Files and Recordkeeping with R

Michael A. Covington
Institute for Artificial Intelligence
The University of Georgia

2011 December 10

Introduction

This tutorial assumes you have already started using R, and now you want to know more about how to preserve and replicate your work.

Recommended reading: Paul Teetor, *R Cookbook* (O'Reilly).

Importance of keeping records

When you do an experiment, your own notes and files are the only way of guaranteeing the accuracy and the honesty of your data. Keep good records.

Do not edit the original data files from your experiment (such as the output of MATTR or CPIDR). Instead, make copies of them, and do all needed editing on the files.

When you do a statistical analysis, keep records of exactly how you did it, so that if any question arises, you can replicate it. This tutorial will tell you how.

Handling data files

Changing the working directory (folder)

If you tell R what folder your working files are in, you won't have to give full paths to the files. To do this, use **File, Change dir...**, or the R command `setwd(path)`. This is very handy when you are going to keep several related files in one folder.

Displaying an open file dialog box

Here is how to make R put up an open file dialog so the user can pick a file. This is handy for use in scripts where you might do the same computation on more than one file.

```
require(tcltk)    # do this just once, to bring in the tcltk library
d <- read.delim(tclvalue(tkgetOpenFile()))
```

Reading tab-delimited and CSV text files

As you know, R can read data frames from tab-delimited text files and from CSV text files, respectively, using **`read.delim("`*`filename`*`")`** and **`read.csv("`*`filename`*`")`**. That is handy because these kinds of text files are compatible with Excel. In both cases R expects the first row to consist of column labels.

R also has methods to read other file formats, including fixed-width columns, columns delimitd with a character you specify, HTML and XML tables, and SQL and other databases. See reference books (especially *R Cookbook*) for more information.

Writing a data frame to a tab-delimited or CSV text file

R can output tab-delimited and CSV text files, too. Examples of commands are:

```
write.delim(dataframe, file="filename", row.names=FALSE)
write.csv(dataframe, file="filename", row.names=FALSE)
```

If you don't include **`row.names=FALSE`**, each row will start with a field giving the row number.

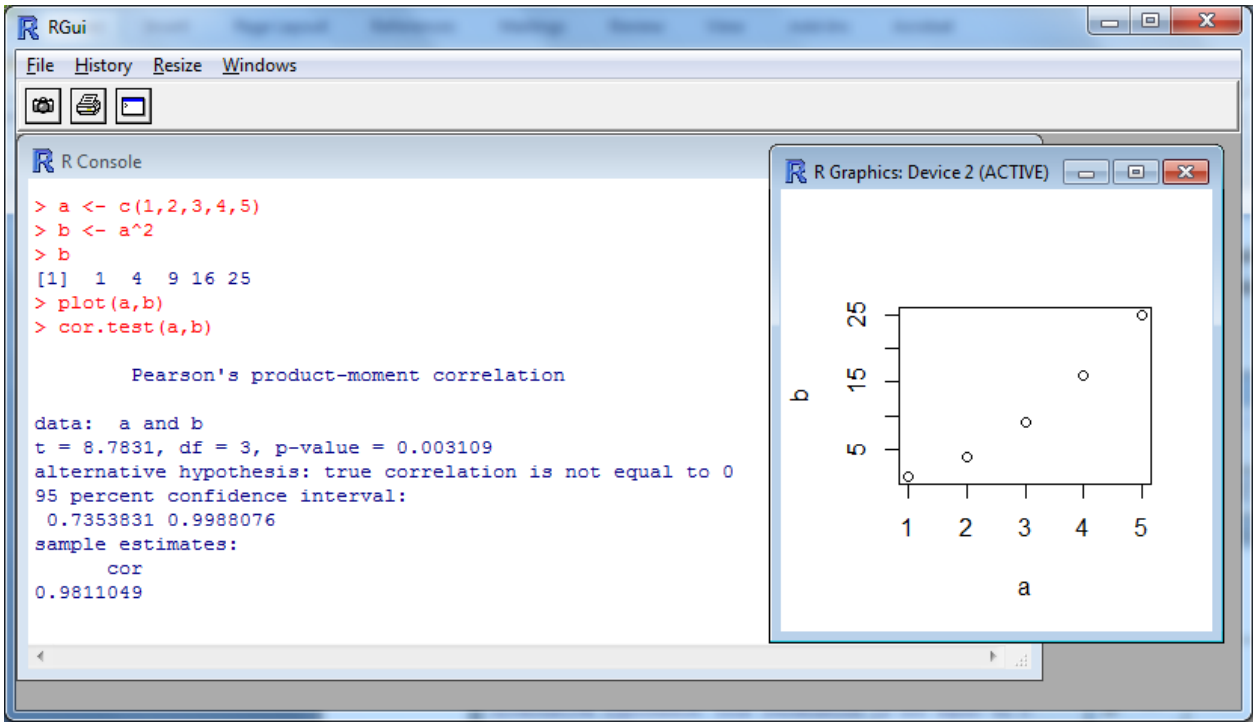
Editing a data frame in memory

You can use **Edit, Data editor...** to edit a data frame within R, in a spreadsheet-like interface. If you do this, make sure to write the data frame out to a file (see previous step) so you'll have a record of it.

Recording and replicating R computations

A concrete example

In what follows, we will use, as an example, this short and somewhat silly R session: Create a vector of numbers, make another vector of numbers by squaring them, plot them against each other, and show that the two vectors of numbers are correlated.



The workspace

The **workspace** consists of everything R has in memory, namely the values of the variables and any user-defined functions. In this example, the workspace contains the two vectors stores in **a** and **b**, and nothing else. It does not include the command history or any printed or plotted output.

R will ask you if you want to save the workspace when you quit, so that you can resume your session with the contents of memory still intact. Also, using **File, Save Workspace** and **File, Load Workspace**, you can put the workspace on a file at any time and load it back.

Loading and saving the workspace is useful for dealing with interrupted work sessions but not for recordkeeping. Workspaces are stored in binary files that other software cannot read.

The history

The **history** is the set of commands you have executed (but not their results or output). You can use **File, Save History** and **File, Load History** to store the commands on a file and to bring them back into R's memory, so that you can recall them with the up-arrow. Loading a history *does not execute the commands;* it only makes R remember you have typed them.

In the example, the history is:

```
a <- c(1,2,3,4,5)
b <- a^2
b
plot(a,b)
cor.test(a,b)
```

Saving a copy of the console session

Using **File, Save To File...** you can save the contents of the "R Console" window, both what you typed and what you got as responses. In our example, this would produce the following text file:

```
> a <- c(1,2,3,4,5)
> b <- a^2
> b
[1]  1  4  9 16 25
> plot(a,b)
> cor.test(a,b)

        Pearson's product-moment correlation
data:  a and b
t = 8.7831, df = 3, p-value = 0.003109
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 0.7353831 0.9988076
sample estimates:
      cor
0.9811049
>
```

Saving a copy of the console session is very useful for recordkeeping. You cannot directly load the session back in, though. Note that the commands all have the command prompt **>** in front of them, just as it appeared on the screen.

Saving a copy of a graphic

Saving the R console does not save the graphics window. To do so, simply right-click on the graphic and you can save it as a Windows metafile or as a PostScript file. You can also copy it to the clipboard as a metafile (ideal for pasting into Word) or as a bitmap.

The saved graphic includes quite a bit of margin space that might hold titles, etc., but is often blank. You may need to paste the graphic into Photoshop or GIMP and trim off the unused white space before using it further.

Scripts

A **script** is a file of ready-to-execute R commands. Running the script is just like typing in the commands *except that values and output usually aren't displayed*. If they were, a long script might clutter up the screen in a very tiresome way.

You can force output to be displayed using **`print()`** (for values of objects) and **`cat()`** (for messages). Remember that comments start with **#**.

Here is a short script that replicates our example session:

```
# Example of an R script
a <- c(1,2,3,4,5)
b <- a^2
cat("The value of b is: ")
print(b)
plot(a,b)
print(cor.test(a,b))
```

Scripts are stored in files with names ending in ".R". To run a script from a file, use **File, Source R code**, or, within R, use the function **`source(filename)`**. There is also a script editor in R.

Important technique: Making a script to replicate your work

Whenever you do any important computation in R, it is a good idea to make a script to replicate it. That way, you can do it again, with or without corrections, in the future.

Very commonly, this script will start with a **`read.delim()`** or **`read.csv()`** operation to read a data file. You can keep the script in the same folder as the data file that it reads. Then you can change R to that folder before running the script, and the script itself need not include the path to the folder.

To make such a script, start by saving the command history of your computation. Edit it with a plain text editor to take out unneeded steps and to add comments, cat() to display messages, and print() to display results, as needed. Then try running it, and make sure it's right.

And now you have your whole computation, ready to be saved for posterity and replicated as needed. This should be an important part of your research records.

Defining your own functions

In R, defining your own functions is a completely separate activity from scripting, although you can (and often will) include a function definition in a script. A script is simply a series of commands to execute. A function definition is a command, which can be many lines long, and which can be typed on the console or in a script.

Example of a function

Here's a rather simple function that takes a number and returns 3 or 4, whichever is nearer the number you gave:

```
f <- function(x) { if (x > 3.5) 4 else 3 }
```

You can have line breaks in the function definition, of course; the final **`}`** makes clear where it ends. There are **`for`** loops and many other control constructs that can be used within functions.

Editing a function in memory

To edit a function in memory, use the command

`fix(`*name*`)`

to bring up the editor (where *name* is the name of the function). If the function does not yet exist, you will be given an empty framework in which to define it.

Saving a function to a file

Caution! So far, all you've done is define and modify the function in memory. If you exit without saving the workspace, it will be lost.

To save it to a file, use `fix(`*name*`)` to get back into the function editor, and choose **File, Save as...** . This is exactly like editing a script – in fact it creates an .R file with the function definition on it, which can be used as a script to define the function.