Lab: Data Cleanup

Introduction

We can loosely split the "Data cleanup" part of a project in three stages:

- Loading the data from a spreadsheet file or some other kind of storage. The readr and readxl packages help us with this stage.
- "Tidying" the loaded data to ensure the data is structured in a way that facilitates analysis. This mainly means that each variable is in one column. For example instead of having measurements over 5 days split across 5 columns, there should instead be one column of 5 times as many observations, with a second column identifying the day index for each. The tidyr package provides methods that help with this stage.
- Manipulating the data in various ways to obtain summaries or prepare for graphing. The dplyr package offers a rich set of methods to help with that.

All the packages mentioned above are part of the "tidyverse[1]" package collection, and are automatically included in our hanoverbase package.

Here is a summary of some of the key methods we will learn about here:

read_excel	Read dataset from Excel spreadsheet
gather	Turn multiple columns to one, use a 2nd variable for the column name
separate	Split an entry into multiple parts and build different columns
rename	Change the name of a variable
mutate	Change or add a variable via computation using other variables
select	Restrict the dataset to a subset of the variables
filter	Only select certain rows based on some criterion
group_by	Group the rows of a dataset according to a factor variable
summarize	Produce a single value from each group in a grouped dataset
bind_rows	Bind datasets together by putting one below the other
recode_factor	Rearrange/rename a factor's levels
factor	Create a new factor or turn a character vector to factor

Importing

We start by importing the data from an external source. For this example, we will use the datasets/compression.xlsx[2] spreadsheet. It might help to view the spreadsheet in Excel.

In terms of providing us access to the data, the sheet is somewhat messy: It has the data spread over multiple tabs/sheets, and sheets often contain non-data components like summaries and graphs.

We will start with reading the data from the first sheet, titled "Upper Arm Circ". You will need to first have the dataset imported into your project directory (you did create a new project with its own project

[1] https://www.tidyverse.org/
[2] ../datasets/compression.xlsx

directory, right?). You can then click on the dataset to open up the import dialog. We will not fully use the import dialog, but it is a good starting point to look at the data.

In the import dialog, you can select the sheet you want, here the “Upper Arm Circ”, from the “Sheet” drop down in the “Import Option” section. Then you would want to copy the couple of lines on the right, which do the import:

```
library(readxl)
compression <- read_excel("~/statsLabPractice/compression/compression.xlsx",
    sheet = "Upper Arm Circ")
```

Then we would paste these lines in an r-chunk after canceling the import dialog. Remember that the first line loads a package for us, and only needs to be placed once.

Then we need to change the second line, to make sure we only try to load the first 8 data points, and also improve the stored data set name:

```
compression.uac <- read_excel("~/statsLabPractice/compression/compression.xlsx",
    sheet = "Upper Arm Circ",
    range = "A1:O9")
```

We learned about this extra parameter by reading the documentation for read_excel.

Let us also load another one of the sheets:

```
compression.lac <- read_excel("~/statsLabPractice/compression/compression.xlsx",
    sheet = "Lower Arm Circ",
    range = "A1:O9")
```

This shows the basics of importing data. You can explore the readr and readxl packages to find out more. Also check out the haven package for loading SPSS, Stata and SAS data. There are also ways to access a database directly; the package dbplyr[3] might be a good place to start for that.

Tidying

We will take various steps to “tidy” the data. Before we do that, let’s discuss the data a bit. The first column represents of course the subject ids, and each row of data corresponds to a subject. The “CS...” variables correspond to the subject using a “Compression Sleeve” while the “NT...” variables correspond to the subject receiving “Pneumatic Compression” therapy. the “...UAC...” part represents that these are “Upper Arm Circulation” measurements, and finally the remaining bits “PR, PO, 1, 2, 3, ...” represent the time variable (pre-exercise, post-exercise, one day later etc).

The data is what is known as a “wide” form: In a “tidy” version of the data the kind of treatment, the quantity being measured, and the time of observation would all be three separate factor variables and we would have a single measurement column. In effect all the measurements currently present in the sheet will be placed in one column, and various information that is currently coded in the variable names will instead be stored in different variables/columns specifically created for that purpose. For instance there will be one variable to distinguish whether the measurement is during to the Compression Sleeve phase or during the Pneumatic Compression phase, another variable to describe what was measured (upper arm

[3] https://dbplyr.tidyverse.org/articles/dbplyr.html

circ, lower arm circ etc), and another variable to distinguish which part of a phase we are in (pre/post/day 1 etc).

We will now convert the data sheet in this form. The first step in doing so is to isolate the CSU columns as well as the subject column. We will use the subset method for that, which picks out a subset of the variables. There are various ways to specify which variables to select, and you can look at the documentation or cheatsheets for that.

```
compression.uac %>%
    select(X__1, CSUACPR:CSUAC5)
```

We could have stored this result in a dataset, but we will instead add another process step in the pipeline, which does the transformation. It is called gather, and it simply collects some columns and puts them together into one column (the -X__1 tells it to combine all the other columns except X__1):

```
sleeve.uac <- compression.uac %>%
    select(X__1, CSUACPR:CSUAC5) %>%
    gather(key="time", value="value", -X__1)
View(sleeve.uac)    # Do in console
```

Notice how the new time column contains the variable names, and we now have 7 times as many rows, one for each combination of a subject and a variable.

Next we need to split the variable time into three parts:

- The fact that it is a “compression sleeve” treatment
- The fact that it is the upper-arm
- The timing (pre, post, day 1 etc)

The separate method helps us with this. Its syntax will look a bit weird. The second argument (into=) specifies the names to give to the three parts, while the third argument (sep=) specifies the cutoff points (after the first two letters, after the first 5 letters). This effectively splits each value like “CSUAPR” in three parts: the first two letters, the next three letters, and the rest.

```
sleeve.uac2 <- sleeve.uac %>%
        separate(time,
                 into=c("treatment", "measurement", "time"),
                 sep=c(2,5))
View(sleeve.uac2)
```

Finally, we will fix the first column: First we want to change its name to “subject”, with the method “rename”:

```
sleeve.uac2 %>%
    rename(subject=X__1)
```

And then change the strings “sub 1” into the numbers themselves, using mutate. For this to work, we also need to load a string-manipulation library called stringr :

```
sleeve.uac3 <- sleeve.uac2 %>%
    rename(subject=X__1) %>%
    mutate(subject = str_extract(subject, "\\d+"))
```

Creating a custom function for repeatable steps

We will likely need to repeat all these steps for each part of the dataset, and they will be more or less the same steps, so let's see if we can make a function out of it.

In order to make a function, we need to determine what it is that may be changing each time we do this. These are then "parameters" for this function. In our case the one thing that is of course changing every time is the data frame, and we will use the `df` parameter name to refer to it. But if we look at some of the variable names in the other sheets, we will notice that the middle part of the name some times uses 2 letters and sometimes 3. So this would be another parameter we can provide, `middleLetters`.

```
convert.to.long <- function(df, middleLetters) {
    df %>%
        gather(key="time", value="value", -X__1) %>%
        separate(time,
                 into=c("treatment", "measurement", "time"),
                 sep=c(2, 2 + middleLetters)) %>%
        rename(subject=X__1) %>%
        mutate(subject = str_extract(subject, "\\d+"))
}
```

Then we could simply have done the following:

```
sleeve.uac <- compression.uac %>%
    select(X__1, CSUACPR:CSUAC5) %>%
    convert.to.long(3)
```

And if the variable names we wanted to use used a different number of "middle letters" we could provide that as a parameter. We can also set a "default value":

```
convert.to.long <- function(df, middleLetters=3) {
    ...
```

Then we do `convert.to.long()` it will have the same effect as `convert.to.long(3)`.

We will now do the same for the other part, the NT columns:

```
pneumatic.uac <- compression.uac %>%
    select(X__1, NTUACPR:NTUAC5) %>%
    convert.to.long(3)
```

We will repeat this for the "lower-arm-circulation" variables:

```
sleeve.lac <- compression.lac %>%
    select(X__1, CSLACPR:CSLAC5) %>%
    convert.to.long(3)
pneumatic.lac <- compression.lac %>%
    select(X__1, NTLACPR:NTLAC5) %>%
    convert.to.long(3)
```

Binding data sets together

Now we have four data sets that all look similar in terms of the columns they contain etc. We would like to merge them together by simply placing them below each other and matching the columns. We can use the `bind_rows` method for this:

```
alldata <- bind_rows(sleeve.uac, pneumatic.uac, sleeve.lac, pneumatic.lac)
```

There are many tools that help with more complicated data merges, look at the dplyr cheatsheet for some examples.

Handling factor variables

Now it is time to more properly code our factor variables, using mutate. In R, there are different *modes* for vectors/variables. One mode is *numeric* for scalar variables, another is *character* for strings. There is also *factor* which is what should be used for factor variables. A factor typically uses numbers for the underlying codes/levels, and it also contains a *value label* for each code/level. For example, if we take the treatment variable and turn it into a factor:

```
subject$treatment
subject$treatment %>% factor()
```

As we may have briefly mentioned brefore, the dollar sign is used to pick out a column from a dataset. You may also use the pluck command, part of the purrr package:

```
alldata %>% pluck("treatment") %>% factor()
```

If you compare the outputs above, you will notice the results looking a tad different when we used the factor function. The important thing to know is that in order to do some of the standard statistical analysis steps, we need variables to be coded as "factors":

```
alldata <- alldata %>% mutate(
    subject = factor(subject),
    treatment = recode_factor(treatment, CS="Compression Sleeve", NT="Pneumatic Compression"),
    measurement = recode_factor(measurement, UAC="Upper Arm Circ", LAC="Lower Arm Circ"),
    time = recode_factor(time, PR="Pre", PO="Post", `1`="Day 1", `2`="Day 2",
                         `3`="Day 3", `4`="Day 4", `5`="Day 5")
    )
```

We are now ready to do some work with this data. Here's some examples of graphs we can produce, you'll see more about this sort of graph in the graphing lab[4].

```
ggplot(alldata) +
    aes(x=time, y=value, group=subject, color=subject) +
    geom_path() +
    facet_grid(measurement~treatment)

ggplot(alldata) +
    aes(x=time, y=value, group=treatment, col=treatment) +
    stat_summary(geom="pointrange", fun.data=mean_se, fatten=3, alpha=0.6,
        position=position_dodge(width=0.1)) +
    geom_line(stat="summary", fun.y=mean, position=position_dodge(width=0.1)) +
    facet_grid(measurement~.)
```

[4] LabIntroGgplot.html

Further work and transformations

Let us move on to doing some numerical work with this data. For instance we may want to compute the means and standard deviations for each day, for each treatment and for each type of measurement. In order to perform such computations, we need to first "group" the data by the grouping variables we want to use. The group_by method helps us with that. Then we can use the summarize function to produce numerical summaries per group.

```
summaries <- alldata %>%
    group_by(treatment, measurement, time) %>%
    summarize(mean=mean(value),
              sd=sd(value),
              median=median(value),
              q1=quantile(value, 0.25))
```

We could then visualize those summaries in some way:

```
ggplot(summaries) +
    aes(x=time, y=mean, color=treatment, group=treatment) +
    geom_line() + facet_grid(measurement~.)
```

Let us try a more complicated transformation. Since different subjects start from different values at the pre-stage (you may have noticed that in the first graph you created earlier), we could try to normalize that initial amount, and compute the other values as relative to that that initial amount. What we need to do here is again group by the subject, treatment and measurement, but not time, and now within each category we need to mutate our values instead of summarizing:

```
relativedata <- alldata %>%
    group_by(subject, treatment, measurement) %>%
        mutate(relChange=value/first(value))
```

We'll also want to remove the "pre" entries, as they are no longer really applicable:

```
relativedata <- alldata %>%
    group_by(subject, treatment, measurement) %>%
        mutate(relChange=value/first(value)) %>%
        filter(time != "Pre")
```

Let's repeat our earlier graphs now. For the second graph we'll adjust the y-axis to show relative increase (more about this stuff no the graphics session):

```
ggplot(relativedata) +
    aes(x=time, y=relChange-1) +
    geom_path(aes(group=subject, color=subject)) +
    facet_grid(measurement~treatment) +
    labs(y="relative change from pre") +
    guides(color=FALSE) +
    scale_y_continuous(labels = scales::percent)

ggplot(relativedata) +
    aes(x=time , y=relChange-1, group=treatment, col=treatment) +
    stat_summary(geom="pointrange", fun.data=mean_se, fatten=3, alpha=0.6,
        position=position_dodge(width=0.1)) +
    geom_line(stat="summary", fun.y=mean, position=position_dodge(width=0.1)) +
    facet_grid(measurement~.) +
    labs(y="relative change from pre") +
    scale_y_continuous(labels = scales::percent)
```

Practice

You likely have a dataset of your own that you might want to work with to practice the above. If you don't, then here's a possible project:

1. The `GapMinder` website has a wealth of information on each country over many years. We will suggest some analyses to perform but you are free to pursue a different analysis. There is actually a R package that can load a lot of this data in more ready form, but we will use this as an opportunity to access data from the web and from multiple sources.
   - The first step would be to access the data sets from the website, `https://www.gapminder.org/data/`. You can search for the requisite datasets there. You have two options: You can either download the file and include it in your project, or you can use the file's URL directly into R. We will recommend this first path in this instance, but keep in mind that the second path is possible. We will give you details on which datasets to load a bit further down.
   - Start by creating a new project. You may also want to start an RMarkdown document to keep your work organized.
2. We will use the `life expectancy (years)` variable.
   - Search for the corresponding dataset, and dowload the file. Then upload the file into your project directory (you may want to rename it to something simpler), and import its first sheet into the project by clicking the uploaded file.
   - Start by using `rename` to rename the first variable to say `country` or something similar.
   - Use `gather` to turn all the different year variables into a single year. The resulting dataset should have three columns: country, year, and "life expectancy". You can use `-country` as the column specification; this will tell `gather` to use all columns except for `country`. Also make sure to check the documentation and include a setting to "remove the missing values".

   One thing we have not covered before is using variable names with spaces in them, like `life expectancy` in this example. In many places, you will need to enclose them in backticks to make that work, like so `` `life expectancy` ``.
3. We will now add another variable, the per-capita income. The variable you want is called "Income per person (GDP/capita, PPP$ inflation-adjusted)". Follow much the same steps as in the previous point. You can call the values column whatever you want of course, but we will be using simply `income` for it.
4. We will also want to get the total population of each country, stored in a dataset named `Population, total`. Use similar steps for it.

   (Optional step) If you are feeling adventurous, you can pursue the possibility of writing a function to make these steps easier, since we more or less repeated the same steps three times.

5. Now we want to bring these three datasets together, by matching entries on the country and year. the inner_join function can help you do that, make sure to look at its documentation. Store the result in a variable (we called it gpdata).

   You have a choice to make here, actually, inner_join and full_join . full_join will include entries for country and year combinations where one of the two values of income and life expectancy is missing. This gives us more data to work with, but we might end up having to throw away some values later on anyway when trying to do graphs or analysis and so on. We will use inner_join instead, which will only keep the entries where both the income and the life expectancy are known.

6. We need to change the "year" value into numeric values, as currently they are being treated as character values. You will need to use mutate and as.numeric for this.

A challenge: Matching Country Names Warning: This section is somewhat challenging, as it attempts to reconcile differences in country names between different datasets. The end result is simply to have a region variable associated with each country. Feel free to skip this on a first go, and straight to the "practice continued" section.

We will incorporate a variable for the various regions of the world. As this variable does not appear to be available in the gapminder data, we will need to get it from another source. In this instance, the world bank seems to include it in some of their available datasets: http://databank.worldbank.org/data/download/WDI_excel.zip. - Download the file and unzip it, then upload the corresponding Excel to your project folder. - The second sheet, named country, contains the information we need. Load that into a variable, we called it wdi. - Since we only care about the country name and continent, use select to only choose the four columns Short Name, Table Name, Long Name, Region.

The challenging part now is to match the country names between the two datasets, and the conventions on naming some of the countries. For instance, let us start by making a list of all the countries in gpdata and in the Short Name and Table Name entries of the wdi data, and look at their difference:

```
gpCountries <- gpdata %>% pluck("country") %>% unique() %>% sort()
wdiCountries <- wdi %>% pluck("Table Name") %>% unique() %>% sort()
setdiff(gpCountries, wdiCountries)    # names in gpCountries but not wdiCountries
```

We'll see a number of countries missing that really shouldn't be missing. We could also try to use the Short Name column instead, but if you do that you'll notice that it too is missing some countries.

We could of course simply edit the Excel sheet and change the country names. But then we may need to do that each time we want to update the data from the originals. Manually changing the Excel sheets is not very sustainable in the long run. So we will look for a programmatic way to do it. Here's the steps we will take:

- We will create a new column gpName in the wdi data that contains the Short Name if that was in the gpCountries list, or the Table Name if *that* was in the gpCountries list, or otherwise it is NA (missing value).
- Next we will find out which countries in gpdata do not have a corresponding entry. Hopefully that will be a small list. We will then manually write a small dataset that contains an appropriate matching name for those countries, by inspecting the names that exist in the Short Name column.
- Then we will join this spreadsheet with the wdi to fill in the remaining gpName entries.
- Lastly, we will then join our gpdata with this updated wdi spreadsheet in order to get the continent names.

So let's start with the first part, adding a new column in wdi. We can use the case_when method for this: It tries a series of alternatives in turn and uses the first that matches:

```
wdi <- wdi %>% mutate(gpName = case_when(
  `Table Name` %in% gpCountries      ~ `Table Name`,
  `Short Name` %in% gpCountries      ~ `Short Name`,
  TRUE                               ~ as.character(NA)))
```

Now we need to create a little dataset that contains these countries and their appropriate wdi entries. We will use the tribble command for that, which allows us to nicely align the values vertically. We do need to use the search feature in the data view in order to find what the appropriate names would be:

```
countryNames <- tribble(
  ~gpName, ~`Short Name`,
  "Bahamas", "The Bahamas",
  "Cape Verde", "Cabo Verde",
  "Cote d'Ivoire", "Côte d'Ivoire",
  "Gambia", "The Gambia",
  "Hong Kong, China", "Hong Kong SAR, China",
  "Lao", "Lao PDR",
  "Macao, China", "Macao SAR, China",
  "North Korea", "Dem. People's Rep. Korea",
  "Sao Tome and Principe", "São Tomé and Principe",
  "South Korea", "Korea",
  "Syria", "Syrian Arab Republic"
)
```

We could not find an entry for Taiwan in the wdi data. We will need to assign its region manually after we do all the needed joins.

Now we want to join this variable into the wdi data. This is a bit tricky: A join sounds like the right thing to do, with Short Name being the common variable. But the way R does this is to create two separate columns: One for the gpNames from wdi and another for those from countryNames. We need to use coalesce along with mutate to merge the two together.

```
wdi2 <- wdi %>% full_join(countryNames, by="Short Name") %>%
  mutate(gpName = coalesce(gpName.x, gpName.y))
setdiff(gpCountries, wdi2$gpName)
```

You should now see the setdiff method return a character(0) value, meaning there are no entries. Good, that means each value in gpCountries is now accounted for in wdi. We will still need to deal with Taiwan.

Now, we want to merge in the region values from wdi into our gpdata set. This will be a simple join, matching the country to the gpName. But we only want to bring in the region variable, so before joining we will select only a few columns from wdi (left_join here tells it to ignore entries in wdi that don't have a matching value in gpdata):

```
gpdata <- gpdata %>% left_join(wdi %>% select(gpName, Region),
     by=c(country="gpName")) %>%
     rename(region="Region") %>%
     mutate(region=replace(region, country=="Taiwan", "East Asia & Pacific")) %>%
     mutate(region=factor(region))
```

Notice the first mutate line above. It tells it to replace the region variable by using whatever was in the region variable before but whenever the country is Taiwan it should set the region to be East Asia and Pacific.

And with that, we should be done: In order to check, try to use group_by region and summarize using n_distinct to count each country only once. If done correctly, this should give you a reasonable count of countries for each region (for example, 3 countries in north america).

Practice Continued This ends the "data tidying part". We can now do some analyses on these numbers.

1. Use arrange to reorder the values and discover what the most recent year in the data is. You can also find that out from the viewer, or you can use summarize or pluck along with the max function.
2. Use filter to consider only the subset of the data that relates to the most recent year, then arrange to order the data according to income, and find the countries with the 5 highest and 5 lowest per capita incomes.
3. For each country, determine the number of data entries present.
4. Determine for each year the country with the highest per-capita income. You can do this with a group_by and a filter .
5. (If you did the region part above) Compute total populations for each region of the world and each year.
6. Compute total income for each row by multiplying together the population and income variables, and add that as a new variable.