Introduction to R – week 5

Indrek Seppo

From the previous sessions

dplyr and piping

Please read the original piaac data in again

```
piaac <- read.csv("http://www.ut.ee/~iseppo/piaacENG.csv")
```

One of the coolest features of R is piping, the "and then" operator: `%>%`. This sends the result of the last command to be the first argument of the next command. So a typical workflow would be something like this:

```
newdataset <- initialdata %>%
  filter(variable>4, !is.na(othervariable))%>%
  group_by(variable1, variable2)%>%
  summarize(meanofvar=mean(variable1, na.rm=TRUE), othervalue=myfunction(variable4, variable5))
```

The main verbs of dplyr were:

1) **select()** for selecting variables, you can use helper functions like **starts_with()**, or select consequetive variables with `: variable1:othervariable`. Separate variables (or criterions) by commas.

For example, if I want to select only variables age, Education and Income, and write them into a new dataset, I would do the following:

```
smallpiaac <- piaac %>%
  select(age, Education, Income)
```

Check it (with `names(smallpiaac)` or `View(smallpiaac)`) – the new dataset only has those variables.

2) Then there was **filter()** for subsetting the dataset. You can separate different conditions by commas, for example `sex=="Male", age > 29`. A useful way to select all the values in a set is to use `%in%` operator: `education %in% c("high", "medium")`.

Lets try it – lets create a new dataset of males, whose studyarea is either "Services" or "General programmes" (remember – to check which study areas are prsent in the studyarea variable you would use `unique(piaac$studyarea)`):

```
ournewdataset <- piaac %>%
  filter(gender=="Male", studyarea %in% c("Services", "General programmes"))
```

And by piping you can put different dplyr commands together. For example, men whose studyarea is "Services", and we only need their age, Numeracy and Income:

```
ournewdataset <- piaac %>%
  filter(gender=="Male", studyarea=="Services")%>%
  select(age, Numeracy, Income)
```

Your turn:

1. Create a new dataset `smallpiaac`, which contains only the variables `seqid`, `age`, `Literacy`, `Numeracy`, `Problem.solving.skills`.
2. overwrite piaac with a dataset derived from piaac where you have removed the lines where either Education, studyarea of health is NA. Remember – you have to use `is.na()` for this (actually `!is.na()`).
3. Create a new dataset `goodhealth`, which contains only those rows from `piaac` where health is either "Excellent" or "Very good". Select only the variables `age`, `gender`, `Health` and `Income` to be in this dataset. What is the average income for these people?

It should be 1034.14.
Then there was a powerful combination of:

- **group_by()** – creates groups in the data by any number of variables
- **summarize()** – computes some summarizing statistics of the variable (by groups if **group_by()** was used previously).

As an example – if we want a dataframe of the average numeracy and average literacy of everybody from `piaac` we could achieve it like this:

```
piaac %>%
  summarize(av_numeracy = mean(Numeracy), av_literacy = mean(Literacy))
```

```
##   av_numeracy av_literacy
## 1    277.2483    279.4468
```

If we would now want it by gender, we just need to add `group_by(gender)`:

```
piaac %>%
  group_by(gender) %>%
  summarize(av_numeracy = mean(Numeracy), av_literacy = mean(Literacy))
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
## # A tibble: 2 x 3
##   gender av_numeracy av_literacy
##   <chr>        <dbl>       <dbl>
## 1 Female        273.        279.
## 2 Male          282.        279.
```

If we want to add some other grouping variables, be my guest:

```
piaac %>%
  group_by(gender, Education) %>%
  summarize(av_numeracy = mean(Numeracy), av_literacy = mean(Literacy))
```

```
## `summarise()` regrouping output by 'gender' (override with `.groups` argument)
```

```
## # A tibble: 6 x 4
## # Groups:   gender [2]
##   gender Education av_numeracy av_literacy
##   <chr>  <chr>           <dbl>       <dbl>
## 1 Female Basic            234.        239.
## 2 Female Higher           285.        289.
## 3 Female Secondary        263.        271.
## 4 Male   Basic            254.        252.
## 5 Male   Higher           299.        292.
## 6 Male   Secondary        274.        274.
```

Your turn:

1. We would now be summarizing the data: create a new dataset which would include average wage (`Income`) per education level (variable Education) and gender (so these would be the grouping variables now). Call the newly created variable `averagewage`. I think you should also use the `na.rm=T` while finding the averages.

2. Lets make it bit more difficult – add 25.th and 75.th percentiles of the wages to the previous `summarize()` call. You can find the percentiles using function `quantile()`, which needs a second argument as well: the percentile (probs = 0...1, so 25.th percentile would be 0.25). You should just add this `probs=` to the quantile call after the variable you want to find the quantile of (and you will also need a third parameter telling it to not take into account the NA values – `na.rm=T`). Call the new variables `pc25` and `pc75`. Then arrange the result by the variable `averagewage`, which you have just created.

   The result should look smth like this:

```
## # A tibble: 3 x 5
## # Groups:   Education [3]
##   Education gender averagewage  pc25  pc75
##   <chr>     <chr>        <dbl> <dbl> <dbl>
## 1 Basic     Female        506.  302.  545.
## 2 Secondary Female        553.  350   644.
## 3 Higher    Female        865.  526. 1013.
```

There was also **mutate** – to add variables to the dataframe:

```
piaac <- piaac %>%
  mutate(relativewage = Income / mean(Income, na.rm=T))
```

Note that the `mean(Income)` would here find the average earnings over the entire dataset. But if you use `group_by()`before it, it would take the average over this group[1].

And we looked at **arrange**, which would just reorder the dataset by some variables.

[1] A quick comment – you would like to add `ungroup()` at the end as the groups would be saved to the dataset otherwise and later on this could cause some unexpected behaviour.

Couple of additions:

- `summarize()` can only deal with functions which return a single value. There is another verb called `do()`, which can deal with functions which return a data frame, but the newest way is to use a package called `purrr` and it's function called `map()` for this. We will not touch it further here though.
- there is a verb called `rename()`, which renames the variables.

```
library(dplyr)

piaac.tmp <- piaac %>%
  rename("personId"="seqid")
```

Conflicts in packages

Install and load the **plyr** package. Note the warnings, they are easy to miss. Now run the previous command again:

```
numeracyaverages <- piaac %>%
  group_by(gender, studyarea)%>%
  summarize(meannum=mean(Numeracy, na.rm=TRUE))
```

See, what happens? We dont have them grouped anymore, as we are now using the functions from `plyr`, not `dplyr`. The only way out is to unload `dplyr` and then load it again (just typing `library(dplyr)` will not help, unloading is neccesary!). There are two ways to do it

– either uncheck dplyr from the `Packages` tab and check it again[2], or write:

[2] Sometimes you'd have to do it two times.

```
detach("package:dplyr", unload=TRUE)
```

One way to solve this problem is to use the package name with the function name (and if you are using a lot of conflicting packages, you learn to do this):

```
numeracyaverages<-piaac %>%
  dplyr::group_by(gender, studyarea)%>%
  dplyr::summarize(meannum=mean(Numeracy, na.rm=TRUE))
```

```
## `summarise()` regrouping output by 'gender' (override with `.groups` argument)
```

Back to dplyr

Lets try it some more. Lets create a new dataset called `numeracyaverages`, so that it would contain mean numeracy levels by gender and studyarea:

```
numeracyaverages <- piaac %>%
  group_by(gender, studyarea)%>%
  summarize(meannum=mean(Numeracy, na.rm=TRUE))
```

Your turn:

- Change the variable names in `piaac` to be more pleasing to the eye and graphs[3]:
  - Problem.solving.skills to "Problem solving skills"[4]
  - age to Age
- Remove all the lines where either Health, Education or Numeracy is missing[5].
- This is now very tricky stuff: find the mean, lower and upper confidence interval for the variable numeracy by Education level. Use the function `CI()` from package **Rmisc** for this. NB! You do not have this package installed yet. What is worse – it will load the package **plyr** when you read it in. And this will f up your `dplyr`. You now need to detach dplyr and then reattach it! One way to do this is unmark it in the Packages pane and then mark it again. Another thing - `CI()` will return 3 values, but summarize can only deal with 1. What should we do? Take a look at what `CI()` returns, by running `CI(piaac$numeracy)` – it returns a vector of three. How to access a single value? Just add `[1]` or `[2]` or `[3]` at the end of the call like this: `lower=CI(numeracy)[3]`. Write the resulting dataframe in a data object called **averages**.

[3] You can do it all in one `rename()` call, just remember to overwrite your previous dataset with the new one.
[4] You will need to use the quatation marks to quote the name if there are spaces in it!

[5] Remember, variable!=NA will not work, you would want to use the `!is.na()` function.

This is what you should have as a result:

```
averages
```

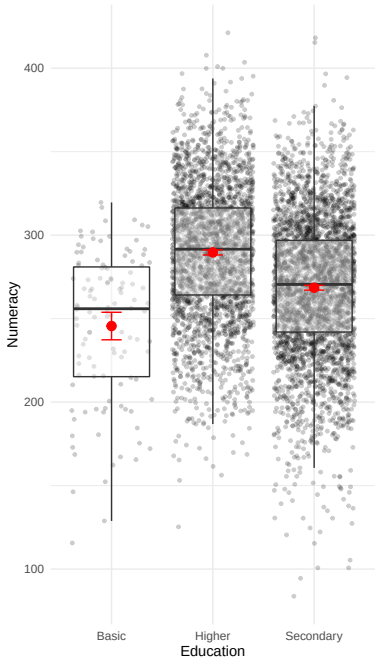
```
## # A tibble: 3 x 4
##   Education upper  mean lower
##   <chr>     <dbl> <dbl> <dbl>
## 1 Basic      254.  246.  237.
## 2 Higher     291.  290.  288.
## 3 Secondary  270.  269.  267.
```

ggplot2

A typical plot in the ggplot language looks something like this:

```
ggplot(data=mydataset, aes(x=firstvariable, y=secondvariable))+
  geom_something(aes(color=groupingvariable))+
  geom_otherthing(size=4)+
  facet_wrap(~othergroupingvariable)
```

If you want something at the graph to be connected to your data, you will need to put it into the **aes()**-call. If you want to tell it yourself (like the `size=4` here), it has to be outside of the `aes()`-call.



Your turn:

Lets combine data from datasets `piaac` and the new dataset `averages` which we just created and create the graph you are seeing on the side.

- First lets add the stuff from piaac dataset: tell it that x should be connected to Education and y to Numeracy. Remember the `aes()`! Then add a new `geom_jitter()`, setting alpha for it to be 0.2 and size to be 1.
- Now add `geom_boxplot()`. Make it transparent as well, setting alpha to be 0.4. Remove the outliers by setting `outlier.shape=NA`.
- Now lets add the averages from the new dataset that we just created. You need to overwrite the `data=` parameter in the following layers! First add a `geom_point()` layer and put a red point with size=3 on the graph to denote the average.
- Then add `geom_errorbar()`. Look at which statistics does it require. You want to connect these ymin and ymax to the variables lower and upper in your dataset. Set the color to be "red" and add a parameter width=0.2. You can add `theme_minimal()` at the end.

Factors

We learned that for dealing with factors there is a package called `forcats`. To manually change the order of factor levels, you can use `fct_relevel()`.

For example: the current order of the `piaac$Health` (if it would be cast to a factor) would be:

```
levels(as.factor(piaac$Health))
```

```
## [1] "Excellent" "Fair"      "Good"      "Poor"      "Very good"
```

Lets change it:

```
library(forcats)
piaac <- piaac %>%
  mutate(Health = fct_relevel(Health, "Poor", "Fair", "Good", "Very good", "Excellent"))
levels(piaac$Health)
```

```
## [1] "Poor"      "Fair"      "Good"      "Very good" "Excellent"
```

But you could also have done it with a one-liner:

```
piaac$Health = fct_relevel(piaac$Health, "Poor", "Fair", "Good", "Very good", "Excellent")
```

To change the factor levels, you can use `fct_recode()`:

```
unique(piaac$children)
```

```
## [1] "Has children" "No children"  NA
```

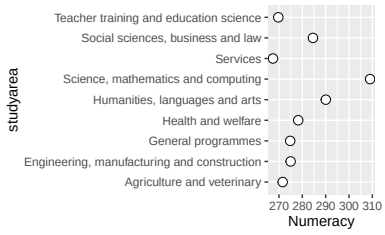
```
piaac <- piaac %>%
  mutate(children=fct_recode(children, "Yes"= "Has children", "No"="No children"))
levels(piaac$children)
```

```
## [1] "Yes" "No"
```

Third thing you would do quite a lot is order the level of factors by some variable. Lets create again a simple summarizing table of average numeracy scores by study area:

```
numscores <- piaac %>%
  group_by(studyarea)%>%
  summarize(Numeracy = mean(Numeracy))

ggplot(numscores, aes(x=Numeracy, y=studyarea))+
  geom_point(shape=21, size=3, fill="white")
```



These are not in a nice order at all. What can we do to plot them so that the highest would be on top and the lowest in the bottom? We can change the order of level manually, but there is an easier way. We can use `fct_reorder()` from `forcats`:

```
numscores$studyarea <- fct_reorder(numscores$studyarea, numscores$Numeracy)
```

Now try to run the plot again!

Your turn:

- Take the piaac dataset and change the levels of factor in Education variable so that "Higher" will become "High", "Basic" to "Low" and "Secondary" to "Medium". Use the **fct_recode()** function from the **forcats** package. The example from the help section will show you how to do it.
- Change the order of Education factor levels so that it will be Low, Medium, High. Use the **fct_relevel()** function from the forcats package to do it. The example in the help section will show you how to do it.

Lists

List is a data structure that we will meet quite often. Data frame consists of vectors of the same length, list can include whatever.

For example[6]:

[6] Note that we give names to list elements. R will not do it automatically, as it does for data frames.

```
name<-"Masha"
surname<-"Mishka"
length<-1.65
peripherals=data.frame(nr.of.fingers=c(5,5), nr.of.toes=c(5,5))
mashadata <- list(first.name=name, surname=surname, length=length, peripherals=peripherals)
class(mashadata)
```

```
## [1] "list"
```

```
mashadata
```

```
## $first.name
## [1] "Masha"
##
## $surname
## [1] "Mishka"
##
## $length
## [1] 1.65
##
## $peripherals
##   nr.of.fingers nr.of.toes
```

```
## 1             5          5
## 2             5          5
```

Lists can contain lists that can contain lists etc.

To address a single element of list we can use the $-sign:

```
mashadata$first.name
```

```
## [1] "Masha"
```

Or double straight brackets (the name of the element has to be quoted then):

```
mashadata[["first.name"]]
```

```
## [1] "Masha"
```

You can in fact use single straight brackets, but they will return a list, not the initial data type (the data.frame, which we included in the list), which you will want to get in practice.

```
class(mashadata["peripherals"])
```

```
## [1] "list"
```

```
class(mashadata[["peripherals"]])
```

```
## [1] "data.frame"
```

If we want to access the vector **nr.of.fingers** from the data frame **peripherals** from the list **mashadata**, then we can do it in the following way:

```
mashadata[["peripherals"]]$nr.of.fingers
```

```
## [1] 5 5
```

but not like this:

```
mashadata["peripherals"]$nr.of.fingers
```

```
## NULL
```

There is one thing to note when dealing with lists. Data frame will not let you include two columns with the same name. List would not care less:

```
list2<-list(a=3, b=4, a=5)
list2$a #Not even a warning!
```

```
## [1] 3
```

The reason we pay so much attention to lists, is that a lot of the results R will give you, are in the form of lists. Take `t.test()` – t test compares two samples and asks if it is statistically viable that they come from the same population, that the differences could be just thanks to sampling error. If we only give it one vector as an input, it will tell us whether the mean is statistically different from 0.

Lets do a t-test for **Literacy**-variable (measuring literacy levels) in our **piaac** dataset[7]:

[7] Note that we are using the piaac data incorrectly here! In reality there are ten different plausible values given for literacy levels, as there is considerable uncertainty involved. Do do it correctly you should use all the information – this is possible with a package called **svyPVpack**. Neither do we use the weights here.

```
t.test(piaac$Literacy)
```

```
##
##  One Sample t-test
##
## data:  piaac$Literacy
## t = 511.9, df = 6203, p-value < 2.2e-16
## alternative hypothesis: true mean is not equal to 0
## 95 percent confidence interval:
##  278.3766 280.5169
## sample estimates:
## mean of x
##  279.4468
```

It will give us 95% confidence intervals for the population. Meaning – considering this sample, believing that this is a random sample, we would believe that the correct average for the whole population should be inside this interval with quite a high certainty. The average for the sample is 275,64, we believe, that the average for the whole population should fall somewhere between 275...277.

`t.test()` shows us the results, but if we save it to a data object, we will find out, that it actually produces a list. Lets do it:

```
result <- t.test(piaac$Literacy)
class(result)
```

```
## [1] "htest"
```

```
typeof(result)
```

```
## [1] "list"
```

```
str(result)
```

Your turn:

- confidence intervals are inside this list as a vector called **conf.int**. Can you access it?

1) create a variable called **lower**, and assign it the lower value of conf.int (the first value in this vector)
2) create a variable called **upper**, and assign it the upper value of conf.int (the second value in this vector)

Functions

Writing a custom function or command in R is extremely simple.
Take a look at the example:

```
timesthree<-function(x){ #we name the function and say that its input will be named x
  result <- x * 3  #we do some stuff
  return(result) #we will return the answer
}
```

You now need to make R aware of the new function - just select it and click Run (or cntrl+enter).

Lets find out how it works:

```
timesthree(4)
```

```
## [1] 12
```

```
timesthree(c(3, 4, 5))
```

```
## [1]  9 12 15
```

Remember, the functions could have default values:

```
multiplydivide<-function(x, multiplier=3, divisor=1){
  result <- x * multiplier / divisor  #use the input
  return(result) #return the result
}
```

If we will not give any values for **multiplier** or **divisor**, then it will use the default values. But we can change them:

```
multiplydivide(x=4)
```

```
## [1] 12
```

```
multiplydivide(x=4, multiplier=1, divisor=2)
```

```
## [1] 2
```

Minimal about conditional statements

You can write conditional statements in R in the following way

```
if (condition) {
  whattodo
} else {
    dosomethingelse #not required
  }
```

For example[8]:

[8] You do not need to use the square brackets, if the `whattodo` is a single line. For more lines you need the brackets.

```
nameOfTheBear<-"George" #let us have a bear named George

if(nameOfTheBear=="George") friendOfABear<-"Tom"
if(nameOfTheBear=="Bill") friendOfABear<-"Mary"

friendOfABear #who is the friend of the bear?

## [1] "Tom"
```

Your turn:

1) create a function called **subtract**, which takes as an input two data objects (for examplem *a* and *b*) and subtracts one from the other.

2) create a function that takes a filename as an input (e.g. filename="piaac.csv"), adds it to "`http://www.ut.ee/~iseppo/`", reads the file from resulting URL with `read.csv()` and returns the data object that it creates this way. How to add two strings? Try `paste("onestring", "anothersting")`. Note that this will result in a string which has a space as a separator. To remove the space (as you will want to), you will need to do `paste("onestring", "otherstring", sep="")`. Note that one of your strings will be inputted as a data object, so you should do smth like `paste("http://www.ut.ee/~iseppo/", filename, sep="")`.

3) for those who are too quick: create a function that takes two inputs – a filename and an additional parameter called readr, which is FALSE by default. If it is false, the function should read the file in with `read.csv()`, if it is TRUE, the function should use the readr version: `read_csv()`