SCR Week 3 Selfstudy Exercises

1. Cut, factors and tables

a.

Create two vectors with 1000 random samples from standard normal distributions.

Answer:

```
N <- 1000
var1 <- rnorm(N)
var2 <- rnorm(N)
```

b.

Use `cut` to cut these two variables up, seperately, such that the observations for each of the variables are in one of 10 bins. Use the `breaks` argument and provide a single number. What type of output does `cut` create?

Answer:

```
my_cut1 <- cut(var1, breaks=10)
my_cut2 <- cut(var2, breaks=10)
```

A factor!

c.

Use `table` to at the distribution of the values over the 10 bins for both variables. Are the distributions and the intervals the same? In what way are the breaks created?

Answer:

```
table(my_cut1)
```

```
## my_cut1
##     (-3.67,-3]     (-3,-2.34]  (-2.34,-1.68]  (-1.68,-1.02] (-1.02,-0.357]
##              2              7             30            113            206
## (-0.357,0.303]  (0.303,0.964]   (0.964,1.62]    (1.62,2.28]    (2.28,2.95]
##            266            213            118             35             10
```

```
table(my_cut2)
```

```
## my_cut2
##  (-3.35,-2.71]  (-2.71,-2.08]  (-2.08,-1.45] (-1.45,-0.821] (-0.821,-0.19]
##              4             19             64            112            220
##   (-0.19,0.44]    (0.44,1.07]     (1.07,1.7]     (1.7,2.33]    (2.33,2.97]
##            238            181            112             41              9
```

It looks like all the intervals are of equal length. It looks like the bins are determined based on the actual data inside the vectors provided to `cut`.

d.

Look at the function `quantile`. Use it to create 10 new cutting points, such that the values of your random variables are uniformly distributed over the intervals.

Answer:

```
my_breaks1 <- quantile(var1, probs=seq(0, 1, length=10))
my_breaks2 <- quantile(var2, probs=seq(0, 1, length=10))

my_cut1_2 <- cut(var1, breaks=my_breaks1)
my_cut2_2 <- cut(var2, breaks=my_breaks2)
```

e.

Try to cross tabulate your original variables with random samples in them. Does it work / is it an interesting table?

Answer: Possible does not work: too many unique observations. Also not very insightful.

```
table(var1, var2)
```

f.

Now cross tabulate variables 1 and 2, but use the two different binned versions you've created. Which is more interesting? Do you notice any pattern?

Answer:

```
table(my_cut1, my_cut2)
```

```
##                 my_cut2
## my_cut1          (-3.35,-2.71] (-2.71,-2.08] (-2.08,-1.45] (-1.45,-0.821]
##   (-3.67,-3]                 0             0             0               0
##   (-3,-2.34]                 0             0             0               3
##   (-2.34,-1.68]              1             0             2               2
##   (-1.68,-1.02]              1             1             5               7
##   (-1.02,-0.357]             0             5            18              22
##   (-0.357,0.303]             1             7            18              31
##   (0.303,0.964]              1             0            11              28
##   (0.964,1.62]               0             4             5              16
##   (1.62,2.28]                0             1             3               2
##   (2.28,2.95]                0             1             2               1
##                 my_cut2
## my_cut1          (-0.821,-0.19] (-0.19,0.44] (0.44,1.07] (1.07,1.7]
##   (-3.67,-3]                 2            0           0          0
##   (-3,-2.34]                 1            2           1          0
##   (-2.34,-1.68]              8            7           4          4
##   (-1.68,-1.02]             30           35          19         12
##   (-1.02,-0.357]            41           53          40         16
##   (-0.357,0.303]            54           58          50         31
##   (0.303,0.964]             52           49          39         23
##   (0.964,1.62]              24           23          22         19
```

```
##   (1.62,2.28]                6            9           5          6
##   (2.28,2.95]                2            2           1          1
##                 my_cut2
## my_cut1          (1.7,2.33] (2.33,2.97]
##   (-3.67,-3]              0           0
##   (-3,-2.34]              0           0
##   (-2.34,-1.68]           2           0
##   (-1.68,-1.02]           2           1
##   (-1.02,-0.357]          8           3
##   (-0.357,0.303]         13           3
##   (0.303,0.964]           8           2
##   (0.964,1.62]            5           0
##   (1.62,2.28]             3           0
##   (2.28,2.95]             0           0
```

```
table(my_cut1_2, my_cut2_2)
```

```
##                  my_cut2_2
## my_cut1_2         (-3.34,-1.22] (-1.22,-0.739] (-0.739,-0.404]
##   (-3.66,-1.19]              12              8              13
##   (-1.19,-0.761]             13             12              13
##   (-0.761,-0.442]            16             12              14
##   (-0.442,-0.164]             8             15               9
##   (-0.164,0.138]             16             12              10
##   (0.138,0.42]               12             13              11
##   (0.42,0.743]               12             14              13
##   (0.743,1.15]               10             12              16
##   (1.15,2.94]                12             13              12
##                  my_cut2_2
## my_cut1_2         (-0.404,-0.133] (-0.133,0.154] (0.154,0.46] (0.46,0.847]
##   (-3.66,-1.19]                15             14           18            9
##   (-1.19,-0.761]                9             14           14           13
##   (-0.761,-0.442]               8             15           14            9
##   (-0.442,-0.164]              12              6           12           15
##   (-0.164,0.138]               15             15           11           12
##   (0.138,0.42]                 16             14            9           15
##   (0.42,0.743]                 10             13           11           16
##   (0.743,1.15]                 16              7           15            9
##   (1.15,2.94]                   9             13            7           12
##                  my_cut2_2
## my_cut1_2         (0.847,1.29] (1.29,2.96]
##   (-3.66,-1.19]             11          11
##   (-1.19,-0.761]            12          11
##   (-0.761,-0.442]           10          13
##   (-0.442,-0.164]           18          16
##   (-0.164,0.138]             9          10
##   (0.138,0.42]              10          11
##   (0.42,0.743]              12          10
##   (0.743,1.15]              14          12
##   (1.15,2.94]               16          17
```

The first is more interesting. There is a pattern in the sense that it looks like a multivariate normal distribution. There is a sort of 'round-ness' to the distribution of the numbers in the table, this is because the variables are uncorrelated.

g.

Try to think of a way to contruct two normally distributed variables, such that if we `cut` them, using equal sized intervals, and cross tabulate them, we see more observations on the diagonal of the crosstable.

Answer: We need to create (positively) correlated variables:

```
var_one <- rnorm(N)
var_two <- var_one + rnorm(N)
cor(var_one, var_two)
```

```
## [1] 0.7041303
```

```
my_cut_one <- cut(var_one, breaks=10)
my_cut_two <- cut(var_two, breaks=10)

table(my_cut_one, my_cut_two)
```

```
##                 my_cut_two
## my_cut_one       (-4.34,-3.44] (-3.44,-2.55] (-2.55,-1.67] (-1.67,-0.777]
##   (-2.95,-2.26]             2             5             6              1
##   (-2.26,-1.58]             2            12            15             12
##   (-1.58,-0.9]              0             7            39             40
##   (-0.9,-0.217]             0             5            37             49
##   (-0.217,0.465]            0             4            10             47
##   (0.465,1.15]              0             0             0             12
##   (1.15,1.83]               0             0             1              2
##   (1.83,2.51]               0             0             0              0
##   (2.51,3.19]               0             0             0              0
##   (3.19,3.88]               0             0             0              0
##                 my_cut_two
## my_cut_one       (-0.777,0.111] (0.111,0.999] (0.999,1.89] (1.89,2.77]
##   (-2.95,-2.26]               0             0            0           0
##   (-2.26,-1.58]               8             0            0           0
##   (-1.58,-0.9]               34             9            0           0
##   (-0.9,-0.217]              94            40           18           2
##   (-0.217,0.465]             70            90           32           8
##   (0.465,1.15]               31            63           46          21
##   (1.15,1.83]                10            21           31          20
##   (1.83,2.51]                 0             3            6           9
##   (2.51,3.19]                 1             1            2           2
##   (3.19,3.88]                 0             0            0           0
##                 my_cut_two
## my_cut_one       (2.77,3.66] (3.66,4.56]
##   (-2.95,-2.26]           0           0
##   (-2.26,-1.58]           0           0
##   (-1.58,-0.9]            0           0
##   (-0.9,-0.217]           0           0
##   (-0.217,0.465]          1           0
##   (0.465,1.15]            3           0
##   (1.15,1.83]             5           1
##   (1.83,2.51]             5           1
##   (2.51,3.19]             1           2
##   (3.19,3.88]             0           1
```

2. Factors and Preparing Data Frames

Denote and decode the following data frame:

```
set.seed(20141120)
dat <- data.frame(
  id = sample(gl(20, 5), replace = TRUE),
  y = rnorm(100, 10),
  time = rep(1:5, 20),
  gender = factor(
    NA,
    levels = 1:3,
    labels = c("male", "female", "other")
  ),
  age = NA,
  treat = factor(
    NA,
    levels = 1:3,
    labels = c("placebo", "active", "control")
  )
)
dat <- dat[order(dat$id), ]  # order the rows of dat by id
```

Read de above code carefully and look at the help file for functions you're not familiar with. Alos look at the examples of the help file, especially when the help file does not sufficiently clarify the code for you.

a.

Each id number represents one individual and during the whole study his/her/its gender and treatment remains the same.

Assign at random the `male` or `female` gender to each `id` number and assign randomly a `placebo` or `active` treatment to each `id` number. Keep the structure (`class` and `attributes`) of the factors intact!

```
attributes(dat$gender)
gndr.att <- attributes(dat$gender)
dat[['gender']] <- sample(c("male", "female"), 100, replace = TRUE)[dat$id] # becomes character!
attributes(dat$gender)
attributes(dat$gender) <- gndr.att
```

Hint: the above code does not work for two reasons: 1. we lose the factor structure, 2. each id number can have a different gender over time

Answer:

```
gndr.att <- attributes(dat$gender)
dat$gender <- sample(1:2,20, replace = TRUE)[dat$id]
attributes(dat$gender) <- gndr.att

trt.att <- attributes(dat$treat)
dat$treat <- sample(1:2,20, replace = TRUE)[dat$id]
attributes(dat$treat) <- trt.att
```

b.

Now let's give our patients an age which makes sense according to the time of measurement:

```
dat$age <- unlist(
  sapply(X = unique(dat$id), # line 4
    FUN = function(i) {
      age.id <- sort(sample(x = 18:55, size = 5, replace = TRUE)) # line 1
      out <- age.id[dat$time[dat$id == i]] # line 2
      return(out)
      } # line 3.
  )
) # line 5
```

Describe what happens at each commented line of code.

Answer:

- line 1: sample 5 values out of the sequence 18:55 with replacement and sort them ascending and store them in the object age.id
- line 2: only select from age.id those indices for which individual i has corresponding time values
- line 3: the function that will be 's'-applied and has argument i that comes from the `X` argument in `sapply`
- line 4: the argument `X` is a vector containing the unique individuals. On each element (= individual) of X, the function FUN is applied
- line 5: make a vector from the list using function `unlist`

c.

Based on `age` create a factor `cat_age` (categorical age) that takes the values

- 1, if $(18 \leq age < 35)$
- 2, if $(35 \leq age < 45)$
- 3, if $(45 \leq age < 55)$

Apart from using logical operators, code using the function `cut` is also allowed.

Answer:

```
dat$cat_age <- cut(dat$age, c(18, 35, 45, 55), label = 1:3,
                   include.lowest = TRUE, right = FALSE)
```

d.

Based on the factor you defined above create the 3-way contingency table, `tabdat1`, that contains the frequencies for each combination of `gender`, `treat` and category of `age` (as defined above). *Challenge yourself: use the function `with()` (or any other function) such that you don't have to write `dat$` three times.*

Answer:

```
tabdat1 <- with(dat, table(gender, treat, cat_age))
```

e.

Did you obtain empty columns and cells? How would you use the `drop` argument to remove those empty rows and columns from your 3-way contingency table? Please do question d again, but use the `drop` argument.

Answer:

```
tabdat2 <- with(dat, table(gender[drop = T], treat[drop = T], cat_age))
```

###f.

For this table compute the marginal sums for each of the **3 dimensions**, then show the three 3-way contigency tables in which the cells represent the percentages of on of the dimensions.

Answer:

```
sum1st <- apply(tabdat2, 1, sum)
sum2nd <- apply(tabdat2, 2, sum)
sum3rd <- apply(tabdat2, 3, sum)

tb.gnd.pc <- 100 * tabdat2 / sum1st # expressed in % for gender
tb.trt.pc <- 100 * tabdat2 / sum2nd # expressed in % for treat
tb.age.pc <- 100 * tabdat2 / sum3rd # expressed in % factorized age
```

g.

As you will observe, some subjects have more than one measurement for some of time points (i.e. column/variable `time`). Create a new data frame, called `new_dat`, in which these measurements are averaged.

NOTE + HINT: This is a difficult exercise, you might want to skip it for now. If you accept the challenge: the function ***tapply****,* ***duplicated*** *and* ***na.omit*** *may save you time in search for an answer.*

Answer:

```
input.dupl <- interaction(dat$id, dat$time)
new_dat <- dat[!duplicated(input.dupl), ]
new_dat <- new_dat[order(with(new_dat, id)), ]
new_dat$y.new <- na.omit(c(tapply(dat$y, list(dat$time, dat$id), mean)))
```

h.

Add a new column to the `new_dat` data frame that contains the average value of `y` per `id` (i.e., for each subject the average should be replicated the number of his/her measurements). *Hint: use the function* ***tapply*** *or* ***ave****.*

Answer:

```
dat$mean.y <- tapply(dat$y, dat$id, mean)[dat$id]
# or
dat$mean.y <- ave(dat$y, dat$id)
```

扩充

3. Functions and scoping: peeling the layers

a.

Read through the following code and try to figure out what these functions do. Especially, try to reason out what the output of the `cat` functions will be.

```
a <- 1
b <- "one"

FirstLayer <- function(){
  cat("Entering the first layer\n")

  cat("Inside the first layer a is:", a, "\n")
  cat("Inside the first layer b is:", b, "\n")

  a <- 2

  cat("Inside the first layer a was changed to:", a, "\n")

  SecondLayer()

  cat("Inside the first layer a is now:", a, "\n")
  cat("Inside the first layer b is now:", b, "\n")

}

SecondLayer <- function(){
  cat("Entering the second layer\n")

  cat("Inside the second layer a is:", a, "\n")
  cat("Inside the second layer b is:", b, "\n")

  a <- 3
  b <- "two"

  cat("Inside the second layer a was changed to:", a, "\n")
  cat("Inside the second layer b was changed to:", b, "\n")

  cat("Exiting the second layer\n")
  return(NULL)
}
```

b.

Run the function `FirstLayer`. Is the printed output as you would expect?

Answer:

```
FirstLayer()
```

```
## Entering the first layer
## Inside the first layer a is: 1
```

```
## Inside the first layer b is: one
## Inside the first layer a was changed to: 2
## Entering the second layer
## Inside the second layer a is: 1
## Inside the second layer b is: one
## Inside the second layer a was changed to: 3
## Inside the second layer b was changed to: two
## Exiting the second layer
## Inside the first layer a is now: 2
## Inside the first layer b is now: one
```

Why is `a` not `2` inside the second layer? This has to do with 'lexical scoping': see the outro 'What did we learn?' at the end for more details.

c.

Move the definition of the `SecondLayer` function inside of the function definition of `FirstLayer`. Run `FirstLayer` again. What do you think of the printed output this time?

Answer:

```
FirstLayer <- function(){
  SecondLayer <- function(){
    cat("Entering the second layer\n")

    cat("Inside the second layer a is:", a, "\n")
    cat("Inside the second layer b is:", b, "\n")

    a <- 3
    b <- "two"

    cat("Inside the second layer a was changed to:", a, "\n")
    cat("Inside the second layer b was changed to:", b, "\n")

    cat("Exiting the second layer\n")
    return(NULL)
  }

  cat("Entering the first layer\n")

  cat("Inside the first layer a is:", a, "\n")
  cat("Inside the first layer b is:", b, "\n")

  a <- 2

  cat("Inside the first layer a was changed to:", a, "\n")

  SecondLayer()

  cat("Inside the first layer a is now:", a, "\n")
  cat("Inside the first layer b is now:", b, "\n")

}
```

```
FirstLayer()
```

```
## Entering the first layer
## Inside the first layer a is: 1
## Inside the first layer b is: one
## Inside the first layer a was changed to: 2
## Entering the second layer
## Inside the second layer a is: 2
## Inside the second layer b is: one
## Inside the second layer a was changed to: 3
## Inside the second layer b was changed to: two
## Exiting the second layer
## Inside the first layer a is now: 2
## Inside the first layer b is now: one
```

Apparantly, by defining `SecondLayer` inside `FirstLayer`, we've changed the `lexical` scope, the environment from which to retrieve non-existing objects in the current scope to that of the executing environment of the `FirstLayer` function.

d.

Rewrite the functions `FirstLayer` and `SecondLayer` as presented in **a** so that the arguments `a` and `b` are explicitly passed along: they should be present in the list of arguments of the function, and therefore also be part of the function call. Run `FirstLayer` again. Did the function behave as expected?

Answer:

```
a <- 1
b <- "one"

FirstLayer <- function(a, b){
  cat("Entering the first layer\n")

  cat("Inside the first layer a is:", a, "\n")
  cat("Inside the first layer b is:", b, "\n")

  a <- 2

  cat("Inside the first layer a was changed to:", a, "\n")

  SecondLayer(a, b)

  cat("Inside the first layer a is now:", a, "\n")
  cat("Inside the first layer b is now:", b, "\n")

}

SecondLayer <- function(a, b){
  cat("Entering the second layer\n")

  cat("Inside the second layer a is:", a, "\n")
  cat("Inside the second layer b is:", b, "\n")
```

```
  a <- 3
  b <- "two"

  cat("Inside the second layer a was changed to:", a, "\n")
  cat("Inside the second layer b was changed to:", b, "\n")

  cat("Exiting the second layer\n")
  return(NULL)
}

FirstLayer(a, b)
```

```
## Entering the first layer
## Inside the first layer a is: 1
## Inside the first layer b is: one
## Inside the first layer a was changed to: 2
## Entering the second layer
## Inside the second layer a is: 2
## Inside the second layer b is: one
## Inside the second layer a was changed to: 3
## Inside the second layer b was changed to: two
## Exiting the second layer
## Inside the first layer a is now: 2
## Inside the first layer b is now: one
```

Outro

The best way to deal with scoping is to explicitly pass arguments and do not depend on scoping UNLESS you are very comfortable with how this works. If you want to go nuts on these concepts, go to e.g. http://adv-r.had.co.nz/Environments.html#function-envs (*warning: confusion ahead*)!

4. Swirl

If you haven't done so yet, follow lessons 10 & 11 from the R package `swirl`

`10: lapply and sapply 11: vapply and tapply`

NB. `vapply` is a fun a neat function to learn. However, in this course you will not see the lecturers using this function, nor will you be specifically used to ask to use this function during exams and assignments.