InfoWorld DeepDive

LEARN TO CRUNCH BIG DATA (with) <R>

PHOTO COLLAGE BY TERRI HAAS/SHUTTERSTOCK

Deep Dive

Learn to crunch big data with R

Get started using the open source R programming language to do statistical computing and graphics on large data sets

BY MARTIN HELLER

A few years ago I was the CTO and co-founder of a startup in the medical practice management software space. One of the problems we were trying to solve was how medical office visit schedules can optimize everyone's time. Too often, office visits are scheduled to optimize the physician's time, and patients have to wait way too long in overcrowded waiting rooms in the company of people coughing contagious diseases out their lungs.

PHOTO COLLAGE BY TERRI HAAS/SHUTTERSTOCK

One of my co-founders, a hospital medical director, had a multivariate linear model that could predict the required length for an office visit based on the reason for the visit, whether the patient needs a translator, the average historical visit lengths of both doctor and patient, and other possibly relevant factors. One of the subsystems I needed to build was a monthly regression task to update all of the coefficients in the model based on historical data.

After exploring many options, I chose to implement this piece in R, taking advantage of the wide variety of statistical (linear and nonlinear modeling, classical statistical tests, time-series analysis, classification, clustering) and graphical techniques implemented in the R system.

One of the attractions for me was the R scripting language, which makes it easy to save and rerun analyses on updated data sets; another attraction was the ability to integrate R and C++. A key benefit for this project was the fact that R, unlike Excel and other GUI analysis programs, is completely auditable.

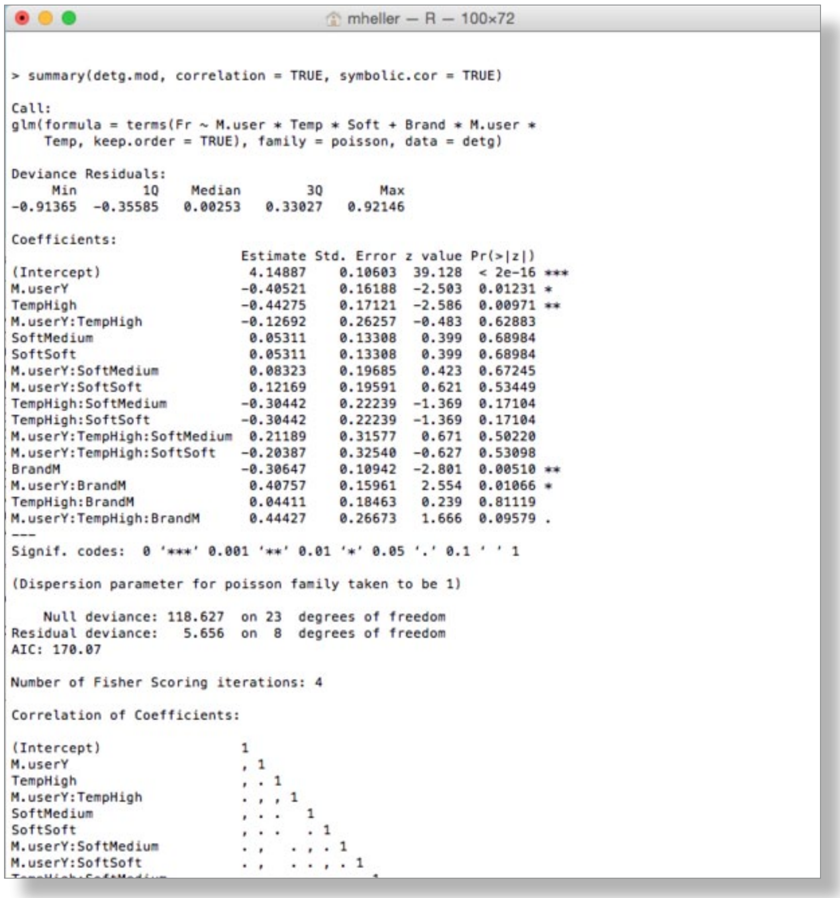
Alas, that startup ran out of money not long after I implemented a proof-of-concept Web application, at least partially because our first hospital customer had to declare Chapter 7 bankruptcy. Nevertheless, I continue to favor R for statistical analysis and data science.

Essential R scripting

Sharon Machlis of Computerworld wrote an excellent set of beginner tutorials on R for business intelligence in 2013. It would be silly for me to reinvent those six articles here, so feel free to go read them and come back. The TL;DR version is as follows.

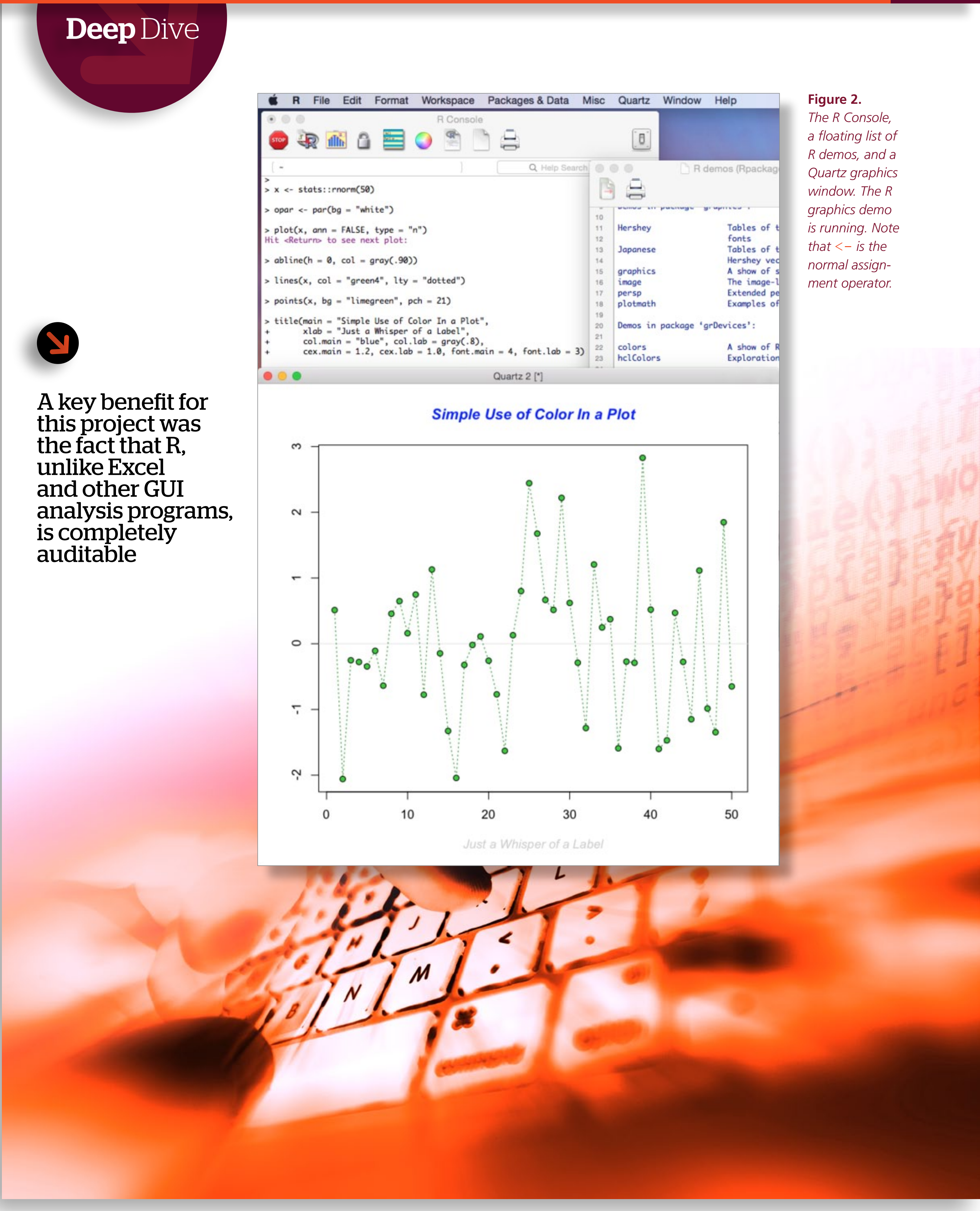
Start by installing R and RStudio on your desktop. Both are free. RStudio is optional, but I like it, and you probably will, too. There are a half-dozen other R IDEs and a dozen editors with some R support, but don't go crazy trying them all.

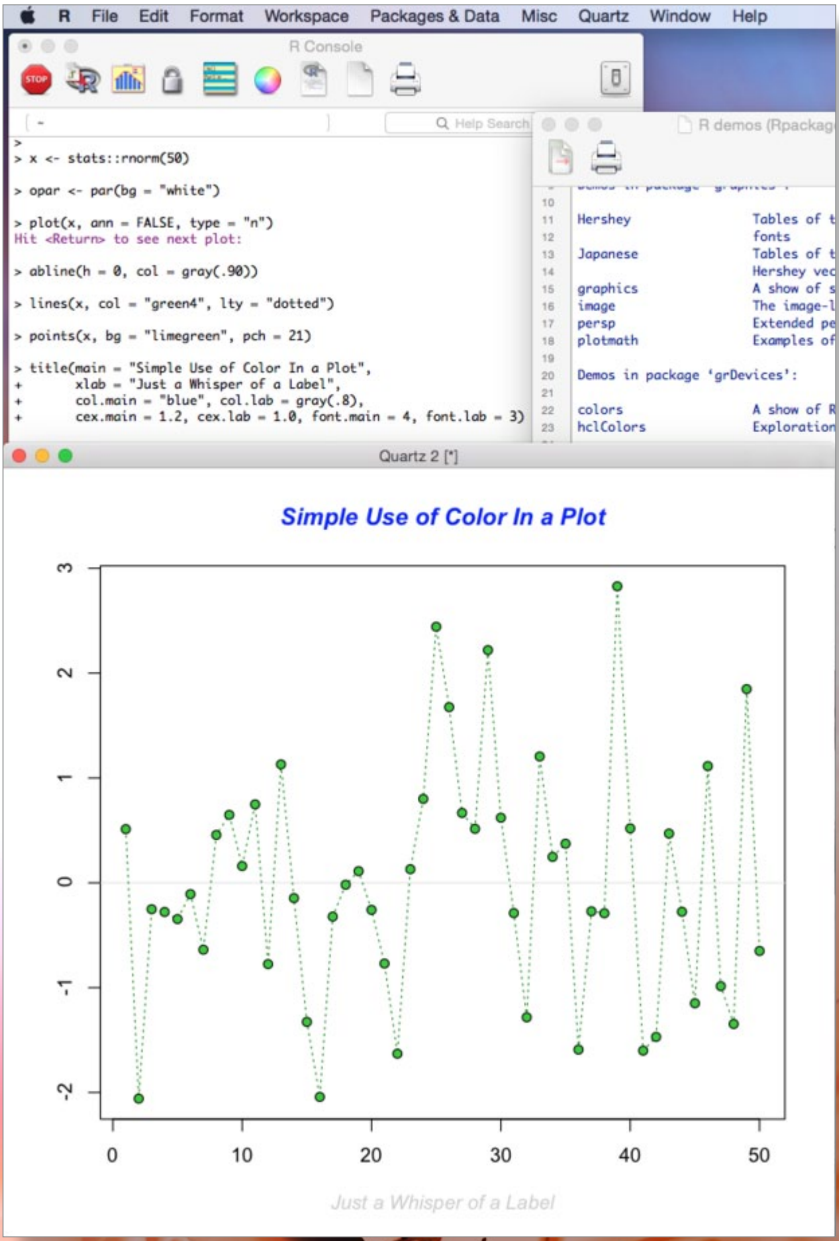
Try running R from a command shell (Figure 1), the R Console (Figure 2), and RStudio (Figure 3). Familiarize yourself with some of the R tutorials and demos.



```
> summary(detg.mod, correlation = TRUE, symbolic.cor = TRUE)

Call:
glm(formula = terms(Fr ~ M.user * Temp * Soft + Brand * M.user *
    Temp, keep.order = TRUE), family = poisson, data = detg)

Deviance Residuals:
     Min        1Q    Median        3Q       Max
-0.91365  -0.35585   0.00253   0.33027   0.92146

Coefficients:
                            Estimate Std. Error z value Pr(>|z|)
(Intercept)                  4.14887    0.10603  39.128  < 2e-16 ***
M.userY                     -0.40521    0.16188  -2.503  0.01231 *
TempHigh                    -0.44275    0.17121  -2.586  0.00971 **
M.userY:TempHigh            -0.12692    0.26257  -0.483  0.62883
SoftMedium                   0.05311    0.13308   0.399  0.68984
SoftSoft                     0.05311    0.13308   0.399  0.68984
M.userY:SoftMedium           0.08323    0.19685   0.423  0.67245
M.userY:SoftSoft             0.12169    0.19591   0.621  0.53449
TempHigh:SoftMedium         -0.30442    0.22239  -1.369  0.17104
TempHigh:SoftSoft           -0.30442    0.22239  -1.369  0.17104
M.userY:TempHigh:SoftMedium  0.21189    0.31577   0.671  0.50220
M.userY:TempHigh:SoftSoft   -0.20387    0.32540  -0.627  0.53098
BrandM                      -0.30647    0.10942  -2.801  0.00510 **
M.userY:BrandM               0.40757    0.15961   2.554  0.01066 *
TempHigh:BrandM              0.04411    0.18463   0.239  0.81119
M.userY:TempHigh:BrandM      0.44427    0.26673   1.666  0.09579 .
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for poisson family taken to be 1)

    Null deviance: 118.627  on 23  degrees of freedom
Residual deviance:   5.656  on  8  degrees of freedom
AIC: 170.07

Number of Fisher Scoring iterations: 4

Correlation of Coefficients:

(Intercept)                 1
M.userY                     , 1
TempHigh                    , . 1
M.userY:TempHigh            . , , 1
SoftMedium                  , . .   1
SoftSoft                    , . .   . 1
M.userY:SoftMedium          . ,   . , . 1
M.userY:SoftSoft            . ,   . . , . 1
```

Figure 1.
R running in a Bash shell, using the supplied glm.vr linear regression demo.





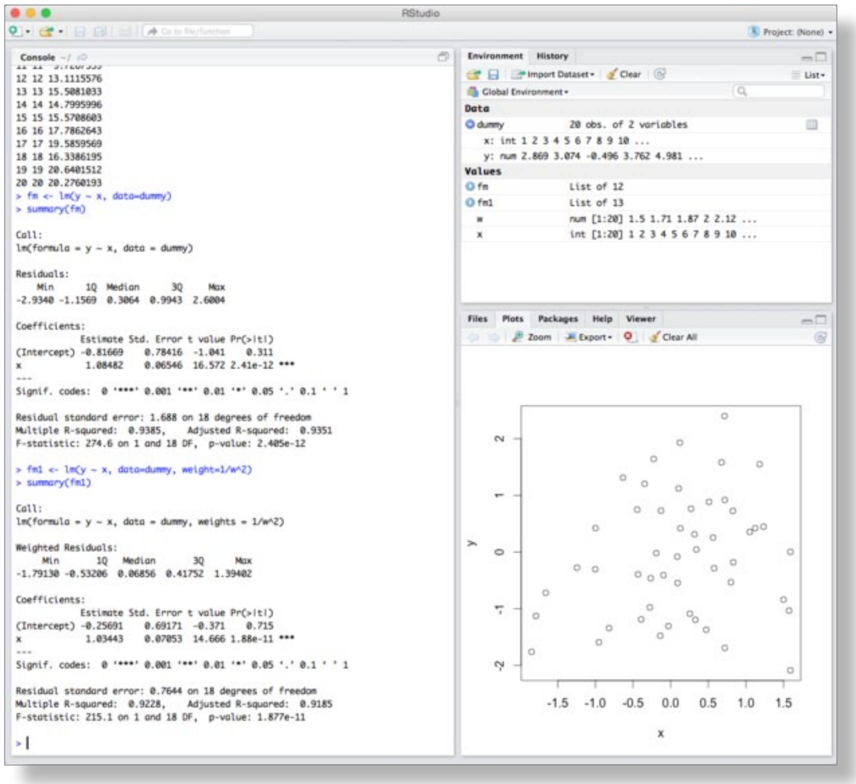
```
>
> x <- stats::rnorm(50)

> opar <- par(bg = "white")

> plot(x, ann = FALSE, type = "n")
Hit <Return> to see next plot:

> abline(h = 0, col = gray(.90))

> lines(x, col = "green4", lty = "dotted")

> points(x, bg = "limegreen", pch = 21)

> title(main = "Simple Use of Color In a Plot",
+       xlab = "Just a Whisper of a Label",
+       col.main = "blue", col.lab = gray(.8),
+       cex.main = 1.2, cex.lab = 1.0, font.main = 4, font.lab = 3)
```

```

Hershey                 Tables of t
                        fonts
Japanese                Tables of t
                        Hershey vec
graphics                A show of s
image                   The image-l
persp                   Extended pe
plotmath                Examples of

Demos in package 'grDevices':

colors                  A show of R
hclColors               Exploration
```

Figure 2. *The R Console, a floating list of R demos, and a Quartz graphics window. The R graphics demo is running. Note that <− is the normal assignment operator.*

A key benefit for this project was the fact that R, unlike Excel and other GUI analysis programs, is completely auditable

PHOTO COLLAGE BY TERRI HAAS/SHUTTERSTOCK



```
12 12 13.1115576
13 13 15.5081033
14 14 14.7995996
15 15 15.5708603
16 16 17.7862643
17 17 19.5859569
18 18 16.3386195
19 19 20.6401512
20 20 20.2760193
> fm <- lm(y ~ x, data=dummy)
> summary(fm)

Call:
lm(formula = y ~ x, data = dummy)

Residuals:
    Min      1Q  Median      3Q     Max
-2.9340 -1.1569  0.3064  0.9943  2.6004

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) -0.81669    0.78416  -1.041    0.311
x            1.08482    0.06546  16.572 2.41e-12 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 1.688 on 18 degrees of freedom
Multiple R-squared:  0.9385,    Adjusted R-squared:  0.9351
F-statistic: 274.6 on 1 and 18 DF,  p-value: 2.405e-12

> fm1 <- lm(y ~ x, data=dummy, weight=1/w^2)
> summary(fm1)

Call:
lm(formula = y ~ x, data = dummy, weights = 1/w^2)

Weighted Residuals:
     Min       1Q   Median       3Q      Max
-1.79130 -0.53206  0.06856  0.41752  1.39402

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) -0.25691    0.69171  -0.371    0.715
x            1.03443    0.07053  14.666 1.88e-11 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.7644 on 18 degrees of freedom
Multiple R-squared:  0.9228,    Adjusted R-squared:  0.9185
F-statistic: 215.1 on 1 and 18 DF,  p-value: 1.877e-11

> |
```

```
Data
dummy          20 obs. of 2 variables
  x: int 1 2 3 4 5 6 7 8 9 10 ...
  y: num 2.869 3.074 -0.496 3.762 4.981 ...
Values
fm             List of 12
fm1            List of 13
 w             num [1:20] 1.5 1.71 1.87 2 2.12 ...
 x             int [1:20] 1 2 3 4 5 6 7 8 9 10 ...
```

Figure 3. *RStudio has four windows (all but the editor are shown here) and multiple tabs in each window. Sample code from the R site is running. Notice the data display in the upper right, which is quite convenient.*

The power of R is illustrated by the deceptively simple calls in Figure 3 to do statistical analysis. For example,

```
fm1 <- lm(y ~ x, data=dummy,
weight=1/w^2)
summary(fm1)
```

This says "find the best fit coefficients, fitted values, and residuals for a linear model where `y` varies with `x` for the supplied data and weight vectors. Save them in object `fm1` and then summarize the results." Earlier in this session we had defined the following:

```
w <- 1 + sqrt(x) / 2
```

Reading this code is straightforward. Writing it takes a little study. But it isn't hard and there's lots of free help available, not to mention dozens of books.

In addition to the R help available on the Web and from the Help menu items in the R Console and RStudio, you can get help from the R command line. For example:

```
?functionName
help(functionName)
example(functionName)
args(functionName)
help.search("your search term")
??("my search term")
```

To get data into R, either use its sample data, listed by the `data()` function, or load it from a file:

```
mydata <- read.csv("filename.txt")
```

R is extremely extensible. The `library()` and `require()` functions load and attach add-on packages; `require()` is designed for use inside other functions. Many add-on packages and the R distributions live in CRAN, the worldwide Comprehensive R Archive Network. The other two common R archives are Omegahat and Bioconductor. Additional packages live in R-Forge.

The R installation copies the base packages and the recommended packages from CRAN into a local library directory, which on a Mac is currently at /Library/Frameworks/R.framework/Versions/3.1/Resources/library/. Running the R `library()` command without any arguments will list the local packages and the library location. RStudio will also generate the correct `library()` command to install a listed package when you check the installation check mark in the Packages tab. The command `help(package = packageName)` will display the functions in the specified package.

There are R packages and functions to load data from any reasonable source, not only CSV files. Beyond the obvious case of delimiters other than commas, which are handled using the `read.table()` function, you can copy and paste data tables, read Excel files, connect Excel to R, bring in SAS and SPSS data, and access databases, Salesforce, and RESTful interfaces. See, for example, the `foreign` package.

You don't really need to learn the syntax for standard data imports, as the RStudio `Tools|Import Dataset` menu item will help you generate the correct commands interactively by looking at the data from a text file or URL and setting the correct conversion options in drop-down lists based on what you see.

You can see a list of the currently available packages by name on CRAN; this list is much more extensive than the list of recommended packages downloaded to your desktop by default. To install a package from one of the default archives, use the `install.packages` function:

```
install.packages("ggplot2")
```

Note that `ggplot2` is a popular advanced graphics package that has more options than the standard `graphics` package. Nevertheless, `graphics` can do a lot. In addition to the graphics in Figures 2 and 3, consider Figures 4 and 5.

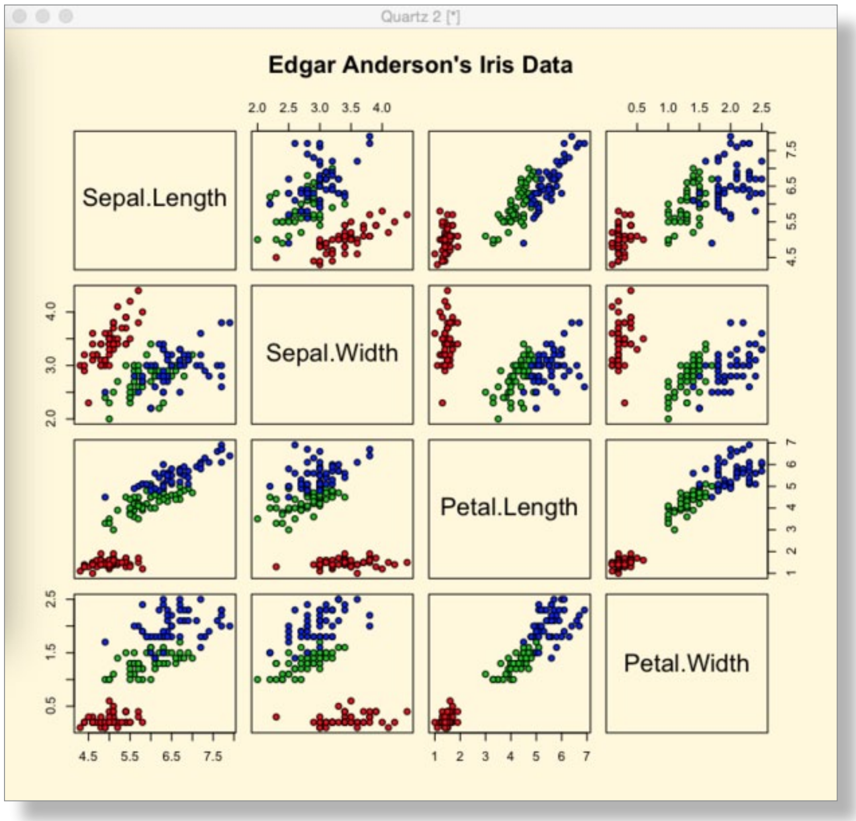


Figure 4. *Edgar Anderson's Iris data is a stock R data set, and this set of scatterplots was produced by the* `graphics` *demo.*

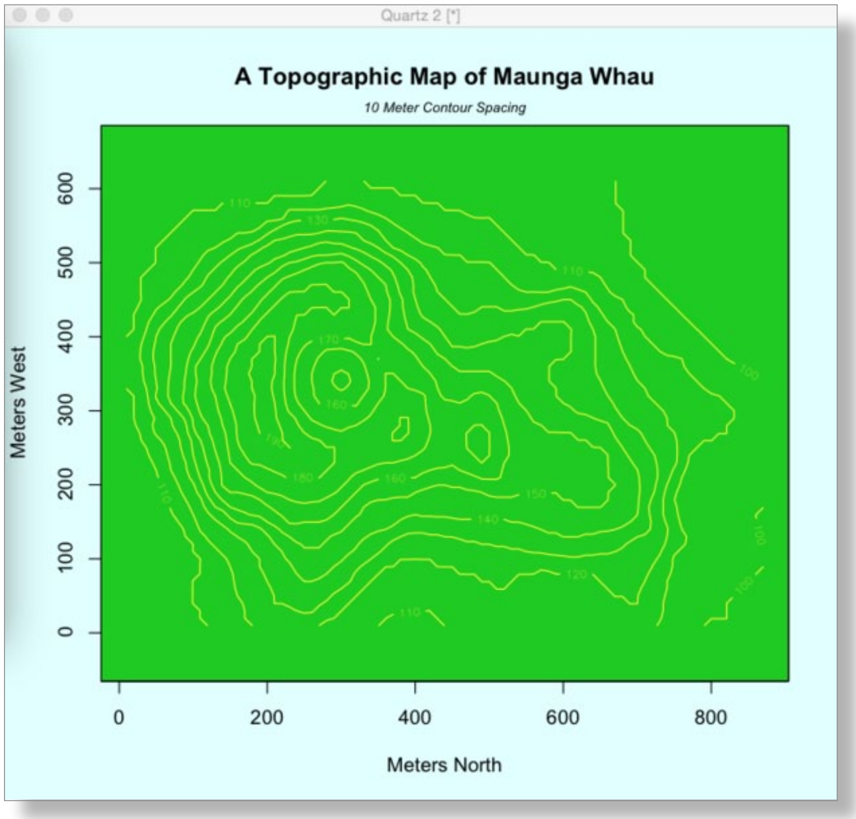


Figure 5. *This topographic map of Maunga Whau was produced from stock data and the* `graphics` *demo.)*

When R programmers talk about "big data," they don't necessarily mean data that goes through Hadoop. They generally use "big" to mean data that can't be analyzed in memory.

R can do much more in terms of graphics and statistical analysis. Do read Sharon Machlis's tutorial and follow up with her links to additional information. At this point, I want to expand my discussion to how you can analyze big data in R.

R in the cloud

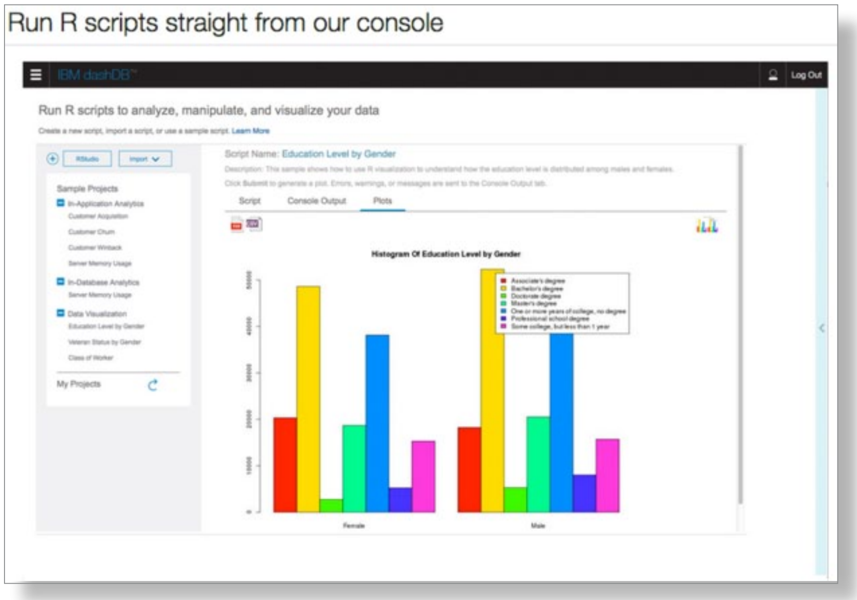
When R programmers talk about "big data," they don't necessarily mean data that goes through Hadoop. They generally use "big" to mean data that can't be analyzed in memory.

The fact is you can easily get 16GB of RAM in a desktop or laptop computer. R running in 16GB of RAM can analyze millions of rows of data with no problem. Times have changed quite a bit since the days when a database table with a million rows was considered big.

One of the first steps many developers take when their program needs more RAM is to run it on a bigger machine. You can run R on a server; a common 4U Intel server can hold up to 2TB of RAM. Of course, hogging an entire 2TB server for one personal R instance might be a bit wasteful. So people run large cloud instances for as long as they need them, run VMs on their server hardware, or run the likes of RStudio Server on their server hardware.

RStudio Server comes in Free and Pro editions. Both have the same features for individual analysts, but the Pro version offers more in the way of scale: authorization and security, management visibility, performance tuning, support, and a commercial license. According to Roger Oberg of RStudio, the company's intent is not to create paid-only features for individuals.

RStudio Server Pro is integrated with several big data systems. For example, when I was reviewing the IBM Bluemix PaaS, I noticed that R and RStudio are part of IBM's DashDB service (Figure 6). In fact, this is an installation of RStudio Server Pro on Bluemix and SoftLayer, according to Oberg and Tareef Kawaf of RStudio.


Run R scripts straight from our console

Run R scripts to analyze, manipulate, and visualize your data

Create a new script, import a script, or use a sample script. Learn More

Script Name: Education Level by Gender

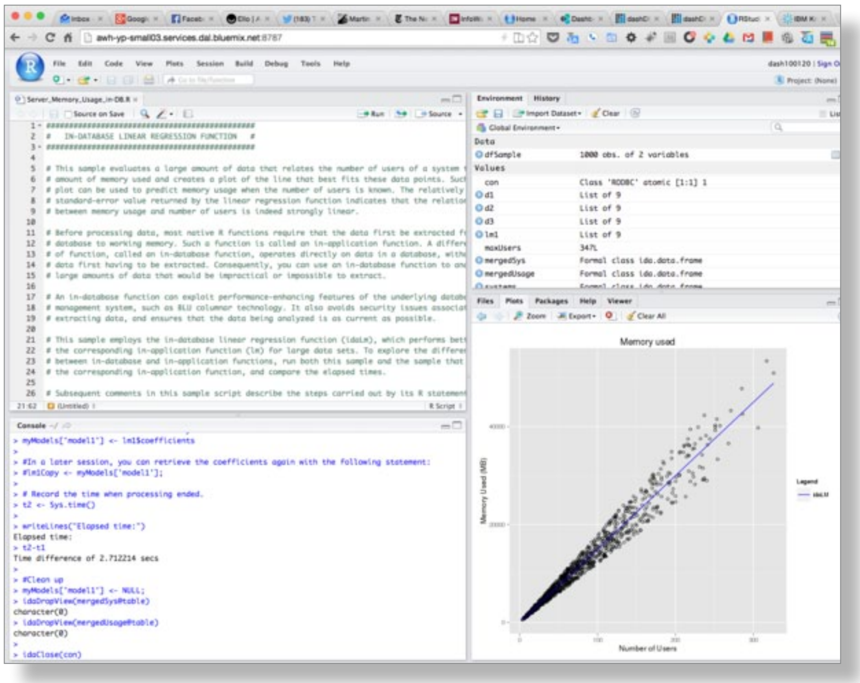
Description: This sample shows how to use R visualization to understand how the education level is distributed among males and females.

Click Submit to generate a plot. Errors, warnings, or messages are sent to the Console Output tab.

Figure 6. *IBM Bluemix documentation touts the ability to run R scripts against a DashDB in-memory database.* `graphics` *demo.)*

There is an additional strategy for running R against big data: Bring down only the data that you need to analyze. In the spirit of MapReduce, Hadoop, Spark, and Storm, you want to winnow the data as you stream it to make in-memory analysis tractable on the reduced data set. To use Kawaf's example, you may have 100TB of data but need "only" 5 columns and 20 million rows, a mere few hundred megabytes of reduced data.

You may also want to perform some of the analysis in the database instead of in the app. IBM has done a good job of providing an example, along with the R source code. Consider the analysis shown in Figure 7.


```
################################################
#   IN-DATABASE LINEAR REGRESSION FUNCTION   #
################################################

# This sample evaluates a large amount of data that relates the number of users of a system 
# amount of memory used and creates a plot of the line that best fits these data points. Suc
# plot can be used to predict memory usage when the number of users is known. The relatively
# standard-error value returned by the linear regression function indicates that the relatio
# between memory usage and number of users is indeed strongly linear.

# Before processing data, most native R functions require that the data first be extracted f
# database to working memory. Such a function is called an in-application function. A differe
# of function, called an in-database function, operates directly on data in a database, withe
# data first having to be extracted. Consequently, you can use an in-database function to an
# large amounts of data that would be impractical or impossible to extract.

# An in-database function can exploit performance-enhancing features of the underlying datab
# management system, such as BLU columnar technology. It also avoids security issues associat
# extracting data, and ensures that the data being analyzed is as current as possible.

# This sample employs the in-database linear regression function (idaLm), which performs bet
# the corresponding in-application function (lm) for large data sets. To explore the differe
# between in-database and in-application functions, run both this sample and the sample that
# the corresponding in-application function, and compare the elapsed times.

# Subsequent comments in this sample script describe the steps carried out by its R statemen
```

```
> myModels["model1"] <- lm1$coefficients
>
> #In a later session, you can retrieve the coefficients again with the following statement:
> #lm1Copy <- myModels['model1'];
>
> # Record the time when processing ended.
> t2 <- Sys.time()
>
> writeLines("Elapsed time:")
Elapsed time:
> t2-t1
Time difference of 2.712214 secs
>
> #Clean up
> myModels["model1"] <- NULL;
> idaDropView(mergedSys@table)
character(0)
> idaDropView(mergedUsage@table)
character(0)
>
> idaClose(con)
```

```
Data
dfSample        1000 obs. of 2 variables
Values
con             Class 'RODBC' atomic [1:1] 1
d1              List of 9
d2              List of 9
d3              List of 9
lm1             List of 9
maxUsers        347L
mergedSys       Formal class ida.data.frame
mergedUsage     Formal class ida.data.frame
```

Figure 7. *We are looking at RStudio Server Pro running in an IBM Bluemix dashDB service. The sample we ran did a regression from a large dataset in-database.*

Streaming the data out of the database and into R can take a significant amount of time. If you eliminate most of the network streaming, you can vastly reduce the time needed for the analysis. You'll notice that the timing for the in-database regression analysis is 2.7 seconds. The same task with the regression done in-application took 1.47 minutes—more than 30 times longer. The regression coefficients computed were exactly the same. All that changed was that one analysis did the regression where the data resided, and the other first streamed the data from the database to the R application.

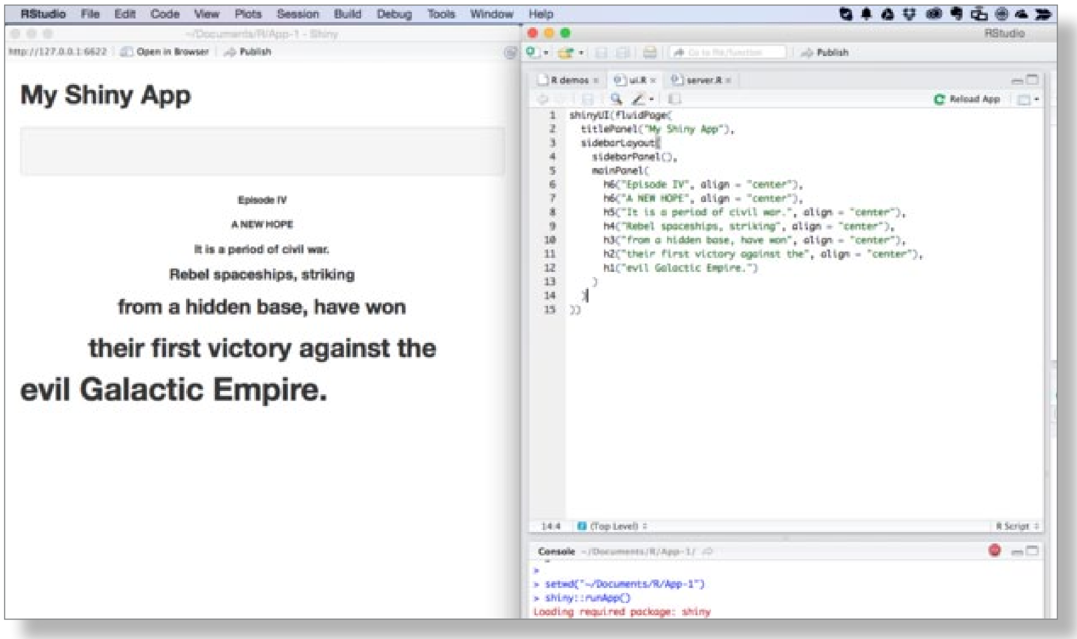
The IBM implementation is not unique; I happened to have a Bluemix account. Vertica (HP), Greenplum (Pivotal), Oracle, and Teradata all have R packages. I'm not sure how far the others have gone in the direction of in-database analytics, however.

By the way, I was pleasantly surprised to find that running RStudio Server Pro in a browser feels exactly like running RStudio on my desktop—nicely done.

Shiny and R Markdown

Of course, developers and analysts never really get away with simply writing the code and determining the results. Top management wants monthly reports, and middle management wants to play with the data without knowing anything about what's under the covers. Enter `shiny` and `rmarkdown`, two R packages from RStudio for Web applications and reporting, respectively.

Figure 8 shows a simple Shiny app running in RStudio. The code is from Lesson 2 of the Shiny tutorial.



My Shiny App

Episode IV

A NEW HOPE

It is a period of civil war.

Rebel spaceships, striking

from a hidden base, have won

their first victory against the

evil Galactic Empire.

```
shinyUI(fluidPage(
  titlePanel("My Shiny App"),
  sidebarLayout(
    sidebarPanel(),
    mainPanel(
      h6("Episode IV", align = "center"),
      h6("A NEW HOPE", align = "center"),
      h5("It is a period of civil war.", align = "center"),
      h4("Rebel spaceships, striking", align = "center"),
      h3("from a hidden base, have won", align = "center"),
      h2("their first victory against the", align = "center"),
      h1("evil Galactic Empire.")
    )
  )
))
```

```
>
> setwd("~/Documents/R/App-1")
> shiny::runApp()
Loading required package: shiny
```

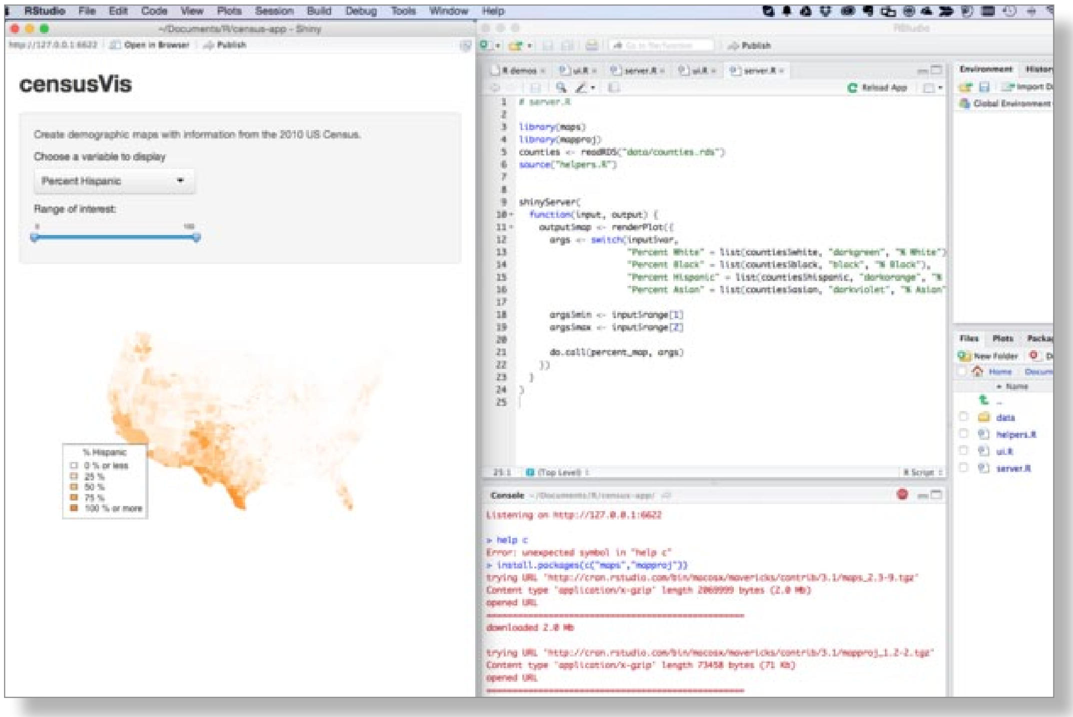
Figure 8.
We're seeing a Web page running a Shiny app, next to the RStudio editor showing the UI code for the app. The Shiny functions generate HTML. For example, `h1("My title")` *generates* `<h1>My title</h1>`.

You can use Shiny to build interactive and "reactive" Web apps, with widgets that correspond to HTML control elements such as `input` fields. By "reactive," RStudio means that when a value changes, all values with dependencies on the changed value are recalculated, as you'd expect from a spreadsheet program. Figure 9 shows an interactive Shiny app with two widgets for input and a shaded choropleth map of U.S. census data for output.

The interactive Shiny app in Figure 9 is a good example of how you can allow middle management to play with the data without their having to know what's under the covers.

To limit what is recomputed when input changes, the reactive wrapper function caches its values and recomputes only those that are invalid. I'll forgo burdening you with an example, although you'll find one in Shiny Lesson 6. Shiny apps can run on your own hardware, or you can publish them to the shinyapps.io server. For a quick example, have a look at Figure 10.

To limit what is recomputed when input changes, the reactive wrapper function caches its values and recomputes only those that are invalid.



censusVis

Create demographic maps with information from the 2010 US Census.

Choose a variable to display

Percent Hispanic

Range of interest:

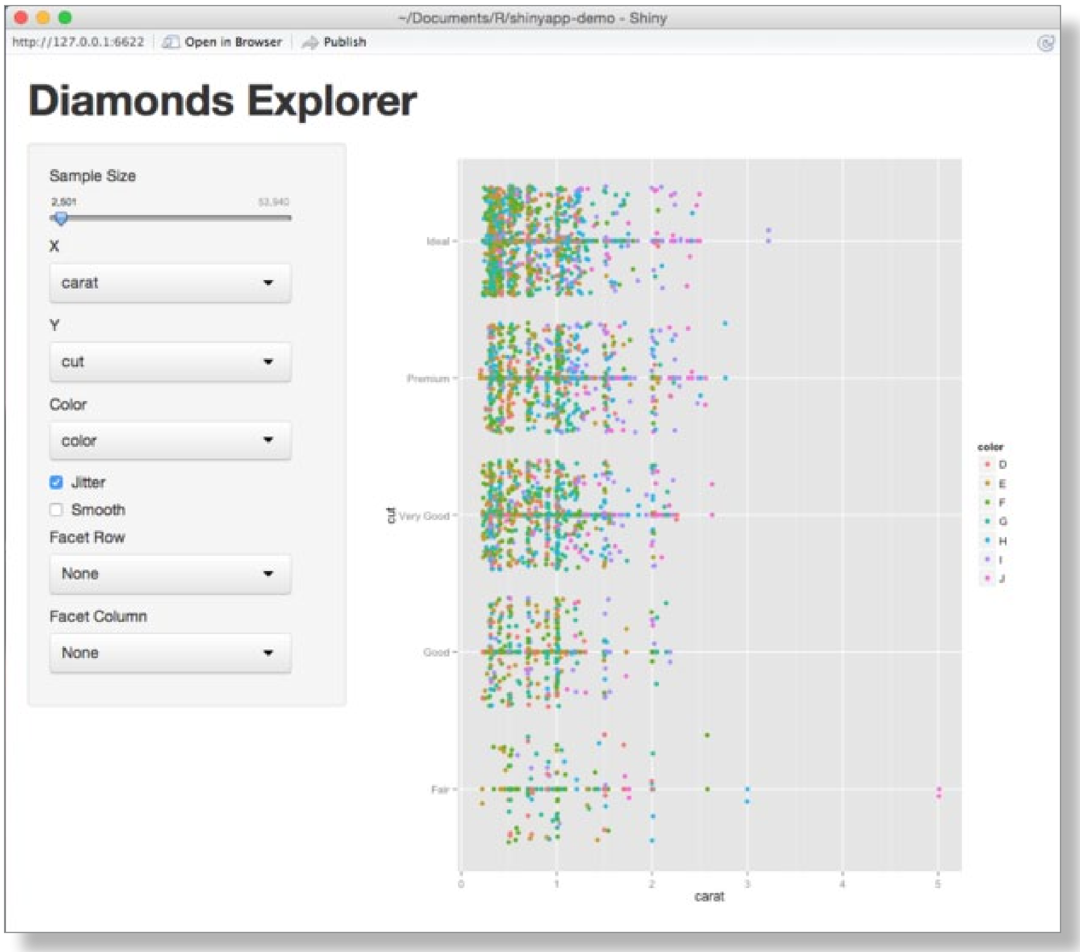
0 100

```
# server.R

library(maps)
library(mapproj)
counties <- readRDS("data/counties.rds")
source("helpers.R")


shinyServer(
  function(input, output) {
    output$map <- renderPlot({
      args <- switch(input$var,
                     "Percent White" = list(counties$white, "darkgreen", "% White"),
                     "Percent Black" = list(counties$black, "black", "% Black"),
                     "Percent Hispanic" = list(counties$hispanic, "darkorange", "%
                     "Percent Asian" = list(counties$asian, "darkviolet", "% Asian"

      args$min <- input$range[1]
      args$max <- input$range[2]

      do.call(percent_map, args)
    })
  }
)
```

```
Listening on http://127.0.0.1:6622

> help c
Error: unexpected symbol in "help c"
> install.packages(c("maps","mapproj"))
trying URL 'http://cran.rstudio.com/bin/macosx/mavericks/contrib/3.1/maps_2.3-9.tgz'
Content type 'application/x-gzip' length 2069999 bytes (2.0 Mb)
opened URL
==================================================
downloaded 2.0 Mb

trying URL 'http://cran.rstudio.com/bin/macosx/mavericks/contrib/3.1/mapproj_1.2-2.tgz'
Content type 'application/x-gzip' length 73458 bytes (71 Kb)
opened URL
==================================================
```

Figure 9.
The U.S. map rendered in the example above changes when the user varies the input values. Note the `readRDS` *base function to read a serialized R object, the* `source` *function to include additional code, the* `renderPlot` *function (from the* `shiny` *package) to render a reactive plot, and the* `do.call` *base function to construct and execute a function call. The* `percent_map` *function is defined in* `helpers.R` *to render the shaded county map and the state outline map.*

With the addition of RStudio as an IDE, developing R applications can be quite productive.



Diamonds Explorer

Sample Size
2,501 — 53,940

X
carat

Y
cut

Color
color

[✓ Jitter]

☐ Smooth

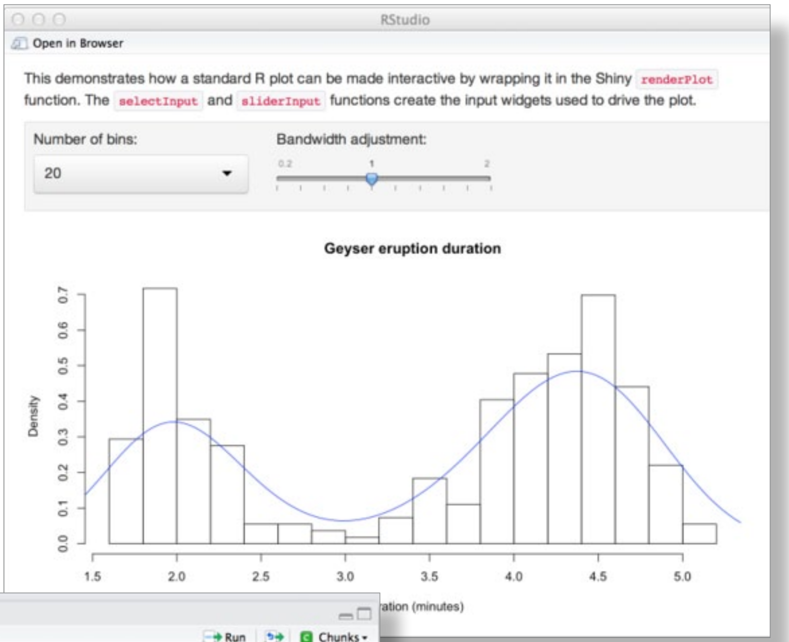
Facet Row
None

Facet Column
None

Figure 10. *The interactive Shiny demo app running on my local system. You can run it yourself at https://mheller.shinyapps.io/shinyapp-demo/.*

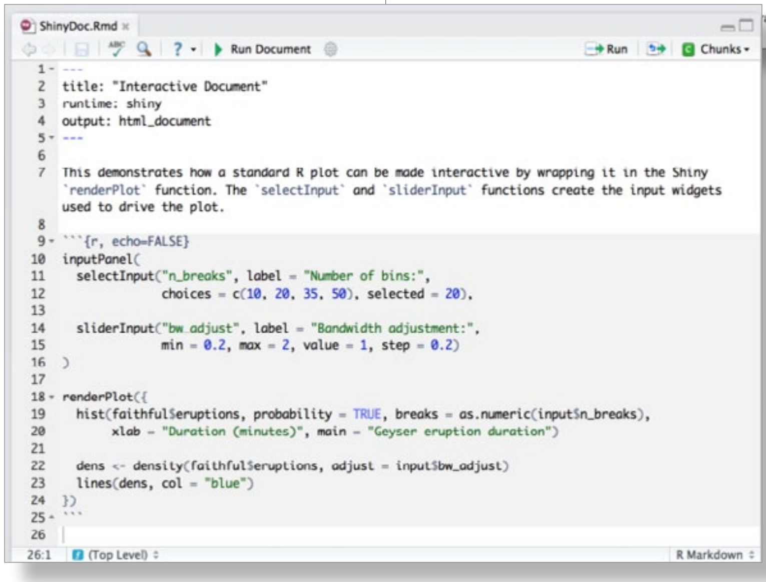
Shiny apps should satisfy the needs of middle managers. Now what about top management?

If you are a GitHub user or have simply been paying attention to the Web and developer landscapes the last 10 years, you'll know about the Markdown language for generating formatted documents in HTML and other tag-based markup languages. RStudio includes a Markdown implementation and extends it to include embedded R code chunks and both LaTeX and MathML in the R Markdown package. You can also create interactive R Markdown documents using Shiny and publish them to your own Shiny server or to shinyapps.io. For an example, see Figure 11.



This demonstrates how a standard R plot can be made interactive by wrapping it in the Shiny `renderPlot` function. The `selectInput` and `sliderInput` functions create the input widgets used to drive the plot.

Figure 11.
An example of R Markdown made interactive. The underlying code is a header block, a few lines of Markdown, and a dozen lines of R. See Figure 12.



~~~
---
title: "Interactive Document"
runtime: shiny
output: html_document
---

This demonstrates how a standard R plot can be made interactive by wrapping it in the Shiny
`renderPlot` function. The `selectInput` and `sliderInput` functions create the input widgets
used to drive the plot.

```{r, echo=FALSE}
inputPanel(
  selectInput("n_breaks", label = "Number of bins:",
              choices = c(10, 20, 35, 50), selected = 20),

  sliderInput("bw_adjust", label = "Bandwidth adjustment:",
              min = 0.2, max = 2, value = 1, step = 0.2)
)

renderPlot({
  hist(faithful$eruptions, probability = TRUE, breaks = as.numeric(input$n_breaks),
       xlab = "Duration (minutes)", main = "Geyser eruption duration")

  dens <- density(faithful$eruptions, adjust = input$bw_adjust)
  lines(dens, col = "blue")
})
```

~~~

Figure 12.
The faithful$eruptions data used in this example is from the Old Faithful geyser data built into the R Datasets package.

The power of R

As we've seen, R is a useful tool for data scientists and statisticians, and its somewhat nonstandard scripting language will be of interest to programmers who might otherwise resort to Python (with NumPy, Pandas, and StatsModels); SQL (for data held in a database); or SAS (and its GUI derivative, JMP) for their data analysis. Compared to Excel, R has considerably more statistical and graphing power, especially if you add packages for your particular needs, and it's much more auditable. It's far easier to validate an R script than a spreadsheet full of formulas.

With the addition of RStudio as an IDE, developing R applications can be quite productive. RStudio Server allows companies to take advantage of the huge RAM and many processors available in big server hardware, Shiny turns R into a Web application server, and R Markdown allows you to use R for reports.

On the other hand, the great power of R and the large number of R packages available can make for a fairly intimidating learning curve. It helps a lot to have some statistics background when learning and using R, but that's true for all data science. As can be said for any other programming language with many libraries available, your best strategy for learning R is to take it one step at a time. ■

Martin Heller *is a contributing editor for InfoWorld reviews.*