Tartu Ülikool
Psühholoogia Instituut

Üldistatud lineaarsed mudelid sotsiaalteadustes: praktikum
SHPH.00.001
Kevadsemester 2010

Kenn Konstabel

Üldistatud lineaarsed mudelid (ÜLM) on lineaarse regressioonimudeli tänapäeval laialdaselt kasutatav üldistus, mis võtab ühtsesse lihtsasse kasutusskeemi kokku mudelid, mida traditsiooniliselt tuntakse paljude erinevate nimede all (nt regressioon-, dispersioon- ja kovariatsioonanalüüs, logistiline ja Poissoni regressioon). Selline üldistus võimaldab erinevatest mudelitest paremini aru saada ning neid paindlikumalt rakendada. ÜLM ideestik on laialdaselt kasutusel paljudes erinevates teadusvaldkondades (sh sotsiaalteadused, psühholoogia, liikumis- ja sporditeadused jne) ning selle tundmine on nende valdkondade uurimistulemustest aru saamiseks üha olulisem. Kursusel käsitletakse ÜLM kasutamist programmiga R. Järgnevas kursuse materjalide kogumikus on (a) sissejuhatav tekst R-i kasutamisest, (b) praktikumijuhendid, ja (c) kursuse lisamaterjalid. Kõigis tekstides tuleks `courier` *kirjas olev osa ise R-is järele proovida – praktikumid on sel viisil ka iseseisvalt läbi tehtavad.*

Võtmesõnad: üldistatud lineaarsed mudelid, regressioonanalüüs, sotsiaalteadused

Üldistatud lineaarsed mudelid sotsiaalteadustes
Praktikum (SHPH.00.001, kevad 2010)

Introduction to R

Kenn Konstabel (nek@psych.ut.ee)

1. How to use this document

Parts of it can be read like ordinary text – no special instructions there. Everything in `Courier font` can be typed into R: type a line, press "enter" and see what happens. These parts are not meant to be understandable without seeing the R output they produce. If in hurry, you can copy and paste many lines of code to R; then scroll back and examine the output.

There is no need to memorize everything; rather, try to figure out how each line of code works and take a look at it again when you need it.

Some parts may seem too abstract. The section on data structures, for instance, however useful, is quite long and can be boring. You can safely skip some of the most boring parts and return to it later.

There are two kinds of exercises here: (1) typing or copying all of the `code` to R, trying to figure out how it works and/or experimenting with it on your own), and (2) some real exercises in the last section.

2. Overview

R is a language and environment for data handling and statistics. Using R may seem difficult at first sight: when you start an ordinary statistical package, you will see a nice mouse-clickable graphical user interface (GUI), but R will be just waiting for you telling it what to do. On the other hand, with a little experience, you can get results more efficiently without a GUI: computing a correlation between X and Y takes at least a dozen mouse clicks in a GUI (and you have to click in the appropriate place – otherwise you'll get something else!), but it's just typing `cor(X,Y)` in R. But lack of GUI is not the only thing that differentiates R from other statistical packages; moreover, if you really want, you can actually use a GUI with R (see section 11). Among R's best features are:

- Besides being a statistical package, R is also a high-level programming language, which makes automating things a lot easier. What is more, the S programming language was specially designed for use with data manipulation and statistics.
- You can easily record your analyses to repeat and/or check them later. It is much more difficult to record menu clicks and checkbox ticks in a GUI.
- Easy recording also means easier communication and asking for help. You can send your lines of code by e-mail to a colleague, or a list, and the reader can just copy and paste it to R.
- Technical details of how R does this or that analysis are accessible to everyone (R is an open-source program); nothing is based on secret proprietary algorithms.
- R functions can easily be customized to do **exactly** what you need.
- A large and often exceptionally helpful community of users. You can, for example, get good help from the list R-help@r-project.org literally in minutes. (Please read the posting guide at http://www.r-project.org/posting-guide.html before posting -- this will really help you writing a good question that is much more likely to be answered!)
- A large number of add-on packages (as of today, 1455 are available on the official R webpage). Modern statistical techniques will often first appear as an R package, and only thereafter, many years later, if at all, be available in SAS or SPSS.

- Excellent print-quality graphics.

Useful links

www.r-project.org	R homepage; from the menu at left pane: Documentation → Manuals (official documentation) Documentation → Other (a collection of links to other documents on R)
cran.r-project.org	Comprehensive R Archive Network – place to download R; select a mirror from cran.r-project.org/mirrors.html From the left pane menu, try: Documentation → Contributed (a number of freely available introductory and not so introductory books and other documents on R; these are often easier to grasp than the official documentation which can be quite technical)
www.jstatsoft.org	Journal of Statistical Software, online journal that frequently publishes papers using R; there is a special issue (vol. 20) on "Psychometrics in R" (www.jstatsoft.org/v20)
personality-project.org/r	Using R for psychological research: a guide by William Revelle

search.r-project.org	Searching R-related sites; you can also access this from R by typing
	`RsiteSearch("things-to-look-for")`
	`# results will open in your favorite browser`

wiki.r-project.org	R Wiki (including a growing collection of larger guides and smaller tips, a graphics gallery, pages on some add-on packages, etc)

3. Downloading and installing R and add-on packages

Installing R

Open CRAN (see previous section for a link) in your favorite web browser; the first thing you see on the front page is a grey heading "Download and Install R". For now, you need the first section, i.e., a "binary distribution", not a "source code". Select your favorite OS (Linux, MacOS, or Windows), and follow the further instructions. For Windows, you'll have a further selection between "base" and "contrib" – select "base", download the file named R-2.7.0-win32.exe (version number in the middle may change), and run it.

In Windows, you'll have a few choices to make.

1. In "Select components" screen, there's no urgent need to change anything unless you want to save a few MBs of disk space.
2. In the next screen, "Startup options", select "Yes (customized startup)".
3. The next screen asks you to choose between MDI and SDI interface. The difference is whether plots and help pages will open as "sub-windows" in a large window (MDI) or as separate windows that can be seen on the taskbar (SDI). Most people prefer MDI, which is the default.
4. Then you're asked about how you want help pages to be shown by default. The default choice for Windows, CHM, is in my experience the worst of all: it is slower than plain text and less convenient than HTML. I recommend choosing "plain text"; later you can access HTML help from a drop-down menu.
5. You can safely just click "next" on the following screens. The last one asks about whether to create Start menu item and desktop icon (you'll probably

need the first but the second is more of a matter of taste), and whether to create registry entries. On a computer where you have no administrator rights, you can't change Windows registry but you can still install R to a "My documents" folder or something alike.

6. Now you can run R and use it.

Tell us if you need advice on installing R on Mac or Linux; some instructions are there on the download pages, and there is also a manual R installation and administration on CRAN (Documentation → Manuals).

Installing add-on packages

You can see a list of available packages – with a very short description – on CRAN (from the left pane, select Software → Packages). Clicking on the package name, you'll see a slightly longer description plus links to the package itself and its reference manual, which gives you an idea about what's in there. You'll also see the name of package maintainer and his or her e-mail; in some cases, there is also a link to a webpage.

For downloading and installing a package (while connected to the Internet), type

```
install.packages("package-name")
```

If you do this for the first time during a session, you'll be asked to select a CRAN mirror; it is a good idea to choose the one nearest to where you are. During the workshop, one of the packages we'll certainly need is called `psych`; so why not install it right now. If you later want to read data from Excel, you might also install a package called `RODBC`. – For using a package, you also need to load it:

```
library(psych)   # no need to quote the package name here.
require(psych)   # equivalent to previous
```

"Task views" are large thematic collections of packages; see CRAN→Task Views on the left pane of CRAN. There is, for example, a task view on psychometrics that you might want to take a look at. You'll find a long list of psychometrics-related packages with short explanations; at the end of the page, all packages in a view are listed alphabetically, and divided into "core" and "not so core" packages. (Take a look at some of the other task views, too, e.g., SocialSciences and Multivariate.)

You can install all packages in a task view together; for that, you'll first need to install another package, `ctv` (CRAN Task Views):

```
install.packages("ctv")
library(ctv)
install.views("Psychometrics")
install.views("Psychometrics", coreOnly=TRUE)
```

Getting help

Don't try to memorize how every function works! It is a good idea to look at the help page of each function you haven't used before (in fact, it is often impossible to use a function without doing so).

```
?some.function
help(some.function)
```

The help pages may look not very helpful, and they aren't if you have no idea what to do. But if you already know the function you're going to use, its help page gives you indispensable information: which arguments you should supply, how are the arguments named, what should be their type, and what the result exactly is.

4. First step: using R as a calculator

R evaluates immediately what you type in, and gives you the result. Try the following: (don't type in the "greater than" sign: this is just what R tells you to say it's ready to take orders; # is a comment sign – everything after it is ignored by the interpreter).

```
> 1+1          # should be 2 :-)
> sin(1)       # 0.841471
> 2*2          # * means multiplication
> 4^2          # ^ means "to the power of"
```

Besides these simple calculations, you can **assign** values to variables with the <- operator[1]:

```
a <- 1+1       # a gets the value from the right side;
               # nothing shown on the screen
3 + 4 -> b     # b gets the value from the left side
c <-(b+a)/2
print(c)       # shows the value of c,
               # same as just typing c
a              # 2
a+1            # 3
b+c            # 11.5
```

There are a few restrictions to the variable names: they can't contain spaces, arithmetical operators (*, /, +, -, etc) or logical operators (&, |, etc); the name cannot begin with a number. Even these restrictions can be overcome if you really need it, but you probably don't[2]. R makes a difference between upper- and lowercase letters, so `r` and `R` are different variables.

5. Basic data structures

Mode of an object

Numeric variables are just one type that R can handle: other types are "logical" (truth values), "character" (character strings), and "complex" (for complex numbers). You can test for an object's mode by using the **function** `mode` (type "mode", and include the object or its name in parentheses)

```
mode(1)                    # numeric
mode(1+1)                  # still numeric!
mode(a)
mode("Hello, World!")      # character
mode(TRUE)                 # logical
```

The results of all logical operations are of mode "logical" and can have three possible values: `TRUE`, `FALSE`, or … `NA`. NA stands for "not available" and is the result when a logical operation cannot be performed because of missing data (numeric and character variables can also have NA values).

```
a < b
c>=a           # is c greater-than-or-equal-to a?
a == b         # is a equal to b? (note the double equality
               # sign!)
```

[1] It is also possible to use equality sign (=) for assignment, and some people find it more intuitive. However, the primary function of "=" in R is to naming the elements of lists or vectors, so using it in a different function may be confusing. In addition, a statement like x=x+1 does look strange from a common sense point of view (there's no way x can equal x+1), although it is a technically correct way to assign a value to x.

[2] To use "non-standard" object names, use the name in "quotation marks"; but then you'll have to use the function **get** to get the object's value.

```
!(c>a)          # is it NOT true that c>a?
                # - exclamation sign means negation
mode(a<b)
d <- (a<b)
d
mode(d)
d <- NA

a < d           # do we know the result?
mode(a<d)       # logical
```

Vectors

All "variables" considered so far are **vectors:** you can store many values in a single object. In other words, vectors have a length, which has been 1 so far, but can be different.

```
length(a)       # 1
length(1+1)==1# TRUE
length(a>b)     # jne.
```

There are many ways to create vectors with length>1. You will do this every time you read in a data file (we'll see about that later). To create a sequence from 1 to 10, you can just type:

```
1:10            # integers from 1 to 10
c(1,7,3)        # creates a vector with values 1, 7 and 3
a <- 1:10
a
length(a)       # 10
```

A vector can only store values of the same type: for example, it can be all numbers, all truth values, or all character strings, but you cannot mix these types within the same vector. In fact, if you try to do this, all elements will be converted to a single type and the mode of the result is sometimes unpredictable:

```
c("Hello!", 007, TRUE)        # all elements converted to
                              # "character"
c(1, TRUE, 0, FALSE)          # all elements converted to
                              # "numeric"
```

With vectors containing more than a single value, you can perform many operations more efficiently:

```
a+1             # 1 is added to all elements of a
                # remember that a was 1:10
b <- a+1        # now the result is assigned to b
mean(a)         # average
mean(a+1) == mean(a)+1        # try to predict the result
sd(a)           # standard deviation
b<5             # the result is now a vector of length 10
                # saying for every element of b, whether it is
                # smaller than 5
```

You can access single (or more) values of a vector by using an **index** or **indices** in square brackets:

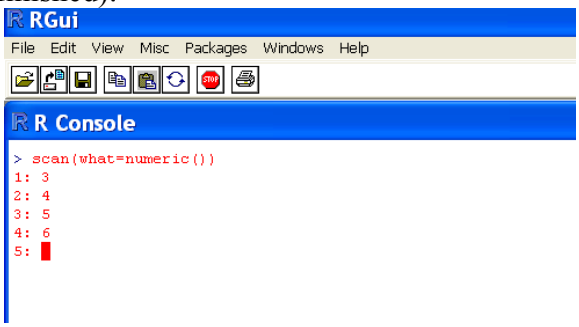
```
a[1]            # the first value of a
a[1:5]          # the first 5 values of a
a[c(1,3,7)]     # 1st, 3rd, and 7th value of a
```

You can also use an index vector of truth values; in that case, the index vector has to have the same length as the vector you're trying to make a subset of.

```
a[b<6]          # returns the elements of a for which the
                # corresponding element of b is smaller than 6
```

How might this be useful? Suppose we have measured the length of 6 trees, and have also recorded the species for each tree. This time, we'll use the function scan to enter the data (press "enter" twice after you've finished):

```
l <- scan(what=numeric())
          # type in 6
          numbers
t <- scan(what=character())
oak
oak
oak
pine
pine
pine
```



Now you can use a logical index vector to get the lengths of all oak trees:

```
l[t=="oak"]
t[l>4]        # gives species for those trees longer than 4
              # meters
```

List

List is similar to vector in that it can contain one or multiple values, but the difference is that each value can be of different type (and possibly another list). You can create a list by using the function `list()`:

```
Peter <- list(
              name = "Peter",
              children = c("Bob", "Sarah", "Vera"),
              favorite.movies = c("Anna Karenina",
              "Batman"),
              height = 186.25,
              is.married = TRUE,
              favorite.R.function = mean)
Peter         # to see how this is printed
              # Peter's favorite R function is printed as
              # R internal code for the function mean
```

You can access individual list elements by their name or number in a list. For getting just one element, you can use `$` (dollar sign) or `[[` (double square brackets):

```
Peter$name
Peter$children
Peter$children[1]           # to get 1st element of a vector
Peter[["name"]]
Peter[[1]]
Peter$favorite.R.function(1:5) # equivalent to mean(1:5)
```

List elements are often named (like in the present case) but they needn't be. With `$` operator, you can get only named elements, whereas with `[[` you can also get an element by its index (position in a list). So `Peter$name` is equivalent to `Peter[[1]]` or `Peter[["name"]]`, or if you have `index <- "name"`, then `Peter[[index]]` will do the same.

To get several list items at once, you can use [(single square brackets); the result will be a list, even if you only specify a single component:

```
Peter[1:2]
Peter[c("name", "favorite.R.function")]
Peter["name"] # note the difference from Peter[["name"]]
names(Peter)  # will give you the names of all elements
```

Lists are often used in output of complicated functions. For example, the function `cor.test` which computes a bivariate correlation and tests for its significance, will output a list whose components are, for example, estimated correlation, t statistic and its degrees of freedom, a character vector specifying whether one- or two-sided t-test was used, and confidence interval for the correlation (a vector of two components: lower and upper CI, plus an attribute specifying the confidence level used). A function's output is usually printed nicely on the screen, including some explanatory text, and excluding the components that have been computed but are rarely used. It is, however, useful to know it is actually a list and you can (if needed) extract its individual components. Let's try to compute a correlation between two vectors. We'll first use the function `rnorm` to create some normally distributed data:

```
x <- rnorm(100) # 100 normally distributed observations
y <- x + rnorm(100) # to make x and y correlated, we
              # add x to another random vector
cor.test(x, y)
# you will see something like this:
#
#        Pearson's product-moment correlation
#
# data:  x and y
# t = 11.0172, df = 98, p-value < 2.2e-16
# alternative hypothesis: true correlation is not equal to 0
# 95 percent confidence interval:
# 0.6410765 0.8203895
# sample estimates:
#     cor
# 0.743832
```

The actual values on your computer will be somewhat different because x and y are randomly generated and should be different each time. So when you generate new x and y with the same formulas, you'll get different values again. The above output is probably easy to understand (except perhaps notation used for p-value: this is a way to say $2.2 \cdot 10^{-16}$). This is because, before being printed, it is handled by another function that adds some explanatory text in appropriate places. You wouldn't like to know how the raw output would look like, but let's see it anyway:

```
my.test <- cor.test(x,y)     # NB! The test is performed
              # again but nothing is shown on the screen
              # - the result is assigned to `my.test`
unclass(my.test)
```

The "class" attribute of an object (if existent) will specify the way it will be printed or plotted (and the behavior of many other functions); if we take that away with `unclass`, it will be printed as an ordinary list. In many cases, a function's complete output will be too long to examine it; so it is safer to use `str()` to examine the structure of a function's output. (Try it with `my.test`.)

You can apply a function to all elements of a list by functions `lapply` and `sapply` (the former always gives its output as a list, whereas `sapply` tries to simplify it, if possible, to a vector). So we can get mode of each element of Peter by …

```
sapply(Peter, mode)
# how to get the length of each element?
```

Matrix

In R, a matrix is, technically, a vector wrapped in two dimensions. (It is also possible to create higher-dimensional matrices, but these are called arrays.) So all elements of a matrix will be of the same type, usually numeric or (sometimes) logical.

```
matrix(1:10, ncol = 5, nrow=2)
matrix(1:10, ncol=5)   # nrow can be inferred by R
```

In creating matrices, vectors are by default wrapped columnwise; if you don't want this, you can use an additional `byrow=TRUE` argument:

```
matrix(1:10, ncol = 5, byrow=TRUE)
```

Like a vector, matrix has a **length** (number of elements in it), but its "**dimension"** – number of rows and columns – is more informative:

```
length(matrix(1:10, ncol=5))   # 10
length(matrix(1:10, nrow=5))   # still 10
dim(matrix(1:10, ncol=5))      # 2 rows, 5 columns
dim(matrix(1:10, nrow=5))      # 5 rows, 2 columns
```

As with vectors, subsetting a matrix is done by square brackets, but the difference is that you would usually specify two numbers, row and column indices, separated by a comma. You can specify an entire row or column by leaving the other index empty (but the comma should still be there).

```
print(A <- matrix(1:16, 4))
A[3,2]      # 3rd row, 2nd column
A[3,1:2]    # 3rd row, 1st and 2nd columns
A[3,]       # the entire 3rd row
A[,1:2]     # columns 1 and 2
```

Note that if the result is a single row or column, it is by default converted to a vector: its dimensions are lost and there is no difference between a row and a column vector (in fact, both would be column vectors if you'd try to use them in a matrix operation). This can be avoided by adding a drop=FALSE parameter after the indices.

```
A[3,,drop=FALSE]   # 3rd row as a 1 by 3 matrix
                   # notice two commas before drop:
                   # one for missing index, another for
                   # separating the named parameter
A[,2,drop=FALSE]
help("[") # to get additional help on matrix subsetting
```

You can also use another matrix ("index matrix") for getting a subset of another matrix. Both matrices should be of the same size, and the index matrix should be of mode **logical** (TRUE meaning "include this cell", and FALSE meaning "throw it away"). This can be used, for example, to get an upper or lower triangle of a square matrix, with functions `upper.tri` and `lower.tri`.

```
upper.tri(A)
lower.tri(A)
?upper.tri
A[upper.tri(A)]   # gets everyting above the main diagonal;
                  # the result is a vector
```

`diag` will extract or set the diagonal elements of a matrix. There are also other uses of diag: if the supplied argument is a vector (instead of a matrix), it will create a matrix with this vector as a diagonal (off-diagonal elements will be zero). If you just supply a number (vector of length 1), then an identity matrix of this size will be created).

```
diag(A)
diag(A) <- 100; print(A)
diag(A) <- c(9,9,0,0); print(A)
diag(1:5)
diag(4)
```

Basic matrix operations (multiplication, transpose, inverse, and, of course, many more) are easy to do with R; these functions will work with matrices and vectors (which are treated as column matrices), but, in general, not with data frames even if they contain only numeric data.

```
t(A)             # transpose of A
t(A) %*% A       # matrix product
t(A) * A         # elementwise multiplication
```

For inverse, we'll first need a non-singular matrix; let's hope we'll get it by putting A's elements in a random order:

```
sample(1:9)
sample(1:9)    # it's different this time
A <- matrix(sample(1:9), ncol=3, nrow=3)
A
solve(A)
```

Note that solve will compute inverse if you give it just one argument; if you provide two arguments (A and B) to `solve`, it will solve a system of linear equations Ax=b. Let's try to solve a very simple system, 2x+y=3; x+y=1

```
(A <- matrix(c(2,1,1,1), byrow=TRUE))
(b <- c(3,1))
solve(A,b)   # you've probably already solved it
```

Row and column sums and means of a matrix can be computed with `rowSums`, `colSums`, `rowMeans`, and `colMeans`; any function can be applied to rows or columns by `apply`:

```
?apply
apply(A, 1, sum)  # equivalent to rowSums
apply(A, 2, mean) # equivalent to colMeans
apply(A, 1, max)  # maximal values of each row
apply(A, 2, sd)   # standard deviations of each column
# exercise: can you compute row and column sums of A
#    using matrix multiplication?
```

Data frame

Technically, a data frame is a list of any number of vectors of any type (mode), all of the same length. This corresponds to the data table in ordinary statistical packages; the data that you read in will typically be represented as a data frame in R. Let's make a simple useless data frame:

```
D <- data.frame( name = c("John", "Mary", "Peter"),
                   sex = c("male", "female", "male"),
                height = c(171, 182, 205),
               plays.tennis = c(TRUE, TRUE, FALSE))
D   # or print(D) – to see what the result looks like
class(D)    # "data.frame"
length(D)   # 4 (number of vectors in the list)
dim(D)      # 3 rows, 4 columns
```

As you can see, data frame looks like a matrix (a rectangular table). It is different from a matrix, though, because its contents can be heterogeneous (like in the present case), and consequently most matrix operations cannot be performed. If you're curious, try transposing our data frame D using `t()`. What happens is that D is first converted (silently) to a matrix (using `as.matrix`), and then transposed; all of its elements become of one type (mode): in this case, character.

Elements of a data frame can be accessed (and changed) like those of a matrix, or those of a list:

```
D[1:2]        # the first elemens of D as a list
D[1,1]        # John
D[,3]         # the entire third column: vector of heights
D[1,]         # the entire first row
D$height      # another way to get the third column
D[[3]]        # ... yet another way
D[["height"]][1]
```

```
D[1, "height"] # to get John's height, == D[1,1]
D$height[3]   # height of the third person
D$height[2] <- 132
D$height <- 1:3*100   # replacing an entire column
              # replacement must be of the right length!
D$weight <- c(80,70,100)  # adding another column
```

With indices, you can also make subsets of your data frame, i.e., selecting cases and/or columns that meet a certain criterion. One way to do it is using square brackets and $ together:

```
D[D$height>100,]
D[D$plays.tennis==TRUE,]
D[D$plays.tennis ,c("height" , "weight")]
              # this selects height and weight for all
              # persons who play tennis
```

You can also use the function `subset`, which automatically looks for indexing variables in the data frame itself (so instead of `my.data.frame.with.a._very.long.name$V1`, you would write just `V1`):

```
subset(D, height>100)
subset(D, plays.tennis, select=c(height, weight))
```

Factor

Something strange happens when we try to know the mode of the first column of D (`D$name`):

```
mode(D$name)  # why is it numeric??
D$name
```

When we `print(D$name)`, it will superficially look like a character vector, but there is another row labeled "levels". What happened is that when creating the data frame, R automatically converted the character vector to a **factor**. A factor is internally represented as a vector of integers, but each number has a value label associated with it. Especially with long vectors, this is more effective than using character representation, but more importantly, factors are used to tell statistical functions about the type of a variable. For example, the function lm (for **l**inear **m**odels) can perform both ANOVA, ANCOVA, and regression analysis, and which one you'll get depends on the type of variables you supply: if your independent variable is continuous (numeric), you'll get regression; if it's factor, you'll get ANOVA; in a mixed case, you'll get ANCOVA. In other words, factor in R stands for "categorical" or "ordinal" variables in other programs. So it makes sense to have "sex" as factor, but name should probably stay "character" (unless you plan to compare Johns and Jacks in a national sample).

When creating a data frame, character vectors are, by default, converted to factors. This is sometimes unwanted, but fortunately, there are ways to cope with it (see `?data.frame` – you can set `stringsAsFactors` argument to `FALSE`). But let's first check what the modes and classes of variables in D are. Remember, data frames are lists, so `sapply` works here too (whereas apply may not).

```
sapply(D, mode)
sapply(D, class)
# now change D$name to mode "character" using as.character
```

When you create a factor yourself, you have total control over how it's done. The things you can change are:

- `levels` – a vector of possible values that the variable can take; for sex, this could be, e.g., `c("male", "female")`. If no value is specified, sorted unique values of the data vector are used.

- `labels` – an optional vector to be used as factor labels, of the same length as levels; can be different from levels, e.g., `c("M", "F").` If no value is specified, values in levels are used.
- `exclude` – which values to exclude from the factor; the default is NA (this can be useful in modeling functions, for example, you probably would not want to compare men or women with those people who forgot to report their sex).
- `ordered` – do the levels of factor have a natural order? If `TRUE`, then the same order is used as in levels. Ordered factors are shown differently on the screen, and are treated in a special way by some modeling functions.

The levels as stored internally are integer numbers from 1 to the number of levels; this is what you get when using `as.numeric` with a factor (don't do it!). Now let's see some examples (including some examples how we can mess things up completely with factors)

```
factor(1:5)
factor(6:10)
# trying to add one to the other with c()
c(factor(1:5), factor(6:10))
# something evil happened; let's try to do it correctly
# first way:
First <- factor(1:5, levels = 1:10)
Second <- factor(6:10, levels = 1:10)
# to see how they're internally represented, use unclass
unclass(First)
unclass(Second)
c(First, Second)   # correct but levels attr is lost
factor(c(First, Second), levels = levels(First))
# second way
First <- factor(1:5)
Second <- factor(6:10)
as.numeric(c(as.character(First), as.character(Second)))
# as.character replaced internal integer levels of
# a factor by their labels
Third <- factor(c(1,7,3,9))
unclass(Third)
# now let's try to get back the original numbers
as.numeric(Third)           # bad!
as.character(Third)         # correct but wrong mode
as.numeric(as.character(Third))  # does the thing
as.integer(levels(Third))[Third] # -- ,, --
Third[4] <- 100             # note the error message
# you can't add a new level this way
```

The moral is that one should be careful with factors. Factors are absolutely necessary in modeling functions, but they can be dangerous, especially if an implicit conversion to their underlying numeric representation occurs.

6. Reading in data

Two of the most common ways to read in data to R are using `read.table` and `scan`; both read data from ordinary text files. `read.table` creates a data frame; it expects a "rectangular" text file, where cases are in rows, variables in columns, and columns are separated by spaces, TAB characters, commas, or something similar. When using `read.table`, don't forget to assign the result, otherwise it's just printed on the

screen. By default, `read.table` will look for files in R's current "working directory"; to know where that directory is, type `getwd()`; to change it, type `setwd()` – or, at least on Windows, you can choose "Change dir…" from the "File" menu.

```
?read.table   # see the help page
X <- read.table("super data.txt", sep=",", header=TRUE)
# this says that columns are separated by commas, and that
# variable names are in the first row of the file
# sep = "\t" means that cols are separated by TAB character
# sep = ""  (the default) will use any white spaces as
              # separator
# try to read in one of your own files!
read.table("clipboard")   # gets a table from clipboard
choose.files()  # to choose files in a windows way
read.table(choose.files(), sep="\t")
```

With `read.table` (and other methods!), it is useful to check if the data are read as you wanted. The following may be useful:

```
names(X)      # are the variable names right?
dim(X)        # are the dimensions right?
sapply(X, class)   # are the variable types right?
summary(X)    # some summary stats for each column
fix(X)        # will open a spreadsheet-like window with the
              # data
```

Scan is quicker for reading in large matrices (or if there are a large number of data all of the same type), and allows more flexibility; see `?scan`. There are a number of variations of `read.table`; of these, `read.csv` will read comma-separated values, and `read.fwf` is for "fixed width" files (no separator). The package foreign includes a number of functions for reading data in different formats:

```
library(foreign)
?read.spss
# do look at the options on help page if you plan to use it
# default output is a list, you can change this by specifying
# to.data.frame=TRUE
?help.start()
# this opens a web browser with HTML help; click on Packages and
# then on foreign to see a list of data import/export functions
```

In package **memisc**, you can find another way of communicating with SPSS and Stata data files (browse to "importers" in the pacakge's documentation). For Excel data, you can use the following function which uses the package **RODBC**:

```
read.xls <- function(file, sheet){
    # no need to understand the next 4 lines
    require(RODBC)
    channel <- odbcConnectExcel(file)
    t <- sqlFetch(channel, sheet)
    odbcClose(channel)
    t
    }
```

Copying these lines to R will initially do nothing but now you have a **function** that you can later use for reading in data in Excel format, provided that you have ODBC installed on your computer. (It is almost always there on Windows, even if you don't have MS Office installed. On other operation systems, you can either install ODBC or use, for example, an identically named function from package **gregmisc**. That one uses Perl which is usually installed on Unix and similar systems, but not on Windows.) Notice how a function is defined: first its name, then assignment operator, then "function" with its **arguments** in parentheses. The arguments are what you must

provide when using the function: in the present case, you must specify a file name, and the name of the Excel sheet you want to import; both should be of mode "character". (We'll see more about functions later.)

```
# having defined the function, you can use it like this:
myNewData <- read.xls(file="workbook.xls", sheet="Sheet1")
# if you supply arguments in exactly the same order as in
# function definition, you can skip their names:
myNewData <- read.xls("workbook.xls", "Sheet1")
# this will also work:
myNewData <- read.xls(sheet="Sheet1", file="workbook.xls")
# but that one won't:
## myNewData <- read.xls("Sheet1", "workbook.xls")
```

Finally, most data import functions have some tricks. With all of them, you have to be careful as to which variables are converted to factors (check this using `sapply`). For instance, if you have one letter "O" among 1's and 0's in a numeric column, the whole thing will be converted to "character" and then to factor. A common mistake is supplying a file name which is not in R's working directory (see next section about this). Using the above version of read.xls, you have to make sure that worksheet names in Excel are short and contain only "basic" letters (no Ümläuts) and numbers.

7. R workspace

Objects in workspace

When you create objects by assigning a value to a name, they will be stored in the "workspace". You can see all of the available object using the function `objects()`; a shorter way to do it is `ls()` which may sound more familiar to Unix people.

```
objects()  # same as ls()
# if you have done everyting so far, the output will look
# something like this:
# > ls()
# [1] "a"        "A"        "b"        "c"        "d"        "D"
# [7] "First"    "my.test"  "Peter"    "read.xls" "Second"   "t"
# [13] "Third"    "x"        "X"        "y"
```

This way you see only the objects (vectors, data frames, lists, functions, and so on) that you have created yourself; the "internal" objects like functions `mean` and `sd` are not shown for now but can be used. If you're curious, you can see which collections of objects (mostly **packages**) are attached by `search()`, and the objects that each one of them contains by `ls(pos=2)`, `ls(3)`, etc. Technically, `search()` gives you the "search path" – if you ask for an object called, say, mean, R will first look for it in the first item of the search path (called the "global environment"), then in the second, then in the third, etc. If more than one attached packages contain an object with this name, the first one encountered will be used. So if you have created a function called "mean", it will normally be stored in the "global environment" and (as long as you haven't deleted it) it will be used instead of the one in the "base" package. You can also attach lists or data frames to the search path using `attach()`, and detach them later using `detach()`. For example, you can attach the data frame D by `attach(D)` and then write simply `name` or `plays.tennis` instead of `D$name` and `D$plays.tennis`. Useful as it may seem, it can create a lot of confusion when you have identically named objects in the data frame and the global environment.

For removing objects, use `remove()` – or `rm()` in Unix-style. You can remove many or even all of the objects by using the argument `list` to `remove`:

```
rm(x)      # removes only x
rm(list=c("First", "Second", "Third"))   # removes 3 objects
rm(list=ls())   # removes everything given by ls()
```

Some useful habits

A very useful habit when working with R is to keep all the files related to a project in a separate folder. For example, for the R workshop, you might create a new folder called "R-workshop"; in Windows, this would probably be a sub-folder under "My Documents". Having started R, first set your **working directory** to that folder; this is the place then that R will look for when you are going to read in a data file. The working directory can be set using `setwd()` or, at least in Windows, using "Change dir…" in the "File" drop-down menu.

```
getwd()    # see the current working directory
setwd("C:/Documents and Settings/Something/My Documents/R-workshop")
# or use Change dir... from File menu
# notice /forward slashes/ in the file name;
#     \backslash\ can also be used but has to be written \\twice\\
#     that's because in R,   \ is a "prefix" for special characters:
#     e.g., "\t" means tabulation mark, and "\n" means new line
getwd()    # to see it's changed
list.files() # same as dir()  -- to see files in w.d.
```

Having done that, quit R using `q()` or "Exit" from the "File" menu, and say **yes** when asked about whether to save the workspace. This can be done in two ways:

```
# q()    # and say "yes" to "save workspace image?"
# q(save="yes") # same as q("yes") – to save without being asked
```

Now you have a file called ".Rdata" in your project folder; next time when you start working with the same project, you can just double-click on that file. This will open an R session with the working directory already set to the right place, and with all the objects that you saved on a previous session.

Another useful thing when you're doing a serious analysis, is to write your code first in a text editor (say, Notepad, or emacs) and then paste it to R. Delete the lines that didn't work or that you don't plan to use in future, and save the text file in your project folder. This way you can later easily redo all the analyses (possibly using a different data set, or making some modifications to the code) using `source("my-code.R")`. Using MS Word or OpenOffice Writer or a similar program to edit R code is usually not a good idea. These programs will try to find spelling errors in your code (as if it were plain English), "autocorrect" things you might not notice, capitalize the first character of each line, and convert "straight quotes" to "smart quotes" (the latter are not understandable to R!).

```
# source can also be used with files on the Internet, e.g.:
source("http://psych.ut.ee/~nek/R2008/greeting.R")
```

R has some "command-line editing" features. When you type an incomplete expression, for example, a comma at the end of a line, the R command prompt changes from ">" to "+" and you will be able to finish your statement on the next line. Previous commands can be recalled by pressing UP and DOWN arrows on keyboard. Finally, when you press the TAB key, R will try to complete what you're saying.

```
x <- 1:100
x[    # press ENTER now
5]    # continue on next line; the result is == x[5]
c(1,2,3,    # press ENTER
4,5,999)    # and so on
# type data.f , press TAB, and see what happens
# or try data. , and press TAB twice
# you see various functions that begin with "data."
# now press the UP arrow until you see "x<- 1:100"
```

8. Using functions

We have already seen and used a number of functions: mean, sd, sum, and a number of others. Even arithmetic operators (`+`, `-`, etc), the assignment operator (`<-`), and indexing operators (`[`, `[[`) are functions in R, although they are ordinarily used in a different way: most people would prefer to write `1+2` instead of `` `+`(1,2) `` although the latter way can also be used. The ordinary use is, as you've already noticed, typing the function name, and giving the arguments in parentheses (round brackets). The arguments may be **named**: in that case, you first type the argument name, then equation sign, and then the value you want to supply as an argument. Names are not necessary if you supply the arguments in exactly the same order as in the function definition. Some of the arguments may have a default value; in this case, you need to supply a value only if you are not satisfied with the default. The usual way is to give the required arguments (usually the first one or two) without a name, and the optional arguments (there can be many!) with a name – so you don't have to worry about their order, and the code will be easier to understand.
Finally, all information regarding the usage of a function is there in the function's documentation (using `?somefunction` or `help(somefunction)`). Let's see this on the example of mean; its documentation page starts like this:

```
mean                    package:base                    R Documentation
Arithmetic Mean
Description:
     Generic function for the (trimmed) arithmetic mean.
Usage:
     mean(x, ...)
     ## Default S3 method:
     mean(x, trim = 0, na.rm = FALSE, ...)
Arguments:
       x: An R object.  Currently there are methods for numeric data
          frames, numeric vectors and dates.  A complex vector is
          allowed for 'trim = 0', only.
    trim: the fraction (0 to 0.5) of observations to be trimmed from
          each end of 'x' before the mean is computed. Values of trim
          outside that range are taken as the nearest endpoint.
   na.rm: a logical value indicating whether 'NA' values should be
          stripped before the computation proceeds.
Value:
     For a data frame, a named vector with the appropriate method being
     applied column by column.
     If 'trim' is zero (the default), the arithmetic mean of the values
     in 'x' is computed, as a numeric or complex vector of length one.
     If 'x' is not logical (coerced to numeric), integer, numeric or
     complex, 'NA' is returned, with a warning.
     If 'trim' is non-zero, a symmetrically trimmed mean is computed
     with a fraction of 'trim' observations deleted from each end
     before the mean is computed.
References:
     Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) _The New S
     Language_. Wadsworth & Brooks/Cole.
See Also:
     'weighted.mean', 'mean.POSIXct'
Examples:
     x <- c(0:10, 50)
     xm <- mean(x)
     c(xm, mean(x, trim = 0.10))
     mean(USArrests, trim = 0.2)
```

First of all, you can see that "mean" is a **generic** function, that means, it can act differently depending on the class of the object you supply as its first argument. The most common way is using `mean` with a vector: it will give the arithmetic mean of your vector's elements as an answer. From the "value" section you can see that `mean` of a data frame will be a vector of its column means (that doesn't apply to a matrix!!). There is also a method for computing means of date/time objects (`mean.POSIXct`).

Secondly, for the most usual method, you can supply 3 arguments named `x`, `trim`, and `na.rm`. `x` is the vector whose mean you need, `trim` allows you to calculate trimmed mean (the default value is 0, which will result in an ordinary arithmetic mean), and `na.rm` says something about missing data. The default value, `na.rm=FALSE` will give an error message if you have `NA`'s (missing data) in your vector – some of the values are unknown and, consequently, the mean cannot be computed. Setting `na.rm=TRUE` will calculate the mean of available elements, throwing `NA`'s away.

Now let's see some of the most commonly used statistical and related functions in a table (you can experiment with them using your own data). See their documentation for additional arguments.

Function	Arguments	Value
`table`	one or more factor-like objects (factors, character vectors, numeric vectors)	One- or two-way frequency table
`mean`	Numeric vector or a data frame	Arithmetic mean; vector of columnwise means for a data frame; grand mean (a single number) for a matrix
`sum`	One or more vectors	Sum of elements of all vectors
`min, max`	One or more numeric or character vectors	For numeric vectors: minimum or maximum For character vectors: The first element in alphabetical order

`quantile`	Numeric vetor; **`probs`** (a vector of probabilities); **`na.rm`** (whether to remove NA's)	A vector of quantiles; usage:
		`quantile(x, 0.9) # 90th percentile of x`
		`quantile(x, c(.25,.5,.75)) # quartiles`

`sd`	Numeric vector, matrix, or data frame	Standard deviation; columnwise sd's for matrix and data frame
`var`	Numeric vector, matrix, or data frame	Variance of a vector; matrix of variances and covariances for a matrix or data frame
`cor`	Two numeric vectors (`x` and `y`), or a matrix, or a data frame Optional args: **`use`** (default is "all.obs", other possible values "pairwise.complete.obs" and "complete.obs" – the default will give an error if there are missing data); **`method`** ("pearson", "kendall", or "spearman")	A correlation coefficient, or a matrix of correlations
`cor.test`	Two numeric vectors (`x` and `y`) , or a **`formula`** and a **`data`** frame (see section 10 for formulas)	Correlation and various test statistics (p.value, t-test for correlation, df, etc). Usage: `cor.test(x,y)` or `cor.test(y~x, data=my.data.frame)`

`t.test`	Two numeric vectors (`x` and `y`), or a **`formula`** and a **`data`** frame
	Optional args: **`paired`** (defaults to FALSE); **`var.equal`** (assume equal variances?, defaults to FALSE), and some more
		T-value and various test statistics
		Usage:
		`t.test(x,y) # mean diff btw x and y`
		`t.test(y~gr) # 2 groups defined by gr`
		`t.test(x,y, paired=TRUE)`

`rnorm`	**`N`** (number of observations); **`mean`** (defaults to 0); **`sd`** (defaults to 1)	A random vector (n observations) with normal distribution

		See `?rnorm` for density, distribution, and quantile functions of normal distribution

`sample`	A numeric vector `x`; optional args: **`size`** (desired length of result); **`replace`** (whether or not elements of x may occur more than once in the result)	A random sample of elements of x, with or without "replacement". The result is (of course) different each time.
		`sample (1:10) # 1:10 in random order`
		`sample(0:1, 20, replace=TRUE)`
		`# 20 0's and 1's in random order`

`lm`	**`Formula`** (and **`data`** frame)
	Optional args: **`subset`** (of the data); **`na.action`**; and some more
	A **`data`** argument is not needed if both variables are in the global environment, or you have the data frame attach'ed.
		Linear model (regression, ANOVA, ANCOVA and the like; see section 10 for more details)
		Usage:
		`lm(y~x1+x2, my.data)`
		`# regression if both x's and y`
		`# are numeric; ANOVA if x's are`
		`# factors, ANCOVA if x1 is`
		`# numeric and x2 is factor`

`glm`	**`Formula`**, **`data`** frame, and "**`family`**" specifying the distribution of dependent variable	Generalized linear models; usage:
		`glm(y~x, my.data, family="binomial")`
		`# logistic regr.`
		`# … family="poisson" for Poisson regr`
		See `?family` for other distributions

It is also useful to know a little about the "generic functions". These are functions that first check the `class` of its primary argument, and then call the appropriate function for this class. Generic functions are usually meant for common tasks that can apply to many different objects: showing something on the screen, plotting or summarizing it. The generic function `print` controls how an object is shown on the screen (some say, how it's "printed" but this has nothing to do with printers). This function is used implicitly when we, in an R session, type a name of an object: in that sense, typing "`x`" just actually says "`print(x)`". For some objects, for example, results of modeling functions (`lm` or even `t.test`), `print` will show only a small part of what was actually computed. In these cases, `summary` will a show a bit more detailed information, and `str` shows the structure of the object (all elements you can access, usually with their first few values). This is a bit different for vectors and data frames: `print` will show all values on the screen (this can be very long output, and typically not very useful), and `summary` gives you some descriptive statistics for each column.

R belongs to a group of *functional* programming languages, so functions are very important, and defining your own functions is rather easy. A function in R is similar to a mathematical function: it takes any number of **arguments**, and it returns a **value** (which can be a single value, a longer vector, a complex R object, such as a list). In general, functions do not change their arguments, so if you want to preserve the value returned by a function, you should assign it to an object. That behavior is also similar to the mathematical functions – as a trivial example, cosine is a function, and *cos* π is equal to -1; however, that function does not change the value of π!

A function in R is like a "black box" in that it can (in general) access only the values that have been passed to it as arguments. This is best explained by an example:

```
sumof3 <- function(first, second, third) {
          cat("first = ", first, "\n",    # cat() is for showing it on screen
              "second = ", second, "\n",  # \n means new line
              "third = ", third, "\n")
          first + second + third
          }
one <- 1
two <- 2
three <- 3
sumof3(first = three, second = one, third = two)
first <- 1
second <- 2
third <- 3
sumof3(first = third, second = second, third = first)
sumof3(third = first, first = third, second = second)
sumof3(1,2,3)
```

```
sumof3(first=1, 2,3)
```

Notice that the last line in the function definition (first + second + third) will become the function's value when you call it. Printing something on the screen is optional, and does not affect the value. The function sumof3 takes exactly three arguments, prints them with their names (as understood by the function) and computes their sum. As you see, it uses the arguments as passed to it in parentheses, without being bothered by other objects in your workspace that may have the same names as its arguments.

9. Graphics

There are books[3] on graphics using R, so here we can only look at the most basic things. The general rule is that when you have a graph in mind, you can do it with R. See, for instance http://addictedtor.free.fr/graphiques/ for many examples; they include source code, so that you can reproduce or modify them on your own. The basic distribution of R includes the so-called base graphics system which is easy to use and extend, and is excellent for most purposes, but does have some limits. The `grid` add-on package implements another type of low-level graphics system, which is very flexible, but rather difficult to use. `grid` is used in two other high-level graphics systems: `lattice`, and `ggplot2`. `lattice` is also included in the basic distribution (it's a separate package, so you have to load it using `library(lattice)` before you can use it) and allows one to do a variety of complex and multi-panel graphs quite easily, but learning how they can be extended and modified takes some time and effort. `ggplot2` is based on a sound theory of statistical graphics[4] and seems likely to be used more in the future. `iplots` (www.iplots.org) is a way to do interactive graphics in R, and `rpanel` (www.stats.gla.ac.uk/~adrian/rpanel/) offers a different range of possibilities. For many more, take a look at the Graphics task view on CRAN.
In large, graphical functions in R can be divided into high- and low-level functions; the former draw whole plots like scatter plots, histograms and box plots, whereas the latter can add points, lines, text, and the like.
The function `plot` usually draws a scatter plot; when used with modeling functions like `lm`, it is a generic function which usually draws several diagnostic plots of the model (see next section about this use). Various arguments can be used to modify the appearance of the plot (color, shape and size of the points, axis labels, plot title, etc.). `points` and `lines` can be used to add points and lines to an existing plot (this works with other high-level plotting functions too); `abline` can be used to add a regression line or horizontal/vertical lines.

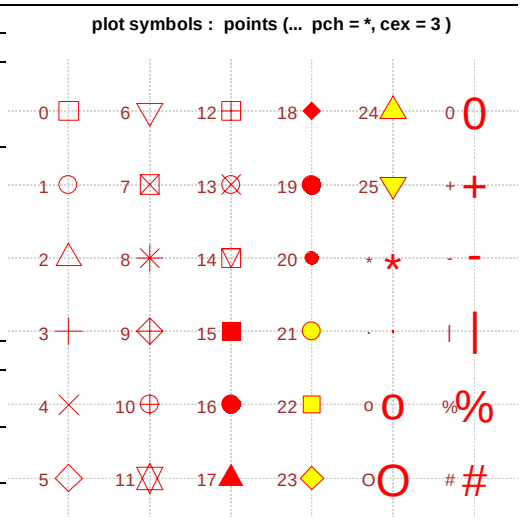
```
# let's use the simulated data from a previous section
x <- rnorm(100)
y <- x + rnorm(100)
plot(y~x)   # same as plot(x,y) but it's clearer with a formula
plot(y~x,     # let's now add some extra parameters
     xlab="This is a label for x axis",
     ylab="This is a label for y axis",
     xlim=c(-5,5),      # x axis from -5 to 5
     ylim=c(-6,6),      # y axis from -6 to 6
     main="Plot title",
     col="blue"         # color of points
     )
abline(lm(y~x), col="green")   # to add a regression line
    # abline can take slope and intercept from the lm output
    # otherwise, use it as abline(a=intercept, b=slope)
```

[3] For example, *R Graphics* by P. Murrell; *Interactive and Dynamic Graphics for Data Analysis* by Cook and Swayne; *Lattice: Multivariate Data Visualization with R* by D. Sarkar.
[4] *A Grammar of Graphics* by L. Wilkinson.

```
     #  .. or abline(h=c(1,2), v=0) to draw some horizontal and/or
     #     vertical lines
text(-2,6, "some text centered at x=-2, y=6")
points(-4:-2, c(-6, -4, -6), col="red")  # add some red points
lines(c(-4:-2,-4), c(-6, -4, -6,-6), col="grey", lwd=3)
     # .. and some thicker grey lines
```

Several graphical parameters can be added to both `plot`, `points`, `lines`, and some other graphical functions:

Name	Meaning	Values
`lwd`	line width	Arbitrary units, default is 1
`lty`	line type	0 or "blank" (no line); 1 or "solid"; 2 or "dashed"; 3 or "dotted"; 4 or "dotdash"; 5 or "longdash"; 6 or "twodash"
`col`	color of lines, points or text	A number (from 1 to many), or one of the character values returned by colors(), or an RGB value: rgb(R, G, B) where the arguments specify the intensity of each basic color from 0 (min) to 1 (max): so rgb(1,0,0) is pure red. Vectors of color values can be used to plot points of different color (e.g., corresponding to different groups).
`cex`	size of points or text	Arbitrary units, default is 1 (2 is twice as large etc)
`pch`	shape of points	One of the numbers or symbols in the following figure
`bg`	background color of points	Same values as for col Applies to pch from 21 to 25
`...`		See `?par  ?points ?plot` and `?plot.default` for more parameters



To finish with scatter plots, let's see how we can make points corresponding to different groups be of different colors. The example below uses simulated data; try it with your own data for some more realistic examples (e.g., a correlation between extraversion and neuroticism, showing men and women in different colors).

```
# let's first simulate some variables correlated with
# a group indicator (in a naive but simple way)
gr <- sample(0:2, 100, replace=TRUE)   # 3 random groups
x <- rnorm(100) + gr   # add gr to make them correlated
y <- x + rnorm(100)
plot(y~x,      # this part is just as before
     col=gr+1) # color values have to start with 1
palette()      # to see which colors correspond to which
               # numbers, by default
plot(y~x, col=gr+2)   # to use different colors
palette(c("blue", "red", "green")) # set a different palette
plot(y~x, col+1)I # now group 1 is blue, gr2 is red etc
# this can be done with xyplot in lattice package in a different way, try
library(lattice)
xyplot(y~x|gr)   # different panels corresp.to each group
xpyplot(y~x, groups=gr)  # groups in diff. colors
```

Some other high-level plotting functions are `barplot` for bar plots, `hist` for histograms, `pie` for pie charts, and `boxplot` for box plots. See their use from examples below:

```
barplot(table(gr)) # plots a frequency table
hist(y)            # histogram of a continuous variable
pie(c(0.2,0.3,0.5), labels=c("SAS", "SPSS", "R"))
pie(table(gr), labels=table(gr), main="an ugly pie of a
                frequency table\n with frequencies as labels")
boxplot(y~gr)    # shows medians, quartiles etc
stripchart(y~gr)
```

Now there are just two things to add. First, when you need a graph to include in a paper or presentation, you'll have a better result if you produce a file directly in the needed format. See `?device` for available formats; some of them are pdf, postscript (including eps), jpeg, and png. You can do it like this:

```
pdf(file="filename.pdf")  # additional args for paper size etc, see ?pdf
plot(y~x)   #or anything else, just as before
dev.off()   # to close the pdf device and finish with the file
```

The second thing is that with par you can make more or less permanent changes to the graphical parameters (they'll last until you change it again, or close the device or graphics window), and change the graph layout (for example, to plot several graphs on one page, or modify white margins around the graph). See `?par` for a complete list of things you can change.

```
par(mfrow=c(2,2))   # to plot 2 x 2 graphs on one page
par(mar=c(2,1,1,1)) # white margins, from bottom to right counterclockwise
# now add 4 plots, e.g
barplot(table(gr)) # plots a frequency table
hist(y)            # histogram of a continuous variable
pie(c(0.2,0.3,0.5), labels=c("SAS", "SPSS", "R"))
plot(y~x, col=gr+1, pch=(15:17)[gr+1])
abline(lm(y~x), col="red", lwd=3, lty="dotted")
```

The easiest way to get the old parameters back is just closing the graphics window or device (by clicking on "close" box on upper-right corner or by `dev.off()`). Another way is to save the graphical parameters before you change them; every time you call par, it returns (invisibly) a value, which contains a list of graphical parameters as they were before changing. You can "save" that list by assigning a name to it, and later use the created object with par to restore the previous settings:

```
olde <- par(mfrow=1:2, mar=c(0,0,0,0))
# plot something
par(olde)  #
```

10. Linear models

The function `lm` is used to fit linear models of any type (regression, ANOVA, ANCOVA) – depending on types of variables you give to it. The first and the only required argument in lm is a `formula` that specifies the model to be fitted; a second argument, `data`, is useful when your variables are in a data frame rather than separately in the workspace. Look at the arguments `subset` and `na.action` in ?lm when you need to use only a subset of data, or when you want to tell R how exactly it should treat missing data.

A **formula** consists of two parts: a response variable (same as *dependent* variable), and the "terms" (*independent* variables), separated by a tilde (~). A simple formula `y ~ x` says that y is regressed on x; intercept is included by default. More terms can be added using the operators "+", "*", and ":":

```
y ~ x1 + x2  # main effects of x1 and x2 on y
y ~ x1 * x2  # main effects of x1 and x2 plus their interaction
y ~ x1 + x2 + x1:x2  # same as previous
y ~ x1 : x2  # only the two-way interaction between x1 and x2
```

"1" in a formula denotes intercept, and "0" or "-1" means its omission:

```
y ~ 1 + x  # same as y ~x
y ~ 0 + x  # y regressed on x, with no intercept
y ~ x -1   # same as previous
```

Some functions can be used directly in a formula:

```
y ~ sin(x1) + cos(x2)  # y regressed on sine of x1 and cosine of x2
```

Because arithmetic operators for addition, multiplication, division, and exponentiation have a special meaning within a formula, these operations need to be "escaped" by a function `I()`

```
y ~ x1 * x2  # main effects of x1 and x2 plus their
               interaction
y ~ I(x1*x2) # product of x1 and x2
y ~ x1 + x2  # main effects of x1 and x2
y ~ I(x1+x2)  # sum of x1 and x2
```

Now that you know almost everything about formulas, we can use lm to ... compute a mean, using a model with only intercept:

```
x <- rnorm(n=100, mean=5, sd=1.5)  # to generate some data
mean(x)
lm(x ~ 1)
confint(lm(x ~ 1))  # simple way to calculate Cis around a mean
summary(lm(x~1))
```

Let's now create two groups, A and B that correspond to the first and last 50 observations in x, and make x differ across these groups:

```
GR <- c(rep("A", 50), rep("B", 50))  # rep is for repeating
x[1:50] <- x[1:50] + 3
lm(x ~GR)       # anova; GR converted to factor; warning
lm(x ~GR -1 )   # different parametrization of the model
tapply(x, GR, mean)
```

As you could already notice, lm does an ANOVA whenever an independent variable can be interpreted as a factor. It is, however, better to convert them to factors before calling lm to avoid surprises (one way to do it is `GR <- factor(GR)`). For example, a grouping variable may contain numeric codes; if you don't tell R it should be treated as a factor, `lm` will do an ordinary regression analysis. When all independent variables are numeric, `lm` does a linear regression; of course, you can also use a mixture of numeric and factor variables.

Now we are prepared to do a little simulation:

```
x <- rnorm(100)
y <- 4 + 3*x + rnorm(100)  # a variable correlated with x
lm(y~x)   # see how close you get the parameters to the
          # "actual" values 4 and 3
```

Notice how y is made of x and "random error"; because x and "random error" have similar variances but x is multiplied by 3, then the result should contain much more x than "random error". To make this simulation a bit more instructive, let's see how the estimate of regression parameter b depends on the proportion of "random error" in y, (which is proportional to the correlation between x and y). Intuitively, with more "random error", parameter estimates should vary more around the "real values", but we can try to get a more exact picture.

```
# let's first make a function to calculate y
fun <- function(x, a, b, e) {
     a + b*x + e*rnorm(length(x))
     }
# now we need a vector to store the calculated values of b
b <- numeric(1000)
# and we need a way to get a single coefficient from lm
y <- fun(x, 4,3,1)  # same as y<-4+3*x+1*rnorm(100)
model <- lm(y~x)
coef(model)
coef(model)[2]   # that's it!
# now we'll use for to loop through different values of e
# a simple use of for would be like this:
```

```
# for(x in 1:10) print(x) # prints numbers from 1 to 10
for(i in 1:1000) {
    y <- fun(x, 4, 3, i/200)
    model <- lm(y~x)
    b[i] <- coef(model)[2]
    }
# let's now plot the results with e on x-axis
e <- (1:1000) / 200
plot( b ~ e)
```

From the plot we can see that the variance of regression coefficients is much larger on the right (i.e., with larger proportions of „random error“ in y), but the mean looks pretty much the same.

```
mean(b)
sd(b)
mean(b[e<1])
sd(b[e<1])
mean(b[e>4])
sd(b[e>4])
```

11.Using R with ... and for ...

There are a number of ways R can be integrated with other programs. It can be integrated with a web server and used remotely; search the web for "Rweb" to see how it works (you can even do something serious this way without having to install R at all).

Some people prefer a graphical user interface (GUI) to typing commands with keyboard; there are several GUIs for R, although none is as sophisticated as SPSS. Two of them seem to be more elaborated than the others: Rcmdr ("R Commander") and pmg ("Poor Man's GUI"); they can be installed from CRAN by typing `install.packages(c("pmg", "Rcmdr"))` – and a GUI starts after you load the add-on package with `library()`. A more or less complete list of GUIs for R can be seen at http://www.sciviews.org/_rgui/ (some of them are not working or not maintained any more).

Some people want to have a better text editor than Notepad, or even a tool that would allow controlling R from a text editor. There are a number of possibilities here, for example, Emacs (see http://socserv.mcmaster.ca/jfox/Books/Companion/ESS/), which is a very sophisticated text editor, or Tinn-R (http://www.sciviews.org/Tinn-R/) which is a bit simpler. JGR (http://stats.math.uni-augsburg.de/JGR/) combines a text editor with an "object explorer" and a number of other features.

Some even more sophisticated people would like to include R code in a text document, so that a plot or a table would be produced automatically, and would change when data change. This can be done with Sweave (a function in standard R distribution, see `?Sweave`), but you have to know LaTeX (a somewhat complicated text preparation system) to use it. There is, however, an easier way: using LYX (which is a WYSIWYG-like text editor, that uses LaTeX so that you don't see much of it); this is described in R newsletter at http://cran.r-project.org/doc/Rnews/Rnews_2008-1.pdf

And what is probably even easier is using the same principle with Open Document Format (ODF; a format used by, e.g., OpenOffice); for that you'll need a package called odfWeave from CRAN, and another Rnews article will be useful: http://cran.univ-lyon1.fr/doc/Rnews/Rnews_2006-4.pdf .

12.Exercises

1. Try to read some of your own data into R. Which are the modes and classes of each column in the resulting data frame?
2. Calculate some basic statistics for each of the variables in your data file. Do a regression analysis with some of your variables, and try to make a related plot.
3. Try to find a way to calculate cronbach's alpha (the function is not in the standard distribution but it is there in more than one add-on packages). Can you write your own function that does that?

Üldistatud lineaarsed mudelid sotsiaalteadustes
Praktikum (SHPH.00.001, kevad 2010)

1. praktikum

1. R-i paigaldamine oma arvutisse

Enne praktikumi tuleks R installeerida oma arvutisse. Selleks võtke lahti R-i kodulehekülg http://www.r-project.org/, valige ekraani vasakpoolsest servast CRAN (Comprehensive R Archive Network), seejärel üks „mirror“ serveritest (soovitavalt mõni, mis on Eestile lähemal, nt Austria, Belgia, …). CRAN-i pealeheküljelt valige oma arvuti operatsioonisüsteem, Windowsi puhul seejärel „base“.

2. Andmestiku avamine

Enne töö alustamist on soovitav tekitada eraldi kaust („folder“), mida hakkate kasutama kogu praktikumi jooksul andmete jm R-i failide hoidmiseks. Selle nimi võib olla näiteks „Praktikum“ (kaustas „My Documents“, või kui töötate mitmes erinevas arvutis, siis hoopis mälupulgal). Kutsume seda kausta edaspidi *töökataloogiks* (*working directory*).

Salvestage oma töökataloogi Euroopa Sotsiaaluuringu andmefail ESS08ef.rda aadressilt http://psych.ut.ee/~nek/glm .

Avage R (Windowsis kas Start menüüst või klõpsates ikooni töölaual). Määrame kõigepealt töökataloogi: selleks valige menüüst „File → Change dir….“ või toksige

```
> setwd(choose.dir())
```

(R-ile antavad korraldused on siin ja edaspidi `Courier` kirjastiilis. Märki `>` pole vaja sisestada, seda teeb R ise andmaks märku, et ta on valmis meid kuulama.)

Nüüd valige hiire abil töökataloog. Kontrollime, kas nüüd sai kõik õigesti – selleks kirjutame R-is:

```
> dir()
```

Kui kõik on korras, siis peaksite nägema vähemalt just alla laetud faili ESS08ef.rda. (Kui seda seal ei ole, siis on midagi eespool valesti läinud.) Loeme nüüd andmestiku sisse:

```
> load("ESS08ef.rda")
```

Kontrollime ka selle üle:

```
> objects()   # näitab kõiki "objekte" R-i tööruumis
>             #### "trellide" taga on kommentaarird
>             # nüüd peaksime nägema midagi sellist:
>             #      [1] "ESS"
```

Salvestame nüüd tulemuse: selleks valime menüüst "File" → "Save workspace". Falili nimeks võib jätta lihtsalt ".Rdata" – edaspidi, kui tahate sama andmestikuga töötada, siis võite töökataloogist lihtsalt avada hiire topeltklõpsuga selle faili. Kasulik on ka pärast töö lõppu töökeskkond salvestada – enne programmist väljumist küsitakse seda ka üle ("Save workspace image?" → sellele võiks siis vastata "Yes":). Sel juhul salvestatakse kõik vahepeal tehtud muutused või uued objektid, samuti käskude "ajalugu" (eelmise käsu juurde saab tagasi minna vajutades "üles" nooleklahvi).

Vahemärkusena: R ei ole kuigi sobiv andmete sisestamiseks. Tavaliselt tehakse seda mõnes teises programmis, nt Excelis või lausa SPSS-is ning imporditakse siis andmed R-i. Kõige kiirem on üldjuhul ASCII tekstina (.txt või *.csv) salvestatud andmete sisselugemine, kuid R on suuteline aru saama ka SPSS-i (*.sav) failidest, samuti näiteks Exceli tabelitest. Kuidas see käib, seda uurime hiljem.*

3. Kirjeldav statistika

Uudistame kõigepealt natuke seda andmestikku ja siis teeme mõned lihtsad analüüsid.

```
> names(ESS)   # näitab kõigi tunnuste nimesid
> head(ESS)    # näitab esimest kuut rida
```

Jube! Tunnuste nimed ei ole üldse arusaadavad, aga õnneks on kursuse kodulehel ka fail kõigi tunnuste lühikirjeldustega (ESS2008coding.xls).

```
> ESS$gndr     # muutuja nimega "gndr" tabelist nimega "ESS"
> table(ESS$gndr)              # sagedustabel
> table(ESS$gndr, ESS$cntry)   # 2D sagedustabel: sugu*maa
> hist(ESS$yrbrn)              # sünniaasta histogramm

> ESS$yrbrn[ESS$cntry=="EE"]   # eestlaste sünniaastad
> hist(ESS$yrbrn[ESS$cntry=="EE"])  # histogramm eelmisest
```

Kas ei saaks kuidagi lühemalt, ilma nende dollarimärkideta? Saab nii:

```
> attach(ESS)
> table(gndr)
> hist(yrbrn[cntry == "EE" & gndr == 1])
> # päris lõpuks tasub öelda ka detach(ESS)
```

Siin on küll väike oht – kui tahame andmestikus midagi muuta (nt mõne tunnuse ümber kodeerida või uusi tnnuseid moodustada), siis see ei kajastu algses andmestikus, vaid töölaual n-ö "lahtiste" muutujatena. Juhul, kui töölaual on näiteks "lahtine" muutuja "gndr", kuid sama nimega muutuja on ka `attach`*'itud andmetabelis, siis kasutatakse esimest. Kui on* `attach`*'itud mitu andmetabelit, milles kõigis on muutuja "gndr", siis kasutatakse viimasena lisatud andmetabelit. Jne. Ühesõnaga, keerulisematel juhtudel võib olla parem kasutada dollarimärke, et oleks selge, millisest andmestikust tunnused võetakse. Praegu aga on meil ainult üks andmetabel, nii et võime seda julgelt kasutada. Lõpuks:* `attach` *käsklus kaotab oma kehtivuse R-i sulgemisel, st uue sessiooni avamisel tuleb seda R-ile uuesti üle öelda.*

Vaatame nüüd kaht tunnust andmestiku lõpust: gener_trust ja gener_satisf. Esimene neist on keskmine viiest ja teine kuuest algtunnusest, milleks on:

gener_trust	***gener_satisf***
Trust in country's parliament	*How satisfied with life as a whole*
Trust in the legal system	*How satisfied with present state of economy in country*
Trust in the police	*How satisfied with the national government*
Trust in politicians	*How satisfied with the way democracy works in country*
Trust in political parties	*State of education in country nowadays*
	State of health services in country nowadays

```
> mean(gener_trust)      # osa andmeid on puudu!
> mean(gener_trust, na.rm=TRUE) # viskame puuduvad
>                               # andmed välja
> median(gener_trust, na.rm=TRUE)
> sd(gener_trust, na.rm=TRUE)
> quantile(gener_trust, na.rm=TRUE)
> tapply(gener_trust, cntry, mean, na.rm=TRUE)
> tapply(gener_trust, cntry, sd, na.rm=TRUE)
```

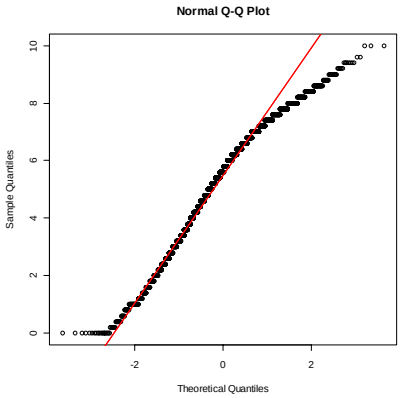
4. Normaal- jt jaotustest

Uurime, millise jaotusega on tunnus gener_trust. Kõige esimese pildi võiks anda histogramm:

```
> hist(gener_trust)
```

Järgmisena proovime "Q-Q joonist" (Q-Q plot), mis näitab, kui hästi tunnuse tegelik jaotus sobib teoreetilise jaotusega (praegu normaaljaotusega). Märgime ideaalse vastavuse punase joonega:

```
> qqnorm(gener_trust)
> qqline(gener_trust, col="red")
>
```



Mida täpsemini "täpid on joone peal", seda paremini tunnuse jaotus normaaljaotusega sobib. Näeme, et praegu on kõrvalekalle suuremate väärtuste piirkonnas; ka histogrammilt nägime, et jaotusel on paremal pool välja venitatud "saba". Silma järgi hinnates ei ole erinevus normaaljaotusest siiski väga suur.

Proovime Shapiro-Wilki ja Kolmogorov-Smirnovi testide abil, kas erinevus normaaljaotusest on statistiliselt oluline:

```
> shapiro.test(gener_trust)
> ks.test(gener_trust, "pnorm")
```

"pnorm" eespool tähendab, et meid huvitab sobivus normaaljaotusega (tehnilisemalt on tegu viitega jaotuse tõenäosusfunktsioonile). Test sobib ka teiste **pidevate** jaotuste sobitamiseks, nt log-normaaljaotuse korral kirjutame sinna "pnorm" asemel "plnorm".

Ülesanne: kontrollige visuaalselt (kasutades histogramme ja QQ-jooniseid), kas usalduse ja rahulolu jaotus on Eestis normaaljaotusele sama sarnane kui Soomes. Vihje: rahulolu andmed nt soome alavalimis: `gener_satisf[cntry=="FI"].`

5. Lihtne regressioonimudel

Tavalise lineaarse regressioonimudeli koostamiseks kasutame funktsiooni "lm". Selle funktsiooni esimeseks argumendiks on **valem** kujul ParemPool ~ VasakPool, kus vasakul pool on sõltuv tunnus ning paremal prediktorid. Teise sageli kasutatava argumendi nimi on "data" -- sellega saab ette anda andmestiku, kust R hakkab tunnuseid otsima. Mõnikord on kasu ka argumendist "subset" – kui tahame kasutada ainult osa andmestikust.

```
> lm(gener_trust ~ gener_satisf, data=ESS, subset=cntry=="EE")
```

Täpsema väljundi saame, kui kirjutame selle ümber "summary" – selleks liigume "ÜLES" nooleklahviga eelmise käsu juurde tagasi ja täiendame seda:

```
> summary(lm(gener_trust ~ gener_satisf, data=ESS, subset=cntry=="EE"))
```

Jäädvustame selle mudeli tunnusesse nimega "m1":

```
> m1 <- lm(gener_trust ~ gener_satisf, data=ESS, subset=cntry=="EE")
> summary(m1)
```

Väljundist leiame muu hulgas regressioonikordajad, nende standardvead ja olulisuse tõenäosused, samuti mudeli headuse näitajana R^2. Lisame nüüd mudelisse ka vanuse tunnuse (andmestikus on see peidetud nime "agea" taha):

```
> m2 <- lm(gener_trust ~ gener_satisf + agea, data=ESS, subset=cntry=="EE")
> summary(m2)
```

Kui palju R^2 suurenes võrreldes eelmise mudeliga?

Ülesanne: (1) koostage samasugused mudelid kasutades ainult soome andmeid. Nimetage need vastavalt m1f ja m2f. (2) Koostage kahe maa andmete põhjal keerulisem regressioonimudel, kus prediktoriteks on nii rahulolu, vanus, kui ka maa ("cntry").

6. Regressioonimudeli eeldused

Kontrollime visuaalselt lineaarse regressioonimudeli eeldusi: (A) lineaarsus, (B) konstantne hajuvus [homoskedastilisus] ja (C) jääkide normaaljaotus. Väga hea silmaga inimene võiks kõik need eeldused teoreetiliselt välja lugeda tavaliselt hajuvusdiagrammilt, mille nüüd kohe joonistame:

```
> plot(gener_trust ~ gener_satisf, data=ESS, subset=cntry=="EE")
> abline(m1)  # lisame regressioonijoone m1-st võetud parameetritega
```

Esimese eelduse selge rikutuse korral peaksime nägema selgelt mittelineaarset seost. Vaatame, kuidas see välja näeb, võttes näiteks sõltuva tunnuse ruutu:

```
> plot(gener_trust^2 ~ gener_satisf, data=ESS, subset=cntry=="EE")
```

Konstantne hajuvus tähendab, et sõltuva tunnuse hajuvus on kogu joonise ulatuses sarnane, see ei tohi olla näiteks vasakus servas nullilähedane ja paremal pool hiigelsuur. Seda on parem vaadata jooniselt, kus y-teljel on regressioonijäägid ning x-teljel prognoositud sõltuva tunnuse väärtused. Kõige lihtsam on sellist joonist teha nii:

```
> plot(m1, which=1)
> # aga saab ka nii: plot(resid(m1)~fitted(m1))
```

Jääkide normaaljaotuse eeldust võiksime vaadata Q-Q-jooniselt, mida saab tekitada jälle kahel moel:

```
> plot(m1, which=2)
> # või ...
> qqnorm(resid(m1))
> qqline(resid(m1))
```

Ülesanne: (1) tehke samad joonised mudeli kohta, kus on prediktorite hulgas ka vanus. (2) tehke mudel kogu andmestikule, ignoreerides elukohta (st tunnust `cntry`). Uurige, kas regressioonimudeli eeldused on paremini täidetud, kui elukohta arvesse võtta.

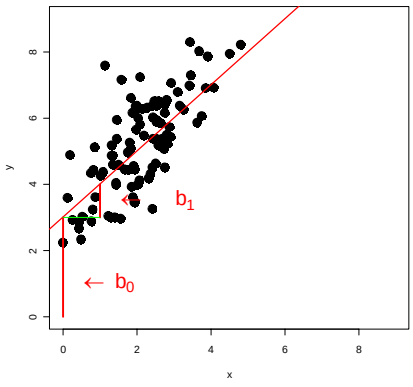
7. Regressioon simuleeritud andmetega

Selle näite mõte on vaadata, kuidas regressioonanalüüs suudab taastada "tõelise olukorra", st tunnuste vahelise teadaoleva seose. Konstrueerime tunnused x ja y, mis on omavahel järgmises seoses:

$$y = 3 + x + \textit{juhuslik viga}$$

Arvutile antakse ette tunnused x ja y ning ta peab nende põhjal püüdma taastada parameetrid: vabaliikme ja regressioonisirge tõusu (praegusel juhul on vabaliikmeks 3.00 ja tõusuks 1.00). Vaatame, kas arvuti saab sellega hakkama.

```
x <- rnorm(100)  # 100 normaaljaotusega juhuslikku arvu
x <- x-min(x)    # teeme nii, et x oleks alati positiivne arv
y <- 3 + x + rnorm(100)
lm(y~x)
# mina sain vabaliikmeks 2.992 ja x-i kordajaks 1.06
# kuna tegu on juhuslike arvudega, siis tulevad need iga kord
# pisut erinevad, kuid küllalt lähedased "tõelistele" väärtustele
```



Kõrvalolev joonis illustreerib regressiooniparameetrite (vabaliige – b0, x-i kordaja b1) tähendust graafiliselt. Kood selle joonise järgitegemiseks on:

```
plot(y~x, ylim=c(0,9), xlim=c(0,9), pch=19,
 cex=2)
abline(3,1, lwd=2, col="red")
arrows(0,0,0,3, col="red", length=0, lwd=3)
text(0.1,1, expression(paste(" "%<-%" ",
 b[0])), pos=4, col="red", cex=2)
arrows(1,3,1,4, col="red", length=0, lwd=3)
arrows(0,3,1,3, col="green", length=0)
text(1.1, 3.5, expression(paste(" "%<-%" ", "
 ", b[1])), pos=4, col="red", cex=2)
```

Ülesanne. Tekitage simuleeritud tunnused y, x1 ja x2 nii, et y = 3*x1 + 7*x2 + 39 + juhuslik viga. Vaadake, kuidas lineaarne regressioon suudab need parameetrid taastada.

Euroopa Liit
Euroopa Sotsiaalfond

Eesti tuleviku heaks

Üldistatud lineaarsed mudelid sotsiaalteadustes
Praktikum (SHPH.00.001, kevad 2010)

1. praktikumi lisamaterjal koduseks tegevuseks

NB! Kursuse ajutine kodulehekülg, kust leiab kõik allpool viidatud failid, on http://psych.ut.ee/~nek/glm/ (kavas on edaspidi kasutada e-õppe keskkonda Moodle).

1. Kuidas saada oma andmed R-i

R-is on andmete sisselugemiseks terve hulk funktsioone; väga sageli kasutatakse funktsiooni read.table, mis võimaldab sisse lugeda andmeid ASCII teksti formaadis. (St selliseid faile, mida saate avada Windowsis näiteks notepad-iga.) Funktsiooniga read.table võib vaja minna järgmisi argumente:

Argumendi nimi	**Tähendus**	**Võimalikud väärtused**
`file`	Faili nimi	-
`header`	Kas esimeses reas on tunnuste nimed?	**TRUE**, FALSE
`sep`	Andmeväljade eraldaja	*Sümbol jutumärkides. Näiteks:* **""** (tühik või tabulatsioonimärk) "\t" (tabulatsioonimärk) " " (tühik) "," (koma) jne ….
`quote`	Kui andmestikus on tekstilisi muutujaid, kus ühes andmeväljas võib olla näiteks tühikuid, siis kuidas on sellised muutujad eristatavad	Vaikimisi: eraldajana kasutatakse jutumärke. Kui jutumärkidel ei ole sellist tähendust, siis tuleks kasutada quote=""
`dec`	Kümnendkoha eraldaja	Vaikimisi "." (punkt), võimalik ka näiteks "," (koma) vm.
`as.is`	Kas teisendada tekstilised muutujad faktoriteks või mitte?	**FALSE**, TRUE
`na.strings`	Kuidas on tähistatud puuduvad andmed?	*Väärtus jutumärkides. Näiteks: "NA" või "" või c("NA", "") – viimasel juhul esineb failis kaks võimalikku puuduvate andmete tähist.*

Funktsiooni read.table kasutatakse niivõrd sageli, et sellest on tehtud mitu "klooni", mis kasutavad erinevaid vaikeväärtusi: read.csv, read.csv2 jne

	`read.csv`	**`read.csv2`**	**`read.delim`**	**`read.delim2`**
`sep`	Koma	semikoolon	TAB ("\t")	TAB ("\t")
`dec`	Punkt	koma	punkt	koma

Funktsiooni read.fwf saab kasutada failide puhul, kus andmeväljade eraldajat ei ole, vaid kõik andmeväljad on mingi teadaoleva pikkusega. Näiteks on teada, et igas reas viis esimest sümbolit on katseisiku kood, edasi kuni rea lõpuni on IQ testi vastused, kus iga vastus võtab enda alla ühe sümboli. read.DIF mõistab nn "Data Interchange Format"it, milles nt ka Excel võimaldab andmeid salvestada.

Lisaks sellele on hulk funktsioone, mis on mõeldud mingi kindla programmiga salvestatud andmete lugemiseks; nende kasutamiseks on kõigepealt vaja kasutusele võtta lisamoodul "foreign":

```
> library(foreign)
```

Funktsioon	Programm või failiformaadi selgitus
`read.arff`	ARF, http://www.cs.waikato.ac.nz/~ml/weka/arff.html
`read.epiinfo`	EpiInfo
`read.dbf`	FoxPro, dBase
`read.mtp`	MiniTab
`read.octave`	Octave
`read.ssd, read.xport`	SAS
`read.S`	S-Plus
`read.spss`	SPSS
`read.dta`	Stata
`read.systat`	Systat

Exceli failide R-ile söödavaks tegemiseks on kaks võimalust. Funktsioon read.xls lisamoodulist "gdata" tõlgib Exceli faili tavaliseks tekstifailiks ning loeb selle sisse kasutades funktsiooni read.csv. "Tõlkimiseks" kasutatakse programmi perl, mis peab siis kas arvutis juba olemas olema või tuleb see hankida. Tavaliselt on see olemas Linuxi ja arvatavasti ka Maci arvutites, kuid mitte Windowsis (v.a. Tiidu hallatavad arvutid).

```
> install.packages("gdata")  # vajalik on internetiühendus!
# installeerimine on vajalik ainult üks kord –
# edaspidi võib alustada järgmisest reast!
> library(gdata)
> read.xls("thefile.xls")    # vajalik on Perl !!!
> read.xls("thefile.xls", perl="C:/Perl/bin/perl.exe")
> U <- read.xls("thefile.xls")  # erinevus eelmisest?
```

Teine võimalus on kasutada lisamoodulit RODBC, mis on mõeldud R-i ühendamiseks andmebaasidega ODBC kaudu. Nii saab jagu näiteks Exceli tabelitest kui ka Accessi andmebaasidest, kuid protseduur on pisut keerulisem (seda on võimalik siiski enda jaoks lihtsustada).

```
> install.packages("RODBC")
> library(RODBC)
> com <- odbcConnectExcel("thefile.xls") # loome ühenduse
# Accessi puhul siis vastavalt odbcConnectAccess
> sqlTables(com)   # millised tabelid selles andmebaasis on?
> sqlFetch(com, "Sheet1$")
# miskipärast lisandub sageli lehe nime lõppu $-märk
> close(com)
```

Mooduli RODBC abil on võimalik ka Exceli faili muuta, sooritada seal SQL päringuid jms (kuigi keerulised päringud ei lähe läbi). Selle mooduli kasutamiseks peab arvutis olema ODBC draiver, mis on Windowsi arvutites peaaegu alati, kuid Maci ja Linuxi puhul tuleb see esmalt muretseda.

Moodulis XML leidub funktsioon readHTMLTable, mis saab aru HTML vormingus tabelitest (ka siis, kui samas failis on mitu tabelit – kuigi võib arvata, et tekib probleem, kui üks tabel on teise sees, nagu HTMLi puhul võib juhtuda).

Näeme, et sama faili on võimalik sisse lugeda mitmel moel. Millist viisi eelistada?

1. Üldjuhul on vähemalt alguses lihtsam salvestada andmed tekstifailina (nt kas csv formaadis või tabulatsioonimärgiga eraldatud andmetena), mida kõik statistika- ja tabelarvutus-programmid võimaldavad. See viis on eelistatav, kui meil ei ole palju erinevaid andmefaile, ja andmete struktuur ei ole väga keeruline.
2. Mõnikord andmete sisselugemine ei õnnestu esimesel katsel. Siis võib olla lihtsam salvestada andmestik mingis teises vormingus ja proovida seda sisse lugeda teise funktsiooni abil – vea leidmine võib olla küllaltki tülikas, kuigi enamasti võimalik.
3. RODBC (või ka teiste andmebaasidega otse suhtlevate moodulite) kasutamine on mõistlik näiteks siis, kui keegi andmestikku pidevalt Excelis uuendab, või kui näiteks andmestikus on kuupäevi, mida muidu oleks tülikas R-ile arusaadavaks vormistada.
4. SPSS-i andmete puhul tasub tähele panna lisaparameetreid, nt to.data.frame .

Ülesanne. Proovige R-i saada andmestik failist andmestikp1a.xls (nimetage vastavat R-i obekti näiteks X).

2. Andmestruktuurid R-is

R-is on hulk erinevaid andmestruktuure, millest tasub natuke teada.

Vektor on sama tüüpi andmete kogum, näiteks arvud ühest kümneni või ladina tähestiku tähed. Vektoris võib olla üks või rohkem elemente. Vektorid on näiteks:

```
> 1
> 1:10
> c(1,2,3,2,2,3)
> "a"
> c("a", "b", "õ")
> c(TRUE, FALSE)
```

Vektori elementidel võivad olla nimed, ja ka vektoril endal võib olla nimi:

```
> IQskoorid <- (Karin = 150, Rene = 150, Andero = 150)
```

Vektori eritüübiks on faktor, mis on täisarvude vektor, kus igale võimalikule väärtusele on omistatud mingi "silt". Faktoreid kasutatakse kategooriatunnustena (nt sugu, rahvus, jne) – nt funktsioon lm eeldab, et kategooriatunnused on faktorid, või üritab ta neid ise faktoriteks muuta.

Mitu sama pikkusega vektorit võivad moodustada andmetabeli (*data.frame*):

```
> X <- data.frame( nimi = c("Karin", "Rene", "Andero"),
                   IQ = c(150, 150, 150),
                   sugu = factor(c("n", "m", "m")))
```

```
> X   # või print(X) – et näha tulemust ekraanil
```

Funktsioonid nagu read.table või read.csv jne annavad tulemuseks enamasti andmetabeli, kus tekstilised tunnused nagu nimi või sugu muudetakse automaatselt faktoriteks.

List on andmestruktuur, mille elementideks on null või rohkem R-i objekti. Paljude funktsioonide väljundiks on list.

3. Joonised R-iga

3.1. Funktsioon plot

`plot` on „geneeriline“ (*generic*) funktsioon, st tema käitumine sõltub argumendi klassist. Samasugused funktsioonid on ka nt `print`, `summary`, `mean`.

Mingil viisil graafiliselt kujutada saab paljusid väga erinevaid objekte. Näited erinevatest joonistest (mida kõiki saab teha plot-funktsiooniga on:

- siinus-funktsiooni graafik: argumendiks on funktsioon `sin`
- histogramm: argumendiks on teatud viisil esitatud sagedustabel
- hajuvusdiagramm: argumendiks on kaks vektorit: x ja y või kahe tunnusega andmetabel või kahe veeruga maatriks (või ka valem kujul `y~x`, kui x ja y on pidevad tunnused)
- karpdiagramm: argumendiks on grupitunnus x ja pidev tunnusvektor y või valem kujul `y~x`
- lineaarse mudeli diagnostilised graafikud: argumendiks on lm-objekt

Kõigi nende jaoks on tegelikult eraldi funktsioonid („meetodid“), nt `plot.function`, `plot.lm`, `plot.histogram` jne, kuid nende nimesid ei pruugi kasutaja teada. Funktsioon „plot“ ise teeb tegelikult n-ö projektijuhi tööd: vaatab, millist tüüpi argument talle on ette antud, ja suunab tegevuse edasi spetsialiseerunud funktsioonidele (kui argumendiks on funktsioon, siis funktsioonile plot.function, kui faktor või faktorid, siis funktsioonile plot.factor, jne).

3.2. Graafika väljund

Graafikafunktsioonide väljundi saab suunata mitmesse kohta – peale arvutiekraani ka näiteks pdf- või jpg-faili. Kui eesmärgiks ei ole ainult joonise vaatamine siin ja praegu, siis on alati mõistlik valida selline formaat, nagu lõpuks vaja läheb: kodulehele riputamiseks png või jpg-fail, wordi või powerpointi dokumenti toppimiseks „windows metafile”, ajakirjale saatmiseks postscript või pdf.

Kuidas see käib?

```
pdf(file=”ehhee.pdf”, width=3, height=6)
# suunab graafika väljundi faili ”ehhee.pdf”, mille
# laius on 3 tolli ja kõrgus 6 tolli
plot(1:10, 10:1)
# jne
hist(rnorm(100))
dev.off()
# paneb pdf-faili kinni, edasine graafika läheb ekraanile
```

Võimalikud väljundid:

- pdf

- bitmap # ghostscripti kaudu; palju erinevaid failitüüpe sõltuvalt gs-i versioonist
- jpeg, bmp, png
- postscript # selle saab teha *.eps-failiks, mida ajakirjad sageli nõuavad
- win.metafile # windows metafile
- pictex # latex-failide jaoks; psühholoogid seda ei kasuta ☺

3.3. „Kõrgtaseme" graafikafunktsioonid

Sellised funktsioonid on mõeldud mingite tüüpiliste jooniste tegemiseks, mida sageli vaja läheb. Funktsioonist plot ja selle omapäradest oli juba juttu. Teised sarnased funktsioonid on näiteks:

```
barplot(height, …)  # tulpdiagramm
# argumendiks on vektor tulpade kõrgustega või
#   … maatriks, mille iga veerg tähistab eraldi gruppi
# nt:
barplot(1:10)
(a <- matrix(1:10, nrow=2))
barplot(a)
(b <- matrix(1:10, ncol=2))
barplot(b)

hist(x, …)    # histogramm
# x on algandmete vektor
x <- rnorm(100)
hist(x)

pie(x, …)      #pirukadiagramm
# x on tordilõikude suuruste vektor

boxplot(…)     #karpdiagramm
```

Kõigile "kõrgtaseme" funktsioonidele saab lisada mitmesuguseid parameetreid, millega saab lisada nt joonise pealkirja, muuta punktide värvi, suurust või kuju jne.

```
x <- rnorm(100)
y <- x+ rnorm(100)
plot(y ~ x, xlim = c(-4,4), ylim=c(-3,3), main="Proov")
# xlim: x-koordinaatide vahemik; main: joonise pealkiri
plot(x,y, pch=21, col="red", cex=5)
# pch: punkti kuju; col: punkti värv; cex: punkti suurus
```

3.4. „Alamad" graafikafunktsioonid ja graafikaparameetrid

Joonistele saab ka ise lisada punkte, jooni, jm. Seda teevad niinimetatud „alamad" (low-level) graafikafunktsioonid.

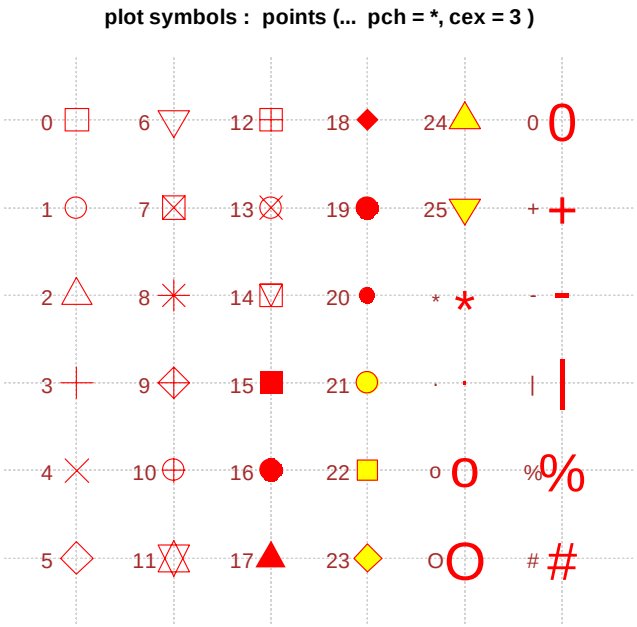
```
abline      # regressioonijoone vm kriipsude lisamine
arrows      # noolte või vuntside lisamine
points      # punktide lisamine
lines       # suvalisi punkte ühendavad jooned
axis        # koordinaatteljed
legend      # joonise legend
title       # joonise pealkiri
```

Graafikaparameetritest (nt joone paksus, joonise “valge serva” laius, punktide värv või kuju). Osa neist on muudetavad “kõrgtaseme” funktsioonides, osa ainult “alamates” funktsioonides või eraldi funktsiooniga par.

Parameetrid, mis muudavad punktide, joonte vms välimust	**Parameetrid, mis muudavad joonise üldilmet vms**
`col  # värv`	`mfrow  # joonise jagamine mitmeks alaosaks`
`cex  # punktide v teksti suurus`	`mar  # valgete servade laius`
`pca  # punkti kuju`	`las  #`
`lwd  # joone paksus`	
`lty  # joone tüüp (pidev, katkendlik #  jne)`	`...`
	`?par`

Par-funktsioon muudab parameetrite väärtuse kuni järgmise muutmiseni. Seda kasutatakse nii:

```
par()      # annab listi kõigist graafikaparameetritest
par()$mfrow          # parameetri mfrow väärtus
par(mfrow = c(2,2))   # joonise-ala jagatakse 2 veeruks ja 2 reaks
vana <- par(no.readonly=TRUE)  # ainult muudetavate parameetrite väärtused
par(vana)  # taastatakse eelmised väärtused
```



Üldistatud lineaarsed mudelid sotsiaalteadustes
Praktikum (SHPH.00.001, kevad 2010)

2. praktikum

1. Kas ikka on normaalne?

ESS andmestikus on tunnuseid, mille puhul normaaljaotuse eeldamine oleks liiast. Proovime näiteks tunnusega hhmmb *(Number of people living regularly as member of household).*

```
> attach(ESS)
> hist(hhmmb)
> barplot(table(hhmb))
```

Normaaljaotus eeldab pidevat tunnust, kuid viisakates leibkondades on ikkagi täisarv liikmeid. Leibkonna liikmete arv on "vasakult" piiratud: alla 1 liikme ei saa leibkonnas olla (sest kui keegi vastab küsimusele ja kedagi teist tema leibkonnas ei ole, siis ta ise ikkagi on). Seega negatiivsed väärtused on sellisel tunnusel keelatud.

Proovime teise tunnusega, emplno (*Number of employees respondent has/had*).

```
> table(emplno)
> hist(emplno)
```

Kas tõesti on rohkem kui 3000 vastajal olnud täpselt 66666 alluvat? Kui palju alluvaid siis peaks ESS valimil kokku olema? Kas ESS valimi alluvaid on rohkem või vähem kui Soomes ja Eestis elanikke? Lisaks tuleb arvestada ka ligi 30 inimesega, kellest igaühel on peaaegu sada tuhat alluvat. Kas Eestis ja Soomes on tõesti nii palju hiidsuurettevõtete juhte, või on tegu aprillinaljaga? Aga proovime vaadata kooditabelist (ESS2008coding.xls). Selgub, et 66666 tähendab "not applicable" ja 99999 "no answer". Mõistlikuma sagedustabeli või histogrammi saamiseks peaksime tunnused ümber kodeerima.

```
> emplno[emplno %in% c(66666, 99999)] <- NA
> table(emplno)
> hist(emplno)
```

2. Üldistatud lineaarne mudel

Üldistatud lineaarse mudeli ja oks kasutame funktsiooni glm, mille argumendid on sarnased eelmises praktikumis käsitletud lm-iga. Esimeseks argumendiks on valem (formula) kujul vasakpool ~ parempool, teise argumendiga ("family") määratleme sõltuva tunnuse jaotuse ja seosefunktsiooni, kolmandaga ("data") andmetabeli, lisaks võib kasu olla argumentidest "subset" ja "na.action", mis toimivad samamoodi nagu funktsiooni lm puhul. Seega saab funktsiooni glm kasutada ka tavalise lineaarse regressiooni jaoks, määratledes jaotusena normaaljaotuse ja seosefunktsioonina identsusfunktsiooni. Selles kontekstis tähistab R-i jaoks normaaljaotust märksõna "gaussian" – Carl Friedrich Gaussi auks, kellest "peaaegu" oleks saanud Tartu Ülikooli professor.

```
> mud1 <- lm(gener_trust ~gener_satisf + cntry, data=ESS)
> mud2 <- glm(gener_trust ~gener_satisf + cntry, data=ESS,
                          family =gaussian(link=identity))
> summary(mud1)
> summary(mud2)
```

Näeme, et mudelite parameetrid on täpselt samad, kuigi väljud on pisut erinev lm väljudnis on tuttavad R2, F ja standardviga, kuid glm kostitab meid selliste toredustega nagu AIC, jääkhälbimus ja iteratsioonide arv.

3. Poissoni mudel

Leibkonna suuruse ennustamiseks ei sobi lineaarne mudel mitmel põhjusel, aga proovime ikkagi, kasutades prediktoritena vanust ja maad:

```
> lm(hhmmb ~ cntry + agea)
```

Milline peaks olema leibkonna suurus, kui vastaja on 20-aastane eestlane? Aga 120-aastane soomlane?

Katsetame nüüd mõistlikuma mudeliga:

```
> mud1 <- glm(hhmmb ~ cntry + agea, family=poisson, data=ESS)
> summary(mud1)
```

Ainuke puudus siin on, et tegelikult sõltuv tunnus ei ole ka päris Poissoni jaotusega: viimasel on ka 0 võimalik väärtus (ja teinekord väga sagedane väärtus!), kuid leibkonna liikmete arvu puhul see nii ei ole. Kui me tahaksime, et Poissoni jaotus ikka päriselt ka võiks sobida, siis võiksime teha mudeli "ülejäänud leibkonnaliikmete" kohta, st vastajat ennast arvestamata.

```
> yll <- hhmmb - 1   # ülejäänud leibkonnaliikmete arv
> mud2 <- glm(yll ~ cntry + agea, family=poisson, data=ESS)
> summary(mud2)
```

Nüüd peaks tulemuseks olema midagi sellist:

```
# Coefficients:
#               Estimate Std. Error z value Pr(>|z|)
# (Intercept)  1.3723784  0.0342530  40.066  < 2e-16 ***
# cntryFI     -0.1732184  0.0253680  -6.828  8.6e-12 ***
# agea        -0.0178253  0.0006975 -25.557  < 2e-16 ***
```

Mida selliste tulemustega peale hakata? Proovime kõigepealt kasutada seda mudelit tundmatu inimese (nt 120-aastase soomlase) pereliikmete arvu ennustamiseks.

```
> jama <- lm(yll ~ cntry + agea) # tavaline lineaarne mudel
> coef(jama)[1] + coef(jama)[2] + coef(jama)[3] * 120
>  # sama mis   3.11478   + (  -0.28701 ) +    -0.02788  * 120
>  exp(coef(mud2)[1] + coef(mud2)[2] + coef(mud2)[3] * 120)
```

```
> # sama mis  exp(1.37238 + (-0.17322) + -0.01783 *120)
> # eriti kaval oleks seda teha nii: exp(sum(coef(mud2) * c(1,1, 120)))
```

Lineaarne regressioon ennustab negatiivset "ülejäänud pereliikmete" arvu, Poissoni regressiooni puhul ei saa seda juhtuda. (Ennustus ei ole küll täisarv, kuid siin tuleks rakendada loovat tõlgendamist: 0.39-st võiks mõelda nii, et enamikul sellistel vastajatel ei ole üldse teisi pereliikmeid, kuid vähemusel on üks või kaks [mõnel ka kolm, aga neid on eriti vähe, ja neli juba peaaegu mitte kellelgi jne]).

Ülesanne. Kui palju oleks eelpool hinnatud Poissoni mudeli järgi "teisi pereliikmeid" 20-aastasel eestlasel? 30-aastasel soomlasel?

4. Üledispersiooni küsimus

Vaatame mudel2 kokkuvõtet uuesti:

```
> summary(mud2)
```

Kui jääkhälbimus on (palju) suurem kui vabadusastmete arv, siis on tegu üledispersiooniga, mis võib tähendada, et mudel on puudulik (tuleks lisada veel mingeid prediktoreid) või standardne Poissoni jaotus ei sobi, st ei saa eeldada, et jaotuse keskväärtus ja dispersioon oleksid võrdsed. Sel juhul saab kasutada "quasipoisson" jaotust (kuigi on ka teisi lahendusi, mis võivad mõnes olukorras olla paremad, nt negatiivne binoomjaotus, samuti vt lisamoodulit pscl ja http://cran.r-project.org/web/packages/pscl/vignettes/countreg.pdf).

Üledispersiooni testimine on ka võimalik, aga pole just päris kindel, kas see on alati (ja eriti mitte-ideaalsel juhul ehk tavaliste andmete korral) hea mõte. Üks selline test näiteks tuvastas üledispersiooni küllalt sageli ka R-i poolt genereeritud juhuslikel Poissoni jaotusega arvudel.

Aga nüüd paistab, et mudelis 2 on vähemalt mingil määral tegu üledispersiooniga, seega võiksime proovida sellega arvestada:

```
> mud3 <- glm(yll ~ cntry + agea, family=quasipoisson, data=ESS)
> summary(mud3)
```

Pange tähele, et mudeli parameetrid on samad, erinevad ainult p-väärtused! Selle näite puhul ei ole suurt vahet, kas üledispersiooni arvestada või mitte, nii vanus kui maa on sellest hoolimata olulised tunnused.

Nüüd aga proovime sellist näidet, kus üledispersiooni arvestamata jätmine viib vale järelduseni. Kasutame eelnevalt vaadeldud tunnust emplno, st alluvate arv.

```
> mud4 <- glm(emplno ~cntry + agea + sex, family= poisson)
> summary(mud4)
```

Hurraa! Kõik p-d on väga väiksed! Kuid samal ajal võiks tähele panna rida, mis ütleb meile, et

```
# Residual deviance: 6318.4  on 338  degrees of freedom
```

```
> mud5 <- glm(emplno ~cntry + agea + gndr, family= quasipoisson)
```

```
> summary(mud5)
```

Näeme, et üledispersiooni arvestades ei ole ükski tunnus mudelis oluline.

Ülesanne.

(1) Proovige viimast mudelit tavalise lineaarse regressioonanalüüsiga. Kas järeldused on samad?
(2) Lisage mudelisse mud5 soo ja riigi interaktsioon.
(3) Tekitage uus tunnus, kus enamikul vastajatest oleks väärtuseks 0 (st juhul, kui vastaja ei ole eraettevõtja, oleks tema heaks töötavate inimeste arv 0, mitte NA). Proovige samu mudeleid selle tunnusega.

Üldistatud lineaarsed mudelid sotsiaalteadustes
Praktikum (SHPH.00.001, kevad 2010)

2. ja 3. praktikumi lisamaterjal koduseks tegevuseks

1. Lisamoodulid

1. aprilli seisuga on CRAN-is (Comprehensive R Archive Network, http://cran.r-project.org/) avalikult kättesaadavad 2289 lisamoodulit (*packages*); peale selle leidub eriti genoomika-teemalisi (aga ka üldisema kasutusalaga) mooduleid Bioconductoris (http://www.bioconductor.org/); üksikutele entusiastidele mõeldud veidrusi levitab Omegahat (http://www.omegahat.org/).

Lisamooduli installeerimiseks CRAN-ist on mitu võimalust, kuid kõige lihtsam on, kui arvutis on olemas internetiühendus. Sel juhul piisab, kui kirjutate:

```
> install.packages("lisamoodul")
```

R küsib nüüd CRAN-i peegelserveri nime, selle valik on maitseasi. Kuid hoopis olulisem on küsimus, kuidas leida vajalik lisamoodul?

Täielik nimekiri: avage CRAN, vasakult menüüst valige "Packages".

Samateemaliste lisamoodulite grupid kannavad nime "Task views", nende uudistamiseks valige CRAN-i lehel vasakult menüüst … ülevalt kolmas link ☺. Samasse gruppi kuuluvaid mooduleid saab ka kõiki korraga installeerida, selles on aga kõigepealt vajalik lisamoodul ctv (CRAN Task Views).

```
> install.packages("ctv")
> install.views("AtaskView")
# Vaadake CRAN-ist järele, kas mõni "Task view" on tervikuna
# teile oluline.
```

Lisamoodulid võivad aeg-ajalt vananeda või tekib neile uusi võimalusi. Siis on kasu funktsioonist update.packages.

Lisamoodulite kasutamiseks kasutatakse funktsiooni library (järgneb sulgudes mooduli nimi); seda tuleb uuesti korrata iga R-i sessiooni alguses (st programmi sulgemisega haagitakse ka lisamoodulid lahti).

```
> library(ctv)
# ja nutikamad panid tähele, et alles nüüd on võimalik kasutada
# funktsiooni install.views!!!
```

2. Otsingutee

Kuidas saada teada, millised lisamoodulid (*packages*) on parajasti laetud, või millised andmetabelid külge haagitud? Tuleb meelde tuletada … või siis kasutada funktsiooni search:

```
> search()
# minul oli tulemus selline:
# [1] ".GlobalEnv"        "package:ctv"       "package:stats"
# [4] "package:graphics"  "package:grDevices" "package:utils"
# [7] "package:datasets"  "package:methods"   "Autoloads"
# [10] "package:base"
```

Kui olete attach'inud mõne andmetabeli, siis leiate sealt nimestikust ka selle, vahel lausa mitu (võimalik, et erinevat) koopiat sellest (kui attach-käsku on kogemata või meelega kasutatud mitu korda).

Kõiki külgehaagitud lisamooduleid ja andmetabeleid on võimalik ka lahti haakida – käsuga detach; samuti on võimalik vaadata nende sisu funktsiooniga ls ehk objects.

```
> objects("package:base")  # sama mis ls("package:base")
> objects(10) # praegu sama, st kui "base" moodul on otsinguteel kümnes
> detach("package:ctv")
> # detach(2) oleks andnud sama tulemuse, aga pärast eelmist
  # avaldist on otsingutee 2. positsiooni juba hõivanud
  # "package:stats": vt uuesti search()
> search()
```

Andmetabeli lisamisel otsinguteele (attach-korraldusega) tehakse sellest koopia, mis ei muutu, kui me muudame algset andmetabelit. See võib tekitada segadust, nagu ütleb ka help-fail:

> `?attach` ütleb:
> *The database is not actually attached. Rather, a new environment is created on the search path and the elements of a list (including columns of a data frame) or objects in a save file or an environment are _copied_ into the new environment. If you use '<<-' or 'assign' to assign to an attached database, you only alter the attached copy, not the original object. (Normal assignment will place a modified version in the user's workspace: see the examples.) For this reason 'attach' can lead to confusion.*

Seetõttu võiks üpriski tungivalt soovitada attach-käsku mõõdukalt, ja üldjuhul ainult siis, kui me algset andmestikku ei taha enam muuta. (Vastasel korral on muudetud andmetabel vaja uuesti attach'ida… või siis leppida sellega, et kasutatakse andmetabelit esialgsel kujul.)

```
> kukeleegu <- data.frame(ahhaa = 1, ohhoo=2)
> attach(kukeleegu)
> ahhaa
> kukeleegu$ahhaa  # sama väärtus
> kukeleegu$ahhaa <- 123223
> ahhaa   # sama mis enne
> kukeleegu$ahhaa  # uus väärtus
> detach(kukeleegu)
```

```
> attach(kukeleegu)
> ahhaa  # nüüd juba uus väärtus
```

3. Töölaua koristamine

Juba pärast paari praktikumi võite avastada, et töölaud ("workspace") on igasuguseid vanu muutujaid täis, ning enamikku neist ei ole enam vaja. Tahaks neist enamiku ära kustutada, kuid jätta alles ESS andmetabeli. Kuidas seda teha? Funktsioonil rm (teise nimega remove) on lisa-argument list, millega saab anda ette ükskõik kui pika nimekirja objektide nimedest, mida me tahame kustutada. Sellise nimekirja saame teha funktsiooniga ls (teise nimega objects), kustutades sealt need nimed, mida tahame alles jätta.

```
> ls() # objektide nimekiri töölaual
> kustutamiseks <- ls()
> kustutamiseks <- ls() # teeme seda kaks korda, et ka
# muutuja "kustutamiseks" kuuluks kustutamisnimekirja ☺
> fix(kustutamiseks)  # nüüd kustutame  nime "ESS"
# minul nägi algne nimekiri välja selline:
# c("emplno", "emplno1", "ESS", "jama", "kukeleegu", "kustutamiseks",
# "lkrspag", "ma", "mud1", "mud2", "mud3", "mud4", "mud5", "mudel1",
# "predage", "predetn", "predsex", "trtbdag", "y", "yll")
# kustutasin sealt "ESS" (ja sellele järgneva koma!!!)
> rm(list=kustutamiseks)
```

Vahel on juba ette teada, et mingi objekt ei kuulu säilitamisele, vaid on vahetulemus, mida on vaja järgnevates arvutustes. Koodifailis (File → new scipt …) võib sellised read paigutada funktsiooni local() sisse, sel juhul tekivad omistamisega objektid "ajutisse töökeskkonda", mis pärast kasutamist hävib pöördumatult ja jäägitult.

```
keerulinearvutus <- local({
    vahetulemus1 <- 1:10
    vahetulemus2 <- exp(vahetulemus1)
    vahetulemus3 <- log(vahetulemus2)
    vahetulemus4 <- vahetulemus3 - 1:10
    sum(vahetulemus4)
})
```

Püüdke ennustada, mis on keerulise arvutuse tulemuseks. Kas ennustus läks täppi? – Funktsiooniga ls kontrollime nüüd, et ühtegi vahetulemustest töölauda risustamas ei ole, mis praegu oli ju funktsiooni local kasutamise eesmärgiks. Vahetulemuste suunamine töölauale on siiski võimalik, kasutades omistamiseks operaatorit `<<-`. Kui tahame lihtsalt midagi proovida, siis võib kasutada ka ühendit `local(browser())` , millest saab välja vajutades kaks korda järjest <enter>.

4. ESS-i andmetabeli parandamine

Kasutame kodukootud funktsioon read.ess failist "read.ess.R". See kasutab funktsiooni read.spss lisamoodulist foreign, kuid üritab viimase puudusi parandada: SPSS-i faili meta-andmeid (nt andmed puuduvate andmete koodide kohta) kasutatakse ära nii, need märgitakse ka R-is puuduvateks andmeteks. Meta-andmed (sh ka tunnuste pikad nimed, samuti tunnuste koodid ja nende tähendused) salvestatakse andmetabeli atribuudina nimega x.

```
> read.ess("ESS2008_EstFin.sav")->ESS2
> names(attributes(ESS2)$x)
> attributes(ESS2)$x$variable.labels -> tunnustenimed
> attributes(ESS2)$x$label.table -> koodid
```

Nüüd saame mugavamalt kasutada tunnuste pikki nimesid, samuti vaadata tunnuste koode ilma Exceli abita. Näiteks Schwartzi väärtusteküsimuste nimed saame kätte nii:

```
> tunnustenimed[508:528]
```

Kasutame neid näiteks koondsagedustabeli esimese tulbana:

```
> tabel<-sapply(ESS2[,508:528], table)
> tabel
# ilusam oleks teistpidine tabel:
> t(tabel)   # t --- "transpose"
> tabel <- t(tabel)
> rownames(tabel)<-tunnustenimed[508:528]
```

Üldistatud lineaarsed mudelid sotsiaalteadustes
Praktikum (SHPH.00.001, kevad 2010)

3. praktikum

1. Seos kahe dihhotoomse tunnuse vahel

Vaatame ESS-i andmestikust tunnuseid sgnptit (*signed petition last 12 months*) ja pbldmn (*taken part in lawful public demonstration last 12 months).* Mõlemal tunnusel 1=jah, 2=ei, ülejäänud variandid on keeldumine jne.

```
> attach(ESS)
> table(sgnptit)
> table(pbldmn)
# kodeerime tunnused ümber (NB! Tekib tunnuse „lokaalne koopia“)
> sgnptit[sgnptit>2] <- NA
> pbldmn[pbldmn>2] <- NA
```

(Hiljem uurime, kuidas lugeda sisse ESS-i andmestik nii, et kõik puuduvate andmete koodid asendataks automaatselt NA-ga.)

```
> table(sgnptit, pbldmn)
```

Ülesanne. Kodeerige tunnused sgnptit ja pbldmn ümber nii, et 1=jah, 0=ei.

Kahe tunnuse vahelist seost on ka lihtne joonisel kujutada, kõige lihtsam nii:

```
> plot(sgnptit, pbldmn)
# kas see joonis on informatiiivne?
# proovime punkte üksteise pealt eemale nihutada, kasutades
# funktsiooni jitter, mis lisab andmetele juhuslikku müra
> plot(jitter(sgnptit), jitter(pbldmn))
> plot(jitter(sgnptit,2), jitter(pbldmn,2))
# lisame värvi: punane=eesti, sinine=soome
> plot(jitter(sgnptit,2), jitter(pbldmn,2),
         col=ifelse(cntry=="EE", "red", "blue"))
# punktid väiksemaks …
> plot(jitter(sgnptit,2), jitter(pbldmn,2),
      col=ifelse(cntry=="EE", "red", "blue"), pch=".", cex=2)
```

See joonis on parem kui esimene, aga kahe tunnuse seose visualiseerimiseks on olemas ka paremaid viise. R-is on sisse ehitatud funktsioonid assocplot ja mosaicplot. Esimene neist kujutab sagedusi üles- või allapoole suunatud tulpadena vastavalt sellele, kas sagedus on oodatust suurem või väiksem. Teisel vastab igale sagedustabeli lahtrile ristkülik, mille pindala on proportsionaalne sagedusega.

```
> assocplot(table(sgnptit, pbldmn))
> mosaicplot(factor(sgnptit)~pbldmn) # sõltuv tunnus peab olema faktor
```

Ülesanne. Proovige näidata joonisel, kuidas sõltub demonstratsioonidel osalemine ja petitsioonidega ühinemine vastaja elukohamaast.

2. Dihhotoomse tunnuse seos pideva tunnusega

```
> plot(sgnptit~gener_trust)
> plot(jitter(sgnptit) ~gener_trust)
> cdplot(factor(sgnptit)~gener_trust, subset=cntry=="FI")
> spineplot(factor(sgnptit)~gener_trust, subset=cntry=="FI")
```

Ülesanne. Illustreerige petitsioonidele alla kirjutamise tõenäosuse sõltuvust vanusest eraldi Eestis ja Soomes. Kasutage graafikaparameetrit "mfow", et Eesti ja Soome pildid panna kõrvuti.

3. Logit-seos

Logistiline regressioon ei modelleeri otseselt dihhotoomse tunnuse keskväärtust (st sündmuse esinemise tõenäosust), vaid funktsiooni sündmuse esinemise šanssidest, kus "šansid" ehk "tõenäosussuhe" on p/(1-p). Kui sündmuse tõenäosus on 25%, siis selle sündmuse "šansid" on "üks kolme vastu", ühe korra kohta, kus juhtub NII, tuleb kolm korda, kus juhtub teisiti. Logit-funktsioon on siis logaritm šanssidest ehk log(p/(1-p)) ehk samaväärselt log p – log(1-p).

Defineerime järgnevaks katsetamiseks R-is logit-funktsiooni ja selle vastandi.

```
> logit <- function(x) log(x) - log(1-x)
# ja vastandfunktsioon, millest on hiljem kasu
> expit <- function(x) exp(x) / (1+exp(x))
> plot(logit)
# näeme, et funktsioon tõuseb järsult 0 ja 1 läheduses
# … ja on x=0.5 ümbruses üsna lame
> logit(0.5)
> logit(0.55)
> logit(0.9)
> logit(0.95)
# peale selle on logit funktsioonil väärtused ainult
# vahemikus 0-1, ning logit(1)=∞, logit(0)=-∞
> logit(0)
> logit(1)
> logit(-1)
> logit(1.01)
# logiti vastandfunktsiooni väärtused on alati vahemikus
# 0 …1
> expit(0)
> expit(3)
> expit(-3)
> 1-expit(-3)
> expit(5)
> expit(15)
> expit(20)
# proovime, kas expit on tõepoolest logit-i pöördfunktsioon
> expit(logit(0.91))
```

```
> logit(expit(9.551))
```

Ülesanne. Proovige vastata järgnevatele küsimustele katsetades eelnevalt defineeritud funktsioonidega logit ja expit. (1) Miks ei saa logistiline regressioon (logit y = a + bx + ….) kunagi ennustada sündmuse tõenäosuseks näiteks 105% või -23%? (2) Miks ei ole võimalik logistilist regressiooni teha nii, et teisendame kõigepealt dihhotoomse tunnuse [mille väärtused algselt on 0 ja 1] logit-funktsiooni abil ning siis teeme tavalise lineaarse regressioonanalüüsi?

4. Šansside suhe

Üks võimalus kahe dihhotoomse tunnuse seost arvuliselt väljendada on šansside suhe (eesti keeles ka riskisuhe; "odds ratio" – sageli lühendatakse OR). Ka logistilise regressiooni tulemusi on mugav väljendada riskisuhtena, teisendamata regressioonikordajate tõlgendamine on keerulisem. Proovime näiteks leida seose valimistel osalemise ja elukohamaa vahel (jättes välja need, kel valimisõigus puuduv või kes mingil muul põhjusel küsimusele ei vastanud):

```
> table(vote, cntry)
> vote[vote>2] <- NA   # tekib tunnuse koopia töölauale!
> vote <- vote == 1    # väärtuseks TRUE, kui käis valimas
> table(vote, cntry)
# tulemus peaks olema umbes selline:
###         cntry
### vote      EE   FI
###   FALSE  513  323
###   TRUE   940 1603
> 940/513  # eestlase "šansid" käia valimas e. "hääletamisrisk"
```

Ülesanne. Mitu korda on soomlaste "hääletamisrisk" suurem kui eestlastel? [Tulemuseks ongi šansside suhe ja õige vastus on umbes 2.71, kuid proovige see eelneva tabeli põhjal kätte saada.]
/Vabatahtlikele koduseks kuugeldamiseks./ Kas ESS-i valimi põhjal saadud hääletajate protsendid on tõepärased? Oletades, et tegu on representatiivse valimiga (mida ta päris ei ole), kui palju neist, kes väidavad, et käisid hääletamas, tegelikult tõenäoliselt ei käinud?

5. Logistiline regressioon

Logistilise regressiooni kasutamiseks peab tunnusel olema täpselt kaks väärtust (puuduvate andmete koodid ei lähe arvesse); R-ile sobivad väärtused 0 ja 1, või siis tõeväärtuse-tüüpi tunnused (TRUE / FALSE, kus TRUE ==1 ja FALSE==0). Proovime eelmist näidet lahendada logistilise regressiooni abil:

```
> mud1 <- glm(vote~cntry, family=binomial)
> summary(mud1)
> exp(coef(mud1)[2]) # võrdle tabeli põhjal leitud š.suhtega
```

Näeme, et soomlaseks olemine tõstab hääletusriski tervel 2.71 korda (võrreldes eestlaseks olemisega). Š.suhte väärtus tunnuste sõltumatuse korral on 1; epidemioloogias on tavalised nimetused "riskitegur" (kui OR>1) ja "kaitsev tegur" (kui OR<1) – selles näites siis soomlaseks olemine on valimistel osalemise riskitegur.

Selle näite puhul saaksime arvutada ka hääletamas käimise tõenäosused: baaskategooria (eestlased) jaoks oleks see pöörd-logit regressioonimudeli vabaliikmest, soomlaste jaoks siis pöörd-logit[vabaliige+regressioonikordaja]:

```
> coef(mud1)
> expit(coef(mud1)[1])
> expit(sum(coef(mud1))
# kontrollige, kas need on võrdsed hääletanute %-dega
# eestlaste ja soomlaste hulgas!
```

Kokkuvõtteks: šansside suhte saame lihtsalt eksponentfunktsiooni abil, päris tõenäosuste ennustamiseks on tarvis pöörd-logit funktsiooni. Logistilise regressiooni parameetrite tõlgendatavus sõltub sellest, mis tüüpi uuringuga on tegu: näiteks juhtkontrolluuringute (kus logistilist regressiooni sageli kasutatakse!) puhul on baastõenäosus määratud uuringu disainiga ja seetõttu mudeli vabaliikme tõlgendamisel ei ole mõtet; järelikult ei saa siis ka ilma lisainfota mudeli põhjal üldse mingeid tõenäosusi arvutada. Seevastu šansside suhete leidmine on alati võimalik.

Leiame nüüd šanside suhtele ka usaldusvahemiku.

```
> exp(coef(mud1))    #š.suhe on see teine number seal!
> confint(mud1)      # usaldusvahemik regr.parameetrile
> exp(confint(mud1)) # usaldusvahemik š.suhtele
```

Ülesanne. Kas vanus (tunnus agea) on hääletamise riskitegur või hoopiski kaitseb selle eest? Joonistage ka "cdplot" (conditional density plot, "tingliku tiheduse diagramm") eestlaste ja soomlaste jaoks eraldi; millisest vanusest alates [silma järgi hinnates] on eestlastel hääletamise tõenäosus juba üle 50%?

6. Kuidas hakkama saada paljude tunnustega?

Oletame nüüd, et me tahame uurida, kuidas valimistel osalemine (tunnus vote) sõltub väärtustest; ESS andmestikus on tunnused 508-528 Schwartzi väärtusteküsimustik:

508	ipcrtiv	Important to think new ideas and being creative
509	imprich	Important to be rich, have money and expensive things
510	ipeqopt	Important that people are treated equally and have equal opportunities
511	ipshabt	Important to show abilities and be admired
512	impsafe	Important to live in secure and safe surroundings
513	impdiff	Important to try new and different things in life
514	ipfrule	Important to do what is told and follow rules
515	ipudrst	Important to understand different people
516	ipmodst	Important to be humble and modest, not draw attention
517	ipgdtim	Important to have a good time
518	impfree	Important to make own decisions and be free
519	iphlppl	Important to help people and care for others well-being
520	ipsuces	Important to be successful and that people recognise achievements
521	ipstrgv	Important that government is strong and ensures safety
522	ipadvnt	Important to seek adventures and have an exiting life
523	ipbhprp	Important to behave properly
524	iprspot	Important to get respect from others
525	iplylfr	Important to be loyal to friends and devote to people close
526	impenv	Important to care for nature and environment
527	imptrad	Important to follow traditions and customs
528	impfun	Important to seek fun and things that give pleasure

Vastusekategooriad olid järgmised:

9=No answer | 8=Don't know | 7=Refusal | 6=Not like me at all | 5=Not like me | 4=A little like me | 3=Somewhat like me | 2=Like me | 1=Very much like me

Kontrollime kõigepealt sagedustabeleid.

```
> table(ipcrtiv)
> table(imprich)
> table(ipeqopt)
# ja nõnda veel 21-3 korda! Väike näpuharjutus ei tee ju paha.
```

Funktsiooniga sapply saame teha sagedustabeli ka kõigi nende 21 tunnuse jaoks korraga:

```
> sapply(ESS[,508:528], table)
```

Näeme, et päris kohe selliste andmetega suurt peale hakata pole, enne tuleks ära muuta need puuduvate andmete koodid (9 == "no answer"); samuti oleks loogilisem, kui suurem number näitaks suuremat nõusoleku määra. Proovime ka seda teha kõigi tunnustega korraga; selleks kasutame korraldust for, mille süntaks on järgmine:

```
for(MUUTUJA in VEKTOR) {tee_midagi}
#  "Muutujale" omistatakse järjest väärtusi, mis võetakse
#  "vektorist", ning iga sellise väärtuse puhul korratakse
#  avaldist {tee_midagi}.
```

```
> for(ii in 1:10) print("Juhhei!")
> for(ii in 1:10) print(ii)
# teise ettevalmistusena defineerime funktsiooni "tee_korda",
# mis kustutab 6-st suuremad väärtused ja seejärel pöörab
# tunnuse skaala teistpidi
# funktsiooni väärtuseks on viimane rida definitsioonist!
> tee_korda <- function(x) {
      x[x>6] <- NA
      x <- 7-x
      x
      }
# proovime, kas töötab
> table(ipcrtiv)
> table(tee_korda(ipcrtiv))
# nüüd muudame selle funktsiooni abil algandmestikku
> for(ii in 508:528) ESS[,ii] <- tee_korda(ESS[,ii])
> table(ipcrtiv)
# häda attach'iga: detach(ESS); attach(ESS)
> table(ipcrtiv)
```

Proovime kõigepealt uurida, kuidas nimekirjas esimene väärtus (ipcrtiv, importance of being creative) on seotud eestlaste valimisaktiivsusega:

```
> mud1 <- glm(vote~ipcrtiv,  family=binomial,
subset=cntry=="EE")
> exp(coef(mud1)[2])   # šansside suhe
```

```
> summary(mud1)
> coef(summary(mud1))
> coef(summary(mud1))[2,4]   # p-väärtus
> exp(confint(mud1))
> exp(confint(mud1))[2,]  # jätame välja vabaliikme rea
# seose iseloomustamiseks piisab 2-4 arvust:
# šansside suhe, selle usaldusvahemik ja ehk ka p-väärtus
# teeme funktsiooni, mis eraldab mudelist ainult need 3 numbrit

> get3 <- function(x){
    mudel <- glm(vote~x, family=binomial, subset=cntry=="EE")
    OR <- exp(coef(mudel)[2])
    CI <- exp(confint(mudel)[2,])
    P <- coef(summary(mudel))[2,4]
    c(OR, CI, P)
    }
> get3(ipcrtiv)  # kas tulemus on sama, mis enne?
```

Nüüd on käkitegu juba defineeritud funktsiooni kasutada kõigi väärtustega:

```
> tulemused <- sapply(ESS[,508:528], get3)
> tulemused
```

Et asi oleks ülevaatlikum, siis teeme ka joonise:

```
> plot(tulemused[1,])
> arrows(1:21, tulemused[2,], 1:21, tulemused[3,], length=0)
# fn arrows joonistab "nooled", mis on etteantud koordinaatide vektoritega:
# x0 ja y0 (esimesed 2 argumenti): noolte alguspunktid
# x1 ja y1 (järgmised 2 argumenti): noolte lõpp-punktide koordinaadid
# length: noolepea "pikkus"
> abline(h=1)
```

Kodus proovimiseks jääb selle joonise ilusamaks tegemine: vaja oleks lisada x-telje jm tähistused, y-telje ulatus parandada nii, et kõik "vuntsid" ära mahuks jne.

Lõpuks teeme ka "täieliku" mudeli, millest järgmine kord saame alustada.

```
> koikvaartused <- paste(names(ESS)[508:528], collapse="+")
> koikvaartused
> valem <- paste("vote", koikvaartused, sep=" ~ ")
> valem
> glm(as.formula(valem), family=binomial, subset=cntry=="EE")
```

Ülesanne: lisage viimasele mudelile maa, sugu ja vanus.

Üldistatud lineaarsed mudelid sotsiaalteadustes
Praktikum (SHPH.00.001, kevad 2010)

4. praktikum

1. Tunnuste valik mudelisse

R-is on olemas protseduurid, mis võimaldavad regressioonimudelisse teatud algoritmi järgi automaatselt tunnuseid valida (*stepwise xyz*: "sammregressioon" ehk "sammuviisiline regressioon" / "sammuviisiline tunnuste valik", samuti "parima alamhulga valik" / *best subset selection*). Enne nende kasutamist peaksime aga koostama keerulise mudeli, kus on palju prediktoreid, mille hulgast valida. Oletame näiteks, et tunnust y ennustame tunnustest x1…x50: sel juhul oleks R-i kood lineaarse regressiooni jaoks:

```
> lm(y ~ x1+ x2+ x3+ x4+ x5+ x6+ x7+ x8+ x9+ x10+ x11+ x12+ x13+ x14+ x15+ x16+ x17+ x18+
x19+ x20+ x21+ x22+ x23+ x24+ x25+ x26+ x27+ x28+ x29+ x30+ x31+ x32+ x33+ x34+ x35+ x36+ x37+
x38+ x39+ x40+ x41+ x42+ x43+ x44+ x45+ x46+ x47+ x48+ x49+ x50)
```

Küllaltki ebamugav oleks neid kõiki x-e käsitsi välja kirjutada – lihtsam oleks, kui seda teeks meie eest arvuti. Kasutame funktsioone paste (tähtede, numbrite jm "kokku kleepimiseks") ja as.formula (tekstijupist valemi tegemiseks.). Katsetame kõigepealt funktsiooni paste argumente sep ja collapse:

```
> paste("tere", "hommikust", sep=" ")
> paste(c("tere", "hommikust"), collapse = "")
> paste("y", "x", sep="~")
> paste("x", 1:10, sep="*")
> paste("x", 1:10, sep="", collapse=" + ")
> parempool <- paste("x", 1:10, sep="", collapse=" + ")
> valem <- paste("y", parempool, sep=" ~ ")
> valem
```

Funktsioon as.formula on palju lihtsam: seda on vaja ainult tekstijupist valemi tekitamiseks (nii et tulemust saaks kasutada funktsioonide lm, glm jne esimese argumendina):

```
> as.formula("y ~ x")
> as.formula(valem)
```

Ülesanne. Proovige koostada valem, kus tunnus "y" sõltub andmestiku ESS tunnustest nr 10-80. Andmestiku tunnuste nimed saate funktsiooniga names: nt names(ESS)[1:9]

Teine viis pika valemi kirjutamiseks on kasutada paremal pool lühendit ".", mis tähendab, et sisse tuleks võtta kõik tunnused mingist andmetabelist peale sõltuva tunnuse enda. Kui tahame mingist andmetabelist kasutada ainult osa tunnuseid, siis on mõnikord lihtsam teha ajutine andmetabel, kus on sees ainult meile vajalikud tunnused. Proovime nii ennustada hääletamisel osalemist Schwartzi väärtuste abil (ESS andmestikus tunnused 508-528). Ajutisest andmetabelist tuleks veel välja jätta puuduvad andmed – muidu võib sammuviisiline

valikuprotseduur sattuda hätta (erinevatel sammudel on tegu natuke erinevate andmestikega, kuna mõnel vastajal on puudu ainult osa prediktoreid).

```
> ajutine <- na.omit(cbind(vote, ESS[,508:528]))
> mud <- glm(vote ~., data=ajutine, family=binomial)
> mud.step <- step(mud)
> summary(mud.step)
```

Sammuviisilise protseduuir tulemuseks (vt ka ?step) on lõplik mudel – esialgsest mudelist jäetakse vastavalt AIC suurenemisele või vähenemisele tunnuseid välja või võetakse sisse tagasi. Sammu suunda saab määrata parameetriga direction, mille võimalikud väärtused on "forward", "backward" ja "both"; veel lisavõimalusi pakub funktsioon stepAIC moodulist MASS. Kui tahame AIC asemel kasutada kriteeriumina BIC-i, siis peaksime lisama parameeri k väärtusena logaritmi vastajate arvust.

```
> dim(ajutine)
> mud.step.bic <- step(mud, k=log(2964))
> summary(mud.step.bic)
```

Lisamoodulis leaps on funktsioon regsubsets, mis võimaldab kasutada "parima alamhulga" valikuprotseduuri.

Ülesanne. Lisage eelmisele mudelile maa tunnus (cntry) – selleks tuleks teha uus ajutine andmetabel. Valige Schwartzi väärtuste (+ asukohamaa) hulgast kasutades kriteeriumina BIC-i ja "tagurpidi" valikut.

2. Tunnuste teisendamine

Mittelineaarsed seosed võivad olla hästi sobitatavad ka tavalise lineaarse regressioonimudeliga – selleks võib kasutada tunnuste teisendusi. Proovime genereeritud andmetega, millal võiks sobida logaritmiline teisendus.

```
> x <- seq(1, 10, .1)
> y <- exp(x)
> plot(x,y)  # log.teisendus sobib, kui x ja y seos on "umbes"
            # sellise kujuga
> plot(x,log(y))
```

Joonistame veel mõned mittelineaarsed seosed (seose lineaarseks muutmiseks tuleks siis kasutada y-tunnuse puhul vastupidist funktsiooni – sama tulemuse saame lisades x-i mittelineaarse teisenduse lineaarses mudelis prediktorina).

```
> par(mfrow=c(3,2))
> plot(x, log(x))
> plot(x, exp(x))
> plot(x, x^2)
> plot(x, sqrt(x))
> plot(x, x^3)
> plot(x, x^(1/3))
```

Euroopa Liit
Euroopa Sotsiaalfond

Eesti tuleviku heaks

Üldistatud lineaarsed mudelid sotsiaalteadustes
Praktikum (SHPH.00.001, kevad 2010)

Lisa 1: Mudelite võrdlemine ja interaktsioonid

1. Mudelite võrdlemine

Mudelite võrdlemiseks saab kasutada funktsiooni anova, mis klassikalise regressioonanalüüsi puhul annab tulemuseks kaht või enamat mudelit võrdleva dispersioonanalüüsi, üldistatud lineaarse mudeli korral hälbimuse analüüsi (*analysis of deviance*). Selleks, et võrdlusel oleks mõte, peavad mudelid olema hierarhiliselt järjestatud, kas lihtsamast keerulisemani või vastupidi; keerulisem mudel peab sisaldama kõiki lihtsama mudeli parameetreid ning lisaks veel mõnda. Kunstlik näide (proovimiseks asendage y-d ja x-id reaalsete sobilike tunnustega: nt y peab esimeses näites olema pidev tunnus, teises dihhotoomne jne)

```
> mud1 <- lm(y ~ x1)
> mud2 <- lm(y ~ x1 + x2)
> mud3 <- lm(y ~ x1 + x2 + x3)
> anova(mud1, mud2, mud3)
```

Üldistatud lineaarsete mudelite puhul vt ka ?anova.glm – siin on olulisuse testimiseks mitu võimalust ning test tuleb eraldi tellida kasutades lisaparameetrit test:

```
> mud1 <- lm(y ~ x1, family=binomial)
> mud2 <- lm(y ~ x1 + x2, family=binomial)
> mud3 <- lm(y ~ x1 + x2 + x3, family=binomial)
> anova(mud1, mud2, mud3, test="Chisq")
```

Õpetus (vt uuesti: ?anova.glm) ütleb, et fikseeritud dispersiooniparameetriga mudelites (Poissoni ja logistiline regressioon, st family parameeter on "binomial" või "poisson") on sobilik χ^2-test (`test="Chisq"`), vaba dispersiooniparameetriga jaotuste puhul (family parameeter kas "gaussian", "quasibinomial" või "quasipoisson") on sobilikum F-test (`test="F"`).

2. Interaktsioonid

Praktikumis proovisime automaatvaliku abil, kas Eestis ja Soomes ennustavad valimas käimist erinevad väärtused.

```
vaartused <- names(ESS)[508:528]
F1 <- paste("vote ~ (", paste(vaartused, collapse=" + "), ") * cntry")
# tekitame valemi, kus oleks lisaks väärtuste peamõjudele
# ka iga väärtuse interaktsioon maa-tunnusega
F1 <- as.formula(F1)
glm(F1, family=binomial)
summary(glm(F1, family=binomial))
step(glm(F1, family=binomial))
# step-funktsiooni ei saa kasutada, kui erinevatel sammudel
# on puuduvate andmete muster erinev. Kõigepealt peame tekitama
# ajutise andmetabeli, kus puuduvad andmed on välja jäetud
```

```
ajutine <- na.omit(cbind(vote, cntry, ESS[,508:528]))
head(ajutine)
step(glm(F1, family=binomial, data=ajutine))
.Last.value -> viimnemudel
# unustasime funktsiooni "step" tulemuse salvestada
# õnneks ei ole see tulemus veel pöördumatult kadunud:
# muutujas .Last.value salvestatakse viimase avaldise väärtus
# seda selgitab järgnev näide
1+1
.Last.value
2+2
.Last.value
##
summary(viimnemudel)
## nüüd proovime sama ka rangema (BIC) kriteeriumiga
## system.time avaldise ümber on ajakulu hindamiseks
viimnemudel.bic   <-   step(glm(F1,   family=binomial,   data=ajutine),
k=log(2964))
system.time(viimnemudel.bic    <-    step(glm(F1,    family=binomial,
data=ajutine), k=log(2964)))
```

Hoiatuseks, et automaatvalikuprotseduuride läbimõtlemata kasutamine ei ole tervislik. Kõige parem on mudel koostada teooria põhjal; kunagi ei maksa loota, et sammregressioon annab meile automaatselt õige ja kehtiva tulemuse nii, et mõelda pole vajagi (rääkimata tulemuste tõlgendamisest). Praeguse näite eesmärgiks on eelkõige illustreerida koosmõjude tähendust – seetõttu on esialgu tähtis ka mudeli ja tõlgenduse lihtsus (lõpliku tõe väljaselgitamine väärtuste mõjust hääletamas käimisele jääb vabatahtlikele koduseks ülesandeks).

Eelpool kirjeldatud põhjustel ei tasu vähemalt selles näites süveneda sammregressiooniga AIC ja BIC kriteeriumi kasutades abil saadud mudelite võrdlemisse: lõpliku mudeli valik on ikkagi uurija enda otsustada. Esialgu piisab pealiskaudsest hinnangust, et BIC abil saame lihtsama mudeli (st vähem prediktoreid ja koosmõjusid).

Valime nüüd pooljuhuslikult kolm väärtust, mille mõju valimas käimisele paistab olevat Eestis ja Soomes oluliselt erinev: iprspot (Important to get respect from others), ipeqopt (Important that people are treated equally and have equal opportunities) ja impfree (Important to make own decisions and be free).

```
> summary(glm(vote~(iprspot + ipeqopt + impfree) * cntry,
binomial))
# saame sellise tulemuse:
                Estimate Std. Error z value Pr(>|z|)
(Intercept)      1.06779    0.33718   3.167 0.001541 **
iprspot         -0.16672    0.04296  -3.881 0.000104 ***
ipeqopt         -0.08068    0.04887  -1.651 0.098745 .
impfree          0.09168    0.05180   1.770 0.076748 .
cntryFI         -0.03587    0.56543  -0.063 0.949423
iprspot:cntryFI  0.36520    0.07169   5.094  3.5e-07 ***
ipeqopt:cntryFI  0.23149    0.08423   2.748 0.005992 **
impfree:cntryFI -0.22450    0.08582  -2.616 0.008900 **
```

Kuidas seda tõlgendada? Püüame kirjutada ennustusvalemid eraldi soomlaste ja eestlaste jaoks:

Eestlastel: *logit vote = 1.07 – 0.17* iprspot -0.08 * ipeqot + 0.09 * impfree)*
Soomlastel: *logit vote = (1.07 -0.036) + (-0.17 + 0.37)* iprspot + (-0.08 + 0.23) * ipeqopt + (0.09 – 0.22) * impfree*

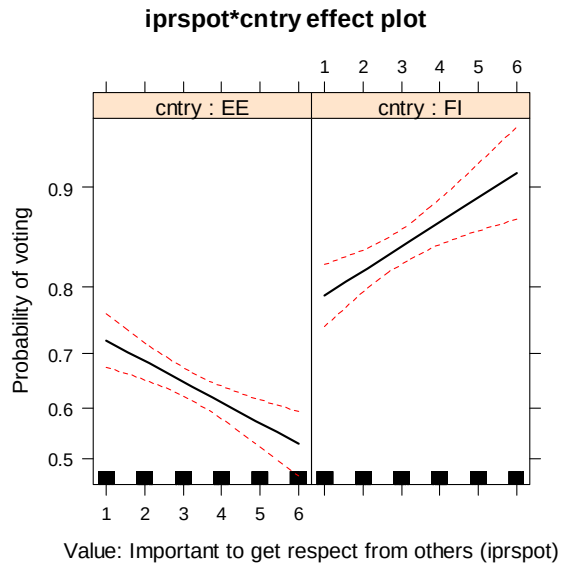
Need valemid varjavad lihtsat tõsiasja, et koosmõju (interaktsiooni) puhul on tegu lihtsalt tunnuste korrutisega. Soomlaste ja eestlaste eristamiseks kasutasime faktorit (cntry), mille program teisendas dihhotoomseks indikaatortunnuseks, mille väärtus eestlaste puhul on 0 ja soomlastel 1. See tähendab, et nii eestlaste kui soomlaste puhul kehtib üldine ennustusvalem:

*logit vote = (1.07 -0.036*FI) + (-0.17 + 0.37*FI)* iprspot + (-0.08 + 0.23*FI) * ipeqopt + (0.09 – 0.22*FI) * impfree*

Vahemärkusena: maa-tunnust saaks mudelis kasutada ka teistmoodi, näiteks nii, et eestlastel oleks indikaatortunnuse väärtuseks -1 ja soomlastel +1. Sel juhul tuleks ka mudeli parameetreid tõlgendada teistmoodi: näiteks maa peamõjule vastav regressioonikordaja tuleks siis eestlastel lahutada vabaliikmest, kuid soomlastel sellele liita; samamoodi maa ja väärtuste interaktsioonidega. Sellised mudelid on matemaatiliselt samaväärsed, st see, kumba eelistada, sõltub tõlgenduse lihtsusest ja mõttekusest.

Interaktsioonide tõlgendamiseks on kõige parem neid kujutada visuaalselt. Selleks on hea kasutada John Foxi kirjutatud lisamoodulit effects. Proovime selle abil illustreerida eelpool avastatud koosmõjusid.

```
> install.packages("effects")
> library(effects)
> glm(vote~(iprspot + ipeqopt + impfree) * cntry, family=binomial)->mud1
> plot(allEffects(mud1), "iprspot:cntry")
> plot(allEffects(mud1), "iprspot:cntry", ylab="Probability of voting",
   xlab="Value: Important to get respect from others (iprspot)")
> plot(allEffects(mud1), "impfree:cntry", ylab="Probability of voting",
   xlab="Value: Important to be free (impfree)")
```



Siin vaatlesime "pideva" tunnuse (kuigi kuue-punki skaala vastused tegelikult muidugi ei ole pidevad tunnused) interaktsiooni dihhotoomse tunnusega. Kahe pideva tunnuse interaktsioon on tehniliselt sama lihtne, kuid nii tõlgendada kui ka visuaalselt kujutada on seda keerulisem. Kahe pideva tunnuse interaktsiooni puhul on üldjuhul soovitav ka tunnused enne analüüsi tsentreerida või standardiseerida – seda saab teha funktsiooniga scale.

Üldistatud lineaarsed mudelid sotsiaalteadustes
Praktikum (SHPH.00.001, kevad 2010)

Lisa 2: Multinomiaalne logistiline regressioon

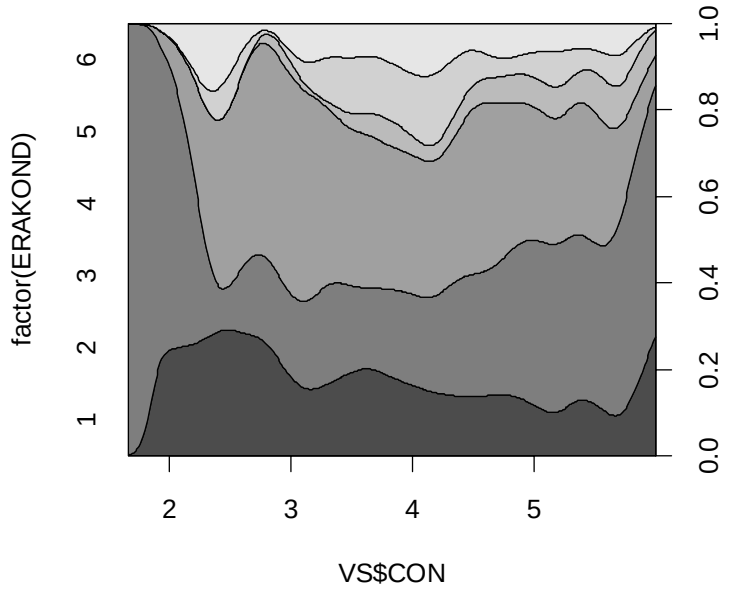
Proovime ennustada, kuidas väärtused ennustavad erakondlikke eelistusi (st seda, millise partei poolt vastaja viimastel valimistel hääletas). Erakonnaeelistus on nominaaltunnus, mida regressioonimudelis saab käsitleda dihhotoomsete tunnuste komplektina; multinomiaalses regressioonis on selline tunnuste komplekt sõltuva tunnuse rollis. Üks sõltuva tunnuse kategooriatest (R-is vaikimisi esimene kategooria) määratakse *taustakategooriaks;* edasi leitakse kõigi teiste kategooriate jaoks log-riskisuhted, mis näitavad sellesse kategooriasse kuulumise šansse võrreldes taustakategooriasse kuulumise šanssidega.

Loeme kõigepealt sisse SPSS vormingus salvestatud andmestiku kasutades funktsiooni read.ess, mis automaatselt asendab kõik puuduvate andmete koodid NA-ga. Seejärel arvutame ESS-i kodulehelt pärit õpetuste järgi väärtuste skoorid, et ei peaks mõtlema liiga suure arvu prediktorite peale.

```
library(foreign)
source("read.ess.R")
source("valuescores.R")
ESS2 <- read.ess("ESS2008_EstFin.sav")
attributes(ESS2)$x$label.table$prtvtbee
table(ESS2$prtvtbee)
names(ESS2)[508:528]
VS <- valuescores(ESS2[,508:528])
is(VS)   # matrix
# teisendame selle andmetabeliks
VS <- as.data.frame(VS)
names(VS)
```

Kasutame eelnevast konservatiivsuse skoori, VS$CON, mis võiks seostuda poliitilise eelistusega. Järgnevalt uurime erakonnaeelistuse tunnust.

```
> ERAKOND <- ESS2$prtvtbee
> table(ERAKOND)
      ERAKOND
        1   2   3   4   5   6   7   8   9  10  11
      133 237 294  46  81  69   6   1   2   1   2
> ERAKOND[!ERAKOND %in% 1:6] <- NA
> erakonnad <- rev(names(attributes(ESS2)$x$label.table$prtvtbee))[1:6]
> erakonnad
      [1] "Pro Patria and Res Publica Union" "The Estonian Centre Party"
      [3] "Estonian Reform Party"            "The People's Union of Estonia"
      [5] "The Social Democratic Party"      "Estonian Greens"
# asendame need eestikeelsete ja lühemate nimedega
> erakonnad <- c("Isamaa", "Kesk", "Reform", "Rahvaliit", "SD", "Rohelised")
# joonistame 3. praktikumist tuttava tingtihedusdiagrammi
> cdplot(factor(ERAKOND)~VS$CON)
```



Multinomiaalse regressiooniga saab hakkama funktsioon multinom lisamoodulist nnet. Multinom käitub teiste mudeldamisfunktsioonidega võrreldes natuke ebatavaliselt, näiteks ei arvuta ta olulisuse tõenäosusi ning Waldi statistik (koefitsiendi ja tema standardvea suhe, mida kasutatakse olulisuse testimisel) tuleb eraldi tellida.

```
# paneme kõigepealt faktori ERAKOND tasemetele nimed
> ERAKOND <- factor(ERAKOND, labels=erakonnad)
> mud1 <- multinom(ERAKOND~VS$CON)
> summary(mud1, Wald=TRUE)

Call:
multinom(formula = ERAKOND ~ VS$CON)

Coefficients:
          (Intercept)     VS$CON
Kesk       -1.9258858 0.58128770
Reform      0.3839627 0.09904489
Rahvaliit  -4.2672723 0.72852659
SD         -0.6189203 0.03223800
Rohelised  -1.1376514 0.11771956

Std. Errors:
          (Intercept)    VS$CON
Kesk        0.6873141 0.1599466
Reform      0.6252662 0.1486685
Rahvaliit   1.1764378 0.2642939
SD          0.8384814 0.1996193
Rohelised   0.8941745 0.2114687
```

```
Value/SE (Wald statistics):
          (Intercept)    VS$CON
Kesk       -2.8020461 3.6342605
Reform      0.6140787 0.6662132
Rahvaliit  -3.6272825 2.7565012
SD         -0.7381444 0.1614974
Rohelised  -1.2722923 0.5566761

Residual Deviance: 2630.18
AIC: 2650.18
```

Vaatame kõigepealt Waldi statistikuid. Kuna see peaks olema enam-vähem t-jaotusega, siis võime arvestada, et väärtused umbes üle 2 võiksid olla "olulised". Näeme, et konservatiivsus paneb inimesi hääletama Keskerakonna ja Rahvaliidu poolt (võrreldes taustakategooriaga, st Isamaaliiduga).

`exp(coef(mud1))` annab meile šansside suhted, st kui palju suurendab konservatiivsuse 1-punktine suurenemine hääletamise tõenäosust mingi erakonna poolt (võrreldes taust-erakonnaga).

Ülesanne. Proovige, kas teiste väärtusedimensioonide lisamine annab mudelile midagi juurde. (Andmetabelist VS: CON- konservatiivsus, OTC- "openness to change", SEN – "self-enhancement", STR – "self-transcendence"). Proovige, kas soomlaste erakondlikke eelistusi ennustavad samasugused väärtustedimensioonid kui eestlastel.

Euroopa Liit
Euroopa Sotsiaalfond
Eesti tuleviku heaks

Üldistatud lineaarsed mudelid sotsiaalteadustes
Praktikum (SHPH.00.001, kevad 2010)

Lisa 3: R-i kruvid ja mutrid

Peale selle, et R on statistikaprogramm, on ta ka küllalt suurte võimalustega programmeerimiskeel, mille hingeelu tundmine võib teha keerulisi ja aeganõudvaid ülesandeid lihtsmaks ja kiiremini lahendatavaks.

Avaldised ja nende väärtustamine

Kõik, mida me R-ilt saame küsida, kannab nime "**avaldis**" (*expression*), ning vastuseks saame avaldise "**väärtuse**" (*value*). Näiteks `1+1` on avaldis, mille väärtus on 2; kuid ka `2` on avaldis, mille väärtuseks on 2. Avaldist ja selle väärtust saab kasutada samaväärsetena, st igal pool, kus saab öelda 1+1, kõlbab ka öelda 2. (Arvestades tehete järjekorda, peaksime võib-olla 1+1 ümbritsema sulgudega.)

Kõik, mida me R-i käsureale toksime, ei pruugi olla korrektsed avaldised; see võib olla ka vigane või poolik avaldis. Esimesel juhul saame tulemuseks veateate:

```
> ///2
Error: unexpected '/' in "/"
> ))
Error: unexpected ')' in ")"
>
```

Vigase avaldise sagedaseks põhjuseks on üleliigsed või puuduvad sulud. Pooliku avaldise puhul näeme rea alguses tavapärase > asemel pluss-märki (+): saame avaldist jätkata järgmisel real. See tähendab ühtlasi, et enamikul juhtudel pole vahet, kas kirjutame avaldise ühele või mitmele reale; peaaegu igal pool, kus võib esineda tühik , võib selle asendada ka reavahetusega.

```
> 1+
+ 2+
+ 3
[1] 6
> mean(
+ 1:6
+ )
[1] 3.5
```

Avaldisi saab grupeerida {loogeliste sulgudega}; liitavaldis võib koosneda ühest või mitmest avaldisest ning selle väärtuseks on viimase "alam-avaldise" väärtus.

```
> {
+ 1+1
+ 2+2
```

```
+ }
[1] 4
```

Ühele reale saame kirjutada ka semikooloniga eraldatult mitu avaldist; sel juhul leitakse nende väärtused korraga

```
> 1; 1+1; 2+2
[1] 1
[1] 2
[1] 4
```

Avaldise väärtust näidatakse tavaliselt ekraanil; selle "meelde jätmiseks" saab kasutada omistamist (*assignment*); omistamise operaatoriks on nool (`<-` või `->`):

```
> a <- 1+1  # muutuja a saab väärtuseks 1+1 (st 2)
> 1+1 -> a  # sama
```

Funktsioonid

Funktsioon on reeglite kogum, mis teisendab oma "argumendid" funktsiooni "väärtuseks": näiteks mean on funktsioon, mille argumendiks on vektor (hulk arve) ja väärtuseks nende arvude aritmeetiline keskmine. R-i puhul peetakse heaks tooniks nn funktsionaalset programmeerimisstiili, mille kõige olulisemaks tunnuseks on, et (a) funktsiooni väärtus sõltub ainult tema argumentidest ja ei millestki muust, ja (b) funktsioon ei tee midagi muud peale oma väärtuse leidmise (näiteks ta ei muuda oma argumentide väärtusi, ei tekita töölauale uusi muutujaid jne – kõike seda nimetatakse "kõrvalmõjudeks"). Erinevalt "puhastest" funktsionaalsetest programmeerimiskeeltest on R-is võimalik neid eeldusi ka rikkuda, kuid seda peetakse üldjuhul halvaks tooniks. Mõned erandid siiski on: näiteks omistamise tulemuseks on uue muutuja tekkimine töölauale ning selle kõrvalmõju pärast omistamisfunktsiooni kasutataksegi; samuti graafikafunktsioonide kõrvalmõjuks on sageli joonis ise, ning neil ei pruugigi olla mingit mõistlikku väärtust.

Funktsiooni poole pöördumisel kirjutatakse kõigepealt funktsiooni nimi, seejärel sulgudes argumendid. Enamikul funktsioonidel on mitu argumenti; neid saab eristada nii järjekorra kui ka nimede alusel. Näiteks funktsioonil mean on kolm argumenti (vt ?mean): x, trim ja na.rm; esimene neist on vektor (arvud, mille keskmist tahetakse leida); trim näitab, kui suur osa suurematest ja väiksematest väärtustest arvutusest välja jätta (vaikimisi 0%); na.rm ütleb, mida teha puuduvate andmetega. Järgnevad avaldised on kõik samaväärsed:

```
> mean(x=1:100, trim=0.9, na.rm=TRUE)
> mean(1:100, 0.9, TRUE)
> mean(na.rm=TRUE, trim=0.9, x=1:100)
> mean(trim=0.9, x=1:100, TRUE)
> mean(1:100, trim=0.9, na.rm=TRUE)
```

Kõige tavalisem on kasutada esimest (või paari esimest) argumenti nimedeta ning ülejäänuid nimedega. See on mugav, sest tavaliselt on esimene (või paar esimest) argumenti kohustuslikud, need tuleb määratleda igal funktsiooni poole pöördumisel; samas kui ülejäänud argumendid on sageli "vabatahtlikud", st neil on vaikimisi mingid mõistlikud väärtused, mida ei ole alati vaja muuta (nt trim väärtuseks vaikimisi on 0, na.rm väärtuseks FALSE).

Tasub tähele panna, et ka omistamine, liitmine, lahutamine jne on R-is funktsioonid, kuid nende poole pöördumisel saab kasutada mugavamat sulgudeta viisi. Kui tahame neid siiski sulgudega, tavalise funktsiooni kombel kasutada, siis tuleks nende nimed esitada jutumärkides ("normaalne" nimi peab algama tähega ning võib sisaldada peale tähtede ja numbrite ainult allkriipse, punkte ja võib-olla veel mingeid märke).

```
> "+"(123223, 323423)
[1] 446646
> 123223 + 323423
[1] 446646
> "<-"(Fernando,0)
> Fernando
[1] 0
```

Funktsioon on välismaailma eest kaitstud "klausuuriga" (*closure*): muutujad, mis on funktsioonile edastatud argumentidena või defineeritud funktsiooni sees, on nähtavad ainult seal, kuid mitte väljaspool. Ja teiselt poolt, üldjuhul ei saa väljaspool defineeritud muutujad funktsiooni sees toimuvat mõjutada.

Funktsiooni definitsioon koosneb märksõnast `function`, sellele järgnevast parameetrite loetelust (sulgudes), ning sellele järgnevast avaldisest (või loogeliste sulgudega ühendatud avaldiste grupist). Proovime näiteks defineerida funktsiooni, mis suurendaks oma argumendi väärtust 1 võrra.

```
> lisa1 <- function(x) x+1
> lisa1(100)
[1] 101
> x <- 32
> lisa1(100)
[1] 101
> x
[1] 32
```

See tähendab, et funktsiooni argumendi nimeks võib olla x, ja see ei mõjuta kuidagi töölaual olevat muutujat x. Et selles päris kindlad olla, kirjutame funktsiooni sisse omistamise, mis muudab muutuja x väärtust:

```
> lisa1 <- function(x){
+ x <- x + 1
+ x
+ }
> lisa1(100)
[1] 101
> lisa1(x)
[1] 33
> x
[1] 32
```

St kui me kasutame töölaual olevat muutujat x funktsiooni argumendina, siis see ei muuda kuidagi tema väärtust. Ainuke mõistlik võimalus muutuja väärtuse muutmiseks on omistamistehe:

```
> x <- liida1(x)
> sqrt(x)
> x  # väärtus ei ole muutunud! Ka fn sqrt ei teinud seda …
```

Kui defineerime funktsiooni töölaual, siis ta saab siiski kasutada teisi töölaual leiduvaid muutujaid, samuti teisi R-is defineeritud muutujaid (sealhulgas funktsioone, näites "+", "mean" jne). See on iseenesest mõistlik, sest ei ole päris hästi ette kujutatav, et näiteks liitmistehe tuleks iga funktsiooni sees uuesti defineerida, kuid võib mõnikord tekitada probleeme. Oletame näiteks, et teeme argumentide loetelus näpuvea:

```
> X <- 3
> liida1 <- function(x) X + 1
> liida1(3)
[1] 4
> liida1(4)
[1] 4
> liida1(5)
[1] 4
> liida1(kukeleegu)
[1] 4
```

Mis nüüd juhtus? Edastasime funktsioonile argumendina 3, 4, 5; selle argumendi nimi oli x, kuid funktsiooni sees kasutasime hoopis muutujat X. Kuna funktsioonil ei ole sellise nimega argumenti ja muutujat X ei ole funktsiooni sees ka defineeritud, siis otsitakse sellise nimega muutujat töölaualt … ja leitaksegi. Aga mis juhtub viimase avaldise puhul? Muutujat kukeleegu meil töölaual ei ole, ning iga enesest lugupidav funktsioon annab sellise argumendi puhul veateate:

```
> mean(kukeleegu)
Error in mean(kukeleegu) : object 'kukeleegu' not found
> kukeleegu + 1
Error: object 'kukeleegu' not found
```

Asi on aga nii, et viimati defineeritud funktsioonis liida1 me argumenti x üldse ei kasutagi. R ei pea sel juhul tarvilikuks selle argumendi väärtust üldse leidagi: argumendi väärtus leitakse alles siis, kui seda tarvis läheb. Selle põhimõtte nimi on "laisk väärtustamine" (*lazy evaluation*). Nii võime argumendi x kohale kirjutada mitmesuguseid totrusi ilma, et R sellest väljagi teeks:

```
> lisa1(sqrt(-1))
> lisa1(kukeleegu – kikerikii ^2)
```

Tõsisem lugu on aga selles, et kui me kustutame töölaualt muutuja X, siis lakkab ka funktsioon lisa1 töötamast. Selliste "globaalsete muutujate" avastamiseks võime kasutada funktsiooni findGlobals lisamoodulist codetools:

```
> library(codetools)
> findGlobals(liida1)
[1] "+" "X"
```

Näeme, et funktsioon liida1 kasutab kahte globaalset muutujat, "+" ja "X"; neist esimene on paratamatult vajalik, kuid teine peaks olema ootamatu ja viitab näpuveale. Töölaual leiduvate muutujate kasutamist funktsioonides on mõistlik vältida, kuid muidugi ei saa vältida teiste funktsioonide kasutamist, mis tehniliselt on ka globaalsed muutujad.

Kontrollstruktuurid (tingimused ja kordused)

Paljusid keerulisi tegevusjuhiseid saab kirjeldada tingimuste ja korduste abil. Näiteks mingist arvujadast suurima arvu leidmist võiks kirjeldada nii: kontrolli, kas esimene arv on suurem teisest; kui jah, jäta meelde esimene, vastasel korral teine; nüüd kontrolli, kas meelde jäetud arv on suurem jada kolmandast arvust jne. R-is on arvulise *vektori* kõige suurema väärtusega elemendi leidmiseks funktsioon max, kuid kui seda ei oleks, siis saaksime tingimus- ja korduslauseid kasutades sellise funktsiooni ise kirjutada.

Tingimusi saab R-is väljendada mitmel moel, kõige levinumad on if-lause ja funktsioon ifelse. If-lause struktuur on järgmine:

```
if(tingimus) avaldis1 else avaldis2
# või siis ...
if(tingimus) {
          avaldis1 # täidetakse kui tingimus on tõene
          } else {
          avaldis2 # täidetakse kui tingimus ei ole tõene
          }
```

Tingimus peab olema avaldis, mille väärtuseks on kas TRUE või FALSE, vastasel korral püüab R avaldise väärtust tõeväärtuseks teisendada, ja kui see ei õnnestu, on tulemuseks veateade.

```
> a <- 1 ; b <- 2
> a>b
> if(a>b) print("a on suurem") else print("b on suurem")
```

Ifelse on eelmise *vektoriseeritud* versioon, -- see tähendab, et tingimuseks ei ole mitte üksik tõeväärtus, vaid tõeväärtuste vektor.

```
> a <- 1:10
> a > 5
> ifelse(a>5, "suurem kui viis", "väiksem kui viis")
```

Ifelse võib kasulik olla näiteks jooniste tegemisel: oletame, et tahame kujutada rahulolu ja sissetuleku seost, kuid nii, et nooremaid inimesi tähistaks punased ja vanemaid sinised punktid.

```
> punktivärv <- ifelse(vanus>=36, "blue", "red")
> plot(rahulolu~sissetulek, col = punktivärv)
```

For-lausega oleme juba kokku puutunud. See omistab etteantud muutujale järjest väärtusi etteantud vektorist ning leiab siis avaldise (või avaldiste grupi) väärtuse, kasutades iga sellist väärtust. For-tsüklist saab "välja murda" märksõnaga break; märksõna next lõpetab korralduste täitmise praeguse väärtusega ning suunab täitmisjärjekorra taas tsükli algusse.

```
for(muutuja in vektor) avaldis
for(muutuja in vektor) {
        avaldis1
        avaldis2
        if(tingimus) break
        if(tingimus) next
        .....
        avaldis3
        }
```

Oletame näiteks, et meil on kataloog "data", kus on 50 csv faili, igaühes on andmetabel erineva maa kohta, kus muuhulgas on tunnused sissetulek ja rahulolu. For-käsuga saaksime joonistada sissetuleku ja rahulolu seose kohta joonise iga maa jaoks umbes nii:

```
failinimed <- dir("data")   # failinimede vektor
setwd("data") # et ei peaks failinime ette lisama kataloogi nime
pdf("kõigimaadejoonised.pdf")   # suuname väljundi faili
for(Fail in failinimed) {
      ajutine <- read.csv(Fail)
      if(!"sissetulek" %in% names(ajutine)) next;
      # kui andmetabelis pole sellist muutujat, siis joonist ei tehta
      plot(sissetulek~rahulolu, data=ajutine, main=Fail)
      }
dev.off()
setwd("..")  # taastame endise töökataloogi
```

Mõnikord võib vaja minna ka kordusstruktuure while ja repeat: esimene jätkab kordamist, kuni mingi avaldise väärtus on TRUE, teine nii kaua, kui teda katkestatakse käsuga break.

Anonüümsed ja kõrgemat järku funktsioonid

R käsitleb funktsioone tavaliste andmetena – see tähendab muuhulgas, et saame konstrueerida funktsiooni ja seda kohapeal kasutada; samuti, et funktsiooni väärtuseks või ka argumendiks võib olla teine funktsioon. Kõrgemat järku funktsiooniks nimetatakse sellist funktsiooni, mille eesmärgiks on mingi etteantud funktsiooni rakendamine mingil kindlal moel: näiteks selle rakendamine kõigile vektori elementidele või andmetabeli tulpadele. Sagedamini kasutatavad kõrgemat järku funktsioonid R-is on sapply (mingi funktsiooni rakendamiseks kõigile listi elementidele või andmetabeli tulpadele), lapply (sama, kuid väljund on alati list; samas kui sapply püüab väljundi võimaluse korral lihtsustada maatriksiks või vektoriks); apply (funktsiooni rakendamine maatriksi ridadele või veergudele); tapply (funktsiooni rakendamine vektorile gruppide kaupa).

```
df <- data.frame(a=1:10, b=5:14, c=1:10*5, d = c(rep(1,5),
rep(2,5)))
df
sapply(df, sd)
```

```
sapply(df, mean)
apply(df, 1, mean) # võimalik ainult kui df sisaldab ainult
    # arvulisi elemente
    # leiab ridade keskmised
apply(df, 2, mean)
tapply(df$a, df$d, mean)
tapply(df$a, df$d, mean)
```

Sapply või lapply abil saame ka asendada for-käsku, näiteks kõigi mingis kataloogis olevate andmefailide sisselugemine ja jooniste tegemine võiks käia nii:

```
failinimed <- dir("data")   # failinimede vektor
setwd("data") # et ei peaks failinime ette lisama kataloogi nime
pdf("kõigimaadejoonised.pdf")   # suuname väljundi faili
andmed <- sapply(failinimed, read.csv)
sapply(andmed, function(andmed) plot(sissetulek~rahulolu,
data=andmed)) # siin ei ole vaja omistamist, sest plot-funktsioon
# ei tagasta mingit väärtust, millest meil kasu oleks
dev.off()
setwd("..")
```

Jooniste tegemiseks kasutasime anaonüümset funktsiooni `function(andmed) plot(sissetulek~rahulolu, data=andmed)).` Anonüümsete funktsioonide peamine kasutuskoht ongi kõrgemat järku funktsioonide poole pöördumisel:

```
sapply(df, function(x) mean(x)/sd(x))
sapply(df, function(x) mean(x, na.rm=TRUE, trim=0.9))
sapply(df, function(x) c(mean(x), sd(x)))
```

Lõpuks, on võimalik ka anonüümse funktsiooni kasutamine "otse":

```
> (function(x) x+1) (32)
```

Tulemuseks on ettearvatult 33. Kuigi enamikul juhtudel oleks seda vastust lihtsam saada avaldisega 32+1, on ka sellisest nipist mõnikord kasu.