Statistics with R

handout for tutorials at the workshop on
Experimental Methods in Language Acquisition Research (EMLAR XI),
Utrecht, 16 April 2015

Hugo Quené

Utrecht institute of Linguistics OTS, Utrecht University
`www.hugoquene.nl`
`http://www.hugoquene.nl/emlar/Rtutorialemlar11.pdf`

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, χ^2 tests, and regression. We will also show how R produces publication-quality figures.

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 cousin 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.

This tutorial assumes that R is already properly installed on your computer. It is further assumed that the reader has 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.

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 (<- or =) 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 good pancake will result. Provided that the necessary objects (ingredients) are available, the R syntax could 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 provided.

```
MIX batter = flour + milk/2 . # what 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
```

[1]Technically speaking, R is an interpreted language, 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.

```
# 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
# 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). The classes or types of these objects are pre-defined (Sound, Spectrum, Periodicity, etc). R takes the same idea even further: users may create their own *classes* of data objects (e.g. GroupedData) and may create their own methods or relations to work with such objects[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 of various classes: numeric, character (strings of letters, always enclosed in

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

double quotes), or logical (i.e., 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
```

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, as illustrated in the examples below.

```
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 of class `list`:

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

In R we often use a particular complex object, a *data frame*, to hold various data together. A data frame is a complex object like an Excel worksheet or SPSS data sheet. The columns represent variables, and the rows represent single observations — these may be "cases" or sampling units, or single measurements repeated for each sampling unit, 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`. (Windows users must remember to use double backslashes in the file specification string). 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. The `file` argument can be a string specifying a local file, or a `url` to a web-based file, or a call of function `file.choose()` to select a file interactively. Argument `na.strings` specifies the character string(s) that indicate missing values in the input file.

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

[4]For repeated measures analyses, R does not require a multivariate or "wide" layout, with repeated measures for each participant on a single row, as SPSS does. Instead R always uses a univariate or "long" layout, with each measurement on a single row of input. See the `reshape` command (§5 below) to convert between layouts.

```
 file=url("http://www.hugoquene.nl/emlar/intra.bysubj.txt"),
 header=TRUE, na.strings=c("NA","MISSING") )
```

The basic R and extension packages already have many datasets pre-defined (see p.21), for immediate use. To see an overview of these datasets, enter

```
R> data().
```

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. The assignment may also be in the other direction, with symbol `->` [5]. Use spaces or brackets to avoid ambiguities:

```
R> x <- 10 # assignment
R> x < -10 # is x less than -10 ?
[1] FALSE
```

`#` 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.

[5]The equal sign = is also available for assignment. Using it is somewhat dangerous, however, because the equal sign does not specify the direction of assignment explicitly.

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 string are shown.

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

`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" )
```

undo There is *no* `undo` command, nor such a menu option. Save your work regularly. If in doubt, work with scratch copies of your data sets.

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.

In the following example, variable `x` contains 30 numbers, but 3 of these are `NA`. Notice that the output of `is.na` is the input of `table`.

```
R> # is.na() returns TRUE/FALSE for each element of 'x'.
R> # table() summarizes categorical data
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
```

Subselection can also be achieved by using the function `subset(data, subset, select)`. The first argument is the input data (set), the second argument is the selector condition, and the optional third argument indicates which columns of a data frame should be kept in the output.

```
R> subset( nlspkr, subset=(!isold & region=="W") )
```

This command selects rows from data frame `nlspkr` (see §2.3) corresponding to speakers who are not old (i.e. who are young) and who are from the West region.

3.3 split and merge

There are useful functions available to split and merge data frames. First we create two example data frames. The first data frame has a list of vowels, with a phonological feature for each vowel, and the average F2 frequency for male speakers[6]. The second data frame has a partially overlapping list of vowels, with key words by John

[6]Taken from: G.E. Peterson & H.L. Barney (1952), Control methods in a study of the vowels, *J. Acoust. Soc. Am.* 24(2), 175-184, Table II.

Wells[7].

```
R> vowelsymb <- c("i","I","e","E","ae","A","V","o","U","u","@")
R> v1df <- data.frame( vowel=vowelsymb,
 feat=factor( c(rep("front",5),rep("back",5),"central" )),
 F2=c(2290,1990,NA,1840,1720, 1090,1190,NA,1020,870,NA) )
R> v2df <- data.frame( vowel=vowelsymb[1:10],
 word=c("fleece","kit","face","dress","trap",
 "lot","strut","goat","foot","goose") )
```

`split` This command divides the data in the first argument (column `F2` in data frame `v1df`) into the groups defined by the second argument.

```
R> with(v1df, split(F2,feat) ) -> v1list
```

This is particularly helpful in combination with `lapply` to apply a function to these grouped data, e.g. to compute the mean of F2 for each vowel category separately:

```
R> with(v1df, lapply( split(F2,feat), mean, na.rm=T ) )
```

Also see `unsplit`.

`merge` This command merges two data frames, by common columns. Specify argument `all=TRUE` if you want to include non-matching rows in the output, with NA's in the appropriate columns.

```
R> m1 <- merge(v1df,v2df)
```

The resulting merged data frame is also sorted on the common columns, unless argument `sort=FALSE`.

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.

[7] From `http://en.wikipedia.org/wiki/Lexical_set`.

`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[8]:

```
R> x<-rnorm(100); hist(x); rug(x,col=4)
```

`boxplot` This yields a boxplot summary[9] of one variable. You can also specify the dependent and independent variable, with argument `dv~iv` [10]; this will produce multiple boxplots for the dependent variable, broken down by the independent variable(s). 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 would be distributed normally. If the variable is indeed distributed normally, then the data should fall on an approximately straight line. Deviations of this line indicate deviations from normality. You can also add the expected 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. Related functions are `sd` to compute standard deviation, `cov` to compute the covariance between two variables, and `cor` to compute their correlation.

`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.

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

[9]Originally proposed by J.W. Tukey (1977), *Exploratory Data Analysis* (Reading, MA: Addison-Wesley), §2C, ISBN 0-201-07616-0.

[10]or `dv~iv1+iv2+...`, etc.

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

In the last command above, we have programmed a new function `valid.n`, using standard functions provided with R.

The *lattice* package (see §8) contains additional functions for higher-order data visualization: `xyplot`, `histogram`, `densityplot`, `cloud`, and more. For more information, enter the command `help(lattice)` (see §9).

Now that we have obtained some insightful figures, we might want 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 PDF format (extension `pdf`) are easy to include in LaTeX documents, and are best for publication. Figures in PNG format (extension `png`) are easy to include in LaTeX and web documents.

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 in the boxplot example in §4 above.

```
 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, normal distrib
R> y2 <- rnorm( n=100, mean=0.2 )
 R> x <- rep( 1:2, each=100/2 ) # to use as IV
```

In a one-sample test, the mean is compared against an expected mean, with
`R> t.test( y2, 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
```

This could also be achieved by specifying the dependent and independent variables in a formula:

```
R> t.test( y1 ~ x ) # equivalent
```

The single equal-sign is used as the assignment operator (footnote 5), and it 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. (Johnson, 2008, illustrates more complex analyses of variance having repeated measures; see also §7 and §9 below). In the example below we create a response variable `aa` which is not normally distributed[11] (check with `hist`, `qqnorm`, etc).

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

`reshape` If you need to perform a Repeated Measures (within-subjects) analysis of variance (RM-ANOVA) in SPSS, your data have to be in "wide" data layout, with all observations from one subject on a single data line. R on the other hand uses the "long" data layout, with one observation per line, and with all descriptors of that observation repeated for each line. There is a convenient command `reshape` to convert data between the wide layout (of SPSS RM-ANOVA) and the long layout (of R). To illustrate, first we read a wide data

[11]The dependent variable `aa` follows a Poisson distribution, with λ varying between conditions of `x1`. "Poisson distributions are often used to model the occurrence of rare events" [R. Rosenthal & R.L. Rosnow (2008), *Essentials of Behavioral Research: Methods and data analysis* (3rd ed., Boston: McGraw-Hill), p.305], e.g. accidents or speech errors by healthy adults.

set:

```
R> widedata <- read.table(
 file=url("http://www.hugoquene.nl/emlar/widedata.txt"),
 header=T)
```

The wide data show the subject id, between-subject group, and three within-subject observations, for 6 subjects (with leading row numbers):

```
R> head(widedata)
 subject group item1 item2 item3
1 1 1 2 3 4
2 2 1 3 4 6
3 3 1 1 3 6
4 4 2 2 4 5
5 5 2 4 5 6
6 6 2 2 5 7
```

These data are then reshaped to long layout with the following command:

```
R> longdata <- reshape( widedata, direction="long",
 varying=c("item1","item2","item3"),
 timevar="item", times=c("1","2","3"),
 v.names="resp", idvar="subject")
```

The observations from columns `varying` are collected into a new column `v.names`, using identifiers in column `idvar`. The information contained in the column names of `varying` is stored in a new column `timevar`, using the values `times`. Inspect the two data frames to verify this.

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, e.g. by entering

```
R> 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` or `~0` 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=nlspkr ) # only intercept
R> model2.lm <- lm( phraselength~age, data=nlspkr) # with intercept
R> anova(model1.lm,model2.lm) # compare models
Analysis of Variance Table
  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.

`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[12]. The response variable `outcome` indicates the death (`0`) or survival (`1`) of 2900 patients in two hospitals.

```
R> ips1525 <- read.table(
 file=url("http://www.hugoquene.nl/emlar/ipsex1525.txt"),
 header=T, sep=",")
R> with(ips1525,table(outcome))
  outcome
   0    1
```

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

```
  79 2821
R> 2821/(2821+79) # survival rate
[1] 0.9727586
R> log(2821/79) # log of odds of survival
[1] 3.575399
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  ***
```

The intercept coefficient of the intercept-only logistic regression model equals the overall log odds of survival; this intercept is significantly different from zero[13]. Thus the intercept-only logistic regression model captures the overall log odds of survival. 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)
```

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> anova(model1.glm, model2.glm, model3.glm, test="Chisq")

   Resid.Df  Resid.Dev  Df  Deviance    Pr(>Chi)
1      2899     725.10
2      2898     722.78   1    2.3253      0.1273
```

[13]The log of odds is zero, if the odds of survival are 1 : 1 or 50%.

```
3       2896      703.96   2   18.8222  0.00008181 ***
```

The 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[14] on the outcome (model 3, $p < .001$). Of course, you should also inspect the models themselves before drawing conclusions.

7 Mixed-effects modeling

Many language (acquisition) studies are based on samples of two random factors: a sample of participants (subjects) and a sample of language items (words, sentences, texts). The two random factors are *crossed*, i.e., each item is presented to each participant — often only once, so that a subject does not respond to the same item repeatedly in multiple conditions. The analysis methods shown above (`aov`, `lm`, `glm`) all fail to acknowledge this particular structure in the random part of the design. They include a single random factor (named `Residual`) that aggregates all random effects.

A relatively new and easy method is to use *mixed-effects modeling*, which may be done in R by using the `lmer` command. Key advantages of this method[15] are (a) it allows multiple random factors, crossed and/or nested, (b) it does not require homogeneity of variance, (c) it is robust against missing data. Hence mixed-effects modeling is quickly gaining in popularity.

For mixed-effects modeling, you need to install two add-on packages to R, named `lme4` and `languageR` [16]. For more information on packages, see §8 below. After activation of these packages, we can simply perform a mixed-effects analysis. First, we read in an example dataset (from Quené & Van den Bergh, 2008) in long data layout:

```
R> x24 <- read.table(
```

[14] or of the *interaction* of hospital and intake condition. As it turns out, however, the interaction is not significant: the effect of intake condition is the same among hospitals.

[15] H. Quené & H. van den Bergh (2004), On multi-level modeling of data from repeated measures designs: A tutorial, *Speech Communication*, 43 (1–2), 103–121; idem (2008), Examples of mixed-effects modeling with crossed random effects and with binomial data, *J. Memory and Language*, 59 (4), 413–425.

[16] H.R. Baayen, D.J. Davidson & D.M. Bates (2008), Mixed-effects modeling with crossed random effects for subjects and items, *J. Memory and Language*, 59 (4), 390–412.

```
 file=url("http://www.hugoquene.nl/emlar/x24r2.txt"),
 header=T )
```
These fictitious responses were provided by 24 subjects, for 36 items, in 3 conditions, with rotation of items over conditions. This rotation may be inspected for a small subset of the data frame:
```
R> with( subset(x24, subj<=3&item<=6),
 table(subj,item,cond)) # output not shown
```
Next, we need to specify that cond is a categorical factor, and not a continuous predictor. In addition, we specify the levels of the factor, we specify its contrasts, and indicate that the second level is the baseline or reference level.
```
R> x24$cond <- as.factor(x24$cond)
R> contrasts(x24$cond) <-
 contr.treatment( c(".A",".B",".C"), base=2 )
```

After these preliminaries we can estimate an appropriate mixed-effects model in a single command. The estimated model is also stored as an object, and a summary is displayed.
```
R> summary( x24.m1 <- lmer(resp ~ 1+cond+(1|subj)+(1|item),
                           data=x24, REML=FALSE) )

Linear mixed model fit by maximum likelihood
Formula: resp ~ 1 + cond + (1 | subj) + (1 | item)
   Data: x24
  AIC  BIC logLik deviance REMLdev
 2047 2075  -1017     2035    2045
Random effects:
 Groups   Name        Variance Std.Dev.
 item     (Intercept) 0.25789  0.50783
 subj     (Intercept) 0.28913  0.53771
 Residual             0.51033  0.71437
Number of obs: 864, groups: item, 36; subj, 24

Fixed effects:
            Estimate Std. Error t value
(Intercept)  0.04569    0.14485   0.315
cond.A       0.17037    0.05953   2.862
cond.C      -0.23696    0.05953  -3.980
```

The output correctly shows that there are two unrelated random effects, plus unexplained residual variance. Each response is now modeled as a unique combination of the intercept (mean of baseline condition B), item effect, subject effect, condition effect, and residual. The average response in the baseline condition B is 0.046 units. Responses in condition A are 0.170 units higher than baseline, and in condition C they are −0.237 units higher than baseline, i.e. 0.237 units lower.

For reasons not discussed here[17], the significance levels of the fixed effects are not reported in the output of `lmer`. There are several solutions to obtain these significance levels.

The most conservative option[18] is to use the critical t value associated with the random effect that has the fewest levels (here `subject`), corrected for the number of fixed coefficients (here 3). If a fixed effect is significant by this very conservative criterion, then it will also be significant by any other criterion that is less conservative and more liberal.

```
R> qt( p=1-.05/2, df=24-3 ) # critical value t*, alpha=.05, two-sided
[1] 2.079614
```

Comparison of the fixed effects with this critical $t^* = 2.08$ shows that both conditions A and C differ significantly from the baseline condition B.

A second option is to estimate 95% confidence intervals for all coefficients in the resulting mixed-effects model. This can be time-consuming, as the mixed-effects model will be re-fit many times on slightly varying datasets.

```
R> print(x24.mi.ci <- confint( x24.m1, method="boot", nsim=250 ))

                          2.5 %     97.5 %
sd_(Intercept)|item   0.3690334  0.6447299
sd_(Intercept)|subj   0.3672621  0.6712502
sigma                 0.6770225  0.7492705
(Intercept)          -0.2371983  0.3091467
cond.A                0.0448017  0.2805097
cond.C               -0.3544133 -0.1259679
```

[17]See Frequently Asked Questions about R, Question 7.35, at `http://cran.r-project.org/doc/FAQ/R-FAQ.html`

[18]J.J. Hox (2010). *Multilevel Analysis: Techniques and Applications* (2nd ed.). Mahwah, NJ: Lawrence Erlbaum.

As the interval for `cond.A` is entirely positive, we may conclude with 95% confidence that condition A yields higher scores than the baseline condition B, and mutatis mutandis that condition C yields lower scores than condition B.

8 Packages

Packages are user-provided extensions to the basic R system (comparable to an "add-on" for Mozilla web browsers). Packages may contain custom datasets, additional functions, re-formulations of existing functions, and more. There are by now thousands of useful packages extending R. A package can be installed by entering

```
R> install.packages("languageR")
R> require(languageR) # including recursively required packages
```

If a package in turn requires other packages, these are also installed and loaded. Some useful packages are the following:

datasets	A wide variety of datasets, for exploration and education. By default, this package is already loaded when R starts.
foreign	Functions for reading and writing data stored by statistical packages such as Minitab, S, SAS, SPSS, Stata, Systat, and for reading CSV files (comma separated values) created by Microsoft Excel, and for reading and writing dBase files.
hqmisc	Various convenience functions and a dataset, by the present author.
lattice	Trellis graphics for R. The functions provide a powerful, elegant and flexible high-level data visualization system, using Trellis graphics, with an emphasis on multivariate data.
MASS	Functions and datasets to support W.N. Venables and B.D. Ripley (2002), *Modern Applied Statistics with S* (4th ed., Berlin: Springer) [ISBN 0-387-95457-0, `http://www.stats.ox.ac.uk/pub/MASS4/`].
languageR	Datasets and functions accompanying H.R. Baayen (2008), *Analyzing Linguistic Data: A practical introduction to statistics using R* (Cambridge: Cambridge Univ Press) [ISBN 978-0-521-70918-7].
nnet	Software for neural networks, includes multinomial log-linear models.

Packages are stored on a so-called *repository*; the CRAN repository is the most important one. You should use a nearby mirror site of the CRAN repository, by giving the command `chooseCRANmirror()`. R remembers the chosen mirror site over multiple sessions.

9 Further reading

A wealth of useful documentation 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(...)` or `?...` with a command or operator as argument. If you wish to search the helpfiles for a keyword, use `help.search("...")` or `??...`; this will provide useful pointers to further help information.

There is also a lot more help available on the internet, in particular from the R project website `http://www.r-project.org`, see under Documentation. A few other useful web resources are:

- Quick-R, `http://statmethods.net`
- `http://math.illinoisstate.edu/dhkim/Rstuff/Rtutor.html`

The following three books on using R and on its applications in linguistics are highly recommended:

- H.R. Baayen (2008), *Analyzing Linguistic Data: A practical introduction to statistics using R* (Cambridge: Cambridge Univ Press), ISBN 978-0-521-70918-7.
- K. Johnson (2008). *Quantitative Methods in Linguistics* (Malden, MA: Blackwell), ISBN 978-1-4051-4425-4.
- J. Adler (2010). *R in a Nutshell* (Sebastopol, CA: O'Reilly), ISBN 978-0-596-8-170-0.

Happy analyses!

Index