R Services 2016
Getting Started Guide

The correct bibliographic citation for this manual is as follows: Microsoft Corporation. 2016. *Microsoft R Services Getting Started Guide*. Microsoft Corporation, Redmond, WA.

Microsoft R Services Getting Started Guide

Copyright © 2016 Microsoft Corporation. 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 Microsoft Corporation.

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, DeployR, RevoPemaR, RevoTreeView, and Revolution Analytics are trademarks of Microsoft Corporation.

Revolution R Enterprise/Microsoft R Server includes the Intel® Math Kernel Library (https://software.intel.com/en-us/intel-mkl). RevoScaleR includes Stat/Transfer software under license from Circle Systems, Inc. Stat/Transfer is a trademark of Circle Systems, Inc.

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

Microsoft
One Microsoft Way
Redmond WA 98052
U.S.A.

Revised on November 18, 2015

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 revodoc@microsoft.com.
We'd love to hear from you.

Contents

Chapter 1. What Is Microsoft R Services? 1

Chapter 2. 1

2.1 R for the Enterprise 1

2.2 Microsoft R Open 2

2.3 DistributedR 2

2.4 ScaleR 3

2.5 ConnectR 3

2.6 DevelopR 4

2.7 DeployR 4

Chapter 3. Microsoft R Services Basics 5

3.1 Starting Microsoft R Services 5

3.1.1 Starting Microsoft R Services on Windows 5

3.1.2 Starting Microsoft R Services on Linux 6

3.2 Stopping Microsoft R Services 7

3.3 Getting Help 7

Chapter 4. An R Tutorial in 25 Functions or So 9

4.1 Creating Vectors 9

4.2 Exploratory Data Analysis 12

4.3 Summary Statistics 17

4.4 Creating Multivariate Data Sets 17

4.5 Linear Models 18

4.6 Matrices and `apply` 21

4.7 Lists and *lapply* 23

4.8 Packages 24

4.9 Using Microsoft R Services via Rscript and R CMD BATCH 25

Chapter 5. Tips on Computing with Big Data in R 27

5.1 Upgrade Your Hardware 28

5.2 Upgrade Your Software 28

5.3 Minimize Copies of Data 28
5.4 Process Data in Chunks 29
5.5 Compute in Parallel Across Cores or Nodes 30
5.6 Take Advantage of Integers 31
5.7 Store Your Data Efficiently 32
5.8 Only Read in The Data That Is Needed 32
5.9 Avoid Loops when Transforming Data 33
5.10 Use C, C++, or FORTRAN for Critical Functions 33
5.11 Process Data Transformations in Batches 34
5.12 User Row-Oriented Data Transformations where Possible 34
5.13 Handle Categorical Variables Efficiently and with Care. 34
5.14 Be Aware of Output with the Same Number of Rows as Your Data. 34
5.15 Think Twice Before Sorting. 35
Chapter 6. Getting Started with Big Data in R 36
6.1 Step 1: Accessing Your Data with *rxImport* 36
6.2 Step 2: A Quick Look at the Data 37
6.3 Step 3: Data Selection and Transformations with *rxDataStep* 38
6.4 Step 4: Visualizing Your Data with *rxHistogram*, *rxCube*, and *rxLinePlot* 39
6.5 Step 5: Analyzing Your Data with *rxLogit* 40
6.6 Step 6: Scaling Your Analysis 41
Chapter 7. Next Steps: A Roadmap to Documentation 44
7.1 Introductory Material 46
7.2 Information on Data Analysis and Statistics 47
7.3 Information on Programming with R 48
7.4 Information on Getting Data Into and Out of R 48
7.5 Information on Creating Graphics with R 49
7.6 Information on Parallel Programming in R 49
Chapter 8. Optimized Math Libraries 50
8.1 A Note on R Numerics 51

8.2 Performance Optimization and Numerics 52
Chapter 9. R Memory Limits in Windows 54
9.1 64-bit R Memory Limits 54
9.2 Using the Memory Tools Together 55
Bibliography 58

Chapter 1.

What Is Microsoft R Services?

This chapter provides an overview of the features and components of Microsoft R Services.

2.1 R for the Enterprise

Microsoft R Services, simply put, is R for the Enterprise. Microsoft provides the software, services, and support that combine to make the very popular R statistical computing environment a compelling tool not only for academia, exploration, and prototyping, but for deployment within an enterprise. The feature set provided by the ***Microsoft R Services*** software can be categorized as follows:

- *Microsoft R Open*: High performance math libraries installed on top of a stable version of Open Source R (including Base and Recommended Packages)
- *DistributedR*: Parallel and distributed computing framework for 'Big Data Big Analytics'
- *ScaleR*: High performance, scalable, parallelized and distributable 'Big Data Big Analytics' in R
- *ConnectR*: Data connections for the 'Big Data Big Analytics'
- *DevelopR*: An integrated development environment (IDE) for R on Windows
- *DeployR*: A web services software development kit for integrating R with third party products (including business intelligence, data visualization, rules engines, etc.)

We briefly discuss each of these feature areas below, with a guide to where to find the corresponding component in ***Microsoft R Services***.

2.2 Microsoft R Open

If you are reading this document, you probably already know that R is the preferred statistical language for experts in a multitude of specialized disciplines—and that those experts frequently provide tools incorporating their expertise in the form of R packages. ***Microsoft R Services*** connects to a version of ***Microsoft R Open*** that delivers Open Source R. This means that any of the amazing third-party packages that are available for that version of Open Source R should also work when you are in ***Microsoft R Services***. And, of course, if you are one of those experts, you can create R packages using ***Microsoft R Services***.

Microsoft R Open leverages high-performance, multi-threaded math libraries to deliver performance boosts. This means that functions in R that use, for example, matrix multiplication, will run faster out of the box.

2.3 DistributedR

One of the limitations of R frequently encountered is scalability. R has many tools and techniques for handling small problems, but when the data set to be analyzed starts to get big, speed and memory limitations can be a problem. The ***Microsoft R Services*** 'Big Data Big Analytics' platform is built upon a high-performance, scalable computing framework that eradicates these technology barriers.

This 'Big Data Big Analytics' compute engine works behind-the-scenes to process computations in parallel and, if available, distribute them across nodes of a distributed compute environment such as clusters or Massively Parallel Processing (MPP) databases. While you don't interact

directly with *DistributedR*, it is the framework that allows ***Microsoft R Services*** to break through the technology barriers in R to deliver blindingly fast results on enterprise compute platforms.

DistributedR allows you to run the same R script on multiple platforms; you can create a model in one environment such as a workstation and then deploy it on a different environment such as an on-site Microsoft SQL Server, a Teradata platform, or a Hadoop cluster in the cloud. You just need to specify the information about where these computations should be performed and what data should be analyzed.

This 'Big Data Big Analytics' compute engine is the core of the RevoScaleR package, included in your distribution of ***Microsoft R Services***. For information on supported computing environments, look for the 'compute contexts' in the RevoScaleR package.

2.4 ScaleR

The 'Big Data Big Analytics' functions built on *DistributedR* provide high performance, parallelized, and distributable analytics functions that scale from small data sets in memory to huge data sets stored on disk on a cluster of computers. The analytics functions provided include summary statistics, cubes and crosstabs, linear models, logistic regression, generalized linear models, kmeans clustering, decision trees, and decision forests. These algorithms are parallelized and distributed automatically, and process data in chunks so that all of your data does not need to be in memory at one time; you can use the same analysis code for your giant data set as you do for a small data set in memory.

RevoScaleR also provides traditional 'high performance computing' (HPC) tools if you prefer to construct your own distributed computations. In addition, in many environments, there are full-featured tools for data cleaning and manipulation.

R is a flexible and powerful statistical programming language. The RevoScaleR package provides efficient, scalable computational power. Combining the two allows for the development of ready-to-deploy suites of data processing and analytics with R.

To learn more, look for the RevoScaleR 'rx' analysis and data manipulation functions and 'rxExec' for HPC functionality. If you are computing decision trees, also check out the included RevoTreeView package that allows you to interactively visualize your decision trees.

2.5 ConnectR

A key to data analysis is, of course, the data. The RevoScaleR package provides a way for you to connect with the data you may have stored in a variety of formats: SAS, SPSS, Teradata, ODBC,

delimited and fixed format text, and Hadoop Distributed File System (HDFS) text files. You have a choice of:

1) keeping the data as is and analyzing it directly with RevoScaleR analysis functions,
2) extracting the data you want to analyze and storing it in the efficient and higher performance .xdf file format provided with the RevoScaleR package, or
3) bringing some or all of your data into memory as an R data frame to use with any R analysis function.

To learn more, look for data sources in the RevoScaleR package.

2.6 DevelopR

Microsoft R Services provides a tool for the R developer to efficiently create sets of R scripts—the R Productivity Environment (RPE). Working on a Windows workstation with the RPE, the R developer has a full-featured Visual Studio-like integrated development environment for R, including an indispensable visual debugger for R. The RPE has a customizable workspace, including an enhanced Script Editor, an Object Browser, a Solution Explorer, and an R Command Console.

2.7 DeployR

In many enterprises, the final step is to deploy an interface to the underlying analysis to a broader audience within the organization. The optional DeployR package provides the tools for doing just that; it is a full-featured web services software development kit for R which allows programmers to use Java, JavaScript or .Net to integrate the R analysis output with a third party package. To expedite this effort, we now provide Accelerators for DeployR which are starter kits for integrating with some of our customers' favorite tools including: Microsoft Excel, Tableau, Jaspersoft, and QlikView.

Chapter 3.

Microsoft R Services Basics

This chapter describes the essentials of using Microsoft R Services: starting Microsoft R Services, stopping Microsoft R Services, and getting help in Microsoft R Services. If you are new to R, the next chapter gives a quick tutorial on what you might do between starting and stopping.

3.1 Starting Microsoft R Services

Exactly how you start Microsoft R Services depends on your operating system.

3.1.1 Starting Microsoft R Services on Windows

On Windows 7 and Windows Server 2008, you start Microsoft R Services as follows:

- Click **Start**, point to **All Programs**, point to **Revolution R**, point to **Enterprise 7.x**, and then click **Revolution R Enterprise 7.x (64)**.

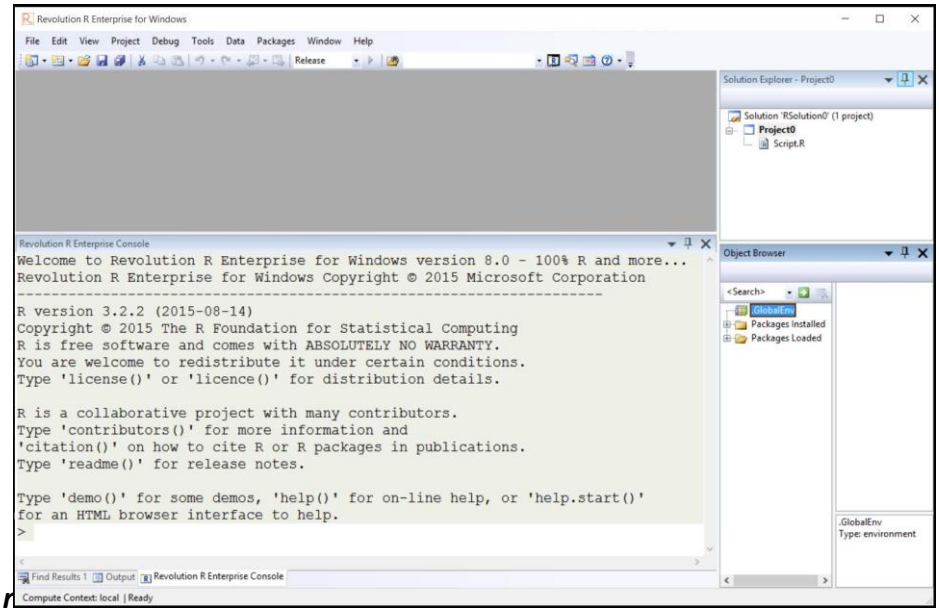
On Windows 8 and Windows Server 2012, you start Revolution R Enterprise as follows:

- Move your mouse to the lower left corner of the Desktop until **Start** pops up.
- Click **Start** to view the **Start** screen.
- Locate the tile for **Revolution R Enterprise 8.x (64).**

On Windows 10, you start Revolution R Enterprise as follows:

- Click **Start**, point to **All apps**, point to **Revolution R**, and then click **Revolution R Enterprise 8.x (64)**.

The Revolution R Enterprise R Productivity Environment opens, as shown in the figure below. For more information on using the R Productivity Environment, see the following manuals: *R Productivity Environment Getting Started Guide* (RevoRPE_Getting_Started.pdf) and *R Productivity Environment User's Guide* (RevoRPE_Users_Guide.pdf).



3.1.2 Starting Microsoft R Services on Linux

On Linux systems, you start ***Microsoft R Server*** by opening a terminal or console window, and typing Revo64. If all is well, you will see a welcome message followed by the ***Microsoft R Services*** *>:*

```
R version 3.2.2 (2015-08-14) -- "Fire Safety"
Copyright (C) 2015 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

Microsoft R Server version 8.0: an enhanced distribution of R
Microsoft packages Copyright (C) 2015 Microsoft Corporation
```

```
Type 'readme()'for release notes.

>
```

If you get the message “Revo64: Command not found," this means that ***Microsoft R Server*** is not in your search path. Check with your system administrator to find the correct path to your ***Microsoft R Server*** installation, then modify your search path (typically in your .bashrc file):

```
PATH=$PATH:/path/to/Microsoft R Server
export PATH
```

3.2 Stopping Microsoft R Services

To stop the Revolution R Enterprise RPE, close the application window.

From any command-line version of R, the standard way to exit is by calling the `q` function. (In the RPE, calling `q` is equivalent to using the Close box or File>Exit to exit.) All R functions are called by typing the name of the function, followed by a pair of parentheses that may include one or more arguments. So, to quit R, you call `q` with no arguments, following the R prompt `>`:

```
q()
```

Whenever you quit R, you are asked if you want to save the workspace image; if you have created functions or data that you want to keep, saving the workspace image will preserve them for future use. (Most R users, however, create their functions and data in script files which can be read, or *sourced*, into R. If you follow this model, you will usually say “no” to saving the workspace image.)

3.3 Getting Help

Know a function’s name, but not how to call it? Need examples of how to set up the data for a function? Help is just a few keystrokes away. R has two main functions for obtaining help: the `?` operator and the `help` function. You can use the operator by simply typing a question mark at the prompt, followed by the name of the function you want to know about:

```
?q
```

Depending on your operating system (and whether you have started the GUI help browser), help for the q function will then appear either in your console window or in a separate window.

The help function is much the same:

```
help(q)
```

Most users will probably use `?` because it is easy to type; `help` allows you to specify a number of arguments that can extend its usefulness.

Chapter 4.

An R Tutorial in 25 Functions or So

To get you started with ***Microsoft R Services***, this chapter gives a brief tutorial introduction in which you will learn 25 (or so) of the most commonly used R functions, learn to load your own small data sets into R, and begin to do useful analysis on them. We'll also provide some initial tips on the next steps for performing scalable data analysis in R. The following two chapters go into greater depth on Big Data Big Analytics in R.

4.1 Creating Vectors

R is an environment for analyzing data, so the natural starting point is with some data to analyze. For small data sets, such as the following 20 measurements of the speed of light taken from the famous Michelson-Morley experiment, it is simplest to use R's `c` function to combine the data into a vector. Type the following at the > prompt at the beginning of the line:

```
c(850, 740, 900, 1070, 930, 850, 950, 980, 980, 880,
1000, 980, 930, 650, 760, 810, 1000, 1000, 960, 960)
```

When you type the closing parenthesis and press *Enter*, R responds as follows:

```
 [1]  850  740  900 1070  930  850  950  980  980  880
[11] 1000  980  930  650  760  810 1000 1000  960  960
```

This indicates that R has interpreted what you typed, created a vector with 20 elements, and returned that vector. But we have a problem. R hasn't saved what we typed. If we want to use this vector again (and that's the usual reason for creating a vector in the first place), we need to *assign* it. The R assignment operator has the suggestive form `<-` to indicate a value is being assigned to a name. You can use most combinations of letters, numbers, and periods to form names (but note that names can't begin with a number); here we'll use `michelson`:

```
michelson <- c(850, 740, 900, 1070, 930, 850, 950, 980, 980, 880,
1000, 980, 930, 650, 760, 810, 1000, 1000, 960, 960)
```

Now R responds with a prompt; the vector is not automatically printed when it is assigned. But now we can view the vector we created by typing its name at the prompt:

```
michelson
```

```
 [1]  850  740  900 1070  930  850  950  980  980  880
[11] 1000  980  930  650  760  810 1000 1000  960  960
```

The `c` function is useful for typing in small vectors such as you might find in textbook examples, and it is also useful for combining existing vectors. For example, if we discovered another five observations that extended the Michelson-Morley data, we could extend the vector using `c` as follows:

```
michelsonNew <- c(michelson, 850, 930, 940, 970, 870)
michelsonNew
```

```
 [1]  850  740  900 1070  930  850  950  980  980  880
[11] 1000  980  930  650  760  810 1000 1000  960  960
[21]  850  930  940  970  870
```

Often for testing purposes you want to use randomly generated data. R has a number of built-in distributions from which you can generate random numbers; two of the most commonly used are the normal and the uniform distributions. To obtain a set of numbers from a normal distribution, you use the *rnorm* function:

```
normalDat <- rnorm(25)
normalDat
```

```
 [1] -0.66184983  1.71895416  2.12166699
 [4]  1.49715368 -0.03614058  1.23194518
 [7] -0.06488077  1.06899373 -0.37696531
[10]  1.04318309 -0.38282188  0.29942160
[13]  0.67423976 -0.29281632  0.48805336
[16]  0.88280182  1.86274898  1.61172529
[19]  0.13547954  1.08808601 -1.26681476
[22] -0.19858329  0.13886578 -0.27933600
[25]  0.70891942
```

By default, the data are generated from a standard normal with mean 0 and standard deviation 1. You can use the `mean` and *`sd`* arguments to *`rnorm`* to specify a different normal distribution:

```
normalSat <- rnorm(25, mean=450, sd=100)
normalSat
```

```
 [1] 373.3390 594.3363 534.4879 410.0630 307.2232
 [6] 307.8008 417.1772 478.4570 521.9336 493.2416
[11] 414.8075 479.7721 423.8568 580.8690 451.5870
[16] 406.8826 488.2447 454.1125 444.0776 320.3576
[21] 236.3024 360.6385 511.2733 508.2971 449.4118
```

Similarly, you use the *`runif`* function to generate random data from a uniform distribution:

```
uniformDat <- runif(25)
uniformDat
```

```
 [1] 0.03105927 0.18295065 0.96637386 0.71535963
 [5] 0.16081450 0.15216891 0.07346868 0.15047337
 [9] 0.49408599 0.35582231 0.70424152 0.63671421
[13] 0.20865305 0.20167994 0.37511929 0.54082887
[17] 0.86681824 0.23792988 0.44364083 0.88482396
[21] 0.41863803 0.42392873 0.24800036 0.22084038
[25] 0.48285406
```

The default uniform distribution is that over the interval 0 to 1; you can specify alternatives by setting the *`min`* and *`max`* arguments:

```
uniformPerc <- runif(25, min=0, max=100)
uniformPerc
```

```
 [1] 66.221400 12.270863 33.417174 21.985229
 [5] 92.767213 17.911602  1.935963 53.551991
 [9] 75.110760 22.436347 63.172258 95.977501
[13] 79.317351 56.767608 89.416080 79.546495
[17]  8.961152 49.315612 43.432128 68.871867
[21] 73.598221 63.888835 35.261694 54.481692
[25] 37.575176
```

Another commonly used vector is the *sequence*, a uniformly-spaced run of numbers. For the common case of a run of integers, you can use the infix operator, `:`, as follows:

```
1:10
```

```
 [1]  1  2  3  4  5  6  7  8  9 10
```

For more general sequences, use the *seq* function:

```
seq(length = 11, from = 10, to = 30)

 [1] 10 12 14 16 18 20 22 24 26 28 30

seq(from = 10,length = 20, by = 4)

 [1] 10 14 18 22 26 30 34 38 42 46 50 54 58 62 66 70 74
[18] 78 82 86
```

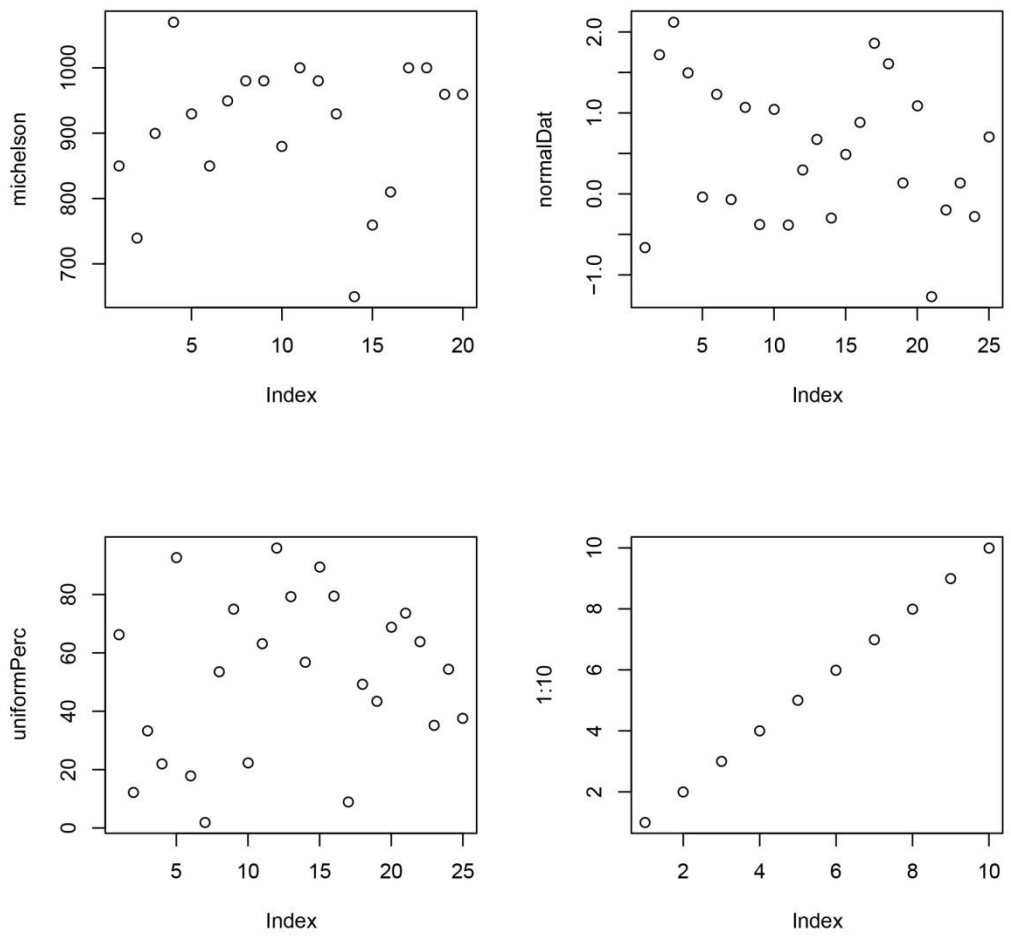
Big Data Big Analytics Tip: If you are working with big data, you'll use vectors regularly to manipulate parameters and information about your data. However, you'll typically want to store your big data sets in the RevoScaleR high-performance .xdf file format.

4.2 Exploratory Data Analysis

Once you have some data, you will want to explore it graphically. For most small data sets, the place to begin is with the `plot` function, which provides a default graphical view of the data:

```
plot(michelson)
plot(normalDat)
plot(uniformPerc)
plot(1:10)
```

For *numeric* vectors such as ours, the default view is a scatter plot of the observations against their index, resulting in the following plots:



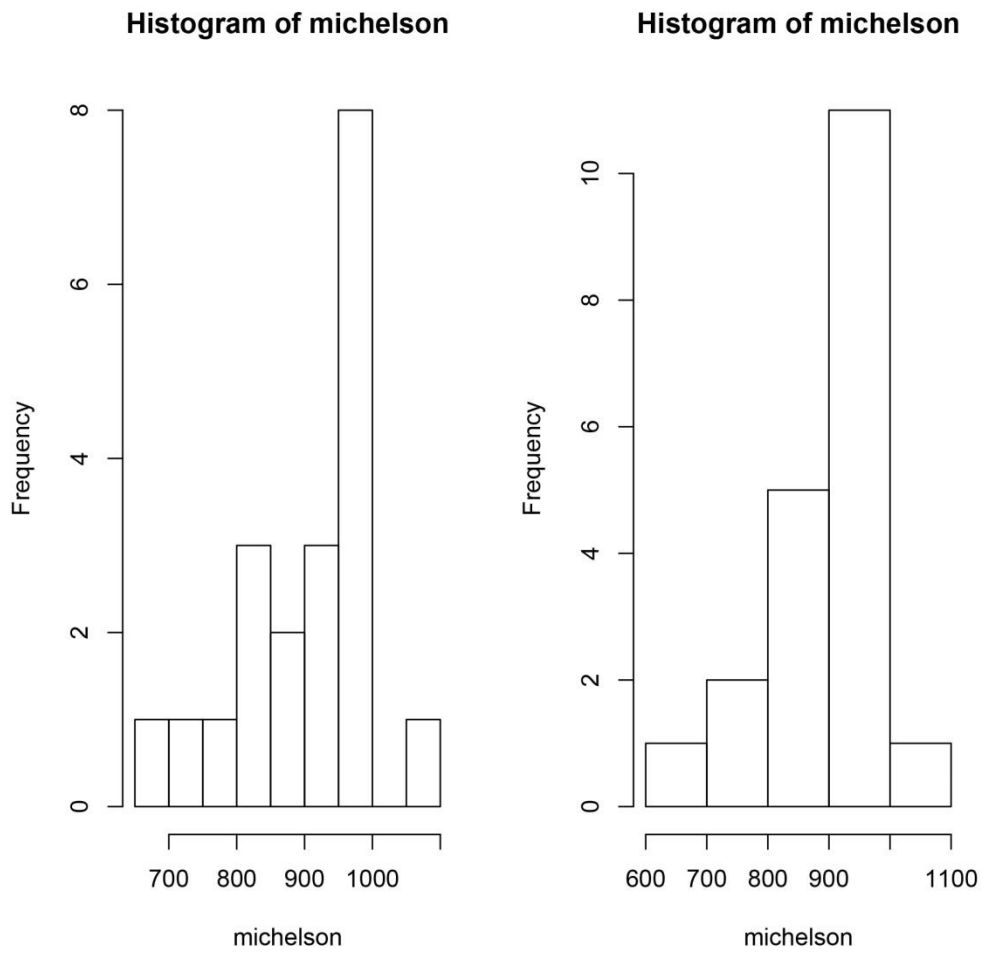
For an exploration of the shape of the data, the usual tools are *stem* (to create a stemplot) and *hist* (to create a histogram):

```
stem(michelson)

  The decimal point is 2 digit(s) to the right of the |

   6 | 5
   7 | 46
   8 | 1558
   9 | 033566888
  10 | 0007


hist(michelson)
```

The resulting histogram is shown as the left plot below. We can make the histogram look more like the stemplot by specifying the *nclass* argument to *hist*:

```
hist(michelson, nclass=5)
```

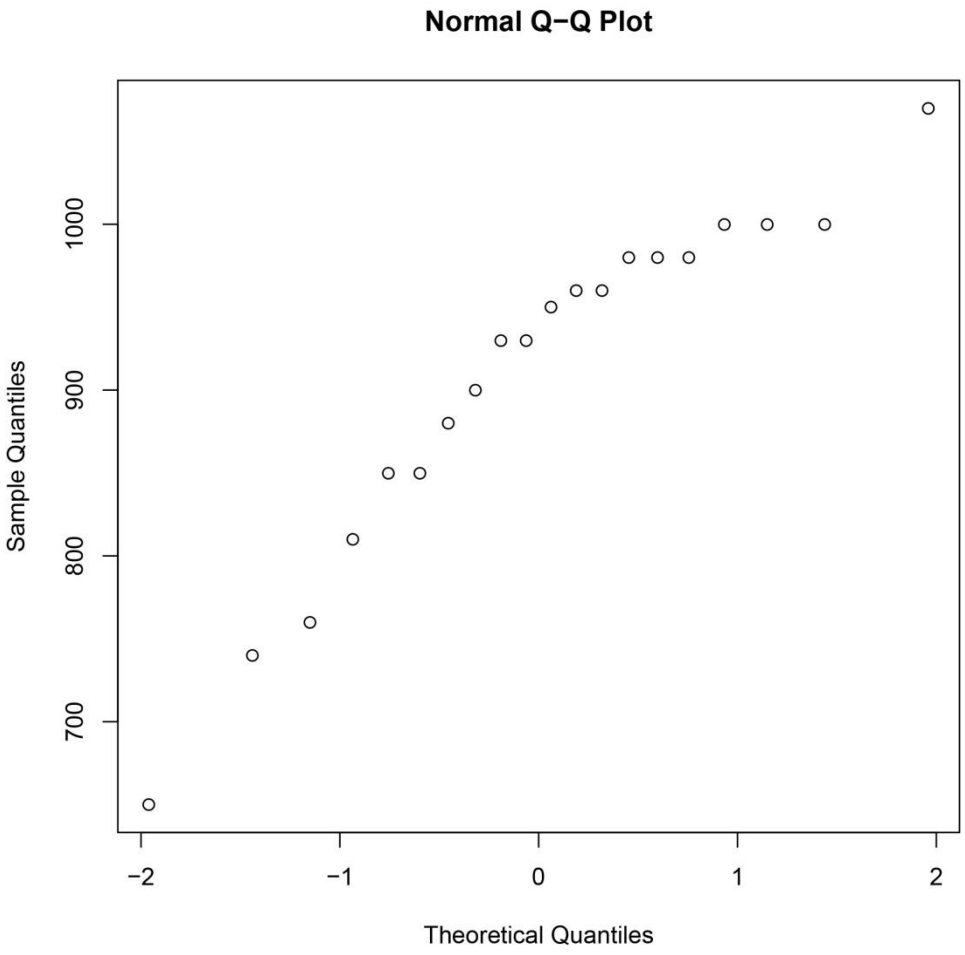
The resulting histogram is shown as the right plot in the figure below.



From the histogram and stemplot, it appears that the Michelson-Morley observations are not obviously normal. A normal Q-Q plot gives a graphical test of whether a data set is normal:

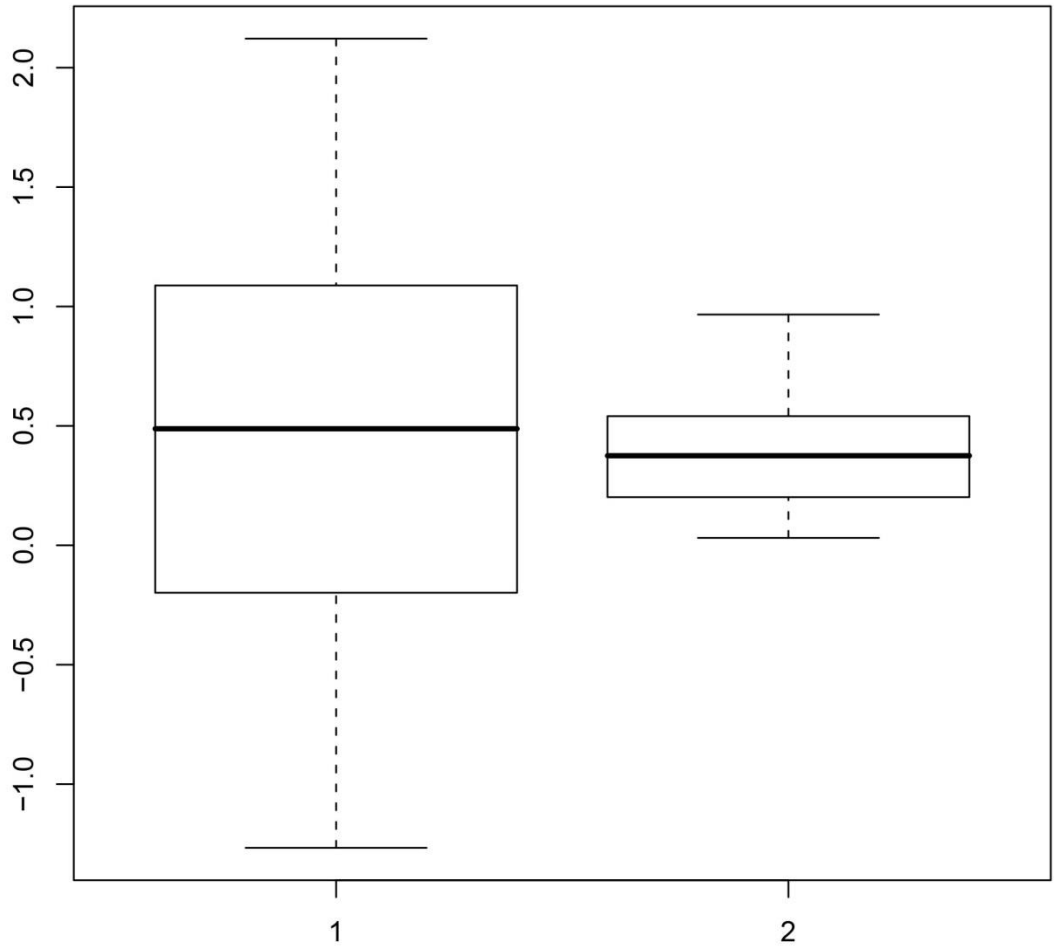
```
qqnorm(michelson)
```

The decided bend in the resulting plot confirms the suspicion that the data are not normal.



Another useful exploratory plot, especially for comparing two distributions, is the boxplot:

```
boxplot(normalDat, uniformDat)
```



Big Data Big Analytics Tip: These plots are great if you have a small data set in memory. When working with big data, some plot types may not be very informative when working directly with the data (e.g., scatter plots can produce a big blob of ink) and others may be computational intensive (e.g., require sorting data). A good starting place is the `rxHistogram` function in RevoScaleR that efficiently computes and renders histograms for large data sets. And remember that RevoScaleR functions such as `rxCube` can provide summary information that is easily amenable to the impressive plotting capabilities provided by R packages.

4.3 Summary Statistics

While an informative graphic often gives the fullest description of a data set, numerical summaries provide a useful shorthand for describing certain features of the data. For example, estimators such as the mean and median help to locate the data set, and the standard deviation and variance measure the scale or spread of the data. R has a full set of summary statistics available:

```
> mean(michelson)
[1] 909
> median(michelson)
[1] 940
> sd(michelson)
[1] 104.9260
> var(michelson)
[1] 11009.47
```

The generic summary function provides a meaningful summary of a data set; for a numeric vector it provides the five-number summary plus the mean:

```
summary(michelson)

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
    650     850     940     909     980    1070
```

> *Big Data Big Analytics Tip:* The `rxSummary` function in RevoScaleR will efficiently compute summary statistics for a data frame in memory or a large data file stored on disk.

4.4 Creating Multivariate Data Sets

In most disciplines, meaningful data sets have multiple variables, typically observations of various quantities and qualities of individual subjects. Such data sets are typically represented as tables in which the columns correspond to variables and the rows correspond to subjects, or cases. In R, such tables can be created as *data frame* objects. For example, at the 2008 All-Star Break, the Seattle Mariners had five players who met the minimum qualifications to be considered for a batting title (that is, at least 3.1 at bats per game played by the team). Their statistics are shown in the following table:

Player	Games	AB	R	H	2B	3B	HR	TB	RBI	BA	OBP	SLG	OPS
"I. Suzuki"	95	391	63	119	11	3	3	145	21	.304	.366	.371	.737
"J. Lopez"	92	379	46	113	26	1	5	156	48	.298	.318	.412	.729
"R. Ibanez"	95	370	41	101	26	1	11	162	55	.273	.338	.438	.776
"Y. Betancourt"	90	326	34	87	22	2	3	122	29	.267	.278	.374	.656

```
"A. Beltre"        92 352  46   91   16   0   16  155   46 .259 .329 .440 .769
```

Copy and paste the table into a text editor (such as Notepad on Windows, or emacs or vi on Unix-type machines) and save the file as **msStats.txt** in the working directory returned by the *getwd* function, for example:

```
getwd()
```

```
 [1] "/Users/joe"
```

You can then read the data into R using the *read.table* function. The argument *header=TRUE* specifies that the first line is a header of variable names:

```
msStats <- read.table("msStats.txt", header=TRUE)
msStats
```

```
 > msStats
           Player Games  AB  R   H X2B X3B HR  TB RBI
1     I. Suzuki     95 391 63 119  11   3  3 145  21
2      J. Lopez     92 379 46 113  26   1  5 156  48
3     R. Ibanez     95 370 41 101  26   1 11 162  55
4 Y. Betancourt     90 326 34  87  22   2  3 122  29
5     A. Beltre     92 352 46  91  16   0 16 155  46
     BA   OBP   SLG   OPS
1 0.304 0.366 0.371 0.737
2 0.298 0.318 0.412 0.729
3 0.273 0.338 0.438 0.776
4 0.267 0.278 0.374 0.656
5 0.259 0.329 0.440 0.769
```

(Notice how *read.table* changed the names of our original “2B" and “3B" columns to be valid R names; R names cannot begin with a numeral.)

Most small R data sets in daily use are data frames. The built-in package, *datasets*, is a rich source of data frames for further experimentation. In the next section, we turn our attention to the built-in data set *attitude*, part of the *datasets* package.

> *Big Data Big Analytics Tip:* Check out the *rxImport* function for an efficient and flexible way to bring data stored in a variety of data formats (e.g., text, SQL Server, ODBC, SAS, SPSS, Teradata) into a data frame in memory or an .xdf file.

4.5 Linear Models

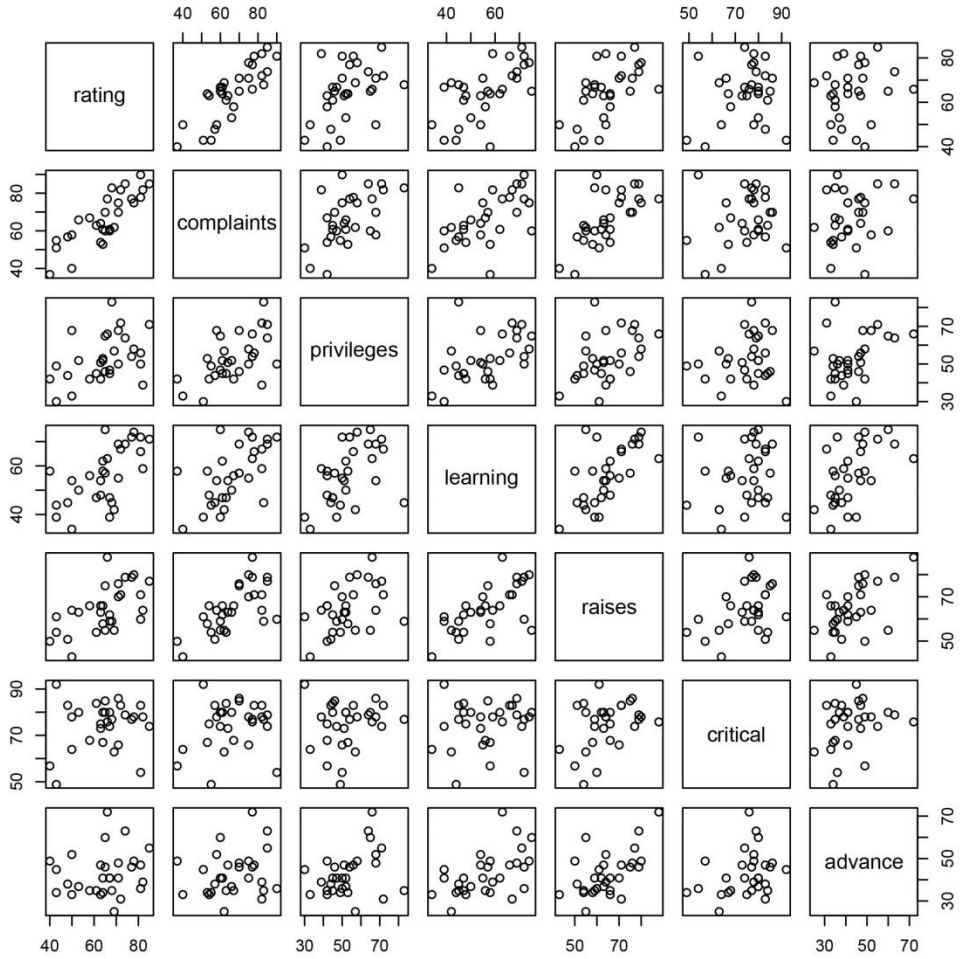
The *attitude* data set is a data frame with 30 observations on 7 variables, measuring the percent proportion of favorable responses to seven survey questions in each of 30

departments. The survey was conducted in a large financial organization; there were approximately 35 respondents in each department.

We mentioned that the *plot* function could be used with virtually any data set to get an initial visualization; let's see what it gives for the *attitude* data:

```
plot(attitude)
```

The resulting plot is a pairwise scatter plot of the numeric variables in the data set.



The first two variables (*rating* and *complaints*) show a strong linear relationship. To model that relationship, we use the `lm` function:

```
attitudeLM1 <- lm(rating ~ complaints, data=attitude)
```

To view a summary of the model, we can use the summary function:

```
summary(attitudeLM1)

Call:
lm(formula = rating ~ complaints, data = attitude)
```

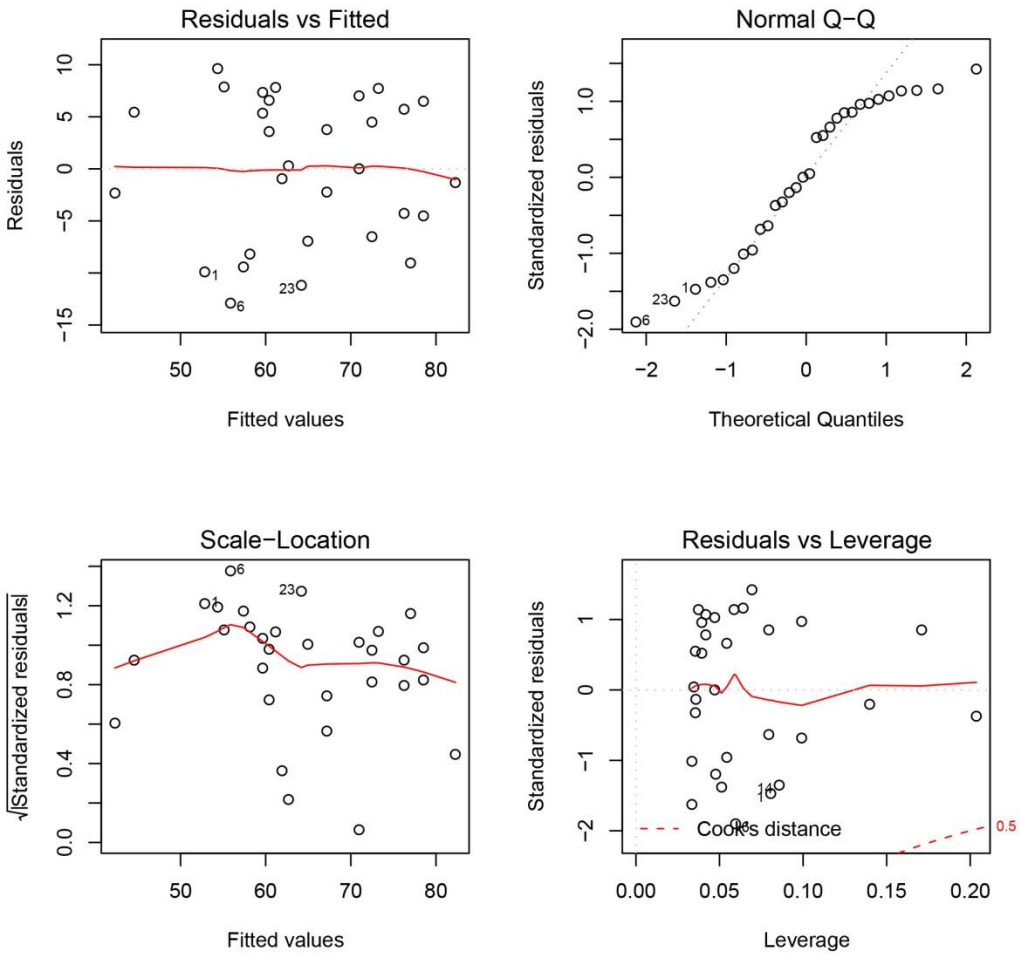
```
Residuals:
     Min       1Q   Median       3Q      Max
-12.8799  -5.9905   0.1783   6.2978   9.6294

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 14.37632    6.61999   2.172   0.0385 *
complaints   0.75461    0.09753   7.737 1.99e-08 ***
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1  1

Residual standard error: 6.993 on 28 degrees of freedom
Multiple R-squared: 0.6813, Adjusted R-squared: 0.6699
F-statistic: 59.86 on 1 and 28 DF,  p-value: 1.988e-08
```

We can also try to visualize the model using `plot`:

```
plot(attitudeLM1)
```

The default plot for a fitted linear model is a set of four plots; by default they are shown one at a time, and you are prompted before each new plot is displayed. To view them all at once, use the *par* function with the *mfrow* parameter to specify a 2×2 layout:

```
par(mfrow=c(2,2))
plot(attitudeLM1)
```



Big Data Big Analytics Tip: The `rxLinMod` function is a full-featured alternative to `lm` that can efficiently handle large data sets. Also look at `rxLogit` and `rxGlm` as alternatives to `glm`, `rxKmeans` as an alternative to `kmeans`, and `rxDTree` as an alternative to `rpart`.

4.6 Matrices and `apply`

A *matrix* is a two-dimensional data array. Unlike data frames, which can have different data types in their columns, matrices may contain data of only one type. Most commonly, matrices are used to hold numeric data. You create matrices with the `matrix` function:

```
A <- matrix(c(3, 5, 7, 9, 13, 15, 8, 4, 2), ncol=3)
A
```

```
     [,1] [,2] [,3]
[1,]    3    9    8
```

```
  [2,]    5   13    4
  [3,]    7   15    2
```

```
B <- matrix(c(4, 7, 9, 5, 8, 6), ncol=3)
B
```

```
      [,1] [,2] [,3]
  [1,]    4    9    8
  [2,]    7    5    6
```

Ordinary arithmetic acts *element-by-element* on matrices:

```
A + A
```

```
      [,1] [,2] [,3]
  [1,]    6   18   16
  [2,]   10   26    8
  [3,]   14   30    4
```

```
A * A
```

```
      [,1] [,2] [,3]
  [1,]    9   81   64
  [2,]   25  169   16
  [3,]   49  225    4
```

Matrix multiplication in the linear algebra sense requires a special operator, `%*%`:

```
 A %*% A
```

```
      [,1] [,2] [,3]
  [1,]  110  264   76
  [2,]  108  274  100
  [3,]  110  288  120
```

Matrix multiplication requires two matrices to be *conformable*, which means that the number of *columns* of the first matrix is equal to the number of *rows* of the second:

```
B %*% A
```

```
      [,1] [,2] [,3]
  [1,]  113  273   84
  [2,]   88  218   88
```

```
A %*% B
```

```
 Error in A %*% B : non-conformable arguments
```

When you need to manipulate the rows or columns of a matrix, an incredibly useful tool is the apply function. With `apply`, you can apply a function to all the rows or columns of matrix at once. For example, to find the column products of `A`, you could use `apply` as follows:

```
apply(A, 2, prod)
```

```
  [1]  105 1755   64
```

The row products are just as simple:

```
apply(A, 1, prod)
```

```
 [1] 216 260 210
```

To sort the columns of *A*, just replace *prod* with *sort*:

```
apply(A, 2, sort)
```

```
      [,1] [,2] [,3]
[1,]    3    9    2
[2,]    5   13    4
[3,]    7   15    8
```

4.7 Lists and *lapply*

A *list* in R is a very flexible data object that can be used to combine data of different types and different lengths for almost any purpose. Arbitrary lists can be created with either the `list` function or the `c` function; many other functions, especially the statistical modeling functions, return their output as list objects.

For example, we can combine a character vector, a numeric vector, and a numeric matrix in a single list as follows:

```
list1 <- list(x = 1:10, y = c("Tami", "Victor", "Emily"),
  z = matrix(c(3, 5, 4, 7), nrow=2))
list1
```

```
$x
 [1]  1  2  3  4  5  6  7  8  9 10

$y
[1] "Tami"   "Victor" "Emily"

$z
     [,1] [,2]
[1,]    3    4
[2,]    5    7
```

The function *lapply* can be used to apply the same function to each component of a list in turn:

```
lapply(list1, length)
```

```
$x
[1] 10

$y
[1] 3
```

```
$z
[1] 4
```

Big Data Big Analytics Tip: You will regularly use lists and functions that manipulate them when handling input and output for your big data analyses.

4.8 Packages

An R *package* is a collection of R objects and documentation. The R objects may be functions, data sets, or a combination, and they are usually related in some way, although this is not an absolute requirement. The standard R distribution consists of the following packages:

stats	graphics	grDevices	utils
datasets	methods	base	
KernSmooth	MASS	Matrix	boot
class	cluster	codetools	compiler
foreign	grid	lattice	mgcv
nlme	nnet	parallel	rpart
spatial	splines	stats4	survival
tcltk	tools	boot	

The packages in the top two lines are automatically loaded when you start R. You can load other packages using the `library` function. For example, to load the `MASS` library, which contains functions and data sets used in the book *Modern Applied Statistics with S* by Venables and Ripley, you call `library` as follows:

```
library(MASS)
```

Other packages are available through the Comprehensive R Archive Network (CRAN); to obtain a package you use the `install.packages` function:

```
install.packages("SuppDists")
```

```
 trying URL 'http://cran.fhcrc.org/src/contrib/SuppDists_1.1-8.tar.gz'
 Content type 'application/x-gzip' length 139864 bytes (136 Kb)
 opened URL
 ==================================================
 downloaded 136 Kb

 * installing *source* package 'SuppDists' ...
 ** libs
```

```
** arch - x86_64
g++ -arch x86_64  -I/opt/REvolution/Revo-
3.2/Revo64/R.framework/Resources/includ
e -I/opt/REvolution/Revo-3.2/Revo64/R.framework/Resources/include/x86_64  -
I/usr
/local/include    -fPIC  -g -O2 -c dists.cc -o dists.o
g++ -arch x86_64 -dynamiclib -Wl,-headerpad_max_install_names -
undefined dynamic
_lookup -single_module -multiply_defined suppress -L/usr/local/lib -
o SuppDists.
so dists.o -F/opt/REvolution/Revo-3.2/Revo64/R.framework/.. -framework R -Wl,-
fr
amework -Wl,CoreFoundation
** R
** preparing package for lazy loading
** help
*** installing help indices
** building package indices ...
* DONE (SuppDists)

The downloaded packages are in
        '/private/var/folders/cy/cy2tNRpEHrmxPZPrN1EINU+++TI/-Tmp-
/RtmpZnKn6Z/do
wnloaded_packages'
Updating HTML index of packages in '.Library'
```

(On Linux systems, you should not use CRAN as a source for third-party packages, because they may require a current version of R that may be different than that distributed with Microsoft R Services. Microsoft R Services sets the default repository to a versioned source repository maintained by Revolution Analytics at packages.revolutionanalytics.com. An alternative is to point to one of the daily CRAN snapshots maintained at mran.revolutionanalytics.com.)

> *Big Data Big Analytics Tip:* The RevoScaleR package is included with every distribution of Revolution R Enterprise, and is automatically loaded into memory when you start the program. So all of the "rx" functions mentioned in these tips are at your fingertips. You can get information on them by using the `?` at the command line, for example: `?rxLinMod`

4.9 Using Microsoft R Services via Rscript and R CMD BATCH

Microsoft R Services is intended for high-performance computing and analytics, and some users are accustomed to running their analyses via batch mode and command-line scripting. `R CMD BATCH` generally works with ***Microsoft R Services*** with no modifications needed, but to get full advantage of the ***Microsoft R Services*** extensions with other command line invocations, you need to know a little bit about how ***Microsoft R Services*** works. ***Microsoft R Services*** is 100% R, with the standard R BLAS and LAPACK libraries substituted out for the Intel Math Kernel Libraries, and with a number of additional packages. Some of these packages are added to the default package list by the `Rprofile.site` file distributed with ***Microsoft R Services***. If you use `Rscript` (or, on

some systems, the equivalent Revoscript) with a ***Microsoft R Services*** script, be sure to add the flag `--default-packages=` to your call; this ensures that the ***Microsoft R Services*** default packages are loaded (including the methods package from base R).

Similarly, you should avoid the –vanilla construction for invoking ***Microsoft R Services***; this method of invocation avoids evaluating the `Rprofile.site` file, so that this is equivalent to calling R without the ***Microsoft R Services*** extensions (except the MKL BLAS and LAPACK libraries).

Chapter 5.

Tips on Computing with Big Data in R

Big data is ubiquitous. The good news is that it provides great opportunities for the data analyst. With more data comes more information and more insight. You can relax assumptions required with smaller data sets and let the data speak for itself. But big data also presents problems. The size of data sets is now increasing much more rapidly than the speed of single cores, of RAM, and of hard drives. Most software tools aren't handling this well; many of us have experienced that moment when our data has grown just a little too big—and our software has stopped working.

The use of the R statistical programming language has seen phenomenal growth because of R's flexibility and power. Cutting-edge algorithms are written in R, and analysts love it for quick prototyping. But R users are faced with two big problems when scaling to big data: capacity and speed. Data sets typically must fit into memory, and even if they can, there are limits to what types of analyses can be done. Even without a capacity limit, computation may be too slow to be useful.

If you are used to working with smaller data sets in R, you will want to think differently about how you perform your analyses when using big data. If you are used to working in a High Performance Computing (HPC) environment, you will also want to think differently when you

add analysis of big data to the picture. High Performance Computing is CPU centric, typically focusing on using many cores to perform lots of processing on small amounts of data. High Performance Analytics (HPA) is data centric. The focus is on feeding data to the cores—on disk I/O, data locality, efficient threading, and data management in RAM.

The ***RevoScaleR*** package that is included with ***Microsoft R Services*** provides tools and examples for addressing the speed and capacity issues involved in High Performance Analytics. It provides data management and analysis functionality that scales from small, in-memory data sets to huge data sets stored on disk. The analysis functions are threaded to use multiple cores, and computations can be distributed across multiple computers (nodes) on a cluster or in the cloud.

Here are some tips for handling big data with R:

5.1 Upgrade Your Hardware

It is always best to start with the easiest things first, and in some cases getting a better computer, or improving the one you have, can help a great deal. Usually the most important consideration is memory. If you are analyzing data that just about fits in R on your current system, getting more memory will not only let you finish your analysis, it is also likely to speed things up by a lot. This is because your operating system starts to "thrash" when it gets low on memory, removing some things from memory to let others continue to run. This can slow your system to a crawl. Getting more cores can also help, but only up to a point. R itself can generally only use one core at a time internally. In addition, for many data analysis problems the bottlenecks are disk I/O and the speed of RAM, so efficiently using more than 4 or 8 cores on commodity hardware can be difficult.

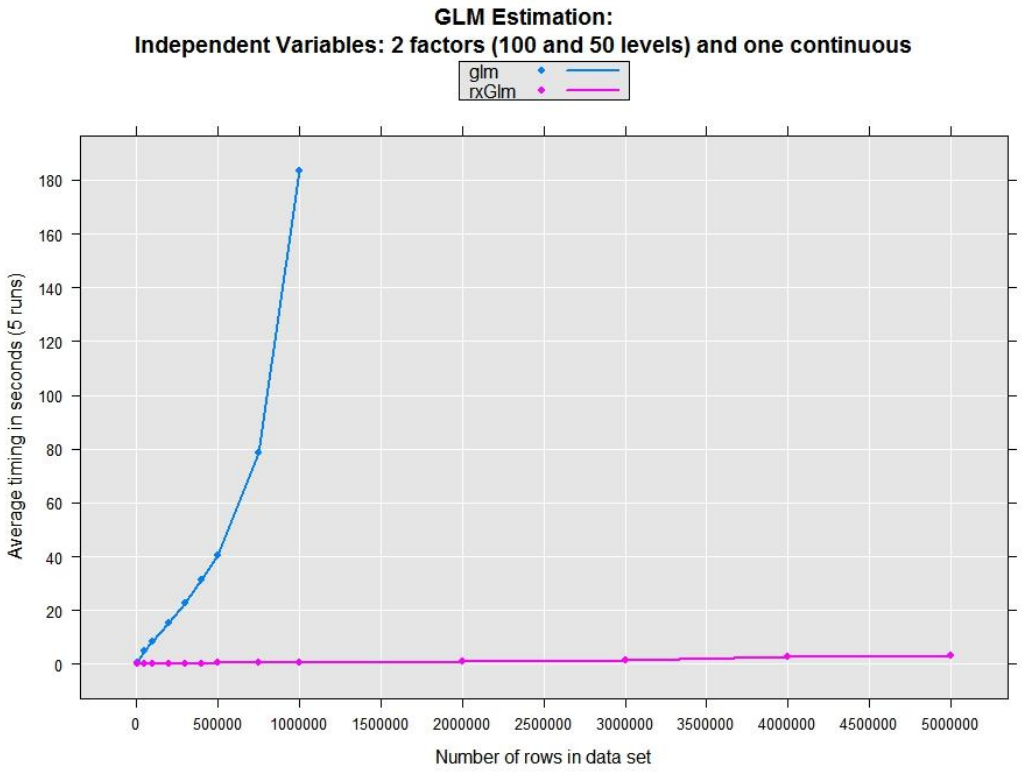
5.2 Upgrade Your Software

Some software is simply more optimized for use with big data. For instance, getting better math libraries can greatly speed some computations. R allows its core math libraries to be replaced, and in ***Microsoft R Services*** they are replaced with extremely fast, threaded libraries. And, of course, the ***RevoScaleR*** package provides the underlying high-performance compute engine used by its Big Data Big Analytics algorithms.

5.3 Minimize Copies of Data

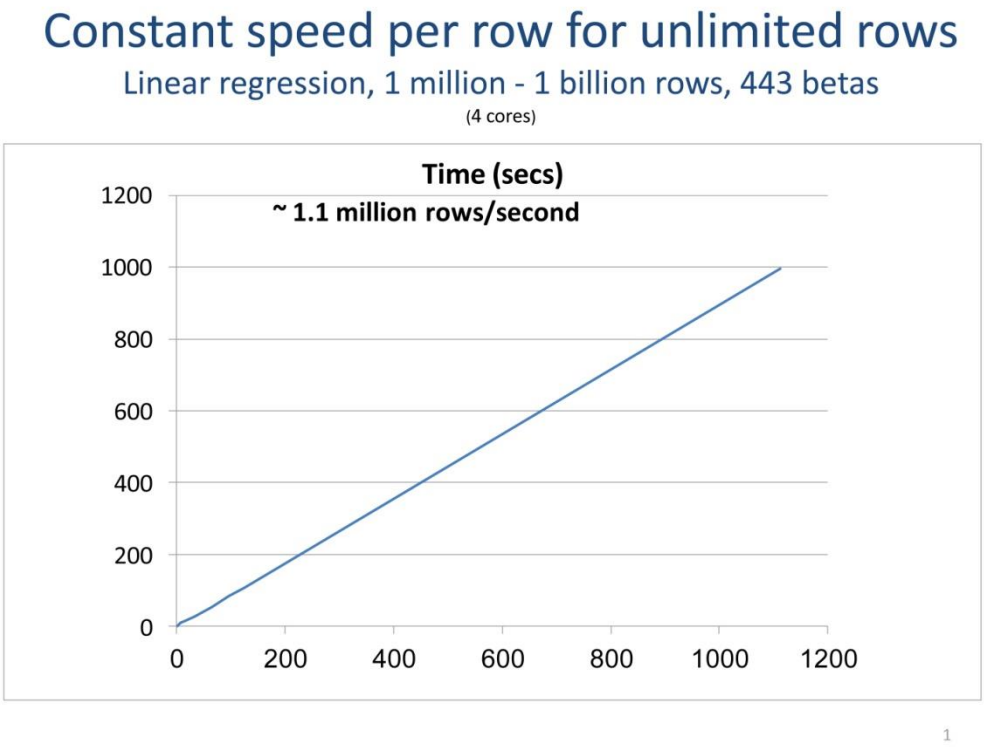
When working with small data sets, an extra copy is not a problem. With big data it can slow the analysis, or even bring it to a screeching halt. Be aware of the 'automatic' copying that occurs in R. For example, if a data frame is passed into a function, a copy is only made if the data frame is modified. But if a data frame is put into a list, a copy is automatically made. In many of the basic analysis algorithms, such as `lm` and `glm`, multiple copies of the data set are

made as the computations progress, resulting in serious limitations in processing big data sets. The RevoScaleR analysis functions (for instance, `rxSummary`, `rxCube`, `rxLinMod`, `rxLogit`, `rxGlm`, `rxKmeans`) are all implemented with a focus on efficient use of memory; data is not copied unless absolutely necessary. The plot below shows an example of how reducing copies of data and tuning algorithms can dramatically increase speed and capacity.



5.4 Process Data in Chunks

Processing your data a chunk at a time is the key to being able to scale your computations without increasing memory requirements. External memory (or "out-of-core") algorithms don't require that all of your data be in RAM at one time. Data is processed a chunk at time, with intermediate results updated for each chunk. When all of the data is processed, final results are computed. The core functions provided with ***RevoScaleR*** all process data in chunks. So, if the number of rows of your data set doubles, you can still perform the same data analyses—it will just take longer, typically scaling linearly with the number of rows. Your analysis is not bound by memory constraints. The plot below shows how data chunking allows unlimited rows in limited RAM.



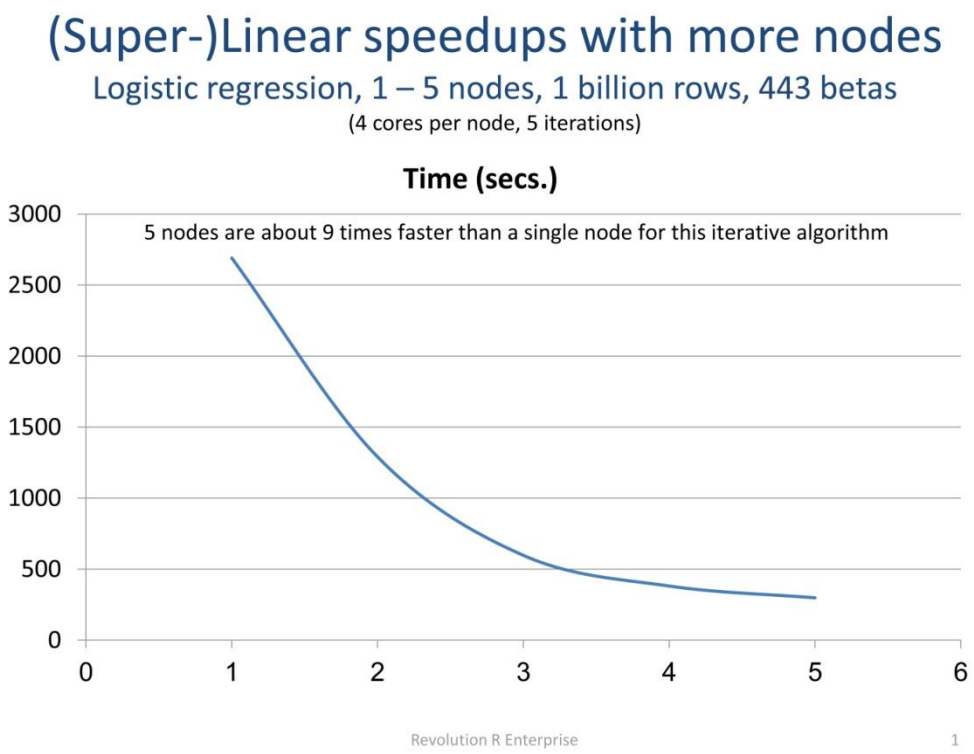
The `biglm` package, available on CRAN, also estimates linear and generalized linear models using external memory algorithms, although they are not parallelized.

5.5 Compute in Parallel Across Cores or Nodes

Using more cores and more computers (nodes) is the key to scaling computations to really big data. Since data analysis algorithms tend to be I/O bound when data cannot fit into memory, the use of multiple hard drives can be even more important than the use of multiple cores.

The ***RevoScaleR*** analysis functions are written to automatically compute in parallel on available cores, and can also be used to automatically distribute computations across the nodes of a cluster. These functions combine the advantages of external memory algorithms (see 'Process Data in Chunks' above) with the advantages of High Performance Computing. That is, these are Parallel External Memory Algorithm's (PEMA's)—external memory algorithms that have been parallelized. Such algorithms process data a chunk at a time in parallel, storing intermediate results from each chunk and combining them at the end. Iterative algorithms repeat this process until convergence is determined. Any external memory algorithm that is not "inherently sequential" can be parallelized; results for one chunk of data cannot depend upon prior results. Dependence on data from a prior chunk is OK, but must be handled specially. The plot below shows an example of how using multiple computers can dramatically increase

speed, in this case taking advantage of memory caching on the nodes to achieve super-linear speedups.



Microsofts' `foreach` package, which is open source and available on CRAN, provides very easy-to-use tools for executing R functions in parallel, both on a single computer and on multiple computers. This is particularly useful for "embarrassingly parallel" types of computations such as simulations, which do not involve lots of data and do not involve communication among the parallel tasks.

5.6 Take Advantage of Integers

In R the two choices for "continuous" data are `numeric`, which is an 8 byte (double) floating point number and `integer`, which is a 4 byte integer. If your data can be stored and processed as an integer, there can be big advantages to doing so. First, it will only take half of the memory. Second, in some cases integers can be processed much faster than doubles. For example, if you have a variable whose values are integral numbers in the range from 1 to 1000 and you want to find the median, it is much faster to count all the occurrences of the integers than it is to sort the variable. A tabulation of all the integers, in fact, can be thought of as a way to compress the data with no loss of information. The resulting tabulation can be converted into an exact empirical distribution of the data by dividing the counts by the sum of the counts,

and all of the empirical quantiles including the median can be obtained from this. The R function `tabulate` can be used for this, and is very fast. The following code illustrates this:

```
nd = sample(as.numeric(1:1000),size = 1e+8, replace = TRUE)
system.time(nit <- tabulate(as.integer(nd)))
system.time(nds <- sort(nd))
```

A vector of 100 million doubles is created, with randomized integral values in the range from 1 to 1,000. Sorting this vector takes about 15 times longer than converting to integers and tabulating, and 25 times longer if the conversion to integers is not included in the timing (this is relevant if you convert to integers once and then operate multiple times on the resulting vector).

Sometimes decimal numbers can be converted to integers without losing information. An example is temperature measurements of the weather, such as 32.7, which can be multiplied by 10 to convert them into integers.

RevoScaleR provides several tools for the fast handling of integral values. For instance, in formulas for linear and generalized linear models and other analysis functions, the "F()" function can be used to virtually "convert" numeric variables into factors, with the levels represented by integers. The `rxCube` function allows rapid tabulations of factors and their interactions (for example, age by state by income) for arbitrarily large data sets.

5.7 Store Your Data Efficiently

If your data doesn't easily fit into memory, you'll want to store it efficiently for fast access from disk. If you use appropriate data types, you can save on storage space and access time. Take advantage of integers, and store data in 32-bit floats not 64-bit doubles. A 32-bit float can represent 7 decimal digits of precision, which is more than enough for most data, and it takes up half the space of doubles. (Save the 64-bit doubles for computations).

Recognize that relational databases are not always optimal for storing data for analysis. Even with the best indexing they are typically not designed to provide fast sequential reads of blocks of rows for specified columns, which is the key to fast access to data on disk. ***RevoScaleR*** provides an efficient .xdf file format that provides storage of a wide variety of data types, and is designed for fast sequential reads of blocks of data. There are tools for rapidly accessing data in .xdf files from R and for importing data into this format from SAS, SPSS, and text files and SQL Server, Teradata and ODBC connections.

5.8 Only Read in The Data That Is Needed

Even though a data set may have many thousands of variables, typically not all of them are being analyzed at one time. If you have your entire data set in memory, you can easily run into

out-of-memory problems. By just reading from disk the actual variables and observations you will use in analysis, you can speed up the analysis considerably. ***RevoScaleR*** functions provide this automatically. For example, when estimating a model, only the variables used in the model are read from the .xdf file.

5.9 Avoid Loops when Transforming Data

It is well-known that processing data in loops in R can be very slow compared with vector operations. For example, if you compare the timings of adding two vectors, one with a loop and the other with a simple vector operation, you will find the vector operation to be orders of magnitude faster:

```
n <- 1000000
x1 <- 1:n
x2 <- 1:n
y <- vector()
system.time( for(i in 1:n){ y[i] <- x1[i] + x2[i] })
system.time( y <- x1 + x2)
```

On a good laptop the loop over the data was timed at about 430 seconds, while the vectorized add is barely timeable. In R the core operations on vectors are typically written in C, C++ or FORTRAN, and these compiled languages can provide much greater speed for this type of code than can the R interpreter.

5.10 Use C, C++, or FORTRAN for Critical Functions

One of the best features of R is its ability to integrate easily with other languages, including C, C++, and FORTRAN. You can pass R data objects to other languages, do some computations, and return the results in R data objects. It is typically the case that only small portions of an R program can benefit from the speedups that compiled languages like C, C++, and FORTRAN can provide. Indeed, much of the code in the base and recommended packages in R is written in this way—the bulk of the code is in R but a few core pieces of functionality are written in C, C++, or FORTRAN. The type of code that benefits the most from this involves loops over data vectors. The package `Rcpp`, which is available on CRAN, provides tools that make it very easy to convert R code into C++ and to integrate C and C++ code into R. Of course, before writing code in another language, it pays to do some research to see if the type of functionality you want is already available in R, either in the base and recommended packages or in a 3rd party package. For example, all of the core algorithms for the ***RevoScaleR*** package are written in optimized C++ code. It also pays to do some research to see if there is publically-available code in one of these compiled languages that does what you want.

5.11 Process Data Transformations in Batches

When working with small data sets, it is common to perform data transformations one at a time. For instance, one line of code might create a new variable, and the next line might multiply that variable by 10. Each of these lines of code processes all rows of the data. With a big data set that cannot fit into memory, there can be substantial overhead to making a pass through the data. With ***RevoScaleR***'s `rxDataStep` function you can specify multiple data transformations that can be performed in just one pass through the data, processing the data a chunk at a time. A little planning ahead can save a lot of time.

5.12 User Row-Oriented Data Transformations where Possible

When data is processed in chunks, basic data transformations for a single row of data should in general not be dependent on values in other rows of data. The key is that your transformation expression should give the same result even if only some of the rows of data are in memory at one time. Data manipulations using lags can be done but require special handling.

5.13 Handle Categorical Variables Efficiently and with Care.

Categorical or factor variables are extremely useful in visualizing and analyzing big data, but they need to be handled efficiently with big data because they are typically expanded when used in modeling. For example, if you use a factor variable with 100 categories as an independent variable in a linear model with `lm`, behind the scenes 100 dummy variables are created when estimating the model. The analysis modeling functions in ***RevoScaleR*** use special handling of categorical data to minimize the use of memory when processing them; they do not generally need to explicitly create dummy variable to represent factors.

Creating factor variables also often takes more careful handling with big data sets. This is because not all of the factor levels may be represented in a single chunk of data. For example, if you want to use the `factor` function in a data transformation used on chunks of data, you must explicitly specify the levels or you might end up with incompatible factor levels from chunk to chunk. The `rxImport` and `rxFactors` functions in ***RevoScaleR*** provide functionality for creating factor variables in big data sets.

5.14 Be Aware of Output with the Same Number of Rows as Your Data.

Most analysis functions return a relatively small object of results that can easily be handled in memory. But occasionally, output will have the same number of rows as your data, for example, when computing predictions and residuals from a model. In order for this to scale, you will want the output written out to a file rather than kept in memory. For this reason, the ***RevoScaleR*** modeling functions such as `rxLinMod`, `rxLogit`, and `rxGlm` do not automatically

compute predictions and residuals. The `rxPredict` function provides this functionality and can add predicted values to an existing .xdf file.

5.15 Think Twice Before Sorting.

When working with small data sets, it is common to sort data at various stages of the analysis process. Although ***RevoScaleR***'s `rxSort` function is very efficient for .xdf files and can handle data sets far too large to fit into memory, sorting is by nature a time-intensive operation, especially on big data.

One of the major reasons for sorting is to compute medians and other quantiles. As noted above in the section on taking advantage of integers, when the data consists of integral values, a tabulation of those values is generally much faster than sorting and gives exact values for all empirical quantiles. Even when the data is not integral, scaling the data and converting to integers can give very fast and accurate quantiles. As an example, if the data consists of floating point values in the range from 0 to 1,000, converting to integers and tabulating will bound the median or any other quantile to within two adjacent integers. Interpolation within those values can get you closer, as can a small number of additional iterations. If the original data falls into some other range (for example, 0 to 1), scaling to a larger range (for example, 0 to 1,000) can accomplish the same thing. The `rxQuantile` function uses this approach to rapidly compute approximate quantiles for arbitrarily large data.

Another major reason for sorting is to make it easier to compute aggregate statistics by groups. If the data are sorted by groups, then contiguous observations can be aggregated. However, it is often possible, and much faster, to make a single pass through the original data and to accumulate the desired statistics by group. The `aggregate` function can do this for data that fits into memory, and ***RevoScaleR***'s `rxSummary`, `rxCube`, and `rxCrossTabs` provide extremely fast ways to do this on large data.

The ***RevoScaleR*** functions `rxRoc`, and `rxLorenz` are other examples of 'big data' alternatives to functions that traditionally rely on sorting.

In summary, by using the tips and tools outlined above you can have the best of both worlds: the ability to rapidly extract information from big data sets using R and the flexibility and power of the R language to manipulate and graph this information.

Chapter 6.

Getting Started with Big Data in R

The ***RevoScaleR*** package included in ***Microsoft R Services*** provides a framework for quickly writing start-to-finish, scalable R code for data analysis. Even if you are relatively new to R, you can get started with just a few basic functions.

6.1 Step 1: Accessing Your Data with *rxImport*

The *rxImport* function allows you to import data from fixed or delimited text files, SAS files, SPSS files, or a SQL Server, Teradata or ODBC connection. There's no need to have SAS or SPSS installed on your system to import those file types, but you will need an ODBC driver for your data base to access it. Let's use an example of a delimited text file available in the sample data directory of the ***RevoScaleR*** package, a small data set containing simulated mortgage default data. We'll store the location of the file in a character string, then import the data into an in-memory data set (data frame) using the default settings:

```
inDataFile <- file.path(rxGetOption("sampleDataDir"),
     "mortDefaultSmall2000.csv")

mortData <- rxImport(inData = inDataFile)
```

If we think that we may want to do the same analysis on a larger data set later, we can prepare for that by putting placeholders in our code for output files. If we set these values to a file path, the data will be stored on disk in the efficient .xdf file format, rather than storing it in memory. The output object returned from *rxImport* would be a small object representing the .xdf file (an *RxXdfData* object), rather than a data frame containing all of the data. But for the time being, we will continue to work with the data in memory. We can do this by setting the *outFile* parameters to NULL. The following code will accomplish the same importing task of that above:

```
outFile <- NULL
outFile2 <- NULL
mortData <- rxImport(inData = inDataFile, outFile = outFile)
```

6.2 Step 2: A Quick Look at the Data

There are a number of basic methods we can use to quickly get some information about the data set and its variables that will work on the output of *rxImport*, whether it is a data frame or *RxXdfData* object. For example, to get the number of rows, cols, and names of the imported data:

```
nrow(mortData)
ncol(mortData)
names(mortData)
```

```
> nrow(mortData)
[1] 10000
> ncol(mortData)
[1] 6
> names(mortData)
[1] "creditScore" "houseAge"    "yearsEmploy" "ccDebt"      "year"
[6] "default"
```

To print out the first few rows of the data set, you can use *head()*:

```
head(mortData, n = 3)
```

```
  creditScore houseAge yearsEmploy ccDebt year default
1         691       16           9   6725 2000       0
2         691        4           4   5077 2000       0
3         743       18           3   3080 2000       0
```

The *rxGetInfo* function allows you to quickly get information about your data set and its variables all at one time, including more information about variable types and ranges. Let's try it on *mortData*, having the first 3 rows of the data set printed out:

```
rxGetInfo(mortData, getVarInfo = TRUE, numRows=3)
```

The output shows us:

```
Data frame: mortData
Data frame: mortData
Number of observations: 10000
Number of variables: 6
Variable information:
Var 1: creditScore, Type: integer, Low/High: (486, 895)
Var 2: houseAge, Type: integer, Low/High: (0, 40)
Var 3: yearsEmploy, Type: integer, Low/High: (0, 14)
Var 4: ccDebt, Type: integer, Low/High: (0, 12275)
Var 5: year, Type: integer, Low/High: (2000, 2000)
Var 6: default, Type: integer, Low/High: (0, 1)
Data (3 rows starting with row 1):
  creditScore houseAge yearsEmploy ccDebt year default
1         691       16           9   6725 2000       0
2         691        4           4   5077 2000       0
3         743       18           3   3080 2000       0
```

6.3 Step 3: Data Selection and Transformations with *rxDataStep*

The *rxDataStep* function provides a framework for the majority of your data manipulation tasks. It allows for row selection (the *rowSelection* argument), variable selection (the *varsToKeep* or *varsToDrop* arguments), and the creation of new variables from existing ones (the *transforms* argument). Here's an example that does all three with one function call:

```
mortDataNew <- rxDataStep(
          # Specify the input data set
          inData = mortData,
          # Put in a placeholder for an output file
          outFile = outFile2,
          # Specify any variables to keep or drop
          varsToDrop = c("year"),
          # Specify rows to select
          rowSelection = creditScore < 850,
          # Specify a list of new variables to create
          transforms = list(
              catDebt = cut(ccDebt, breaks = c(0, 6500, 13000),
                  labels = c("Low Debt", "High Debt")),
              lowScore = creditScore < 625))
```

Our new data set, *mortDataNew*, will not have the variable year, but adds two new variables: a categorical variable named *catDebt* that uses R's cut function to break the *ccDebt* variable into two categories, and a logical variable, *lowScore*, that will be TRUE for individuals with low credit scores. These *transforms* expressions follow the rule that they must be able to operate on a chunk of data at a time; that is, the computation for a single row of data cannot depend on values in other rows of data. With the *rowSelection* argument, we have also removed any observations with high credit scores, above or equal to 850. We can use the *rxGetVarInfo* function to confirm:

```
rxGetVarInfo(mortDataNew)
```

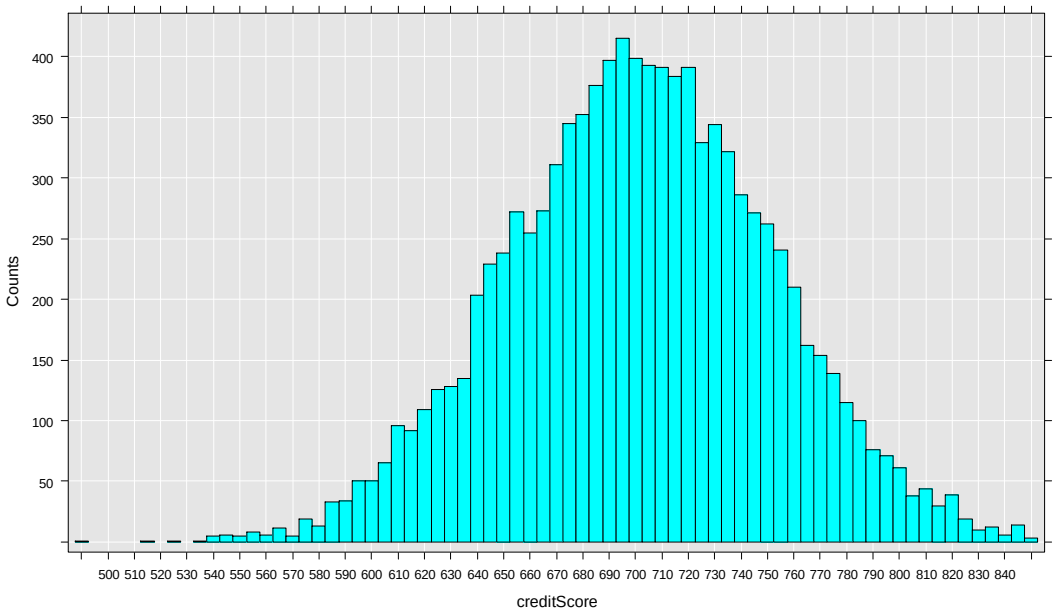
```
Var 1: creditScore, Type: integer, Low/High: (486, 847)
Var 2: houseAge, Type: integer, Low/High: (0, 40)
Var 3: yearsEmploy, Type: integer, Low/High: (0, 14)
Var 4: ccDebt, Type: integer, Low/High: (0, 12275)
Var 5: default, Type: integer, Low/High: (0, 1)
```

```
Var 6: catDebt
       2 factor levels: Low Debt High Debt
Var 7: lowScore, Type: logical, Low/High: (0, 1)
```

6.4 Step 4: Visualizing Your Data with *rxHistogram*, *rxCube*, and *rxLinePlot*

The *rxHistogram* function will show us the distribution of any of the variables in our data set. For example, let's look at credit score:

```
rxHistogram(~creditScore, data = mortDataNew )
```

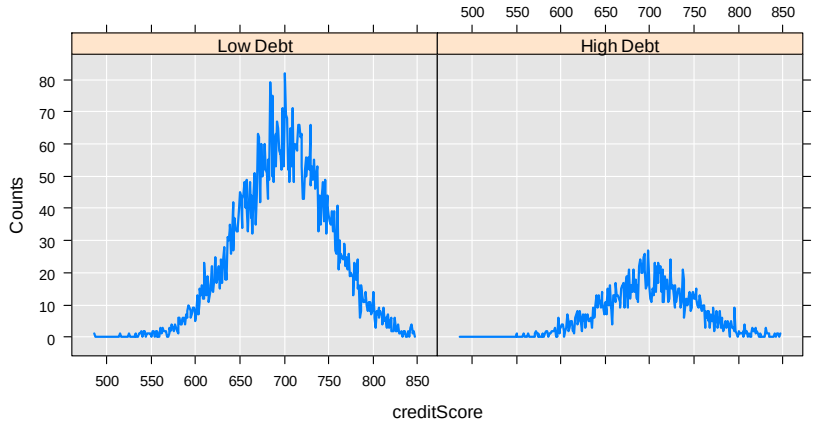


The *rxCube* function will compute category counts, and can operate on the interaction of categorical variables. Using the *F()* notation to convert a variable into an on-the-fly categorical factor variable (with a level for each integer value), we can compute the counts for each credit score for the two groups who have low and high credit card debt:

```
mortCube <- rxCube(~F(creditScore):catDebt, data = mortDataNew)
```

The *rxLinePlot* function is a convenient way to plot output from *rxCube*. We use the *rxResultsDF* helper function to convert cube output into a data frame convenient for plotting:

```
rxLinePlot(Counts~creditScore|catDebt, data=rxResultsDF(mortCube))
```



6.5 Step 5: Analyzing Your Data with *rxLogit*

RevoScaleR provides the foundation for a variety of high performance, scalable data analyses. Here we'll do a logistic regression, but you'll probably also want to take look at computing summary statistics (*rxSummary*), computing cross-tabs (*rxCrossTabs*), estimating linear models (*rxLinMod*) or generalized linear models (*rxGlm*), and estimating variance-covariance or correlation matrices (*rxCovCor*) that can be used as inputs to other R functions such as principal components analysis and factor analysis. Now, let's estimate a logistic regression on whether or not an individual defaulted on their loan, using credit card debt and years of employment as independent variables:

```
myLogit <- rxLogit(default~ccDebt+yearsEmploy , data=mortDataNew)
summary(myLogit)
```

We get the following output:

```
Call:
rxLogit(formula = default ~ ccDebt + yearsEmploy, data = mortDataNew)

Logistic Regression Results for: default ~ ccDebt + yearsEmploy
Data: mortDataNew
Dependent variable(s): default
Total independent variables: 3
Number of valid observations: 9982
Number of missing observations: 0
-2*LogLikelihood: 100.6036 (Residual deviance on 9979 degrees of freedom)

Coefficients:
             Estimate Std. Error z value Pr(>|z|)
(Intercept) -1.614e+01  2.074e+00  -7.781 2.22e-16 ***
ccDebt       1.414e-03  2.139e-04   6.610 3.83e-11 ***
yearsEmploy -3.317e-01  1.608e-01  -2.063   0.0391 *
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Condition number of final variance-covariance matrix: 1.4455
Number of iterations: 9
```

6.6 Step 6: Scaling Your Analysis

So, we've finished experimenting with our small data set in memory. Let's scale up to a data set with a million rows rather than just 10000. These larger text data files are available online. (Windows users should download the zip version, mortDefault.zip, and Linux users mortDefault.tar.gz). After downloading and unpacking the data, set your path to the correct location in the code below. It will be more efficient to store the imported data on disk, so we'll also specify the locations for our imported and transformed data sets:

```
# bigDataDir <- "C:/Revolution/Data" # Specify the location
inDataFile <- file.path(bigDataDir, "mortDefault",
    "mortDefault2000.csv")
outFile <- "myMortData.xdf"
outFile2 <- "myMortData2.xdf"
```

That's it! Now you can re-use all of the importing, data step, plotting and analysis code above on the larger data set.

```
# Import data
mortData <-    rxImport(inData = inDataFile, outFile = outFile)
```

```
Rows Read: 500000, Total Rows Processed: 500000, Total Chunk Time: 1.043
seconds
Rows Read: 500000, Total Rows Processed: 1000000, Total Chunk Time: 1.001
seconds
```

Note that because we have specified an output file when importing the data, the returned *mortData* object is a small object in memory representing the .xdf data file, rather than a full data frame containing all of the data in memory. It can be used in ***RevoScaleR*** analysis functions in the same way as data frames.

```
# Some quick information about my data
rxGetInfo(mortData, getVarInfo = TRUE, numRows=5)
```

```
File name: C:\Revolution\Data\myMortData.xdf
Number of observations: 1e+06
Number of variables: 6
Number of blocks: 2
Compression type: zlib
Variable information:
Var 1: creditScore, Type: integer, Low/High: (459, 942)
Var 2: houseAge, Type: integer, Low/High: (0, 40)
Var 3: yearsEmploy, Type: integer, Low/High: (0, 15)
Var 4: ccDebt, Type: integer, Low/High: (0, 14639)
Var 5: year, Type: integer, Low/High: (2000, 2000)
Var 6: default, Type: integer, Low/High: (0, 1)
Data (5 rows starting with row 1):
  creditScore houseAge yearsEmploy ccDebt year default
1         615       10           5   2818 2000       0
2         780       34           5   3575 2000       0
3         735       12           1   3184 2000       0
4         713       15           5   6236 2000       0
```

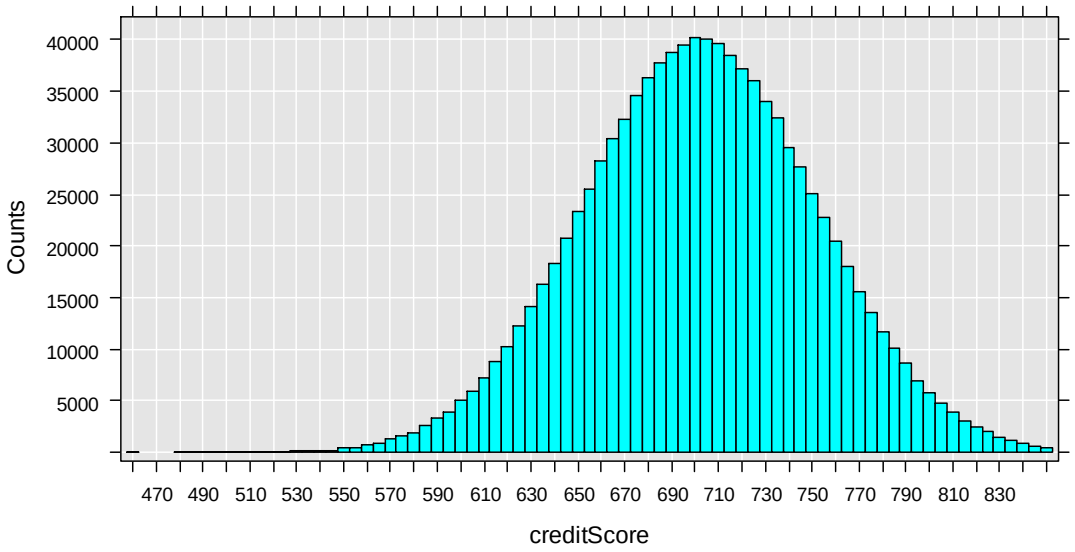
```
  5          689        10            5    6817 2000        0
```

```
# The data step
mortDataNew <- rxDataStep(
     # Specify the input data set
     inData = mortData,
     # Put in a placeholder for an output file
     outFile = outFile2,
     # Specify any variables to keep or drop
     varsToDrop = c("year"),
     # Specify rows to select
     rowSelection = creditScore < 850,
     # Specify a list of new variables to create
     transforms = list(
         catDebt = cut(ccDebt, breaks = c(0, 6500, 13000),
             labels = c("Low Debt", "High Debt")),
         lowScore = creditScore < 625))
```

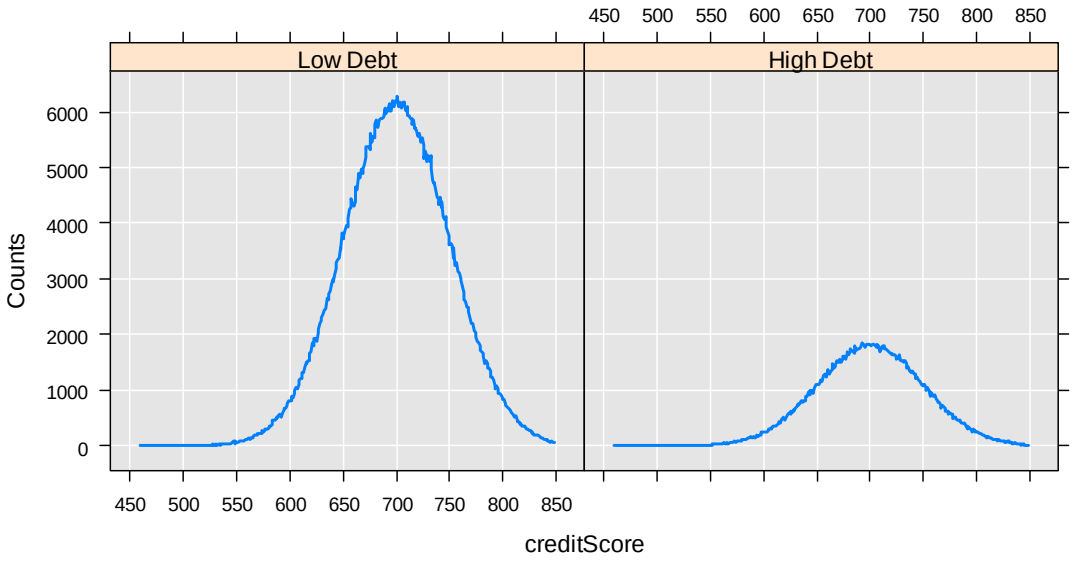
```
 Rows Read: 500000, Total Rows Processed: 500000, Total Chunk Time: 0.673
 seconds
 Rows Read: 500000, Total Rows Processed: 1000000, Total Chunk Time: 0.448
 seconds
 >
```

```
# Looking at the data
rxHistogram(~creditScore, data = mortDataNew )
```

```
 Rows Read: 499294, Total Rows Processed: 499294, Total Chunk Time: 0.329
 seconds
 Rows Read: 499293, Total Rows Processed: 998587, Total Chunk Time: 0.335
 seconds
 Computation time: 0.678 seconds.
```




```
myCube = rxCube(~F(creditScore):catDebt, data = mortDataNew)
rxLinePlot(Counts~creditScore|catDebt, data=rxResultsDF(myCube))
```




```
# Compute a logistic regression
myLogit <- rxLogit(default~ccDebt+yearsEmploy , data=mortDataNew)
summary(myLogit)
```

```
Call:
rxLogit(formula = default ~ ccDebt + yearsEmploy, data = mortDataNew)

Logistic Regression Results for: default ~ ccDebt + yearsEmploy
File name:
    C:\Users\RUser\myMortData2.xdf
Dependent variable(s): default
Total independent variables: 3
Number of valid observations: 998587
Number of missing observations: 0
-2*LogLikelihood: 8837.7644 (Residual deviance on 998584 degrees of freedom)

Coefficients:
              Estimate Std. Error z value Pr(>|z|)
(Intercept) -1.725e+01  2.330e-01  -74.04 2.22e-16 ***
ccDebt       1.509e-03  2.327e-05   64.85 2.22e-16 ***
yearsEmploy -3.054e-01  1.713e-02  -17.83 2.22e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Condition number of final variance-covariance matrix: 1.3005
Number of iterations: 9
```

Chapter 7.

Next Steps: A Roadmap to Documentation

Having completed the ***Microsoft R Services*** tutorials, you are ready to dive right in and start using R for your own purposes. While the tutorials have given you the basic tools to begin exploring, you may still want more guidance for your specific tasks. Luckily, there is a huge library of Microsoft R Services, R, and S documentation that can help you perform almost any task with R. This brief roadmap points you toward some of the most useful documentation that Microsoft is aware of. (If you find other useful resources, drop us a line at revodoc@microsoft.com!)

The obvious place to start is with the rest of the ***Microsoft R Services*** document set, which includes documentation on the ***R Productivity Environment*** (on Windows) and the ***RevoScaleR*** package for scalable data analysis (on all platforms):

- *Revolution R Enterprise R Productivity Environment Getting Started Guide* (RevoRPE_Getting_Started.pdf). A tutorial introduction to the RPE, which provides an integrated development environment together with a console interface to the R interpreter and features interactive debugging of your R code.
- *Revolution R Enterprise R Productivity Environment User's Guide* (RevoRPE_Users_Guide.pdf). A more detailed explanation of using the RPE.

- *RevoScaleR Getting Started Guide* (RevoScaleR_Getting_Started.pdf). A tutorial introduction to RevoScaleR, providing extended examples of using RevoScaleR to analyze huge data sets via parallel external memory algorithms.
- *RevoScaleR User's Guide* (RevoScaleR_Users_Guide.pdf). A more detailed explanation of the features of RevoScaleR, including data manipulation, linear models, logistic regression, generalized linear models, a contingency table analysis, and decision trees and forests.
- *RevoScaleR Distributed Computing Guide* (RevoScaleR_Distributed_Computing.pdf). A tutorial introduction to RevoScaleR's distributed computing features (currently supported on Windows HPC Server, Hadoop, and Teradata).
- *RevoScaleR ODBC Import Guide* (RevoScaleR_ODBC.pdf). Additional information on accessing ODBC data from RevoScaleR.
- *RevoScaleR HPC Server Getting Started Guide* (RevoScaleR_HPC_Server_Getting_Started.pdf) A tutorial introduction using RevoScaleR's distributed computing features on Microsoft HPC Server.
- *RevoScaleR Hadoop Getting Started Guide* (RevoScaleR_Hadoop_Getting_Started.pdf). A tutorial introduction using RevoScaleR's distributed computing features on Hadoop.
- *RevoScaleR Teradata Getting Started Guide* (RevoScaleR_Teradata_Getting_Started.pdf). A tutorial introduction using RevoScaleR's high-speed Teradata connection and distributed computing features in Teradata.
- If you are planning to integrate R analysis with a third party package, there is also a complete set of RevoDeployR documentation: *DeployR Administration Console Guide* (DeployR_Administration_Console_Guide.pdf)*, DeployR Deployment Planning Guide* (DeployR_Deployment_Planning_Guide.pdf), *DeployR Enterprise Security Guide* (DeployR_Enterprise_Security_Guide.pdf), *DeployR Migration Guide* (DeployR_Migration_Guide.pdf), and *DeployR Overview Guide* (DeployR_Overview_Guide.pdf).

Next, you should be aware of the R Core Team manuals, included with every R distribution:

- *An Introduction to R* (R-intro.pdf). A more complete tutorial introduction to R, this manual gives an introduction to both the language itself and its use for statistics and graphics.
- *The R Language Definition (Draft)* (R-lang.pdf). This still-incomplete manual describes the R programming language, giving detailed information on parsing, evaluation, and other specifics of the language itself.
- *Writing R Extensions* (R-exts.pdf). This manual describes how to extend R by creating your own packages.
- *R Data Import/Export* (R-data.pdf). This manual describes the various external data formats R can read from and write to, either using base R functions or additional packages.
- *R Installation and Administration* (R-admin.pdf). This manual describes how to build and install R from source code; Revolution R users should not need to refer to this manual, but may find it of interest.

- *R Internals* (R-int.pdf). This manual is a guide to R's internal structures, and gives coding standard for the R core team.
- *R Reference Index* (fullrefman.pdf). A compilation of the help files of the R standard and recommended packages, ready to print.

Beyond the standard R manuals, there are many books available to help you learn R, and to help you use R to do particular things. The rest of this chapter will help point you in the right direction. If you are looking for...

- **Introductory material.** See Section 7.1.
- **Information on analyzing data with R.** See Section 7.2.
- **Information on programming with R.** See Section 7.3.
- **Information on getting data into and out of R.** See Section 7.4.
- **Information on creating graphics with R.** See Section 7.5.
- **Information on parallel programming with R.** See Section 7.6.

7.1 Introductory Material

R for Dummies by Andrie de Vries and Joris Meys (de Vries & Meys, 2012) is an excellent starting place for those new to R, filled with examples and tips. O'Reilly's *R Cookbook*, by Paul Teetor (Teetor, 2011), is just what it claims to be–a book filled with recipes to help you accomplish very specific tasks. The *Essential R Reference* by Mark Gardener (Gardener, 2013) provides a dictionary-like reference to more than 400 R commands, including cross-references and examples.

As mentioned above, you can also continue your tutorial with the R Core Group's *An Introduction to R* (R Deveopment Core Team, 2008). Based on "Notes on S-PLUS" by Bill Venables and David Smith, this has been the jumping off point for R documentation for years. It includes an extensive sample session as its Appendix A.

The original S documentation consisting of the "Blue Book" (*The New S Language* by Rick Becker, John Chambers, and Allan Wilks (Becker, Chambers, & Wilks, 1988)), the "White Book" (*Statistical Models in S*, edited by John Chambers and Trevor Hastie (Chambers & Hastie, Statistical Models in S, 1992), and the "Green Book" (Chambers, Programming with Data: A Guide to the S Language, 1998) contains much introductory material that is still useful in today's R. However, there are some S functions that either are not available or work differently in R, and many of the example data sets are not available in R, so typing along with the examples is not always possible.

Another very useful document that all R users should read is the R FAQ by Kurt Hornik (http://cran.r-project.org/doc/FAQ/R-FAQ.html).

Users of SAS or SPSS who are new to R might usefully start with Robert Muenchen's *R for SAS and SPSS Users* (Muenchen, 2009). Two other books that address both the SAS and R user communities are *SAS and R: Data Management, Statistical Analysis, and Graphics* by Ken Kleinman and Nicholas J. Horton (Kleinman & Horton, 2010) and *Analysis of Correlated Data with SAS and R* by Mohamed M. Shoukri and Mohammad A. Chaudhary (Shoukri & Chaudhary, 2007).

7.2 Information on Data Analysis and Statistics

A good source of information on introductory data analysis and statistics is Peter Dalgaard's *Introductory Statistics with R* (Dalgaard, 2002). After a chapter on basic R operations, Dalgaard discusses probability and distributions, descriptive statistics and graphics, one- and two-sample tests, regression and correlation, ANOVA and Kruskal-Wallis, tabular data, power and computation of sample size, multiple regression, linear models, logistic regression, and survival.

For more advanced techniques, the obvious starting point is *Modern Applied Statistics with S* by Bill Venables and Brian Ripley (Venables & Ripley, Modern Applied Statistics with S, 2002). This book starts with four introductory chapters on R, then gets into statistics from univariate statistics (chapter 5) to optimization (chapter 16). Along the way, the authors touch on many widely used techniques, including linear models, generalized linear models, clustering, tree-based methods, survival analysis, and many others.

Rapidly becoming *the* book for aspiring data scientists is *The Elements of Statistical Learning* by Trevor Hastie, Robert Tibshirani, and Jerome Friedman (Hastie, Tibshirani, & Friedman, 2009). This book covers a variety of statistical techniques important in big data analysis and machine learning, including various tree-based methods, support vector machines, graphical models, and more.

Linear models, generalized linear models, and other regression techniques are the subject of a number of texts, including Frank Harrell's *Regression Modeling Strategies* (Harrell, 2001), John Fox's *Applied Regression Analysis and Generalized Linear Models* (Fox, Applied Regression Analysis and Generalized Linear Models, 2008) and his R-specific companion volume, *An R and S-PLUS Companion to Applied Regression* (Fox, An R and S-PLUS Companion to Applied Regression, 2002), and *Data Analysis Using Regression and Multilevel/Hierarchical Models* by Andrew Gelman and Jennifer Hill (Gelman & Hill, 2007).

Other useful books that take you into more advanced statistics are *R in Action* by Robert I. Kabacoff (Kabacoff, 2011), *A Handbook of Statistical Analyses Using R* by Brian Everitt and Torsten Hothorn (Everitt & Hothorn, 2006), *Data Analysis and Graphics Using R* by John Maindonald and

John Braun (Maindonald & Braun, 2007), and *The R Book* by Michael Crawley (Crawley, The R Book, 2013).

If you are interested in an overview of the multiple uses of big data analytics, the book *Big Data, Big Analytics* by Michael Minelli, Michele Chambers, and Ambiga Dhiraj (Minelli, Chambers, & Dhiraj, 2013) will give you a an excellent start in understanding what big data is and how it is used in real-world business applications.

7.3 Information on Programming with R

The newest book from John M. Chambers, *Software for Data Analysis: Programming with R* (Chambers, Software for Data Analysis: Programming with R., 2008), gives a thorough description of programming in R, including tips on debugging, writing packages, creating classes and methods, and interfacing to code in other languages. It also includes a useful chapter describing how R works.

R in a Nutshell by Joseph Adler (Adler, 2010) is unlike most books on R in that it deals with R first and foremost as a programming language; it does touch on statistical topics, but that is not its main focus.

The book *S Programming* by Venables and Ripley (Venables & Ripley, S Programming, 1999) is a concise, readable guide to programming in the S family of languages. Most of their advice remains valid, but the book was published when R was still at a pre-release version (0.90.1),so some details have changed over time.

The Blue Book, White Book, and Green Book all have one or more chapters devoted to programming in S, with different points of emphasis. The Blue Book focuses on basic function writing. The White Book describes the S Version 3 class system and how to define classes, generic functions, and methods in that system. The Green Book describes the S Version 4 class system and how to define classes and methods in that system.

The manual *Writing R Extensions* by the R Core Team describes how to write complete R packages, including documentation.

7.4 Information on Getting Data Into and Out of R

The manual *R Data Import/Export* by the R Core Team describes how to read data into R from a variety of sources using both built-in R tools and additional packages. The book Data Manipulation with R by Phil Spector (Spector, 2008) includes information on reading and writing data, and also further manipulation within R. And, of course, be sure to look at the

RevoScaleR User's Guide for information on data import and export capabilities provided by ***RevoScaleR***.

7.5 Information on Creating Graphics with R

All of the references mentioned up to now contain at least some material on graphics, because graphical exploration is a primary motivation for using R in the first place. The Blue Book, in particular, describes in detail the "traditional S graphics" framework.

A popular graphics package that is rapidly growing its own complete package ecosystem is Hadley Wickham's ggplot2 package, documented in Wickham's *ggplot2: Elegant Graphics for Data Analysis* (Wickham, 2009). The ggplot2 package implements in R many of the ideas from Leland Wilkinson's *The Grammar of Graphics* (Wilkinson, 2005).

The ggplot2 package is a high-level graphics package. For lower-level graphics functionality, the definitive reference is Paul Murrell's *R Graphics* (Murrell, 2006), which describes both the traditional S graphics framework (in particular, its implementation in R by Ross Ihaka) and the grid graphics framework developed by Murrell. It also describes the lattice system, developed by Deepayan Sarkar, that uses the grid framework to implement the Trellis graphics system developed by Rick Becker and Bill Cleveland. Serious users of the lattice system will also want to consult Sarkar's book, *Lattice: Multivariate Data Visualization with R* (Sarkar, 2008).

Trellis graphics are discussed thoroughly in Cleveland's *Visualizing Data* (Cleveland, Visualizing Data, 1993). Cleveland's earlier book, now its second edition, *The Elements of Graphing Data* (Cleveland, The Elements of Graphing Data, 1994) remains essential reading for anyone interested in data visualization.

Interactive and Dynamic Graphics for Data Analysis by Dianne Cook and Deborah Swayne (Cook & Swayne, 2008) describes using R together with the GGobi visualization program for dynamic graphics.

7.6 Information on Parallel Programming in R

While the *RevoScaleR Distributed Computing Guide* focuses on running computations over multiple nodes, it also includes basic examples of parallel programming that can be run on a multiple-core workstation. *Parallel R* by Q. Ethan McCallum and Stephen Weston (McCallum & Weston, 2011) describes some additional tools for parallel computing in R.

Chapter 8.

Optimized Math Libraries

One feature of Microsoft R Services is its inclusion of optimized libraries for linear algebra. These libraries are used throughout R's modeling applications, including linear models, principal components analysis, and others.

Matrix multiplication, eigenvalue calculations, and singular value decompositions are significantly faster using these optimized libraries. For example, we ran the following computations, first with R built from source using the standard R BLAS, then with Microsoft R Services built with the optimized math libraries:

```
set.seed(14)
x <- matrix(rnorm(1000000),nrow=1000)
xout <- numeric(20)
for (i in 1:20) xout[i] <- system.time(eigen(x))[3]
xout2 <- numeric(20)
for (i in 1:20) xout2[i] <- system.time(svd(x))[3]
xout3 <- numeric(20)
for (i in 1:20) xout3[i] <- system.time(qr(x))[3]
xout4 <- numeric(20)
for (i in 1:20) xout4[i] <- system.time(lm(x[,i]~x[,-i]))[3]
xout5 <- numeric(20)
for (i in 1:20) xout5[i] <- system.time(t(x)%*%x)[3]
```

The mean times for each calculation are shown below. Matrix multiplication, eigenvalue, and singular value decomposition calculations show the greatest speedups.

Calculation	Reference Libraries	Optimized Libraries
eigen	11.45350	3.63485
svd	7.34740	1.19455
gr	1.16765	0.93600
lm	1.47795	1.23430
Matrix multiplication	1.75610	0.09604

8.1 A Note on R Numerics

Most mathematical computations in R are performed using binary double-precision floating-point numbers. Arithmetic performed using these numbers is called *floating-point arithmetic*. In floating-point arithmetic, numbers are stored with finite precision according to internationally-recognized standards established by the IEEE. There are only finitely many floating-point numbers. In particular, there is a largest (and smallest) floating-point number. There is also a smallest nonnegative floating-point number. Consider, for example, the following R statements:

```
10^308 * 10
```

```
[1] Inf
```

```
2^(-1074) / 2
```

```
[1] 0
```

illustrating that R (and floating-point arithmetic) has a somewhat limited sense of the infinite and the minutely small. When one or more terms in a computation involve the special floating-point value `inf`, the computation is said to *overflow*. Similarly, *underflow* occurs when a number too small in magnitude to be represented is encountered.

The distance between the floating-point number 1 and the next-largest floating-point number is typically called *machine epsilon*, or just `eps` for short. We can compute `eps` with a simple R program:

```
eps <- 1
while ((1 + eps/2) != 1) { eps <- eps/2 }
eps
```

```
[1] 2.220446e-16
```

In a relative sense `eps` is as large as the gaps between floating-point numbers get.

In between their largest and smallest values, floating-point numbers generally only approximate real numbers. For example, although 1 and 10 *are* floating-point numbers, the

rational number 1/10 has no exact binary floating-point representation. It is *very* closely approximated by a nearby floating-point number, but not close enough that arithmetic operations are always unaffected. Consider the following somewhat counter-intuitive R example:

```
(11/10 - 1)*10-1

 [1] 8.881784e-16
```

Yet,

```
(11/10)*10 - 1*10-1

 [1] 0
```

This example shows that floating-point arithmetic does not always obey the usual algebraic laws; here the distributive law is violated. The following example shows a situation in which the associative law is violated:

```
x <- rnorm(100000)
sum(x) - sum(sort(x))

 [1] -8.526513e-14
```

The violation of associativity results from the accumulation many small approximations over the summation. Re-ordering the computation can affect the result.

For a good introduction to the pitfalls involving numeric computation, see the excellent article by Goldberg (Goldberg, 1991).

8.2 Performance Optimization and Numerics

Many of the most widely-used and compute-intensive linear algebra and arithmetic operations performed by R are computed by low-level numerics libraries, including the basic linear algebra subroutine (BLAS) library. Revolution R includes highly-tuned low-level numerics libraries that are optimized for speed on a wide variety of x86, x86-64 and IA-64 processor architectures.

Many of the performance optimizations try to

- Make use of available SIMD-style vector registers and operations
- Make efficient re-use of speedy processor memory caches.

Both approaches often require blocking or otherwise re-ordering computations, for example to fit in a small cache.

Because floating-point arithmetic always involves approximation, and the exact nature of the approximation depends upon the particular algorithms used, it is possible that the results from

the highly-tuned numerics available in Microsoft R Services differ from results computed by other implementations of R, just as results can differ across system architectures. These computational differences typically manifest themselves on the order of machine epsilon.

Some high-performance numerics routines, including those that use x86 SIMD vector instructions (SSE, etc.), require that their input data be loaded into memory addresses divisible by 16 bytes. Unfortunately, R does not presently guarantee alignment of data on 16-byte boundaries. Therefore, it is possible that, depending on data alignment, parts of some computations may be grouped off differently (for less efficient computation). This alignment-dependent blocking of some computations can also result in differences from run to run on the order of machine epsilon.

Chapter 9.

R Memory Limits in Windows

This chapter discusses several tools available in Windows versions of R to help you monitor and manage memory usage.

9.1 64-bit R Memory Limits

The primary advantage 64-bit architectures bring to R is an increase in the amount of memory available to a given R process. The first benefit of that increase is an increase in the size of data objects you can create. For example, on most 32-bit versions of R, the largest data object you can create is roughly 3GB; attempts to create 4GB objects result in errors with the message "cannot allocate vector of length *xxxx*." On 64-bit versions of R, you can generally create larger data objects. From R 3.0.0 onward, the old limitation of $2^{31} - 1$ elements in a vector (about 2 billion elements) has been removed (except for character vectors).

The functions `memory.size` and `memory.limit` help you manage the memory used by Windows versions of R. In 64-bit Microsoft R Services, R sets the memory limit by default to the amount of physical RAM minus half a gigabyte, so that, for example, on a machine with 8GB of RAM, the default memory limit is 7.5GB:

```
memory.limit()
```

```
[1] 7678.328
```

You can also use *memory.limit* to increase the amount of memory used by R, but not decrease it:

```
memory.limit(size = 15000)

NULL
```

Objects larger than physical memory can be allocated and used, as long as the total memory used does not exceed *memory.limit()*. However, using such large objects may be unacceptably slow.

The *memory.size* function returns the amount of memory currently being used by R, or, if you specify *max=TRUE*, the maximum amount of memory used in the current R session:

```
memory.size()

 [1] 8207.29

memory.size(max=TRUE)

 [1] 8214.75
```

The result is in MB.

Another useful function for managing memory, which is available on all platforms and not just Windows, is gc. The gc function causes R to perform garbage collection and report the current state of memory usage. The report gives information on the number of "Ncells" and "Vcells" currently in use; very roughly, the "Ncells" can be thought of the memory used by the language itself and "Vcells" the memory used for the data. The "Ncells" occupy 28 bytes on a 32-bit platform and typically 56 bytes on a 64-bit platform. "Vcells" occupy 8 bytes on all platforms. Typical output from gc is as follows:

```
> gc()
         used (Mb) gc trigger    (Mb) limit (Mb)  max used    (Mb)
Ncells 137107  7.4     350000   18.7     229376    350000   18.7
Vcells 142395  1.1  451286991 3443.1      32768 537014530 4097.1
```

9.2 Using the Memory Tools Together

The following function combines *memory.limit*, *memory.size*, and *gc* into a single function with friendly formatting; it can be used only on Windows platforms

```
reportMem <- function()
{
   # Perform garbage collection
   gc(verbose=FALSE)
   #How much memory is being used by malloc
   intvar <- round(memory.size())
   cat("Memory currently allocated is", intvar, "Mb\n")
```

```
   #Maximum amount of memory that has been obtained from the OS
   intvar <- round(memory.size(max=TRUE))
   cat("Maximum amount of memory allocated during this session is",
     intvar, "Mb\n")

   #Memory limit
   intvar <- round(memory.limit())
   cat("Current limit for total allocation is", intvar, "Mb\n")
}
```

The *reportObjSize* function defined below is a simple wrapper for *object.size* that prints the size of an object in MB:

```
reportObjSize <- function(x=x)
{
   bm_conv <- 1024*1024
   objsize <- round(object.size(x)/bm_conv)
   cat("Object Size =", objsize, "Mb", "\n")
}
```

Together, we can use these functions to show how memory use changes as we allocate vectors of different sizes. (We used a 64-bit Windows machine for our examples; you can do similar things on 32-bit Windows, but obviously with smaller objects sizes.) For example, here we create a vector of size roughly 1GB, and see its size and effect on the memory profile:

```
reportMem()


 Memory currently allocated is 21 Mb
 Maximum amount of memory allocated during this session is 23 Mb
 Current limit for total allocation is 7678 Mb

xlen1gb <- 2^27
x <- rnorm(xlen1gb)
reportMem()

 Memory currently allocated is 1039 Mb
 Maximum amount of memory allocated during this session is 1047 Mb
 Current limit for total allocation is 7678 Mb
```

To delete an object, use the *rm* function. The *reportMem* function handles the garbage collection to ensure that the deleted object's memory is in fact freed:

```
rm(x)
reportMem()

 Memory currently allocated is 15 Mb
 Maximum amount of memory allocated during this session is 1047 Mb
 Current limit for total allocation is 7678 Mb
```

To create a 5GB object, make sure that the memory limit is high enough. (Allocating a 5GB object can be done even on systems with less than 5GB of physical memory, but it can be very slow.)

```
x <- rnorm(xlen1gb*5)
reportObjSize(x)

 Object Size = 5120 Mb

reportMem()

 Memory currently allocated is 5135 Mb
 Maximum amount of memory allocated during this session is 5143 Mb
 Current limit for total allocation is 7678 Mb
```

Bibliography

Adler, J. (2010). *R in a Nutshell.* Sebastopol, CA: O'Reilly.

Becker, R. A., Chambers, J. M., & Wilks, A. R. (1988). *The New S Language: A Programming Environment for Data Analysis and Graphics.* New York: Chapman and Hall.

Chambers, J. M. (1998). *Programming with Data: A Guide to the S Language.* New York: Springer.

Chambers, J. M. (2008). *Software for Data Analysis: Programming with R.* New York: Springer.

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

Cleveland, W. S. (1993). *Visualizing Data.* Summit, New Jersey: Hobart Press.

Cleveland, W. S. (1994). *The Elements of Graphing Data* (second ed.). Summit, New Jersey: Hobart Press.

Cook, D., & Swayne, D. F. (2008). *Interactive and Dynamic Graphics for Data Analysis: With R and GGobi.* New York: Springer.

Crawley, M. J. (2013). *The R Book* (Second ed.). Chichester: John Wiley & Sons Ltd.

Dalgaard, P. (2002). *Introductory Statistics with R.* New York: Springer.

de Vries, A., & Meys, J. (2012). *R for Dummies.* Chichester: John Wiley & Sons.

Everitt, B. S., & Hothorn, T. (2006). *A Handbook of Statistical Analyses Using R.* Boca Raton, Florida: Chapman & Hall/CRC.

Fox, J. (2002). *An R and S-PLUS Companion to Applied Regression.* Thousand Oaks, CA: Sage.

Fox, J. (2008). *Applied Regression Analysis and Generalized Linear Models.* Thousand Oaks, CA: Sage.

Gardener, M. (2013). *The Essential R Reference.* Indianapolis, IN: John Wiley & Sons.

Gelman, A., & Hill, J. (2007). *Data Analysis Using Regression and Multilevel/Hierarchical Models.* New York: Cambridge University Press.

Goldberg, D. (1991). What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv., 23*(1), 5-48.

Harrell, F. E. (2001). *Regression Model Strategies: with applications to linear models, logistic regression, and survival analysis.* New York: Springer.

Hastie, T., Tibshirani, R., & Friedman, J. (2009). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction* (2nd ed.). New York: Springer.

Ihaka, R., & Gentleman, R. (1996). R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics, 5*(3), 299-314.

Kabacoff, R. I. (2011). *R in Action.* Shelter Island, NY: Manning.

Kleinman, K., & Horton, N. J. (2010). *SAS and R: Data Management, Statistical Analysis, and Graphics.* Boca Raton, FL: Chapman & Hall/CRC.

Maindonald, J., & Braun, J. (2007). *Data Analysis and Graphics Using R: An Example-based Approach* (second ed.). Cambridge: Cambridge University Press.

Matloff, N. (2011). *The Art of R Programming.* San Francisco: no starch press.

Minelli, M., Chambers, M., & Dhiraj, A. (2013). *Big Data, Big Analytics.* Hoboken, NJ: John Wiley & Sons.

Muenchen, R. A. (2009). *R for SAS and SPSS Users.* New York: Springer.

Murrell, P. (2006). *R Graphics.* Boca Raton, FL: Chapman & Hall/CRC.

R Development Core Team. (2008). *R: A Language and Environment for Statistical Computing.* Vienna: R Foundation for Statistical Computing.

R Deveopment Core Team. (2008). *An Introduction to R.* Vienna: R Foundation for Statistical Computing.

Sarkar, D. (2008). *Lattice: Multivariate Data Visualization with R.* New York: Springer.

Shoukri, M. M., & Chaudhary, M. A. (2007). *Analysis of Correlated Data with SAS and R* (third ed.). Boca Raton, FL: Chapman & Hall/CRC.

Spector, P. (2008). *Data Manipulation with R.* New York: Springer.

Teetor, P. (2011). *R Cookbook.* Sebastopol, CA: O'Reilly.

Venables, W. N., & Ripley, B. D. (1999). *S Programming.* New York: Springer.

Venables, W. N., & Ripley, B. D. (2002). *Modern Applied Statistics with S* (Fourth Edition ed.). New York: Springer.

Wickham, H. (2009). *ggplot2: Elegant Graphics for Data Analysis.* New York: Springer.

Wilkinson, L. (2005). *The Grammar of Graphics* (second ed.). New York: Springer.