Statistics with R and S-Plus

tutorial presented at the Third Workshop on Experimental Methods in Language Acquisition Research (EMLAR)

Hugo Quené

Utrecht institute of Linguistics OTS, Utrecht University

hugo.quene@let.uu.nl
www.hugoquene.nl

7 November 2006

Abstract

This workshop will introduce the R programming environment for statistical analysis. Contrary to SPSS which is procedure-oriented (commands are verbs, e.g. "compute"), R is object-oriented (objects are nouns, e.g. "factor"). In this workshop, we will try to ease the learning curve of using R for your data analysis. Experience with statistical software is NOT required! We will use data simulation as well as real data sets, to explore topics like t-tests, chi-square tests, and logistic regression. We will also show how R produces publication-quality figures. If time permits, we will also explore how to extend R with your own routines for analyzing and/or plotting data. You are encouraged to bring your own data set, if accompanied by a "codebook" file specifying variables (columns), data codes, etc. (Both files must be in plain ASCII).

1 Introduction

This tutorial offers a first introduction into R, which is an improved and freeware version of S. For most tasks, the freeware R and its commercial sister S-Plus work in the same way and produce similar results. Most of the ideas in this tutorial apply to both R and S-Plus, although this document focuses on R in the interest of clarity.

R is available as freeware from http://www.r-project.org, where one can also find a wealth of information and documentation. S-Plus is dis-

tributed in the Netherlands from CAN (`http://www.candiensten.nl`) in Amsterdam.

This document assumes that R is already properly installed in an MS Windows environment. You, the reader, are assumed to have a some basic knowledge about statistics, equivalent to an introductory course in statistics. This tutorial introduces the R software for statistical analyses, and not the statistical analyses themselves. This tutorial occasionally mentions differences with SPSS, but the tutorial is also intended for novice users of statistical software.

One interesting property of R is that users can develop their own extensions, called *packages*, and distribute them to other users (similar to "extensions" for Mozilla web browsers). Packages may contain custom datasets, additional functions, re-formulations of existing functions, and more.

1.1 What is R?

Somewhat surprisingly, R is several things at once:

- a program for statistical analyses
```
one.lm <- lm(mlu~age,data=mydata) # linear-model regression
```
- a calculator
```
(log(110)-log(50)) / log(2^(1/12)) # compute and show
```
- a programming language (based on the S language)
```
# function to convert hertz to semitones, by Mark Liberman
h2st <- function(h,base=50) {
 semi1<-log(2^(1/12)); return((log(h)-log(base))/semi1) }
```

The assignment operator (`<-`) is further explained in §3.1 below. The hash `#` indicates comment which is not processed.

1.2 object-oriented philosophy

R works in an object-oriented way. This means that *objects* are the most important things in R, and *not* the actions we perform with these objects. Let's use a culinary example to illustrate this. In order to obtain pancakes, a cook needs flour, milk, eggs, some mixing utensils, a pan, oil, and a fire. An object-oriented approach places primary focus on these six objects. If the relations between these are properly specified, then a nice pancake will result. If all necessary objects exist, then the R syntax for my personal recipe would be as follows:

```
batter <- mixed(flour,milk/2) # mix flour and half of milk
batter <- mixed(batter,egg*2) # add 2 eggs
batter <- mixed(batter,milk/2,use=whisk) # add other half of milk
```

```
while (enough(batter)) # FALSE if insufficient for next
   pancake <- baked(batter,in=oil,with=pan,temp=max(fire))
```

This example illustrates that R is indeed a full programming language[1]. In fact, there is no recipe, in the traditional sense. This "pancake" script merely specifies the relations between the ingredients and the result. Note that some relations are recursive: batter can be both input and output of the mixing operation. Also note that the `mixed` relation takes an optional argument `use=whisk`, which will produce a fatal error message if there is no whisk in the kitchen. Such arguments, however, allow for greater flexibility of the `mixed` relation. Likewise, we might specify `baked(in=grease)` if there is no oil in the kitchen. The only requirement for the object supplied as `in` argument is that one can bake in it, so this object must have some attribute `goodforbaking==TRUE`.

For contrast, we might imagine how the pancake recipe would be formulated in a more traditional, procedure-oriented approach. Ingredients and a spoon are again assumed to be present.

```
MIX batter = flour + milk/2 . # utensil?
MIX batter = batter + eggs .
MIX batter = batter + milk/2 .
BAKE batter IN oil .
BAKE batter IN water . # garbage in garbage out
```

The programmer of this recipe has defined the key procedures `MIX` and `BAKE`, and has stipulated boundary conditions such as utensils and temperatures. Optional arguments are allowed for the `BAKE` command, but only within the limits set by the programmer[2].

So far, you may have thought that the difference between the two recipes was semantic rather than pragmatic. To demonstrate the greater flexibility of an object-oriented approach, let us consider the following variant of the recipe, again in R syntax:

```
# batter is done
while (number(pancakes)<2) # first bake 2 pancakes
 pancake <- baked(batter,in=oil,with=pan,temp=max(fire))
feed(pancake,child) # feed one to hungry spectator
# define new function, data 'x' split into 'n' pieces
chopped <- function(x,n=1000) { return(split(x,n)) }
pieces <- chopped(pancake) # new data object, array of 1000 pieces
batter <- mixed(batter,pieces) # mix pancake pieces into batter
```

[1] Technically speaking, R and S are interpreted languages, and not compiled languages. This allows for great flexibility during an interactive session, at the cost of computational speed. Indeed R can be slow for some tasks, although this is hardly an issue with the present hardware configurations.

[2] Moreover, because this is a pre-compiled language, the inner workings of the BAKE command remain a mystery.

```
# etc
```

Such complex relations between objects are quite difficult to specify, if there are strong a priori limits to what one can `MIX` or `BAKE`. Thus, object-oriented programs such as R allow for greater flexibility than procedure-oriented programs.

Users of `Praat` (`http://www.praat.org`) are already familiar with this basic idea. `Praat` has an object window, listing the known objects. These objects are the output of previous operations (e.g. Create, Read, ToSpectrum), as well as input for subsequent operations (e.g. Write, Draw). R takes this basic idea even further: users may create their own *classes* of data objects (e.g. ReversedSound) and may create their own methods or relations to work with such objects (e.g. HideInSong, etc etc)[3].

This object-oriented philosophy results in a different behavior than observed in procedure-oriented software:

> There is an important difference in philosophy between S (and hence R) and the other main statistical systems. In S a statistical analysis is normally done as a series of steps, with intermediate results being stored in objects. Thus whereas SAS and SPSS will give copious output from a regression or discriminant analysis, R will give minimal output and store the results in a fit object for subsequent interrogation by further R functions.
>
> — `http://cran.r-project.org/doc/manuals/R-intro.html`

2 Objects

2.1 vectors

A vector is a simple, one-dimensional list of data, like a single column in Excel or in SPSS. Typically a single vector holds a single variable of interest. The data in a vector can be either numeric, character (strings of letters, always enclosed in double quotes), or boolean (`TRUE` or `FALSE`, may be abbreviated to `T` or `F`).

`c` Atomic data are combined into a vector by means of the `c` (combine, concatenate) operator.

`seq` The sequence operator, also abbreviated as a colon `:`, creates subsequent values.

```
R> x <- 1:5
R> x
[1] 1 2 3 4 5
R> 2*(x-1)
[1] 0 2 4 6 8
```

[3] `Praat` allows the latter but not the former.

Computations are also done on whole vectors, as exemplified above. In the last example, we see that the result of the computation is *not* assigned to a new object. Hence the result is displayed — and then lost. This may still be useful however when you use R as a pocket calculator.

`rep` Finally, the repeat operator is very useful in creating repetitive sequences, e.g. for levels of an independent variable.

```
R> x <- rep(1:5,each=2)
R> x
[1] 1 1 2 2 3 3 4 4 5 5
```

2.2 factors

Factors constitute a special class of variables. A factor is a variable that holds categorical, character-like data. R realizes that variables of this class hold categorical data, and that the values are category labels or *levels* rather than real characters or digits.

```
R> x1 <- rep(1:4,each=2) # create vector of numbers
R> print(x1) # numeric
[1] 1 1 2 2 3 3 4 4
R> summary(x1) # numeric
 Min.  1st Qu.  Median Mean 3rd Qu.  Max.
  1.00 1.75 2.50 2.50 3.25 4.00
R> x2 <- as.character(x1) # convert to char
R> print(x2) # character
[1] "1" "1" "2" "2" "3" "3" "4" "4"
R> x3 <- as.factor(x1) # convert to factor
R> print(x3) # factor
[1] 1 1 2 2 3 3 4 4
Levels:  1 2 3 4
R> summary(x3) # cf summary(x1)
1 2 3 4
2 2 2 2
```

2.3 complex objects

Simple objects, like the ones introduced above, may be combined into composite objects. For example, we can combine all pancake ingredients into a complex object:

```
R> pancake.ingr <- list(flour,milk,eggs,...)
```

In R we often use a *data frame* object to hold data; this is a complex object like an Excel worksheet or SPSS data sheet. The columns represent variables, and the rows represent observations — these may be "cases", or

participants, or repeated measurements, depending on the study[4].

The easiest way to create a data object is to read it from a plain-text (ASCII) file, using the command `read.table`. Remember to use double backslashes in the file specification. An optional `header=TRUE` argument indicates whether the first line contains the names of the variables; argument `sep` specifies the character(s) that separate the variables in the input file.

```
R> myexp <- read.table(file="f:\\temp\\myexp.txt",header=T,sep=",")
```

3 Basic operations

3.1 basics

`<-` This is the assignment operator: the expression to its right is evaluated (if applicable) and then assigned to the object on the left of the operator. Hence the expression `a<-10` means that the object `a`, a single number, "gets" (is assigned) the value of 10. The symbol resembles an arrow in the direction of assignment.

`#` indicates a comment: everything following this symbol, on the same line of input, is ignored.

`scan` This command reads a simple vector from the keyboard. Make sure to assign the result to a new object! Read in the numbers 1 to 10, and assign them to a new object.

> A missing value in any vector is indicated by the special code `NA` (Not Available). R treats all other values as valid data, and you have to specify other missing data values explicitly.
>
> R is case-sensitive, so that `X` and `x` are different objects.
>
> Some common functions and operators in R have single-character names: e.g. `c` and `t`. Do not use these for your own objects, because these functions will then no longer be accessible.
> You can always check whether an intended object name is already in use, by typing the intended object name (see below).

`objects` This command shows a list of all objects in memory (similar to the contents of the `Praat` Objects window). With `objects(pattern="...")` the list is filtered so that only the objects matching the pattern are shown.

`rm` Objects are removed *forever* with this command.

[4]For repeated measures analyses, R does not require a multivariate layout, with repeated measures for each participant on a single row, as in SPSS. R uses a univariate layout, with each measurement on a single row of input.

`print` Contents of an object can be inspected with this command, or by just entering the name of the object, as in some examples above.

`summary` This command offers a summary of an object. The result depends on the data class of the object, as illustrated in section 2.2 above.

workspace R holds its objects in memory. The whole workspace, containing all data objects, can be stored from the RGui console window (`File > Save Workspace ...`). This allows you to save a session, and continue your work later (`File > Load Workspace...`).

`save` (to write) and

`load` (to read) an object from/to memory to hard disk. By default, R data objects have the extension `.rda`.

> The backslash \ is a special character in R. If you specify a path (folder) in the filename, you must use *double* backslashes.
> `R> save(x3,file="f:\\temp\\x3.rda")`

3.2 subselection

Subselection within an object is a very powerful tool in R. The subselection operator `x[...]` selects only those data from object `x` that match the expression within brackets. This expression can be a single index number, a sequence or list of numbers, or an evaluated expression, as illustrated in the following example.

```
R> # 'x' contains 30 numbers from normal distribution,
R> # but 3 of them are NA.
R> # is.na returns TRUE/FALSE for each member of 'x'.
R> # table summarizes categorical data, e.g.  output of is.na
R> table( is.na(x) )
FALSE TRUE
   27     3
R> ok <- !is.na(x) # exclamation mark means NOT
R> which(!ok) # which index numbers are NOT ok?  inspect!
[1] 14 15 16
R> mean(x[ok]) # select ok values, compute mean, display
[1] -0.4491975
R> x[!ok] <- mean(x[ok]) # replace NAs by mean
```

4 Exploratory data analyses

R is more graphically oriented than most other statistical packages; it relies more on plots and figures for initial exploratory data analysis. Numerical

summaries are of course also available.

`hist` This command produces a histogram. There is a useful optional argument `breaks` to specify the number of bins (bars), or a vector of breaks between bins.

`plot` The default version of this command produces a scattergram of two variables. If you enter just one variable, then the index numbers of the observations are used on the horizontal axis, and the values on the vertical axis. Useful arguments are `title`, and `xlab` and `ylab` for axis labels. In addition, you can use a third variable to vary the plot symbols.

`rug` This command produces tick marks at the actual data values, yielding the visual effect of a rug. This is useful in combination with a scattergram or histogram. Try it out, with the following commands[5]:
`R> x<-rnorm(100); hist(x); rug(x,col=4)`

`boxplot` This yields a boxplot summary of one variable. You can also specify the dependent and independent variable, with argument `dv~iv`; this will produce multiple boxplots for the dependent variable, broken down by the independent variable. Two useful arguments for this command are:
`notch=T` to give additional information about the distribution, and
`varwidth=T` to scale the size of the boxes to the numbers of observations.

`qqnorm` This produces a quantile–quantile (QQ) plot. This plots the observed quantiles against the expected quantiles *if* the argument variable is distributed normally. If the variable is indeed distributed normally, then the data should fall on a straight line. Deviations of this line indicate deviations from normality. You can also plot the expected regression line with `qqline`.

`summary` This command produces a numerical summary of the argument variable. However it does not supply standard measures of variability. We often need

`var` to compute the variance of the argument variable. Standard deviation and standard error may be computed with self-defined functions, e.g.
`sd<-function(x,...){ return(sqrt(var(x,...)))  }`
Here the dots are used to pass along additional arguments to function `var` when calling function `sd`.

[5]The semicolon `;` separates multiple commands on a single line of input.

`length` returns the length of the argument variable, i.e. the number of observations in that vector. This is useful for checking the number of data, as a preliminary for further analyses.

```
valid.n <- function(x){ length(x)-length(which(is.na(x))) }
```

Now that we have obtained such insightful figures, we like to include these in our documents. The best procedure is to activate the graphics window, by clicking on its title bar. This changes the menu and buttons in the main R window. Choose `File > Save as...` and select your desired output format. Figures in (MS Windows) Metafile format (with extensions `emf,wmf`) are easy to import into MS Office applications. Figures in PNG format (extension `png`) are easy to include in LaTeXdocuments.

5 Testing hypotheses

formula When testing hypotheses, and building regression models, we need to specify the relations between variables. This is done in R by means of a *formula*, which is needed in many statistical functions. In general, such a formula consists of a response variable, followed by the tilde symbol `~`, followed by a list of independent variables and/or factors. In this list, the colon `:` indicates an interaction effect (instead of the sequence operator), and the asterisk `*` is shorthand for main effects plus interactions (instead of the multiplication operator). By default, the intercept `~1` is included in the formula, unless suppressed explicitly (`~-1`). We have already encountered formulas above, e.g. in the boxplot example.

```
y ~ x1+x2 # only main effects
y ~ x1*x2 # x1 + x2 + (x1:x2)
```

Further shorthand abbreviations are also available:

```
# only main effects and second-order interactions
y ~ (x1*x2*x3*x4)^2
```

Consult the help files for further information on how to specify complex models.

`t.test` There are three ways to use the t test. First we create some simulated data to work with:

```
R> y1 <- rnorm(n=100,mean=0) # random from normal distr
R> y2 <- rnorm(n=100,mean=0.2)
R> x <- rep(1:2,each=50) # to use as IV
```

In a one-sample test, the mean is compared against an expected mean, with

```
R> t.test(x2,mu=0).
```

In a two-sample test with independent observations, we often compare the same dependent variable, broken down by an independent variable.

```
R> t.test( y1[x==1], y1[x==2] ) # y1 broken down by x
```

> The single equal-sign is used to pass parameters to functions, as illustrated above when using `rnorm`. The double symbol `==` is the is-equal-to operator; `!=` is the is-not-equal-to operator.

In a two-sample test with paired observations, we often compare two different observations, stored in two different variables.

```
R> t.test( y1, y2 )
```

Note that the number of observations in the test (and hence d.f.) varies in these examples.

`chisq.test` First, let us create two categorical variables, derived from a speaker's `age` (in years) and average `phraselength` (in syllables), for 80 speakers in the Corpus of Spoken Dutch. Categorical variables are created here with the `cut` function, to create `breaks=2` categories of `age` (young and old) and of `phraselength` (short and long).

```
R> age.cat <- cut(age,breaks=2)
R> phraselength.cat <- cut(phaselength,breaks=2)
```

The hypothesis under study is that older speakers tend to produce shorter phrases. This hypothesis may be tested with a χ^2 (chi square) test.

```
R> table(age.cat,phraselength.cat) # show 2x2 table
          phraselength.cat
age.cat   (6.09,10.4] (10.4,14.6]
  (21,40]          24          16
  (40,59]          32           8
R> chisq.test(age.cat,phraselength.cat)
 Pearson's Chi-squared test with Yates' continuity correction
 X-squared = 2.9167, df = 1, p-value = 0.08767
```

Although the data in the table seem to support the research hypothesis, the probability of these data under H_0 is still $p = .088$, which exceeds the conventional $\alpha = .05$. Hence H_0 is not rejected.

`aov` This function performs a between-subjects analysis of variance, with only fixed factors. (More complex analyses of variance, with repeated measures, are beyond the scope of this introductory tutorial.) In the example below we create a response variable `aa` which is not normally distributed[6] (check with `hist`, `qqnorm`, etc).

```
R> a1 <- rpois(20,lambda=2)
```

[6]The dependent variable `aa` follows a Poisson distribution, with λ varying between conditions of `x1`. The Poisson distribution "expresses the probability of a number of events occurring in a fixed period of time if these events occur with a known average rate [λ], and are independent of the time since the last event"

```
R> a2 <- rpois(20,lambda=4)
R> a3 <- rpois(20,lambda=6)
R> aa <- c(a1,a2,a3)
R> x1 <- as.factor(rep(1:3,each=20))
R> x2 <- as.factor(rep( rep(1:2,each=10), 3))
R> model1.aov <- aov(aa~x1*x2)
```

6 Regression

`lm` This function is used for regression according to a linear model, i.e. linear regression. It returns a model-class object. There are specialized functions for such models, e.g. to extract residuals (`resid`), to extract regression coefficients (`coef`), to modify (`update`) the model, etc.

In the following example, we construct two regression models. As a preliminary, you should make scatterplots of the variables under study (here with `plot(age,phraselength)`).
The first model is `phraselength` $= b_0$, i.e., with only a constant intercept. The second model includes the speakers' age as a predictor, i.e. `phraselength` $= b_0 + b_1$`age`. (The intercept is included in this model too, by default, unless suppressed explicitly with `~-1` in the regression formula). The key question here is whether inclusion of a predictor yields a better model, with significantly smaller residuals and significantly higher R^2. The intercept-only model and the linear-regression model are compared with the `anova` function.

```
R> model1.lm<-lm(phraselength~1,data=intra) # only intercept
R> model2.lm<-lm(phraselength~age,data=intra) # with intercept
R> anova(model1.lm,model2.lm) # compare models
Analysis of Variance Table
Model 1:  phraselength ~ 1
Model 2:  phraselength ~ 1 + age
  Res.Df    RSS Df Sum of Sq      F  Pr(>F)
1     79 318.36
2     78 305.42  1     12.94 3.3056 0.07288 .
```

Including the `age` predictor does improve the model a little bit, as indicated by the somewhat smaller residual sums-of-squares (`RSS`). The improvement, however, is too small to be of significance. The linear effect of a speaker's age on his or her average phrase length (in syllables) is not significant.

(`http://en.wikipedia.org/wiki/Poisson_distribution`). Counts of language events, e.g. counts of speech errors or counts of discourse events, tend to follow this non-normal distribution.

`glm` For logistic regression we use function `glm(family=binomial)`, again with a regression formula as an obligatory argument. Logistic regression can be imagined as computing the logit of the hit-rate for each cell, and then regressing these logits on the predictor(s). Here is an annotated example[7]. The response variable `outcome` indicates the death (`0`) or survival (`1`) of 2900 patients in two hospitals.

```
R> ips1525 <- read.table( header=T,sep=","
 + file="e:\\hugo\\emlar\\ipsex1525.txt" )
R> with(ips1525,table(outcome))
  outcome
   0    1
  79 2821
R> 2821/(2821+79) # mean survival rate
[1] 0.9727586
R> # intercept-only logistic-regression model
R> model1.glm <- glm(outcome~1,data=ips1525,family=binomial)

R> summary(model1.glm)
...
Coefficients:
            Estimate  Std.Error z value Pr(>|z|)
(Intercept)   3.5754     0.1141   31.34  <2e-16  ***
R> antilogit # show my function to convert logit to prob
function(x) { exp(x)/(1+exp(x)) }
R> antilogit(3.5754)
[1] 0.9727587
```

Here we see that the intercept-only logistic regression does indeed model the overall survival rate, converted from probability to logit. Next, let's try to improve this model, by including two predictors: first, the `hospital` where the patient was treated, and second, the patient's `condition` at intake, classified as bad (`0`) or good (`1`) .

```
R> model2.glm <- glm(outcome~hospital,
 + data=ips1525,family=binomial)
R> model3.glm <- glm(outcome~hospital*condition,
 + data=ips1525,family=binomial)
R> anova(model1.glm,model2.glm,model3.glm)
Analysis of Deviance Table
Model 1:  outcome ~ 1
Model 2:  outcome ~ hospital
Model 3:  outcome ~ hospital * condition
```

[7]The example is from D.S. Moore & G.P. McCabe (2003) *Introduction to the Practice of Statistics* (4th ed.); New York, Freeman [ISBN 0-7167-9657-0]; Exercise 15.25.

```
  Resid.Df  Resid.Dev   Df  Deviance
1     2899     725.10
2     2898     722.78    1     2.33
3     2896     703.96    2    18.82
```

The deviance among logistic-regression models follows a χ^2 distribution. Hence we can compare models by computing the χ^2 probability of their deviances, for which we use the `pchisq` function. Both model 2 and model 3 are compared here against model 1.

```
R> 1-pchisq(2.33,df=1)
[1] 0.1269019
R> 1-pchisq(18.82,df=2)
[1] 8.190095e-05
```

These results indicate that there is no significant difference among hospitals in their survival rates (model 2, $p > .05$), but there is a significant effect of intake condition on the outcome (model 3, $p < .001$). Of course, you should also inspect the models themselves before drawing conclusions.

7 Further reading

A wealth of useful information is available through the `Help` option in the RGui window. Browse in the FAQ files, the help files, and the manuals, that come with R.

More help is available within R by giving the command `help(...)` with a command or operator in parentheses. If you wish to search helpfiles for a keyword, use `help.search("...")`; this will provide useful pointers to further help information.

There is also a lot more help available on the internet. Here is a brief and personal selection of web resources:

- `http://wiki.r-project.org/rwiki/doku.php`
- `http://maven.smith.edu/~lqian/tutorial/R-intro.pdf`
- `http://mercury.bio.uaf.edu/mercury/splus/splus.html`
- `http://www.math.ilstu.edu/dhkim/Rstuff/Rtutor.html`

— Good luck!