R Training

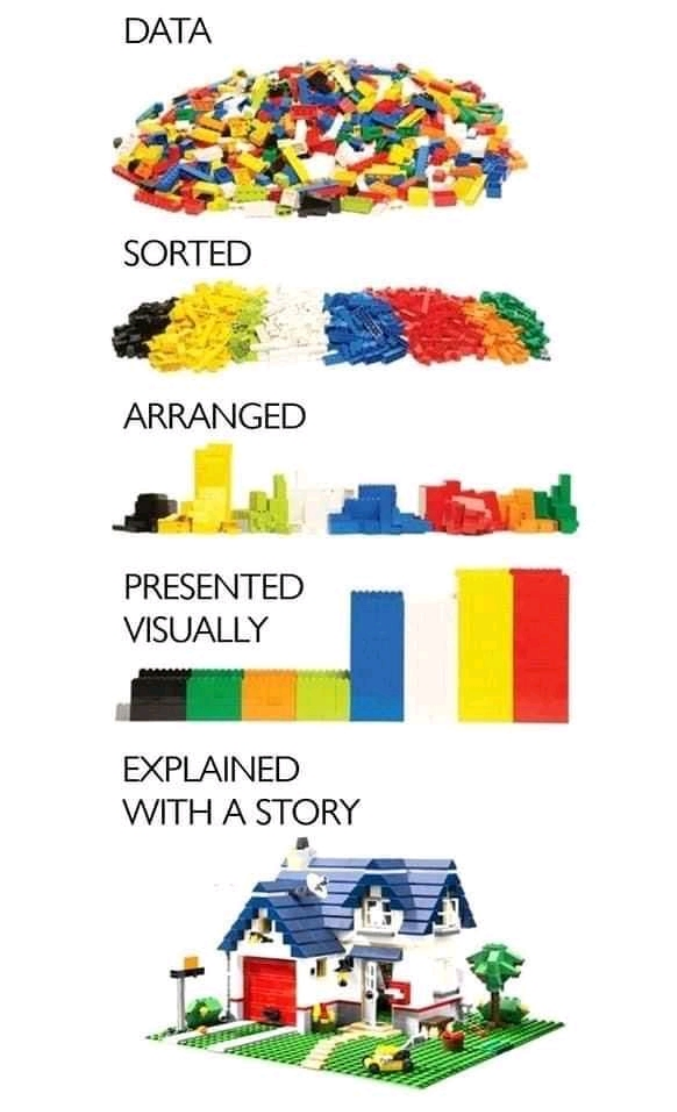
Data Manipulation

Yebelay B. Berehan

September 22, 2021

Email: yebelay.ma@gmail.com

Data Manipulation



Data management is a very vast tasks like:

- Read in data and examine data for errors
- Manipulate data to create summaries,
- Creating new variables (including recoding and renaming existing variables).
- Sorting and merging datasets, aggregating data,
- Reshaping data,and sub setting datasets (randomly sampling observation, dropping or keeping variables).
- Therefore, for data manipulation we install tidyverse package

What is Tidyverse?

- Tidyverse a suite of packages that implement tidy methods for data importing, cleaning, and wrangling.
- Contains series of packages useful for data analysis which work together well.
- To look the series of these packages, run the code

```
tidyverse_packages()
```

- Some of them are considered core packages (**tidyr, dplyr, ggplot2**).
- Other packages, like
  - lubridate (to work with **dates**) or
  - haven (for **SPSS, Stata, and SAS data**) that you are likely to use.

What are dplyr and tidyr?

- The package **dplyr** provides easy tools for the most common data manipulation tasks and **tidyr** addresses the common problem of wanting to reshape our data.
- Most common dplyr functions are:
  - select(): Subset columns
  - filter(): Subset rows on conditions
  - mutate(): Create new columns
  - groupby()&summarize(): Create summary by category variable
  - arrange(): Sort results
  - count(): Count discrete values

Selecting columns and filtering rows

- To select columns of a data frame, use `select()`.

```
library(readxl)
mydata<-read.csv("Ethiopia_R_training_edit.csv")

# View(mydata)
```

```
library(dplyr)
sampledata<-select(mydata, Sex, Age, Weight1, Adherance, Ba
```

- To select all columns except certain ones, put a - in front of the variable to exclude it.

```
select(mydata,-FacilityName)
```

Pipeline or piping operator

- Pipes in R look like %>% and strings together commands to be performed sequentially
- The pipe passes the data frame output that results from the function right before the pipe to input it as the first argument of the function right after the pipe.

```
third(second(first(x)))
```

- This nesting is not a natural way to think about a sequence of operations.
- The %>% operator allows you to string operations in a left-to-right fashion.

```
first(x) %>%
second %>% third
```

Advantages of Pipe oprator

- Pipes used to reduce multiple steps, that can be hard to keep track of.
- difficult to read if too many functions are nested, so pipes make it easy
- Look at the three syntax

```
data1<-filter(sampledata,Age > 15)
data2<-select(data1,Sex,Weight1,Age,Adherance,BaselineCD4)
```

```
data_2<-select(filter(mydata,Age>15),Sex,Weight1,Age,
                    Adherance,BaselineCD4)
View(data_2)
```

```
selecteddata<-mydata %>% filter(Age > 15) %>%
  select(Sex, Weight1, Height1, Age, Adherance, BaselineCD4
```

Mutate

- Frequently you'll want to create new columns based on the values in existing columns, for example to do unit conversions, or to find the ratio of values in two columns. For this we'll use mutate().
- To create a new column of BMI:

```
newdata <- mydata %>%  select(Weight1, Height1) %>%
  mutate(BMI = Weight1/((Height1/100)^2))
```

- If this runs off your screen and you just want to see the first few rows, you can use a pipe to view the head() of the data.

```
newdata %>%  select(Weight1, Height1) %>%
  mutate(BMI = Weight1/((Height1/100)^2)) %>%  head()
```

Mutate

- The first few rows of the output are full of NAs, so let's start by removing missing observations for sex, weight and adherence using filter().

```
complete <- sampledata %>%
  filter(!is.na(Weight1), # remove missing weight
         !is.na(Adherance),# remove missing Adherance
         !is.na(Sex))  # remove missing sex


View(complete)
```

- is.na() is a function that determines whether something is an NA. The ! symbol negates the result, so we're asking for every row where weight is not an NA.

Summarize() function

- Many data analysis tasks can be approached using the split-apply-combine paradigm: split the data into groups, apply some analysis to each group, and then combine the results.
- dplyr makes this very easy through the use of the **group_by()** function.
- The **summarize()** function
- **group_by()** is often used together with **summarize()**, which collapses each group into a single-row summary of that group.
- **group_by()** takes as arguments the column names that contain the categorical variables for which you want to calculate the summary statistics.

Summarize() function

So to compute the mean weight by Adherence:

```
sampledata %>% filter(!is.na(Adherance)) %>%
 group_by(Adherance) %>%
  summarize(mean_weight = mean(Weight1, na.rm = T))
```

- alternatively, first remove missing values

```
sampledata %>%
  filter(!is.na(Weight1)) %>%
  group_by(Adherance) %>%
  summarize(mean_weight = mean(Weight1))%>%
  print(n = 15)
```

Summarize() function

- You can also group by multiple columns by adding categorical variables in **group_by()**.
- Once the data are grouped, you can also summarize multiple variables at the same time (and not necessarily on the same variable).

```
library(pander)
sampledata %>%
  filter(!is.na(Weight1)) %>%
  group_by(Adherance) %>%
  summarize(mean_weight = mean(Weight1),
            min_weight=min(Weight1),
            median_weight=median(Weight1),
            max_weight=max(Weight1),
            n=n())%>% pander() %>%
  head()
```

Counting

- When working with data, we often want to know the number of observations found for each factor or combination of factors. For this task, dplyr provides count().
- For example, if we wanted to count the number of rows of data for each sex category, we would do:

```
sampledata %>%
  filter(!is.na(Sex)) %>% count(Sex) %>% pander()
```

- The **count()** function is shorthand for something we've already seen: grouping by a variable, and summarizing it by counting the number of observations in that group.

Counting

- In other words, **sampledata %>% count()** is equivalent to:

```
sampledata %>%
    group_by(Sex, Adherance) %>%
    summarise(count = n())
```

- For convenience, **count()** provides the sort argument:

```
sampledata %>%
  filter(!is.na(Sex), !is.na(Adherance)) %>%
    count(Sex, Adherance, sort = TRUE)
```

Counting

- With the above code, we can proceed with arrange() to sort the table according to a number of criteria so that we have a better comparison.
- For instance, we might want to arrange the table above in

(i) an alphabetical order of the levels of the species and
(ii) in descending order of the count:

```
sampledata %>%
  filter(!is.na(Sex), !is.na(Adherance))%>%
  count(Sex, Adherance) %>% arrange(Sex, desc(n))
```

Reshaping data using tidyr package

- The tidyr package, provides four functions to change the layout of data set:
- gather(): gather (collapse) columns into rows
- spread(): spread rows into columns
- separate(): separate one column into multiple
- unite(): unite multiple columns into one

lets loading tidyr package

```
library("tidyr")# Loading
```

gather(): collapse columns into rows

- The function **gather()** collapses multiple columns into key-value pairs.
- It produces long data format from wide data.
- It's an alternative of **melt()** function [in reshape2 package].

1. Syntax:

```
gather(data, key, value, ...)
```

- **data**: A data frame
- **key, value**: Names of key and value columns to create in output
- **...**: Specification of columns to gather. Allowed values are:
  - variable names
  - if you want to select all variables between a and e, use a:e
  - if you want to exclude a column name y use -y

gather(): collapse columns into rows

2. Examples of usage:

```
HIV<-read.csv("HIV.csv")
HIV
```

```
library(kableExtra)
HivdataLong<- HIV %>% gather(key=Time, value=Cd4,
-c(Id,BF,Sex)) %>% arrange(Id)
```

spread(): spread two columns into multiple columns

- The function **spread()** does the reverse of **gather()**.
- It takes two columns (key and value) and spreads into multiple columns.
- It produces a "wide" data format from a "long" one.
- It's an alternative of the function **cast()** [in reshape2 package].

1. Syntax:

```
spread(data, key, value)
```

- `data`: A data frame
- `key`: Name of the column whose values will be used as column headings.
- `value`:Names of the column whose values will populate the cells.

spread(): spread two columns into multiple columns

2. **Examples of usage:**
   - Spread "Hivdata" to turn back to the original data:

```
 Widedata <- HivdataLong %>% spread(Time, Cd4) %>%
  arrange(Id)
View(Widedata)
```

unite(): Unite multiple columns into one

- The function unite() takes multiple columns and paste them together into one.

1. Syntax:

```
unite(data, col, ..., sep = "_")
```

- data: A data frame
- col: The new (unquoted) name of column to add.
- sep: Separator to use between values
- The R code below uses the data set "HivdataLong" and unites the columns Breast feed and Sex.

```
unite <- unite(HivdataLong, col = "BF_Sex",
                      BF, Sex,  sep = "_")
unite
```

separate(): separate one column into multiple

- The function **sperate()** is the reverse of **unite()**.
- It takes values inside a single character column and separates them into multiple columns.

1. Syntax:

```
separate(data, col, into, sep = " ")
```

- **data**: A data frame
- **col**: Unquoted column names
- **into**: Character vector specifying the names of new variables to be created.
- **sep**: Separator between columns

separate(): separate one column into multiple

2. **Examples of usage:**
   - Separate the column “BF_Sex” [in unite] into two columns BF and Sex:

```
separate(unite, col = "BF_Sex",
         into = c("Breast Feed", "Sex"), sep = "_")
```

Reshaping Data pivot function

We can also use the **pivot_()** function

- The **pivot_longer()** function reshapes data from wide format to long format
- The **pivot_wider()** function reshapes data from long format to wide format

```
long <-  HIV %>%  pivot_longer(cols = -c(Id,BF,Sex),
                  names_to = 'Time', values_to= 'Cd4')
 head(long)
```

Now, go backwards, and recreate the wide data!

```
wide <- long%>%pivot_wider(names_from=Time,values_from=Cd4)
head(wide)
```