Statistical Computing
Master Data Science
Winter Semester 2019/20

BEUTH HOCHSCHULE FÜR TECHNIK BERLIN
University of Applied Sciences

Prof. Tim Downie

Workshop 8 — Multiple regression, functions in R

In today's workshop you will learn how to fit a multiple regression model, and choose which variables should be included in that model, a process called model development. In the second part you will learn about writing user defined functions in R, which follows on from the programming theme in workshop 7, control structures.

There is no lecture today. The lecture material on multiple regression is in Lecture 7.

1 Multiple regression

Exercise 1 the Marketing data set
In Moodle there is a data set called `marketing`, originally obtained in the `datarium` package.[1] This package set has a very brief synopsis for these data.
A data frame containing the impact of three advertising media (youtube, facebook and newspaper) on sales. Data are the advertising budget in thousands of dollars along with the sales. The advertising experiment has been repeated 200 times.

Download the data from Moodle, and load it using

```
> marketing<-readRDS(file = "pathname/marketing.rds")
```

Run a few basic commands to get a feel for the data

```
> dim(marketing)
> head(marketing)
> pairs(marketing)
```

Fit a linear regression model with outcome variable `sales` with one predictor variable `youtube`, and look at the model summary.

```
> lm.obj1<-lm(sales~youtube,data=marketing)
> summary(lm.obj1)
```

It seems that YouTube advertising has a significant influence on sales.

[1] https://cran.r-project.org/web/packages/datarium/datarium.pdf

Now add the variable `newspaper`

```
> lm.obj2<-lm(sales~youtube+???,data=marketing)
> summary(lm.obj2)
```

What do you conclude about the variable `newspaper`?

Add the variable `facebook` to the regression model.

```
> lm.obj3<-lm(sales~youtube+???+???,data=marketing)
> summary(lm.obj3)
```

What do you conclude about the variables `facebook` and `newspaper`?

This issue is called collinearity. It arises because `facebook` and `newspaper` have a non-negligible correlation.

```
> cor(marketing$newspaper,marketing$facebook)
```

If `newspaper` is in the regression model without `facebook` then some of the `facebook` effect will leak into the model through `newspaper` because the two are correlated. In this case the choice is clear. If the correlation between two predictor variables is very strong then deciding which variable to keep in the model can be a difficult choice.

Model checking

Fit a regression model with just `youtube` and `facebook`, and look at the residual plot.

```
> lm.obj4<-lm(sales~???+???,data=marketing)
> summary(lm.obj4)
> plot(lm.obj4,which=1)
```

The residual plot is not very good. There seems to be a U-shape to the residuals. Try transforming the `sales` by taking the logarithm.

```
> lm.obj5<-lm(log(sales)~youtube+facebook,data=marketing)
> plot(lm.obj5,which=1)
```

This might be better but now there are two extreme outliers, in rows 131 and 156. A rough and ready method when there are *extreme outliers* is to delete thes values from the regression, which can be done using the subset command.

```
> lm.obj6<-lm(log(sales)~youtube+facebook,data=marketing,subset=c(-131,-156))
> plot(lm.obj6,which=1)
> summary(lm.obj6)
```

This model is not perfect but is much better than the plot for model 4.

As an example we can obtain a predicted value for the approximate median values for youtube and `facebook` advertising using model 6.

```
> newdata <- data.frame(
+   youtube = 180, facebook = 27)
# Predict sales values
> predict(lm.obj6,newdata)
> exp(predict(lm.obj6,newdata))
```

Why is it necessary to use the `exp()` function?

Lecturer's comment: fitting a statistical model to real data is difficult, and is a skill which develops with practical experience. This model with more investigation could certainly be improved, but in the context of this Statistical Computing course, this last regression model is sufficient.

Exercise 2 The Auto data set

Now you will develop a multiple regression model using an easier data example.

Install and load the ISLR package. This package contains data sets that accompany the book *An Introduction to Statistical Learning with applications in R*[2], which you will be use in Machine Learning courses.

```
> install.packages("ISLR")
> require(ISLR)
```

You will fit a multiple regression model to fuel consumption for cars in the `Auto` data set. Read the help file and get a feel for the data:

```
> ?Auto
> dim(Auto)
> head(Auto)
> hist(Auto$mpg)
```

The variable `mpg` is the fuel consumption in "miles per US gallon", and large values are good, but correspond to low fuel consumption. In Europe the standard scale is "litres per 100 km", where a large value corresponds to high fuel consumption. We can add a new variable called `fuel` to the data frame using

```
> Auto$fuel<-235/Auto$mpg
```

Use what you have learnt in the previous exercise to do the following

(a) Fit a linear regression model of fuel consumption dependent on `weight`

(b) Add the variables `horsepower`, `displacement`, and `year` to the model one by one, obtaining the model estimates after each step.

[2]James, G., Witten, D., Hastie, T., and Tibshirani, R. (2013) www.StatLearning.com, Springer-Verlag, New York

(c) Which variables should be in your final model?

(d) Obtain a residual plot for your final model. Are there any obvious problems with this plot?

(e) Obtain a prediction for fuel consumption for `weight`=2500, `horsepower`=120, `displacement`=150 and `year`=75

2 Functions

In Workshop 7, you learn about *control structures*. This week you will learn about writing user defined functions in R. Writing functions is important in R and makes it a very flexible programming and data analysis environment.

Functions enable you to:

- do similar things repeatedly without having to type them in each time,
- do multiple things by typing in just one command,
- make it easier to understand your code by grouping together a set of commands and referring to them by a sensible name,
- customise existing R functions,
- develop detailed programs and algorithms.

Most of the R commands you have used so far are actually functions, which have been supplied with R. You can also define your own, and use them in a similar way.

2.1 Single line functions

Example: we wish to write a simple function that computes logarithms with basis a, $\log_a(x)$, given values for x and a.

```
> log.base.a<-function(x,a=10) log(x)/log(a)
```

This creates a function called `log.base.a`. It takes two arguments `x` and `a`. If no value for `a` is given then its default value will be 10 (this is specified by the argument specification `a=10`). The value $\log(x)/\log(a)$ is computed and returns the appropriate logarithm given an x (and optionally an a)

```
> log.base.a(3.5,5)
[1] 0.7783854
```

We can check the answer because the R function `log()` takes an argument called `base`!

```
> log(3.5,base=5)
[1] 0.7783854
```

To see that the default value for a in `log.base.a` is 10, try:

```
> log.base.a(3.5)
[1] 0.544068
> log.base.a(3.5,a=10)
[1] 0.544068
```

2.2 Multiline functions

Usually functions are longer than one command, and then they must be enclosed in braces (curly brackets).

Example: below is a function that returns `TRUE` if a number is a factorial number and `FALSE` otherwise

```
is.factorial<-function(n)
{
  i<-0
  prod.sofar<-1
  while(prod.sofar<n){
    i<-i+1
    prod.sofar<-prod.sofar*i
  }

  prod.sofar==n
}
```

Notes:
This is a function called `is.factorial`, which takes one argument called `n`. Inside the brackets there is a `while` loop that is very similar to the while loop in Workshop 7, except now the loop stops once the current factorial number `prod.sofar` is greater than or equal to `n`.

The variables within the function are *local* — `prod.sofar` will not be in the object list when you type `objects()`, or in the global environment window. This is useful because perhaps you already have an object called `prod.sofar` in your global environment. Calling `is.factorial` will not overwrite your value. There are thousands of functions in R. It would be very annoying if calling an R function trashed one of your objects just because an R programmer in 2002 happened to use the same object name as you!

The last line of the function `prod.sofar==n` looks odd. It is a logical variable which is `TRUE` if `n` equals `prod.sofar` and `FALSE` otherwise. Thus if `n` (the value supplied by the user) is a factorial number, this line is `TRUE` otherwise it is `FALSE`. **Because this is the last statement in the function**, the value that is returned is the value of `prod.sofar==n`, either `TRUE` or `FALSE`.

So typing

```
> is.factorial(6)
[1] TRUE
```

prints the return value to the screen and

```
> a<-is.factorial(12)
```

assigns the return value (`FALSE`) to object `a`.

Functions can call other functions, both the examples above use R supplied functions.

Comment: Some students find the general concept of writing functions confusing at first, and don't appreciate the difference between a sequence of commands in a script file and bundling these those commands into a function. A useful way to think about writing functions is to pretend that you have been asked to write the function by a client. The client does not want to know what commands you use inside the function any more than you need to know which commands are used in the function to load a CSV file `read.csv()`. All the client needs to know is the name of the function its arguments.

2.3 Some useful points and hints about functions

2.3.1 Breaking mid-function

The return value is normally the last line in the function. If you want to return a value mid-function, you can use `return(x)` to return the object `x`. This is typically done using an `if` statement. The following function returns a *complex* number if the user asks for the square root of a negative number.

```
general.sqrt<-function(x)
{
# this function returns the square root of
# any real numeric value x, whether positive or negative.

# and by the way you can and should comment your functions
# by using the hash sign.

   if(x>=0) return(sqrt(x))

# If we get here then x is negative

# the last line in the function returns a complex number
   complex(real=0,imaginary=sqrt(-x))
}
> general.sqrt(-1)
[1] 0+1i
```

2.3.2 Returning more than one object

An R functions can only return one object, but sometimes you want to return more than one object. The way round this is to bundle all those objects into one list. For example:

```
{
.....
### main body of function
.....
   list(temp=tt,lm.res=lm.obj)
}
```

Sometimes you can return several numeric values in one vector. E.g.the function `range` returns the maximum and minimum of the vector, supplied as a vector in `c(xmin,xmax)` form.

2.3.3 Functions are objects

Typing

```
> objects()
```

lists all your objects that you have created and are stord in you "environment". Notice that `is.factorial` and `general.sqrt` are in your list of objects. If you type an object name (e.g. `x`) then the value of that object is output to the console. Typing the function name without brackets will output the text definition of the function. Every object has at least one "class", a function has the class called `function`

```
> is.factorial
.............
> class(x)
[1] "numeric"
> class(is.factorial)
[1] "function"
```

You can also look the R supplied functions, but usually what is printed is not so helpful.

2.4 Function arguments

2.4.1 Supplying default values

Sometimes we will wish a particular argument of a function to take a default value unless instructed otherwise. This can be achieved using the following general form,

```
fname <- function(arg1=default, ...) statement
```

Here the value of `arg1` will be `default`, if the user does not supply an alternative.

Example

The following function calculates the sum of the `p`th power of the elements of the vector `x`, with the default value of `p` being 2.

```
> sumpow <- function(x,p=2) {
+  sump <- sum(x^p)
+  cat("\n","Sum of elements to power", p,"is",sump,"\n")
+  invisible(sump)
+ }
> sumpow(x=c(1,-1,2))

Sum of elements to power 2 is 6
> sp.out<-sumpow(x=c(3,3),p=5)

Sum of elements to power 5 is 486
> sp.out
[1] 486
```

The `invisible()` command at the end of the function is a useful trick to know. In this example the value of `sump` does not appear in the console; there is no need, we have printed out its value in the `cat()` command. The user can however assign this value to an object if they want. `invisible()` is very useful when the output object is large, the user does not want to be bombarded with a huge amount of output in the console that is very difficult to read.

2.4.2 Local variables

As mentioned in 2.2, variables defined inside a function are local. Any assignments or changes hgave no effect outside of the function. However, when variables are not initialised in the function, tghe R will try to find an object with than name outside of the function, i.e. in your global environment or in the search path.

This has definite advantages, but forgetting to initialise local variables in a function is bad programming and can cause problems. Here's an example:

```
> badfunction <- function() { x+1 }
```

What you might expect this function to "think" is:
calculate the value of `x+1`*; hang on, you haven't said anything about* `x` *so I don't know what it is; I will terminate with an error message!*
What it *actually* "thinks" is:
calculate the value of `x+1`*; hang on, you haven't said anything about* `x` *so I don't know what it is; maybe there's an object called* `x` outside *the function that I'm supposed to be accessing.*
In this case, if there *is* an object called `x`, the function will use it without telling you. This will probably

lead to unexpected results — beware!

2.4.3 The . . . argument

A special argument is called `...` (exactly 3 points), which allows the user to specify an argument intended for a second function nested within the first function. This is very common with plot parameters. E.g.:

```
convert.temp<-function(fahrenheit,...)
{
celsius<-(fahrenheit -32)*5/9
plot(fahrenheit,celsius,...)
celcius
}
```

The `...` allows the user to specify graphics parameters such as `col`, and `main` to the `plot` function. `convert.temp()` doesn't need to know what those arguments are for it to do the temperature conversion.

```
> convert.temp(c(0,10,20,30,40,50,60,70,80,90,100),
+ main="Celcius against fahrenheit")
```

Exercise: Repeat the above but plotting both points and lines: `type="b"`, You do not need to change the function definition of `convert.temp` in order to do this.

2.5 Editing functions

Small functions can be included directly in your script file. Every time you edit the function, you need to run the function definition again. If you have one large function or use many user defined functions, it is best to put them in a separate `.R` file (ascii text). If you update the functions, use the command `source(file="???")` to parse the text in that file.

3 Additional Exercises

Check that each function works properly by calling the function with appropriate arguments.

Exercise 3 Inner product

Write a function called `innerp`. This function takes two vectors of the same length as arguments and computes $\sum_{i=1}^{n} x_i \cdot y_i$, where n is the length of the vectors. This function should:

1. check that `x` and `y` both have the same length, returning an error message if they are not, and
2. If `y` is not specified, use `x` as the default value for `y`.

First write the function using a for loop. Then replace the for loop using the function `sum`.

Exercise 4 Is a matrix symmetric?

A Matrix M is symmetric if M is equal to its transpose. $M = M^\top$. The matrix must have the same number of rows as columns.

Write a function called `is.symmetric`, that takes a square matrix and returns `TRUE` if it symmetric and `FALSE` if it is not. If the matrix is not square then an error should be returned. Hint: make use of the R functions `dim()` and `t()`.