Local, world-class services for the pharmaceutical industry
data management, data warehousing, statistics, information technology and scientific writing
[Beyond Your Data

Data analysis with R

Lecture 11

Additional topics about R

Jouni Junnila

Lecture contents

- During the last lecture of the course we'll focus on issues about R.
- We'll go through some functions, proven to be useful in diverse data analysis situations, starting from data manipulations and ending in functions related to analysing the data.
- We'll also learn how to write our own functions.

String functions

- Let’s start by looking into functions related to character variables.
- In some situations we need to recognize strings from a character vector.
- There are a few functions we can use in these situations.

substring

- A function called *substring* is for selecting a string of certain length from a longer character vector.
- The function needs the starting and stopping point to be determined.
  - Examples

```
> substr("abcdef",2,4)
[1] "bcd"
> substring("abcdef",1:6,1:6)
[1] "a" "b" "c" "d" "e" "f"
```

strsplit

- *Strsplit-function* splits the elements of a character vector x into substrings according to some splitting character.
- With this function we can also separate data points, unlike with *substr* and it also more efficient.
- Let’s see two examples about use of *strsplit* in R

grep

- A function called *grep* is used for pattern matching.
- With this function we can find a certain pattern inside character vectors and determine where in the character vector was the pattern found.

```
  if(length(i <- grep("foo",txt)))
 cat("'foo' appears at least once
   in\n\t",txt,"\n")
 'foo' appears at least once in
            arm foot lefroo bafoobar
```

sub & gsub

- *sub* and *gsub* perform replacement to character vectors.
- i.e it recognizes a certain pattern in the character vector and replaces it with another pattern.
- *gsub* is a global function, *sub* is not.
- Let’s have examples of *sub* and *gsub* and see the difference between the two.

Avoiding errors

- Sometimes, when we have big datasets and we need to perform a certain calculation several times in a row, (eg. a model fit) we want to control, what happens if an error occurs (eg. model does not converge).
- *try* is a wrapper to run an expression that might fail and allow the user's code to handle error-recovery.
  - So we can tell R to move forward even though an error has occured.
  - *Try* can be used with any function.

unique

- In several occasions in data analysis we need to only select the unique values of certain variable.
  - For example select only one observations per patient.
- *Unique-function* deletes duplicate elements/rows from the dataset.

```
x <- c(3:5, 11:8, 8 + 0:5)
(ux <- unique(x))
(u2 <- unique(x, fromLast = TRUE))
```

setdiff

- With *setdiff* we can find the values from object 1 that aren't in the object 2.
- Similar functions can be found for union of two objects, i.e find values that are in either object.
- And intersect of two objects, i.e find values that are in both objects.
- Let's look into these three with an example.

Memory issues

- When working with big datasets sometimes memory issues are a problem. With function *memory.limit* you can adjust the maximum memory used.
  - Note, in Windows there is a maximum usage of memory, in Linux no such a maximum exists.

Contrasts

- With statistical models quite often we have to create our own contrasts. For example comparing end-of-study values to baseline separately in the treatment group.
- Good package for this is called *contrast* (the function is also called *contrast*).
- Also in package *gmodels* there are functions (*estimable*, *fit.contrast*) to fit custom contrasts.
- If you need to do stuff like this, going through these functions would be a good way to start.

Writing your own functions

- In R it's fairly easy to write your own functions
- Using your own functions, usually helps your code to be more efficient, a lot shorter (so less time consuming ☺) and less errors.
  - Example function calculates mean and sd

```
mean.and.sd <- function(x) {
av <- mean(x)
sd <- sqrt(var(x))
c(Mean=av,SD=sd)
}
```

Own functions

- Note that variables av and sd in the previous example are local -> they cannot be assessed outside the function.
- Advanced R-programmers basically have a large set of general functions, that they have written and are validated to work correctly.
  - Then it is easy just to select the ones you need.

Outputting

- If we want to write out a data-frame, a useful function is called *write.table*.
  - *write.table(elastic1, file="bands.txt")*
  - Row and column names are printed out in the file by default. With *row.names=F /col.names=F* we can change this.
  - We can also select what will be used as a separator etc.

Redirection of screen output to a file

- The function *sink()* takes as a argument the name of a file. Screen output is then directed to that file.
- To direct output back again to the screen, call *sink()* without specifying an argument.
  - For example

```
sink("bands2.txt")
elastic1
sink()
```

Ordered factors

- We have talked a lot about factors during the course.
- As they are important, let's introduce also ordered factors.
- With ordered factors, we can have other order of factor-levels than alphaphetical.

```
stress.level<-rep(c("low","medium","high"),2)
ord.str<-
  ordered(stress.level,levels=c("low","medium","high")); ord.str
[1] low    medium high   low    medium high
Levels: low < medium < high
```

MERRY CHRISTMAS