R Help Files

Contents

1 Introduction

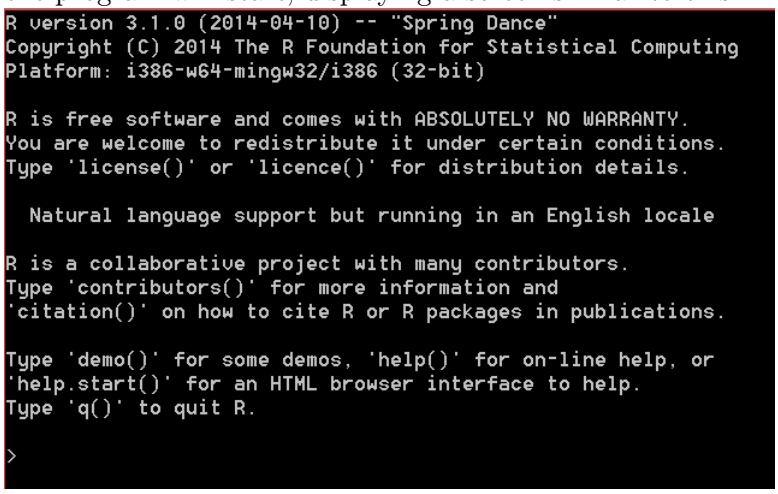
This document is meant for students to get an introductory look at R (and how it compares to and differs from STATA), and to facilitate a switch for both students and instructors from STATA to R, using the INSERT PACKAGE NAME HERE package developed by Scott Emerson, Brian Williamson, and Andrew Spieker in the University of Washington Department of Biostatistics.
R is a functional language at the basic level, meaning that even the simple expression `3 + 5` is handed to a function called `evaluate()`, which then returns the result. Most objects and functions have a print function, so we can either print and display results or store them as an object. STATA, on the other hand, is a command style language. The user types in a command and the result is displayed. In this file we will present some of the useful R functions, but there are many more that we leave to the reader to go out and find. Also, even if we do present a function here, we will be

doing so at the most basic level. Many of the functions have more capabilities than what we show.

2 Project Management

2.1 The Workspace

There are a few differences between R and STATA when it comes to data and project management. The first concept that we must cover is that of the workspace. In STATA, you are only allowed to read in one data set at a time. This data set is your entire workspace, and thus all functions and calls manipulate the data set. In R, however, the workspace consists of many different data sets, values, and functions. Any variable that is assigned a name becomes part of the workspace. Upon exit from R, the program asks the user if they wish to save the workspace. Saying yes saves all variables, functions, and data sets and then loads them back in to the workspace upon startup of the program the next time. It is generally good practice to save the workspace, especially if you know that you will continue to work with the same data next time.
To start, we will assume you are running R from the command line. Later on, if you wish, we will introduce a convenient graphical user interface. However, it is good to learn the basics from the command line and know what is happening behind the buttons in the GUI. Once you navigate to the correct directory for R (in a Windows machine this is generally in "C:\Program Files\R\R-version\bin" - substitue the version of R you downloaded for "version"), you simply type "R" and the program will start, displaying a screen similar to this:



If you type 'q()', you are prompted to `Save workspace image? [y/n/c]:` , where '`y`' is yes, '`n`' is no, and '`c`' is cancel. If you type '`y`', then two files are created: a `.Rhistory` file and a `.RData` file. You can load the `.RData` file at the beginning of a session with the `load()` function, and it brings in all of your variables, data sets, and functions. The `.Rhistory` file saves all of your commands from the session. If you are working with a few different projects, it is a good idea to have a workspace for each of these projects. Then any one workspace doesn't get too cluttered.

2.2 Scripts

Scripts are useful files that can be run by R. A script file can be created in any directory on your computer. Then it can be "sourced", using the `source()` function (given the file path) to run all of the functions saved in the script file. A script is a useful place to write all of your function calls

so that you can save them for later.

2.3 Comments

Commenting your files is always a good idea. Comments provide information as to why you are using certain functions at certain times. The comment key in R is `#`. If you type this at any point on a line, the rest of the line will not be run by R.

3 Data

3.1 Data Types

A huge reason for using R at all is its data capabilities. There are many types of data in R, and thus it is important to know what kind of data you are dealing with at a given time. The "`storage.mode()`" function returns what type the data is. The basic data types are:

3.1.1 Character

A character is an object which represents string values. For example, "3" is a character. To convert an object into a character, use the `as.character()` function.

3.1.2 Complex

A complex number - has an imaginary component.

3.1.3 Integer

A whole number. Integers must be created, since the default computational data type in R is `numeric`.

3.1.4 Logical

`TRUE` and `FALSE` are the logical values in R, usually produced by comparisons. The standard logical operators in R are:

"&"	and
"|"	or
"!"	negation
"=="	equals

3.1.5 Numeric

Decimal values are numerics. Numeric is the default computational data type in R. *Note for the curious: all real numbers in R are stored as "doubles", meaning that the IEEE floating point number is stored as a 64 bit word rather than a 32 bit word.*

3.1.6 NA, NULL, Inf, and NaN

`NA` denotes a missing value in R. `NA` is a logical constant of length 1.
`NULL` is the null object in R. Functions and expressions whose values are undefined sometimes return `NULL`. If a `NULL` value is compared to another (for example, `NULL == 0`), then the result will be the expression `logical(0)`. The length of the `NULL` object is 0.
`Inf` is returned by dividing any number besides 0 by 0.
`NaN` (not a number) is returned by the expression dividing 0 by 0.

```
> 0/0
[1] NaN
> 1/0
[1] Inf
```

3.2 Data Structures

There are a variety of data structures in R. They include vectors, matrices, arrays, data frames, lists, and factors.

3.2.1 Vectors

A vector is the most simple data structure in R. A vector stores any number of individual values, but they must be all of the same type. A vector is created with the `c()` function.

3.2.2 Matrices

A matrix is a table, is essentially a two-dimensional vector. All values in the matrix must be of the same data type, and a matrix must be rectangular - that is, each row must have the same number of columns as the others and each column must have the same number of rows as the others. A matrix is created with the `matrix()` function.

3.2.3 Arrays

An array is a high-dimensional matrix. An array is created with the `array` function.

3.2.4 Data frame

A data frame is a table, like a matrix. However, the columns in a data frame may have different data types. A data frame is created with the `data.frame()` function.

3.2.5 List

A list is an ordered collection of objects. The values in a list can be any data type or data structure, and they don't have to have the same size. A list is created with the `list()` function. *A note for the curious: lists indirectly reference their values. They are stored in different places in memory as addresses.* Lists are sometimes returned from functions.

3.2.6 Factor

A factor stores the nominal values of the data (starting with 1) in a vector, and stores the associated names of the data as a vector of character strings. A factor is created with the `factor()` function. For example, let's say that we have a vector with twenty ``male''s and twenty ``female''s. Then a factor would store this data as twenty 1s and twenty 2s, and would have a key telling us that `1` was male and `2` was female.

3.3 Assigning a Value to a Variable

Let's say we want to creat a variable called `junk`. Then we must assign `junk` a value. Here we will assign it the number `1`. Thus, we type

```
junk <- 1
```

And now any time that we wish to see what `junk` is, we type `junk` and get the output

```
> junk
[1] 1
```

Any time we wish to assign a value to a variable, we use the "`<-`" function. This guarantees that we perform the correct assignment (some functions and packages were developed in a time where using "=" as assignment will not work correctly).

3.4 Indexing Values in a Data Structure

There are three main ways of accessing data in a data set. The main ways deal with data sets in vector or matrix form (similar syntax), list form, or data frame form.

3.4.1 The "[]" Operator

The first way we will cover is for vectors and matrices. Let's say we have a vector representing the numbers 1-10. This vector can be stored as

```
> test <- c(1,2,3,4,5,6,7,8,9,10)
```

Now if we wish to access the 6, we type

```
> test[6]
```

Which returns

```
[1] 6
```

Recall that R is a functional language at its base. Thus, if we want to see the sixth value of `test`, we can type in any expression which will evaluate to 6. For example:

```
> test[3+3]
[1] 6
```

If we want to see the first five elements of the vector, we type

```
test[1:5]
```

which returns

```
[1] 1 2 3 4 5
```

Now, a vector is simply a matrix with one row. Thus let's consider a matrix with 735 rows and 30 columns (we will be using this matrix later as well) called `mri`. If we wish to access the entire first row (a vector!) we type

```
> mri[1,]
```

Which means that we are selecting the first row and all of the columns. The `[row,column]` format is followed for matrices. The output of this call is

```
  ptid mridate age male race weight height packyrs yrsquit alcoh physact chf chd
1    1  120791  72    1    2    173    169      54       0     0    9.84   0   1
  stroke diabetes genhlth ldl alb crt plt sbp    aai   fev dsst atrophy whgrd
1      2        0       3 135 3.7 1.4 275 139 1.0303 1.284   25      20     2
  numinf volinf obstime death
1      1 7.4613    2110     0
```

If we want to access a specific value, we enter in both the row and column of interest. For example, the first value in the matrix.

```
> mri[1,1]
[1] 1
```

3.4.2 Special Issues for Indexing a Matrix

However, when we are subscripting a matrix, we may wish to be sure what class our returned value is. For example, a test matrix

```
> m <- matrix(rep(1, 30), nrow=3)
> m
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]    1    1    1    1    1    1    1    1    1     1
[2,]    1    1    1    1    1    1    1    1    1     1
[3,]    1    1    1    1    1    1    1    1    1     1
```

If we want the first row of the matrix, and want it to be a matrix, we will run into problems if we simply type `m[1,]`.

```
> m[1,]
 [1] 1 1 1 1 1 1 1 1 1 1
```

Notice that this returned a vector! If we want a matrix, we must type

```
> m[1,,drop=FALSE]
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]    1    1    1    1    1    1    1    1    1     1
```

Adding the `drop=FALSE` argument after our row and column specifications told R not to drop dimensions and create a vector.

3.4.3 The "$" Operator

A data frame, as we discussed earlier, is a list of vectors. Let's use the `mriTest` data frame that we created earlier. Now we can access a column of the data frame either by the `[,colnum]` or `[,"colname"]` function or by the `$colname` function, as we see here (we only get the first three rows to save paper):

```
> mriTest[1:3,1]
[1] 1 2 3
> mriTest[1:3,"ptid"]
[1] 1 2 3
> mriTest$ptid[1:3]
[1] 1 2 3
```

Last, variables in matrices, lists, and data frames can both be accessed using the `$` operator. Create `sampleList` to have a vector of strings, a matrix, and a single number as follows:

```
> sampleList <- list(c("One", "Two", "Three"), mri, 1)
```

Now we wish to name the elements of `sampleList`:

```
> names(sampleList) <- c("strings", "matrix", "number")
```

Now we can access the vector of strings either with the `[[1]]` function or with the `$strings` function, as we can see here:

```
> sampleList[[1]]
[1] "One"   "Two"   "Three"
> sampleList$strings
[1] "One"   "Two"   "Three"
```

3.4.4 The "[[]]" Operator

In a list, we use the `[[]]` operator to access elements of the list. For example, if `sampleList` is a list, then `sampleList[[1]]` returns the first element in `sampleList`. Each individual element in a list can be a different type. Thus we can have a list where the first element is a vector of strings, the second element is a matrix, and the third element is a single number.

3.5 Advanced Indexing

3.5.1 Indexing with a Vector

Say we wish to access columns 3, 5, and 30 from `mri`, along with the first five rows. Then we use a vector to access these columns:

```
> mri[1:3, c(3,5,30)]
  age race death
1  72    2     0
2  81    2     0
3  90    2     0
```

We can accomplish the same thing by using the names of the columns, in `mri[1:3, c("age", "race", "death")]`.

3.5.2 Indexing with a Matrix

If we have a matrix (say `mri`) and we wish to reference certain cells, we can use a matrix as a reference. The reference matrix has two columns, the first corresponding to the row of the cell in question and the second corresponding to the column of the cell in question. So if we are interested in seeing cells `[1,1]`, `[2,2]`, and `[3,3]`, we create this matrix:

```
> ref <- matrix(c(1,1,2,2,3,3), ncol=2, byrow=TRUE)
> ref
     [,1] [,2]
[1,]    1    1
[2,]    2    2
[3,]    3    3
```

Notice that when we create a matrix, we can specify how many columns (or rows) we want with the `ncol` (`nrow`) argument, and we can specify how the matrix is filled in with the `byrow` argument. Now we use `ref` to reference these cells of `mri`

```
> mri[ref]
[1]     1 90192    90
> mri[1,1]
[1] 1
> mri[2,2]
[1] 90192
> mri[3,3]
[1] 90
```

Notice that using `ref` has returned the same values as indexing each cell individually!

3.5.3 Indexing with a Negative Number

Recall our vector `test`, which has ten values. If we want all of the values of test but the first, rather than indexing with the sequence `2:10`, we can simply write:

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

We can do the same thing in a matrix - if we write `mri[-1,-1]`, R will print the whole matrix minus the first row and the first column.

3.5.4 Indexing using Logicals

Sometimes we wish to exclude values in the data based on some logical comparison. For example, in `mri`, we might to only see the values for females, or we wish to only see the values in `test` which are greater than 5. We accomplish this by using logical comparisons in our indexing. To get the values in `test` which are greater than 5, we type

```
> test[test > 5]
[1]  6  7  8  9 10
```

To get only the females in `mri` (and all of the columns) we type `mri[mri[,"male"] == 0,]`.

4 Data Input

4.1 Entering from the Keyboard

We can enter data from the keyboard as we have done before (for example `test <- 1:10`), or we can create a data frame or some other data structure and edit using R's built in spreadsheet editor.

```
# create a data frame
newdata <- matrix()
# bring up the spreadsheet editor and save changes into newdata
newdata <- edit(newdata)
```

4.2 Reading in Data from a File

Now we are ready to read in a data set. Since we can assign a value to a variable, assigning a data set to a variable is easy. For most purposes, the "`read.table()`" function is sufficient to read in data (most data is available in `.txt` format). The "`read.table(filename, ...)`" function can read in files from the internet or from your computer - you just have to give it the full path name (denoted by `filename`, and `...` denotes any options you wish to specify). It is also important to view the text file with the data beforehand to determine if there are headers to include. For example, we can read in the '`mri`' data set from Scott Emerson's webpage:

```
mri <- read.table("http://www.emersonstatistics.com/datasets/mri.txt",
                  header=TRUE, quote="\"")
```

Where here we want to include headers, and the '`quote`' option determines what character will give us a quote string. If we had the text file saved on the hard drive, rather than entering in the web address we would put the path name in quotation marks. Another useful method for reading in data is the "`read.csv()`" function, which follows a similar syntax. Also, if we wish to view the entire data set, use the `View(dataname)` function. This will pop up a new window with the data.

5 Manipulating a Data Set

5.1 Changing Variable Names

Say that we read in the data set `mri`, but we want to change the name of `stroke`, one of the variables, to `hadStroke`. We first make a test data set called `mriTest` to store this while we make

sure the name is correct, then find which column is the stroke column, and then change the name:

```
> names(mri)
 [1] "ptid"     "mridate"  "age"      "male"     "race"     "weight"   "height"
 [8] "packyrs"  "yrsquit"  "alcoh"    "physact"  "chf"      "chd"      "stroke"
[15] "diabetes" "genhlth"  "ldl"      "alb"      "crt"      "plt"      "sbp"
[22] "aai"      "fev"      "dsst"     "atrophy"  "whgrd"    "numinf"   "volinf"
[29] "obstime"  "death"
> mriTest <- mri
> names(mriTest)[14] <- "hadStroke"
> names(mriTest)
 [1] "ptid"      "mridate"   "age"       "male"      "race"      "weight"
 [7] "height"    "packyrs"   "yrsquit"   "alcoh"     "physact"   "chf"
[13] "chd"       "hadStroke" "diabetes"  "genhlth"   "ldl"       "alb"
[19] "crt"       "plt"       "sbp"       "aai"       "fev"       "dsst"
[25] "atrophy"   "whgrd"     "numinf"    "volinf"    "obstime"   "death"
```

5.2 Changing Data Types

Notice that if we type `storage.mode(mri)` we learn that `mri` is a matrix. What if we wanted `mri` to be a data frame? Data frames are a special concept in R. A data frame is essentially a list of vectors, which all must be the same length. It is similar to a matrix, but there are some functions which require the data to be a data frame. If we wish to change `mri` to be a data frame, we type

```
mriTest <- as.data.frame(mri)
```

Now we have a data frame. If we wanted to change it back, we could say

```
mriTest <- as.matrix(mriTest)
```

5.3 Attaching a Data Set

We have the `mri` data set, but typing out `mri[,1]` or `mri[,"ptid"]` is too time consuming for every time we want to access the patient id vector. Similarly, typing out `mriTest$ptid` is far too much. If we want to save ourselves some time, we can use the `attach` function to simply type the variable names and use them. However, if we bring in a data set with similar names later on and attach it, we will lose the ability to refer to the original data set. Thus good data management means that when we are done with a data set, we will use the `detach` function. For example,

```
> attach(mri)
> ptid[1:3]
[1] 1 2 3
> detach(mri)
```

5.4 Attaching with the Same Name

Let's say that we have attached the `mri` data set. If we update `mri` and wish to attach it again, we will get a message similar to the following:

```
> attach(mri)
The following objects are masked from mri (position 3):

    aai, age, alb, alcoh, atrophy, chd, chf, crt, death, diabetes, dsst, fev,
    genhlth, height, ldl, male, mridate, numinf, obstime, packyrs, physact, plt,
    ptid, race, sbp, stroke, volinf, weight, whgrd, yrsquit
```

This means that if we refer to one of the names in the list, we are referring to these columns of our new updated `mri`. There are ways to look up the values in the old `mri`, but those are out of the scope of this document.

5.5 Adding Variables

In the `mri` dataset, say we wanted to convert `physact` from kilocalories to calories. We can create a new variable for this, called `calPhysAct`. Assume that we have attached the `mri` data set. Now we have some choices of where to store this new variable. If we simply type

```
> calPhysAct <- 1000*physact
```

we will create a variable in the workspace called `calPhysAct`. However, this variable will not be part of the `mri` data set. If we wish to make it part of the data set, we type

```
> mri$calPhysAct <- 1000*physact
```

Now if we want to access `calPhysAct` by name like we can access `physact`, we must attach the data set again. We use the same functions to add a variable to a data frame.

5.6 Missing Data

In STATA, missing data is coded as a ".". In R, missing values are treated as `NA`. To test if there are missing values, use the `is.na()` function. When reading in data, be careful of missing values. Sometimes they may not be coded in a way that R will know to code them as `NA`. Thus, look at the data first! If there are odd codings, use the `na.strings` option in the `read.table()` function to set which other strings should be treated as `NA`.

6 Descriptive Statistics

Thus far, all we have covered are functions relating to data and data manipulation. However, R also has many powerful functions for doing statistics. Many of these must be downloaded by the user.

6.1 Packages

A package is a suite of functions and methods developed for download and use by anyone who has R. The base package includes many useful functions, but often these aren't enough. Other people - the original developers of R, and other dedicated users - have developed packages which contain useful functions. For instance, the `stats` package contains many useful functions for ANOVA and linear

models. In order to download a package, type `install.packages("packagename")`. Then you may be prompted to enter the CRAN mirror closest to you - this simply tells R where to download the package files from. Once the package has been downloaded, use the `library(packagename)` function to load the package. A package must be loaded only once per R session; however, when you re-open R, you will need to reload the package. Each package has documentation online.

6.1.1 Function Help

Recall that R is a functional language. What do we do if we don't know exactly what a function does? We can either look up the function online (there is documentation for each function in R available online) or we can type the request into R itself. Say we want to know more about the `read.table()` function. Then we simply type `?read.table()`, and the documentation will be brought up for us.

6.2 Simple Descriptives

To find the mean or the median, we simply type `mean(data)` or `median(data)`. Many of the other simple functions (`sum` - compute a sum, `dim` - return the dimensions of an object) operate in a similar way.

6.3 Other Descriptives and the INSERT PACKAGE NAME HERE package

For some complex descriptive statistics, as well as some statistical tests and plotting, we will turn to the INSERT PACKAGE NAME HERE package. Developed by Scott Emerson, Andrew Spieker, and Brian Williamson at the University of Washington, this package seeks to facilitate an easy switch from STATA to R. The functions follow a similar syntax to STATA, and simplify the work of finding some output in R by placing it all in an easy to manage result. However, some of the functions presented come from other R packages (including the base R package).

6.4 Summarize vs descrip()

In STATA, a simple summary of the data may be obtained by the `summarize` command. In R, using the INSERT PACKAGE NAME HERE package, we will use the `descrip()` function. Details can be found in the documentation for the package. Now, for example, we have a data on FEV1 - the volume of air that can be forcibly blown out in one second, after full inspiration - in a clinical trial. This data can be found in the BIOST 511 webpage. Here we illustrate the difference between the STATA and R output:

STATA Code and Output

```
* Read in data set *
use "https://courses.washington.edu/b511/Data/FEV1ClinTrial.dta"

* Create the table *
summarize

* STATA Output: *
------------------------------------------------------------------------------
Variable |     Obs        Mean     Std. Dev. Min    Max
---------+--------------------------------------------------------------------
```

```
      Y1 |     520    1.546192     .6434104   .31    3.75
      Y0 |     520    1.521327     .5980118   .33    3.68
       T |     520    .4961538     .5004667   0      1
-------------------------------------------------------------------------------
```

R Code and Output

```
#- Read in data set -#
clin.trial <- read.table("http://courses.washington.edu/b511/Data/FEV1ClinTrial.dat", sep="")
names(clin.trial) <- c("Y1", "Y0", "T")
attach(clin.trial)

#- Create the table -#
descrip(clin.trial)

#- R Output -#
                        N    Msng      Mean     Std Dev    Min       25%        Mdn        75%       Max
FEV1_at_24_weeks:     519       0     1.547      0.6433    0.3100    1.0650000  1.430      1.945     3.750
FEV1_at_Baseline:     519       0     1.522      0.5980    0.3300    1.0900000  1.420      1.910     3.680
       Treatment:     519       0     0.4971     0.5005    0.0000    0.0000000  0.0000     1.000     1.000
```

7 Plots

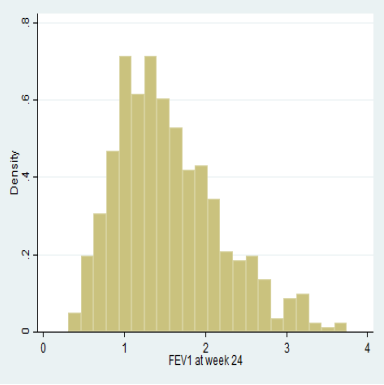
7.1 Boxplots

7.2 Histograms

The syntax for histograms is very similar between STATA and the base R package. However, in R, we must label the axes. Thus we use the `ylab` and `xlab` arguments to apply a y and x-axis label to the plot. The `breaks` command allows us to enter the number of bins we want displayed on the histogram.

STATA Code and Output

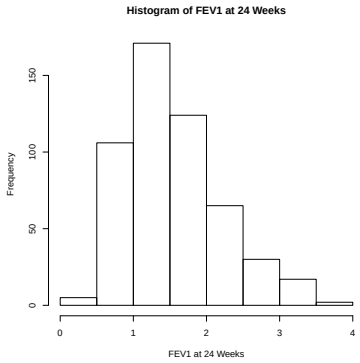
```
* Read in data set *
use "https://courses.washington.edu/b511/Data/FEV1ClinTrial.dta"

* Create a scatterplot with lowess curves and least squares fitted regression lines *
histogram Y1

* STATA Output: *
```



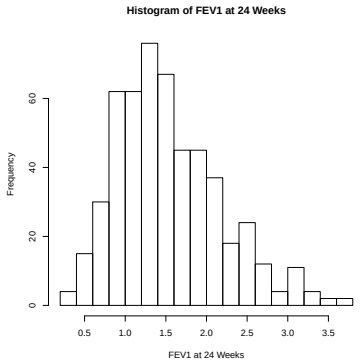

```
* Save the graph *
graph export "test directory\example1.png"
```

R Code and Output

```
#- Read in data set -#
clin.trial <- read.table("http://courses.washington.edu/b511/Data/FEV1ClinTrial.dat", sep="")
names(clin.trial) <- c("Y1", "YO", "T")
attach(clin.trial)

#- Create a scatterplot with lowess curves and least squares fitted regression lines -#
hist(Y1, ylab="Frequency", xlab="FEV1 at 24 Weeks", main="Histogram of FEV1 at 24 Weeks")

#- R Output -#
```




```
#- Save the graph -#
pdf(file="testdirectory/example1.pdf")
> hist(Y1, ylab="Frequency", xlab="FEV1 at 24 Weeks", main="Histogram of FEV1 at 24 Weeks")
> dev.off()

#- Version with 22 bins -#
hist(Y1, ylab="Frequency", xlab="FEV1 at 24 Weeks", main="Histogram of FEV1 at 24 Weeks", breaks=22)
```



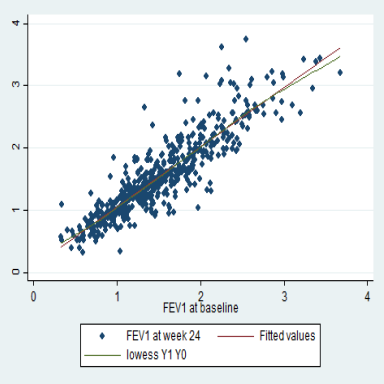

```
#- Save the graph -#
pdf(file="testdirectory/example1.pdf")
hist(Y1, ylab="Frequency", xlab="FEV1 at 24 Weeks", main="Histogram of FEV1 at 24 Weeks", breaks=22)
dev.off()
```

7.3 Scatterplots

The STATA version of scatterplot and the R version (in our package) of scatterplot are very similar. The syntax in each is y-variable followed by x-variable, followed by any arguments. However, in our version there are some extra arguments that need to be added to the function call in order to get exactly what we want.

STATA Code and Output

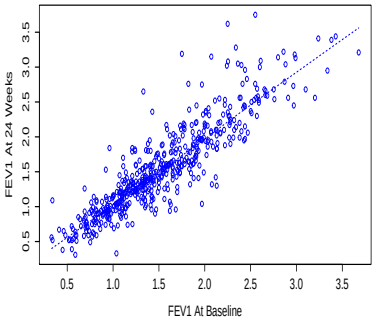
```
* Read in data set *
use "https://courses.washington.edu/b511/Data/FEV1ClinTrial.dta"

* Create a scatterplot with lowess curves and least squares fitted regression lines *
scatter Y1 Y0 || lfit Y1 Y0 || lowess Y1 Y0

* STATA Output: *
```




```
* Save the graph *
graph export "test directory\example1.png"
```

R Code and Output

```
#- Read in data set -#
clin.trial <- read.table("http://courses.washington.edu/b511/Data/FEV1ClinTrial.dat", sep="")
names(clin.trial) <- c("Y1", "Y0", "T")
attach(clin.trial)

#- Create a scatterplot with lowess curves and least squares fitted regression lines -#
scatter(Y1, Y0, ylab="FEV1 At 24 Weeks", xlab="FEV1 at Baseline")

#- R Output -#
```




```
#- Save the graph -#
pdf(file="test directory\example1.pdf")
scatter(Y1, Y0, ylab="FEV1 At 24 Weeks", xlab="FEV1 at Baseline")
dev.off()
```

8 Correlations

In STATA, the command to compute a correlation matrix is `correlate`. In our package, the function called is `correlate()` and it follows much of the same syntax. The basic function takes in an arbitrary number of variables and returns the correlation matrix, calculating Pearson's correlation coefficient. Note that STATA will not display one of the diagonal elements since the two values are the same, while R does display it.

STATA Code and Output

```
* Read in data set *
use "https://courses.washington.edu/b511/Data/FEV1ClinTrial.dta"

* Calculate the correlation matrix between Y1 and Y0 *
correlate Y1 Y0

* STATA Output: *
(obs=520)

             |       Y1       Y0
-------------+------------------
          Y1 |   1.0000
          Y0 |   0.8932   1.0000
```

R Code and Output

```
#- Read in data set -#
clin.trial <- read.table("http://courses.washington.edu/b511/Data/FEV1ClinTrial.dat", sep="")
names(clin.trial) <- c("Y1", "Y0", "T")
attach(clin.trial)

#- Calculate the correlation matrix -#
correlate(Y1, Y0)

#- R Output -#
Tabled correlation statistics by strata
Call:
      correlate(Y1, Y0)
     Method:  Pearson
     Data  :  Pairwise Complete
            - NaN denotes strata with no observations
            - NA arises from missing data

##### ALL DATA
   ##  Estimated Correlation Coefficients
     Y1:     Y0:
Y1:   1.0000  0.8932
Y0:   0.8932  1.0000
```

8.1 Pairwise Correlations

The command to compute pairwise correlations in STATA is `pwcorr`. In R, using our package, you simply add the `use="pairwise.complete.obs"` argument in the `correlate()` function. If `"pairwise.complete.obs"` is entered, then the pairwise correlations are computed using all cases that are not missing data. In the examples presented below, there will be no difference between using pairwise correlation and standard correlation coefficients.

STATA Code and Output

```
* Read in data set *
use "https://courses.washington.edu/b511/Data/FEV1ClinTrial.dta"

* Calculate the pairwise correlation matrix between Y1 and Y0 *
 pwcorr Y1 Y0

* STATA Output: *

             |       Y1       Y0
-------------+------------------
          Y1 |   1.0000
          Y0 |   0.8932   1.0000
```

R Code and Output

```
#- Read in data set -#
clin.trial <- read.table("http://courses.washington.edu/b511/Data/FEV1ClinTrial.dat", sep="")
names(clin.trial) <- c("Y1", "Y0", "T")
attach(clin.trial)

#- Calculate the pairwise correlation matrix -#
correlate(Y1, Y0, use="pairwise.complete.obs")

#- R Output -#
Tabled correlation statistics by strata
Call:
      correlate(Y1, Y0, use = "pairwise.complete.obs")
     Method:  Pearson
     Data  :  Pairwise Complete
            - NaN denotes strata with no observations
            - NA arises from missing data

##### ALL DATA
   ##  Estimated Correlation Coefficients
     Y1:     Y0:
Y1:   1.0000  0.8932
Y0:   0.8932  1.0000
```

9 Cumulative Distribution Functions

Say we calculate a test statistic (z-score, t-statistic, chi-squared statistic) and want to find the p-value. Recall that for a p-value we want to find the probability of values at least as extreme as our value. Thus in STATA, if we have a z-score of 1, the correct command is `display 1-normprob(1)`. In R, the function is `1-pnorm(1)`. Similar functions - `p` followed by the name of the distribution - can be found for many of the probability distributions.

STATA Code and Output

```
* Calculate the p-value for a z-score of 1 *
display normprob(1)

* STATA Output: *
.15865525
```

R Code and Output

```
#- Calculate the p-value -#
1-pnorm(1)

#- R Output -#
[1] 0.1586553
```

10 Linear Regression

In both STATA and our regression function in R, the syntax is dependent variable (y) followed by independent variable (x). The correct STATA command is `regress`, while the function in R is `regress()`. However, we must specify some of the arguments to `regress()` before we use it. First is the `fnctl` argument. This allows the user to specify the desired summary measure of the distribution - acceptable choices are `"mean"`, `"geometric mean"`, `"odds"`, `"rate"`, and `"hazard"`. Next, the dependent variable is entered. Last, the independent variables are entered (in vector, matrix, or list of variables).
Also, the `regress()` function relies on the `sandwich` package to compute robust standard errors (requested by default), so install and load it if you do not already have it loaded.

STATA Code and Output

```
* Read in the data *
import delimited "http://www.emersonstatistics.com/datasets/mri.txt", delimiters("whitespace", collapse)
* Edit the data if necessary *

* Perform the regression *
regress stroke race male

* STATA Output: *
```

```
      Source |       SS       df       MS              Number of obs =     735
-------------+------------------------------           F(  2,   732) =    4.23
       Model |  3.22884311     2  1.61442156           Prob > F      =  0.0150
    Residual |   279.57932   732  .381938962           R-squared     =  0.0114
-------------+------------------------------           Adj R-squared =  0.0087
       Total |  282.808163   734  .385297225           Root MSE      =  .61801

------------------------------------------------------------------------------
      stroke |      Coef.   Std. Err.      t    P>|t|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
        race |  -.0014108   .0342548    -0.04   0.967    -.0686601    .0658385
        male |   .1325506   .0455919     2.91   0.004     .0430442    .2220571
       _cons |   .1725898   .0554123     3.11   0.002     .0638039    .2813758
------------------------------------------------------------------------------
```

R Code and Output

```
#- Read in the data -#
mri <- read.table("http://www.emersonstatistics.com/datasets/mri.txt",
                  header=TRUE, quote="\"")
#- Attaching the data -#
attach(mri)

#- Regressing stroke vs race and male (and calculating based on the mean) -#
regress("mean", stroke, cbind(race, male))

#- R Output -#
regress(fnctl = "mean", y = stroke, model = cbind(race, male))



Call:
regress(fnctl = "mean", y = stroke, model = cbind(race, male))

Residuals:
    Min      1Q  Median      3Q     Max
-0.3037 -0.3037 -0.1712 -0.1712  1.8316
```

```
Coefficients:
             Estimate  Naive SE Robust SE      95%L      95%H    F stat df  Pr(>F)
Intercept    0.172590  0.055412  0.052482  0.069558  0.275622 10.814780  1 0.00106
model              NA        NA        NA        NA        NA  4.217292  2 0.01510
 model.race -0.001411  0.034255  0.034842 -0.069813  0.066992  0.001640  1 0.96771
 model.male  0.132551  0.045592  0.045648  0.042934  0.222167  8.431797  1 0.00380

Residual standard error: 0.618 on 732 degrees of freedom
Multiple R-squared:  0.01142,Adjusted R-squared:  0.008716
F-statistic: 4.227 on 2 and 732 DF,  p-value: 0.01496
```