScaleR SampleData directory contains a SAS version of the claims data as `claims.sas7bdat`. We can read it into `.xdf` format most simply as follows:

```

inFileSAS <- file.path(rxGetOption("sampleDataDir"), "claims.sas7bdat")
xdfFileSAS <- "claimsSAS.xdf"
claimsSAS <- rxImport(inData = inFileSAS, outFile = xdfFileSAS)
rxGetInfo(claimsSAS, getVarInfo=TRUE)

```

This gives the following output:

```

File name: C:\YourOutputPath\claimsSAS.xdf
Number of observations: 128
Number of variables: 6
Number of blocks: 1
Compression type: zlib
Variable information:
Var 1: RowNum, Type: character
Var 2: age, Type: character
Var 3: car_age, Type: character
Var 4: type, Type: character
Var 5: cost, Type: numeric, Low/High: (11.0000, 850.0000)
Var 6: number, Type: numeric, Low/High: (0.0000, 434.0000)

```

Sometimes, SAS data files on Windows come in two pieces, a `.sas7bdat` file containing the data and a `.sas7bcat` file containing value label information. You can read both the data and the value label information by specifying the `.sas7bdat` file with the `inData` argument and the

18 Importing SPSS Data

.sas7bcat file with the *formatFile* argument. To do so, in the following code replace *myfile* with your SAS file name:

```
myData <- rxImport(inData = "myfile.sas7bdat",  
  outFile = "myfile.xdf",  
  formatFile = "myfile.sas7bcat")
```

2.5 Importing SPSS Data

The *rxImport* function can also be used to read data from SPSS files.

The RevoScaleR SampleData directory contains an SPSS version of the claims data as *claims.sav*. We can read it into .xdf format as follows:

```
inFileSpss <- file.path(rxGetOption("sampleDataDir"), "claims.sav")  
xdfFileSpss <- "claimsSpss.xdf"  
claimsSpss <- rxImport(inData = inFileSpss, outFile = xdfFileSpss)  
rxGetInfo(claimsSpss, getVarInfo=TRUE)
```

This gives the following output:

```
File name: C:\YourOutputPath\claimsSpss.xdf  
Number of observations: 128  
Number of variables: 6  
Number of blocks: 1  
Compression type: zlib  
Variable information:  
Var 1: RowNum, Type: character  
Var 2: age, Type: character  
Var 3: car_age, Type: character  
Var 4: type, Type: character  
Var 5: cost, Type: numeric, Low/High: (11.0000, 850.0000)  
Var 6: number, Type: numeric, Low/High: (0.0000, 434.0000)
```

Variables in SPSS data sets often contain value labels with important information about the data. SPSS variables with value labels are typically most usefully imported into R as categorical “factor” data; that is, there is a one-to-one mapping between the values in the data set (e.g., 1, 2, 3) and the labels that apply to them (e.g. TOO LITTLE, ABOUT RIGHT, and TOO MUCH). However, SPSS allows for value labels where this is not the case. For example, a variable with values from 1 to 99 might have value labels of “NOT HOME” for 97, “DIDN’T KNOW” for 98, and “NOT APPLICABLE” for 99. If this variable were converted to a factor in R using the value labels as the factor labels, all of the values from 1 to 96 would be set to missing because they would have no corresponding factor level. Essentially all of the actual data would be thrown away. To prevent this, use the flag *useValueLabels=FALSE*.

2.6 Specifying Variable Data Types

The *rxImport* function supports three arguments for specifying variable data types: *stringsAsFactors*, *colClasses*, and *colInfo*. For example, consider storing character data.

Often data stored in text files as string data actually represents categorical or *factor* data, which can be more compactly represented as a set of integers denoting the distinct *levels* of the factor. This is common enough that users frequently want to transform *all* string data to factors. This can be done using the *stringsAsFactors* argument:

```
infile <- file.path(rxGetOption("sampleDataDir"), "claims.sts")
rxImport(infile, outFile = "claimsSAF.xdf", stringsAsFactors = TRUE)
rxGetInfo("claimsSAF.xdf", getVarInfo = TRUE)

File name: C:\YourOutputPath\claimsSAF.xdf
Number of observations: 128
Number of variables: 6
Number of blocks: 1
Variable information:
Var 1: RowNum, Type: integer, Low/High: (1, 128)
Var 2: age
      8 factor levels: 17-20 21-24 25-29 30-34 35-39 40-49 50-59 60+
Var 3: car.age
      4 factor levels: 0-3 4-7 8-9 10+
Var 4: type
      4 factor levels: A B C D
Var 5: cost, Type: numeric, Storage: float32, Low/High: (11.0000, 850.0000)
Var 6: number, Type: numeric, Storage: float32, Low/High: (0.0000, 434.0000)
```

You can also specify data types for individual variables using the *colClasses* argument, and even more specific instructions for converting each variable using the *colInfo* argument. Here we use the *colClasses* argument to specify that the variable *number* in the claims data be stored as an integer:

```
outfileColClass <- "claimsCCNum.xdf"
rxImport(infile, outFile = outfileColClass, colClasses=c(number = "integer"))
rxGetInfo("claimsCCNum.xdf", getVarInfo = TRUE)

File name: C:\YourOutputPath\claimsCCNum.xdf
Number of observations: 128
Number of variables: 6
Number of blocks: 1
Compression type: zlib
Variable information:
Var 1: RowNum, Type: integer, Low/High: (1, 128)
Var 2: age, Type: character
Var 3: car.age, Type: character
Var 4: type, Type: character
Var 5: cost, Type: numeric, Storage: float32, Low/High: (11.0000, 850.0000)
Var 6: number, Type: integer, Low/High: (0, 434)
```

We can use the *colInfo* argument to specify the levels of the *car.age* column. This is particularly useful when reading data in chunks, to assure that factors levels are in the desired order.

```
outfileCAOrdered <- "claimsCAOrdered.xdf"
colInfoList <- list("car.age"= list(type = "factor", levels = c("0-3",
  "4-7", "8-9", "10+")))
rxImport(infile, outFile = outfileCAOrdered, colInfo = colInfoList)
rxGetInfo("claimsCAOrdered.xdf", getVarInfo = TRUE)
```

20 Specifying Variable Data Types

This gives the following output:

```
File name: C:\YourOutputPath\claimsCAOrdered.xdf
Number of observations: 128
Number of variables: 6
Number of blocks: 1
Compression type: zlib
Variable information:
Var 1: RowNum, Type: integer, Low/High: (1, 128)
Var 2: age, Type: character
Var 3: car.age
      4 factor levels: 0-3 4-7 8-9 10+
Var 4: type, Type: character
Var 5: cost, Type: numeric, Storage: float32, Low/High: (11.0000, 850.0000)
Var 6: number, Type: numeric, Storage: float32, Low/High: (0.0000, 434.0000)
```

These various methods of providing column information can be combined as follows:

```
outfileCAOrdered2 <- "claimsCAOrdered2.xdf"
colInfoList <- list("car.age" = list(type = "factor", levels = c("0-3",
  "4-7", "8-9", "10+")))
claimsOrdered <- rxImport(infile, outfileCAOrdered2,
  colClasses = c(number = "integer"),
  colInfo = colInfoList, stringsAsFactors = TRUE)
rxGetInfo(claimsOrdered, getVarInfo = TRUE)
```

This produces the following output:

```
File name: C:\YourOutputPath\claimsCAOrdered2.xdf
Number of observations: 128
Number of variables: 6
Number of blocks: 1
Compression type: zlib
Variable information:
Var 1: RowNum, Type: integer, Low/High: (1, 128)
Var 2: age
      8 factor levels: 17-20 21-24 25-29 30-34 35-39 40-49 50-59 60+
Var 3: car.age
      4 factor levels: 0-3 4-7 8-9 10+
Var 4: type
      4 factor levels: A B C D
Var 5: cost, Type: numeric, Storage: float32, Low/High: (11.0000, 850.0000)
Var 6: number, Type: integer, Low/High: (0, 434)
```

In general, variable specifications provided by the `colInfo` argument are used in preference to `colClasses`, and those in `colClasses` are used in preference to the `stringsAsFactors` argument.

Also note that the .xdf data format supports a wider variety of data types than R, allowing for efficient storage. For example, by default floating point variables are stored as 32-bit floats in .xdf files. When they are read into R for processing, they are converted to doubles (64-bit floats).

2.7 Specifying Additional Variable Information

If you need to modify the name of a variable or the names of the factor levels, or add a description of a variable, you can do this using the `colInfo` argument. For example, the claims data includes a variable `type` specifying the type of car, but the levels A, B, C, and D give us no particular information. If we knew what the types signified, perhaps “Subcompact”, “Compact”, “Mid-size”, and “Full-size”, we could relabel the levels as follows:

```
inFileAddVars <- file.path(rxGetOption("sampleDataDir"), "claims.txt")
outfileTypeRelabeled <- "claimsTypeRelabeled.xdf"
colInfoList <- list("type" = list(type = "factor", levels = c("A",
  "B", "C", "D"), newLevels=c("Subcompact", "Compact", "Mid-size",
  "Full-size"), description="Body Type"))
claimsNew <- rxImport(inFileAddVars, outFile = outfileTypeRelabeled,
  colInfo = colInfoList)
rxGetInfo(claimsNew, getVarInfo = TRUE)
```

This produces the following output:

```
File name: C:\YourOutputPath\claimsTypeRelabeled.xdf
Number of observations: 128
Number of variables: 6
Number of blocks: 1
Compression type: zlib
Variable information:
Var 1: RowNum, Type: integer, Low/High: (1, 128)
Var 2: age, Type: character
Var 3: car.age, Type: character
Var 4: type, Body Type
      4 factor levels: Subcompact Compact Mid-size Full-size
Var 5: cost, Type: numeric, Storage: float32, Low/High: (11.0000, 850.0000)
Var 6: number, Type: numeric, Storage: float32, Low/High: (0.0000, 434.0000)
```

To specify `newLevels`, you must also specify `levels`, and it is important to note that the `newLevels` argument can only be used to rename levels—it cannot be used to fully recode the factor. That is, the number of `levels` and number of `newLevels` must be the same.

2.8 Appending to an Existing File

If you have observations on the same variables in multiple input files, you can use the `append` argument to `rxImport` to combine them into one file. For example, we could append another copy of the claims text data set in a second block:

```
inFile <- file.path(rxGetOption("sampleDataDir"), "claims.txt")
colInfoList <- list("car.age" = list(type = "factor", levels = c("0-3",
  "4-7", "8-9", "10+")))
outfileCAOrdered2 <- "claimsCAOrdered2.xdf"

claimsAppend <- rxImport(inFile, outFile = outfileCAOrdered2,
  colClasses = c(number = "integer"),
  colInfo = colInfoList, stringsAsFactors = TRUE, append = "rows")
rxGetInfo(claimsAppend, getVarInfo=TRUE)
```

22 Transforming Data on Import

```
File name: C:\YourOutputPath\claimsCAOrdered2.xdf
Number of observations: 256
Number of variables: 6
Number of blocks: 2
Compression type: zlib
Variable information:
Var 1: RowNum, Type: integer, Low/High: (1, 128)
Var 2: age
      8 factor levels: 17-20 21-24 25-29 30-34 35-39 40-49 50-59 60+
Var 3: car.age
      4 factor levels: 0-3 4-7 8-9 10+
Var 4: type
      4 factor levels: A B C D
Var 5: cost, Type: numeric, Storage: float32, Low/High: (11.0000, 850.0000)
Var 6: number, Type: integer, Low/High: (0, 434)
```

2.9 Transforming Data on Import

You can use the *transforms* argument to *rxImport* to create new variables or modify existing variables when you initially read the data into .xdf format. For example, we could create a new variable, *logcost*, by taking the log of the existing cost variable as follows:

```
infile <- file.path(rxGetOption("sampleDataDir"), "claims.txt")
outfile <- "claimsXform.xdf"
claimsDS <- rxImport(infile, outFile = outfile,
  transforms=list(logcost=log(cost)))
rxGetInfo(claimsDS, getVarInfo=TRUE)
```

This gives the following output, showing the new variable:

```
File name: C:\YourOutputPath\claimsXform.xdf
Number of observations: 128
Number of variables: 7
Number of blocks: 1
Compression type: zlib
Variable information:
Var 1: RowNum, Type: integer, Low/High: (1, 128)
Var 2: age, Type: character
Var 3: car.age, Type: character
Var 4: type, Type: character
Var 5: cost, Type: numeric, Storage: float32, Low/High: (11.0000, 850.0000)
Var 6: number, Type: numeric, Storage: float32, Low/High: (0.0000, 434.0000)
Var 7: logcost, Type: numeric, Low/High: (2.3979, 6.7452)
```

2.10 Converting Dates Stored As Character Strings

The .xdf format can store dates using the standard R *Date* class. When importing data from other data formats that support dates such as SAS or SPSS, the *rxImport* function will convert dates data automatically. However, some data sets even in those formats include dates as character string data. You can store such data more efficiently by converting it to *Date* data using the *transforms* argument. For example, suppose you have a character variable *TransactionDate* with a representative date of the form “14 Sep 2010”. You can convert this to a *Date* variable using the following *transforms* argument:

```
transforms=list(TransactionDate=as.Date(TransactionDate, format="%d %b %Y"))
```

(The format argument is a character string that may contain conversion specifications, as in the example shown. These conversion specifications are described in the *strptime* help file.)

2.11 Splitting Data Files

RevoScaleR makes it possible to analyze huge data sets easily and efficiently, and for most purposes the most efficient computations are done on a single .xdf file. However, there are many circumstances when you will want to work with only a portion of your data. For example, you may want to distribute your data over the nodes of a cluster; in such a case, RevoScaleR's analysis functions will process each node's data separately, combining all the results for the final return value. You might also want to split your data into training and test data so that you can fit a model using the training data and validate it using the test data.

Use the function *rxSplit* to split your data. For example, to split variables of interest in the large 2000 U.S. Census data into five files for distribution on a five node cluster, you could use *rxSplit* as follows (change the location of the *bigDataDir* to the location of your downloaded file):

```
rxSetComputeContext("local")
bigDataDir <- "C:/Revolution/Data"
bigCensusData <- file.path(bigDataDir, "Census5PCT2000.xdf")

splitFiles <- rxSplit(bigCensusData, numOutFiles = 5, splitBy = "blocks",
  varsToKeep = c("age", "inccarn", "incwelfr", "educrec", "metro", "perwt"))
names(splitFiles)
```

By default, *rxSplit* simply appends a number in the sequence from 1 to *numOutFiles* to the base file name to create the new file names, and in this case the resulting file names, for example, "Census5PCT20001.xdf", are a bit confusing. You can exercise greater control over the output file names by using the *outFilesBase* and *outFilesSuffixes* arguments. With *outFilesBase*, you can specify either a single character string to be used for all files or a character vector the same length as the desired number of files. The latter option is useful, for example, if you would like to create four files with the same file name, but different paths:

```
nodePaths <- paste("compute", 10:13, sep="")
baseNames <- file.path("C:", nodePaths, "DistCensusData")
splitFiles2 <- rxSplit(bigCensusData, splitBy = "blocks",
  outFilesBase = baseNames,
  varsToKeep = c("age", "inccarn", "incwelfr", "educrec", "metro", "perwt"))
names(splitFiles2)
```

This creates the four directories C:/compute10, etc., and creates a file named "DistCensusData.xdf" in each directory. You will want to do something like this when using distributed data with the standard RevoScaleR analysis functions such as *rxLinMod* and *rxLogit*.

24 Reading Data from an .xdf File into a Data Frame

You can supply the `outFilesSuffixes` arguments to exercise greater control over what is appended to the end of each file. Returning to our first example, we can add a hyphen between our base file name and the sequence 1 to 5 using `outFilesSuffixes` as follows:

```
splitFiles3 <- rxSplit(bigCensusData, splitBy = "blocks",
  outFileSuffixes=paste("-", 1:5, sep=""),
  varsToKeep = c("age", "inccarn", "incwelfr", "educrec", "metro", "perwt"))
names(splitFiles3)
```

The `splitBy` argument specifies whether to split your data file row-by-row or block-by-block. The default is `splitBy="rows"`, which distributes data from each block into different files. The examples above use the faster split by blocks instead. The `splitBy` argument is ignored if you also specify the `splitByFactor` argument as a character string representing a valid factor variable. In this case, one file is created per level of the factor variable.

You can use the `splitByFactor` argument and a transforms argument to easily create test and training data sets from an .xdf file. Note that this will take longer than the previous examples because each block of data is being processed:

```
splitFiles4 <- rxSplit(inData = bigCensusData,
  outFilesBase="censusData",
  splitByFactor="testSplitVar",
  varsToKeep = c("age", "inccarn", "incwelfr", "educrec", "metro", "perwt"),
  transforms=list(testSplitVar = factor(
    sample(0:1, size=rxNumRows, replace=TRUE, prob=c(.10, .9)),
    levels=0:1, labels = c("Test", "Train"))))
names(splitFiles4)
rxSummary(~age, data = splitFiles4[[1]], reportProgress = 0)
rxSummary(~age, data = splitFiles4[[2]], reportProgress = 0)
```

This takes approximately 10% of the data as a test data set, with the remainder going into the training data.

If your .xdf file is relatively small, you may want to set `outFilesBase = ""` so that a list of data frames is returned instead of having files created. You can also use `rxSplit` to split data frames; see the `rxSplit` help file for details.

2.12 Reading Data from an .xdf File into a Data Frame

It is often convenient to store a large amount of data in an .xdf file and then read a subset of columns and rows of the data into a data frame in memory for analysis. The `rxDataStep` function makes this easy. For example, let's consider taking subsamples from the sample data set `CensusWorkers.xdf`. Using a `rowSelection` expression and list of `varsToKeep`, we can extract the `age`, `perwt`, and `sex` variables for individuals over the age of 40 living in Washington State:

```
inFile <- file.path(rxGetOption("sampleDataDir"), "CensusWorkers.xdf")
myCensusDF <- rxDataStep(inData=inFile,
```



```
rowSelection = state == "Washington" & age > 40,
varsToKeep = c("age", "perwt", "sex"))
```

When subsampling rows, we need to be aware that the `rowSelection` is processed on each chunk of data after it is read in. Consider an example where we want to extract every 10th row of the data set. For each chunk we will create a sequence starting with the start row number in that chunk (provided by the internal variable, `.rxStartRow`) with a length equal to the number of rows in the data chunk. We will determine that number of rows by using the length of the of one of the variables that has been read from the data set, `age`. We will keep only the rows where the remainder after dividing the row number by 10 is 0:

```
myCensusSample <- rxDataStep(inData=inFile,
  rowSelection= (seq(from=.rxStartRow,length.out=.rxNumRows) %% 10) == 0 )
```

We can also create transformed variables while we are reading in the data. For example, create an 10-year `ageGroup` variable, starting with 20 (the minimum age in the data set):

```
myCensusDF2 <- rxDataStep(inData=inFile,
  varsToKeep = c("age", "perwt", "sex"),
  transforms=list(ageGroup = cut(age, seq(from=20, to=70, by=10))))
```

2.13 Data Sources

A *data source* in RevoScaleR can be thought of as a small R object representing a data set. We have already encountered them as the return objects of `rxImport` and `rxDataStep`. The data itself may be on disk, but the data source is an in-memory object that allows us to treat data from disparate sources in a consistent manner within RevoScaleR. Behind the scenes, `rxImport` often creates data sources “on the fly” to facilitate data import. You can create your own data sources to give you finer control over how data is imported. To create data sources, use the constructors listed in the following table:

Source Data	Data Source Constructor
Text (fixed-format or delimited)	<code>RxTextData</code>
SAS	<code>RxSasData</code>
SPSS	<code>RxSpssData</code>
Database (must have	<code>RxOdbcData</code>

26 Data Sources

appropriate ODBC driver installed)	
Teradata Database	RxTeradata
.xdf data files	RxXdfData

For simple data import, you do not need to create a data source—you can simply specify a file of the appropriate type and RevoScaleR will read it using the default settings. However, if you need to provide additional options specific to that data source type, you will want to refer to the data source’s documentation. (For more information on accessing databases via ODBC, see the *RevoScaleR ODBC Import Guide*.)

A number of standard R methods for looking at data have been provided for RevoScaleR data sources. We’ve already seen the use of *names* to view the variable names and *head* to view the first few rows. Returning to the *claimsDS* data source created in the section Importing Delimited Text Data, we can obtain the dimensions of our claims data using the *dim* function:

```
readPath <- rxGetOption("sampleDataDir")
infile <- file.path(readPath, "claims.txt")
claimsDS <- rxImport(inData = infile, outFile = "claims.xdf",
  overwrite = TRUE)
dim(claimsDS)

[1] "RowNum" "age" "car.age" "type" "cost" "number"
```

Similarly, an alternative method of obtaining the variable names is to use the *colnames* or *dimnames* functions, but notice that for .xdf file data sources, *dimnames* returns only column names; row names are not provided, because .xdf files do not contain row names:

```
colnames(claimsDS)

[1] "RowNum" "age" "car.age" "type" "cost" "number"

dimnames(claimsDS)

[[1]]
NULL

[[2]]
[1] "RowNum" "age" "car.age" "type" "cost" "number"
```

You can obtain the number of variables using the *length* function:

```
length(claimsDS)

[1] 6
```

If you want to see just the top few rows of data, you can use the *head* function:

```
head(claimsDS)
```

	RowNum	age	car.age	type	cost	number	logcost
1	1	17-20	0-3	A	289	8	5.666427
2	2	17-20	4-7	A	282	8	5.641907
3	3	17-20	8-9	A	133	4	4.890349
4	4	17-20	10+	A	160	1	5.075174
5	5	17-20	0-3	B	372	10	5.918894
6	6	17-20	4-7	B	249	28	5.517453

You can view the *last* rows of a data source using the *tail* function:

```
tail(claimsDS)
```

	RowNum	age	car.age	type	cost	number
123	123	60+	8-9	C	227	20
124	124	60+	10+	C	119	6
125	125	60+	0-3	D	385	62
126	126	60+	4-7	D	324	22
127	127	60+	8-9	D	192	6
128	128	60+	10+	D	123	6

2.14 Using Data from the Hadoop Distributed File System

If you are using RevoScaleR on a Linux system that is part of a Hadoop cluster, you can store and access text or xdf data files on your system's native file system or the Hadoop Distributed File System (HDFS). By default, data is expected to be found on the native file system. If all your data is on HDFS, you can use *rxSetFileSystem* to specify this as a global option. You need to know the host name of your Hadoop name node, and the port number used for your Hadoop connection. You can then specify your global option as follows:

```
rxSetFileSystem(RxHdfsFileSystem(hostName = "hostname", port = ####))
```

If only some files are on HDFS, you can use *RxTextData* or *RxXdfData* data sources to specify these as files on HDFS. For example,

```
hdfsFS <- RxHdfsFileSystem( hostName = "hostname", port = ####)
txtSource <- RxTextData("/test/HdfsData/AirlineCSV/CSVs/1987.csv",
  fileSystem=hdfsFS)
xdfSource <- RxXdfData("/test/HdfsData/AirlineData1987.xdf",
  fileSystem=hdfsFS)
```

The *RxHdfsFileSystem* function is used to create a file system object for the HDFS file system; the *RxNativeFileSystem* function does the same thing for the native file system.

You can currently use data files on HDFS as input files only if you are using a local compute context; output must be written to the native file system.

As a more complete example, consider again the mortgage default example from the *Getting Started Guide*. If our mortgage data is available on the HDFS file system, we can run the example as follows:

```
rxSetFileSystem(RxHdfsFileSystem(hostName = "hostname", port = ####))
mortDS <- RxXdfData("/share/SampleData/mortDefaultSmall.xdf")
```

28 Using Data from the Hadoop Distributed File System

```
rxGetInfo(mortDS, numRows = 5)
rxSummary(~., data = mortDS, blocksPerRead = 2)
logitObj <- rxLogit(default~F(year) + creditScore + yearsEmploy + ccDebt,
  data = mortDS, blocksPerRead = 2, reportProgress = 1)
summary(logitObj)
```

2.14.1 Note on Using RevoScaleR with rhdfs

If you are using both RevoScaleR and the RHadoop connector package rhdfs, you need to ensure that the two do not interfere with each other. The rhdfs package depends upon the rJava package, which will prevent access to HDFS by RevoScaleR if it is called before RevoScaleR makes its connection to HDFS. To prevent this interaction, use the function `rxHdfsConnect` to establish a connection between RevoScaleR and HDFS. An install-time option on Linux can be used to trigger such a call from the Rprofile.site startup file. If the install-time option is not chosen, you can add it later by setting the `REVOHADOOPHOST` and `REVOHADOOPPORT` environment variables with the host name of your Hadoop name node and the name node's port number, respectively. You can also call `rxHdfsConnect` interactively within a session, provided you have not yet attempted any other rJava or rhdfs commands. For example, the following call will fix a connection between the Hadoop host sandbox-01 and RevoScaleR; if you make a subsequent call to rhdfs, RevoScaleR can continue to use the previously established connection. Note, however, that once rhdfs (or any other rJava call) has been invoked, you cannot change the host or port you use to connect to RevoScaleR:

```
rxHdfsConnect(hostName = "sandbox-01", port = 8020)
```

Chapter 3.

Transforming and Subsetting Data

RevoScaleR provides a full set of functions for modifying, transforming, and subsetting data stored in memory in a data frame or in an .xdf file on disk:

- *rxDataStep* is used to subset rows and/or variables and to create new variables by transforming existing variables. It also allows for easy conversion between data in data frames and .xdf files.
- *rxSetVarInfo* is used to change variable information such as the name or description of a variable in an .xdf file or data frame. *rxSetInfo* is used to add or change a data set description.
- *rxFactors* is used to create or modify factors (categorical variables) based on existing variables.
- *rxSort* is used to sort a data set by one or more key variables.
- *rxMerge* is used to merge two data sets by one or more key variables.

3.1 Creating a Subset of Rows and Columns

A common use of *rxDataStep* is to create a new data set with a subset of rows and variables. The following simple example uses a data frame as the input data set. The call to *rxDataStep*

30 Creating a Subset of Rows and Columns

uses the `rowSelection` argument to select only the rows where the variable `y` is greater than `.5`, and the `varsToKeep` argument to keep only the variables `y` and `z`. The `rowSelection` argument is an R expression that evaluates to `TRUE` if the observation should be kept. The `varsToKeep` argument contains a list of variable names to read in from the original data set. Because no `outFile` is specified, a data frame is returned:

```
# Create a data frame
set.seed(59)
myData <- data.frame(
  x = rnorm(100),
  y = runif(100),
  z = rep(1:20, times = 5))

# Subset observations and variables
myNewData <- rxDataStep( inData = myData,
  rowSelection = y > .5,
  varsToKeep = c("y", "z"))

# Get information about the new data frame
rxGetInfo(data = myNewData, getVarInfo = TRUE)

Data frame: myNewData
Number of observations: 52
Number of variables: 2
Variable information:
Var 1: y, Type: numeric, Low/High: (0.5516, 0.9941)
Var 2: z, Type: integer, Low/High: (1, 20)
```

Subsetting is particularly useful if your original data set contains millions of rows or hundreds of thousands of variables. As a smaller example, the `CensusWorkers.xdf` sample data file has six variables and 351,121 rows. To create a subset containing only workers who have worked less than 30 weeks in the year and five variables, we can again use `rxDataStep` with the `rowSelection` and `varsToKeep` arguments. Since the resulting data set will clearly fit in memory, we omit the `outFile` argument and assign the result of the data step, then use `rxGetInfo` to see our results as usual:

```
readPath <- rxGetOption("sampleDataDir")
censusWorkers <- file.path(readPath, "CensusWorkers.xdf")
partWorkers <- rxDataStep(inData = censusWorkers, rowSelection = wkswork1 < 30,
  varsToKeep = c("age", "sex", "wkswork1", "incwage", "perwt"))
rxGetInfo(partWorkers, getVarInfo = TRUE)
```

The result is a data set with 14,317 rows:

```
Data frame: partWorkers
Number of observations: 14317
Number of variables: 5
Variable information:
Var 1: age, Age
  Type: integer, Low/High: (20, 65)
Var 2: sex, Sex
  2 factor levels: Male Female
Var 3: wkswork1, Weeks worked last year
```

```

      Type: integer, Low/High: (21, 29)
Var 4: incwage, Wage and salary income
      Type: integer, Low/High: (0, 354000)
Var 5: perwt, Type: integer, Low/High: (2, 163)

```

Alternatively, and in this case more easily, you can use `varsToDrop` to prevent variables from being read in from the original data set. This time we'll specify an `outFile`; use the `overwrite` argument to allow the output file to be replaced.

```

partWorkersDS <- rxDataStep(inData = censusWorkers,
  outFile = "partWorkers.xdf",
  rowSelection = wkswork1 < 30,
  varsToDrop = c("state"), overwrite = TRUE)

```

As noted above, if you omit the `outFile` argument to `rxDataStep`, then the results will be returned in a data frame in memory. This is true whether or not the input data is a data frame or an .xdf file (assuming the resulting data is small enough to reside in memory). If an `outFile` is specified, a data source object representing the new .xdf file is returned, which can be used in subsequent RevoScaleR function calls.

```
rxGetVarInfo( partWorkersDS )
```

3.2 Transforming Data with `rxDataStep`

A crucial step in many analyses is transforming the data into the form best suited for the chosen analysis. For example, variations in scale between variables can sometimes make it convenient to take a log or a power of the original variable before fitting a linear model. In RevoScaleR, transforming data is an important issue because we are typically reading and processing data from disk, so we want to be as efficient as possible, minimizing the passes through the data. To provide maximum flexibility, RevoScaleR allows you to perform data transformations in virtually all of its functions, from the `rxImport` function discussed in the previous chapter, to the `rxDataStep` function that is the primary focus here, to the analysis functions `rxSummary`, `rxLinMod`, `rxLogit`, `rxGlm`, `rxCrossTabs`, `rxCube`, `rxCovCor`, and `rxKmeans` discussed in subsequent chapters. In all these cases, the basic procedures for transforming the data are the same.

The heart of the RevoScaleR data step is a list of *transforms*, each of which specifies an R expression to be evaluated and typically is an assignment either creating a new variable or modifying an existing variable from the original data set.

3.2.1 Creating and Transforming Variables

To create or modify variables, we use the `transforms` argument to `rxDataStep`. The `transforms` argument is specified as a list of expressions. Any R expression that operates row-by-row (that is, the computed value of the new variable for an observation is only dependent

32 Transforming Data with rxDataStep

on values of other variables for that observation) can be used. Let's begin with a simple data frame containing a small sample of transaction data:

```
expData <- data.frame(  
  BuyDate = c("2011/10/1", "2011/10/1", "2011/10/1",  
    "2011/10/2", "2011/10/2", "2011/10/2", "2011/10/2",  
    "2011/10/3", "2011/10/4", "2011/10/4"),  
  Food = c( 32, 102, 34, 5, 0, 175, 15, 76, 23, 14),  
  Wine = c( 0, 212, 0, 0, 425, 22, 0, 12, 0, 56),  
  Garden = c( 0, 46, 0, 0, 0, 45, 223, 0, 0, 0),  
  House = c( 22, 72, 56, 3, 0, 0, 0, 37, 48, 23),  
  Sex = factor(c("F", "F", "M", "M", "M", "F", "F", "F", "M", "F")),  
  Age = c( 20, 51, 32, 16, 61, 42, 35, 99, 29, 55),  
  stringsAsFactors = FALSE)
```

Now we would like to perform a series of data transformations:

- Compute the total expenditures for each store visit
- Compute the average category expenditure for each store visit
- Set the variable *Age* to missing if the value is 99
- Create a new logical variable *UnderAge* is the purchaser is under 21
- Find the day of week the purchase was made using R's *as.POSIXlt* function, then convert it to a factor. [Note that when creating factors in a transform, you must specify the levels or you may get unpredictable results. Alternatively use *rxFactors*.]
- Create a factor variable for categories of expenditure levels
- Create a logical variable for expenditures of \$50 or more on either Food or Wine
- Remove the variable *BuyDate* from the data set

The following call to *rxDataStep* will accomplish all of the above, returning a new data frame with the transformed variables:

```
newExpData = rxDataStep( inData = expData,  
  transforms = list(  
    Total      = Food + Wine + Garden + House,  
    AveCat     = Total/4,  
    Age        = ifelse(Age == 99, NA, Age),  
    UnderAge   = Age < 21,  
    Day        = (as.POSIXlt(BuyDate))$wday,  
    Day        = factor(Day, levels = 0:6,  
      labels = c("Su", "M", "Tu", "W", "Th", "F", "Sa")),  
    SpendCat   = cut(Total, breaks=c(0, 75, 250, 10000),  
      labels=c("low", "medium", "high"), right=FALSE),  
    FoodWine   = ifelse( Food > 50, TRUE, FALSE),  
    FoodWine   = ifelse( Wine > 50, TRUE, FoodWine),  
    BuyDate    = NULL))  
newExpData
```

The new data frame shows all of the transformed data:

	Food	Wine	Garden	House	Sex	Age	Total	AveCat	UnderAge	Day	SpendCat	FoodWine
1	32	0	0	22	F	20	54	13.50	TRUE	Sa	low	FALSE
2	102	212	46	72	F	51	432	108.00	FALSE	Sa	high	TRUE

3	34	0	0	56	M	32	90	22.50	FALSE	Sa	medium	FALSE
4	5	0	0	3	M	16	8	2.00	TRUE	Su	low	FALSE
5	0	425	0	0	M	61	425	106.25	FALSE	Su	high	TRUE
6	175	22	45	0	F	42	242	60.50	FALSE	Su	medium	TRUE
7	15	0	223	0	F	35	238	59.50	FALSE	Su	medium	FALSE
8	76	12	0	37	F	NA	125	31.25	NA	M	medium	TRUE
9	23	0	0	48	M	29	71	17.75	FALSE	Tu	low	FALSE
10	14	56	0	23	F	55	93	23.25	FALSE	Tu	medium	TRUE

If we had a large data set of containing expenditure data in an .xdf file, we could use exactly the same transformation code; the only changes in the call to `rxDataStep` would be the `inData` and specifying an `outFile` for the newly created data set.

Sometimes it is useful to use computed values as part of a transformation. For example, you might want to impute missing values, replacing any missings with the variable mean. Let's look at a simple data frame example; again the same basic code could be used for a huge data set. First create a data set with missing values:

```
# Create a data frame with missing values
set.seed(59)
myData1 <- data.frame(x = rnorm(100), y = runif(100))

xmiss <- seq.int(from = 5, to = 100, by = 5)
ymiss <- seq.int(from = 2, to = 100, by = 5)
myData1$x[xmiss] <- NA
myData1$y[ymiss] <- NA
rxGetInfo(myData1, numRows = 5)
```

The call to `rxGetInfo` shows the following:

```
Data frame: myData1
Number of observations: 100
Number of variables: 2
Data (5 rows starting with row 1):
      x      y
1 -1.8621337 0.06206201
2  1.1398069          NA
3  0.3176267 0.84132161
4  1.3998593 0.26298559
5          NA 0.97069679
```

Now use `rxSummary` (discussed in detail in a later chapter) to compute summary statistics, putting the computed means into a named vector:

```
# Compute the summary statistics and extract
# the means in a named vector
sumStats <- rxResultsDF(rxSummary(~., myData1))
sumStats
meanVals <- sumStats$Mean
names(meanVals) <- row.names(sumStats)
```

The computed statistics are:

	Mean	StdDev	Min	Max	ValidObs	MissingObs
x	0.07431126	0.9350711	-1.94160646	1.9933814	80	20
y	0.54622241	0.3003457	0.04997869	0.9930338	80	20

34 Transforming Data with rxDataStep

Next we pass the computed means into a `rxDataStep` using the `transformObjects` argument:

```
# Use rxDataStep to replace missings with imputed mean values
myData2 <- rxDataStep(inData = myData1, transforms = list(
  x = ifelse(is.na(x), meanVals["x"], x),
  y = ifelse(is.na(y), meanVals["y"], y)),
  transformObjects = list(meanVals = meanVals))
rxGetInfo(myData2, numRows = 5)
```

The resulting data set information is:

```
Data frame: myData2
Number of observations: 100
Number of variables: 2
Data (5 rows starting with row 1):
      x      y
1 -1.86213372 0.06206201
2  1.13980693 0.54622241
3  0.31762673 0.84132161
4  1.39985928 0.26298559
5  0.07431126 0.97069679
```

3.2.2 Subsetting and Transforming Variables

Returning to our earlier example with `CensusWorkers`, we will now combine subsetting and transformations in one data step operation. Suppose we want to extract the same five variables as before from the `CensusWorkers` data set, but also add a factor variable based on the integer variable `age`. For example, to create our factor variable, we can use the following `transforms` argument:

```
transforms = list(ageFactor = cut(age, breaks=seq(from = 20, to = 70,
  by = 5), right = FALSE))
```

In doing a data step operation, `RevoScaleR` reads in a chunk of data read from the original data set, including only the variables indicated in `varsToKeep`, or omitting variables specified in `varsToDrop`. It then passes the variables needed for data transformations back to R for manipulation:

```
rxDataStep (inData = censusWorkers, outFile = "newCensusWorkers",
  varsToDrop = c("state"), transforms = list(
    ageFactor = cut(age, breaks=seq(from = 20, to = 70, by = 5),
      right = FALSE)))
```

The `rxGetInfo` function reveals the added variable:

```
rxGetInfo("newCensusWorkers", getVarInfo = TRUE)

File name: C:\YourOutputPath\newCensusWorkers.xdf
Number of observations: 351121
Number of variables: 6
Number of blocks: 6
Compression type: zlib
Variable information:
Var 1: age, Age
      Type: integer, Low/High: (20, 65)
Var 2: incwage, Wage and salary income
```

```

      Type: integer, Low/High: (0, 354000)
Var 3: perwt, Type: integer, Low/High: (2, 168)
Var 4: sex, Sex
      2 factor levels: Male Female
Var 5: wkswork1, Weeks worked last year
      Type: integer, Low/High: (21, 52)
Var 6: ageFactor
      10 factor levels: [20,25) [25,30) [30,35) [35,40) [40,45) [45,50)
[50,55) [55,60) [60,65) [65,70)

```

We can combine the *transforms* argument with the *transformObjects* argument to create new variables from objects in your global environment (or other environments in your current search path).

For example, suppose you would like to estimate a linear model using wage income as the dependent variable, and want to include state-level of per capita expenditure on education as one of the independent variables. We can define a named vector to contain this state-level data as follows:

```
educExp <- c(Connecticut=1795.57, Washington=1170.46, Indiana = 1289.66)
```

We can then use *rxDataStep* to add the per capita education expenditure as a new variable using the *transforms* argument, passing *educExp* to the *transformObjects* argument as a named list:

```

censusWorkers <- file.path(rxGetOption("sampleDataDir"), "CensusWorkers.xdf")
rxDataStep(inData = censusWorkers, outFile = "censusWorkersWithEduc",
  transforms = list(
    stateEducExpPC = educExp[match(state, names(educExp))] ),
  transformObjects= list(educExp=educExp))

```

The *rxGetInfo* function reveals the added variable:

```

rxGetInfo("censusWorkersWithEduc.xdf",getVarInfo=TRUE)

File name: C:\YourOutputPath\censusWorkersWithEduc.xdf
Number of observations: 351121
Number of variables: 7
Number of blocks: 6
Compression type: zlib
Variable information:
Var 1: age, Age
      Type: integer, Low/High: (20, 65)
Var 2: incwage, Wage and salary income
      Type: integer, Low/High: (0, 354000)
Var 3: perwt, Type: integer, Low/High: (2, 168)
Var 4: sex, Sex
      2 factor levels: Male Female
Var 5: wkswork1, Weeks worked last year
      Type: integer, Low/High: (21, 52)
Var 6: state
      3 factor levels: Connecticut Indiana Washington
Var 7: stateEducExpPC, Type: numeric, Low/High: (1170.4600, 1795.5700)

```

36 Using the Data Step to Create an .xdf File from a Data Frame

3.3 Using the Data Step to Create an .xdf File from a Data Frame

You can use all of the functionality provided by the `rxDataStep` function to create an .xdf file from a data frame for further use. For example, create a simple data frame:

```
set.seed(39)
myData <- data.frame(x1 = rnorm(10000), x2 = runif(10000))
```

Now create an .xdf file, using a row selection and creating a new variable. The `rowsPerRead` argument will specify how many rows of the original data frame to process at a time before writing out a block of the new .xdf file.

```
rxDataStep(inData = myData, outFile = "testFile.xdf",
  rowSelection = x2 > .1,
  transforms = list( x3 = x1 + x2 ),
  rowsPerRead = 5000 )
rxGetInfo("testFile.xdf")

rxGetInfo("testFile.xdf")
File name: C:\Users\...\testFile.xdf
Number of observations: 8970
Number of variables: 3
Number of blocks: 2
```

3.4 Converting .xdf Files to Text

If you need to share data with others not using .xdf data files, you can export your .xdf files to text format using the `rxDataStep` function. For example, we can write the claims.xdf file we created earlier to text format as follows:

```
claimsCsv <- RxTextData(file="claims.csv")
claimsXdf <- RxXdfData(file="claims.xdf")
rxDataStep(inData=claimsXdf,
  outFile=claimsCsv)
```

By default, the text file created is comma-delimited, but you can change this by specifying a different delimiter with the `delimiter` argument to the `RxTextData` function:

```
claimsTxt <- RxTextData(file="claims.txt", delimiter="\t")
rxDataStep(inData=claimsXdf, outFile=claimsTxt)
```

If you have a large number of variables, you can choose to write out only a subsample by using the `VarsToKeep` or `VarsToDrop` argument. Here we write out the claims data, omitting the `number` variable:

```
rxDataStep(inData=claimsXdf, outFile=claimsTxt, varsToDrop="number",
  overwrite=TRUE)
```

3.5 Re-Blocking an .xdf File

After a series of data import or row selection steps, you may find that you have an .xdf file with very uneven block sizes. This may make it difficult to efficiently perform computations by “chunk.” To find the sizes of the blocks in your .xdf file, use `rxGetInfo` with the `getBlockSizes` argument set to TRUE. For example, let’s look at the block sizes for the sample `CensusWorkers.xdf` file:

```
fileName <- file.path(rxGetOption("sampleDataDir"), "CensusWorkers.xdf")
rxGetInfo(fileName, getBlockSizes = TRUE)
```

The following information is provided:

```
File name: C:\Revolution\R-Enterprise-7.0\R-
3.0.2\library\RevoScaleR\SampleData\CensusWorkers.xdf
Number of observations: 351121
Number of variables: 6
Number of blocks: 6
Compression type: zlib
Rows per block: 95420 42503 1799 131234 34726 45439
```

We see that, in fact, the number of rows per block varies from a low of 1799 to a high of 131,234. To create a new file with more even-sized blocks, use the `rowsPerRead` argument in `rxDataStep`:

```
newFile <- "censusWorkersEvenBlocks.xdf"
rxDataStep(inData = fileName, outFile = newFile, rowsPerRead = 60000)
rxGetInfo(newFile, getBlockSizes = TRUE)
```

The new file has blocks sizes of 60,000 for all but the last slightly smaller block:

```
File name: C:\Users\...\censusWorkersEvenBlocks.xdf
Number of observations: 351121
Number of variables: 6
Number of blocks: 6
Compression type: zlib
Rows per block: 60000 60000 60000 60000 60000 51121
```

3.6 Modifying Variable Information

To change variable information (rather than the data values themselves), use the function `rxSetVarInfo`. For example, using the data file created above, we can change the names of two variables and add descriptions:

```
newVarInfo <- list(
  incwage = list(newName = "WageIncome"),
  state = list(newName = "State", description = "State of Residence"),
  stateEducExpPC = list(description = "State Per Capita Educ Exp"))
fileName <- "censusWorkersWithEduc.xdf"
rxSetVarInfo(varInfo = newVarInfo, data = fileName)
rxGetVarInfo(fileName)
```

38 Sorting Data

```
Var 1: age, Age
      Type: integer, Low/High: (20, 65)
Var 2: WageIncome, Wage and salary income
      Type: integer, Low/High: (0, 354000)
Var 3: perwt, Type: integer, Low/High: (2, 168)
Var 4: sex, Sex
      2 factor levels: Male Female
Var 5: wkswork1, Weeks worked last year
      Type: integer, Low/High: (21, 52)
Var 6: State, State of Residence
      3 factor levels: Connecticut Indiana Washington
Var 7: stateEducExpPC, State Per Capita Educ Exp
      Type: numeric, Low/High: (1170.4600, 1795.5700)
```

3.7 Sorting Data

Many analysis and plotting algorithms require as a first step that the data be sorted. Sorting a massive data set is both memory-intensive and time-consuming, but the `rxSort` function provides an efficient solution. The `rxSort` function allows you to sort by one or many keys. A *stable* sorting routine is used, so that, in the case of ties, remaining columns are left in the same order as they were in the original data set.

As a simple example, we can sort the census worker data by `age` and `incwage`. We will sort first by `age`, using the default increasing sort, and then by `incwage`, which we will sort in decreasing order:

```
censusWorkers <- file.path(rxGetOption("sampleDataDir"), "CensusWorkers.xdf")
outXDF <- "censusWorkersSorted.xdf"
rxSort(inData = censusWorkers, outFile = outXDF, sortByVars=c("age",
  "incwage"), decreasing=c(FALSE, TRUE))
```

The first few lines of the sorted file can be viewed as follows:

```
rxGetInfo(outXDF, numRows=10)

File name: C:\YourOutputPath\censusWorkersSorted.xdf
Number of observations: 351121
Number of variables: 6
Number of blocks: 6
Compression type: zlib
Data (10 rows starting with row 1):
  age incwage perwt sex wkswork1 state
1  20  336000    3  Male        40 Washington
2  20  336000   23  Male        46 Washington
3  20  336000   11  Male        52 Washington
4  20  314000   33  Male        52 Indiana
5  20  168000   13  Male        24 Indiana
6  20  163000   16  Male        26 Washington
7  20  144000   27 Female       24 Indiana
8  20   96000   21  Male        48 Washington
9  20   93000   24  Male        24 Indiana
10 20   90000    6  Male        52 Washington
```

If the sort keys contain missing values, you can use the `missingsLow` flag to specify whether they are sorted as low values (`missingsLow=TRUE`, the default) or high values (`missingsLow=FALSE`).

3.7.1 Removing Duplicates While Sorting

In many situations, you are sorting a large data set by a particular key, for example, `userID`, but are looking for a sorted list of unique `userIDs`. The `removeDupKeys` argument to `rxSort` allows you to remove the duplicate entries from a sorted list. This argument is supported only for `type="auto"` and `type="mergeSort"`; it is ignored for `type="varByVar"`. When you use `removeDupKeys=TRUE`, the first record containing a unique combination of the `sortByVars` is retained; subsequent matching records are omitted from the sorted results, but, if desired, a count of the matching records is maintained in a new `dupFreqVar` output column. For example, the following artificial data set simulates a small amount of transaction data, with a user name, a state, and a transaction amount. When we sort by the variables `users` and `state` and specify `removeDupKeys=TRUE`, the `transAmt` shown for duplicate entries is the transaction amount for the *first* transaction encountered:

```
set.seed(17)
users <- sample(c("Aiden", "Ella", "Jayden", "Ava", "Max", "Grace", "Riley",
                 "Lolita", "Liam", "Emma", "Ethan", "Elizabeth", "Jack",
                 "Genevieve", "Avery", "Aurora", "Dylan", "Isabella",
                 "Caleb", "Bella"), 100, replace=TRUE)
state <- sample(c("Washington", "California", "Texas", "North Carolina",
                 "New York", "Massachusetts"), 100, replace=TRUE)
transAmt <- round(runif(100)*100, digits=3)
df <- data.frame(users=users, state=state, transAmt=transAmt)

rxSort(df, sortByVars=c("users", "state"), removeDupKeys=TRUE,
       dupFreqVar = "DUP_COUNT")
```

```
Number of rows written to file: 66, Variable(s): users, state, transAmt,
DUP_COUNT, Total number of rows in file: 66
Time to sort data file: 0.100 seconds
```

	users	state	transAmt	DUP_COUNT
1	Aiden	New York	11.010	1
2	Aiden	North Carolina	73.307	1
3	Aiden	Texas	98.250	2
4	Aurora	California	8.787	1
5	Aurora	Massachusetts	55.187	1
6	Aurora	New York	91.648	1
7	Aurora	Texas	30.566	1
8	Aurora	Washington	27.374	1
9	Ava	California	63.641	2
10	Ava	Massachusetts	50.390	2
11	Ava	New York	58.950	2
12	Ava	North Carolina	51.748	1
13	Ava	Texas	52.674	1
14	Avery	California	35.035	4
15	Avery	Massachusetts	63.715	1
16	Avery	New York	93.430	1
17	Avery	North Carolina	90.648	2
18	Bella	California	9.410	2
19	Bella	Texas	32.684	1

40 Sorting Data

20	Bella	Washington	37.163	2
21	Caleb	Massachusetts	94.527	1
22	Caleb	North Carolina	69.227	2
23	Dylan	California	73.665	1
24	Dylan	North Carolina	98.384	1
25	Dylan	Texas	27.067	1
26	Dylan	Washington	36.697	3
27	Elizabeth	California	38.458	2
28	Elizabeth	New York	80.161	3
29	Elizabeth	North Carolina	81.140	2
30	Ella	California	11.644	1
31	Ella	North Carolina	72.289	1
32	Ella	Washington	26.723	2
33	Emma	Massachusetts	28.502	1
34	Emma	North Carolina	63.067	1
35	Ethan	Massachusetts	31.480	1
36	Ethan	New York	95.639	1
37	Ethan	North Carolina	6.561	1
38	Ethan	Texas	29.963	1
39	Ethan	Washington	2.278	2
40	Genevieve	Massachusetts	90.783	1
41	Grace	New York	18.232	1
42	Grace	North Carolina	52.546	2
43	Grace	Washington	91.084	1
44	Isabella	New York	4.115	1
45	Isabella	North Carolina	53.891	2
46	Isabella	Texas	38.661	2
47	Jack	California	40.905	1
48	Jack	Massachusetts	42.091	2
49	Jack	New York	32.145	2
50	Jack	North Carolina	93.898	3
51	Jack	Texas	57.578	2
52	Jayden	Massachusetts	83.949	1
53	Jayden	North Carolina	67.769	1
54	Jayden	Washington	4.360	1
55	Liam	California	3.300	1
56	Liam	Massachusetts	87.585	1
57	Liam	New York	96.599	1
58	Liam	North Carolina	32.997	1
59	Lolita	California	18.102	1
60	Lolita	Washington	74.571	2
61	Max	Massachusetts	1.394	2
62	Max	New York	5.061	2
63	Max	North Carolina	63.804	2
64	Max	Texas	25.536	2
65	Max	Washington	43.947	2
66	Riley	New York	85.421	2

Removing duplicates can be a useful way to reduce the size of a data set without losing information of interest. For example, consider an analysis of using data from the sample *AirlineDemoSmall* xdf file. It has 600,000 observations and contains the variables *DayOfWeek*, *CRSDepTime*, and *ArrDelay*. We can create a smaller data set reducing the number of observations, and adding a variable that contains the frequency of the duplicated observation.

```
sampleDataDir <- rxGetOption("sampleDataDir")
airDemo <- file.path(sampleDataDir, "AirlineDemoSmall.xdf")
airDedup <- file.path(tempdir(), "rxAirDedup.xdf")
rxSort(inData = airDemo, outFile = airDedup,
       sortByVars = c("DayOfWeek", "CRSDepTime", "ArrDelay"),
```



```
removeDupKeys = TRUE, dupFreqVar = "FreqWt")
rxGetInfo(airDedup)
```

The new data file contains about 1/3 of the observations and one additional variable:

```
File name: C:\YourTempDir\rxAirDedup.xdf
Number of observations: 232451
Number of variables: 4
Number of blocks: 2
Compression type: zlib
```

By using the frequency weights argument, we can use many of the RevoScaleR analysis functions on this smaller data set and get same results as we would using the full data set. For example, a linear model for Arrival Delay can be specified as follows, using the *fweights* argument:

```
linModObj <- rxLinMod(ArrDelay~CRSDepTime + DayOfWeek, data = airDedup,
  fweights = "FreqWt")
summary(linModObj)
```

Call:

```
rxLinMod(formula = ArrDelay ~ CRSDepTime + DayOfWeek, data = airDedup,
  fweights = "FreqWt")
```

Linear Regression Results for: ArrDelay ~ CRSDepTime + DayOfWeek

File name: C:\YourTempDir\rxAirDedup.xdf

Frequency weights: FreqWt

Dependent variable(s): ArrDelay

Total independent variables: 9 (Including number dropped: 1)

Sum of weights of valid observations: 582628

Number of missing observations: 3503

Coefficients: (1 not defined because of singularities)

	Estimate	Std. Error	t value	Pr(> t)	
(Intercept)	-3.19458	0.20413	-15.650	2.22e-16	***
CRSDepTime	0.97862	0.01126	86.948	2.22e-16	***
DayOfWeek=Monday	2.08100	0.18602	11.187	2.22e-16	***
DayOfWeek=Tuesday	1.34015	0.19881	6.741	1.58e-11	***
DayOfWeek=Wednesday	0.15155	0.19679	0.770	0.441	
DayOfWeek=Thursday	-1.32301	0.19518	-6.778	1.22e-11	***
DayOfWeek=Friday	4.80042	0.19452	24.679	2.22e-16	***
DayOfWeek=Saturday	2.18965	0.19229	11.387	2.22e-16	***
DayOfWeek=Sunday	Dropped	Dropped	Dropped	Dropped	

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

Residual standard error: 40.39 on 582620 degrees of freedom

Multiple R-squared: 0.01465

Adjusted R-squared: 0.01464

F-statistic: 1238 on 7 and 582620 DF, p-value: < 2.2e-16

Condition number: 10.6542

Using the full data set, we get the following results:

```
linModObjBig <- rxLinMod(ArrDelay~CRSDepTime + DayOfWeek, data = airDemo)
```

42 Merging Data

```
summary(linModObjBig)
```

Call:

```
rxLinMod(formula = ArrDelay ~ CRSDepTime + DayOfWeek, data = airDemo)
```

Linear Regression Results for: ArrDelay ~ CRSDepTime + DayOfWeek

File name: C:\YourSampleDir\AirlineDemoSmall.xdf

Dependent variable(s): ArrDelay

Total independent variables: 9 (Including number dropped: 1)

Number of valid observations: 582628

Number of missing observations: 17372

Coefficients: (1 not defined because of singularities)

	Estimate	Std. Error	t value	Pr(> t)	
(Intercept)	-3.19458	0.20413	-15.650	2.22e-16	***
CRSDepTime	0.97862	0.01126	86.948	2.22e-16	***
DayOfWeek=Monday	2.08100	0.18602	11.187	2.22e-16	***
DayOfWeek=Tuesday	1.34015	0.19881	6.741	1.58e-11	***
DayOfWeek=Wednesday	0.15155	0.19679	0.770	0.441	
DayOfWeek=Thursday	-1.32301	0.19518	-6.778	1.22e-11	***
DayOfWeek=Friday	4.80042	0.19452	24.679	2.22e-16	***
DayOfWeek=Saturday	2.18965	0.19229	11.387	2.22e-16	***
DayOfWeek=Sunday	Dropped	Dropped	Dropped	Dropped	

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

Residual standard error: 40.39 on 582620 degrees of freedom

Multiple R-squared: 0.01465

Adjusted R-squared: 0.01464

F-statistic: 1238 on 7 and 582620 DF, p-value: < 2.2e-16

Condition number: 10.6542

3.7.2 The rxQuantile Function and the Five-Number Summary

Sorting data is, in the general case, a prerequisite to finding exact quantiles, including medians. However, it is possible to compute approximate quantiles by counting binned data then computing a linear interpolation of the empirical cdf. If the data are integers, or can be converted to integers by exact multiplication, and integral bins are used, the computation is exact. The RevoScaleR function rxQuantile does this computation, and by default returns an approximate five-number summary:

```
readPath <- rxGetOption("sampleDataDir")
AirlinePath <- file.path(readPath, "AirlineDemoSmall.xdf")
rxQuantile("ArrDelay", AirlinePath)
```

```
Rows Processed: 600000
  0%  25%  50%  75% 100%
-86  -9   0  16 1490
```

3.8 Merging Data

Merging allows you to combine the information from two data sets into a third data set that can be used for subsequent analysis. One example is merging account information such as account number and billing address with transaction data such as account number and

purchase details to create invoices. In this case, the two files are merged on the common information, that is, the account number.

In RevoScaleR, you merge .xdf files and/or data frames with the rxMerge function. This function supports a number of types of merge that are best illustrated by example. The available types are as follows:

- Inner
- Outer: left, right, and full
- One-to-One
- Union

We describe each of these types in the following sections.

3.8.1 Inner Merge

In the default inner merge type, one or more merge key columns is specified, and only those observations for which the specified key columns match exactly are combined to create new observations in the merged data set.

Suppose we have the following data from a dentist's office:

AccountNo	Billee	Patient
0538	Rich C	1
0538	Rich C	2
0538	Rich C	3
0763	Tom D	1
1534	Kath P	1

We can create a data frame with this information:

```
acct <- c(0538, 0538, 0538, 0763, 1534)
billee <- c("Rich C", "Rich C", "Rich C", "Tom D", "Kath P")
patient <- c(1, 2, 3, 1, 1)
acctDF<- data.frame( acct=acct, billee= billee, patient=patient)
```

Suppose further we have the following information about procedures performed:

AccountNo	Patient	Procedure
0538	3	OffVisit
0538	2	AdultPro
0538	2	OffVisit
0538	3	2SurfCom
0763	1	OffVisit
0763	1	AdultPro
0763	2	OffVisit

This data is put into another data frame:

```
acct <- c(0538, 0538, 0538, 0538, 0763, 0763, 0763)
```

44 Merging Data

```
patient <- c(3, 2, 2, 3, 1, 1, 2)
type <- c("OffVisit", "AdultPro", "OffVisit", "2SurfCom", "OffVisit", "AdultPro",
"OffVisit")
procedureDF <- data.frame(acct=acct, patient=patient, type=type)
```

Then we use `rxMerge` to create an inner merge matching on the columns `acct` and `patient`:

```
rxMerge(inData1 = acctDF, inData2 = procedureDF, type = "inner",
matchVars=c("acct", "patient"))
```

	acct	billee	patient	type
1	538	Rich C	2	AdultPro
2	538	Rich C	2	OffVisit
3	538	Rich C	3	OffVisit
4	538	Rich C	3	2SurfCom
5	763	Tom D	1	OffVisit
6	763	Tom D	1	AdultPro

Because the patient 1 in account 538 and patient 1 in account 1534 had no visits, they are omitted from the merged file. Similarly, patient 2 in account 763 had a visit, but does not have any information in the accounts file, so it too is omitted from the merged data set. Also, note that the two input data files are automatically sorted on the merge keys before merging.

3.8.2 Outer Merge

There are three types of outer merge: left, right, and full. In a left outer merge, all the lines from the first file are present in the merged file, either matched with lines from the second file that match on the key columns, or if no match, filled out with missing values. A right outer merge is similar, except all the lines from the second file are present, either matched with matching lines from the first file or filled out with missings. A full outer merge includes all lines in both files, either matched or filled out with missings. We can use the same dentist data to illustrate the various types of outer merge:

```
rxMerge(inData1 = acctDF, inData2 = procedureDF, type = "left",
matchVars=c("acct", "patient"))
```

	acct	billee	patient	type
1	538	Rich C	1	<NA>
2	538	Rich C	2	AdultPro
3	538	Rich C	2	OffVisit
4	538	Rich C	3	OffVisit
5	538	Rich C	3	2SurfCom
6	763	Tom D	1	OffVisit
7	763	Tom D	1	AdultPro
8	1534	Kath P	1	<NA>

```
rxMerge(inData1 = acctDF, inData2 = procedureDF, type = "right",
matchVars=c("acct", "patient"))
```

	acct	billee	patient	type
1	538	Rich C	2	AdultPro
2	538	Rich C	2	OffVisit

3	538	Rich C	3	OffVisit
4	538	Rich C	3	2SurfCom
5	763	Tom D	1	OffVisit
6	763	Tom D	1	AdultPro
7	763	<NA>	2	OffVisit

```
rxMerge(inData1 = acctDF, inData2 = procedureDF, type = "full",
        matchVars=c("acct", "patient"))
```

	acct	billee	patient	type
1	538	Rich C	1	<NA>
2	538	Rich C	2	AdultPro
3	538	Rich C	2	OffVisit
4	538	Rich C	3	OffVisit
5	538	Rich C	3	2SurfCom
6	763	Tom D	1	OffVisit
7	763	Tom D	1	AdultPro
8	763	<NA>	2	OffVisit
9	1534	Kath P	1	<NA>

3.8.3 One-to-one Merge

In the one-to-one merge type, the first observation in the first data set is paired with the first observation in the second data set to create the first observation in the merged data set, the second observation is paired with the second observation to create the second observation in the merged data set, and so on. The data sets must have the same number of rows. It is equivalent to using `append="cols"` in a data step.

For example, suppose our first data set contains three observations as follows:

1	a	x
2	b	y
3	c	z

Create a data frame with this data:

```
myData1 <- data.frame( x1 = 1:3, y1 = c("a", "b", "c"), z1 = c("x", "y", "z"))
```

Suppose our second data set contains three different variables:

101	d	u
102	e	v
103	f	w

Create a data frame with this data:

```
myData2 <- data.frame( x2 = 101:103, y2 = c("d", "e", "f"),
                      z2 = c("u", "v", "w"))
```

A one-to-one merge of these two data sets combines the columns from the two data sets into one data set:

46 Merging Data

```
rxMerge(inData1 = myData1, inData2 = myData2, type = "oneToOne")
```

	x1	y1	z1	x2	y2	z2
1	1	a	x	101	d	u
2	2	b	y	102	e	v
3	3	c	z	103	f	w

3.8.4 Union Merge

A union merge is simply the concatenation of two files with the same set of variables. It is equivalent to using `append="rows"` in a data step.

Using the example from one-to-one merge, we rename the variables in the second data frame to be the same as in the first:

```
names(myData2) <- c("x1", "x2", "x3")
```

Then use a union merge:

```
rxMerge(inData1 = myData1, inData2 = myData2, type = "union")
```

	x1	y1	z1
1	1	a	x
2	2	b	y
3	3	c	z
4	101	d	u
5	102	e	v
6	103	f	w

3.8.5 Using rxMerge with .xdf files

You can use `rxMerge` with a combination of .xdf files and/or data frames. For example, you specify the two the paths for two input .xdf files as the `inData1` and `inData2` arguments, and the path to an output file as the `outFile` argument. As a simple example, we can stack two copies of the claims data using the union merge type as follows:

```
claimsXdf <- file.path(rxGetOption("sampleDataDir"), "claims.xdf")  
  
rxMerge(inData1 = claimsXdf, inData2 = claimsXdf, outFile = "claimsTwice.xdf",  
        type = "union")
```

A new .xdf file is creating containing double the number of rows of the original claims file.

You can also merge an .xdf file and data frame into a new .xdf file. For example, suppose that you would like to add a variable on state expenditure on education into each observation in the `censusWorkers` sample .xdf file. First, take a quick look at the state variable in the .xdf file:

```
censusWorkers <- file.path(rxGetOption("sampleDataDir"), "CensusWorkers.xdf")  
rxGetVarInfo(censusWorkers, varsToKeep = "state")
```

```
Var 1: state
      3 factor levels: Connecticut Indiana Washington
```

We can create a data frame with per capita educational expenditures for the same three states. (Note that because R alphabetizes factor levels by default, the factor levels in the data frame will be in the same order as those in the .xdf file).

```
educExp <- data.frame(state=c("Connecticut", "Washington", "Indiana"),
  EducExp = c(1795.57, 1170.46, 1289.66 ))
```

Now use `rxMerge`, matching by the variable `state`:

```
rxMerge(inData1 = censusWorkers, inData2 = educExp,
  outFile="censusWorkersEd.xdf", matchVars = "state", overwrite=TRUE)
```

The new .xdf file has an additional variable, `EducExp`:

```
rxGetVarInfo("censusWorkersEd.xdf")

Var 1: age, Age
      Type: integer, Low/High: (20, 65)
Var 2: incwage, Wage and salary income
      Type: integer, Low/High: (0, 354000)
Var 3: perwt, Type: integer, Low/High: (2, 168)
Var 4: sex, Sex
      2 factor levels: Male Female
Var 5: wkswork1, Weeks worked last year
      Type: integer, Low/High: (21, 52)
Var 6: state
      3 factor levels: Connecticut Indiana Washington
Var 7: EducExp, Type: numeric, Low/High: (1170.4600, 1795.5700)
```

3.9 Recoding Factors

Factors are variables that represent categories. An example is “sex”, which has the categories “Male” and “Female”. There are two parts to a factor variable:

1. A vector of integer indexes with values in the range of 1:K, where K is the number of categories. For example, “sex” has two categories with indexes 1 and 2. The length of the vector corresponds to the number of observations.
2. A vector of K character strings that are used when the factor is displayed. In R, these are normally printed without quote marks, to indicate that the variable is a factor instead of a character string.

Recoding a factor means changing from one set of indexes to another. For example, if the levels of “sex” are currently arranged in the order “Male”, “Female” and you want to change that to “Female”, “Male”, you need to change the index for every observation.

48 Recoding Factors

You can use the `rxFactors` function to recode factors in RevoScaleR. For example, suppose we have some test scores for a set of male and female subjects. We can generate such data randomly as follows:

```
set.seed(100)
sex <- factor(sample(c("M", "F"), size = 10, replace = TRUE),
              levels = c("M", "F"))
DF <- data.frame(sex = sex, score = rnorm(10))
```

If we look at just the sex variable, we see the levels M or F for each observation:

```
DF[["sex"]]

[1] M M F M M M F M F M
Levels: M F
```

To recode this factor so that “Female” is the first level and “Male” the second, we can use `rxFactors` as follows:

```
newDF <- rxFactors(inData = DF, overwrite = TRUE,
                   factorInfo = list(Gender = list(newLevels = c(Female = "F",
                                                                Male = "M"),
                                                                varName = "sex")))

```

Looking at the new Gender variable, we see how the levels have changed:

```
newDF$Gender

[1] Male   Male   Female Male   Male   Male   Female Male   Female Male
Levels: Female Male
```

By default, RevoScaleR codes factor levels in the order in which they are encountered in the input file(s). This could lead you to have a State variable ordered as “Maine”, “Vermont”, “New Hampshire”, “Massachusetts”, “Connecticut”, etc. Usually, you would prefer to have the levels of such a variable sorted in alphabetical order. You can do this with `rxFactors` using the `sortLevels` flag. It is most useful to specify this flag as part of the `factorInfo` list for each variable, although if you have a large number of factors and want most of them to be sorted, you can also set the flag to `TRUE` globally and then specify `sortLevels=FALSE` for those variables you want to order in a different way.

When using the `sortLevels` flag, it is useful to keep in mind that it is the *levels* that are being sorted, not the data itself, and that the levels are always character data. If you are using the individual values of a continuous variable as factor levels, you may be surprised by the sorted order of the levels: for example, the levels 1, 3, 20 are sorted as “1”, “20”, “3”.

Another common use of factor recoding is in analyzing survey data gathered using Likert items with five or seven level responses. For example, suppose a customer satisfaction survey offered the following seven-level responses to each of four questions:

1. Completely satisfied
2. Mostly satisfied
3. Somewhat satisfied
4. Neither satisfied nor dissatisfied
5. Somewhat dissatisfied
6. Mostly dissatisfied
7. Completely dissatisfied

In analyzing this data, the survey analyst may recode the factors to focus on those who were largely satisfied (combining levels 1 and 2), largely dissatisfied (combining levels 6 and 7) and somewhere in-between (combining levels 3, 4, and 5). The CustomerSurvey.xdf file in the SampleData directory contains 25 responses to each of the four survey questions. We can read it into a data frame as follows:

```
surveyDF <- rxDataStep(inData =
  file.path(rxGetOption("sampleDataDir"), "CustomerSurvey.xdf"))
```

To recode each question as desired, we can use rxFactors as follows:

```
sl <- levels(surveyDF[[1]])
quarterList <- list(newLevels = list(
  "Largely Satisfied" = sl[1:2],
  "Neither Satisfied Nor Dissatisfied" = sl[3:5],
  "Largely Dissatisfied" = sl[6:7]))
surveyDF <- rxFactors(inData = surveyDF,
  factorInfo <- list(
    Q1 = quarterList,
    Q2 = quarterList,
    Q3 = quarterList,
    Q4 = quarterList))
```

Looking at just Q1, we see the recoded factor:

```
surveyDF[["Q1"]]

[1] Neither Satisfied Nor Dissatisfied Largely Satisfied
[3] Neither Satisfied Nor Dissatisfied Largely Satisfied
[5] Neither Satisfied Nor Dissatisfied Neither Satisfied Nor Dissatisfied
[7] Largely Dissatisfied Neither Satisfied Nor Dissatisfied
[9] Neither Satisfied Nor Dissatisfied Largely Satisfied
[11] Neither Satisfied Nor Dissatisfied Largely Dissatisfied
[13] Largely Satisfied Neither Satisfied Nor Dissatisfied
[15] Largely Dissatisfied Neither Satisfied Nor Dissatisfied
[17] Largely Satisfied Neither Satisfied Nor Dissatisfied
[19] Neither Satisfied Nor Dissatisfied Neither Satisfied Nor Dissatisfied
[21] Neither Satisfied Nor Dissatisfied Neither Satisfied Nor Dissatisfied
[23] Neither Satisfied Nor Dissatisfied Largely Dissatisfied
[25] Neither Satisfied Nor Dissatisfied
3 Levels: Largely Satisfied ... Largely Dissatisfied
```

3.9.1 Recoding Factors to Ensure Variable Compatibility

One important use of factor recoding in RevoScaleR is to ensure that the factor variables in two files are compatible, that is, have the same levels with the same coding. This use comes up in a variety of contexts, including prediction, merging, and distributed computing. For example, suppose you are creating a logistic regression model of whether a given airline flight will be late, and are using the first fifteen years of the airline data as a training set. You then want to test the model on the remaining years of the airline data. You need to ensure that the two files, the training set and the test set, have compatible factor variables. You can generally do this easily using `rxGetInfo` (with `getVarInfo=TRUE`) together with `rxFactors`. Use `rxGetInfo` to find all the levels in all the files, then use `rxFactors` to recode each file so that every factor variable contains all the levels found in any of the files.

The `rxMerge` function automatically checks for factor variable compatibility and recodes on the fly if necessary.

Chapter 4.

Models in RevoScaleR

Specifying a model with RevoScaleR is similar to specifying a model with the standard R statistical modeling functions, but there are some significant differences. Understanding these differences can help you make better use of RevoScaleR. The purpose of this chapter is to explain these differences at a high level so that when we begin fitting models in the next chapter, the terminology does not seem too foreign.

4.1 External Memory Algorithms

The first thing to know about RevoScaleR's statistical algorithms is that they are *external memory* algorithms that work on one chunk of data at time. All of RevoScaleR's algorithms share a basic underlying structure:

- An *initialization* step, in which data structures are created and given initial values, a data source is identified and opened for reading, and any other initial conditions are satisfied.
- A *process data* step, in which a chunk of data is read, some calculations are performed, and the result is passed back to the master process.
- An *update results* step, in which the results of the process data step are merged with previous results.
- A *finalize results* step, in which any final processing is performed and a result is returned.

An important consideration in using external memory algorithms is deciding how big a *chunk* of data is. You want to use a chunk size that's as large as possible while still allowing the process data step to complete without swapping. All of the RevoScaleR modeling functions allow you to specify a *blocksPerRead* argument. To make effective use of this, you need to know how many blocks your data file contains and how large the data file is. The number of blocks can be obtained using the *rxGetInfo* function.

4.2 Formulas in RevoScaleR

RevoScaleR uses a variant of the Wilkinson-Rogers formula notation (Wilkinson & Rogers, 1973) that is similar to, but not exactly the same as, that used in the standard R modeling functions. The response, or dependent, variables are separated from the predictor, or independent, variables by a tilde (~). In most of the modeling functions, multiple response variables are permitted, specified using the *cbind* function to combine them.

Independent variables (*predictors*) are separated by plus signs (+). Interaction terms can be created by joining two or more variables with a colon (:). Interactions between two categorical variables add one coefficient to the model for every combination of levels of the two variables. For example, if we have the categorical variables sex with two levels and education with four levels, the interaction of sex and education will add eight coefficients to the fitted model. Interactions between a continuous variable and a categorical variable add one coefficient to the model representing the continuous variable for each level of the categorical variable. (However, in RevoScaleR, such interactions cannot be used with the *rxCube* or *rxCrossTabs* functions.) The interaction of two continuous variables is the same as multiplying the two variables.

An asterisk (*) between two variables adds all subsets of interactions to the model. Thus, *sex*education* is equivalent to *sex + education + sex:education*.

The special function syntax $F(x)$ can be used to have RevoScaleR treat a numeric variable *x* as a categorical variable (factor) for the purposes of the analysis. You can include additional arguments *low* and *high* to specify the minimum and maximum values to be included in the factor variable; RevoScaleR creates a bin for each integer value from the low to high values. You can use the logical flag *exclude* to specify whether values outside that range are omitted from the model (*exclude=TRUE*) or included as a separate level (*exclude=FALSE*).

Similarly, the special function syntax $N(x)$ can be used to have RevoScaleR treat *x* as a continuous numeric variable. (This syntax is provided for completeness, but is not recommended. In general, if you want to recover numeric data from a factor, you will want to do so from the *levels* of the factor. We give several examples of this in Chapter 6, 7, and 11.)

In RevoScaleR, formulas in which the first independent variable is categorical may be handled specially, as *cubes*. See the following section for more details.

4.3 Letting the Data Speak For Itself

Classical statistics is largely concerned with fitting predictive models to limited observations. Data analysts and statisticians use exploratory techniques to visualize the data and then make informed guesses as to the appropriate form for a predictive model. Large data analysis, on the other hand, provides the opportunity to let the data speak for itself—with millions of observations in hand, we don't have to guess about the form of relationships between variables: they become clear. We will give several examples of this in the chapters that follow, but here's a general idea of how to find the relationship between two numeric variables x and y . First, bin the x values. Then, for each bin, plot the mean of the y values contained in that bin. As the bins become narrower and narrower, the resulting plot becomes a plot of $E(y|x)$ for each value of x .

4.4 Cubes and Cube Regression

In RevoScaleR, a *cube* is a type of multi-dimensional array. It may have any number of dimensions, thus making it a hypercube, and each dimension consists of categorical data. The most common operation on a cube is simple cross-tabulation, computing the cell counts for every combination of levels within the cube; this is done using the `rxCube` function. But cubes can also be passed as the first independent variable in a linear or logistic regression, in which case the regression can be computed in a special way. Because the cube consists of categorical data, one part of the moment matrix is diagonal. This makes it possible to compute the regression using a partitioned inverse method, which may be faster and may use less memory than the usual regression method. When a linear or logistic regression is fitted with the argument `cube=TRUE` (and a categorical variable as the first predictor), the partitioned inverse method is used and the model is fitted without an intercept term. Because the first term in the regression is categorical, it is equivalent to a complete set of dummy variables and thus is collinear with a constant term. By dropping the intercept term, the collinearity is resolved and a coefficient can be computed for each level of the categorical predictor. The R-squared, however, is computed as if an intercept were included.

Chapter 5.

Data Summaries

In previous chapters, we've used the `rxGetVarInfo` and `rxSummary` functions to view summary information about data files and the data itself. In this chapter we explore the use of these functions in more detail.

5.1 Using Variable Information

The `rxGetVarInfo` function returns a list containing information about each variable in a .xdf file. For numeric data, this information includes Low/High values. These Low/High values do not necessarily indicate the minimum and maximum of a numeric or integer variable. Rather, they indicate the values RevoScaleR uses to establish the lowest and highest factor levels when treating the variable as a factor. For practical purposes, this is what you will want—for example, if you want to create histograms or hexbin plots, you want to treat the numerical data as categorical, with levels corresponding to the plotting bins. It is often convenient to cut off the highest and lowest data points, and the Low/High information allows you to do this.

For example, consider again the census data we manipulated in Chapter 3.

```
readPath <- rxGetOption("sampleDataDir")
censusWorkers <- file.path(readPath, "CensusWorkers.xdf")
censusWorkerInfo <- rxGetVarInfo(censusWorkers)
names(censusWorkerInfo)
```

This returns the following:

```
[1] "age"      "incwage" "perwt"   "sex"      "wkswork1" "state"
```

We then drill down to examine the age component:

```
names(censusWorkerInfo$age)
```

```
[1] "description" "varType"     "storage"     "low"         "high"
```

We obtain the High value as follows:

```
censusWorkerInfo$age$high
```

```
[1] 65
```

Similarly, the Low value is obtained as follows:

```
censusWorkerInfo$age$low
```

```
[1] 20
```

If we are interested in workers between the ages of 30 and 50, we could create a copy of the data file in the working directory, set the High/Low fields as follows and then treat age as a factor in our subsequent analysis:

```
outputDir <- rxGetOption("outDataPath")
tempCensusWorkers <- file.path(outputDir, "tempCensusWorkers.xdf")
file.copy(from = censusWorkers, to = tempCensusWorkers)
censusWorkerInfo$age$low <- 35
censusWorkerInfo$age$high <- 50
rxSetVarInfo(censusWorkerInfo, tempCensusWorkers)
rxSummary(~F(age), data = tempCensusWorkers)
```

The resulting output is restricted to the 35 to 50 age range:

```
Call:
rxSummary(formula = ~F(age), data = tempCensusWorkers)

Summary Statistics Results for: ~F(age)
File name:
  C:\YourOutDir\tempCensusWorkers.xdf
Number of valid observations: 351121
```

```
Category Counts for F_age
Number of categories: 16
Number of valid observations: 158309
Number of missing observations: 192812
```

```
F_age Counts
35      9743
36      9888
37      9860
38     10211
39     10378
40     10756
41     10503
42     10511
43     10296
44     10122
```

56 Formulas for rxSummary

```
45      10074
46      9703
47      9527
48      9093
49      8776
50      8868
```

To reset the low and high values, we use the same process with the original values:

```
censusWorkerInfo$age$low <- 20
censusWorkerInfo$age$high <- 65
rxSetVarInfo(censusWorkerInfo, "tempCensusWorkers.xdf")
```

5.2 Formulas for rxSummary

The `rxSummary` function provides descriptive statistics using a *formula* argument similar to that used in R's modeling functions. With one exception, the formula given to `rxSummary` may not contain a response variable, so that `rxSummary` is generally given a formula of the form “~ predictors.” For example, returning to the `CensusWorkers` data file we used in Chapter 3, we can obtain a data summary of that file as follows:

```
readPath <- rxGetOption("sampleDataDir")
censusWorkers <- file.path(readPath, "CensusWorkers.xdf")
rxSummary(~ age + incwage + perwt + sex + wkswork1, data = censusWorkers)
```

This gives the following output:

```
Call:
rxSummary(formula = ~age + incwage + perwt + sex + wkswork1,
  data = censusWorkers)

Summary Statistics Results for: ~age + incwage + perwt + sex + wkswork1
File name:
  C:\Revolution\R-Enterprise-7.0\R-
  3.0.2\library\RevoScaleR\SampleData\CensusWorkers.xdf
Number of valid observations: 351121

  Name      Mean      StdDev      Min Max      ValidObs MissingObs
age      40.42814    11.385017    20    65    351121      0
incwage  35333.83894  40444.544084    0  354000    351121      0
perwt    20.34423     9.633100     2    168    351121      0
wkswork1  48.62566     6.953843    21    52    351121      0

Category Counts for sex
Number of valid observations: 351121
Number of missing observations: 0

  sex      Counts
Male  189344
Female 161777
```

For each term in the formula, the mean, standard deviation, minimum, maximum, and number of valid observations is shown. If `byTerm=FALSE`, observations (rows) containing missing values for any of the specified variables are omitted in calculating the summary. Cell counts for categorical variables are included.

5.3 Computation of Summary Statistics by Group Using Interactions

If you specify an interaction between a numeric variable and a factor variable, you obtain a summary of the numeric variable for each level of the factor. For example, using the sample data set `AirlineDemoSmall.xdf`, we can use the following command to ask for a summary of arrival delay by day of week:

```
rxSummary(~ ArrDelay:DayOfWeek, data = file.path(readPath,
  "AirlineDemoSmall.xdf"))
```

The request produces the following output:

```
Call:
rxSummary(formula = ~ArrDelay:DayOfWeek, data = file.path(readPath,
  "AirlineDemoSmall.xdf"))

Summary Statistics Results for: ~ArrDelay:DayOfWeek
File name: C:\Revolution\R-Enterprise-7.0\R-3.0.2\library\
RevoScaleR\SampleData\AirlineDemoSmall.xdf
Number of valid observations: 6e+05

Name           Mean      StdDev   Min Max   ValidObs MissingObs
ArrDelay:DayOfWeek 11.31794 40.68854 -86 1490 582628 17372

Statistics by category (7 categories):
```

Category	DayOfWeek	Means	StdDev	Min	Max	ValidObs
ArrDelay for DayOfWeek=Monday	Monday	12.025604	40.02463	-76	1017	95298
ArrDelay for DayOfWeek=Tuesday	Tuesday	11.293808	43.66269	-70	1143	74011
ArrDelay for DayOfWeek=Wednesday	Wednesday	10.156539	39.58803	-81	1166	76786
ArrDelay for DayOfWeek=Thursday	Thursday	8.658007	36.74724	-58	1053	79145
ArrDelay for DayOfWeek=Friday	Friday	14.804335	41.79260	-78	1490	80142
ArrDelay for DayOfWeek=Saturday	Saturday	11.875326	45.24540	-73	1370	83851
ArrDelay for DayOfWeek=Sunday	Sunday	10.331806	37.33348	-86	1202	93395

Interactions provide the one exception to the “responseless” formula mentioned above. If you want to obtain the interaction of a continuous variable with one or more factors, you can use a formula of the form $y \sim x:z$, where y is the continuous variable and x and z are factors. This has precisely the same effect as specifying the formula as $\sim y:x:z$, but is more suggestive of the result—that is, summary statistics for y at every combination of levels of x and z .

5.4 On-the-Fly Factors in Formulas

You can force RevoScaleR to treat a variable as a factor (with a level for each integer value from the low to high value) by wrapping it with the function call syntax `F()`. Returning to the `CensusWorkers.xdf` file, in the following example a factor level will temporarily be created for each age from 20 through 65:

```
rxSummary(~ incwage:F(age), data = censusWorkers)
```

58 On-the-Fly Factors in Formulas

The results provide us not only summary statistics for wage income in the overall data set, but summary statistics on wage income for each age:

```
Call:
rxSummary(formula = ~incwage:F(age), data = censusWorkers)

Summary Statistics Results for: ~incwage:F(age)
File name:
  C:\Revolution\R-Enterprise-7.0\R-
  3.0.2\library\RevoScaleR\SampleData\CensusWorkers.xdf
Number of valid observations: 351121
```

Name	Mean	StdDev	Min	Max	ValidObs	MissingObs
incwage:F(age)	35333.84	40444.54	0	354000	351121	0

Statistics by category (46 categories):

Category	F_age	Means	StdDev	Min	Max	ValidObs
incwage for F(age)=20	20	12669.94	12396.99	0	336000	6500
incwage for F(age)=21	21	14114.23	12107.81	0	336000	6479
incwage for F(age)=22	22	15982.00	12374.14	0	336000	6676
incwage for F(age)=23	23	18503.92	15093.53	0	336000	6884
incwage for F(age)=24	24	20672.06	14315.67	0	354000	6931
incwage for F(age)=25	25	23856.25	17319.42	0	336000	7273
incwage for F(age)=26	26	25938.17	20707.39	0	354000	7116
incwage for F(age)=27	27	26902.97	20608.09	0	354000	7584
incwage for F(age)=28	28	28531.59	24185.48	0	354000	8184
incwage for F(age)=29	29	30153.10	25715.94	0	354000	8889
incwage for F(age)=30	30	30691.10	26955.27	0	354000	9055
incwage for F(age)=31	31	31647.06	27331.38	0	354000	8670
incwage for F(age)=32	32	33459.31	33405.36	0	354000	8459
incwage for F(age)=33	33	34208.33	33497.33	0	354000	8574
incwage for F(age)=34	34	34364.06	33874.67	0	354000	9058
incwage for F(age)=35	35	35739.92	36549.20	0	354000	9743
incwage for F(age)=36	36	36945.24	40035.96	0	354000	9888
incwage for F(age)=37	37	36970.63	39981.26	0	354000	9860
incwage for F(age)=38	38	37331.39	39574.54	0	354000	10211
incwage for F(age)=39	39	38899.67	44190.88	0	354000	10378
incwage for F(age)=40	40	38279.34	42388.61	0	354000	10756
incwage for F(age)=41	41	39678.52	45528.92	0	354000	10503
incwage for F(age)=42	42	40748.10	46685.33	0	354000	10511
incwage for F(age)=43	43	39910.90	45446.32	0	354000	10296
incwage for F(age)=44	44	40524.19	46282.94	0	354000	10122
incwage for F(age)=45	45	41450.27	47370.00	0	354000	10074
incwage for F(age)=46	46	40521.07	44746.48	0	354000	9703
incwage for F(age)=47	47	41371.40	45954.91	0	354000	9527
incwage for F(age)=48	48	42061.04	47094.85	0	354000	9093
incwage for F(age)=49	49	41618.36	45835.39	0	354000	8776
incwage for F(age)=50	50	42789.36	48414.56	0	354000	8868
incwage for F(age)=51	51	41912.11	46185.42	0	354000	8506
incwage for F(age)=52	52	43169.23	49483.34	0	354000	8690
incwage for F(age)=53	53	41864.13	46522.74	0	354000	8362
incwage for F(age)=54	54	42920.45	50279.99	0	354000	6275
incwage for F(age)=55	55	42939.81	50010.80	0	354000	6171
incwage for F(age)=56	56	41157.10	48345.51	0	354000	5915
incwage for F(age)=57	57	40984.69	48426.82	0	354000	5881
incwage for F(age)=58	58	40553.04	50305.92	0	354000	5047
incwage for F(age)=59	59	38738.45	46745.03	0	354000	4512
incwage for F(age)=60	60	37200.02	46858.16	0	354000	3775
incwage for F(age)=61	61	35978.18	44335.98	0	354000	3704
incwage for F(age)=62	62	35000.53	46405.31	0	354000	3206
incwage for F(age)=63	63	34098.00	45353.94	0	354000	2563

```
incwage for F(age)=64 64      32964.57 46318.27 0      354000 2248
incwage for F(age)=65 65      31698.98 51777.97 0      354000 1625
```

If you include an interaction between two factors, the summary provides cell counts for all combinations of levels of the factors. Since the census data has probability weights, we can use the *pweights* argument to get weighted counts:

```
rxSummary(~ sex:state, pweights = "perwt", data = censusWorkers)
```

Call:
 rxSummary(formula = ~sex:state, data = censusWorkers, pweights = "perwt")

Summary Statistics Results for: ~sex:state
 File name:
 C:\Revolution\R-Enterprise-7.0\R-
 3.0.2\library\RevoScaleR\SampleData\CensusWorkers.xdf
 Probability weights: perwt
 Number of valid observations: 351121

Category Counts for sex
 Number of categories: 6

sex	state	Counts
Male	Connecticut	843736
Female	Connecticut	755843
Male	Indiana	1517966
Female	Indiana	1289412
Male	Washington	1504840
Female	Washington	1231489

It is also possible to perform on-the-fly row selections. In the example below, we restrict the analysis to people ages 30 to 39. We can use the *low* and *high* arguments of the *F* function to restrict the creation of on-the-fly factor levels to the same range (the fourth argument to *F*, *exclude*, defaults to TRUE; this is reflected in the T that appears in the output variable name):

```
rxSummary(~ sex:F(age, low = 30, high = 39), data = censusWorkers,
  pweights="perwt", rowSelection = age >= 30 & age < 40)
```

Call:
 rxSummary(formula = ~sex:F(age, low = 30, high = 39), data = censusWorkers,
 pweights = "perwt", rowSelection = age >= 30 & age < 40)

Summary Statistics Results for: ~sex:F(age, low = 30, high = 39)
 File name:
 C:\Revolution\R-Enterprise-7.0\R-
 3.0.2\library\RevoScaleR\SampleData\CensusWorkers.xdf
 Probability weights: perwt
 Number of valid observations: 93896

Category Counts for sex
 Number of categories: 20

sex	F_age_30_39_T	Counts
Male	30	103242
Female	30	84209
Male	31	100234
Female	31	77947

60 Writing By-Group Summary Statistics to an .xdf File

Male	32	96325
Female	32	75469
Male	33	97734
Female	33	77133
Male	34	103380
Female	34	81812
Male	35	110358
Female	35	89681
Male	36	113444
Female	36	91394
Male	37	110828
Female	37	91563
Male	38	113838
Female	38	95988
Male	39	115552
Female	39	97209

5.5 Writing By-Group Summary Statistics to an .xdf File

By-group statistics are often computed for further analysis or plotting. It can be convenient to store these results in an .xdf file, especially if there are a large number of groups. For example, let's compute the mean and standard deviation of wage income and number of weeks work for each year of age for both men and women using the CensusWorkers.xdf file with data from three states:

```
readPath <- rxGetOption("sampleDataDir")
censusWorkers <- file.path(readPath, "CensusWorkers.xdf")
rxSummary(~ incwage:F(age):sex + wkswork1:F(age):sex, data = censusWorkers,
  byGroupOutFile = "ByAge.xdf",
  summaryStats = c("Mean", "StdDev", "SumOfWeights"),
  pweights = "perwt", overwrite = TRUE)
```

Call:

```
rxSummary(formula = ~incwage:F(age):sex + wkswork1:F(age):sex,
  data = censusWorkers, byGroupOutFile = "ByAge.xdf", summaryStats =
  c("Mean",
    "StdDev", "SumOfWeights"), pweights = "perwt", overwrite = TRUE)
```

Summary Statistics Results for: ~incwage:F(age):sex +
wkswork1:F(age):sex

File name:

C:\Revolution\R-Enterprise-7.0\R-

3.0.2\library\RevoScaleR\SampleData\CensusWorkers.xdf

Probability weights: perwt

Number of valid observations: 351121

Name	Mean	StdDev	SumOfWeights
incwage:F_age:sex	35788.4675	40605.12565	7143286
wkswork1:F_age:sex	48.6373	6.94423	7143286

By-group statistics for incwage:F(age):sex contained in C:\YourDir\ByAge.xdf

By-group statistics for wkswork1:F(age):sex contained in C:\YourDir\ByAge.xdf

We can take a quick look at the first five rows in the data set to see that the first variables are the two factor variables determining the groups: F_age and sex. The remaining variables are the computed by-group statistics.

```
rxGetInfo("ByAge.xdf", numRows = 5)
```

```
File name: C:\YourDir\ByAge.xdf
```

```
Number of observations: 92
```

```
Number of variables: 8
```

```
Number of blocks: 1
```

```
Compression type: zlib
```

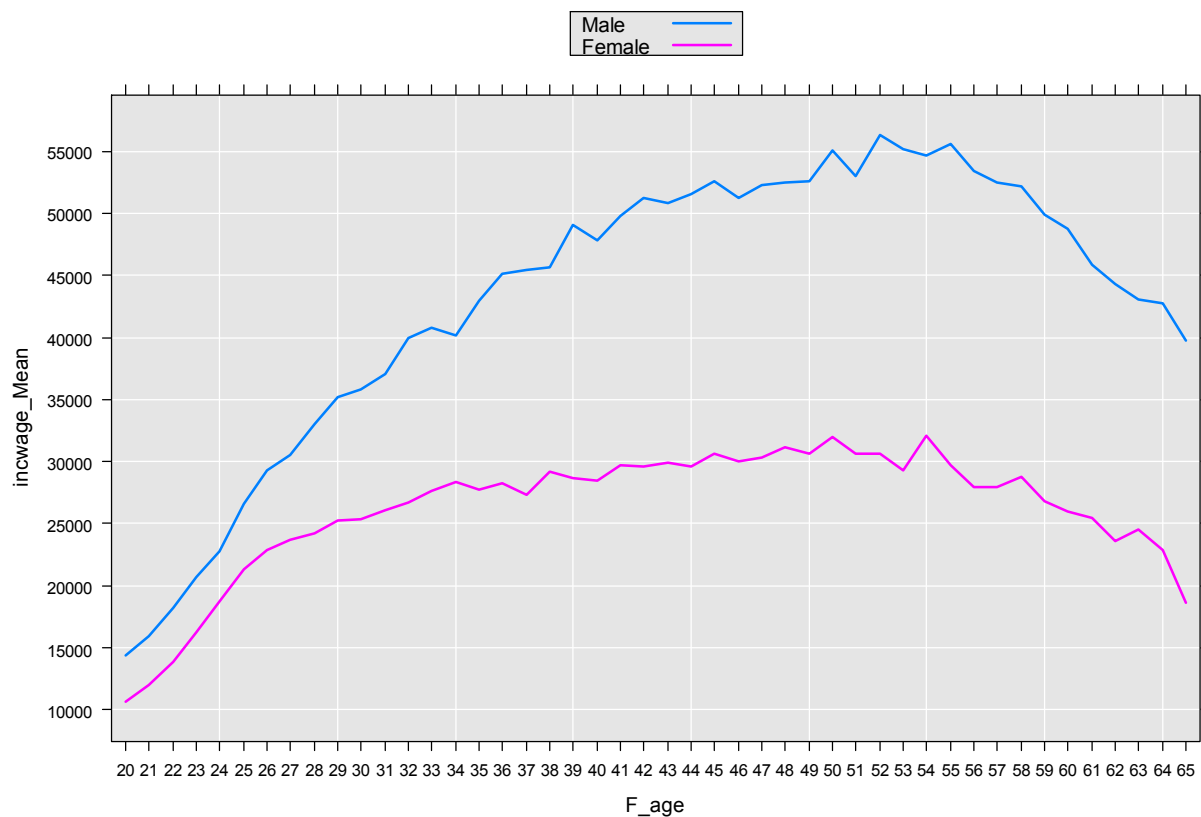
```
Data (5 rows starting with row 1):
```

	F_age	sex	incwage_Mean	incwage_StdDev	incwage_SumOfWeights	wkswork1_Mean
1	20	Male	14437.68	14118.49	71089	44.29758
2	21	Male	15981.48	13191.73	71150	45.27770
3	22	Male	18258.04	13919.44	75979	46.07166
4	23	Male	20739.91	16511.88	79663	46.75025
5	24	Male	22737.17	15345.41	81412	47.51487

	wkswork1_StdDev	wkswork1_SumOfWeights
1	9.755092	71089
2	9.110743	71150
3	8.903617	75979
4	8.451298	79663
5	7.942226	81412

We can plot directly from the .xdf file to visualize our results:

```
rxLinePlot(incwage_Mean~F_age, groups = sex, data = "ByAge.xdf")
```



5.6 Transforming Data in rxSummary

62 Transforming Data in rxSummary

You can use the *transforms* argument to modify your data set on the fly before computing a summary. When used in this way, the original data is unmodified and no permanent copy of the modified data is written to disk. The data summaries returned, however, reflect the modified data.

You can also transform data in the formula itself, by specifying simple functions of the original variables. For example, you can get a summary based on the natural logarithm of a variable as follows :

```
rxSummary(~ log(incwage), data = censusWorkers)
```

This gives the following output:

```
Call:
rxSummary(formula = ~log(incwage), data = censusWorkers)

Summary Statistics Results for: ~log(incwage)
File name:
  C:\Revolution\R-Enterprise-7.0\R-
  3.0.2\library\RevoScaleR\SampleData\CensusWorkers.xdf
Number of valid observations: 351121
```

Name	Mean	StdDev	Min	Max	ValidObs	MissingObs
log(incwage)	10.19694	0.8387598	1.386294	12.77705	331625	19496

Chapter 6.

Crosstabs

Crosstabs, also known as *contingency tables* or *crosstabulations*, are a convenient way to summarize cross-classified categorical data—that is, data that can be tabulated according to multiple levels of two or more factors. If only two factors are involved, the table is sometimes called a *two-way table*. If three factors are involved, the table is sometimes called a *three-way table*.

For large data sets, cross-tabulations of binned numeric data, that is, data which has been converted to a factor where the levels represent ranges of values, can be a very fast way to get insight into the relationships among variables. In RevoScaleR, the `rxCube` function is the primary tool to create contingency tables.

For example, the built-in data set `UCBAdmissions` includes information on admissions by gender to various departments at the University of California at Berkeley. We can look at the contingency table as follows:

```
UCBADF <- as.data.frame(UCBAdmissions)
z <- rxCube(Freq ~ Gender:Admit, data = UCBADF)
```

64 Transforming Data in rxSummary

(Because cross-tabulations are explicitly about exploring interactions between variables, multiple predictors must always be specified using the interaction operator ":", and not the terms operator "+".)

Typing `z` yields the following:

```
Call:
rxCube(formula = Freq ~ Gender:Admit, data = UCBADF)

Cube Results for: Freq ~ Gender:Admit
Data: UCBADF
Dependent variable(s): Freq
Number of valid observations: 24
Number of missing observations: 0
Statistic: Freq means
```

	Gender	Admit	Freq	Counts
1	Male	Admitted	199.66667	6
2	Female	Admitted	92.83333	6
3	Male	Rejected	248.83333	6
4	Female	Rejected	213.00000	6

This data set is widely used in statistics texts because it illustrates Simpson's paradox, which is that in some cases a comparison that holds true in a number of groups is reversed when those groups are aggregated to form a single group. From the above table, in which admissions data is aggregated across all departments, it would appear that males are admitted at a higher rate than women. However, if we look at the more granular analysis by department, we find that in four of the six departments, women are admitted at a higher rate than men:

```
z2 <- rxCube(Freq ~ Gender:Admit:Dept, data = UCBADF)
z2
```

This yields the following output:

```
Call:
rxCube(formula = Freq ~ Gender:Admit:Dept, data = UCBADF)

Cube Results for: Freq ~ Gender:Admit:Dept
Data: UCBADF
Dependent variable(s): Freq
Number of valid observations: 24
Number of missing observations: 0
Statistic: Freq means
```

	Gender	Admit	Dept	Freq	Counts
1	Male	Admitted	A	512	1
2	Female	Admitted	A	89	1
3	Male	Rejected	A	313	1
4	Female	Rejected	A	19	1
5	Male	Admitted	B	353	1
6	Female	Admitted	B	17	1
7	Male	Rejected	B	207	1
8	Female	Rejected	B	8	1
9	Male	Admitted	C	120	1
10	Female	Admitted	C	202	1
11	Male	Rejected	C	205	1

12	Female	Rejected	C	391	1
13	Male	Admitted	D	138	1
14	Female	Admitted	D	131	1
15	Male	Rejected	D	279	1
16	Female	Rejected	D	244	1
17	Male	Admitted	E	53	1
18	Female	Admitted	E	94	1
19	Male	Rejected	E	138	1
20	Female	Rejected	E	299	1
21	Male	Admitted	F	22	1
22	Female	Admitted	F	24	1
23	Male	Rejected	F	351	1
24	Female	Rejected	F	317	1

6.1 Letting the Data Speak Example 1: Analyzing U.S. 2000 Census Data

The `CensusWorkers.xdf` data set contains a subset of the U.S. 2000 5% Census for individuals aged 20 to 65 who worked at least 20 weeks during the year from three states. Let's examine the relationship between wage income (represented in the data set by the variable `incwage`) and age.

As we mentioned in Chapter 4, a useful way to observe the relationship between numeric variables is to bin the predictor variable (in our case, age), and then plot the mean of the response for each bin. The simplest way to bin age is to use the `F()` wrapper within our initial formula; it creates a separate bin for each distinct value of age. (More precisely, it creates a bin of length one from the low value of age to the high value of age—if some ages are missing in the original data set, bins are created for them anyway.)

We create our original model as follows:

```
censusWorkers <- file.path(readPath, "CensusWorkers.xdf")
censusWorkersCube <- rxCube(incwage ~ F(age), data=censusWorkers)
```

We first look at the results in tabular form by typing the returned object name, `censusWorkersCube`, which yields the following:

```
Call:
rxCube(formula = incwage ~ F(age), data = censusWorkers)

Cube Results for: incwage ~ F(age)
File name:
  C:\Revolution\R-Enterprise-7.0\R-
  3.0.2\library\RevoScaleR\SampleData\CensusWorkers.xdf
Dependent variable(s): incwage
Number of valid observations: 351121
Number of missing observations: 0
Statistic: incwage means
```

	F_age	incwage	Counts
1	20	12669.94	6500
2	21	14114.23	6479
3	22	15982.00	6676
4	23	18503.92	6884
5	24	20672.06	6931

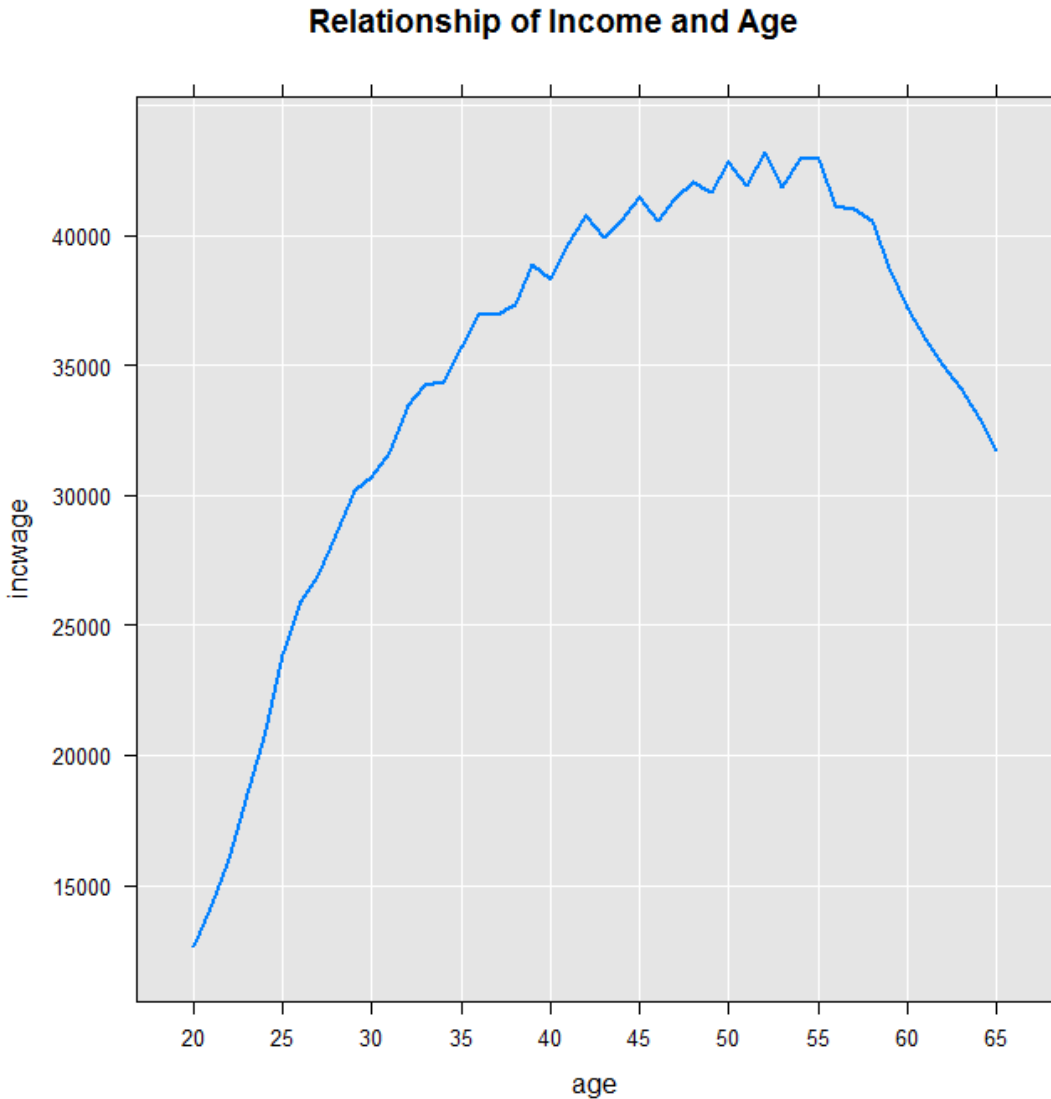
66 Letting the Data Speak Example 1: Analyzing U.S. 2000 Census Data

6	25	23856.25	7273
7	26	25938.17	7116
8	27	26902.97	7584
9	28	28531.59	8184
10	29	30153.10	8889
11	30	30691.10	9055
12	31	31647.06	8670
13	32	33459.31	8459
14	33	34208.33	8574
15	34	34364.06	9058
16	35	35739.92	9743
17	36	36945.24	9888
18	37	36970.63	9860
19	38	37331.39	10211
20	39	38899.67	10378
21	40	38279.34	10756
22	41	39678.52	10503
23	42	40748.10	10511
24	43	39910.90	10296
25	44	40524.19	10122
26	45	41450.27	10074
27	46	40521.07	9703
28	47	41371.40	9527
29	48	42061.04	9093
30	49	41618.36	8776
31	50	42789.36	8868
32	51	41912.11	8506
33	52	43169.23	8690
34	53	41864.13	8362
35	54	42920.45	6275
36	55	42939.81	6171
37	56	41157.10	5915
38	57	40984.69	5881
39	58	40553.04	5047
40	59	38738.45	4512
41	60	37200.02	3775
42	61	35978.18	3704
43	62	35000.53	3206
44	63	34098.00	2563
45	64	32964.57	2248
46	65	31698.98	1625

As we wanted, the table contains average values of *incwage* for each level of *age*. If we want to create a plot of the results, we can use the *rxResultsDF* function to conveniently convert the output into a data frame. The *F_age* factor variable will automatically be converted back to an integer *age* variable. Then we can plot the data using *rxLinePlot*:

```
censusWorkersCubeDF <- rxResultsDF(censusWorkersCube)
rxLinePlot(incwage ~ age, data=censusWorkersCubeDF,
  title="Relationship of Income and Age")
```

The resulting plot shows clearly the relationship of income on age:



6.2 Transforming Data

Because crosstabs require categorical data for the predictors, you have to do some work to crosstabulate continuous data. In the previous section, we saw that the `F()` wrapper can do a simple transformation within a formula. The `transforms` argument to `rxCrossTabs` can be used to give you greater control over such transformations.

For example, the `kyphosis` data from the `rpart` package consists of one categorical variable, `Kyphosis`, and three continuous variables `Age`, `Number`, and `Start`. The `Start` variable indicates the topmost vertebra involved in a certain type of spinal surgery, and has a range of 1 to 18. Since there are 7 cervical vertebrae and 12 thoracic vertebrae, we can classify the start variable as either cervical or thoracic as follows:

```
cut(Start, breaks=c(0, 7.5, 19.5), labels=c("cervical", "thoracic"))
```

68 Transforming Data

Similarly, we can create a factorized Age variable as follows (in the original data, age is given in months; with our set of breaks, we cut the data into ranges of years):

```
cut(Age, breaks=c(0, 12, 60, 119, 180, 220), labels=c("<1", "1-4",  
"5-9", "10-15", ">15"))
```

We can now crosstabulate the data using the above transforms and it is instructive to start by looking at the three two-way tables formed by tabulating Kyphosis with the three predictor variables:

```
library(rpart)  
rxCube(~ Kyphosis:Age, data = kyphosis,  
       transforms=list(Age = cut(Age, breaks=c(0, 12, 60, 119,  
180, 220), labels=c("<1", "1-4", "5-9", "10-15", ">15"))))
```

Call:

```
rxCube(formula = ~Kyphosis:Age, data = kyphosis, transforms = list(Age =  
cut(Age,  
    breaks = c(0, 12, 60, 119, 180, 220), labels = c("<1", "1-4",  
"5-9", "10-15", ">15"))))
```

Cube Results for: ~Kyphosis:Age

Data: kyphosis

Number of valid observations: 81

Number of missing observations: 0

	Kyphosis	Age	Counts
1	absent	<1	13
2	present	<1	0
3	absent	1-4	13
4	present	1-4	4
5	absent	5-9	17
6	present	5-9	6
7	absent	10-15	19
8	present	10-15	7
9	absent	>15	2
10	present	>15	0

```
rxCube(~ Kyphosis:F(Number), data = kyphosis)
```

Call:

```
rxCube(formula = ~Kyphosis:F(Number), data = kyphosis)
```

Cube Results for: ~Kyphosis:F(Number)

Data: kyphosis

Number of valid observations: 81

Number of missing observations: 0

	Kyphosis	F_Number	Counts
1	absent	2	12
2	present	2	0
3	absent	3	19
4	present	3	4
5	absent	4	16
6	present	4	2
7	absent	5	12
8	present	5	5
9	absent	6	2
10	present	6	2
11	absent	7	2

12	present	7	3
13	absent	8	0
14	present	8	0
15	absent	9	1
16	present	9	0
17	absent	10	0
18	present	10	1

```
rxCube(~ Kyphosis:Start, data = kyphosis,
       transforms=list(Start = cut(Start, breaks=c(0, 7.5, 19.5),
                                   labels=c("cervical", "thoracic"))))
```

Call:

```
rxCube(formula = ~Kyphosis:Start, data = kyphosis, transforms = list(Start =
cut(Start,
    breaks = c(0, 7.5, 19.5), labels = c("cervical", "thoracic"))))
```

Cube Results for: ~Kyphosis:Start

Data: kyphosis

Number of valid observations: 81

Number of missing observations: 0

	Kyphosis	Start	Counts
1	absent	cervical	8
2	present	cervical	9
3	absent	thoracic	56
4	present	thoracic	8

From these, we see that the probability of the post-operative complication Kyphosis seems to be greater if the *Start* is a cervical vertebra and as more vertebrae are involved in the surgery. Similarly, it appears that the dependence on age is non-linear: it first increases with age, peaks in the range 5-9, and then decreases again.

6.3 Cross-Tabulation with `rxCrossTabs`

The `rxCrossTabs` function is an alternative to the `rxCube` function, which performs the same calculations, but displays its results in format similar to the standard R `xtabs` function. For some purposes, this format can be more informative than the matrix-like display of `rxCube`, and in some situations can be more compact as well.

As an example, consider again the admission data example:

```
z3 <- rxCrossTabs(Freq ~ Gender:Admit:Dept, data = UCBAADF)
z3
```

Call:

```
rxCrossTabs(formula = Freq ~ Gender:Admit:Dept, data = UCBAADF)
```

Cross Tabulation Results for: Freq ~ Gender:Admit:Dept

Data: UCBAADF

Dependent variable(s): Freq

Number of valid observations: 24

Number of missing observations: 0

Statistic: sums

70 Cross-Tabulation with rxCrossTabs

```

Freq, Dept = A (sums):
      Admit
Gender  Admitted Rejected
Male      512      313
Female     89      19

.....

Freq, Dept = B (sums):
      Admit
Gender  Admitted Rejected
Male      353      207
Female     17       8

.....

Freq, Dept = C (sums):
      Admit
Gender  Admitted Rejected
Male      120      205
Female     202      391

.....

Freq, Dept = D (sums):
      Admit
Gender  Admitted Rejected
Male      138      279
Female     131      244

.....

Freq, Dept = E (sums):
      Admit
Gender  Admitted Rejected
Male      53      138
Female     94      299

.....

Freq, Dept = F (sums):
      Admit
Gender  Admitted Rejected
Male      22      351
Female     24      317

```

You can see the row, column, and total percentages by calling the `summary` function on the `rxCrossTabs` object:

```
summary(z3)

Call:
rxCrossTabs(formula = Freq ~ Gender:Admit:Dept, data = UCBADF)

Cross Tabulation Results for: Freq ~ Gender:Admit:Dept
Data: UCBADF
Dependent variable(s): Freq
Number of valid observations: 24
Number of missing observations: 0
Statistic: sums

Freq, Dept = A (sums):
```

	Admitted	Rejected	Row Total
Male	512.000000	313.000000	825.000000
Row%	62.060606	37.939394	
Col%	85.191348	94.277108	
Tot%	54.876742	33.547696	88.42444
Female	89.000000	19.000000	108.000000
Row%	82.407407	17.592593	
Col%	14.808652	5.722892	
Tot%	9.539121	2.036442	11.57556
Col Total	601.000000	332.000000	
Grand Total	933.000000		

.....

Freq, Dept = B (sums):

	Admitted	Rejected	Row Total
Male	353.000000	207.000000	560.000000
Row%	63.035714	36.964286	
Col%	95.405405	96.279070	
Tot%	60.341880	35.384615	95.726496
Female	17.000000	8.000000	25.000000
Row%	68.000000	32.000000	
Col%	4.594595	3.720930	
Tot%	2.905983	1.367521	4.273504
Col Total	370.000000	215.000000	
Grand Total	585.000000		

.....

Freq, Dept = C (sums):

	Admitted	Rejected	Row Total
Male	120.000000	205.000000	325.000000
Row%	36.92308	63.07692	
Col%	37.26708	34.39597	
Tot%	13.07190	22.33115	35.40305
Female	202.000000	391.000000	593.000000
Row%	34.06408	65.93592	
Col%	62.73292	65.60403	
Tot%	22.00436	42.59259	64.59695
Col Total	322.000000	596.000000	
Grand Total	918.000000		

.....

Freq, Dept = D (sums):

	Admitted	Rejected	Row Total
Male	138.000000	279.000000	417.000000
Row%	33.09353	66.90647	
Col%	51.30112	53.34608	
Tot%	17.42424	35.22727	52.65152
Female	131.000000	244.000000	375.000000
Row%	34.93333	65.06667	
Col%	48.69888	46.65392	
Tot%	16.54040	30.80808	47.34848
Col Total	269.000000	523.000000	
Grand Total	792.000000		

.....

Freq, Dept = E (sums):

	Admitted	Rejected	Row Total
Male	53.000000	138.000000	191.000000
Row%	27.748691	72.25131	

72 A Large Data Example

Col%	36.054422	31.57895	
Tot%	9.075342	23.63014	32.70548
Female	94.000000	299.00000	393.00000
Row%	23.918575	76.08142	
Col%	63.945578	68.42105	
Tot%	16.095890	51.19863	67.29452
Col Total	147.000000	437.00000	
Grand Total	584.000000		

.....

```
Freq, Dept = F (sums):
```

	Admitted	Rejected	Row Total
Male	22.000000	351.00000	373.0000
Row%	5.898123	94.10188	
Col%	47.826087	52.54491	
Tot%	3.081232	49.15966	52.2409
Female	24.000000	317.00000	341.0000
Row%	7.038123	92.96188	
Col%	52.173913	47.45509	
Tot%	3.361345	44.39776	47.7591
Col Total	46.000000	668.00000	
Grand Total	714.000000		

You can see, for example, that in Department A, 62 percent of male applicants are admitted, but 82 percent of female applicants are admitted, and in Department B, 63 percent of male applicants are admitted, while 68 percent of female applicants are admitted.

6.4 A Large Data Example

The power of *rxCrossTabs* is most evident when you need to tabulate a data set that won't fit into memory. For example, in the large airline data set *AirOnTime87to12.xdf*, you can obtain the mean arrival delay by carrier and day of week as follows (if you have downloaded the data set, modify the first line below to reflect your local path):

```
bigDataDir <- "C:/Revolution/Data"
bigAirData <- file.path(bigDataDir, "AirOnTime87to12/AirOnTime87to12.xdf")
arrDelayXT <- rxCrossTabs(ArrDelay ~ UniqueCarrier:DayOfWeek,
  data = bigAirData, blocksPerRead = 30)
print(arrDelayXT)
```

This gives the following output:

```
Call:
rxCrossTabs(formula = ArrDelay ~ UniqueCarrier:DayOfWeek, data = bigAirData,
  blocksPerRead = 30)
```

```
Cross Tabulation Results for: ArrDelay ~ UniqueCarrier:DayOfWeek
File name: C:\Revolution\Data\AirOnTime87to12\AirOnTime87to12.xdf
Dependent variable(s): ArrDelay
Number of valid observations: 145576737
Number of missing observations: 3042918
Statistic: sums
```

```
ArrDelay (sums):
      DayOfWeek
UniqueCarrier Mon    Tues    Wed    Thur    Fri    Sat    Sun
```


AA	15956852	13367087	16498840	20554660	20714146	9359729	14577582
US	11797366	11688903	14065606	17379113	19541862	5865427	10264583
AS	3144578	2677858	3182356	3980209	4415144	2433581	3039129
CO	8464507	7966834	9537366	11901028	11749616	3553719	6562487
DL	18146092	15962559	19474389	24077435	24933864	10483280	16414060
EA	782103	832332	796811	1152825	1405399	638911	670924
HP	3577460	3170343	3700890	4734543	5015896	2864314	3985741
NW	7750970	7818040	9256994	11199718	10294116	3726129	6504924
PA (1)	191137	235924	225260	290844	345238	174284	229677
PI	1164688	1391526	1456173	1515403	1568266	939820	986642
PS	88144	111282	122520	133567	173422	44362	88891
TW	3356944	3459185	4060151	5027427	5267750	1669048	2377671
UA	15941096	14731587	17801128	21060697	20920843	9174567	13688577
WN	12438738	8978339	12215989	21556781	26787623	4972506	15973176
ML (1)	20735	50927	55881	75030	62855	44549	18173
KH	20744	-26425	-24265	30078	98529	33595	43026
MQ	7065052	5152746	5893882	7136735	8087443	2947023	5540099
B6	1886261	1340744	1736450	2373151	2930423	1012698	1969546
DH	795614	527649	708129	801968	986930	227907	504644
EV	5733212	3684210	4005374	5262924	5874647	1753361	4418290
FL	2666677	1694294	1810548	2928247	3068538	819827	2188420
OO	4717107	3106319	3438056	4725854	5481441	2797745	4764041
XE	4870453	3904752	4532069	5349375	5315818	1636826	3531446
TZ	228508	147963	197371	224693	275340	39722	148940
HA	-72468	-92714	-66578	4840	153830	-2082	-22196
OH	2276399	1567510	1830571	2336032	2702519	922531	1659708
F9	551932	484426	566122	858027	729273	337695	526887
YV	1959906	1419073	1463954	1930992	2152270	1270104	1830749
9E	787776	579608	590038	709161	869358	304151	586378
VX	10208	37079	12956	42661	73457	2943	39987

6.5 Tests of Independence on Cross-Tabulated Data

One common use of contingency tables is to test whether the tabulated variables are independent. **RevoScaleR** includes several tests of independence, all of which expect data in the standard R *xtabs* format. You can get data in this format from the *rxCrossTabs* function by using the argument *returnXtabs=TRUE*:

```
bigDataDir <- "C:/Revolution/Data"
bigAirData <- file.path(bigDataDir, "AirOnTime87to12/AirOnTime87to12.xdf")
arrDelayXTab <- rxCrossTabs(ArDel15~ UniqueCarrier:DayOfWeek,
  data = bigAirData, blocksPerRead = 30, returnXtabs=TRUE)
```

You can then use this as input to any of the following functions:

- *rxChiSquaredTest*: performs Pearson's chi-squared test of independence.
- *rxFisherTest*: performs Fisher's exact test of independence.
- *rxKendallCor*: performs a Kendall tau test of independence. There are three flavors of test, a, b, and c; by default, the b flavor, which accounts for ties, is used.

(In fact, regular *rxCrossTabs* or *rxCube* output can be used as input to these functions, but they will be converted to *xtabs* format first, so it is generally somewhat more efficient to have *rxCrossTabs* return the *xtabs* format directly.)

74 Tests of Independence on Cross-Tabulated Data

Here we use the `arrDelayXTab` data created above and perform a Pearson's chi-squared test of independence on it:

```
rxChiSquaredTest(arrDelayXTab)
```

This gives the following output:

```
Chi-squared test of independence between UniqueCarrier and DayOfWeek
X-squared  df p-value
105645.8  174      0
```

For large contingency tables such as this one, the chi-squared test is the tool of choice. For smaller tables, particularly those with cells with expected counts fewer than five, Fisher's exact test is useful. On a large table, however, Fisher's exact test may not be an option. For example, if we try it on our airline table, it returns an error:

```
rxFisherTest(arrDelayXTab)

Error in FUN(tbl[, , i], ...) : FEXACT error 40.
Out of workspace.
```

To show the Fisher test, we return to the admissions data from the beginning of the chapter.

This time we use `rxCrossTabs` to return an `xtabs` object:

```
UCBADF <- as.data.frame(UCBAdmissions)
admissCTabs <- rxCrossTabs(Freq ~ Gender:Admit, data = UCBADF,
  returnXtabs=TRUE)
```

We then call `rxFisherTest` on the resulting table:

```
rxFisherTest(admissCTabs)

Fisher's Exact Test for Count Data
estimate 1 95% CI Lower 95% CI Upper      p-value
1.840856   1.621356   2.091246 4.835903e-22
HA: two.sided
H0: odds ratio = 1
```

The chi-squared test works equally well on this example:

```
rxChiSquaredTest(admissCTabs)

Chi-squared test of independence between Gender and Admit
X-squared df      p-value
91.6096   1 1.055797e-21
```

In both cases, we are given indisputable evidence of the independence of our two predictor factors. For this example, we could have just as easily used the standard R functions `chisq.test` and `fisher.test`. The RevoScaleR enhancements, however, permit `rxChiSquaredTest` and `rxFisherTest` to work on `xtabs` objects with multiple tables. For example, if we expand our examination of the admissions data to include the department info, we obtain a multi-way contingency table:

```
admissCTabs2 <- rxCrossTabs (Freq ~ Gender:Admit:Dept, data = UCBAADF,
  returnXtabs=TRUE)
```

The chi-squared and Fisher's exact test results are shown below; notice that they provide a test of independence between Gender and Admit for each level of Dept:

```
rxChiSquaredTest(admissCTabs2)

Chi-squared test of independence between Gender and Admit
      X-squared df      p-value
Dept==A 16.37177373 1 5.205468e-05
Dept==B  0.08509801 1 7.705041e-01
Dept==C  0.63322380 1 4.261753e-01
Dept==D  0.22159370 1 6.378283e-01
Dept==E  0.80804765 1 3.686981e-01
Dept==F  0.21824336 1 6.403817e-01

rxFisherTest(admissCTabs2)

Fisher's Exact Test for Count Data
      odds ratio 95% CI Lower 95% CI Upper      p-value
Dept==A  0.3495628    0.1970420    0.5920417 1.669189e-05
Dept==B  0.8028124    0.2944986    2.0040231 6.770899e-01
Dept==C  1.1329004    0.8452173    1.5162918 3.866166e-01
Dept==D  0.9213798    0.6789572    1.2504742 5.994965e-01
Dept==E  1.2211852    0.8064776    1.8385155 3.603964e-01
Dept==F  0.8280944    0.4332888    1.5756278 5.458408e-01
      HA: two.sided
      H0: odds ratio = 1
```

Like Fisher's exact test, the Kendall tau correlation test works best on smaller contingency tables. Here is an example of what it returns when applied to our admissions data (the results will differ from run to run as the underlying algorithm relies on sampling):

```
rxKendallCor(admissCTabs2)

      taub p-value
Dept==A -0.13596550 0.000
Dept==B -0.02082575 0.666
Dept==C  0.02865045 0.380
Dept==D -0.01939676 0.585
Dept==E  0.04140240 0.383
Dept==F -0.02319366 0.530
      HA: two.sided
```

6.6 Odds Ratios and Risk Ratios

Another common task associated with 2 x 2 contingency tables is the calculation of odds ratios and risk ratios (also known as relative risk). The two functions `rxOddsRatio` and `rxRiskRatio` in **RevoScaleR** can be used to compute these quantities. The odds ratio and the risk ratio are closely related: the odds ratio computes the relative odds of an event among two or more groups, while the risk ratio computes the relative probabilities of an event. Consider again the contingency table `admissCTabs`:

76 Odds Ratios and Risk Ratios

```
admissCTabs
```

Gender	Admit	
	Admitted	Rejected
Male	1198	1493
Female	557	1278

In this example, the odds of being admitted as a male are 1198/1493, or about 4 to 5 against. The odds of being admitted as a female are 557/1278, or about 4 to 9 against. The odds ratio is $(1198/1493)/(557/1278)$, or 1.8 greater odds that a male will be admitted as opposed to a woman.

```
rxOddsRatio(admissCTabs)
```

```
data:
Z = 0.6104, p-value < 2.2e-16
alternative hypothesis: two.sided
95 percent confidence interval:
 1.624377 2.086693
sample estimates:
oddsRatio
 1.84108
```

The risk ratio, by contrast, compares the probabilities of being *rejected*, that is, $1493/(1198+1493)$ for a man versus $1278/(557+1278)$ for a woman. So here the risk ratio is 0.697 (the probability of a woman being rejected) divided by 0.555 (the probability of a man being rejected), or 1.255:

```
rxRiskRatio(admissCTabs)
```

```
data:
Z.Female = 0.2274, p-value < 2.2e-16
alternative hypothesis: two.sided
95 percent confidence interval:
 1.199631 1.313560
sample estimates:
riskRatio.Female
 1.255303
```

Chapter 7.

Fitting Linear Models

Linear regression models are fitted in RevoScaleR using the *rxLinMod* function. Like other RevoScaleR functions, *rxLinMod* uses an updating algorithm to compute the regression model. The R object returned by *rxLinMod* includes the estimated model coefficients and the call used to generate the model, together with other information that allows RevoScaleR to recompute the model—because *rxLinMod* is designed to work with arbitrarily large data sets, quantities such as residuals and fitted values are not included in the return object, although these can be obtained easily once the model has been fitted.

As a simple example, let's use the sample data set *AirlineDemoSmall.xdf* and fit the arrival delay by day of week:

```
readPath <- rxGetOption("sampleDataDir")
airlineDemoSmall <- file.path(readPath, "AirlineDemoSmall.xdf")
rxLinMod(ArrDelay ~ DayOfWeek, data = airlineDemoSmall)

Call:
rxLinMod(formula = ArrDelay ~ DayOfWeek, data = airlineDemoSmall)

Linear Regression Results for: ArrDelay ~ DayOfWeek
File name:
  C:\Revolution\R-Enterprise-7.0\R-
  3.0.2\library\RevoScaleR\SampleData\AirlineDemoSmall.xdf
Dependent variable(s): ArrDelay
Total independent variables: 8 (Including number dropped: 1)
```

78 Obtaining a Summary of the Model

```
Number of valid observations: 582628
Number of missing observations: 17372
```

```
Coefficients:
              ArrDelay
(Intercept)  10.3318058
DayOfWeek=Monday    1.6937981
DayOfWeek=Tuesday   0.9620019
DayOfWeek=Wednesday -0.1752668
DayOfWeek=Thursday  -1.6737983
DayOfWeek=Friday    4.4725290
DayOfWeek=Saturday  1.5435207
DayOfWeek=Sunday      Dropped
```

Because our predictor is categorical, we can use the `cube` argument to `rxLinMod` to perform the regression using a partitioned inverse, which may be faster and may use less memory than the standard algorithm. The output object also includes a data frame with the averages or counts for each category:

```
arrDelayLm1 <- rxLinMod(ArrDelay ~ DayOfWeek, cube = TRUE,
  data = airlineDemoSmall)
```

Typing the name of the object `arrDelayLm1` yields the following output:

```
Call:
rxLinMod(formula = ArrDelay ~ DayOfWeek, data = airlineDemoSmall,
  cube = TRUE)

Cube Linear Regression Results for: ArrDelay ~ DayOfWeek
File name:
  C:\Revolution\R-Enterprise-7.0\R-
  3.0.2\library\RevoScaleR\SampleData\AirlineDemoSmall.xdf
Dependent variable(s): ArrDelay
Total independent variables: 7
Number of valid observations: 582628
Number of missing observations: 17372

Coefficients:
              ArrDelay
DayOfWeek=Monday    12.025604
DayOfWeek=Tuesday   11.293808
DayOfWeek=Wednesday 10.156539
DayOfWeek=Thursday   8.658007
DayOfWeek=Friday    14.804335
DayOfWeek=Saturday  11.875326
DayOfWeek=Sunday    10.331806
```

7.1 Obtaining a Summary of the Model

The print method for the `rxLinMod` model object shows only the call and the coefficients. You can obtain more information about the model by calling the summary method:

```
summary(arrDelayLm1)
```

This produces the following output, which includes substantially more information about the model coefficients, together with the residual standard error, multiple R-squared, and adjusted R-squared:

```
Call:
rxLinMod(formula = ArrDelay ~ DayOfWeek, data = airlineDemoSmall,
  cube = TRUE)

Cube Linear Regression Results for: ArrDelay ~ DayOfWeek
File name:
  C:\Revolution\R-Enterprise-7.0\R-
  3.0.2\library\RevoScaleR\SampleData\AirlineDemoSmall.xdf
Dependent variable(s): ArrDelay
Total independent variables: 7
Number of valid observations: 582628
Number of missing observations: 17372

Coefficients:
              Estimate Std. Error t value Pr(>|t|)      | Counts
DayOfWeek=Monday      12.0256    0.1317   91.32 2.22e-16 *** | 95298
DayOfWeek=Tuesday      11.2938    0.1494   75.58 2.22e-16 *** | 74011
DayOfWeek=Wednesday    10.1565    0.1467   69.23 2.22e-16 *** | 76786
DayOfWeek=Thursday      8.6580    0.1445   59.92 2.22e-16 *** | 79145
DayOfWeek=Friday       14.8043    0.1436  103.10 2.22e-16 *** | 80142
DayOfWeek=Saturday     11.8753    0.1404   84.59 2.22e-16 *** | 83851
DayOfWeek=Sunday       10.3318    0.1330   77.67 2.22e-16 *** | 93395
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 40.65 on 582621 degrees of freedom
Multiple R-squared:  0.001869 (as if intercept included)
Adjusted R-squared:  0.001858
F-statistic: 181.8 on 6 and 582621 DF,  p-value: < 2.2e-16
Condition number: 1
```

7.2 Using Probability Weights

Probability weights are common in survey data; they represent the probability that a case was selected into the sample from the population, and are calculated as the inverse of the sampling fraction. The variable *perwt* in the census data represents a probability weight. You pass probability weights to the RevoScaleR analysis functions using the *pweights* argument, as in the following example:

```
rxLinMod(incwage ~ F(age), pweights = "perwt", data = censusWorkers)
```

This yields the following output:

```
Call:
rxLinMod(formula = incwage ~ F(age), data = censusWorkers, pweights = "perwt")

Linear Regression Results for: incwage ~ F(age)
File name:
  C:\Revolution\R-Enterprise-7.0\R-
  3.0.2\library\RevoScaleR\SampleData\CensusWorkers.xdf
Probability weights: perwt
Dependent variable(s): incwage
```

80 Using Frequency Weights

```
Total independent variables: 47 (Including number dropped: 1)
Number of valid observations: 351121
Number of missing observations: 0
```

Coefficients:

```
incwage
(Intercept) 32111.3012
F_age=20    -19488.0874
F_age=21    -18043.7738
F_age=22    -15928.5538
F_age=23    -13498.2138
F_age=24    -11248.8583
F_age=25     -7948.6603
F_age=26    -5667.3599
F_age=27    -4682.8375
F_age=28    -3024.1648
F_age=29    -1480.8682
F_age=30     -971.4461
F_age=31      143.5648
F_age=32     2009.0019
F_age=33     2861.3031
F_age=34     2797.9986
F_age=35     4038.2131
F_age=36     5511.8633
F_age=37     5148.1841
F_age=38     5957.2190
F_age=39     7652.9870
F_age=40     6969.9630
F_age=41     8387.6821
F_age=42     9150.0006
F_age=43     8936.7590
F_age=44     9267.0820
F_age=45    10148.1702
F_age=46     9099.0659
F_age=47     9996.0450
F_age=48    10408.1712
F_age=49    10281.1324
F_age=50    12029.6751
F_age=51    10247.2529
F_age=52    12105.0654
F_age=53    10957.8211
F_age=54    11881.4307
F_age=55    11490.8457
F_age=56     9442.3856
F_age=57     9331.2347
F_age=58     9353.1980
F_age=59     7526.4912
F_age=60     6078.4463
F_age=61     4636.6756
F_age=62     3129.5531
F_age=63     2535.4858
F_age=64     1520.3707
F_age=65           Dropped
```

7.3 Using Frequency Weights

Frequency weights are useful when your data has a particular form: a set of data in which one or more cases is exactly replicated. If you then compact the data to remove duplicated observations and create a variable to store the number of replications of each observation, you can use that new variable as a frequency weights variable in the RevoScaleR analysis functions.

For example, the sample data set `fourthgraders.xdf` contains the following 44 rows:

	height	eyecolor	male	reps
14	44	Blue	Male	1
23	44	Brown	Male	3
9	44	Brown	Female	1
38	44	Green	Female	1
8	45	Blue	Male	1
31	45	Blue	Female	5
2	45	Brown	Male	2
51	45	Brown	Female	1
6	45	Green	Male	1
47	45	Hazel	Female	1
37	46	Blue	Male	1
21	46	Blue	Female	3
12	46	Brown	Male	2
5	46	Brown	Female	4
19	46	Green	Male	1
80	46	Green	Female	3
64	47	Blue	Male	1
69	47	Blue	Female	3
39	47	Brown	Male	4
18	47	Brown	Female	3
41	47	Green	Male	2
7	47	Green	Female	4
26	47	Hazel	Male	3
17	48	Blue	Male	4
36	48	Blue	Female	3
13	48	Brown	Male	6
62	48	Brown	Female	4
61	48	Green	Female	2
56	48	Hazel	Male	1
10	49	Blue	Male	1
15	49	Blue	Female	5
22	49	Brown	Male	2
4	49	Brown	Female	5
3	49	Green	Male	3
91	49	Green	Female	1
94	50	Blue	Male	1
96	50	Blue	Female	1
97	50	Brown	Male	1
1	50	Brown	Female	1
28	50	Green	Male	4
52	50	Hazel	Male	1
98	51	Blue	Male	1
57	51	Brown	Female	1
90	51	Green	Female	1

The `reps` column shows the number of replications for each observation; the sum of the `reps` column indicates the total number of observations, in this case 100. We can fit a model (admittedly not very useful) of height on eye color with `rxLinMod` as follows:

```
fourthgraders <- file.path(rxGetOption("sampleDataDir"),
  "fourthgraders.xdf")
fourthgraderslm <- rxLinMod(height ~ eyecolor, data = fourthgraders,
  fweights="reps")
```

Typing the name of the object shows the following output:

```
Call:
```

82 Using rxLinMod with R Data Frames

```
rxLinMod(formula = height ~ eyecolor, data = fourthgraders, fweights = "reps")

Linear Regression Results for: height ~ eyecolor
File name:
  C:\Revolution\R-Enterprise-7.0\R-
  3.0.2\library\RevoScaleR\SampleData\fourthgraders.xdf
Linear Regression Results for: height ~ eyecolor
Frequency weights: reps
Dependent variable(s): height
Total independent variables: 5 (Including number dropped: 1)
Number of valid observations: 44
Number of missing observations: 0

Coefficients:
              height
(Intercept)  47.3333333
eyecolor=Blue -0.01075269
eyecolor=Brown -0.08333333
eyecolor=Green  0.40579710
eyecolor=Hazel      Dropped
```

7.4 Using rxLinMod with R Data Frames

While RevoScaleR is primarily designed to work with data on disk, it can also be used with in-memory R data frames. As an example, consider the R sample data frame “cars”, which contains data recorded in the 1920s on the speed of cars and the distances taken to stop.

```
rxLinMod(dist ~ speed, data = cars)
```

We get the following output (which matches the output given by the R function `lm`):

```
Call:
rxLinMod(formula = dist ~ speed, data = cars)

Linear Regression Results for: dist ~ speed
Data: cars
Dependent variable(s): dist
Total independent variables: 2
Number of valid observations: 50
Number of missing observations: 0

Coefficients:
              dist
(Intercept) -17.579095
speed        3.932409
```

7.5 Using the Cube Option for Conditional Predictions

If the `cube` argument is set to `TRUE` and the first term of the independent variables is categorical, `rxLinMod` will compute and return a data frame with category counts. If there are no other independent variables, or if the `cubePredictions` argument is set to `TRUE`, the data frame will also contain predicted values. Let’s create a simple data frame to illustrate:

```
xfac1 <- factor(c(1,1,1,1,2,2,2,2,3,3,3,3), labels=c("One1", "Two1", "Three1"))
xfac2 <- factor(c(1,1,1,1,1,1,2,2,2,3,3,3), labels=c("One2", "Two2", "Three2"))
set.seed(100)
```

```
y <- as.integer(xfac1) + as.integer(xfac2)* 2 + rnorm(12)
myData <- data.frame(y, xfac1, xfac2)
```

If we estimate a simple linear regression of y on $xfac1$, the coefficients are equal to the within-group means:

```
myLinMod <- rxLinMod(y ~ xfac1, data = myData, cube = TRUE)
myLinMod

Call:
rxLinMod(formula = y ~ xfac1, data = myData, cube = TRUE)

Cube Linear Regression Results for: y ~ xfac1
Data: myData
Dependent variable(s): y
Total independent variables: 3
Number of valid observations: 12
Number of missing observations: 0

Coefficients:
              y
xfac1=One1    3.109302
xfac1=Two1    5.142086
xfac1=Three1  8.250260
```

In addition to the standard output, the returned object contains a *countDF* data frame with information on within-group means and counts in each category. In this simple case the within-group means are the same as the coefficients:

```
myLinMod$countDF

  xfac1      y Counts
1 One1 3.109302     4
2 Two1 5.142086     4
3 Three1 8.250260     4
```

Using *rxLinMod* with the *cube* option also allows us to compute conditional within-group means. For example:

```
myLinMod1 <- rxLinMod(y~xfac1 + xfac2, data = myData,
  cube = TRUE, cubePredictions = TRUE)
myLinMod1

Call:
rxLinMod(formula = y ~ xfac1 + xfac2, data = myData, cube = TRUE,
  cubePredictions=TRUE)

Cube Linear Regression Results for: y ~ xfac1 + xfac2
Data: myData
Dependent variable(s): y
Total independent variables: 6 (Including number dropped: 1)
Number of valid observations: 12
Number of missing observations: 0

Coefficients:
              y
xfac1=One1    7.725231
xfac1=Two1    8.833730
xfac1=Three1  8.942099
```

84 Fitted Values, Residuals, and Prediction

```
xfac2=One2    -4.615929
xfac2=Two2    -2.767359
xfac2=Three2   Dropped
```

If we look at the `countDF`, we will see the within-group means, conditional on the average value of the conditioning variable, `xfac2`:

```
myLinMod1$countDF

  xfac1      y Counts
1  One1 4.725427     4
2  Two1 5.833926     4
3 Three1 5.942295     4
```

For the variable `xfac2`, 50% of the observations have the value “One2”, 25% of the observations have the value “Two2”, and 25% have the value “Three2”. We can compute the weighted average of the coefficients for `xfac2` as:

```
myCoef <- coef(myLinMod1)
avexfac2c <- .5*myCoef[4] + .25*myCoef[5]
```

To compute the conditional within-group mean shown in the `countDF` for `xfac1` equal to “One1”, we add this to the coefficient computed for “One1”:

```
condMean1 <- myCoef[1] + avexfac2c
condMean1

xfac1=One1
4.725427
```

Conditional within-group means can also be computed using additional continuous independent variables.

7.6 Fitted Values, Residuals, and Prediction

When you fit a model with `lm` or any of the other core R model-fitting functions, you get back an object that includes as components both the fitted values for the response variable and the model residuals. For models fit with `rxLinMod` or other RevoScaleR functions, it is usually impractical to include these components, as they can be many megabytes in size. Instead, they are computed on demand using the `rxPredict` function. This function takes an `rxLinMod` object as its first argument, an input data set as its second argument, and an output data set as its third argument. If the input data set is the same as the data set used to fit the `rxLinMod` object, the resulting predictions are the fitted values for the model. If the input data set is a different data set (but one containing the same variable names used in fitting the `rxLinMod` object), the resulting predictions are true predictions of the response for the new data from the original model. In either case, residuals for the predicted values can be obtained by setting the flag `computeResiduals` to `TRUE`.

For example, we can draw from the 7% sample of the large airline data set (available from the Revolution Analytics web site) training and prediction data sets as follows (remember to customize the first line below for your own system):

```
bigDataDir <- "C:/Revolution/Data"
sampleAirData <- file.path(bigDataDir, "AirOnTime7Pct.xdf")
trainingDataFile <- "AirlineData06to07.xdf"
targetInfile <- "AirlineData08.xdf"
rxDataStep(sampleAirData, trainingDataFile, rowSelection = Year == 1999 |
  Year == 2000 | Year == 2001 | Year == 2002 | Year == 2003 |
  Year == 2004 | Year == 2005 | Year == 2006 | Year == 2007)
rxDataStep(sampleAirData, targetInfile, rowSelection = Year == 2008)
```

We can then fit a linear model with the training data and compute predicted values on the prediction data set as follows:

```
arrDelayLm2 <- rxLinMod(ArrDelay ~ DayOfWeek + UniqueCarrier + Dest,
  data = trainingDataFile)
rxPredict(arrDelayLm2, data = targetInfile, outData = targetInfile)
```

To see the first few rows of the result, use `rxGetInfo` as follows:

```
rxGetInfo(targetInfile, numRows = 5)
```

File name: C:\Revolution\Data\OutData\AirlineData08.xdf
Number of observations: 489792
Number of variables: 14
Number of blocks: 2
Compression type: zlib
Data (5 rows starting with row 1):

	Year	DayOfWeek	UniqueCarrier	Origin	Dest	CRSDepTime	DepDelay	TaxiOut	TaxiIn
1	2008	Mon	9E	ATL	HOU	7.250000	-2	14	5
2	2008	Mon	9E	ATL	HOU	7.250000	-2	16	2
3	2008	Sat	9E	ATL	HOU	7.250000	0	18	5
4	2008	Wed	9E	ATL	SRQ	12.666667	6	20	4
5	2008	Sat	9E	HOU	ATL	9.083333	-9	15	9

	ArrDelay	ArrDel15	CRSElapsedTime	Distance	ArrDelay_Pred
1	-15	FALSE	140	696	8.981548
2	0	FALSE	140	696	8.981548
3	0	FALSE	140	696	5.377572
4	6	FALSE	86	445	7.377952
5	-23	FALSE	120	696	6.552242

7.7 Prediction Standard Errors, Confidence Intervals, and Prediction Intervals

You can also use `rxPredict` to obtain prediction standard errors, provided you have included the variance-covariance matrix in the original `rxLinMod` fit. If you choose to compute the prediction standard errors, you can also obtain either of two kinds of intervals: *confidence intervals* that for a given confidence level tell us how confident we are that the expected value is within the given interval, and *prediction intervals* that specify, for a given confidence level, how likely future observations are to fall within the interval given what has already been observed. Standard error computations are computationally intensive, and they may become prohibitive

86 Prediction Standard Errors, Confidence Intervals, and Prediction Intervals

on large data sets with a large number of predictors. To illustrate the computation, we start with a small data set, using the example on page 132 of *Introduction to the Practice of Statistics*, (5th Edition)(Moore & McCabe, 2006). The predictor, *neaInc*, is the increase in “non-exercise activity” in response to an increase in caloric intake. The response, *fatGain*, is the associated increase in fat. We first read in the data and create a data frame to use in our analysis:

```
neaInc <- c(-94, -57, -29, 135, 143, 151, 245, 355, 392, 473, 486, 535, 571,
           580, 620, 690)
fatGain <- c( 4.2, 3.0, 3.7, 2.7, 3.2, 3.6, 2.4, 1.3, 3.8, 1.7, 1.6, 2.2, 1.0,
            0.4, 2.3, 1.1)
ips132df <- data.frame(neaNc = neaInc, fatGain=fatGain)
```

Next we fit the linear model with *rxLinMod*, setting *covCoef*=TRUE to ensure we have the variance-covariance matrix in our model object:

```
ips132lm <- rxLinMod(fatGain ~ neaInc, data=ips132df, covCoef=TRUE)
```

Now we use *rxPredict* to obtain the fitted values, prediction standard errors, and confidence intervals. By setting *writeModelVars* to TRUE, the variables used in the model will also be included in the output data set. In this first example, we obtain confidence intervals:

```
ips132lmPred <- rxPredict(ips132lm, data=ips132df, computeStdErrors=TRUE,
                          interval="confidence", writeModelVars = TRUE)
```

The standard errors are by default put into a variable named by concatenating the name of the response variable with an underscore and the string “StdErr”:

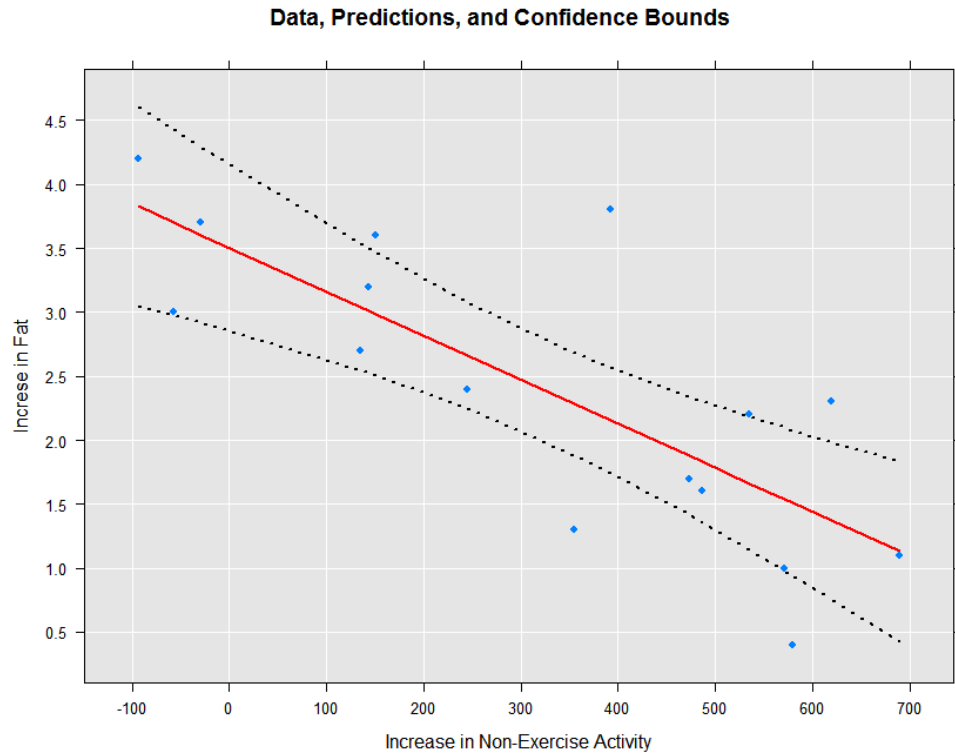
```
ips132lmPred$fatGain_StdErr

[1] 0.3613863 0.3381111 0.3209344 0.2323853 0.2288433 0.2254018 0.1941840
[8] 0.1863180 0.1915656 0.2151569 0.2202366 0.2418892 0.2598922 0.2646223
[15] 0.2865818 0.3279390
```

We can view the original data, the fitted prediction line, and the confidence intervals as follows:

```
rxLinePlot(fatGain + fatGain_Pred + fatGain_Upper + fatGain_Lower ~ neaInc,
           data = ips132lmPred, type = "b",
           lineStyle = c("blank", "solid", "dotted", "dotted"),
           lineColor = c(NA, "red", "black", "black"),
           symbolStyle = c("solid circle", "blank", "blank", "blank"),
           title = "Data, Predictions, and Confidence Bounds",
           xTitle = "Increase in Non-Exercise Activity",
           yTitle = "Increase in Fat", legend = FALSE)
```

The resulting plot is shown below:

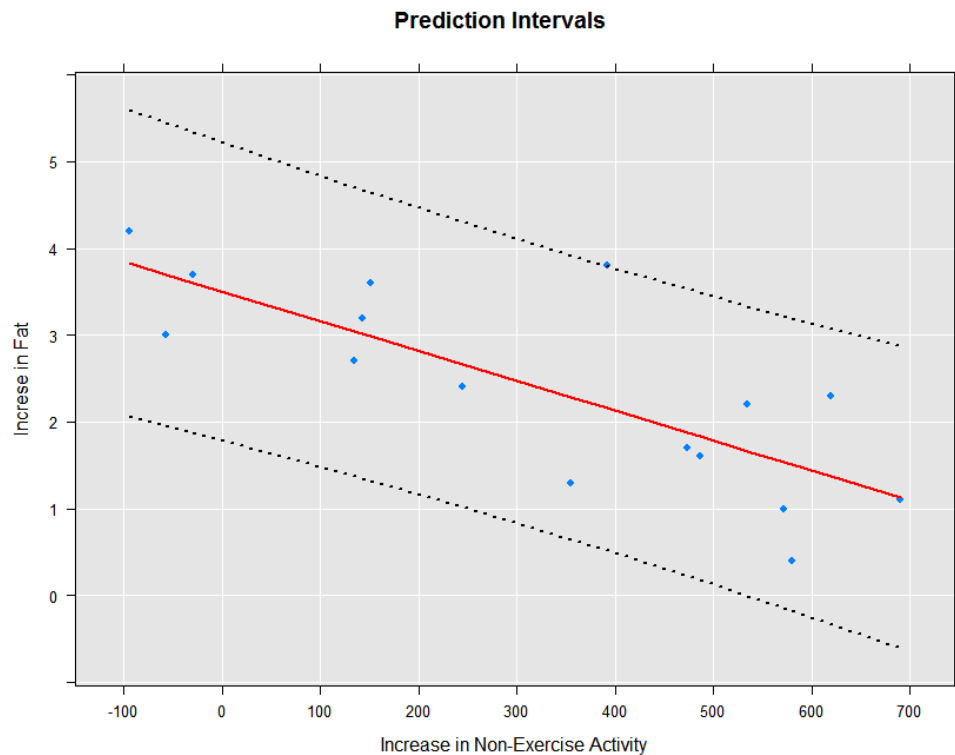


The prediction intervals can be obtained and plotted as follows:

```
ips132lmPred2 <- rxPredict(ips132lm, data=ips132df, computeStdErrors=TRUE,
  interval="prediction", writeModelVars = TRUE)
rxLinePlot(fatGain + fatGain_Pred + fatGain_Upper + fatGain_Lower ~ neaInc,
  data = ips132lmPred2, type = "b",
  lineStyle = c("blank", "solid", "dotted", "dotted"),
  lineColor = c(NA, "red", "black", "black"),
  symbolStyle = c("solid circle", "blank", "blank", "blank"),
  title = "Prediction Intervals",
  xTitle = "Increase in Non-Exercise Activity",
  yTitle = "Increase in Fat", legend = FALSE)
```

88 Prediction Standard Errors, Confidence Intervals, and Prediction Intervals

The resulting plot is shown below:



We can fit the prediction standard errors on our big airline regression model if we first refit it with `covCoef=TRUE`:

```
arrDelayLmVC <- rxLinMod(ArrDelay ~ DayOfWeek + UniqueCarrier + Dest,  
  data = trainingDataFile, covCoef=TRUE)
```

We can then obtain the prediction standard errors and a confidence interval as before:

```
rxPredict(arrDelayLmVC, data = targetInfile, outData = targetInfile,  
  computeStdErrors=TRUE, interval = "confidence", overwrite=TRUE)
```

We can then look at the first few lines of `targetInfile` to see the first few predictions and standard errors:

```
rxGetInfo(targetInfile, numRows=10)
```

File name: C:\YourOutputPath\AirlineData08.xdf

Number of observations: 489792

Number of variables: 17

Number of blocks: 2

Compression type: zlib

Data (10 rows starting with row 1):

	Year	DayOfWeek	UniqueCarrier	Origin	Dest	CRSDepTime	DepDelay	TaxiOut	TaxiIn
1	2008	Mon	9E	ATL	HOU	7.250000	-2	14	5
2	2008	Mon	9E	ATL	HOU	7.250000	-2	16	2
3	2008	Sat	9E	ATL	HOU	7.250000	0	18	5
4	2008	Wed	9E	ATL	SRQ	12.666667	6	20	4
5	2008	Sat	9E	HOU	ATL	9.083333	-9	15	9

6	2008	Mon	9E	ATL	IAH	16.416666	-3	25	4
7	2008	Thur	9E	IAH	ATL	18.500000	9	12	7
8	2008	Sat	9E	IAH	ATL	18.500000	-3	19	8
9	2008	Mon	9E	IAH	ATL	18.500000	-5	16	6
10	2008	Thur	9E	ATL	IAH	13.250000	-2	15	7
	ArrDelay	ArrDel15	CRSElapsedTime	Distance	ArrDelay_Pred	ArrDelay_StdErr			
1	-15	FALSE	140	696	8.981548	0.3287748			
2	0	FALSE	140	696	8.981548	0.3287748			
3	0	FALSE	140	696	5.377572	0.3293198			
4	6	FALSE	86	445	7.377952	0.6380760			
5	-23	FALSE	120	696	6.552242	0.2838803			
6	-11	FALSE	145	689	7.530244	0.2952031			
7	-8	FALSE	125	689	12.168922	0.2833356			
8	-16	FALSE	125	689	6.552242	0.2838803			
9	-24	FALSE	125	689	10.156218	0.2833283			
10	21	TRUE	130	689	9.542948	0.2952058			
	ArrDelay_Lower	ArrDelay_Upper							
1	8.337161	9.625935							
2	8.337161	9.625935							
3	4.732117	6.023028							
4	6.127346	8.628559							
5	5.995847	7.108638							
6	6.951657	8.108832							
7	11.613594	12.724249							
8	5.995847	7.108638							
9	9.600905	10.711532							
10	8.964355	10.121540							

7.8 Stepwise Variable Selection

Stepwise linear regression is an algorithm that helps you determine which variables are most important to a regression model. You provide a minimal, or lower, model formula and a maximal, or upper, model formula, and using forward selection, backward elimination, or bidirectional search, the algorithm determines the model formula that provides the best fit based on an AIC selection criterion.

In SAS, stepwise linear regression is implemented through PROC REG. In open source R, it is implemented through the function `step`. The problem with using the function `step` in R is that the size of the data set that can be analyzed is severely limited by the requirement that all computations must be done in memory.

RevoScaleR provides an implementation of stepwise linear regression that is not constrained by the use of "in-memory" algorithms. Stepwise linear regression in RevoScaleR is implemented by the functions `rxLinMod` and `rxStepControl`.

Stepwise linear regression begins with an initial model of some sort. Consider for example the airline training data set `AirlineData06to07.xdf` we created in section 7.6:

```
rxGetVarInfo(trainingDataFile)

Var 1: Year, Type: integer, Low/High: (1999, 2007)
Var 2: DayOfWeek
      7 factor levels: Mon Tues Wed Thur Fri Sat Sun
```

90 Stepwise Variable Selection

```
Var 3: UniqueCarrier
      30 factor levels: AA US AS CO DL ... OH F9 YV 9E VX
Var 4: Origin
      373 factor levels: JFK LAX HNL OGG DFW ... IMT ISN AZA SHD LAR
Var 5: Dest
      377 factor levels: LAX HNL JFK OGG DFW ... ESC IMT ISN AZA SHD
Var 6: CRSDepTime, Type: numeric, Storage: float32, Low/High: (0.0000, 23.9833)
Var 7: DepDelay, Type: integer, Low/High: (-1199, 1930)
Var 8: TaxiOut, Type: integer, Low/High: (0, 1439)
Var 9: TaxiIn, Type: integer, Low/High: (0, 1439)
Var 10: ArrDelay, Type: integer, Low/High: (-926, 1925)
Var 11: ArrDel15, Type: logical, Low/High: (0, 1)
Var 12: CRSElapsedTime, Type: integer, Low/High: (-34, 1295)
Var 13: Distance, Type: integer, Low/High: (11, 4962)
```

We are interested in fitting a model that will predict arrival delay (*ArrDelay*) as a function of some of the other variables. To keep things simple, we'll start with our by now familiar model of arrival delay as a function of *DayOfWeek* and *CRSDepTime*:

```
initialModel <- rxLinMod(ArrDelay ~ DayOfWeek + CRSDepTime,
  data = trainingDataFile)
initialModel

Call:
rxLinMod(formula = ArrDelay ~ DayOfWeek + CRSDepTime, data = trainingDataFile)

Linear Regression Results for: ArrDelay ~ DayOfWeek + CRSDepTime
File name:

C:\Users\richcalaway\Documents\Revolution\LMPrediction\AirlineData06to07.xdf
Dependent variable(s): ArrDelay
Total independent variables: 9 (Including number dropped: 1)
Number of valid observations: 3945964
Number of missing observations: 98378

Coefficients:
              ArrDelay
(Intercept)  -4.91416806
DayOfWeek=Mon   0.49172258
DayOfWeek=Tues -1.41878850
DayOfWeek=Wed   0.09677481
DayOfWeek=Thur  2.52841304
DayOfWeek=Fri   3.29474667
DayOfWeek=Sat  -2.86217838
DayOfWeek=Sun      Dropped
CRSDepTime      0.86703378
```

The question is, can we improve this model by adding more predictors? If so, which ones? To use stepwise selection in *RevoScaleR*, you add the *variableSelection* argument to your call to *rxLinMod*. The *variableSelection* argument is a list, most conveniently created by using the *rxStepControl* function. Using *rxStepControl*, you specify the method (the default, "stepwise", specifies a bidirectional search), the scope (lower and upper formulas for the search), and various control parameters. With our model, we first want to try some more numeric predictors, so we specify our model as follows:

```
airlineStepModel <- rxLinMod(ArrDelay ~ DayOfWeek + CRSDepTime,
  data = trainingDataFile,
```

```

variableSelection = rxStepControl(method="stepwise",
  scope = ~ DayOfWeek + CRSDepTime + CRSElapsedTime +
    Distance + TaxiIn + TaxiOut ))

Call:
rxLinMod(formula = ArrDelay ~ DayOfWeek + CRSDepTime, data = trainingDataFile,
  variableSelection = rxStepControl(method = "stepwise", scope = ~DayOfWeek +
    CRSDepTime + CRSElapsedTime + Distance + TaxiIn + TaxiOut))

Linear Regression Results for: ArrDelay ~ DayOfWeek + CRSDepTime +
  CRSElapsedTime + Distance + TaxiIn + TaxiOut
File name:

C:\Users\richcalaway\Documents\Revolution\LMPrediction\AirlineData06to07.xdf
Dependent variable(s): ArrDelay
Total independent variables: 13 (Including number dropped: 1)
Number of valid observations: 3945964
Number of missing observations: 98378

Coefficients:
ArrDelay
(Intercept)    -9.87474950
DayOfWeek=Mon    0.09830827
DayOfWeek=Tues  -1.81998781
DayOfWeek=Wed   -0.62761606
DayOfWeek=Thur   1.53941124
DayOfWeek=Fri    2.45565510
DayOfWeek=Sat   -2.20111789
DayOfWeek=Sun      Dropped
CRSDepTime       0.75995355
CRSElapsedTime  -0.20256102
Distance         0.02135381
TaxiIn           0.16194266
TaxiOut          0.99814931

```

7.8.1 Methods of Variable Selection

Three methods of variable selection are supported by rxLinMod:

- *"forward"*: starting from the minimal model, variables are added one at a time until no additional variable satisfies the selection criterion, or until the maximal model is reached.
- *"backward"*: starting from the maximal model, variables are removed one at a time until the removal of another variable won't satisfy the selection criterion, or until the minimal model is reached.
- *"stepwise"* (the default): a combination of forward and backward selection, in which variables are added to the minimal model, but at each step, the model is reanalyzed to see if any variables that have been added are candidates for deletion from the current model.

You specify the desired method by supplying a named component *"method"* in the list supplied for the *variableSelection* argument, or by specifying the *method* argument in a call to *rxStepControl* that is then passed as the *variableSelection* argument.

7.8.2 Specifying Model Scope

You use the `scope` argument in `rxStepControl` (or a named component `"scope"` in the list supplied for the `variableSelection` argument) to specify which variables should be considered for inclusion in the final model selection and which should be ignored. Whether you specify a separate value for `scope` also determines which models the algorithm will try next.

You can specify the scope as a simple formula (which will be treated as the upper bound, or maximal model), or as a named list with components `"lower"` and `"upper"` (either of which may be missing). For example, to analyze the iris data with a minimal model involving the single predictor `Sepal.Width` and a maximal model involving `Sepal.Width`, `Petal.Length`, and the interaction between `Petal.Width` and `Species`, we can specify our variable selection model as follows:

```
form <- Sepal.Length ~ Sepal.Width + Petal.Length
scope <- list(
  lower = ~ Sepal.Width,
  upper = ~ Sepal.Width + Petal.Length + Petal.Width * Species)

varsel <- rxStepControl(method = "stepwise", scope = scope)
rxlm.step <- rxLinMod(form, data = iris, variableSelection = varsel,
  verbose = 1, dropMain = FALSE, coefLabelStyle = "R")
```

In general, the models considered are determined from the `scope` argument as follows:

- `"lower"` scope only: All models considered will include this lower model up to the base model specified in the `formula` argument to `rxLinMod`.
- `"upper"` scope only: All models considered will include the terms in the base model (specified by the `formula` argument to `rxLinMod`), plus additional terms as specified in the upper scope.
- Both `"lower"` and `"upper"` scope supplied: All models considered will include at least the terms in the lower scope and additional terms will be added using terms from the upper scope until the stopping criterion is reached or the full upper scope model is selected.
- No scope supplied: The minimal model will include just the intercept term and the maximal model will include at most the terms specified in the base model.

(This convention is identical to that used by R's `step` function.)

7.8.3 Specifying the Selection Criterion

By default, variable selection is determined using the Akaike Information Criterion, or AIC; this is the R standard. If you want a stepwise selection that is more SAS-like, you can specify `stepCriterion="SigLevel"`. If this is set, `rxLinMod` uses either an F-test (default) or Chi-square test to determine whether to add or drop terms from the model. You can specify which

test to perform using the `test` argument. For example, we can refit our airline model using the `SigLevel` step criterion with an F-test as follows:

```
airlineStepModelSigLevel <- rxLinMod(ArrDelay ~ DayOfWeek + CRSDepTime,
  data = trainingDataFile, variableSelection =
    rxStepControl( method = "stepwise", scope = ~ DayOfWeek +
      CRSDepTime + CRSElapsedTime + Distance + TaxiIn + TaxiOut,
      stepCriterion = "SigLevel" ))
```

In this case (and with many other well-behaved models), the results of using the `SigLevel` step criterion are identical to those using AIC.

You can control the significance levels for adding and dropping models using the `maxSigLevelToAdd` and `minSigLevelToDrop`. The `maxSigLevelToAdd` specifies a significance level below which a variable can be considered for inclusion in the model; the `minSigLevelToDrop` specifies a significance level above which a variable currently included can be considered for dropping. By default, for `method="stepwise"`, the both levels are set to .15. We can tighten the selection criterion to the .10 level as follows:

```
airlineStepModelSigLevel.10 <- rxLinMod(ArrDelay ~ DayOfWeek + CRSDepTime,
  data = trainingDataFile, variableSelection =
    rxStepControl( method = "stepwise", scope = ~ DayOfWeek +
      CRSDepTime + CRSElapsedTime + Distance + TaxiIn + TaxiOut,
      stepCriterion = "SigLevel",
      maxSigLevelToAdd=.10, minSigLevelToDrop=.10))
```

7.9 Fixed-Effects Models

Fixed-effects models are commonly associated with studies in which multiple observations are recorded for each test subject, for example, yearly observations of median housing price by city, or measurements of tensile strength from samples of steel rods by batch. To fit such a model with `rxLinMod`, include a factor variable specifying the subject (the cities, or the batch identifier) as the first predictor, and specify `cube=TRUE` to use a partitioned inverse and omit the intercept term.

For example, the `MASS` library contains the data set `petrol`, which consists of measurements of the yield of a certain refining process with possible predictors including specific gravity, vapor pressure, ASTM 10% point, and volatility measured as the ASTM endpoint for 10 samples of crude oil. Following Venables and Ripley (2002), we first scale the numeric predictors, then fit the fixed-effects model:

```
library(MASS)
Petrol <- petrol
Petrol[,2:5] <- scale(Petrol[,2:5], scale=F)
rxLinMod(Y ~ No + EP, data=Petrol, cube=TRUE)

Call:
rxLinMod(formula = Y ~ No + EP, data = Petrol, cube = TRUE)
```

94 Least Squares Dummy Variable (LSDV) Models

```
Cube Linear Regression Results for: Y ~ No + EP
Data: Petrol
Dependent variable(s): Y
Total independent variables: 11
Number of valid observations: 32
Number of missing observations: 0

Coefficients:
               Y
No=A 32.5493917
No=B 24.2746407
No=C 27.7820456
No=D 21.1541642
No=E 21.5191269
No=F 20.4355218
No=G 15.0359067
No=H 13.0630467
No=I  9.8053871
No=J  4.4360767
EP    0.1587296
```

7.10 Least Squares Dummy Variable (LSDV) Models

RevoScaleR is capable of estimating huge models where fixed effects are estimated by dummy variables, that is, binary variables set to 1 or TRUE if the observation is in a particular category. Creation of these dummy variables is often accomplished by interacting two or more factor variables using “:” in the formula. If the first term in an rxLinMod (or rxLogit) model is purely categorical and the “cube” argument is set to TRUE, the estimation uses a partitioned inverse to save on computation time and memory.

7.10.1 A Quick Review of Interacting Factors

First, let’s do a quick, cautionary review of interacting factor variables by experimenting with a small made-up data set.

```
set.seed(50)
income <- rep(c(1000,1500,2500,4000), each=5) + 100*rnorm(20)
race <- rep(c("Black","White"), each=10)
sex <- rep(c("Female", "Male"), each=5)
sex <- c(sex,sex)
myData <- data.frame(income, race, sex)
```

The data set has 20 observations with three variables: a numeric variable income and two factor variables representing race and sex. There are three easy ways to compute the within group means of every combination of age and race using RevoScaleR. First, we can use

rxSummary:

```
rxSummary(income~race:sex, data=myData)
```

which includes in its output:

	Category	race	sex	Means	StdDev	Min
income for race=Black,		Black	Female	970.7522	98.01983	827.2396

```
income for race=White, sex=Female White Female 2501.8303 47.10861 2450.1364
income for race=Black, sex=Male Black Male 1480.4378 92.29320 1355.4250
income for race=White, sex=Male White Male 3944.5127 40.82421 3883.3983
```

Second, we could use `rxCube`:

```
rxCube(income~race:sex, data=myData)
```

which includes in its output:

	race	sex	income	Counts
1	Black	Female	970.7522	5
2	White	Female	2501.8303	5
3	Black	Male	1480.4378	5
4	White	Male	3944.5127	5

Or, we can use `rxLinMod` with `cube=TRUE`. The intercept is automatically omitted, and four dummy variables are created from the two factors: one for each combination of race and sex. The coefficients are simply the within group means:

```
summary(rxLinMod(income~race:sex, cube=TRUE, data=myData))
```

which includes in its output:

Coefficients:							Counts
	Estimate	Std. Error	t value	Pr(> t)			
race=Black, sex=Female	970.75	33.18	29.26	2.66e-15 ***		5	
race=White, sex=Female	2501.83	33.18	75.41	2.22e-16 ***		5	
race=Black, sex=Male	1480.44	33.18	44.62	2.22e-16 ***		5	
race=White, sex=Male	3944.51	33.18	118.90	2.22e-16 ***		5	

The same model could be estimated using: `lm(income~race:sex + 0, data=myData)`. Below we will refer to these within group means as *MeanIncBlackFemale*, *MeanIncWhiteFemale*, *MeanIncBlackMale*, and *MeanIncWhiteMale*.

If we add an intercept, we will encounter perfect multicollinearity and one of the coefficients will be dropped. The intercept is then the mean of a reference group and the other coefficients represent the differences or contrasts between the within group means and the reference group. For example,

```
lm(income~race:sex, data=myData)
```

produces:

Coefficients:		
(Intercept)	raceBlack:sexFemale	raceWhite:sexFemale
3945	-2974	-1443
raceBlack:sexMale	raceWhite:sexMale	
-2464	NA	

We can see that the dummy variable for white males was dropped; white males are the reference group and the Intercept is equal to *MeanIncWhiteMale*. The other coefficients

96 Least Squares Dummy Variable (LSDV) Models

represent contrasts from the reference group, so for example, `raceBlack:sexFemale` is equal to $\text{MeanIncBlackFemale} - \text{MeanIncWhiteMale}$.

Another variation is to use "*" in the formula for factor crossing. Using `a*b` is equivalent to using `a + b + a:b`. For example:

```
lm(income~race*sex, data = myData)
```

which results in:

```
Coefficients:
      (Intercept)      raceWhite      sexMale  raceWhite:sexMale
           970.8          1531.1          509.7          933.0
```

Note that the dropping of coefficients in `rxLinMod` can be controlled to obtain the same results:

```
rxLinMod(income~race*sex, data = myData, dropFirst = TRUE, dropMain = FALSE)
```

Coefficients using this model can be more difficult to interpret, and in fact are highly dependent on the order of the factor levels. In this case, we see the following relationship between the estimated coefficients and the within group means

<i>(Intercept)</i>	<i>MeanIncBlackFemale</i>
<i>raceWhite</i>	<i>MeanIncWhiteFemale - MeanIncBlackFemale</i>
<i>sexMale</i>	<i>MeanIncBlackMale - MeanIncBlackFemale</i>
<i>raceWhite:sexMale</i>	<i>MeanIncWhiteMale - MeanIncWhiteFemale -</i> <i>MeanIncBlackMale + MeanIncBlackFemale</i>

If we set up our data slightly differently, we will get quite different results. Let's use the same income data but a different naming convention for the factors:

```
race <- rep(c("Black", "Caucasian"), each=10)
sex <- rep(c("Woman", "Man"), each=5)
sex <- c(sex, sex)
myData1 <- data.frame(income, race, sex)
```

Using the same model, `lm(income~race*sex, data=myData1)`, results in:

```
Coefficients:
      (Intercept)      raceCaucasian      sexWoman
           1480.4          2464.1          -509.7
raceCaucasian:sexWoman
           -933.0
```

With this superficial modification to the data, the coefficient for the dummy variable for White/Caucasian has jumped from \$1531 to \$2464. This is due to the change in reference group; in this model the White/Caucasian coefficient represents the difference between mean white and black male income, while in the previous model it was the difference between mean white and black female income. That is, the relationship between the within group means and the new coefficients are:

<i>(Intercept)</i>	<i>MeanIncBlackMale</i>
<i>raceCaucasian</i>	<i>MeanIncWhiteMale - MeanIncBlackMale</i>
<i>sexWoman</i>	<i>MeanIncBlackFemale - MeanIncBlackMale</i>
<i>raceWhite:sexMale</i>	<i>MeanIncWhiteFemale - MeanIncWhiteMale - MeanIncBlackFemale + MeanIncBlackMale</i>

Omitting the intercept provides yet another combination of results. For example, using `rxLinMod` with the original factor labeling:

```
rxLinMod(income~race*sex, data=myData, cube=TRUE)
```

results in:

```
Call:
rxLinMod(formula = income ~ race * sex, data = myData, cube = TRUE)

Cube Linear Regression Results for: income ~ race * sex
Data: myData
Dependent variable(s): income
Total independent variables: 8 (Including number dropped: 4)
Number of valid observations: 20
Number of missing observations: 0

Coefficients:
                                income
race=Black                    1480.4378
race=White                    3944.5127
sex=Female                   -1442.6824
sex=Male                      Dropped
race=Black, sex=Female       932.9968
race=White, sex=Female       Dropped
race=Black, sex=Male         Dropped
race=White, sex=Male         Dropped
```

<i>race=Black</i>	<i>MeanIncBlackMale</i>
<i>race=White</i>	<i>MeanIncWhiteMale</i>
<i>Sex=Female</i>	<i>MeanIncWhiteFemale - MeanIncWhiteMale</i>
<i>race=Black, sex=Female</i>	<i>(MeanIncBlackFemale - MeanIncBlackMale) - (MeanIncWhiteFemale - MeanIncWhiteMale)</i>

With large data sets it is common to estimate many interaction terms, and if some categories have zero counts, it may not even be obvious what the reference group is. Also note that setting `cube=TRUE` in the above model is of limited use: only the first term from the expanded expression (in this case *race*) is estimated using a partitioned inverse.

7.10.2 Using Dummy Variables in `rxLinMod`: Letting the Data Speak Example 2

In the Chapter 6, we looked at the `CensusWorkers.xdf` data set and examined the relationship between wage income and age. Now let's add another variable, and examine the relationship between wage income and sex and age.

98 Least Squares Dummy Variable (LSDV) Models

We can start with a simple dummy variable model, computing the mean wage income by sex:

```
censusWorkers <- file.path(rxGetOption("sampleDataDir"), "CensusWorkers.xdf")
rxLinMod(incwage~sex, data=censusWorkers, pweights="perwt", cube=TRUE)
```

which computes:

```
Call:
rxLinMod(formula = incwage ~ sex, data = censusWorkers, pweights = "perwt",
  cube = TRUE)

Cube Linear Regression Results for: incwage ~ sex
File name:
  C:\Revolution\R-Enterprise-7.0\R-
  3.0.2\library\RevoScaleR\SampleData\CensusWorkers.xdf
Probability weights: perwt
Dependent variable(s): incwage
Total independent variables: 2
Number of valid observations: 351121
Number of missing observations: 0

Coefficients:
              incwage
sex=Male    43472.71
sex=Female  26721.09
```

Similarly, we could look at a simple linear relationship between wage income and age:

```
linMod1 <- rxLinMod(incwage~age, data=censusWorkers, pweights="perwt")
summary(linMod1)
```

resulting in:

```
Call:
rxLinMod(formula = incwage ~ age, data = censusWorkers, pweights = "perwt")

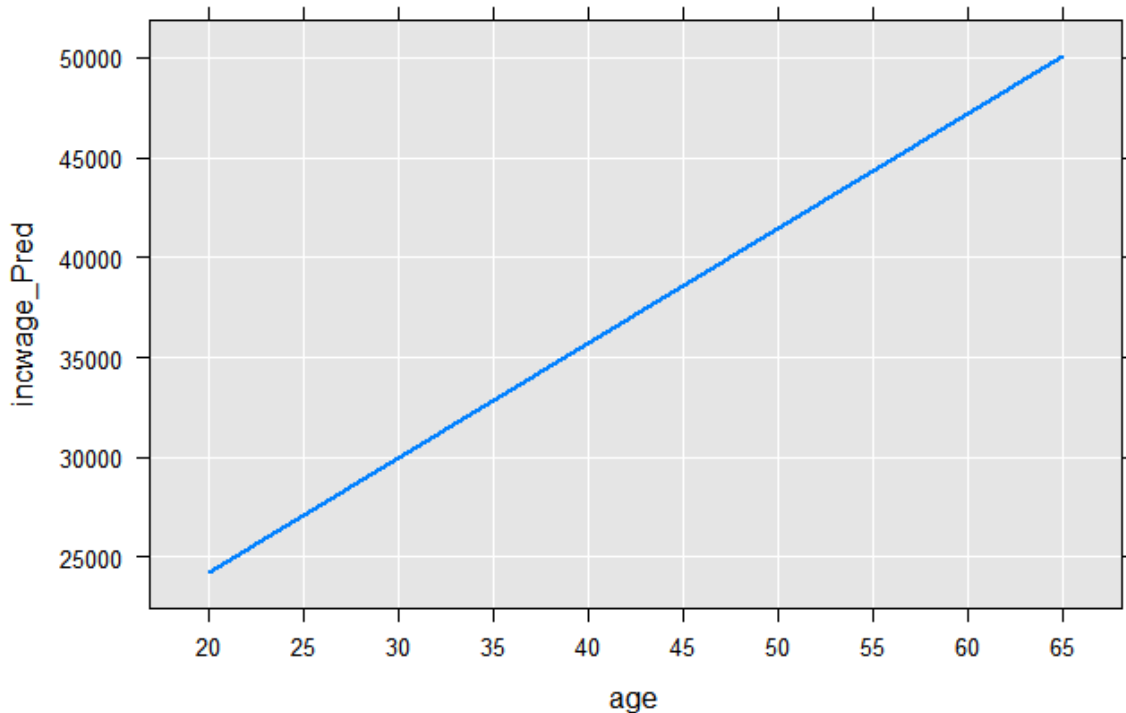
Linear Regression Results for: incwage ~ age
File name:
  C:\Revolution\R-Enterprise-7.0\R-
  3.0.2\library\RevoScaleR\SampleData\CensusWorkers.xdf
Probability weights: perwt
Dependent variable(s): incwage
Total independent variables: 2
Number of valid observations: 351121
Number of missing observations: 0

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept) 12802.980    247.963   51.63 2.22e-16 ***
age          572.980      5.947   96.35 2.22e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 180800 on 351119 degrees of freedom
Multiple R-squared:  0.02576
Adjusted R-squared:  0.02576
F-statistic:  9284 on 1 and 351119 DF,  p-value: < 2.2e-16
Condition number: 1
```

Computing the two end points on the regression line, we can plot it:

```
age <- c(20,65)
coefLinMod1 <- coef(linMod1)
incwage_Pred <- coefLinMod1[1] + age*coefLinMod1[2]
plotData1 <- data.frame(age, incwage_Pred)
rxLinePlot(incwage_Pred~age, data=plotData1)
```



The next typical step is to combine the two approaches by estimating separate intercepts for males and females:

```
linMod2 <- rxLinMod(incwage~sex+age, data = censusWorkers, pweights = "perwt",
  cube=TRUE)
summary(linMod2)
```

which results in:

```
Call:
rxLinMod(formula = incwage ~ sex + age, data = censusWorkers,
  pweights = "perwt", cube = TRUE)

Cube Linear Regression Results for: incwage ~ sex + age
File name:
  C:\Revolution\R-Enterprise-7.0\R-
  3.0.2\library\RevoScaleR\SampleData\CensusWorkers.xdf
Probability weights: perwt
Dependent variable(s): incwage
Total independent variables: 3
Number of valid observations: 351121
Number of missing observations: 0

Coefficients:
      Estimate Std. Error t value Pr(>|t|)      | Counts
```

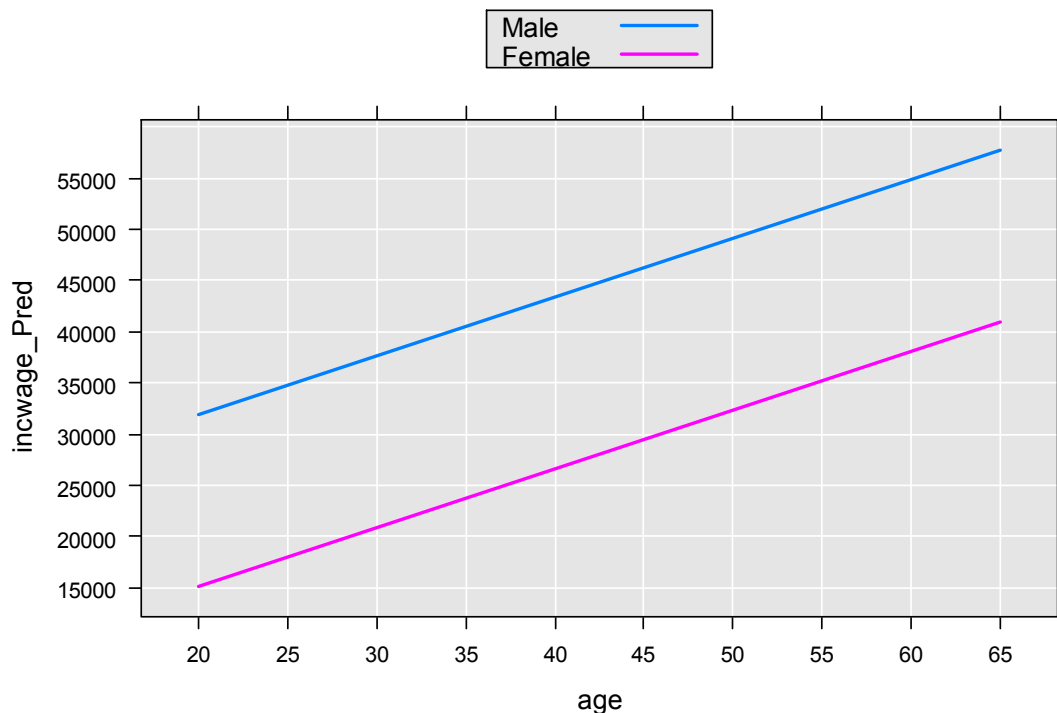
100 Least Squares Dummy Variable (LSDV) Models

```
sex=Male    20479.178    250.033    81.91 2.22e-16 *** | 3866542
sex=Female   3723.067    252.968    14.72 2.22e-16 *** | 3276744
age          573.232      5.816    98.56 2.22e-16 *** |
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 176800 on 351118 degrees of freedom
Multiple R-squared:  0.06804 (as if intercept included)
Adjusted R-squared:  0.06804
F-statistic: 1.282e+04 on 2 and 351118 DF,  p-value: < 2.2e-16
Condition number: 1
```

We will create a small sample data set with the same variables we use in `censusWorkers`, but with only four observations representing the high and low value of age for both sexes. Using the `rxPredict` function, the predicted values for each one of these sample observations is computed, which we then plot:

```
age <- c(20,65,20,65)
sex <- factor(rep(c(1, 2), each=2), labels=c("Male", "Female"))
perwt <- rep(1, times=4)
incwage <- rep(0, times=4)
plotData2 <- data.frame(age, sex, perwt, incwage)
plotData2p <- rxPredict(linMod2, data=plotData2, outData=plotData2)
rxLinePlot(incwage_Pred~age, groups=sex, data=plotData2p)
```



These types of models are often relaxed further by allowing both the slope and intercept to vary by group:

```
linMod3 <- rxLinMod(incwage~sex+sex:age, data = censusWorkers,
  pweights = "perwt", cube=TRUE)
```

```
summary(linMod3)
```

```
Call:
rxLinMod(formula = incwage ~ sex + sex:age, data = censusWorkers,
  pweights = "perwt", cube = TRUE)

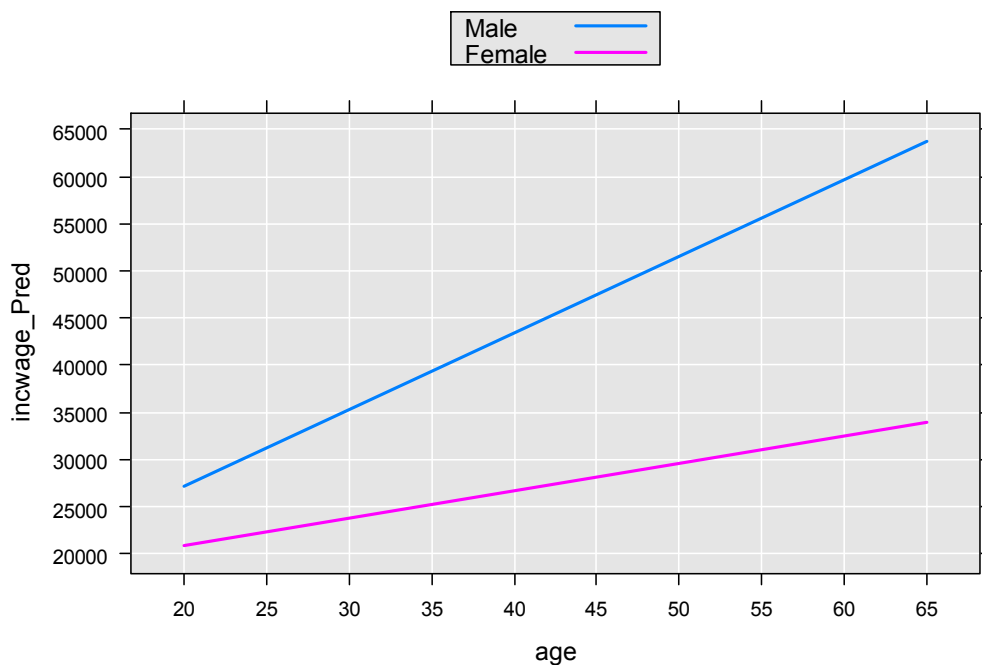
Cube Linear Regression Results for: incwage ~ sex + sex:age
File name:
  C:\Revolution\R-Enterprise-7.0\
R-3.0.2\library\RevoScaleR\SampleData\CensusWorkers.xdf
Probability weights: perwt
Dependent variable(s): incwage
Total independent variables: 4
Number of valid observations: 351121
Number of missing observations: 0

Coefficients:
              Estimate Std. Error t value Pr(>|t|)      | Counts
sex=Male      10783.449   328.871   32.79 2.22e-16 *** | 3866542
sex=Female    15131.422   356.806   42.41 2.22e-16 *** | 3276744
age for sex=Male    814.948     7.888  103.31 2.22e-16 *** |
age for sex=Female  288.876     8.556   33.76 2.22e-16 *** |
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 176300 on 351117 degrees of freedom
Multiple R-squared:  0.07343 (as if intercept included)
Adjusted R-squared:  0.07343
F-statistic:  9276 on 3 and 351117 DF,  p-value: < 2.2e-16
Condition number: 1.1764
```

Again getting predictions and plotting:

```
plotData3p <- rxPredict(linMod3, data=plotData2, outData=plotData2)
rxLinePlot(incwage_Pred~age, groups=sex, data=plotData3p)
```



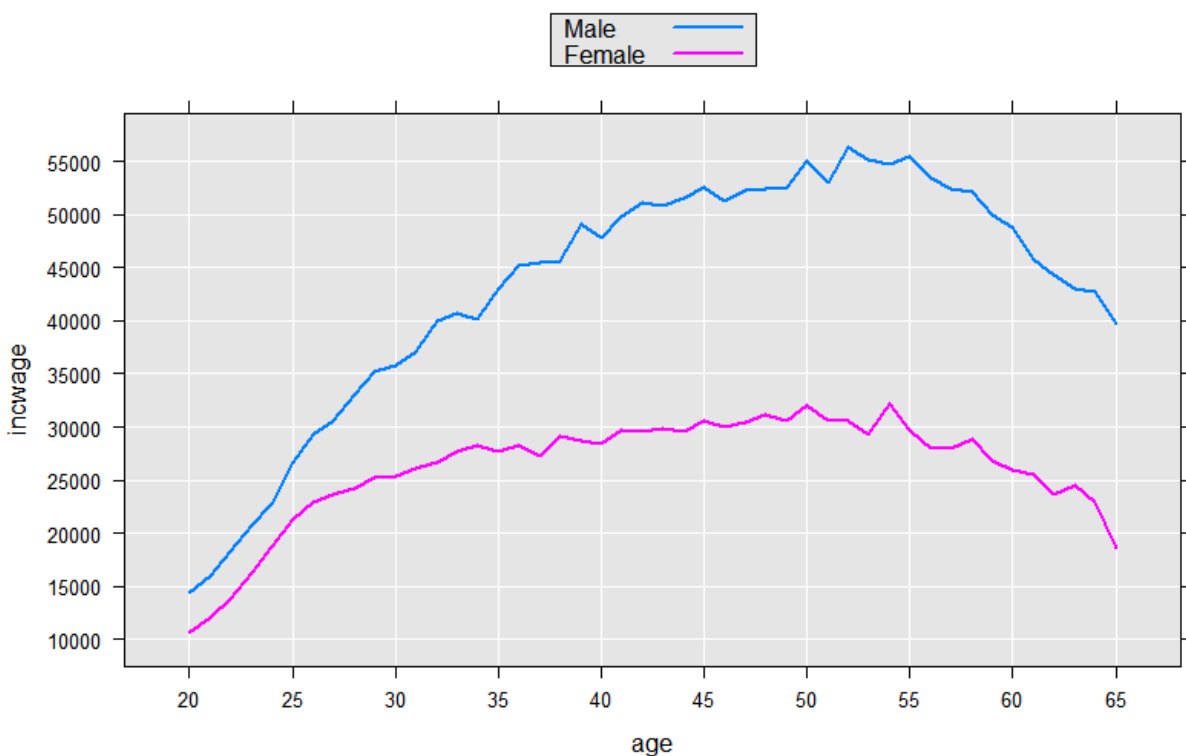
102 Least Squares Dummy Variable (LSDV) Models

We could continue the process, experimenting with functional forms for age. But, since we have many observations (and therefore many degrees of freedom), we can take advantage of the `F()` function available in `revoScaleR` to let the data speak for itself. As we saw in Chapter 6, the `F()` function creates a factor variable from a numeric variable “on-the-fly”, creating a level for every integer value. This allows us to compute and observe the shape of the functional form using a purely dummy variable model:

```
linMod4 <- rxLinMod(incwage~sex:F(age), data=censusWorkers, pweights="perwt",  
  cube=TRUE)
```

This model estimated a total of 92 coefficients, all for dummy variables representing every age in the range of 20 to 65 for each sex. To visually examine the coefficients we could add observations to our `plotData` data frame and use `rxPredict` to compute predicted values for each of the 92 groups represented, but since the model contains only dummy variables in an initial cube term, we can instead use the “counts” data frame returned with the `rxLinMod` object:

```
plotData4 <- linMod4$countDF  
# Convert the age factor variable back to an integer  
plotData4$age <- as.integer(levels(plotData4$F.age.)) [plotData4$F.age.]  
rxLinePlot(incwage~age, groups=sex, data=plotData4)
```



7.11 Intercept-Only Models

You may have seen intercept-only models fitted with R's `lm` function, where the model formula is of the form `response ~ 1`. In RevoScaleR these models should be fitted using `rxSummary`, because the intercept-only model simply returns the mean of the response. For example:

```
airlineDF <- rxDataStep(inData =
  file.path(rxGetOption("sampleDataDir"), "AirlineDemoSmall.xdf"))
lm(ArrDelay ~ 1, data = airlineDF)
```

```
Call:
lm(formula = ArrDelay ~ 1, data = airlineDF)
```

```
Coefficients:
(Intercept)
      11.32
```

```
rxSummary(~ ArrDelay, data = airlineDF)
```

```
Call:
rxSummary(formula = ~ArrDelay, data = airlineDF)
```

```
Summary Statistics Results for: ~ArrDelay
Data: airlineDF
Number of valid observations: 6e+05
Number of missing observations: 0
```

Name	Mean	StdDev	Min	Max	ValidObs	MissingObs
ArrDelay	11.31794	40.68854	-86	1490	582628	17372

Chapter 8.

Fitting Logistic Regression Models

Logistic regression is a standard tool for modeling data with a binary response variable. In R, you fit a logistic regression using the `glm` function, specifying a binomial family and the logit link function. In RevoScaleR, you can use `rxGlm` in the same way (see Chapter 9, [Fitting Generalized Linear Models](#)) or you can fit a logistic regression using the optimized `rxLogit` function; because this function is specific to logistic regression, you need not specify a family or link function.

As an example, consider the `kyphosis` data set in the `rpart` package. This data set consists of 81 observations of four variables (Age, Number, Kyphosis, Start) in children following corrective spinal surgery; it is used as the initial example of `glm` in the White Book (Chambers & Hastie, 1992). The variable Kyphosis reports the absence or presence of this deformity.

We can use `rxLogit` to model the probability that kyphosis is present as follows:

```
library(rpart)
rxLogit(Kyphosis ~ Age + Start + Number, data = kyphosis)
```

The following output is returned:

```
Logistic Regression Results for: Kyphosis ~ Age + Start + Number
Data: kyphosis
Dependent variable(s): Kyphosis
Total independent variables: 4
```



```
Number of valid observations: 81
Number of missing observations: 0
```

```
Coefficients:
              Kyphosis
(Intercept) -2.03693354
Age          0.01093048
Start       -0.20651005
Number       0.41060119
```

The same model can be fit with *glm* (or *rxGlm*) as follows:

```
glm(Kyphosis ~ Age + Start + Number, family = binomial, data = kyphosis)

Call: glm(formula = Kyphosis ~ Age + Start + Number, family = binomial,
data = kyphosis)

Coefficients:
(Intercept)      Age      Start      Number
    -2.03693     0.01093    -0.20651     0.41060

Degrees of Freedom: 80 Total (i.e. Null);  77 Residual
Null Deviance:      83.23
Residual Deviance: 61.38  AIC: 69.38
```

8.1 Stepwise Logistic Regression

Stepwise logistic regression is an algorithm that helps you determine which variables are most important to a logistic model. You provide a minimal, or lower, model formula and a maximal, or upper, model formula, and using forward selection, backward elimination, or bidirectional search, the algorithm determines the model formula that provides the best fit based on an AIC or significance level selection criterion.

RevoScaleR provides an implementation of stepwise logistic regression that is not constrained by the use of "in-memory" algorithms. Stepwise linear regression in RevoScaleR is implemented by the functions *rxLogit* and *rxStepControl*.

Stepwise logistic regression begins with an initial model of some sort. We can look at the kyphosis data again and start with a simpler model: *Kyphosis ~ Age*:

```
initModel <- rxLogit(Kyphosis ~ Age, data=kyphosis)
initModel

Logistic Regression Results for: Kyphosis ~ Age
Data: kyphosis
Dependent variable(s): Kyphosis
Total independent variables: 2
Number of valid observations: 81
Number of missing observations: 0

Coefficients:
              Kyphosis
(Intercept) -1.809351230
Age          0.005441758
```

We can specify a stepwise model using `rxLogit` and `rxStepControl` as follows:

```
KyphStepModel <- rxLogit(Kyphosis ~ Age,
  data = kyphosis,
  variableSelection = rxStepControl(method="stepwise",
    scope = ~ Age + Start + Number ))
KyphStepModel

Logistic Regression Results for: Kyphosis ~ Age + Start + Number
Data: kyphosis
Dependent variable(s): Kyphosis
Total independent variables: 4
Number of valid observations: 81
Number of missing observations: 0

Coefficients:
                Kyphosis
(Intercept) -2.03693354
Age           0.01093048
Start        -0.20651005
Number       0.41060119
```

The methods for variable selection (forward, backward, and stepwise), the definition of model scope, and the available selection criteria are all the same as for stepwise linear regression; see section 7.8 and the `rxStepControl` help file for more details.

8.2 Prediction

As described above for linear models, the objects returned by the `RevoScaleR` model-fitting functions do not include fitted values or residuals. We can obtain them, however, by calling `rxPredict` on our fitted model object, supplying the original data used to fit the model as the data to be used for prediction.

For example, consider the mortgage default example in Section 6 of the manual *RevoScaleR: Getting Started Guide*. For that example, we used ten input data files to create the data set used to fit the model. But suppose instead we use nine input data files to create the training data set and use the remaining data set for prediction. We can do that as follows (again, remember to modify the first line for your own system):

```
bigDataDir <- "C:/Revolution/Data"
mortCsvDataName <- file.path(bigDataDir, "mortDefault", "mortDefault")
trainingDataFileName <- "mortDefaultTraining"
mortCsv2009 <- paste(mortCsvDataName, "2009.csv", sep = "")
targetDataFileName <- "mortDefault2009.xdf"
ageLevels <- as.character(c(0:40))
yearLevels <- as.character(c(2000:2009))
colInfo <- list(list(name = "houseAge", type = "factor",
  levels = ageLevels), list(name = "year", type = "factor",
  levels = yearLevels))
append= FALSE
for (i in 2000:2008)
{
  importFile <- paste(mortCsvDataName, i, ".csv", sep = "")
```

```

    rxImport(inData = importFile, outFile = trainingDataFileName,
             colInfo = colInfo, append = append)
    append = TRUE
}

rxImport(inData = mortCsv2009, outFile = targetDataFileName,
         colInfo = colInfo)

```

We can then fit a logistic regression model to the training data and predict with the prediction data set as follows:

```

logitObj <- rxLogit(default ~ year + creditScore + yearsEmploy + ccDebt,
                   data = trainingDataFileName, blocksPerRead = 2, verbose = 1,
                   reportProgress=2)
rxPredict(logitObj, data = targetDataFileName,
          outData = targetDataFileName, computeResiduals = TRUE)

```

To view the first 30 rows of the output data file, use `rxGetInfo` as follows:

```
rxGetInfo(targetDataFileName, numRows = 30)
```

8.3 Prediction Standard Errors and Confidence Intervals

You can use `rxPredict` to obtain prediction standard errors and confidence intervals for models fit with `rxLogit` in the same way as for those fit with `rxLinMod`. The original model must be fit with `covCoef=TRUE`:

```

logitObj2 <- rxLogit(default ~ year + creditScore + yearsEmploy + ccDebt,
                    data = trainingDataFileName, blocksPerRead = 2, verbose = 1,
                    reportProgress=2, covCoef=TRUE)

```

You then specify `computeStdErr=TRUE` to obtain prediction standard errors; if this is TRUE, you can also specify `interval="confidence"` to obtain a confidence interval:

```

rxPredict(logitObj2, data = targetDataFileName,
          outData = targetDataFileName, computeStdErr = TRUE,
          interval = "confidence", overwrite=TRUE)

```

The first ten lines of the file with predictions can be viewed as follows:

```

rxGetInfo(targetDataFileName, numRows=10)

File name: C:\Users\yourname\Documents\Revolution\mortDefault2009.xdf
Number of observations: 1e+06
Number of variables: 10
Number of blocks: 2
Compression type: zlib
Data (10 rows starting with row 1):
  creditScore houseAge yearsEmploy ccDebt year default default_Pred
1         617         20          8   4410 2009          0 6.620773e-06
2         623         11          7   5609 2009          0 4.610861e-05
3         758         17          4   7250 2009          0 4.259884e-04
4         687         22          5   3761 2009          0 3.770789e-06
5         663         15          6   6746 2009          0 2.312827e-04
6         676         10          2   7106 2009          0 1.092593e-03

```

108 Prediction Standard Errors and Confidence Intervals

7	721	23	2	2280	2009	0	8.515912e-07
8	680	18	7	2831	2009	0	6.011109e-07
9	734	9	5	3867	2009	0	3.144299e-06
10	688	16	8	6238	2009	0	5.350031e-05
	default_StdErr	default_Lower	default_Upper				
1	3.143695e-07	6.032422e-06	7.266507e-06				
2	1.953612e-06	4.243427e-05	5.010109e-05				
3	1.594783e-05	3.958500e-04	4.584203e-04				
4	1.739047e-07	3.444893e-06	4.127516e-06				
5	8.733193e-06	2.147838e-04	2.490486e-04				
6	3.952975e-05	1.017797e-03	1.172880e-03				
7	4.396314e-08	7.696409e-07	9.422675e-07				
8	3.091885e-08	5.434655e-07	6.648706e-07				
9	1.469334e-07	2.869109e-06	3.445883e-06				
10	2.224102e-06	4.931401e-05	5.804197e-05				

Chapter 9.

Fitting Generalized Linear Models

Generalized linear models (GLM) are a framework for a wide range of analyses. They relax the assumptions for a standard linear model in two ways. First, a functional form can be specified for the conditional mean of the predictor. This is referred to as the “link” function. Second, you can specify a distribution for the response variable. The `rxGlm` function in `RevoScaleR` provides the ability to estimate generalized linear models on large data sets.

The following family/link combinations are implemented in C++ for performance enhancements: binomial/logit, gamma/log, poisson/log, and Tweedie. Other family/link combinations use a combination of C++ and R code. Any valid R family object that can be used with `glm()` can be used with `rxGlm()`, including those that are user-defined. The following table shows all of the supported family/link combinations (in addition to user-defined):

Family	Default Link Function	Other Available Link Functions
binomial	"logit"	"probit", "cauchit", "log", "cloglog"
gaussian	"identity"	"log", "inverse"
Gamma	"inverse"	"identity", "log"
inverse.gaussian	"1/mu^2"	"inverse", "identity", "log"
poisson	"log"	"identity", "sqrt"
quasi	"identity" with variance = "constant"	"logit", "probit", "cloglog", "inverse", "log", "1/mu^2", "sqrt"
quasibinomial	"logit"	Same as binomial, but dispersion parameter not fixed at one
quasipoisson	"log"	Same as poisson, but dispersion parameter not fixed at one
rxTweedie	requires arguments instead of link function	

9.1 A Simple Example Using the Poisson Family

The Poisson family is used to estimate models of count data. Examples from the literature include the following types of response variables:

- Number of drinks on a Saturday night
- Number of bacterial colonies in a Petri dish
- Number of children born to married women
- Number of credit cards a person has

We'll start with a simple example from Kabacoff's **R in Action** book (2011), using data provided with the **robust** R package. The data are from a placebo-controlled clinical trial of 59 epileptics. Patients with partial seizures were enrolled in a randomized clinical trial of the anti-epileptic drug, progabide. Counts of epileptic seizures were recorded during the trial. The data set also includes a baseline 8-week seizure count and the age of the patient.

To access this data, first make sure the **robust** package is installed, then use the `data` command to load the the data frame:

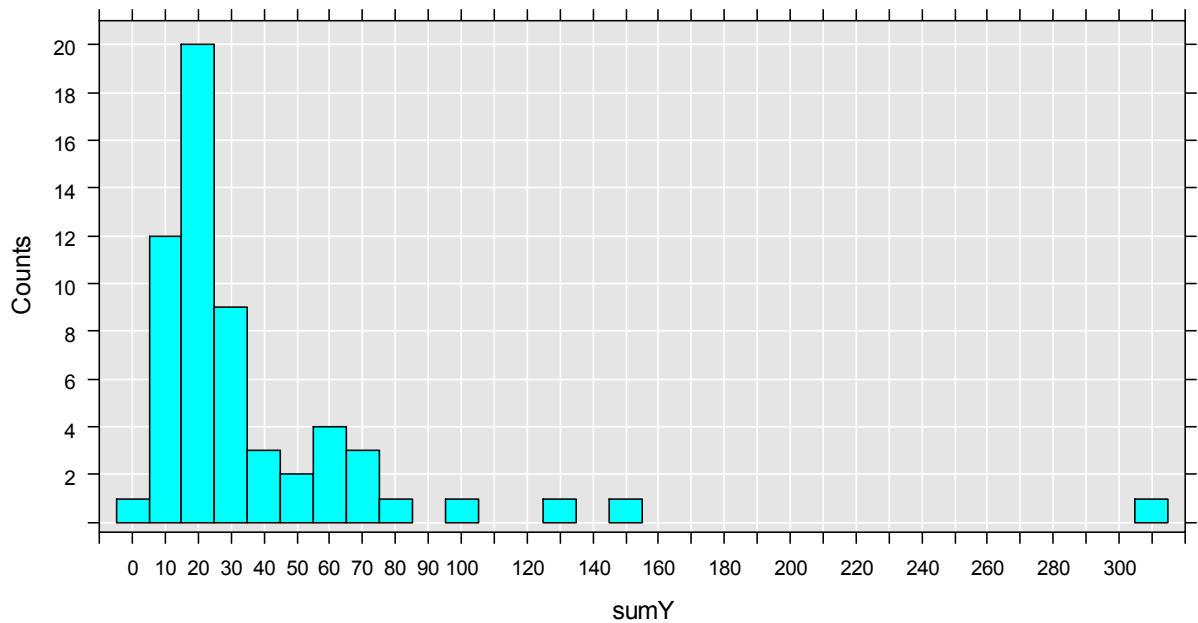
```
data(breslow.dat, package = "robust")
```

First, let's get some basic information on the data set, then draw a histogram of the sumY variable, containing the total count of seizures during the trial.

```
rxGetInfo(breslow.dat, getVarInfo = TRUE)
rxHistogram(~sumY, numBreaks = 25, data = breslow.dat)
```

The data set has 59 observations, and 12 variables. The variables of interest are *Base*, *Age*, *Trt*, and *sumY*.

```
Data frame: breslow.dat
Number of observations: 59
Number of variables: 12
Variable information:
Var 1: ID, Type: integer, Low/High: (101, 238)
Var 2: Y1, Type: integer, Low/High: (0, 102)
Var 3: Y2, Type: integer, Low/High: (0, 65)
Var 4: Y3, Type: integer, Low/High: (0, 76)
Var 5: Y4, Type: integer, Low/High: (0, 63)
Var 6: Base, Type: integer, Low/High: (6, 151)
Var 7: Age, Type: integer, Low/High: (18, 42)
Var 8: Trt
      2 factor levels: placebo progabide
Var 9: Ysum, Type: integer, Low/High: (0, 302)
Var 10: sumY, Type: integer, Low/High: (0, 302)
Var 11: Age10, Type: numeric, Low/High: (1.8000, 4.2000)
Var 12: Base4, Type: numeric, Low/High: (1.5000, 37.7500)
```



To estimate a model with *sumY* as the response variable and the *Base* number of seizures, *Age*, and the treatment as explanatory variables, we can use *rxGlm*. A benefit to using *rxGlm* is that the code will scale for use with a much bigger data set.

```
myGlm <- rxGlm(sumY~ Base + Age + Trt, dropFirst = TRUE,
  data = breslow.dat, family = poisson())
```

112 A Simple Example Using the Poisson Family

```
summary(myGlm)
```

This results in:

```
Call:
rxGlm(formula = sumY ~ Base + Age + Trt, data = breslow.dat,
      family = poisson(), dropFirst = TRUE)

Generalized Linear Model Results for: sumY ~ Base + Age + Trt
Data: breslow.dat
Dependent variable(s): sumY
Total independent variables: 5 (Including number dropped: 1)
Number of valid observations: 59
Number of missing observations: 0
Family-link: poisson-log

Residual deviance: 559.4437 (on 55 degrees of freedom)

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept)  1.9488259   0.1356191   14.370 2.22e-16 ***
Base         0.0226517   0.0005093   44.476 2.22e-16 ***
Age          0.0227401   0.0040240    5.651 5.85e-07 ***
Trt=placebo   Dropped      Dropped Dropped Dropped
Trt=progabide -0.1527009  0.0478051   -3.194 0.00232 **
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for poisson family taken to be 1)

Condition number of final variance-covariance matrix: 3.3382
Number of iterations: 4
```

To interpret the coefficients, it is sometimes useful to transform them back to the original scale of the dependent variable. In this case:

```
exp(coef(myGlm))
```

(Intercept)	Base	Age	Trt=placebo	Trt=progabide
7.0204403	1.0229102	1.0230007	NA	0.8583864

This suggests that, controlling for the base number of seizures and age, those taking progabide during the trial had 85% of the expected number seizures compared with those who didn't.

A common method of checking for overdispersion is to calculate the ratio of the residual deviance with the degrees of freedom, which should be about 1 to fit the assumptions of the model.

```
myGlm$deviance/myGlm$df[2]

[1] 10.1717
```

We can see that the ratio is well above one.

The quasi-poisson family can be used to handle over-dispersion. In this case, instead of assuming that the variance and mean are one, a relationship is estimated from the data:

```
myGlm1 <- rxGlm(sumY ~ Base + Age + Trt, dropFirst = TRUE,
               data = breslow.dat, family = quasipoisson())

summary(myGlm1)
```

Call:
 rxGlm(formula = sumY ~ Base + Age + Trt, data = breslow.dat,
 family = quasipoisson(), dropFirst = TRUE)

Generalized Linear Model Results for: sumY ~ Base + Age + Trt
 Data: breslow.dat
 Dependent variable(s): sumY
 Total independent variables: 5 (Including number dropped: 1)
 Number of valid observations: 59
 Number of missing observations: 0
 Family-link: quasipoisson-log

Residual deviance: 559.4437 (on 55 degrees of freedom)

Coefficients:

	Estimate	Std. Error	t value	Pr(> t)	
(Intercept)	1.948826	0.465091	4.190	0.000102	***
Base	0.022652	0.001747	12.969	2.22e-16	***
Age	0.022740	0.013800	1.648	0.105085	
Trt=placebo	Dropped	Dropped	Dropped	Dropped	
Trt=progabide	-0.152701	0.163943	-0.931	0.355702	

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

(Dispersion parameter for quasipoisson family taken to be 11.76075)

Condition number of final variance-covariance matrix: 3.3382
 Number of iterations: 4

Notice that the coefficients are the same as when using the poisson family, but that the standard errors are larger; the effect of the treatment is no longer significant.

9.2 An Example Using the Gamma Family

The Gamma family is used with data containing positive values with a positive skew. A classic example is estimating the value of auto insurance claims. Using the sample *claims.xdf* data set:

```
claimsXdf <- file.path(rxGetOption("sampleDataDir"), "claims.xdf")

claimsGlm <- rxGlm(cost ~ age + car.age + type, family = Gamma,
                  dropFirst = TRUE, data = claimsXdf)

summary(claimsGlm)
```

Call:

114 An Example Using the Tweedie Family

```
rxGlm(formula = cost ~ age + car.age + type, data = claimsXdf,
       family = Gamma, dropFirst = TRUE)

Generalized Linear Model Results for: cost ~ age + car.age + type
File name:
  C:\Revolution\R-Enterprise-7.0\
R-3.0.2\library\RevoScaleR\SampleData\claims.xdf
Dependent variable(s): cost
Total independent variables: 17 (Including number dropped: 3)
Number of valid observations: 123
Number of missing observations: 5
Family-link: Gamma-inverse

Residual deviance: 15.6397 (on 109 degrees of freedom)

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept)  0.0032807  0.0005126   6.400 4.02e-09 ***
age=17-20      Dropped      Dropped Dropped  Dropped
age=21-24      0.0006593  0.0005471   1.205 0.230843
age=25-29      0.0003911  0.0005183   0.755 0.452114
age=30-34      0.0012388  0.0005982   2.071 0.040720 *
age=35-39      0.0017152  0.0006514   2.633 0.009685 **
age=40-49      0.0012649  0.0006007   2.106 0.037516 *
age=50-59      0.0002863  0.0005087   0.563 0.574771
age=60+        0.0013006  0.0006041   2.153 0.033519 *
car.age=0-3     Dropped      Dropped Dropped  Dropped
car.age=4-7     0.0003444  0.0003535   0.974 0.332120
car.age=8-9     0.0011005  0.0004161   2.645 0.009375 **
car.age=10+     0.0034437  0.0006107   5.639 1.36e-07 ***
type=A          Dropped      Dropped Dropped  Dropped
type=B         -0.0004443  0.0004880  -0.911 0.364508
type=C         -0.0004189  0.0004912  -0.853 0.395668
type=D         -0.0016209  0.0004344  -3.732 0.000304 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for Gamma family taken to be 0.1785316)

Condition number of final variance-covariance matrix: 12.4648
Number of iterations: 4
```

But, note that these estimates are conditional on the fact that a claim was made.

9.3 An Example Using the Tweedie Family

The Tweedie family of distributions provide flexible models for estimation. The power parameter *var.power* determines the shape of the distribution, with familiar models as special cases: if *var.power* is set to 0, Tweedie is a normal distribution; when set to 1, it is Poisson; when 2, it is Gamma; when 3, it is inverse Gaussian. If *var.power* is between 1 and 2, it is a compound Poisson distribution and is appropriate for positive data that also contains exact zeros, for example, insurance claims data, rainfall data, or fish-catch data. If *var.power* is greater than 2, it is appropriate for positive data.

In this example, we'll use a subsample from the 5% sample of the U.S. 2000 census. We will consider the annual cost of property insurance for heads of household ages 21 through 89, and

its relationship to age, sex, and region. A variable “perwt” in the data set represents the probability weight for that observation. First, to create the subsample (specify the correct data path for your downloaded data):

```
bigDataDir = "C:/Revolution/Data"
bigCensusData <- file.path(bigDataDir, "Census5PCT2000.xdf")
propinFile <- "CensusPropertyIns.xdf"

propinDS <- rxDataStep(inData = bigCensusData, outFile = propinFile,
  rowSelection = (related == 'Head/Householder') & (age > 20) & (age < 90),
  varsToKeep = c("propinsr", "age", "sex", "region", "perwt"),
  blocksPerRead = 10, overwrite = TRUE)
rxGetInfo(propinDS)

File name: C:\YourWorkingDir\CensusPropertyIns.xdf
Number of observations: 5175270
Number of variables: 5
Number of blocks: 10
Compression type: zlib
```

An Xdf data source representing the new data file is returned. The new data file has over 5 million observations.

Let’s do one more step in data cleaning. The variable *region* has some very long factor level character strings, and it also has a number of levels for which there are no observations. We can see this using *rxSummary*:

```
rxSummary(~region, data = propinDS)

Call:
rxSummary(formula = ~region, data = propinDS)

Summary Statistics Results for: ~region
File name: C:\YourWorkingDir\CensusPropertyIns.xdf
Number of valid observations: 5175270

Category Counts for region
Number of categories: 17
Number of valid observations: 5175270
Number of missing observations: 0
```

region	Counts
New England Division	265372
Middle Atlantic Division	734585
Mixed Northeast Divisions (1970 Metro)	0
East North Central Div.	847367
West North Central Div.	366417
Mixed Midwest Divisions (1970 Metro)	0
South Atlantic Division	981614
East South Central Div.	324003
West South Central Div.	553425
Mixed Southern Divisions (1970 Metro)	0
Mountain Division	328940
Pacific Division	773547
Mixed Western Divisions (1970 Metro)	0
Military/Military reservations	0
PUMA boundaries cross state lines-1% sample	0

116 An Example Using the Tweedie Family

```
State not identified                                0
Inter-regional county group (1970 Metro samples)    0
```

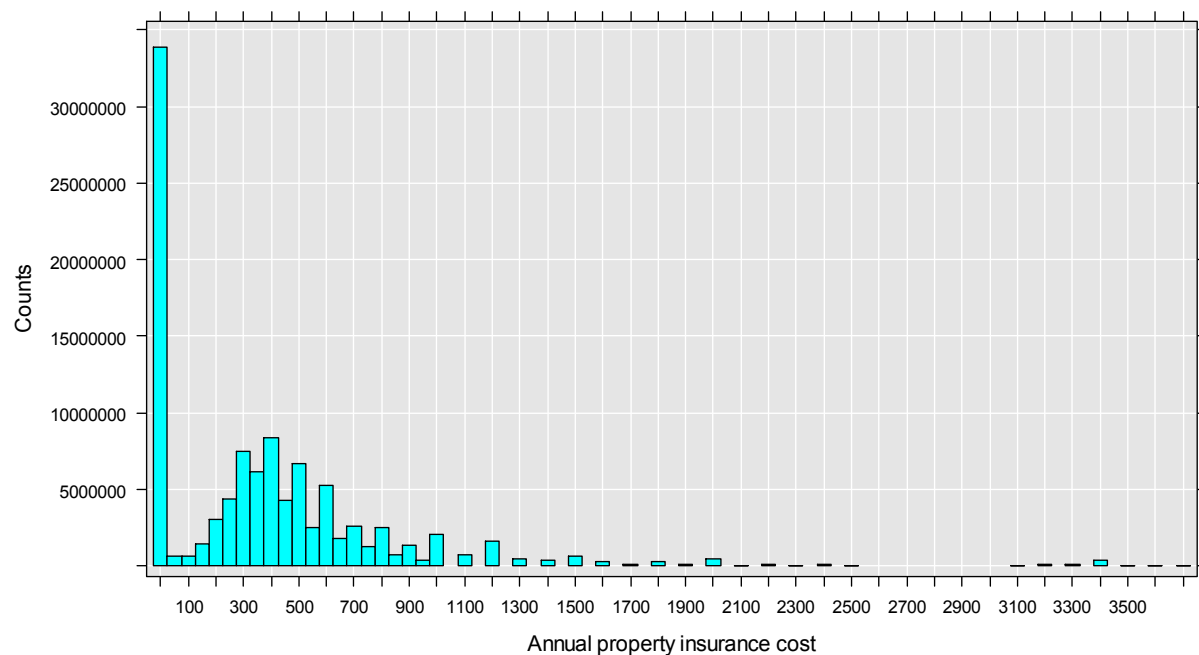
We can use the `rxFactors` function rename and reduce the number of levels:

```
regionLevels <- list( "New England" = "New England Division",
  "Middle Atlantic" = "Middle Atlantic Division",
  "East North Central" = "East North Central Div.",
  "West North Central" = "West North Central Div.",
  "South Atlantic" = "South Atlantic Division",
  "East South Central" = "East South Central Div.",
  "West South Central" = "West South Central Div.",
  "Mountain" = "Mountain Division",
  "Pacific" = "Pacific Division")

rxFactors(inData = propinDS, outFile = propinDS,
  factorInfo = list(region = list(newLevels = regionLevels,
    otherLevel = "Other")),
  overwrite = TRUE)
```

As a first step to analysis, let's look at a histogram of the property insurance cost:

```
rxHistogram(~propinsr, data = propinDS, pweights = "perwt")
```



This appears to be a good match for the Tweedie family with a variance power parameter between 1 and 2, since it has a “clump” of exact zeros in addition to a distribution of positive values.

We can estimate the parameters using `rxGlm`, setting the `var.power` argument to 1.5. As explanatory variables we’ll use sex, an “on-the-fly” factor variable with a level for each age, and region:

```
propinGlm <- rxGlm(propinsr~sex + F(age) + region,
  pweights = "perwt", data = propinDS,
  family = rxTweedie(var.power = 1.5), dropFirst = TRUE)
summary(propinGlm)
```

Call:

```
rxGlm(formula = propinsr ~ sex + F(age) + region, data = propinDS,
  family = rxTweedie(var.power = 1.5), pweights = "perwt",
  dropFirst = TRUE)
```

Generalized Linear Model Results for: propinsr ~ sex + F(age) + region

File name: C:\YourWorkingDir\CensusPropertyIns.xdf

Probability weights: perwt

Dependent variable(s): propinsr

Total independent variables: 82 (Including number dropped: 4)

Number of valid observations: 5175270

Number of missing observations: 0

Family-link: Tweedie-mu^{-0.5}

Residual deviance: 3292809839.3236 (on 5175192 degrees of freedom)

Coefficients:

	Estimate	Std. Error	t value	Pr(> t)	
(Intercept)	1.231e-01	5.893e-04	208.961	2.22e-16	***
sex=Male	Dropped	Dropped	Dropped	Dropped	
sex=Female	9.026e-03	3.164e-05	285.305	2.22e-16	***
F_age=21	Dropped	Dropped	Dropped	Dropped	
F_age=22	-9.208e-03	7.523e-04	-12.240	2.22e-16	***
F_age=23	-1.980e-02	6.966e-04	-28.430	2.22e-16	***
F_age=24	-2.856e-02	6.648e-04	-42.955	2.22e-16	***
F_age=25	-3.652e-02	6.432e-04	-56.776	2.22e-16	***
F_age=26	-4.371e-02	6.289e-04	-69.500	2.22e-16	***
F_age=27	-4.894e-02	6.182e-04	-79.162	2.22e-16	***
F_age=28	-5.398e-02	6.099e-04	-88.506	2.22e-16	***
F_age=29	-5.787e-02	6.043e-04	-95.749	2.22e-16	***
F_age=30	-6.064e-02	6.020e-04	-100.716	2.22e-16	***
F_age=31	-6.336e-02	6.004e-04	-105.522	2.22e-16	***
F_age=32	-6.526e-02	5.991e-04	-108.933	2.22e-16	***
F_age=33	-6.721e-02	5.975e-04	-112.489	2.22e-16	***
F_age=34	-6.854e-02	5.962e-04	-114.948	2.22e-16	***
F_age=35	-6.942e-02	5.949e-04	-116.688	2.22e-16	***
F_age=36	-7.090e-02	5.941e-04	-119.342	2.22e-16	***
F_age=37	-7.184e-02	5.936e-04	-121.023	2.22e-16	***
F_age=38	-7.265e-02	5.931e-04	-122.498	2.22e-16	***
F_age=39	-7.354e-02	5.926e-04	-124.090	2.22e-16	***
F_age=40	-7.401e-02	5.923e-04	-124.954	2.22e-16	***
F_age=41	-7.462e-02	5.923e-04	-125.994	2.22e-16	***
F_age=42	-7.508e-02	5.920e-04	-126.819	2.22e-16	***
F_age=43	-7.568e-02	5.920e-04	-127.846	2.22e-16	***
F_age=44	-7.597e-02	5.919e-04	-128.344	2.22e-16	***

118 An Example Using the Tweedie Family

F_age=45	-7.642e-02	5.918e-04	-129.139	2.22e-16	***
F_age=46	-7.693e-02	5.919e-04	-129.973	2.22e-16	***
F_age=47	-7.727e-02	5.918e-04	-130.564	2.22e-16	***
F_age=48	-7.749e-02	5.919e-04	-130.927	2.22e-16	***
F_age=49	-7.783e-02	5.919e-04	-131.488	2.22e-16	***
F_age=50	-7.809e-02	5.919e-04	-131.941	2.22e-16	***
F_age=51	-7.853e-02	5.919e-04	-132.678	2.22e-16	***
F_age=52	-7.888e-02	5.916e-04	-133.326	2.22e-16	***
F_age=53	-7.919e-02	5.916e-04	-133.859	2.22e-16	***
F_age=54	-7.909e-02	5.931e-04	-133.348	2.22e-16	***
F_age=55	-7.938e-02	5.929e-04	-133.873	2.22e-16	***
F_age=56	-7.930e-02	5.929e-04	-133.751	2.22e-16	***
F_age=57	-7.959e-02	5.928e-04	-134.276	2.22e-16	***
F_age=58	-7.935e-02	5.937e-04	-133.644	2.22e-16	***
F_age=59	-7.923e-02	5.942e-04	-133.336	2.22e-16	***
F_age=60	-7.894e-02	5.946e-04	-132.753	2.22e-16	***
F_age=61	-7.917e-02	5.947e-04	-133.122	2.22e-16	***
F_age=62	-7.912e-02	5.949e-04	-133.003	2.22e-16	***
F_age=63	-7.904e-02	5.954e-04	-132.746	2.22e-16	***
F_age=64	-7.886e-02	5.956e-04	-132.405	2.22e-16	***
F_age=65	-7.878e-02	5.952e-04	-132.359	2.22e-16	***
F_age=66	-7.871e-02	5.961e-04	-132.031	2.22e-16	***
F_age=67	-7.864e-02	5.963e-04	-131.869	2.22e-16	***
F_age=68	-7.861e-02	5.966e-04	-131.766	2.22e-16	***
F_age=69	-7.845e-02	5.967e-04	-131.490	2.22e-16	***
F_age=70	-7.861e-02	5.965e-04	-131.790	2.22e-16	***
F_age=71	-7.856e-02	5.970e-04	-131.600	2.22e-16	***
F_age=72	-7.850e-02	5.971e-04	-131.460	2.22e-16	***
F_age=73	-7.813e-02	5.977e-04	-130.714	2.22e-16	***
F_age=74	-7.818e-02	5.981e-04	-130.722	2.22e-16	***
F_age=75	-7.800e-02	5.986e-04	-130.302	2.22e-16	***
F_age=76	-7.781e-02	5.993e-04	-129.825	2.22e-16	***
F_age=77	-7.763e-02	6.002e-04	-129.342	2.22e-16	***
F_age=78	-7.735e-02	6.009e-04	-128.728	2.22e-16	***
F_age=79	-7.724e-02	6.024e-04	-128.221	2.22e-16	***
F_age=80	-7.646e-02	6.045e-04	-126.495	2.22e-16	***
F_age=81	-7.651e-02	6.060e-04	-126.244	2.22e-16	***
F_age=82	-7.643e-02	6.081e-04	-125.693	2.22e-16	***
F_age=83	-7.600e-02	6.109e-04	-124.411	2.22e-16	***
F_age=84	-7.546e-02	6.145e-04	-122.798	2.22e-16	***
F_age=85	-7.529e-02	6.183e-04	-121.775	2.22e-16	***
F_age=86	-7.441e-02	6.259e-04	-118.882	2.22e-16	***
F_age=87	-7.422e-02	6.324e-04	-117.363	2.22e-16	***
F_age=88	-7.339e-02	6.463e-04	-113.553	2.22e-16	***
F_age=89	-7.310e-02	6.569e-04	-111.284	2.22e-16	***
region=New England	Dropped	Dropped	Dropped	Dropped	
region=Middle Atlantic	1.710e-03	6.893e-05	24.807	2.22e-16	***
region=East North Central	3.552e-03	6.867e-05	51.723	2.22e-16	***
region=West North Central	4.200e-04	7.697e-05	5.457	4.83e-08	***
region=South Atlantic	-1.227e-03	6.521e-05	-18.821	2.22e-16	***
region=East South Central	-7.894e-04	7.793e-05	-10.130	2.22e-16	***
region=West South Central	-5.857e-03	6.732e-05	-87.011	2.22e-16	***
region=Mountain	1.821e-03	8.060e-05	22.596	2.22e-16	***
region=Pacific	-5.990e-04	6.732e-05	-8.897	2.22e-16	***
region=Other	Dropped	Dropped	Dropped	Dropped	

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

(Dispersion parameter for Tweedie family taken to be 546.4888)

Condition number of final variance-covariance matrix: 5980.277
 Number of iterations: 8

A good way to begin examining the results of the estimated model is to look at predicted values for given explanatory characteristics. For example, let's create a prediction data set for the South Atlantic region for all ages and sexes:

```
# Get the region factor levels
varInfo <- rxGetVarInfo(propinDS)
regionLabels <- varInfo$region$levels

# Create a prediction data set for region 5, all ages, both sexes
region <- factor(rep(5, times=138), levels = 1:10, labels = regionLabels)
age <- c(21:89, 21:89)
sex <- factor(c(rep(1, times=69), rep(2, times=69)),
              levels = 1:2,
              labels = c("Male", "Female"))
predData <- data.frame(age, sex, region)
```

Now we'll use that as a basis for a similar prediction data set for the Middle Atlantic region:

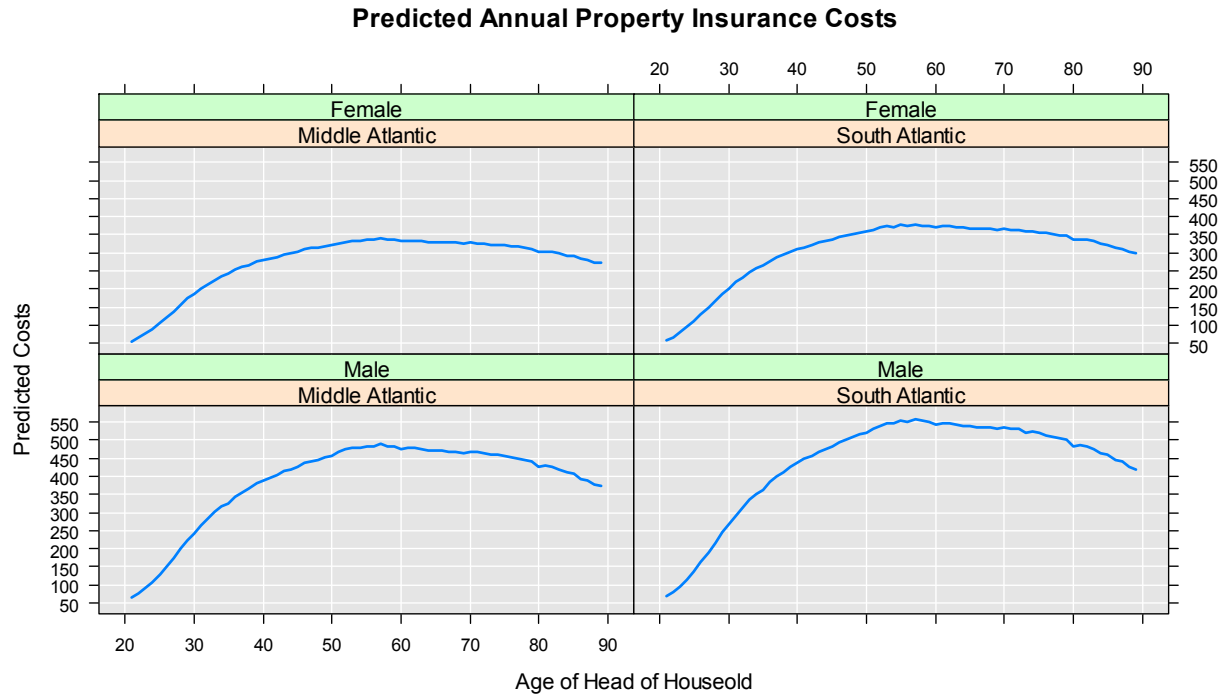
```
# Create a prediction data set for region 2, all ages, both sexes
predData2 <- predData
predData2$region <- factor(rep(2, times=138), levels = 1:10,
                          labels = varInfo$region$levels)
```

Next we'll combine the two data sets, and compute the predicted values for annual property insurance cost using our estimated `rxGlm` model:

```
# Combine data sets and compute predictions
predData <- rbind(predData, predData2)
outData <- rxPredict(propinGlm, data = predData)
```

Last, we can combine the predicted values with our prediction data frame and plot:

```
predData$predicted <- outData$propinsr_Pred
rxLinePlot(predicted ~ age | region + sex, data = predData,
            title = "Predicted Annual Property Insurance Costs",
            xTitle = "Age of Head of Household",
            yTitle = "Predicted Costs")
```



9.4 Stepwise Generalized Linear Models

Stepwise generalized linear models help you determine which variables are most important to include in the model. You provide a minimal, or lower, model formula and a maximal, or upper, model formula, and using forward selection, backward elimination, or bidirectional search, the algorithm determines the model formula that provides the best fit based on an AIC selection criterion or a significance level criterion.

As an example, consider again the Gamma family model from section 9.2:

```
claimsXdf <- file.path(rxGetOption("sampleDataDir"), "claims.xdf")
claimsGlm <- rxGlm(cost ~ age + car.age + type, family = Gamma,
  dropFirst = TRUE, data = claimsXdf)
summary(claimsGlm)
```

We can recast this as a stepwise model by specifying a `variableSelection` argument using the `rxStepControl` function to provide our stepwise arguments:

```
claimsGlmStep <- rxGlm(cost ~ age, family = Gamma, dropFirst=TRUE,
  data=claimsXdf, variableSelection =
  rxStepControl(scope = ~ age + car.age + type ))
summary(claimsGlmStep)
```

Call:

```
rxGlm(formula = cost ~ age, data = claimsXdf, family = Gamma,
  variableSelection = rxStepControl(scope = ~age + car.age +
  type), dropFirst = TRUE)
```

Generalized Linear Model Results for: cost ~ car.age + type
File name:


```

C:\Revolution\R-Enterprise-7.0\R-
3.0.2\library\RevoScaleR\SampleData\claims.xdf
Dependent variable(s): cost
Total independent variables: 9 (Including number dropped: 2)
Number of valid observations: 123
Number of missing observations: 5
Family-link: Gamma-inverse

Residual deviance: 18.0433 (on 116 degrees of freedom)

Coefficients:
      Estimate Std. Error t value Pr(>|t|)
(Intercept)  0.0040354  0.0004661   8.657 3.24e-14 ***
car.age=0-3      Dropped      Dropped Dropped  Dropped
car.age=4-7   0.0003568  0.0004037   0.884  0.37868
car.age=8-9   0.0011825  0.0004688   2.522  0.01302 *
car.age=10+   0.0035478  0.0006853   5.177 9.57e-07 ***
type=A         Dropped      Dropped Dropped  Dropped
type=B        -0.0004512  0.0005519  -0.818  0.41528
type=C        -0.0004135  0.0005558  -0.744  0.45837
type=D        -0.0016307  0.0004923  -3.313  0.00123 **
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for Gamma family taken to be 0.2249467)

Condition number of final variance-covariance matrix: 9.1775
Number of iterations: 5

```

We see that in the stepwise model fit, age no longer appears in the final model.

Chapter 10.

Estimating Decision Tree Models

The *rxDTree* function in RevoScaleR fits tree-based models using a binning-based recursive partitioning algorithm. The resulting model is similar to that produced by the recommended R package *rpart*. Both classification-type trees and regression-type trees are supported; as with *rpart*, the difference is determined by the nature of the response variable: a factor response generates a classification tree; a numeric response generates a regression tree.

10.1 The *rxDTree* Algorithm

Decision trees are effective algorithms widely used for classification and regression. Building a decision tree generally requires that all continuous variables be sorted in order to decide where to split the data. This sorting step becomes time and memory prohibitive when dealing with large data. Various techniques have been proposed to overcome the sorting obstacle, which can be roughly classified into two groups: performing data pre-sorting or using approximate summary statistic of the data. While pre-sorting techniques follow standard decision tree algorithms more closely, they cannot accommodate very large data sets. These big data decision trees are normally parallelized in various ways to enable large scale learning: data parallelism partitions the data either horizontally or vertically so that different processors see different observations or variables and task parallelism builds different tree nodes on different processors.

The *rxDTree* algorithm is an approximate decision tree algorithm with horizontal data parallelism, especially designed for handling very large data sets. It uses histograms as the approximate compact representation of the data and builds the decision tree in a breadth-first fashion. The algorithm can be executed in parallel settings such as a multicore machine or a distributed environment with a master-worker architecture. Each worker gets only a subset of the observations of the data, but has a view of the complete tree built so far. It builds a histogram from the observations it sees, which essentially compresses the data to a fixed amount of memory. This approximate description of the data is then sent to a master with constant low communication complexity independent of the size of the data set. The master integrates the information received from each of the workers and determines which terminal tree nodes to split and how. Since the histogram is built in parallel, it can be quickly constructed even for extremely large data sets.

With *rxDTree*, you can control the balance between time complexity and prediction accuracy by specifying the maximum number of bins for the histogram. The algorithm builds the histogram with roughly equal number of observations in each bin and takes the boundaries of the bins as the candidate splits for the terminal tree nodes. Since only a limited number of split locations are examined, it is possible that a suboptimal split point is chosen causing the entire tree to be different from the one constructed by a standard algorithm. However, it has been shown analytically that the error rate of the parallel tree approaches the error rate of the serial tree, even though the trees are not identical. You can set the number of bins in the histograms to control the tradeoff between accuracy and speed: a large number of bins allows a more accurate description of the data and thus more accurate results, whereas a small number of bins reduces time complexity and memory usage.

In the case of integer predictors for which the number of bins equals or exceeds the number of observations, the *rxDTree* algorithm produces the same results as the standard sorting algorithms.

10.2 A Simple Classification Tree

In Chapter 8, we fit a simple logistic regression model to *rpart*'s *kyphosis* data. That model is easily recast as a classification tree using *rxDTree* as follows:

```
data("kyphosis", package="rpart")
kyphTree <- rxDTree(Kyphosis ~ Age + Start + Number, data = kyphosis,
  cp=0.01)
kyphTree

Call:
rxDTree(formula = Kyphosis ~ Age + Start + Number, data = kyphosis,
  cp = 0.01)
Data: kyphosis
Number of valid observations: 81
```

124 A Simple Regression Tree

```
Number of missing observations:  0

Tree representation:
n= 81

node), split, n, loss, yval, (yprob)
  * denotes terminal node

1) root 81 17 absent (0.79012346 0.20987654)
 2) Start>=8.5 62  6 absent (0.90322581 0.09677419)
   4) Start>=14.5 29  0 absent (1.00000000 0.00000000) *
   5) Start< 14.5 33  6 absent (0.81818182 0.18181818)
      10) Age< 55 12  0 absent (1.00000000 0.00000000) *
      11) Age>=55 21  6 absent (0.71428571 0.28571429)
          22) Age>=111 14  2 absent (0.85714286 0.14285714) *
          23) Age< 111 7  3 present (0.42857143 0.57142857) *
 3) Start< 8.5 19  8 present (0.42105263 0.57894737) *
```

Recall our conclusions from fitting this model earlier with `rxCube`: the probability of the post-operative complication Kyphosis seems to be greater if the Start is a cervical vertebra and as more vertebrae are involved in the surgery. Similarly, it appears that the dependence on age is non-linear: it first increases with age, peaks in the range 5-9, and then decreases again.

The `rxDTree` model seems to confirm these earlier conclusions—for Start < 8.5, 11 of 19 observed subjects developed Kyphosis, while none of the 29 subjects with Start >= 14.5 did. For the remaining 33 subjects, Age was the primary splitting factor, and as we observed earlier, ages 5 to 9 had the highest probability of developing Kyphosis.

The returned object `kyphTree` is an object of class `rxDTree`. The `rxDTree` class is modeled closely on the `rpart` class, so that objects of class `rxDTree` have most essential components of an `rpart` object: frame, cptable, splits, etc. By default, however, `rxDTree` objects do not inherit from class `rpart`. You can, however, use the `rxAddInheritance` function to add `rpart` inheritance to `rxDTree` objects.

10.3 A Simple Regression Tree

As a simple example of a regression tree, consider the `mtcars` data set and let's fit gas mileage (`mpg`) using displacement (`disp`) as a predictor:

```
mtcarTree <- rxDTree(mpg ~ disp, data=mtcars)
mtcarTree

Call:
rxDTree(formula = mpg ~ disp, data = mtcars)
Data:  mtcars
Number of valid observations:  32
Number of missing observations:  0

Tree representation:
n= 32

node), split, n, deviance, yval
```

```

      * denotes terminal node
1) root 32 1126.0470 20.09063
  2) disp>=163.5 18 143.5894 15.99444 *
  3) disp< 163.5 14 292.1343 25.35714 *

```

There's a clear split between larger cars (those with engine displacement greater than 163.5 cubic inches) and smaller cars.

10.4 A Larger Regression Tree Model

As a more complex example, we return to the `censusWorkers` data. We will create a regression tree predicting wage income from age, sex, and weeks worked, using the `perwt` variable as probability weights:

```

censusWorkers <- file.path(rxGetOption("sampleDataDir"),
  "CensusWorkers.xdf")
rxGetInfo(censusWorkers, getVarInfo=TRUE)
incomeTree <- rxDTree(incwage ~ age + sex + wkswork1, pweights = "perwt",
  maxDepth = 3, minBucket = 30000, data = censusWorkers)
incomeTree

```

```

Call:
rxDTree(formula = incwage ~ age + sex + wkswork1, data = censusWorkers,
  pweights = "perwt", minBucket = 30000, maxDepth = 3)
File: C:\Revolution\R-Enterprise-7.0\R-
3.0.2\library\RevoScaleR\SampleData\CensusWorkers.xdf
Number of valid observations: 351121
Number of missing observations: 0

```

```

Tree representation:
n= 351121

```

```

node), split, n, deviance, yval
      * denotes terminal node
1) root 351121 1.177765e+16 35788.47
  2) sex=Female 161777 2.271425e+15 26721.09
    4) wkswork1< 51.5 56874 5.757587e+14 19717.74 *
    5) wkswork1>=51.5 104903 1.608813e+15 30505.87
      10) age< 34.5 31511 2.500078e+14 25836.32 *
      11) age>=34.5 73392 1.338235e+15 32576.74 *
  3) sex=Male 189344 9.008506e+15 43472.71
    6) age< 31.5 48449 6.445334e+14 27577.80 *
    7) age>=31.5 140895 8.010642e+15 49221.82
      14) wkswork1< 51.5 34359 1.550839e+15 37096.62 *
      15) wkswork1>=51.5 106536 6.326896e+15 53082.08 *

```

The primary split here (not surprising given our analysis of this data set in the *Getting Started Guide*) is sex; women on average earn substantially less than men. The additional splits are also not surprising; older workers earn more than younger workers, and those who work more hours tend to earn more than those who work fewer hours.

10.5 Controlling the Model Fit

The *rxDTree* function has a number of options for controlling the model fit. Most of these control parameters will be familiar to *rpart* users, but the defaults have been modified in some cases to better support large data tree models. A full listing of these options can be found in the *rxDTree* help file, but the following have been found in our testing to be the most useful at controlling the time required to fit a model with *rxDTree*:

- *xVal*: this controls the number of folds used to perform cross-validation. The default of 2 allows for some pruning; once you have closed in a model you may want to increase the value for final fitting and pruning.
- *maxDepth*: this sets the maximum depth of any node of the tree. Computations grow rapidly more expensive as the depth increases, so we recommend a *maxDepth* of 10 to 15.
- *maxCompete*: this specifies the number of “competitor splits” retained in the output. By default, *rxDTree* sets this to 0, but a setting of 3 or 4 can be useful for diagnostic purposes in determining why a particular split was chosen.
- *maxSurrogate*: this specifies the number of surrogate splits retained in the output. Again, by default *rxDTree* sets this to 0. Surrogate splits are used to assign an observation when the primary split variable is missing for that observation.
- *maxNumBins*: this controls the maximum number of bins used for each variable. Managing the number of bins is important in controlling memory usage. The default is to use the larger of 101 and the square root of the number of observations for small to moderate size data sets (up to about one million observations), but for larger sets to use 1001 bins. For small data sets with continuous predictors, you may find that you need to increase the *maxNumBins* to obtain models that resemble those from *rpart*.

For large data sets (100000 or more observations), you may need to adjust the following parameters to obtain meaningful models:

- *cp*: this is a complexity parameter and sets the bar for how much a split must reduce the complexity before being accepted. We have set the default to 0 and recommend using *maxDepth* and *minBucket* to control your tree sizes. If you want to specify a *cp* value, start with a conservative value, such as *rpart*’s 0.01; if you don’t see an adequate number of splits, decrease the *cp* by powers of 10 until you do. For our large airline data, we have found interesting models begin with a *cp* of about 1e-4.
- *minSplit*, *minBucket*: these determine how many observations must be in a node before a split is attempted (*minSplit*) and how many must remain in a terminal node (*minBucket*).

10.6 Large Data Tree Models

Scaling decision trees to very large data sets is possible with *rxDTree* but should be done with caution—the wrong choice of model parameters can easily lead to models that take hours or longer to estimate, even in a distributed computing environment. For example, in the *Getting Started Guide*, we estimated linear models using the big airline data and used the variable *Origin* as a predictor in several models. The *Origin* variable is a factor variable with 373 levels with no obvious ordering. Incorporating this variable into an *rxDTree* model that is performing more than two level classification can easily consume hours of computation time. To prevent such unintended consequences, *rxDTree* has a parameter *maxUnorderedLevels* which defaults to 32; in the case of *Origin*, this parameter would flag an error. However, a factor variable of “Region” which groups the airports of *Origin* by location may well be a useful proxy, and can be constructed to have only a limited number of levels. Numeric and ordered factor predictors are much more easily incorporated into the model.

As an example of a large data classification tree, consider the following simple model using the 7% subsample of the full airline data (this uses the variable *ArrDel15* indicating flights with an arrival delay of 15 minutes or more):

```
bigDataDir <- "C:/Revolution/Data"
sampleAirData <- file.path(bigDataDir, "AirOnTime7Pct.xdf")
airlineTree <- rxDTree(ArrDel15 ~ CRSDepTime + DayOfWeek, data = sampleAirData,
  blocksPerRead = 30, maxDepth = 5, cp = 1e-5)
```

The default *cp* of 0 produces a very large number of splits; specifying *cp* = 1e-5 produces a more manageable set of splits in this model:

```
airlineTree

Call:
rxDTree(formula = ArrDel15 ~ CRSDepTime + DayOfWeek, data = sampleAirData,
  maxDepth = 5, cp = 1e-05, blocksPerRead = 30)
File: C:\Revolution\Data\AirOnTime7Pct.xdf
Number of valid observations: 10186272
Number of missing observations: 213483

Tree representation:
n= 10186272

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

1) root 10186272 1630331.000 0.20008640
 2) CRSDepTime< 13.1745 4941190 642452.000 0.15361830
   4) CRSDepTime< 8.3415 1777685 189395.700 0.12123970
     8) CRSDepTime>=0.658 1717573 178594.900 0.11787560
       16) CRSDepTime< 6.7665 599548 52711.450 0.09740671
         32) CRSDepTime>=1.625 578762 49884.260 0.09526714 *
         33) CRSDepTime< 1.625 20786 2750.772 0.15698070 *
       17) CRSDepTime>=6.7665 1118025 125497.500 0.12885220
         34) DayOfWeek=Sun 134589 11722.540 0.09638975 *
```

128 Large Data Tree Models

```

35) DayOfWeek=Mon,Tues,Wed,Thur,Fri,Sat 983436 113613.80 0.13329490 *
9) CRSDepTime< 0.658 60112 10225.960 0.21736090
18) CRSDepTime>=0.2415 9777 1429.046 0.17776410 *
19) CRSDepTime< 0.2415 50335 8778.609 0.22505220 *
5) CRSDepTime>=8.3415 3163505 450145.400 0.17181290
10) CRSDepTime< 11.3415 1964400 268472.400 0.16335320
20) DayOfWeek=Sun 271900 30839.160 0.13043400
40) CRSDepTime< 9.7415 126700 13381.800 0.12002370 *
41) CRSDepTime>=9.7415 145200 17431.650 0.13951790 *
21) DayOfWeek=Mon,Tues,Wed,Thur,Fri,Sat 1692500 237291.300 0.16864170
42) DayOfWeek=Tues,Wed,Sat 835355 113384.500 0.16196470 *
43) DayOfWeek=Mon,Thur,Fri 857145 123833.200 0.17514890 *
11) CRSDepTime>=11.3415 1199105 181302.000 0.18567180
22) DayOfWeek=Mon,Tues,Wed,Sat,Sun 852016 124610.900 0.17790390
44) DayOfWeek=Tues,Sun 342691 48917.520 0.17250230 *
45) DayOfWeek=Mon,Wed,Sat 509325 75676.600 0.18153830 *
23) DayOfWeek=Thur,Fri 347089 56513.560 0.20474000 *
3) CRSDepTime>=13.1745 5245082 967158.500 0.24386220
6) DayOfWeek=Mon,Tues,Wed,Sat,Sun 3708992 651771.300 0.22746990
12) DayOfWeek=Sat 635207 96495.570 0.18681000
24) CRSDepTime>=20.2745 87013 12025.600 0.16564190 *
25) CRSDepTime< 20.2745 548194 84424.790 0.19016990 *
13) DayOfWeek=Mon,Tues,Wed,Sun 3073785 554008.600 0.23587240
26) CRSDepTime< 16.508 1214018 203375.700 0.21281150
52) CRSDepTime< 15.1325 709846 114523.300 0.20223400 *
53) CRSDepTime>=15.1325 504172 88661.120 0.22770400 *
27) CRSDepTime>=16.508 1859767 349565.800 0.25092610
54) DayOfWeek=Mon,Tues 928523 168050.900 0.23729730 *
55) DayOfWeek=Wed,Sun 931244 181170.600 0.26451500 *
7) DayOfWeek=Thur,Fri 1536090 311984.200 0.28344240
14) CRSDepTime< 15.608 445085 82373.020 0.24519140
28) CRSDepTime< 14.6825 273682 49360.240 0.23609880 *
29) CRSDepTime>=14.6825 171403 32954.030 0.25970960 *
15) CRSDepTime>=15.608 1091005 228694.300 0.29904720
30) CRSDepTime>=21.9915 64127 11932.930 0.24718140 *
31) CRSDepTime< 21.9915 1026878 216578.100 0.30228620
62) CRSDepTime< 17.0745 264085 53451.260 0.28182970 *
63) CRSDepTime>=17.0745 762793 162978.000 0.30936830 *

```

Looking at the fitted objects `cptable` component, we can look at whether we have overfitted the model:

```
airlineTree$cptable
```

	CP	nsplit	relerror	xerror	xstd
1	1.270950e-02	0	1.0000000	1.0000002	0.0004697734
2	2.087342e-03	1	0.9872905	0.9873043	0.0004629111
3	1.785488e-03	2	0.9852032	0.9852215	0.0004625035
4	7.772395e-04	3	0.9834177	0.9834381	0.0004608330
5	6.545095e-04	4	0.9826404	0.9826606	0.0004605065
6	5.623968e-04	5	0.9819859	0.9820200	0.0004602950
7	3.525848e-04	6	0.9814235	0.9814584	0.0004602578
8	2.367018e-04	7	0.9810709	0.9811071	0.0004600062
9	2.274981e-04	8	0.9808342	0.9808700	0.0004597725
10	2.112635e-04	9	0.9806067	0.9806567	0.0004596187
11	2.097651e-04	10	0.9803955	0.9804365	0.0004595150
12	1.173008e-04	11	0.9801857	0.9803311	0.0004594245
13	1.124180e-04	12	0.9800684	0.9800354	0.0004592792
14	1.089414e-04	13	0.9799560	0.9800354	0.0004592792
15	9.890134e-05	14	0.9798471	0.9799851	0.0004592187
16	9.125152e-05	15	0.9797482	0.9798766	0.0004591605
17	4.687397e-05	16	0.9796569	0.9797504	0.0004591074

18	4.510554e-05	17	0.9796100	0.9797292	0.0004590784
19	3.603837e-05	18	0.9795649	0.9796812	0.0004590301
20	2.771093e-05	19	0.9795289	0.9796383	0.0004590247
21	1.577140e-05	20	0.9795012	0.9796013	0.0004590000
22	1.122899e-05	21	0.9794854	0.9795671	0.0004589736
23	1.025944e-05	22	0.9794742	0.9795560	0.0004589678
24	1.000000e-05	23	0.9794639	0.9795455	0.0004589660

We see a steady decrease in cross-validation error (*xerror*) as the number of splits increase, but note that at about *nsplit*=11 the rate of change slows dramatically. The optimal model is probably very near here. (The total number of passes through the data is equal to a base of *maxDepth* + 3, plus *xVal* times (*maxDepth* + 2), where *xVal* is the number of folds for cross-validation and *maxDepth* is the maximum tree depth. Thus a depth 10 tree with 4-fold cross-validation will require 13 + 48, or 61, passes through the data.)

To prune the tree back, use the *prune.rxDTree* function:

```
airlineTree4 <- prune.rxDTree(airlineTree, cp=1e-4)
airlineTree4
```

Call:

```
rxDTree(formula = ArrDel15 ~ CRSDepTime + DayOfWeek, data = sampleAirData,
  maxDepth = 5, cp = 1e-05, blocksPerRead = 30)
File: C:\Revolution\Data\AirOnTime7Pct.xdf
Number of valid observations: 10186272
Number of missing observations: 213483
```

Tree representation:

```
n= 10186272
node), split, n, deviance, yval
* denotes terminal node
```

- 1) root 10186272 1630331.00 0.20008640
 - 2) CRSDepTime< 13.1745 4941190 642452.00 0.15361830
 - 4) CRSDepTime< 8.3415 1777685 189395.70 0.12123970
 - 8) CRSDepTime>=0.658 1717573 178594.90 0.11787560
 - 16) CRSDepTime< 6.7665 599548 52711.45 0.09740671 *
 - 17) CRSDepTime>=6.7665 1118025 125497.50 0.12885220 *
 - 9) CRSDepTime< 0.658 60112 10225.96 0.21736090 *
 - 5) CRSDepTime>=8.3415 3163505 450145.40 0.17181290
 - 10) CRSDepTime< 11.3415 1964400 268472.40 0.16335320
 - 20) DayOfWeek=Sun 271900 30839.16 0.13043400 *
 - 21) DayOfWeek=Mon,Tues,Wed,Thur,Fri,Sat 1692500 237291.30 0.16864170 *
 - 11) CRSDepTime>=11.3415 1199105 181302.00 0.18567180
 - 22) DayOfWeek=Mon,Tues,Wed,Sat,Sun 852016 124610.90 0.17790390 *
 - 23) DayOfWeek=Thur,Fri 347089 56513.56 0.20474000 *
 - 3) CRSDepTime>=13.1745 5245082 967158.50 0.24386220
 - 6) DayOfWeek=Mon,Tues,Wed,Sat,Sun 3708992 651771.30 0.22746990
 - 12) DayOfWeek=Sat 635207 96495.57 0.18681000 *
 - 13) DayOfWeek=Mon,Tues,Wed,Sun 3073785 554008.60 0.23587240
 - 26) CRSDepTime< 16.508 1214018 203375.70 0.21281150
 - 52) CRSDepTime< 15.1325 709846 114523.30 0.20223400 *
 - 53) CRSDepTime>=15.1325 504172 88661.12 0.22770400 *
 - 27) CRSDepTime>=16.508 1859767 349565.80 0.25092610
 - 54) DayOfWeek=Mon,Tues 928523 168050.90 0.23729730 *
 - 55) DayOfWeek=Wed,Sun 931244 181170.60 0.26451500 *
 - 7) DayOfWeek=Thur,Fri 1536090 311984.20 0.28344240
 - 14) CRSDepTime< 15.608 445085 82373.02 0.24519140 *

130 Large Data Tree Models

```
15) CRSDepTime>=15.608 1091005 228694.30 0.29904720
30) CRSDepTime>=21.9915 64127 11932.93 0.24718140 *
31) CRSDepTime< 21.9915 1026878 216578.10 0.30228620 *
```

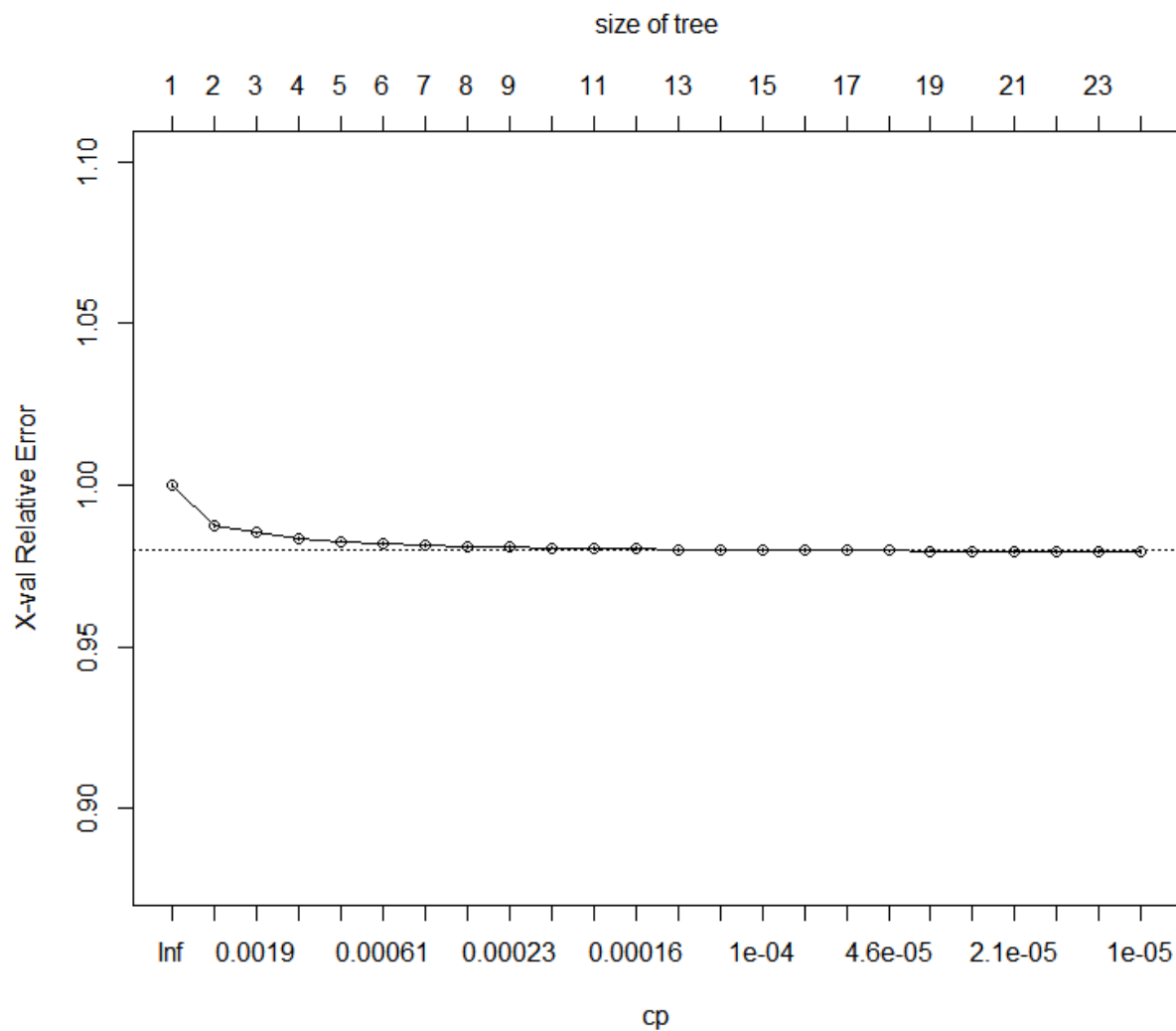
If `rpart` is installed, `prune.rxDTree` acts as a method for the `prune` function, so you can call it more simply:

```
airlineTree4 <- prune(airlineTree, cp=1e-4)
```

For models fit with 2-fold or greater cross-validation, it is useful to use the cross-validation standard error (part of the `cptable` component) as a guide to pruning. The `rpart` function `plotcp` can be useful for this:

```
plotcp(rxAddInheritance(airlineTree))
```

This yields the following plot:



From this plot, it appears we can prune even further, to perhaps seven or eight splits. Looking again at the `cptable`, a `cp` of $2.5e-4$ seems a reasonable pruning choice:

```
airlineTreePruned <- prune.rxDTree(airlineTree, cp=2.5e-4)
airlineTreePruned

Call:
rxDTree(formula = ArrDel15 ~ CRSDepTime + DayOfWeek, data = sampleAirData,
        maxDepth = 5, cp = 1e-05, blocksPerRead = 30)
File: C:\Revolution\Data\AirOnTime7Pct.xdf
Number of valid observations: 10186272
Number of missing observations: 213483

Tree representation:
n= 10186272

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

1) root 10186272 1630331.00 0.2000864
 2) CRSDepTime< 13.1745 4941190 642452.00 0.1536183
   4) CRSDepTime< 8.3415 1777685 189395.70 0.1212397
    8) CRSDepTime>=0.658 1717573 178594.90 0.1178756 *
    9) CRSDepTime< 0.658 60112 10225.96 0.2173609 *
   5) CRSDepTime>=8.3415 3163505 450145.40 0.1718129 *
 3) CRSDepTime>=13.1745 5245082 967158.50 0.2438622
   6) DayOfWeek=Mon,Tues,Wed,Sat,Sun 3708992 651771.30 0.2274699
    12) DayOfWeek=Sat 635207 96495.57 0.1868100 *
    13) DayOfWeek=Mon,Tues,Wed,Sun 3073785 554008.60 0.2358724
     26) CRSDepTime< 16.508 1214018 203375.70 0.2128115 *
     27) CRSDepTime>=16.508 1859767 349565.80 0.2509261 *
   7) DayOfWeek=Thur,Fri 1536090 311984.20 0.2834424
    14) CRSDepTime< 15.608 445085 82373.02 0.2451914 *
    15) CRSDepTime>=15.608 1091005 228694.30 0.2990472 *
```

10.7 Handling Missing Values

The `removeMissings` argument to `rxDTree`, as in most RevoScaleR analysis functions, controls how the function deals with missing data in the model fit. If `TRUE`, all rows containing missing values for the response or any predictor variable are removed before model fitting. If `FALSE` (the default), only those rows for which the value of the response or all values of the predictor variables are missing are removed. Using `removeMissings=TRUE` is roughly equivalent to the effect of the `na.omit` function for `rpart`, in that if the file is written out, all rows containing NAs are simply removed. There is no equivalent for `rxDTree` to the `na.exclude` function, which pads the output with NAs for observations that cannot be predicted. Using `removeMissings=FALSE` is the equivalent of using the `na.rpart` or `na.pass` functions; the data is passed through unchanged, but rows which have no data for either all predictors or the response are excluded from the model.

10.8 Prediction

As with other RevoScaleR analysis functions, prediction is performed using the `rxPredict` function, to which you supply a fitted model object and a set of new data (which may be the original data set, but in any event must contain the variables used in the original model).

The adult data set (Kohavi, 1996) is a widely used machine learning data set, similar to the censusWorkers data we have already analyzed. The data set is available from the machine learning data repository at UC Irvine (<http://archive.ics.uci.edu/ml/datasets/Adult>) (Frank & Asuncion, 2010) and comes in two pieces: a training data set (adult.data) and a test data set (adult.test). This makes it ready-made for use in prediction. To run the examples below, download this data and add a .txt extension, so that you have adult.data.txt and adult.test.txt. (A third file, adult.names, gives a description of the variables; we use this in the code below as a source for the variable names, which are not part of the data files):

```
bigDataDir <- "C:/Revolution/Data"
adultDataFile <- file.path(bigDataDir, "adult.data.txt")
adultTestFile <- file.path(bigDataDir, "adult.test.txt")

newNames <- c("age", "workclass", "fnlwgt", "education",
              "education_num", "marital_status", "occupation", "relationship",
              "race", "sex", "capital_gain", "capital_loss", "hours_per_week",
              "native_country", "income")
adultTrain <- rxImport(adultDataFile, stringsAsFactors = TRUE)
names(adultTrain) <- newNames
adultTest <- rxImport(adultTestFile, rowsToSkip = 1,
                     stringsAsFactors=TRUE)
names(adultTest) <- newNames
adultTree <- rxDTree(income ~ age + sex + hours_per_week, pweights = "fnlwgt",
                    data = adultTrain)
adultPred <- rxPredict(adultTree, data = adultTest, type="vector")
sum(adultPred == as.integer(adultTest$income))/length(adultTest$income)
[1] 0.7734169
```

The result shows that the fitted model accurately classifies about 77% of the test data.

When using `rxPredict` with `rxDTree` objects, you should keep in mind how it differs from `predict` with `rpart` objects. First, a `data` argument is always required—this can be either the original data or new data; there is no `newData` argument as in `rpart`. Prediction with the original data provides fitted values, not predictions, but the predicted variable name still defaults to `varname_Pred`.

10.9 Visualizing Trees

The new RevoTreeView package can be used to plot decision trees from `rxDTree` or `rpart` in an HTML page. Both classification and regression trees are supported. By plotting the tree objects returned by RevoTreeView's `createTreeView` function in a browser, you can interact with your

decision tree. The resulting tree's HTML page can also be shared with other people or displayed on different machines using the package's `zipTreeView` function.

As an example, consider a classification tree built from the `kyphosis` data that is included in the `rpart` package. It produces the following text output:

```
data("kyphosis", package="rpart")
kyphTree <- rxDTree(Kyphosis ~ Age + Start + Number,
  data = kyphosis, cp=0.01)
kyphTree

Call:
rxDTree(formula = Kyphosis ~ Age + Start + Number, data = kyphosis,
  cp = 0.01)
Data: kyphosis
Number of valid observations: 81
Number of missing observations: 0

Tree representation:
n= 81

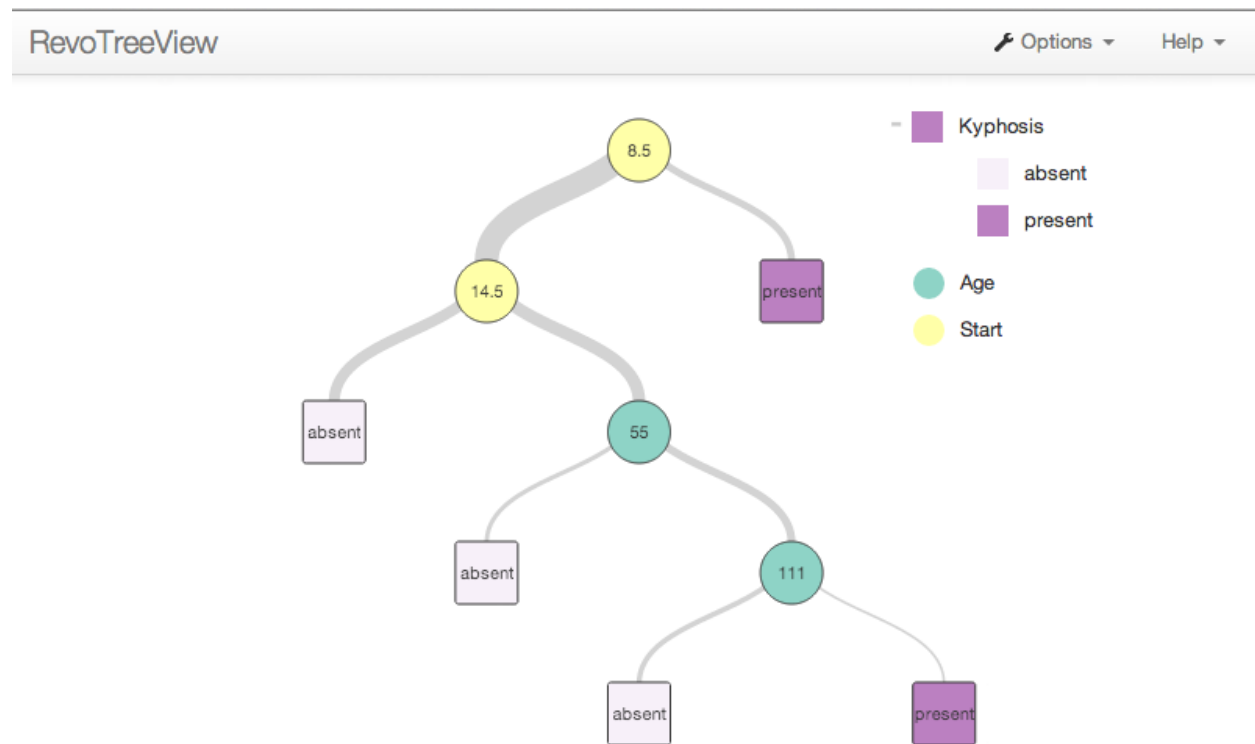
node), split, n, loss, yval, (yprob)
* denotes terminal node
1) root 81 17 absent (0.79012346 0.20987654)
  2) Start>=8.5 62 6 absent (0.90322581 0.09677419)
    4) Start>=14.5 29 0 absent (1.00000000 0.00000000) *
    5) Start< 14.5 33 6 absent (0.81818182 0.18181818)
      10) Age< 55 12 0 absent (1.00000000 0.00000000) *
      11) Age>=55 21 6 absent (0.71428571 0.28571429)
        22) Age>=111 14 2 absent (0.85714286 0.14285714) *
        23) Age< 111 7 3 present (0.42857143 0.57142857) *
  3) Start< 8.5 19 8 present (0.42105263 0.57894737) *
```

Now, you can display a HTML version of the tree output by plotting the object produced by the

createTreeView function. After running the preceding R code, run

the following to load the **RevoTreeView** package and display an interactive decision tree in your browser:

```
library(RevoTreeView)
plot(createTreeView(kyphTree))
```



In this interactive tree, click on the circular split nodes to expand or collapse the tree branch. Clicking a node will expand and collapse the node to the last view of that branch. If you use a

CTRL + Click, the tree will display only the children of the selected node. If

you click **ALT + Click**, the tree will display all levels below the selected node.

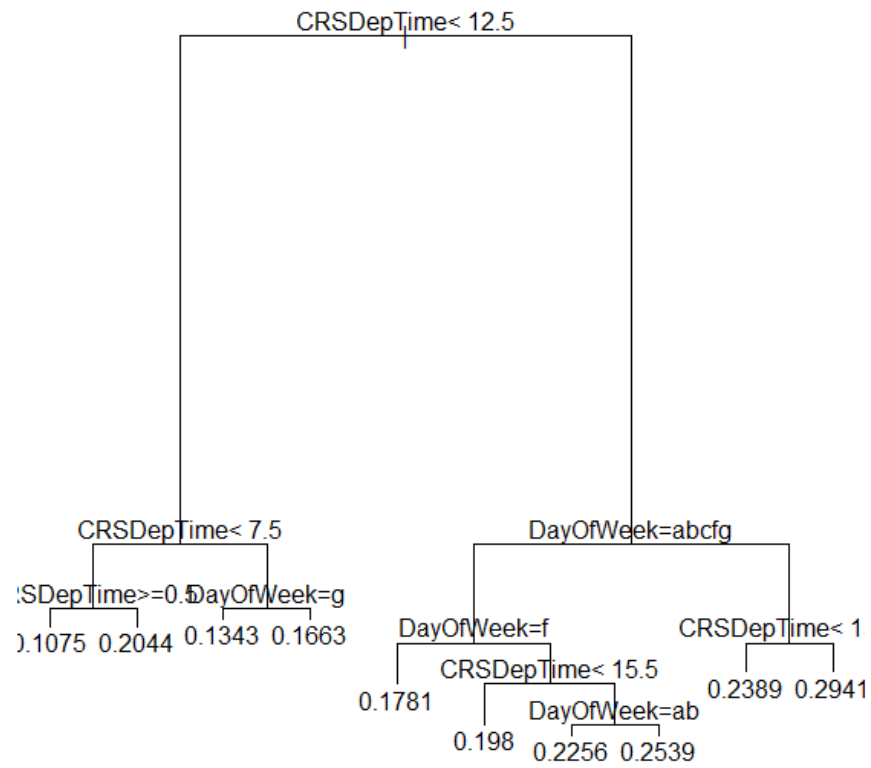
The square-shaped nodes, called leaf or terminal nodes, cannot be expanded.

To get additional information, hover over the node to expose the node details such as its name, the next split variable, its value, the n , the predicted value, and other details such as loss or deviance.

You can also use the `rpart` `plot` and `text` methods with `rxDTree` objects, provided you use the `rxAddInheritance` function to provide `rpart` inheritance:

```
plot(rxAddInheritance(airlineTreePruned))
text(rxAddInheritance(airlineTreePruned))
```

This provides the following plot:



Chapter 11.

Estimating Decision Forest Models

The `rxDForest` function in RevoScaleR fits a *decision forest*, which is an ensemble of decision trees. Each tree is fitted to a bootstrap sample of the original data, which leaves about 1/3 of the data unused in the fitting of each tree. Each data point in the original data is fed through each of the trees for which it was unused; the decision forest prediction for that data point is the statistical *mode* of the individual tree predictions, that is, the majority prediction (for classification; for regression problems, the prediction is the mean of the individual predictions).

Unlike individual decision trees, decision forests are not prone to overfitting, and they are consistently shown to be among the best machine learning algorithms. RevoScaleR implements decision forests in the `rxDForest` function, which uses the same basic tree-fitting algorithm as `rxDTree` (see Section 10.1, The `rxDTree` Algorithm). To create the forest, you specify the number of trees using the `nTree` argument and the number of variables to consider for splitting in each tree using the `mTry` argument. In most cases, you will also want to specify the maximum depth to grow the individual trees: greater depth typically results in greater accuracy, but as with `rxDTree`, also results in significantly longer fitting times.

11.1 A Simple Classification Forest

In Chapter 10, we fit a simple classification tree model to `rpart`'s `kyphosis` data. That model is easily recast as a classification decision forest using `rxDForest` as follows (we set the `seed` argument to ensure reproducibility; in most cases you can omit this):

```
data("kyphosis", package="rpart")
kyphForest <- rxDForest(Kyphosis ~ Age + Start + Number, seed = 10,
  data = kyphosis, cp=0.01, nTree=500, mTry=3)
kyphForest
```

Call:

```
rxDForest(formula = Kyphosis ~ Age + Start + Number, data = kyphosis,
  cp = 0.01, nTree = 500, mTry = 3, seed = 10)
```

```

      Type of decision forest: class
      Number of trees: 500
No. of variables tried at each split: 3

      OOB estimate of error rate: 19.75%
Confusion matrix:
      Predicted
Kyphosis  absent present class.error
absent      56      8    0.1250000
present     8      9    0.4705882
```

While decision forests do not produce a unified model, as logistic regression and decision trees do, they do produce reasonable predictions for each data point. In this case, we can obtain predictions using `rxPredict` as follows:

```
dfPreds <- rxPredict(kyphForest, data=kyphosis)
```

Comparing this to the `Kyphosis` variable in the original `kyphosis` data, we see that approximately 88 percent of cases are classified correctly:

```
sum(as.character(dfPreds[,1]) ==
  as.character(kyphosis$Kyphosis))/81

[1] 0.8765432
```

11.2 A Simple Regression Forest

As a simple example of a regression forest, consider the classic `stackloss` data set, containing observations from a chemical plant producing nitric acid by the oxidation of ammonia, and let's fit the stack loss (`stack.loss`) using air flow (`Air.Flow`), water temperature (`Water.Temp`), and acid concentration (`Acid.Conc.`) as predictors:

```
stackForest <- rxDForest(stack.loss ~ Air.Flow + Water.Temp + Acid.Conc.,
  data=stackloss, nTree=200, mTry=2)
stackForest
```

138 A Larger Regression Forest Model

```
Call:
rxDForest(formula = stack.loss ~ Air.Flow + Water.Temp + Acid.Conc.,
  data = stackloss, maxDepth = 3, nTree = 200, mTry = 2)

Type of decision forest: anova
Number of trees: 200
No. of variables tried at each split: 2

Mean of squared residuals: 44.54992
% Var explained: 65
```

11.3 A Larger Regression Forest Model

As a more complex example, we return to the `censusWorkers` data to which we earlier fit a decision tree. We will create a regression forest predicting wage income from age, sex, and weeks worked, using the `perwt` variable as probability weights (note that we retain the `maxDepth` and `minBucket` parameters from our earlier decision tree example):

```
censusWorkers <- file.path(rxGetOption("sampleDataDir"),
  "CensusWorkers.xdf")
rxGetInfo(censusWorkers, getVarInfo=TRUE)
incForest <- rxDForest(incwage ~ age + sex + wkswork1, pweights = "perwt",
  maxDepth = 3, minBucket = 30000, mTry=2, nTree=200, data = censusWorkers)
incForest
```

```
Call:
rxDForest(formula = incwage ~ age + sex + wkswork1, data = censusData,
  pweights = "perwt", maxDepth = 5, nTree = 200, mTry = 2)

Type of decision forest: anova
Number of trees: 200
No. of variables tried at each split: 2

Mean of squared residuals: 1458969472
% Var explained: 11
```

.

11.4 Large Data Decision Forest Models

As with decision trees, scaling decision forests to very large data sets should be done with caution—the wrong choice of model parameters can easily lead to models that take hours or longer to estimate, even in a distributed computing environment, or that simply cannot be fit at all. For non-binary classification problems, as with decision trees, categorical predictors should have a small to moderate number of levels.

As an example of a large data classification forest, consider the following simple model using the 7% subsample of the full airline data (this uses the variable *ArrDel15* indicating flights with an arrival delay of 15 minutes or more):

```
bigDataDir <- "C:/Revolution/Data"
sampleAirData <- file.path(bigDataDir, "AirOnTime7Pct.xdf")
airlineForest <- rxDForest(ArrDel15 ~ CRSDepTime + DayOfWeek,
  data = sampleAirData, blocksPerRead = 30, maxDepth = 5,
  nTree=20, mTry=2, method="class", seed = 8)
```

This yields the following:

```
airlineForest

Call:
rxDForest(formula = ArrDel15 ~ CRSDepTime + DayOfWeek, data = sampleAirData,
  method = "class", maxDepth = 5, nTree = 20, mTry = 2, seed = 8,
  blocksPerRead = 30)

Type of decision forest: class
Number of trees: 20
No. of variables tried at each split: 2

OOB estimate of error rate: 20.01%
Confusion matrix:
Predicted
ArrDel15  FALSE TRUE class.error
FALSE 8147274    0          0
TRUE  2037941    0          1
```

Looking at the fitted object's forest component, we see that a number of the fitted trees do not split at all:

```
airlineForest$forest

[[1]]
Number of valid observations: 6440007
Number of missing observations: 3959748

Tree representation:
n= 10186709

node), split, n, loss, yval, (yprob)
* denotes terminal node

1) root 10186709 2038302 FALSE (0.7999057 0.2000943) *

[[2]]
Number of valid observations: 6440530
Number of missing observations: 3959225

Tree representation:
n= 10186445

node), split, n, loss, yval, (yprob)
* denotes terminal node

1) root 10186445 2038249 FALSE (0.7999057 0.2000943) *
```

140 Large Data Decision Forest Models

```

[[3]]
...

[[6]]
Number of valid observations: 6439485
Number of missing observations: 3960270

Tree representation:
n= 10186656

node), split, n, loss, yval, (yprob)
      * denotes terminal node

1) root 10186656 2038291 FALSE (0.7999057 0.2000943) *

[[7]]
Number of valid observations: 6439307
Number of missing observations: 3960448

Tree representation:
n= 10186499

node), split, n, loss, yval, (yprob)
      * denotes terminal node

1) root 10186499 2038260 FALSE (0.7999057 0.2000943) *
. . .

```

This may well be because our response is extremely unbalanced--that is, the percentage of flights that are late by 15 minutes or more is quite small. We can tune the fit by providing a *loss matrix*, which allows us to penalize certain predictions in favor of others. You specify the loss matrix using the *parms* argument, which takes a list with named components. The loss component is specified as either a matrix, or equivalently, a vector that can be coerced to a matrix. In the binary classification case, it can be useful to start with a loss matrix with a penalty roughly equivalent to the ratio of the two classes. So, in our case we know that the on-time flights outnumber the late flights approximately 4 to 1 :

```

airlineForest2 <- rxDForest(ArrDel15 ~ CRSDepTime + DayOfWeek,
  data = sampleAirData, blocksPerRead = 30, maxDepth = 5, seed = 8,
  nTree=20, mTry=2, method="class", parms=list(loss=c(0,4,1,0)))

Call:
rxDForest(formula = ArrDel15 ~ CRSDepTime + DayOfWeek, data = sampleAirData,
  method = "class", parms = list(loss = c(0, 4, 1, 0)), maxDepth = 5,
  nTree = 20, mTry = 2, seed = 8, blocksPerRead = 30)

Type of decision forest: class
Number of trees: 20
No. of variables tried at each split: 2

OOB estimate of error rate: 42.27%
Confusion matrix:
Predicted
ArrDel15  FALSE  TRUE class.error
FALSE 4719374 3427900 0.420742
TRUE 877680 1160261 0.430670

```

This model no longer predicts all flights as on time, but now over-predicts late flights. Adjusting the loss matrix again, this time reducing the penalty, yields the following:

```
Call:
rxDForest(formula = ArrDel15 ~ CRSDepTime + DayOfWeek, data = sampleAirData,
  method = "class", parms = list(loss = c(0, 3, 1, 0)), maxDepth = 5,
  nTree = 20, mTry = 2, seed = 8, blocksPerRead = 30)
```

```

      Type of decision forest: class
      Number of trees: 20
No. of variables tried at each split: 2
```

```

      OOB estimate of error rate: 30.15%
Confusion matrix:
      Predicted
ArrDel15  FALSE    TRUE class.error
FALSE 6465439 1681835  0.2064292
TRUE  1389092  648849  0.6816154
```

Chapter 12.

Estimating Correlation and Variance/Covariance Matrices

The `rxCovCor` function in RevoScaleR calculates the covariance, correlation, or sum of squares/cross-product matrix for a set of variables in an .xdf file or data frame. The size of these matrices is determined by the number of variables rather than the number of observations, so typically the results can easily fit into memory in R. A broad category of analyses can be computed from some form of a cross-product matrix, for example, factor analysis and principal components.

A cross-product matrix is a matrix of the form $X'X$, where X represents an arbitrary set of raw or standardized variables. More generally, this is a matrix of the form $X'WX$, where W is a diagonal weighting matrix.

12.1 Computing Cross-Product Matrices

While `rxCovCor` is the primary tool for computing covariance, correlation, and other cross-product matrices, you will seldom call it directly. Instead, it is usually simpler to use one of the following convenience functions:

- `rxCov`: Use `rxCov` to return the covariance matrix
- `rxCor`: Use `rxCor` to return the correlation matrix

- **rxSSCP:** Use `rxSSCP` to return the augmented cross-product matrix, that is, we first add a column of 1's (if no weights are specified) or a column equaling the square root of the weights to the data matrix, and then compute the cross-product.

12.2 Computing a Correlation Matrix for Use in Factor Analysis

The 5% sample of the U.S. 2000 census has over 14 million observations. In this example we will compute the correlation matrix for 16 variables derived from variables in the data set for individuals over the age of 20. This correlation matrix is then used as input into the standard R factor analysis function, `factanal`.

First, we specify the name and location of the data set:

```
bigDataDir <- "C:/Revolution/Data"
bigCensusData <- file.path(bigDataDir, "Census5PCT2000.xdf")
```

Next, we can take a quick look at some basic demographic and socio-economic variables by calling `rxSummary`. (For more information on variables in the census data, see <http://usa.ipums.org/usa-action/variables/group>.) Throughout the analysis we will use the probability weighting variable, `perwt`, and restrict the analysis to people over the age of 20.

```
rxSummary(~phone + speakeng + wkswork1 + incwelfr + incss + educrec + metro +
  ownershd + marst + lingisol + nfams + yrsusal + movedin + racwht + age,
  data = bigCensusData, blocksPerRead = 5, pweights = "perwt",
  rowSelection = age > 20)
```

This call provides summary information about the variables in this weighted sub-sample, including cell counts for factor variables:

```
Call:
rxSummary(formula = ~phone + speakeng + wkswork1 + incwelfr +
  incss + educrec + metro + ownershd + marst + lingisol + nfams +
  yrsusal + movedin + racwht + age, data = censusData, pweights = "perwt",
  rowSelection = age > 20, blocksPerRead = 5)
```

```
Summary Statistics Results for: ~phone + speakeng + wkswork1 + incwelfr
+ incss + educrec + metro + ownershd + marst + lingisol + nfams +
yrsusal + movedin + racwht + age
File name: C:\Revolution\Data\Census5PCT2000.xdf
Probability weights: perwt
Number of valid observations: 9822124
```

Name	Mean	StdDev	Min	Max	SumOfWeights	MissingWeights
wkswork1	32.068473	23.2438663	0	52	196971131	0
incwelfr	61.155293	711.0955602	0	25500	196971131	0
incss	1614.604835	3915.7717233	0	26800	196971131	0
nfams	1.163434	0.5375238	1	48	196971131	0
yrsusal	2.868573	9.0098343	0	90	196971131	0
age	46.813005	17.1797905	21	93	196971131	0

```
Category Counts for phone
Number of categories: 3
```

144 Computing a Correlation Matrix for Use in Factor Analysis

phone	Counts
N/A	5611380
No, no phone available	3957030
Yes, phone available	187402721

Category Counts for speakeng
Number of categories: 10

speakeng	Counts
N/A (Blank)	0
Does not speak English	2956934
Yes, speaks English...	0
Yes, speaks only English	162425091
Yes, speaks very well	17664738
Yes, speaks well	7713303
Yes, but not well	6211065
Unknown	0
Illegible	0
Blank	0

Category Counts for educrec
Number of categories: 10

educrec	Counts
N/A or No schooling	0
None or preschool	2757363
Grade 1, 2, 3, or 4	1439820
Grade 5, 6, 7, or 8	10180870
Grade 9	4862980
Grade 10	5957922
Grade 11	5763456
Grade 12	63691961
1 to 3 years of college	55834997
4+ years of college	46481762

Category Counts for metro
Number of categories: 5

metro	Counts
Not applicable	13829398
Not in metro area	32533836
In metro area, central city	32080416
In metro, area, outside central city	60836302
Central city status unknown	57691179

Category Counts for ownershd
Number of categories: 8

ownershd	Counts
N/A	5611380
Owned or being bought	0
Check mark (owns?)	0
Owned free and clear	40546259
Owned with mortgage or loan	94626060
Rents	0
No cash rent	3169987
With cash rent	53017445

Category Counts for marst
Number of categories: 6

marst	Counts
-------	--------

Married, spouse present	112784037
Married, spouse absent	5896245
Separated	4686951
Divorced	21474299
Widowed	14605829
Never married/single (N/A)	37523770

Category Counts for lingisol
Number of categories: 3

lingisol	Counts
N/A (group quarters/vacant)	5611380
Not linguistically isolated	182633786
Linguistically isolated	8725965

Category Counts for movedin
Number of categories: 7

movedin	Counts
NA	92708540
This year or last year	20107246
2-5 years ago	30328210
6-10 years ago	16959897
11-20 years ago	16406155
21-30 years ago	10339278
31+ years ago	10121805

Category Counts for racwht
Number of categories: 2

racwht	Counts
No	40684944
Yes	156286187

Next we will call the `rxCor` function, a convenience function for `rxCovCor` that returns just the Pearson's correlation matrix for the variables specified. We will make heavy use of the `transforms` argument to create a series of logical (or dummy) variables from factor variables to be used in the creation of the correlation matrix.

```
censusCor <- rxCor(formula=~poverty + noPhone + noEnglish + onSocialSecurity +
  onWelfare + working + inearn + noHighSchool + inCity + renter +
  noSpouse + langIsolated + multFamilies + newArrival + recentMove +
  white + sei + older,
  data = bigCensusData, pweightsb= "perwt", blocksPerRead = 5,
  rowSelection = age > 20,
  transforms= list(
    noPhone = phone == "No, no phone available",
    noEnglish = speakeng == "Does not speak English",
    working = wkswork1 > 20,
    onWelfare = incwelfr > 0,
    onSocialSecurity = incss > 0,
    noHighSchool =
      !(educrec %in%
        c("Grade 12", "1 to 3 years of college", "4+ years of college")),
    inCity = metro == "In metro area, central city",
    renter = ownershd %in% c("No cash rent", "With cash rent"),
    noSpouse = marst != "Married, spouse present",
    langIsolated = lingisol == "Linguistically isolated",
    multFamilies = nfams > 2,
    newArrival = yrsusa2 == "0-5 years",
```

146 Computing a Correlation Matrix for Use in Factor Analysis

```
recentMove = movedin == "This year or last year",
white = racwht == "Yes",
older = age > 64
))
```

The resulting correlation matrix is used as input into the factor analysis function provided by the stats package in R. In interpreting the results, note that the variable *poverty* represents family income as a percentage of a poverty threshold, so increases as family income increases. First, specify two factors:

```
censusFa <- factanal(covmat = censusCor, factors=2)
print(censusFa, digits=2, cutoff = .2, sort= TRUE)
```

This results in:

```
Call:
factanal(factors = 2, covmat = censusCor)

Uniquenesses:
      poverty      noPhone      noEnglish onSocialSecurity
      0.53         0.96         0.93         0.23
onWelfare      working
      0.96         0.51         0.68         noHighSchool 0.82
      inCity      renter
      0.97         0.79         0.90         langIsolated 0.90
multFamilies    newArrival recentMove      white
      0.96         0.93         0.97         0.88
      sei      older
      0.57         0.24

Loadings:
      Factor1 Factor2
onSocialSecurity 0.83 -0.29
working          -0.68
sei              -0.59 -0.29
older            0.82 -0.29
poverty          -0.28 -0.63
noPhone
noEnglish
onWelfare
inearn          -0.46 -0.34
noHighSchool    0.31  0.29
inCity
renter
noSpouse
langIsolated
multFamilies
newArrival
recentMove
white           -0.34

      Factor1 Factor2
SS loadings  2.60  1.67
Proportion Var 0.14  0.09
Cumulative Var 0.14  0.24
```

The degrees of freedom for the model is 118 and the fit was 0.6019

We can use the same correlation matrix to estimate three factors:

```
censusFa <- factanal(covmat = censusCor, factors=3)
print(censusFa, digits=2, cutoff = .2, sort= TRUE)
```

Call:

```
factanal(factors = 3, covmat = censusCor)
```

Uniquenesses:

poverty	noPhone	noEnglish	onSocialSecurity
0.49	0.96	0.72	0.24
onWelfare	working	inearn	noHighSchool
0.96	0.50	0.62	0.80
inCity	renter	noSpouse	langIsolated
0.96	0.80	0.90	0.59
multFamilies	newArrival	recentMove	white
0.96	0.79	0.97	0.88
sei	older		
0.56	0.22		

Loadings:

	Factor1	Factor2	Factor3
onSocialSecurity	0.87		
working	-0.62	0.34	
sei	-0.51	0.42	
older	0.88		
poverty		0.68	
inearn	-0.36	0.51	
noEnglish			0.53
langIsolated			0.64
noPhone			
onWelfare		-0.21	
noHighSchool	0.25	-0.26	0.26
inCity			
renter		-0.36	0.24
noSpouse		-0.30	
multFamilies			
newArrival			0.46
recentMove			
white		0.24	-0.24

	Factor1	Factor2	Factor3
SS loadings	2.41	1.50	1.18
Proportion Var	0.13	0.08	0.07
Cumulative Var	0.13	0.22	0.28

The degrees of freedom for the model is 102 and the fit was 0.343

12.3 Computing A Covariance Matrix for Principal Components Analysis

Principal components analysis, or PCA, is a technique closely related to factor analysis. PCA seeks to find a set of orthogonal axes such that the first axis, or *first principal component*, accounts for as much variability as possible and subsequent axes or components are chosen to maximize variance while maintaining orthogonality with previous axes. Principal components are typically computed either by a singular value decomposition of the data matrix or an eigenvalue decomposition of a covariance or correlation matrix; the latter permits us to use `rxCovCor` and its relatives with the standard R function `princomp`.

As an example, we will use the `rxCov` function to calculate a covariance matrix for the log of the iris data, and pass this to the `princomp` function; this will reproduce the example from pages 303-304 of *Modern Applied Statistics with S* (Venables & Ripley, 2002):

```
irisLog <- as.data.frame(lapply(iris[,1:4], log))
irisSpecies <- iris[,5]
irisCov <- rxCov(~ Sepal.Length + Sepal.Width + Petal.Length + Petal.Width,
  data=irisLog)
irisPca <- princomp(covmat=irisCov, cor=TRUE)
summary(irisPca)
```

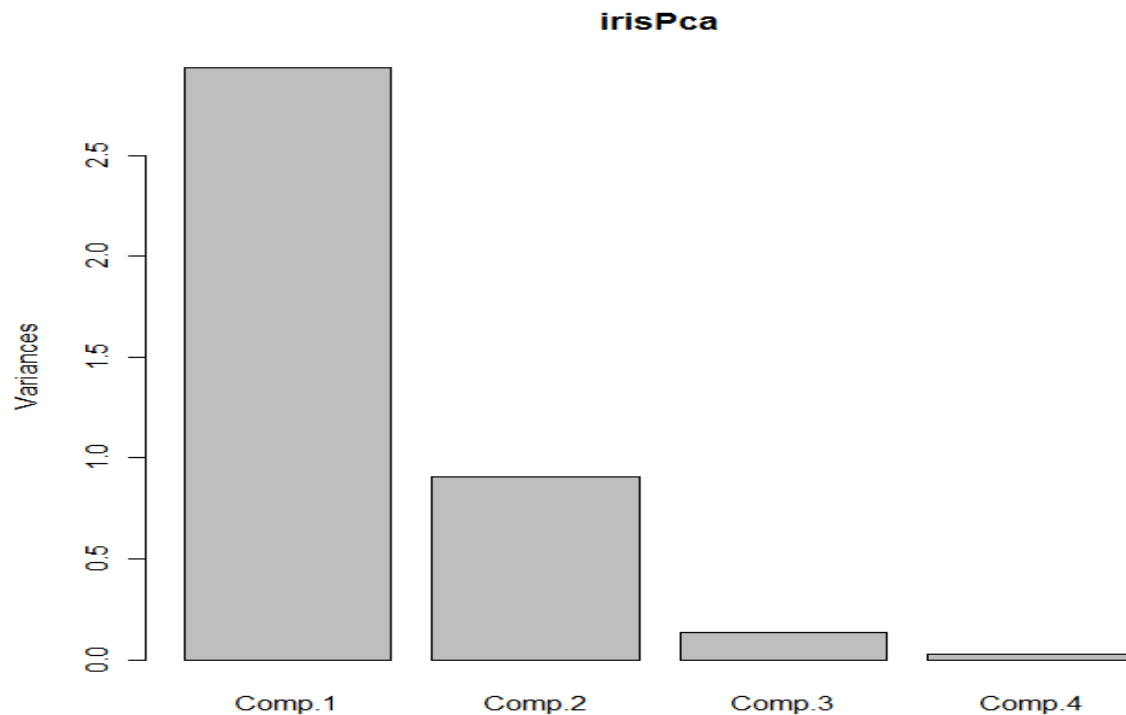
This yields the following:

```
Importance of components:
              Comp.1      Comp.2      Comp.3      Comp.4
Standard deviation  1.7124583  0.9523797  0.36470294  0.1656840
Proportion of Variance 0.7331284  0.2267568  0.03325206  0.0068628
Cumulative Proportion 0.7331284  0.9598851  0.99313720  1.0000000
```

The default plot method for objects of class `princomp` is a *screeplot*, which is a barplot of the variances of the principal components. We can obtain the plot as usual by calling `plot` with our principal components object:

```
plot(irisPca)
```

This yields the following plot:



Another useful bit of output is given by the loadings function, which returns a set of columns showing the linear combinations for each principal component:

```
loadings(irisPca)
```

Loadings:	Comp.1	Comp.2	Comp.3	Comp.4
Sepal.Length	0.504	-0.455	0.709	0.191
Sepal.Width	-0.302	-0.889	-0.331	
Petal.Length	0.577		-0.219	-0.786
Petal.Width	0.567		-0.583	0.580

	Comp.1	Comp.2	Comp.3	Comp.4
SS loadings	1.00	1.00	1.00	1.00
Proportion Var	0.25	0.25	0.25	0.25
Cumulative Var	0.25	0.50	0.75	1.00

You may have noticed that we supplied the flag `cor=TRUE` in the call to `princomp`; this tells `princomp` to use the correlation matrix rather than the covariance matrix to compute the principal components. We can obtain the same results by omitting the flag but submitting the correlation matrix as returned by `rxCor` instead:

```
irisCor <- rxCor(~ Sepal.Length + Sepal.Width + Petal.Length + Petal.Width,
  data=irisLog)
irisPca2 <- princomp(covmat=irisCor)
summary(irisPca2)
loadings(irisPca2)
plot(irisPca2)
```

12.4 A Large Data Principal Components Analysis

Stock market data for open, high, low, close, and adjusted close from 1962 to 2010 is available at <http://www.infochimps.com/datasets/nyse-daily-1970-2010-open-close-high-low-and-volume/downloads/15407>. The full data set includes 9.2 million observations of daily open-high-low-close data for some 2800 stocks. As you might expect, these data are highly correlated, and principal components analysis can be used for data reduction. We read the original data into an .xdf file, `NYSE_daily_prices.xdf`, using the same process we used in the *Getting Started* Guide to read our mortgage data (set `revoDataDir` to the full path to the NYSE directory containing the .csv files when you unpack the download):

```
bigDataDir <- "C:/Revolution/Data"
nyseCsvFiles <- file.path(bigDataDir, "NYSE_daily_prices", "NYSE",
  "NYSE_daily_prices_")

nyseXdf <- "NYSE_daily_prices.xdf"
append <- "none"
for (i in LETTERS)
{
  importFile <- paste(nyseCsvFiles, i, ".csv", sep="")
  rxImport(importFile, nyseXdf, append=append)
  append <- "rows"
```

150 A Large Data Principal Components Analysis

```
}
```

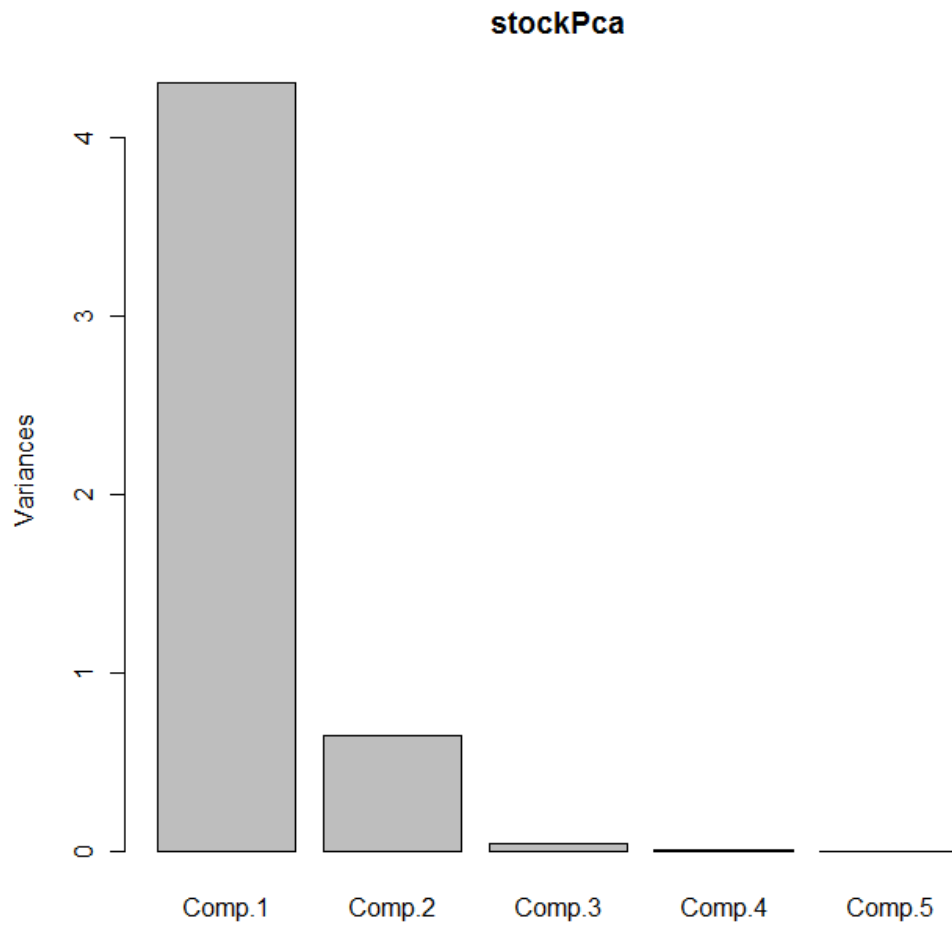
Once we have our .xdf file, we proceed as before:

```
stockCor <- rxCor(~ stock_price_open + stock_price_high +  
  stock_price_low + stock_price_close +  
  stock_price_adj_close, data="NYSE_daily_prices.xdf")  
stockPca <- princomp(covmat=stockCor)  
summary(stockPca)  
loadings(stockPca)  
plot(stockPca)
```

This yields the following output:

```
> summary(stockPca)  
Importance of components:  
                Comp.1      Comp.2      Comp.3      Comp.4  
Standard deviation  2.0756631  0.8063270  0.197632281  0.0454173922  
Proportion of Variance 0.8616755  0.1300327  0.007811704  0.0004125479  
Cumulative Proportion 0.8616755  0.9917081  0.999519853  0.9999324005  
                Comp.5  
Standard deviation  1.838470e-02  
Proportion of Variance 6.759946e-05  
Cumulative Proportion 1.000000e+00  
> loadings(stockPca)  
  
Loadings:  
                Comp.1 Comp.2 Comp.3 Comp.4 Comp.5  
stock_price_open  -0.470 -0.166  0.867  
stock_price_high  -0.477 -0.151 -0.276  0.410 -0.711  
stock_price_low   -0.477 -0.153 -0.282  0.417  0.704  
stock_price_close -0.477 -0.149 -0.305 -0.811  
stock_price_adj_close -0.309  0.951  
  
                Comp.1 Comp.2 Comp.3 Comp.4 Comp.5  
SS loadings      1.0    1.0    1.0    1.0    1.0  
Proportion Var   0.2    0.2    0.2    0.2    0.2  
Cumulative Var   0.2    0.4    0.6    0.8    1.0
```

The screeplot is shown below:



Between them, the first two principal components explain 99% of the variance; we can therefore replace the five original variables by these two principal components with no appreciable loss of information.

Chapter 13.

Clustering

Clustering is the general name for any of a large number of classification techniques that involve assigning observations to membership in one of two or more clusters on the basis of some distance metric.

13.1 K-means Clustering

K-means clustering is a classification technique that groups observations of numeric data using one of several *iterative relocation* algorithms—that is, starting from some initial classification, which may be random, points are moved from cluster to another so as to minimize sums of squares. In RevoScaleR, the algorithm used is that of Lloyd (1982).

To perform k-means clustering with RevoScaleR, use the `rxKmeans` function.

13.1.1 Clustering the Airline Data

As a first example of k-means clustering, we will cluster the arrival delay and scheduled departure time in the airline data 7% subsample. To start, we extract variables of interest into a new working data set to which we'll be writing additional information :

```
bigDataDir <- "C:/Revolution/Data"
sampleAirData <- file.path(bigDataDir, "AirOnTime7Pct.xdf")
rxDataStep(inData = sampleAirData, outFile = "AirlineDataClusterVars.xdf",
  varsToKeep=c("DayOfWeek", "ArrDelay", "CRSDepTime", "DepDelay"))
```


We specify the variables to cluster as a formula, and specify the number of clusters we'd like; initial centers for these clusters are then chosen at random.

```
kclus1 <- rxKmeans(formula= ~ArrDelay + Distance,
  data = "AirlineDataClusterVars.xdf",
  outFile = "airlineDataClusterVars.xdf", numClusters=5)
kclus1
```

This produces the following output (because the initial centers are chosen at random, your output will probably look different):

```
Call:
rxKmeans(formula = ~ArrDelay + CRSDepTime, data = "AirlineDataClusterVars.xdf",
  outFile = "AirlineDataClusterVars.xdf", numClusters = 5)

Data: "AirlineDataClusterVars.xdf"
Number of valid observations: 10186272
Number of missing observations: 213483
Clustering algorithm:

K-means clustering with 5 clusters of sizes 922985, 38192, 4772791, 261779,
4190525

Cluster means:
  ArrDelay CRSDepTime
1  45.258179  14.86596
2 275.363820  14.81432
3 -10.284426  13.08375
4 118.365205  15.52079
5   7.803893  13.53811

Within cluster sum of squares by cluster:
      1      2      3      4      5
223220709 501736748 354763376 233533349 312403604

Available components:
[1] "centers"      "size"          "withinss"      "valid.obs"
[5] "missing.obs"  "numIterations" "tot.withinss"  "totss"
[9] "betweenss"    "cluster"       "params"        "formula"
[13] "call"
```

The value returned by `rxKmeans` is a list similar to the list returned by the standard R `kmeans` function. The printed output shows a subset of this information, including the number of valid and missing observations, the cluster sizes, the cluster centers, and the within-cluster sums of squares.

The cluster membership component is returned if the input is a data frame, but if the input is an `.xdf` file, cluster membership is returned only if `outFile` is specified, in which case it is returned not as part of the return object, but as a column in the specified file. In our example, we specified an `outFile`, and we see the cluster membership variable when we look at the file with `rxGetInfo`:

```
rxGetInfo("AirlineDataClusterVars.xdf", getVarInfo=TRUE)

File name: AirlineDataClusterVars.xdf
```

154 K-means Clustering

```
Number of observations: 10399755
Number of variables: 5
Number of blocks: 19
Compression type: zlib
Variable information:
Var 1: DayOfWeek
      7 factor levels: Mon Tues Wed Thur Fri Sat Sun
Var 2: ArrDelay, Type: integer, Low/High: (-1233, 2453)
Var 3: CRSDepTime, Type: numeric, Storage: float32, Low/High: (0.0000, 24.0000)
Var 4: DepDelay, Type: integer, Low/High: (-1199, 2467)
Var 5: .rxCluster, Type: integer, Low/High: (1, 5)
```

13.1.2 Using the Cluster Membership Information

A common follow-up to clustering is to use the cluster membership information to see whether a given model varies appreciably from cluster to cluster. Since we can use the `rowSelection` argument to extract a single cluster on the fly, there is no need to sort the data first. As an example, we fit our original linear model of `ArrDelay` by `DayOfWeek` for two of the clusters:

```
clust1Lm <- rxLinMod(ArrDelay ~ DayOfWeek, "AirlineDataClusterVars.xdf",
  rowSelection = .rxCluster == 1 )
clust5Lm <- rxLinMod(ArrDelay ~ DayOfWeek, "AirlineDataClusterVars.xdf",
  rowSelection = .rxCluster == 5)
summary(clust1Lm)
summary(clust5Lm)
```

Looking at the summary for `clust1Lm` shows the following:

```
Call:
rxLinMod(formula = ArrDelay ~ DayOfWeek, data = "AirlineDataClusterVars.xdf",
  rowSelection = .rxCluster == 1)

Linear Regression Results for: ArrDelay ~ DayOfWeek
File name: AirlineDataClusterVars.xdf
Dependent variable(s): ArrDelay
Total independent variables: 8 (Including number dropped: 1)
Number of valid observations: 922985
Number of missing observations: 0

Coefficients: (1 not defined because of singularities)
              Estimate Std. Error t value Pr(>|t|)
(Intercept)  45.21591    0.04237 1067.199 2.22e-16 ***
DayOfWeek=Mon   0.23053    0.05893   3.912 9.16e-05 ***
DayOfWeek=Tues -0.06496    0.05968  -1.089  0.2764
DayOfWeek=Wed   0.10139    0.05869   1.727  0.0841 .
DayOfWeek=Thur   0.06098    0.05708   1.068  0.2854
DayOfWeek=Fri   0.23222    0.05660   4.103 4.08e-05 ***
DayOfWeek=Sat  -0.43444    0.06364  -6.827 8.68e-12 ***
DayOfWeek=Sun   Dropped      Dropped  Dropped  Dropped
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 14.89 on 922978 degrees of freedom
Multiple R-squared: 0.0001705
Adjusted R-squared: 0.000164
F-statistic: 26.24 on 6 and 922978 DF, p-value: < 2.2e-16
Condition number: 12.8655
```

Similarly, the summary for `clust5Lm` shows the following:

```
Call:
rxLinMod(formula = ArrDelay ~ DayOfWeek, data = "AirlineDataClusterVars.xdf",
          rowSelection = .rxCluster == 5)
```

```
Linear Regression Results for: ArrDelay ~ DayOfWeek
File name: AirlineDataClusterVars.xdf
Dependent variable(s): ArrDelay
Total independent variables: 8 (Including number dropped: 1)
Number of valid observations: 4190525
Number of missing observations: 0
```

```
Coefficients: (1 not defined because of singularities)
```

	Estimate	Std. Error	t value	Pr(> t)	
(Intercept)	7.808093	0.009593	813.960	2.22e-16	***
DayOfWeek=Mon	-0.131001	0.013320	-9.835	2.22e-16	***
DayOfWeek=Tues	-0.228087	0.013374	-17.055	2.22e-16	***
DayOfWeek=Wed	-0.035954	0.013292	-2.705	0.00683	**
DayOfWeek=Thur	0.231958	0.013170	17.613	2.22e-16	***
DayOfWeek=Fri	0.313961	0.013171	23.838	2.22e-16	***
DayOfWeek=Sat	-0.257716	0.014036	-18.361	2.22e-16	***
DayOfWeek=Sun	Dropped	Dropped	Dropped	Dropped	

```
---
```

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

```
Residual standard error: 7.238 on 4190518 degrees of freedom
Multiple R-squared:  0.0007911
Adjusted R-squared:  0.0007897
F-statistic: 553 on 6 and 4190518 DF,  p-value: < 2.2e-16
Condition number: 12.0006
```

Chapter 14.

Converting RevoScaleR Model Objects for Use in PMML

The objects returned by RevoScaleR predictive analytics functions have similarities to objects returned by the predictive analytics functions provided by base and recommended packages in R. A key difference between these two types of model objects is that RevoScaleR model objects typically do not contain any components that have the same length as the number of rows in the original data set. For example, if you use base R's `lm` function to estimate a linear model, the result object contains not only all of the data used to estimate the model, but components such as residuals and fitted.values that contain values corresponding to every observation in the data set. For estimating models using big data, this is not appropriate.

However, there is overlap in the model object components that can be very useful when working with other packages. For example, the `pmm1` package will generate PMML (Predictive Model Markup Language) code for a number of R model types. PMML is an XML-base language which allows users to share models between PMML compliant applications. For more information about PMML, visit <http://www.dmg.org>.

RevoScaleR provides a set of coercion methods to convert a RevoScaleR model object to a standard R model object: `as.lm`, `as.glm`, `as.rpart`, and `as.kmeans`. As suggested above, these coerced model objects do not have all of the information available in a standard R model object, but do contain information about the fitted model that is similar to standard R.

For example, let's create an `rxLinMod` object by estimating a linear regression on a very large dataset: the airline data with almost 150 million observations.

```
bigDataDir <- "C:/Revolution/Data"
bigAirData <- file.path(bigDataDir, "AirOnTime87to12/AirOnTime87to12.xdf")

# About 150 million observations
rxLinModObj <- rxLinMod(ArrDelay~Year + DayOfWeek, data = bigAirData,
  blocksPerRead = 10)
```

Next, the R `pmml` package should be downloaded and installed from CRAN. After you have done so, generate the RevoScaleR `rxLinMod` object, and use the `as.lm` method when generating the PMML output:

```
library(pmml)
pmml(as.lm(rxLinModObj))
```

The generated output looks as follows:

```
> pmml(as.lm(rxLinModObj))
<PMML version="4.1" xmlns="http://www.dmg.org/PMML-4_1"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.dmg.org/PMML-4_1 http://www.dmg.org/v4-1/pmml-4-1.xsd">
  <Header copyright="Copyright (c) 2013 sue" description="Linear Regression
Model">
    <Extension name="user" value="yourname" extender="Rattle/PMML"/>
    <Application name="Rattle/PMML" version="1.3"/>
    <Timestamp>2013-09-05 11:39:28</Timestamp>
  </Header>
  <DataDictionary numberOfFields="3">
    <DataField name="ArrDelay" optype="continuous" dataType="double"/>
    <DataField name="Year" optype="continuous" dataType="double"/>
    <DataField name="DayOfWeek" optype="categorical" dataType="string">
      <Value value="Mon"/>
      <Value value="Tues"/>
      <Value value="Wed"/>
      <Value value="Thur"/>
      <Value value="Fri"/>
      <Value value="Sat"/>
      <Value value="Sun"/>
    </DataField>
  </DataDictionary>
  <RegressionModel modelName="Linear_Regression_Model" functionName="regression"
algorithmName="least squares" targetFieldName="ArrDelay">
    <MiningSchema>
      <MiningField name="ArrDelay" usageType="predicted"/>
      <MiningField name="Year" usageType="active"/>
      <MiningField name="DayOfWeek" usageType="active"/>
    </MiningSchema>
    <Output>
      <OutputField name="Predicted_ArrDelay" feature="predictedValue"/>
    </Output>
    <RegressionTable intercept="112.856953212332">
      <NumericPredictor name="Year" exponent="1" coefficient="-
0.0533799688606163"/>
      <CategoricalPredictor name="DayOfWeek" value="Mon"
coefficient="0.303666227836942"/>
```

```

    <CategoricalPredictor name="DayOfWeek" value="Tues" coefficient="-
0.593700918634743"/>
    <CategoricalPredictor name="DayOfWeek" value="Wed"
coefficient="0.473693749859764"/>
    <CategoricalPredictor name="DayOfWeek" value="Thur"
coefficient="2.3393087933048"/>
    <CategoricalPredictor name="DayOfWeek" value="Fri"
coefficient="2.91671831592945"/>
    <CategoricalPredictor name="DayOfWeek" value="Sat" coefficient="-
2.31671307739631"/>
    <CategoricalPredictor name="DayOfWeek" value="Sun" coefficient="0"/>
  </RegressionTable>
</RegressionModel>
</PMML>

```

Similarly, `rxLogit` and `rxGlm` functions have `as.glm` methods:

```

form <- case ~ age + parity + spontaneous + induced
rxLogitObj <- rxLogit(form, data =infert)
pmml(as.glm(rxLogitObj))

rxGlmObj <- rxGlm(form, data=infert, family = binomial())
pmml(as.glm(rxGlmObj))

```

RevoScaler's `rxKmeans` objects can be coerced to `kmeans` objects:

```

set.seed(17)
irow <- unique(sample.int(nrow(women), 4L,
  replace = TRUE))[seq(2)]
centers <- women[irow,, drop = FALSE]
rxKmeansObj <- rxKmeans(~height + weight, data = women,
  centers = centers)
pmml(as.kmeans(rxKmeansObj))

```

And RevoScaler's `rxDTree` objects can be coerced to `rpart` objects. (Note that `rpart` is a recommended package that you may need to load.)

```

library(rpart)
method <- "class"
form <- Kyphosis ~ Number + Start
parms <- list(prior = c(0.8, 0.2), loss = c(0, 2, 3, 0),
  split = "gini")
control <- rpart.control(minsplit = 5, minbucket = 2, cp = 0.01,
  maxdepth = 10, maxcompete = 4, maxsurrogate = 5,
  usesurrogate = 2, surrogatestyle = 0, xval = 0)
cost <- 1 + seq(length(attr(terms(form), "term.labels")))

rxDTreeObj <- rxDTree(formula = Kyphosis ~ Number + Start,
  data = kyphosis, method = method, parms = parms,
  control = control, cost = cost)
pmml(as.rpart(rxDTreeObj))

```

Chapter 15.

Data Sources

We briefly encountered data sources in Chapter 2 when importing data from external-format data files. In this chapter, we describe how to create a variety of data sources and use them with the RevoScaleR High Performance Analytics functions discussed in the previous chapters. For occasional analysis of external data, such as a monthly regression on the data returned by a given database query, running the analysis on the original data source can save you some time. However, when you will be performing repeated analysis of a single data set, it will almost always be faster for you to import the data into the .xdf format and run your analyses on the the .xdf data source.

This chapter also describes how to use .xdf data sources to easily access your .xdf file in blocks so that you can write your own “chunking” algorithms.

15.1 Using Data Sources with Analysis Functions

The following table was shown in Chapter 2 as the set of constructors for data sources:

Source Data	Data Source Constructor
Text (fixed-format or delimited)	RxTextData
SAS	RxSasData
SPSS	RxSpssData
Database (must have appropriate ODBC driver installed)	RxOdbcData
Teradata Database	RxTeradata

Suppose, for example, that you would like to use *RxLinMod* to perform a linear regression on the data in the SAS file `claims.sas7bdat`. You would first create an *RxSasData* data source as follows:

```
inFileSAS <- file.path(rxGetOption("sampleDataDir"), "claims.sas7bdat")
sourceDataSAS <- RxSasData(inFileSAS, stringsAsFactors=TRUE)
```

Once we have the data source, we can remind ourselves of the variables in the data by calling the *names* function:

```
names(sourceDataSAS)

[1] "RowNum" "age" "car_age" "type" "cost" "number"
```

We can now compute our desired regression using our data source as the data argument to *rxLinMod*:

```
rxLinMod(cost ~ age + car_age, data = sourceDataSAS)

Rows Read: 128, Total Rows Processed: 128, Total Chunk Time: 0.003 seconds
Computation time: 0.014 seconds.
Call:
rxLinMod(formula = cost ~ age + car_age, data = sourceDataSAS)

Linear Regression Results for: cost ~ age + car_age
Data: sourceDataSAS (RxSasData Data Source)
File name:
  C:/Revolution/R-Enterprise-7.0/
  R-3.0.2/library/RevoScaleR/SampleData/claims.sas7bdat
```



```

Dependent variable(s): cost
Total independent variables: 13 (Including number dropped: 2)
Number of valid observations: 123
Number of missing observations: 5

```

```

Coefficients:
               cost
(Intercept) 117.38544
age=17-20    88.15174
age=21-24    34.15903
age=25-29    54.68750
age=30-34     2.93750
age=35-39   -20.77430
age=40-49     1.68750
age=50-59    63.12500
age=60+      Dropped
car_age=0-3 159.30531

```

Similarly, you could use the SPSS version of the claims data as follows:

```

inFileSpss <- file.path(rxGetOption("sampleDataDir"), "claims.sav")
sourceDataSpss <- RxSpssData(inFileSpss, stringsAsFactors=TRUE)
rxLinMod(cost ~ age + car_age, data=sourceDataSpss)

```

When you specify `stringsAsFactors=TRUE`, as we did here, `RxLinMod` will automatically generate factor levels on a first come, first served basis before performing the regression—thus requiring two passes through the data. For a small data set like this one, this is not too big a computational hit, but if you will be repeating the analysis on a very large data set, this can be significant slowdown. You can help things by using the `colInfo` argument to the data source constructor to specify a complete set of factor levels, as described in Chapter 2. And, as mentioned before, you can save time on repeated analyses by importing in the xdf format before performing any analysis.

15.2 Working with an Xdf Data Source

To create an .xdf data source for reading, you can use the `RxXdfData` function. For example, the following creates a data source from the built-in claims.xdf data set:

```

claimsPath <- file.path(rxGetOption("sampleDataDir"), "claims.xdf")
claimsDs <- RxXdfData(claimsPath)

```

Use the open method `rxOpen` to open the data source:

```
rxOpen(claimsDs)
```

Use the method `rxReadNext` to read the next block of data from the data source:

```
claims <- rxReadNext(claimsDs)
```

Use the `rxClose` method to close the data source:

```
rxClose(claimsDs)
```

15.3 Using an Xdf Data Source with biglm

Since data sources for xdf files read data in chunks, it is a good match for the CRAN package *biglm*. The *biglm* package does a linear regression on an initial chunk of data, then updates the results with subsequent chunks. Below is a function that loops through an xdf file object and creates and updates the *biglm* results.

```
require(biglm)
biglmxdf <- function(dataSource, formula)
{
  moreData <- TRUE
  df <- rxReadNext(dataSource)
  biglmRes <- biglm(formula, df)
  while (moreData)
  {
    df <- rxReadNext(dataSource)
    if (length(df) != 0)
    {
      biglmRes <- update(biglmRes, df)
    }
    else
    {
      moreData <- FALSE
    }
  }
  return(biglmRes)
}
```

To use the function, we first open the data file. For example, we can again use the large airline data set *AirOnTime87to12.xdf* (if you have downloaded the data set, modify the first line below to reflect your local path):

```
bigDataDir <- "C:/Revolution/Data"
bigAirData <- file.path(bigDataDir, "AirOnTime87to12/AirOnTime87to12.xdf")
dataSource <- RxXdfData(bigAirData,
  varsToKeep = c("DayOfWeek", "DepDelay", "ArrDelay"), blocksPerRead = 15)
rxOpen(dataSource)
```

Then we will time the computation, doing the regression for all the rows— 148,619,655 if you are using the full data set. (Note that in this case it will take at least 5 minutes, even on a very fast machine.)

```
system.time(bigLmRes <- biglmxdf(dataSource, ArrDelay~DayOfWeek))
rxClose(dataSource)
```

We can see the coefficients by looking at a summary of the object returned:

```
summary(bigLmRes)
```

It is, of course, much faster to compute a linear model using the *rxLinMod* function, but the *biglm* package provides alternative methods of computation.

Chapter 16.

Transform Functions

In previous chapters, we have used the `transforms` argument and inline transformations within formulas whenever we needed to specify a data transformation, and performed row selection with the `rowSelection` argument. In this chapter, we introduce another method for performing transformations within RevoScaleR that is extremely powerful and can be used both for transforming variables and for creating row selection variables. This method is the use of transform functions.

A transform function is a function that takes as input a list of variables and returns a list of (possibly different) variables. If defined and specified (using the `transformFunc` argument to most RevoScaleR functions), a transform function is evaluated before any transformations specified in the `transforms`, `rowSelection`, or formula arguments. The list of variables to be passed to the transform function is specified as the `transformVars` argument.

16.1 Creating Variables

Suppose we want to extract five variables from the `CensusWorkers` data set, but also add a factor variable based on the integer variable `age`. We considered this example in Chapter 3, when we used the `transforms` argument to create the new variable. We can also use a transform function to create new variables. For example, to create our factor variable, we can create the following function:

164 Creating Variables

```
ageTransform <- function(dataList)
{
  dataList$ageFactor <- cut(dataList$age, breaks=seq(from = 20, to = 70,
                                                    by = 5), right = FALSE)
  return(dataList)
}
```

We can test the function by reading an arbitrary chunk out of the data set. For efficiency reasons, the data passed to the transformation function is stored as a list rather than a data frame, so when reading from the .xdf file we set the `returnDataFrame` argument to `FALSE` to emulate this behavior. Since we only use the variable `age` in our transformation function, we restrict the variables extracted to that.

```
censusWorkers <- file.path(rxGetOption("sampleDataDir"), "CensusWorkers.xdf")
testData <- rxReadXdf(file = censusWorkers, startRow = 100, numRows = 10,
                     returnDataFrame = FALSE, varsToKeep = c("age"))

as.data.frame(ageTransform(testData))
```

The resulting list of data (displayed as a data frame) shows us that our transformations are working as expected:

```
> as.data.frame(ageTransform(testData))
  age ageFactor
1  20   [20,25)
2  48   [45,50)
3  44   [40,45)
4  29   [25,30)
5  28   [25,30)
6  43   [40,45)
7  20   [20,25)
8  23   [20,25)
9  32   [30,35)
10 42   [40,45)
```

In doing a data step operation, RevoScaleR reads in a chunk of data read from the original data set, including only the variables indicated in `varsToKeep`, or omitting variables specified in `varsToDrop`. It then passes the variables needed for data transformations back to R for manipulation. We specify the variables needed to process the transformation in the `transformVars` argument. Including extra variables does not alter the analysis, but it does reduce the efficiency of the data step. In this case, since `ageFactor` depends only on the `age` variable for its creation, the `transformVars` argument needs to specify just that:

```
rxDataStep(inData = censusWorkers, outFile = "newCensusWorkers",
           varsToDrop = c("state"), transformFunc = ageTransform,
           transformVars=c("age"), overwrite=TRUE)
```

The `rxGetInfo` function reveals the added and dropped variables:

```
rxGetInfo("newCensusWorkers", getVarInfo = TRUE)

File name: C:\YourOutputPath\newCensusWorkers.xdf
```

```

Number of rows: 351121
Number of variables: 6
Number of blocks: 6
Compression type: zlib
Variable information:
Var 1: age, Age
      Type: integer, Low/High: (20, 65)
Var 2: incwage, Wage and salary income
      Type: integer, Low/High: (0, 354000)
Var 3: perwt, Type: integer, Low/High: (2, 168)
Var 4: sex, Sex
      2 factor levels: Male Female
Var 5: wkswork1, Weeks worked last year
      Type: integer, Low/High: (21, 52)
Var 6: ageFactor
      10 factor levels: [20,25) [25,30) [30,35) [35,40) [40,45) [45,50)
                       [50,55) [55,60) [60,65) [65,70)

```

16.2 Using Additional Objects or Data in a Transform Function

It is sometimes useful to access additional information from within a transform function. For example, you might want to match additional data in the process of creating new variables. Transform functions are evaluated in a “sterilized” environment which includes the parent environment of the function closure. To provide access to additional data within the function, you can use the *transformObjects* argument.

For example, suppose you would like to estimate a linear model using wage income as the dependent variable, and want to include state-level of per capita expenditure on education as one of the independent variables. We can define a named vector to contain this state-level data as follows:

```
educExpense <- c(Connecticut=1795.57, Washington=1170.46, Indiana = 1289.66)
```

We can then define a transform function that uses this information as follows:

```

transformFunc <- function(dataList)
{
  # Match each individual's state and add the variable for educ. exp.
  dataList$stateEducExpPC = educExp[match(dataList$state, names(educExp))]
  return(dataList)
}

```

We can then use the transform function and our named vector in a call to *rxLinMod* as follows:

```

censusWorkers <- file.path(rxGetOption("sampleDataDir"), "CensusWorkers.xdf")
linModObj <- rxLinMod(incwage~sex + age + stateEducExpPC,
  data = censusWorkers, pweights = "perwt",
  transformFun = transformFunc, transformVars = "state",
  transformObjects = list(educExp = educExpense))
summary(linModObj)

```

When the transform function is evaluated, it will have access to the *educExp* object. The final results show:

166 Creating a Row Selection Variable

```
Call:
rxLinMod(formula = incwage ~ sex + age + stateEducExpPC, data = censusWorkers,
  pweights = "perwt", transformObjects = list(educExp = educExp),
  transformFunc = transformFunc, transformVars = "state")

Linear Regression Results for: incwage ~ sex + age + stateEducExpPC
File name:
  C:\Revolution\R-Enterprise-7.0\
  R-3.0.2\library\RevoScaleR\SampleData\CensusWorkers.xdf
Probability weights: perwt
Dependent variable(s): incwage
Total independent variables: 5 (Including number dropped: 1)
Number of valid observations: 351121
Number of missing observations: 0

Coefficients: (1 not defined because of singularities)
              Estimate Std. Error t value Pr(>|t|)
(Intercept)  -1.809e+04  4.414e+02  -40.99 2.22e-16 ***
sex=Male      1.689e+04  1.321e+02   127.83 2.22e-16 ***
sex=Female    Dropped      Dropped Dropped  Dropped
age           5.593e+02  5.791e+00    96.58 2.22e-16 ***
stateEducExpPC 1.643e+01  2.731e-01    60.16 2.22e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 175900 on 351117 degrees of freedom
Multiple R-squared:  0.07755
Adjusted R-squared:  0.07754
F-statistic:  9839 on 3 and 351117 DF,  p-value: < 2.2e-16
Condition number: 1.1176
```

16.3 Creating a Row Selection Variable

One common use of the `transformFunc` argument is to create a logical variable to use as a row selection variable. For example, suppose you want to create a random sample from a massive data set. You can use the `transformFunc` argument to specify a transformation that creates a random binomial variable, which can then be coerced to logical and used for row selection. The object name `.rxRowSelection` is reserved for the row selection variable; if RevoScaleR finds this object, it is used for row selection.

Warning: If you both specify a `rowSelection` argument and define a `.rxRowSelection` variable in your transform function, the one specified in your transform function will be overwritten by the contents of the `rowSelection` argument, so that expression takes precedence.

The following code creates a random selection variable to create a data frame with a random 10% subset of the census workers file:

```
createRandomSample <- function(data)
{
  data$.rxRowSelection <- as.logical(rbinom(length(data[[1]]), 1, .10))
  return(data)
}
censusWorkers <- file.path(rxGetOption("sampleDataDir"), "CensusWorkers.xdf")
```

```
df <- rxXdfToDataFrame(file = censusWorkers, transformFunc = createRandomSample,
  transformVars = "age")
```

The resulting data frame, `df`, has approximately 35,000 rows. You can look at the first few rows using `head` as follows:

```
rxGetInfo(df)
head(df)
```

	age	incwage	perwt	sex	wkswork1	state
1	50	9000	30	Male	48	Indiana
2	41	35000	20	Female	48	Indiana
3	55	40400	21	Male	52	Indiana
4	56	45000	30	Female	52	Indiana
5	46	17200	60	Female	52	Indiana
6	49	35000	21	Female	52	Indiana

Equivalently, we could create the temporary row selection variable using the `rowSelection` argument with the internal `.rxNumRows` variable, which provides the number of rows in the current chunk of data being processed:

```
df <- rxDataStep(inData = censusWorkers,
  rowSelection = as.logical(rbinom(.rxNumRows, 1, .10)) == TRUE)
```

16.4 Using Internal Variables in a Transformation Function: An Example Computing Moving Averages

Four additional variables providing information on the RevoScaleR processing are available for use in your transform functions:

- `.rxStartRow`: The row number from the original data that was read as the first row of the current chunk.
- `.rxChunkNum`: The current chunk being processed.
- `.rxReadFileName`: The name of the `.xdf` file currently being read.
- `.rxIsTestChunk`: Whether the chunk being processed is being processed as a test sample of data.

These are particularly useful if you need to access additional rows of data when processing a chunk. This is the case, for example, when you want to include lagged data in a calculation. The following example is a transformation function “generator” to compute a simple moving average. By creating this “function within a function” we are able to easily pass arguments into a transformation function. This one takes as arguments the number of days (or time units) for the moving average, the name of the variable that will be used to compute the moving average, and the name of the new variable to create. The transformation function computes the number of lagged observations to read in, then reads in that data to create a long data vector with the lags. The simple moving average calculations are performed and put into a new

168 Using Internal Variables in a Transformation Function: An Example Computing Moving Averages

variable. The additional lagged rows are removed from the original data vector before returning from the function.

```
makeMoveAveTransFunc <- function(numDays = 10, varName="", newVarName="")
{
  function (dataList)
  {
    numRowsToRead <- 0
    varForMoveAve <- 0
    # If no variable is named, use the first one sent
    # to the transformFunc
    if (is.null(varName) || nchar(varName) == 0)
    {
      varForMoveAve <- names(dataList)[1]
    } else {
      varForMoveAve <- varName
    }

    # Get the number of rows in the current chunk
    numRowsInChunk <- length(dataList[[varForMoveAve]])

    # .rxStartRow is the starting row number of the
    # chunk of data currently being processed
    # Read in previous data if we are not starting at row 1
    if (.rxStartRow > 1)
    {
      # Compute the number of lagged rows we'd like
      numRowsToRead <- numDays - 1
      # Check to see if enough data is available
      if (numRowsToRead >= .rxStartRow)
      {
        numRowsToRead <- .rxStartRow - 1
      }
      # Compute the starting row of previous data to read
      startRow <- .rxStartRow - numRowsToRead
      # Read previous rows from the .xdf file
      previousRowsDataList <- RevoScaleR::rxReadXdf(
        file=.rxReadFileName,
        varsToKeep=names(dataList),
        startRow=startRow, numRows=numRowsToRead,
        returnDataFrame=FALSE)
      # Concatenate the previous rows with the existing rows
      dataList[[varForMoveAve]] <-
        c(previousRowsDataList[[varForMoveAve]],
          dataList[[varForMoveAve]])
    }
    # Create variable for simple moving average
    # It will be added as the last variable in the data list
    newVarIdx <- length(dataList) + 1

    # Initialize with NA's
    dataList[[newVarIdx]] <- rep(as.numeric(NA), times=numRowsInChunk)

    for (i in (numRowsToRead+1):(numRowsInChunk + numRowsToRead))
    {
      j <- i - numRowsToRead
      lowIdx <- i - numDays + 1
      if (lowIdx > 0 && ((lowIdx == 1) ||
        (j > 1 && (is.na(dataList[[newVarIdx]][j-1])))))
      {
        # If it's the first computation or the previous value
      }
    }
  }
}
```



```

        # is missing, take the mean of all the relevant lagged data
        dataList[[newVarIdx]][j] <-
            mean(dataList[[varForMoveAve]][lowIdx:i])
    } else if (lowIdx > 1)
    {
        # Add and subtract from the last computation
        dataList[[newVarIdx]][j] <- dataList[[newVarIdx]][j-1] -
            dataList[[varForMoveAve]][lowIdx-1]/numDays +
            dataList[[varForMoveAve]][i]/numDays
    }
}
# Remove the extra rows we read in from the original variable
dataList[[varForMoveAve]] <-

    dataList[[varForMoveAve]][(numRowsToRead + 1):(numRowsToRead +
        numRowsInChunk)]

# Name the new variable
if (is.null(newVarName) || (nchar(newVarName) == 0))
{
    # Use a default name if no name specified
    names(dataList)[newVarIdx] <-
        paste(varForMoveAve, "SMA", numDays, sep=".")
} else {
    names(dataList)[newVarIdx] <- newVarName
}
return(dataList)
}
}

```

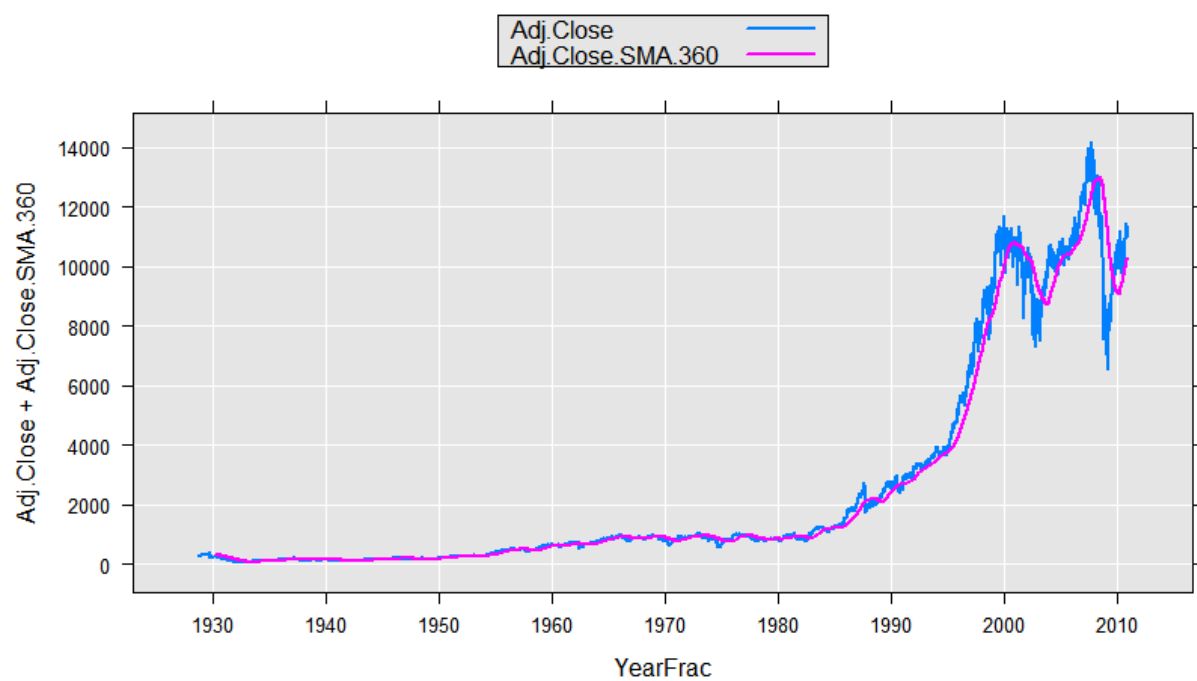
We could use this function with `rxDataStep` to add a variable to our data set, or on-the-fly, for example for plotting. Here we will use the sample data set containing daily information on the Dow Jones Industrial Average. We compute a 360-day moving average for the adjusted close (adjusted for dividends and stock splits) and plot it:

```

DJIAdaily <- file.path(rxGetOption("sampleDataDir"), "DJIAdaily.xdf")
rxLinePlot(Adj.Close+Adj.Close.SMA.360~YearFrac, data=DJIAdaily,
    transformFunc=makeMoveAveTransFunc(numDays=360),
    transformVars=c("Adj.Close"))

```

170 Using Internal Variables in a Transformation Function: An Example Computing Moving Averages



16.5 How Transforms Are Evaluated

User-defined transforms and transform functions are evaluated in essentially the same way. An evaluation environment is constructed from the base environment together with the `utils`, `stats`, and `methods` packages (you can modify this basic list by setting the `rxOption` `transformPackages`), packages specified with the `transformPackages` argument, and any specified `transformObjects`, and the closure of the transform function. (User-defined transforms are combined into a single, “hidden” transform function for evaluation.) The transform function is then evaluated in this evaluation environment. Functions that are in packages that are not part of the evaluation environment can be used, but must be prefixed by their package name and two or three colons, depending on whether the function is exported.

16.6 Transformations to Avoid

Transform functions are very powerful, and we recommend their use. However, there are four types of transformation that should be avoided:

- 1) transformations that change the length of a variable (this includes, naturally, most model-fitting functions)
- 2) transformations that depend upon all observations simultaneously (because `RevoScaleR` works on chunks of data, such transformations will not have access to all the data at once). Examples of such transformations are matrix operations such as `poly` or `solve`.
- 3) transformations that have the possibility of creating different mappings of factor codes to factor labels.
- 4) transformations that involve sampling with replacement. Again, this is because `RevoScaleR` works on chunks of data, and sampling with replacement chunk by chunk is not equivalent to sampling with replacement from the full data set.

If you change the length of one variable, you will get errors from subsequent analysis functions that not all variables are the same length. If you try to change the length of all variables (essentially, performing some sort of row selection), you need to pass all of your data through the transform function, and this can be very inefficient. We therefore recommend the row selection procedures described in previous chapters.

If you create a factor within a transformation function, you may get unexpected results because all of the data is not in memory at one time. It is recommended that if creating a factor within a transformation function, you always explicitly set the values and labels. For example,

```
dataList$xfac <- as.factor(dataList$x, levels = c(1, 2, 3),
  labels = c("One", "Two", "Three"))
```

Chapter 17.

Visualizing Huge Data Sets: An Example from the U.S. Census

By combining the power and flexibility of the open-source R language with the fast computations and rapid access to huge datasets provided by RevoScaleR, it is easy and efficient to not only do fine-grained calculations “on the fly” for plotting, but to visually drill down into these patterns.

This example focuses on a basic demographic pattern: in general, more boys than girls are born and the death rate is higher for males at every age. So, typically we observe a decline in the ratio of males to females as age increases.

17.1 Examining the Data

We can examine this pattern in the United States using the 5% Public Use Microdata Sample (PUMS) of the 2000 United States Census, stored in an .xdf file of about 12 gigabytes.¹ Using the `rxGetInfo` function, we can get a quick summary of the data set:

```
bigDataDir <- "C:/Revolution/Data"
bigCensusData <- file.path(bigDataDir, "CensusUS5Pct2000.xdf")
rxGetInfo(bigCensusData)

File name: C:\Revolution\Data\CensusUS5Pct2000.xdf
Number of observations: 14058983
Number of variables: 264
```

¹ The analysis in this paper uses the Integrated Public Use Microdata Sample provided by the Minnesota Population Center (Ruggles, et al., 2006) stored in Revolution Analytics' .xdf file format.

```
Number of blocks: 98
Compression type: zlib
```

It contains over 14 million rows, has 264 variables, and has been stored in 98 data blocks in the .xdf file.

First let's use the `rxCube` function to count the number of males and females for each age, using the weighting variable provided by the census bureau. We'll read in the data in chunks of 15 blocks, or about 2 million rows at a time.

```
ageSex <- rxCube(~F(age):sex, pweights = "perwt", data = bigCensusData,
  blocksPerRead = 15)
```

In the computation, we're treating age as a categorical variable so we'll get counts for each age. Since we'll be converting this factor data back to integers on a regular basis, we'll write a function to do the conversion:

```
factoi <- function(x)
{
  as.integer(levels(x))[x]
}
```

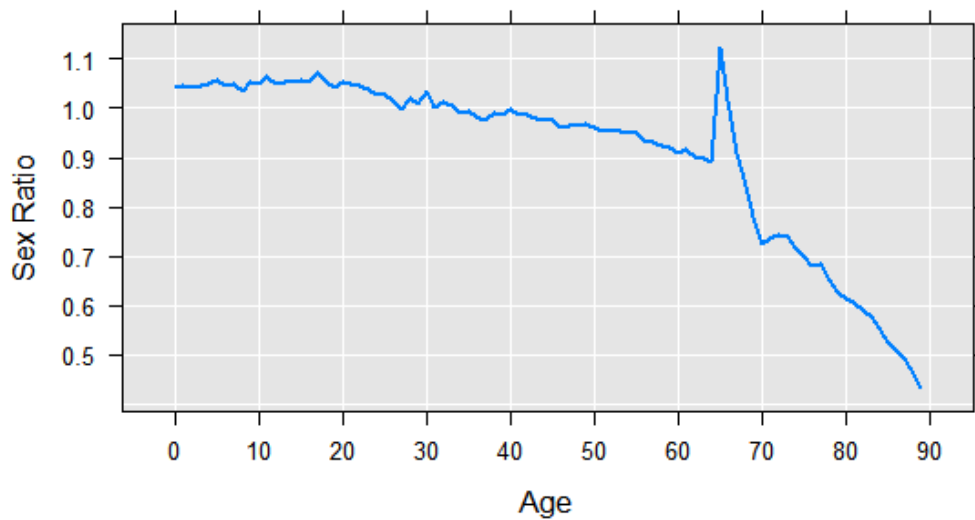
We can now write a simple function in R to take the counts information returned from `rxCube` and do the arithmetic to compute the sex ratio for each age.

```
getSexRatio <- function(ageSex)
{
  ageSexDF <- as.data.frame(ageSex)
  sexRatioDF <- subset(ageSexDF, sex == 'Male')
  names(sexRatioDF)[names(sexRatioDF) == 'Counts'] <- 'Males'
  sexRatioDF$sex <- NULL
  females <- subset(ageSexDF, sex == 'Female')
  sexRatioDF$Females <- females$Counts
  sexRatioDF$age <- factoi(sexRatioDF$F_age)
  sexRatioDF <- subset(sexRatioDF, Females > 0 & Males > 0 & age <= 90)
  sexRatioDF$SexRatio <- sexRatioDF$Males/sexRatioDF$Females
  return(sexRatioDF)
}
```

It returns a data frame with the counts for Males and Females, the SexRatio, and age for all groups in which there are positive counts for both Males and Females. Ages over 90 are also excluded.

Let's use that function and then plot the results:

```
sexRatioDF <- getSexRatio(ageSex)
rxLinePlot(SexRatio~age, data = sexRatioDF,
  xlab = "Age", ylab = "Sex Ratio",
  main = "Figure 1: Sex Ratio by Age, U.S. 2000 5% Census")
```

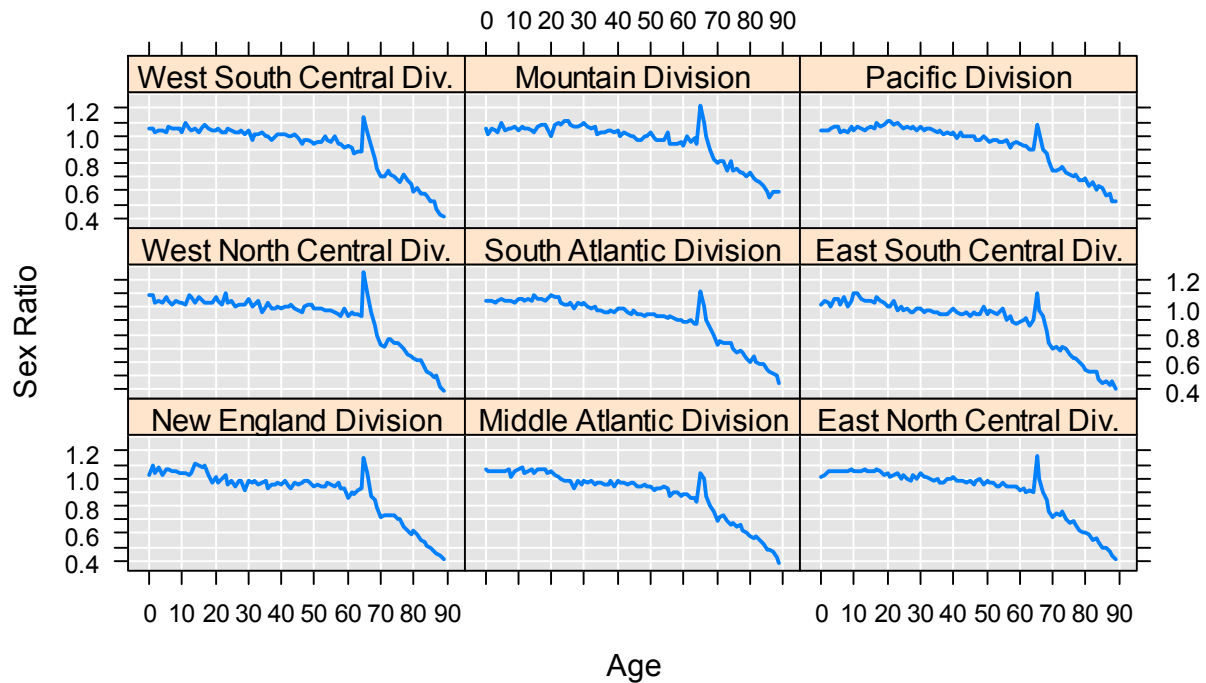
Figure 1: Sex Ratio by Age, U.S. 2000 5% Census

The graph shows the expected downward trend at the younger ages. But look at what happens at the age of 65! At the age of 65, there are suddenly about 12 men for every 10 women.

We can quickly drill down, and do the same computation for each region:

```
ageSex <- rxCube(~F(age):sex:region, pweights = "perwt", data = bigCensusData,
  blocksPerRead = 15)
sexRatioDF <- getSexRatio(ageSex)
rxLinePlot(SexRatio~age|region, data = sexRatioDF,
  xlab = "Age", ylab = "Sex Ratio",
  main = "Figure 2: Sex Ratio by Age and Region, U.S. 2000 5% Census")
```

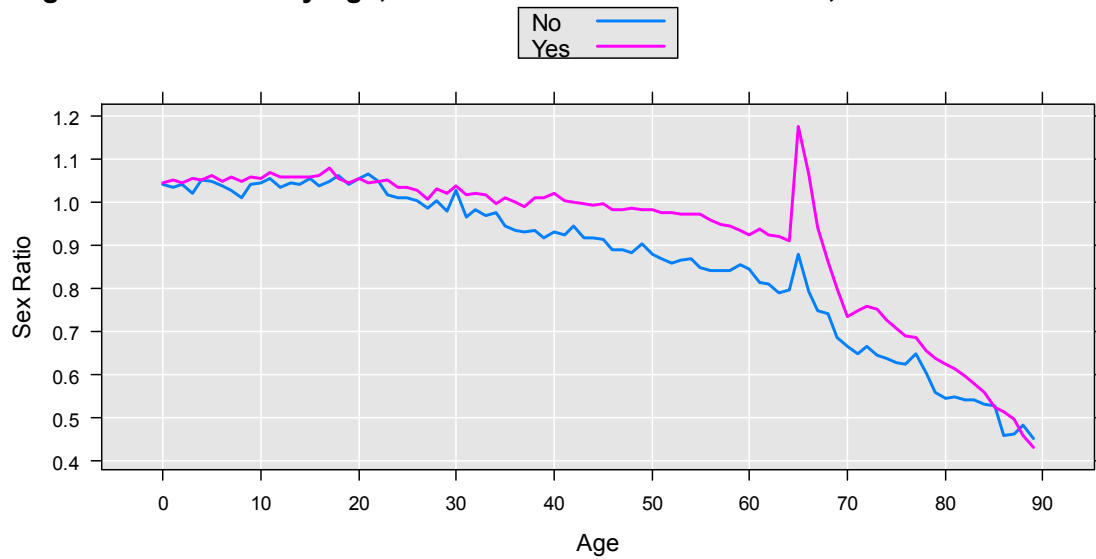
We see the unlikely “spike” at age 65 in all regions:

Figure 2: Sex Ratio by Age and Region, U.S. 2000 5% Census

Let's try looking at race, comparing whites with non-whites.

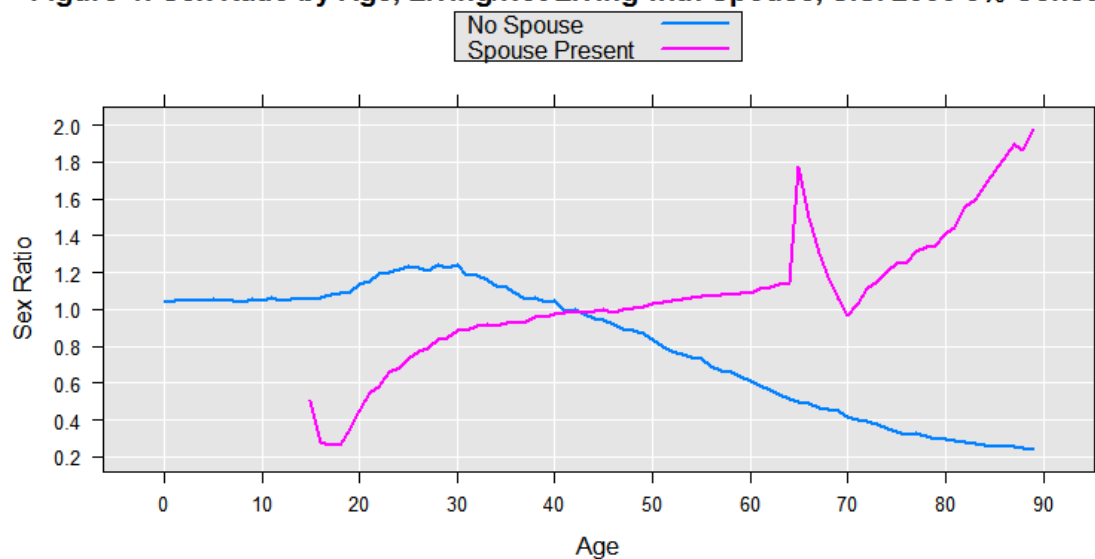
```
ageSex <- rxCube(~F(age):sex:racwht, pweights = "perwt", data = bigCensusData,
  blocksPerRead = 15)
sexRatioDF <- getSexRatio(ageSex)
rxLinePlot(SexRatio~age, groups = racwht, data = sexRatioDF,
  xlab = "Age", ylab = "Sex Ratio",
  main = "Figure 3: Sex Ratio by Age, Conditioned on 'Is Race White?', U.S.
  2000 5% Census")
```

There are interesting differences between the two groups, but again there is the familiar spike at age 65 in both cases.

Figure 3: Sex Ratio by Age, Conditioned on 'Is Race White?', U.S. 2000 5% Census

How about comparing married people with those not living with a spouse? We can create a temporary variable using the transform argument to do this computation:

```
ageSex <- rxCube(~F(age):sex:married, pweights = "perwt", data = bigCensusData,
  transforms = list(married = factor(marst == 'Married, spouse present',
    levels = c(FALSE, TRUE), labels = c("No Spouse", "Spouse Present"))),
  blocksPerRead = 15)
sexRatioDF <- getSexRatio(ageSex)
rxLinePlot(SexRatio~age, groups = married, data = sexRatioDF,
  xlab="Age", ylab = "Sex Ratio",
  main="Figure 4: Sex Ratio by Age, Living/Not Living with Spouse, U.S.
  2000 5% Census")
```

Figure 4: Sex Ratio by Age, Living/Not Living with Spouse, U.S. 2000 5% Census

First, notice that the spike at age 65 is absent for unmarried people. But also look at the very different trends. For married 20 year-olds, there are about 5 men for every 10 women, but for married 60 year-olds, there would be about 11 men for every 10 women. This may at first seem counter-intuitive, but it's consistent with the notion that men tend to marry younger women. Let's explore that next.

17.2 Extending the Analysis

We'd like to compare the ages of men with ages of their wives. This is more complicated than the earlier computations because the spouse's age is stored in a different record in the data file. To handle this, we'll create a new data set using RevoScaleR's data step functionality with a transformation function. This transformation function makes use of RevoScaleR's ability to go back and read additional rows from an .xdf file as it is reading through it in chunks. It also uses internal variables that are provided inside a transformation function: `.rxStartRow`, which is the row number from the original data set for the first row of the chunk of data being processed; `.rxReadFileName` gives the name of the file being read. It first checks the spouse location for the relative position of the spouse in the data set. It then determines which of the observations have spouses in the previous, current, or next chunk of data. Then, in each of the three cases, it looks up the spouse information and adds it to the original observation.

```
spouseAgeTransform <- function(data)
{
  # Use internal variables
  censusUS2000 <- .rxReadFileName
  startRow <- .rxStartRow

  # Calculate basic information about input data chunk
  numRows <- length(data$sploc)
  endRow <- startRow + numRows - 1

  # Create a new variable. A spouse is present if the spouse locator
  # (relative position of spouse in data) is positive
  data$hasSpouse <- data$sploc > 0

  # Create variables for spouse information
  spouseVars <- c("age", "incwage", "sex")
  data$spouseAge <- rep.int(NA_integer_, numRows)
  data$spouseIncwage <- rep.int(NA_integer_, numRows)
  data$sameSex <- rep.int(NA, numRows)

  # Create temporary row numbers for this block
  rowNum <- seq_len(numRows)
  # Find the temporary row number for the spouse
  spouseRow <- rep.int(NA_integer_, numRows)
  if (any(data$hasSpouse))
  {
    spouseRow[data$hasSpouse] <-
      rowNum[data$hasSpouse] +
      data$sploc[data$hasSpouse] - data$pernum[data$hasSpouse]
  }

  #####
```

```

# Handle possibility that spouse is in previous or next chunk
# Create a variable indicating if the spouse is in the previous,
# current, or next chunk
blockBreaks <- c(-.Machine$integer.max, 0, numRows, .Machine$integer.max)
blockLabels <- c("previous", "current", "next")
spouseFlag <- cut(spouseRow, breaks = blockBreaks, labels = blockLabels)
blockCounts <- tabulate(spouseFlag, nbins = 3)
names(blockCounts) <- blockLabels

# At least one spouse in previous chunk
if (blockCounts[["previous"]] > 0)
{
  # Go back to the original data set and read the
  # required rows in the previous chunk
  needPreviousRows <- 1 - min(spouseRow, na.rm = TRUE)
  previousData <- rxReadXdf(censusUS2000,
    startRow = startRow - needPreviousRows,
    numRows = needPreviousRows, varsToKeep = spouseVars,
    returnDataFrame = FALSE, reportProgress = 0)

  # Get the spouse locations
  whichPrevious <- which(spouseFlag == "previous")
  spouseRowPrev <- spouseRow[whichPrevious] + needPreviousRows

  # Set the spouse information for everyone with a spouse
  # in the previous chunk
  data$spouseAge[whichPrevious] <- previousData$age[spouseRowPrev]
  data$spouseIncwage[whichPrevious] <- previousData$incwage[spouseRowPrev]
  data$sameSex[whichPrevious] <-
    data$sex[whichPrevious] == previousData$sex[spouseRowPrev]
}

# At least one spouse in current chunk
if (blockCounts[["current"]] > 0)
{
  # Get the spouse locations
  whichCurrent <- which(spouseFlag == "current")
  spouseRowCurr <- spouseRow[whichCurrent]

  # Set the spouse information for everyone with a spouse
  # in the current chunk
  data$spouseAge[whichCurrent] <- data$age[spouseRowCurr]
  data$spouseIncwage[whichCurrent] <- data$incwage[spouseRowCurr]
  data$sameSex[whichCurrent] <-
    data$sex[whichCurrent] == data$sex[spouseRowCurr]
}

# At least one spouse in next chunk
if (blockCounts[["next"]] > 0)
{
  # Go back to the original data set and read the
  # required rows in the next chunk
  needNextRows <- max(spouseRow, na.rm=TRUE) - numRows
  nextData <- rxReadXdf(censusUS2000, startRow = endRow+1,
    numRows = needNextRows, varsToKeep = spouseVars,
    returnDataFrame = FALSE, reportProgress = 0)

  # Get the spouse locations
  whichNext <- which(spouseFlag == "next")
  spouseRowNext <- spouseRow[whichNext] - numRows

  # Set the spouse information for everyone with a spouse
  # in the next block

```

```

    data$spouseAge[whichNext] <- nextData$age[spouseRowNext]
    data$spouseIncwage[whichNext] <- nextData$incwage[spouseRowNext]
    data$sameSex[whichNext] <-
      data$sex[whichNext] == nextData$sex[spouseRowNext]
  }

  # Now caculate age difference
  data$ageDiff <- data$age - data$spouseAge
  data
}

```

We can test the transform function by reading in a small number of rows of data. First we will read in a chunk that has a spouse in the previous block and a spouse in the next block, and call the transform function. We will repeat this, expanding the chunk of data to include both spouses, and double check to make sure the results are the same for the equivalent rows:

```

varsToKeep=c("age", "region", "incwage", "racwht", "nchild", "perwt", "sploc",
             "pernum", "sex")
testDF <- rxReadXdf(bigCensusData, numRows = 6, startRow=9,
                   varsToKeep = varsToKeep, returnDataFrame=FALSE)
.rxStartRow <- 9
.rxReadFileName <- bigCensusData
newTestDF <- as.data.frame(spouseAgeTransform(testDF))
.rxStartRow <- 8
testDF2 <- rxReadXdf(bigCensusData, numRows = 8, startRow=8,
                   varsToKeep = varsToKeep, returnDataFrame=FALSE)
newTestDF2 <- as.data.frame(spouseAgeTransform(testDF2))
newTestDF[,c("age", "incwage", "sploc", "hasSpouse", "spouseAge", "ageDiff")]
newTestDF2[,c("age", "incwage", "sploc", "hasSpouse", "spouseAge", "ageDiff")]

> newTestDF[,c("age", "incwage", "sploc", "hasSpouse", "spouseAge", "ageDiff")]
  age incwage sploc hasSpouse spouseAge ageDiff
1  46   1000     1      TRUE        43        3
2  16        0     0     FALSE        NA       NA
3  14        NA     0     FALSE        NA       NA
4   7        NA     0     FALSE        NA       NA
5  19   1500     0     FALSE        NA       NA
6  62  42600     2      TRUE        55        7

> newTestDF2[,c("age", "incwage", "sploc", "hasSpouse", "spouseAge",
               "ageDiff")]
  age incwage sploc hasSpouse spouseAge ageDiff
1  43 150000     2      TRUE        46       -3
2  46   1000     1      TRUE        43        3
3  16        0     0     FALSE        NA       NA
4  14        NA     0     FALSE        NA       NA
5   7        NA     0     FALSE        NA       NA
6  19   1500     0     FALSE        NA       NA
7  62  42600     2      TRUE        55        7
8  55        0     1      TRUE        62       -7

```

To create the new data set, we'll use the transformation function with `rxDataStep`. Observations for males living with female spouses will be written to a new data .xdf file named `spouseCensus2000.xdf`. It will include information about the age of their spouse.

```

spouseCensusXdf <- "spouseCensus2000"
rxDataStep(inData = bigCensusData, outFile=spouseCensusXdf,

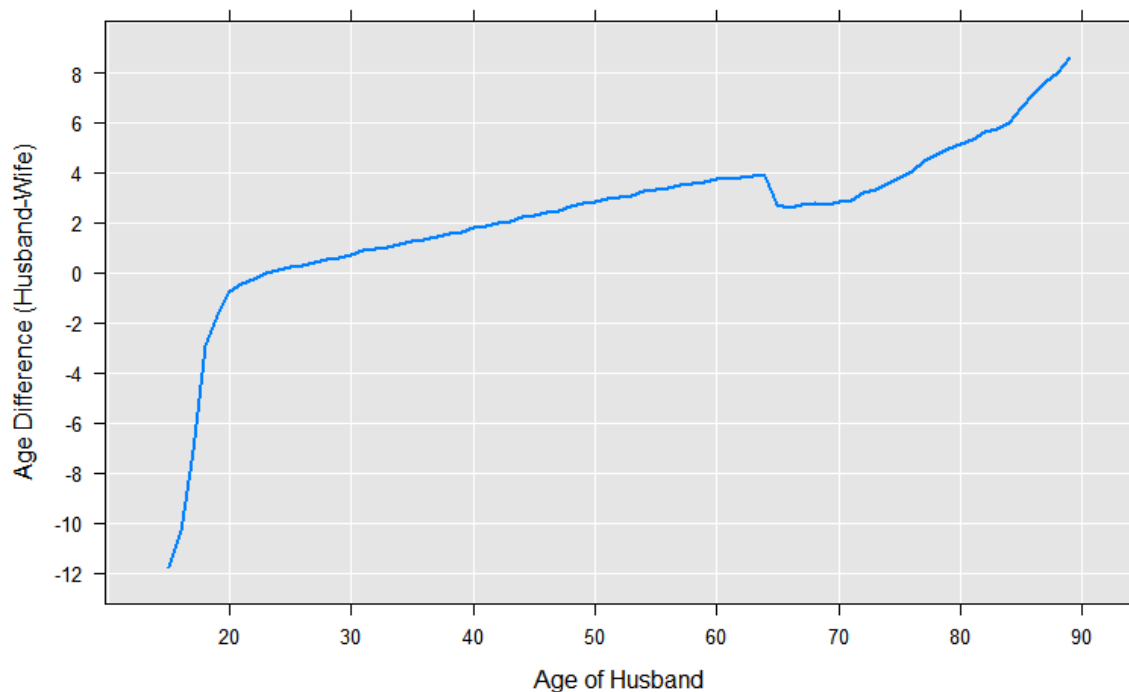
```

```
varsToKeep=c("age", "region", "incwage", "racwht", "nchild", "perwt"),
transformFunc = spouseAgeTransform,
transformVars = c("age", "incwage", "sploc", "pernum", "sex"),
rowSelection = sex == 'Male' & hasSpouse == 1 & sameSex == FALSE &
  age <= 90,
blocksPerRead = 15, overwrite=TRUE)
```

Now for each husband age we can compute the distribution of spouse age. Then, after converting age back to an integer, we can plot the age difference by age of husband:

```
ageDiffData <- rxCube(ageDiff~F(age) , pweights="perwt", data = spouseCensusXdf,
  returnDataFrame = TRUE, blocksPerRead = 15)
ageDiffData$ownAge <- factoi(ageDiffData$F_age)
rxLinePlot(ageDiff~ownAge, data = ageDiffData,
  xlab="Age of Husband", ylab = "Age Difference (Husband-Wife)",
  main="Figure 5: Age Difference of Spouses Living Together, U.S. 2000 5%
  Census")
```

Figure 5: Age Difference of Spouses Living Together, U.S. 2000 5% Census



Beginning at ages in the early 20's, men tend to be married to younger women. The age difference increases as men get older. But beginning at age 65, our smooth trend stops and we see more erratic behavior, suggesting that our data has misinformation about ages of spouses within households at age 65 and above.

With our new data set we can also calculate the counts for each combination of husband's age and wife's age:

```
aa <- rxCube(~F(age):F(spouseAge), pweights = "perwt", data = spouseCensusXdf,
```

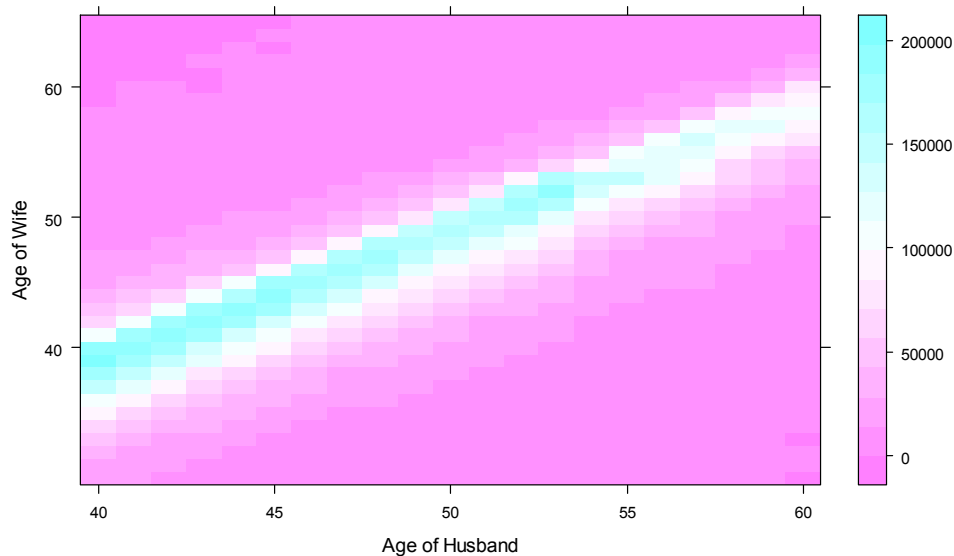
```
returnDataFrame = TRUE, blocksPerRead = 7)
```

A level plot is a good way to visualize the results, where the color indicates the count of each category of combination of husband's and wife's age.

```
# Convert factors to integers
aa$age <- factoi(aa$F_age)
aa$spouseAge <- factoi(aa$F_spouseAge)

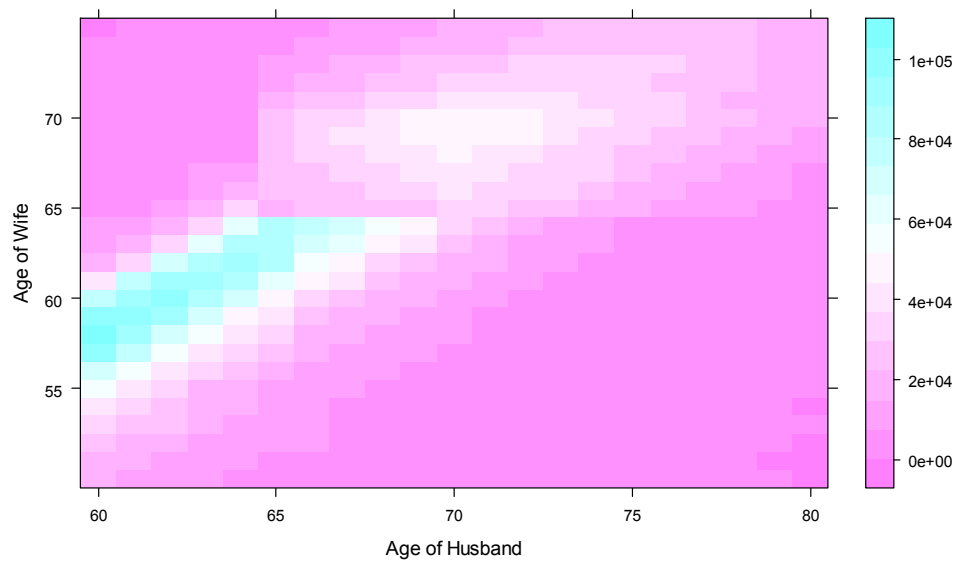
# Do a level plot showing the counts for husbands aged 40 to 60
ageCompareSubset <- subset(aa, age >= 40 & age <= 60 & spouseAge >= 30 &
  spouseAge <= 65)
levelplot(Counts~age*spouseAge, data=ageCompareSubset,
  xlab="Age of Husband", ylab = "Age of Wife",
  main="Figure 6: Counts by Age (40-60) and Spouse Age, U.S. 2000 5% Census")
```

Figure 6: Counts by Age (40-60) and Spouse Age, U.S. 2000 5% Census



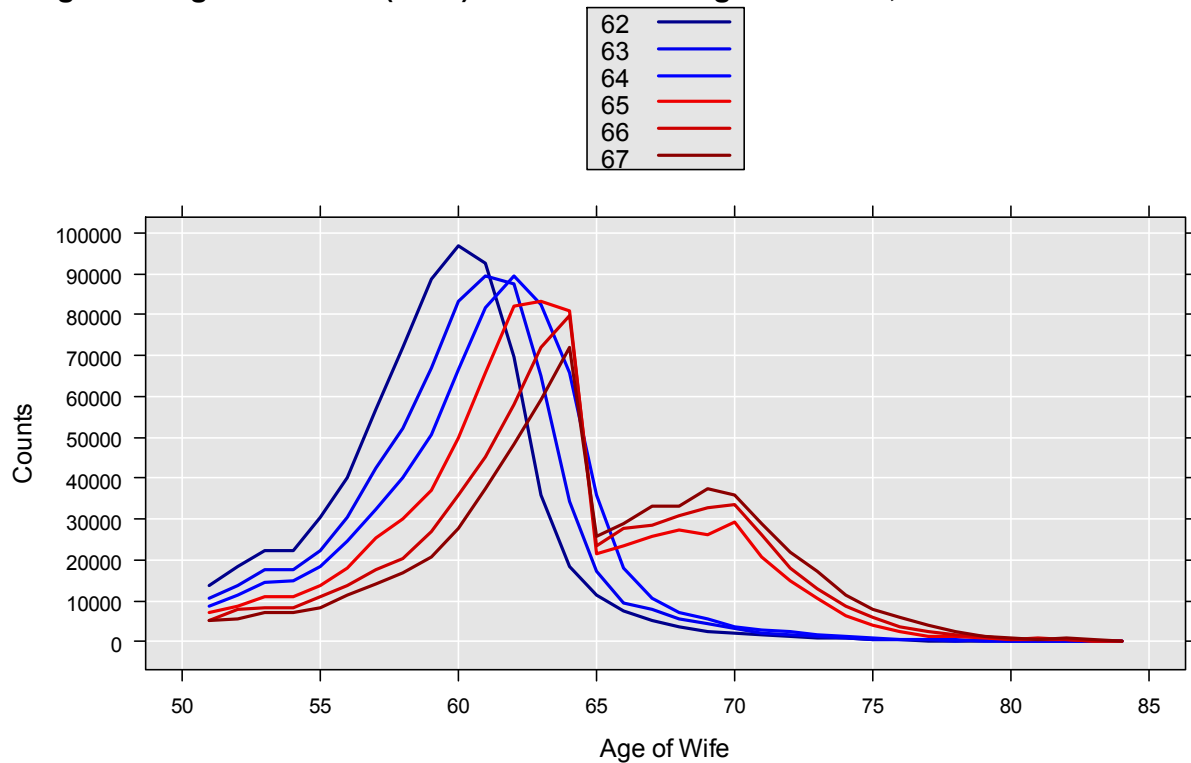
In the level plot, there is a very clear pattern with the mode of the relative age of wife dropping gradually as the age of husband increases. Now, repeat with husbands aged 60 to 80.

```
ageCompareSubset <- subset(aa, age >= 60 & age <= 80 & spouseAge >= 50 &
  spouseAge <= 75)
levelplot(Counts~age*spouseAge, data = ageCompareSubset,
  xlab = "Age of Husband", ylab = "Age of Wife",
  main = "Figure 7: Counts by Age(60-80)and Spouse Age , U.S. 2000 5% Census")
```

Figure 7: Counts by Age(60-80)and Spouse Age , U.S. 2000 5% Census

This shows a different story. Notice that there are very few men in the 60 to 80 age range married to 65 year-old women, and in particular, there are very few 65-year-old men married to 65-year-old women. To examine this further, we can look at line plots of the distributions of wife's ages for men ages 62 to 67:

```
ageCompareSubset <- subset(aa, age > 61 & age < 68 & spouseAge > 50 &
  spouseAge < 85)
rxLinePlot(Counts~spouseAge, groups = age, data = ageCompareSubset,
  xlab = "Age of Wife", ylab = "Counts",
  lineColor = c("Blue4", "Blue2", "Blue1", "Red2", "Red3", "Red4"),
  main = "Figure 8: Ages of Wives (> 45) for Husband's Ages 62 to 67, U.S.
  2000 5% Census")
```

Figure 8: Ages of Wives (> 45) for Husband's Ages 62 to 67, U.S. 2000 5% Census

The blue lines show husbands ages 62 through 64. The mode of the wife's age is a couple of years younger than the husband, and we have a reasonable looking distribution in both tails. But starting at age 65, with the red lines, we have a very different pattern. It appears that when husbands reach the age of 65 they begin to leave (or lose) their 65-year-old wives in droves – and marry older women. This certainly makes one suspicious of the data. In fact, it turns out the aberration in the data was introduced by disclosure avoidance techniques applied to the data by the census bureau.²

². For information on the inconsistent ratios, see (U.S. Bureau of the Census, April 2009) and (Alexander, Davern, & Stevenson, 2009).

Bibliography

Alexander, J. T., Davern, M., & Stevenson, B. (2009). *Inaccurate Age and Sex Data in the Census PUMS Files: Evidence and Implications*.

<http://bpp.wharton.upenn.edu/betseys/papers/Inaccurate%20Age%20and%20Sex%20Data%20in%20Census%20PUMS%20Files.pdf>.

Ben-Haim, Y., & Tom-Tov, E. (2010). A streaming parallel decision tree algorithm. *Journal of Machine Learning Research*, 849-872.

Chambers, J. M., & Hastie, T. J. (1992). *Statistical Models in S*. Pacific Grove, CA: Wadsworth & Brooks/Cole.

Fox, J. (2002). *An R and S-PLUS Companion to Applied Regression*. Los Angeles: Sage Publications.

Fox, J. (2008). *Applied Regression Analysis and Generalized Linear Models, Second Edition*. Los Angeles: Sage Publications.

Frank, A., & Asuncion, A. (2010). (University of California, Irvine, School of Information and Computer Science) Retrieved August 2012, from UCI Machine Learning Repository: <http://archive.ics.uci.edu/ml>

Kabacoff, R. I. (2011). *R In Action*. Shelter Island, NY: Manning.

Kohavi, R. (1996). Scaling Up the Accuracy of Naive-Bayes Classifiers: a Decision-Tree Hybrid. *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*.

- Lloyd, S. P. (1982). Least squares quantization in PCM. *IEEE Transactions on Information Theory*(28), 128-137.
- Moore, D. S., & McCabe, G. P. (2006). *Introduction to the Practice of Statistics* (Fifth ed.). New York: Freeman.
- Ruggles, S., Sobek, M., Alexander, T., Fitch, C. A., Goeken, R., Hall, P. K., et al. (2006). *Integrated Public Use Microdata Series: Version 3.0. [Machine-readable database]*. Available at: <http://usa.ipums.org> . Minneapolis, MN:: Minnesota Population Center [producer and distributor].
- Therneau, T., & Atkinson, E. (1997). *An Introduction to Recursive Partitioning Using the RPART Routines*. Rochester, MN: Mayo Clinic.
- U.S. Bureau of the Census. (April 2009). *How to Use the Data: Errata, User Note 47*. Available at: <http://www.census.gov/acs/www/UseData/Errata.htm>.
- Venables, W. N., & Ripley, B. R. (2002). *Modern Applied Statistics with S, Fourth Edition*. New York: Springer.
- Wilkinson, G. N., & Rogers, C. E. (1973). Symbolic Description of Factorial Models for Analysis of Variance. *Applied Statistics*, 22, 392-399.