Egocentric network analysis with R

Raffaele Vacca

2024-02-05

Contents

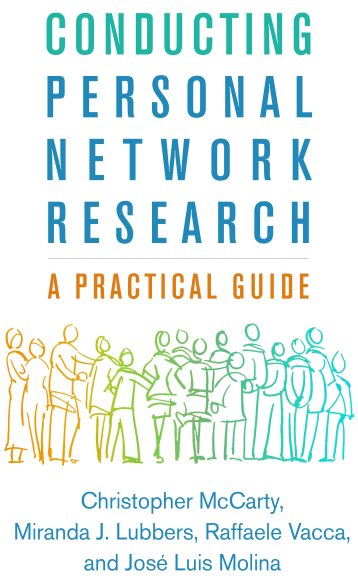
Chapter 1

Introduction

This book is a companion to my workshop on egocentric network analysis with R. Over the past several years I have taught this workshop at different conferences and summer schools on social network analysis, personal networks, and network science – such as INSNA Sunbelt conferences, NetSci conferences, and the UAB Barcelona Course on Personal Network Analysis. The book is a work in progress and I'll keep updating it as I continue to teach the workshop and related courses.

A Spanish translation of part of this book has appeared in the journal REDES - Revista hispana para el análisis de redes sociales. My colleagues and I provide a more in-depth discussion of personal network analysis and the methods covered in this workshop in our textbook on personal network research [McCarty et al., 2019]. See the bibliography at the end of the book for a list of other useful references on egocentric or personal network analysis.

Feel free to contact me to know more about this workshop and how to take it, or to give me feedback or report any issue about this book.



1.1 Workshop setup

To take this workshop you need to:

1. Download the last version of **R** here (select a location near you)
   - Follow instructions to install R in your computer
2. Download **RStudio** (free version) here
   - Follow instructions to install RStudio in your computer
3. Install the **R packages** listed below
   - Open RStudio and go to `Top menu > Tools > Install packages...`
   - Install each package in the list
4. Bring your **laptop** to the workshop
5. Download the **workshop folder** and save it to your computer: see below
   - I recommend that you do this in class at the beginning of the workshop so as to download the most updated version of the folder.
6. Once in class, go to the workshop folder on your computer (point 5 above) and double-click on the *R project* file in it (`.Rproj` extension).
   - That will open RStudio: you're all set!

NOTE: It's very important that you save the workshop folder *as downloaded* to a location in your computer, and open the `.Rproj` *within that folder*. By doing so, you will be opening RStudio *and* setting the workshop folder as your R working directory. All our R scripts assume that working directory. In particular, they assume that the `Data` subfolder is in your R working directory. You

can type `getwd()` in your R console to see the path to your R working directory and make sure that it's correctly pointed to the location of the workshop folder in your computer.

1.2 Workshop materials

The materials for this workshop consist of this book and the workshop folder.

You can **download the workshop folder** from this GitHub repository:

1. Click on the `Code` green button > Download ZIP
2. Unzip the folder and save it to your computer

The workshop folder contains several files and folders, but you only need to focus on the following:

- `Scripts` subfolder: all the R code shown in this book.
- `Data` subfolder: all the data we're going to use.
- `egocentric-r.Rproj`: the workshop's R project file (you use this to launch RStudio).

The `Scripts` subfolder includes different R script (.R) files. You can access and run the R code in each script by opening the corresponding .R file in RStudio. Each script in `Scripts` corresponds to one of the following chapters (see the table of contents):

2. Basics of the R language (`02_Basics` script and slideshow).
3. Representing and visualizing ego-networks in R (`03_Representing_egonets.R` script and slideshow).
4. Analyzing ego-network composition (`04_Composition` script and slideshow).
5. Analyzing ego-network structure (`05_Structure` script and slideshow).
6. Modeling tie- or alter-level variables with multilevel models (`06_Multilevel` script and slideshow).
7. Introduction to the `egor` package (`07_egor` script and slideshow).
8. Supplementary topics (`08_Supplementary` script and slideshow).

The **slides** used for this workshop can be downloaded here.

1.3 R settings

1.3.1 Required R packages

- For descriptive statistics:
  - `janitor`
  - `skimr`
- For network data, measures, visualization:
  - `egor`

  - `ggraph`
  - `igraph`
  - `intergraph`
  - `statnet`
  - `tidygraph`
- To fit multilevel statistical models and view their results:
  - `broom.mixed`
  - `car`
  - `ggeffects`
  - `lme4`
- General:
  - `tidyverse`. This is a collection of different packages that share a common language and set of principles, including `dplyr`, `ggplot2`, and `purrr`. See Wickham and Grolemund (2017) for more information.

1.3.2 RStudio options

RStudio gives you the ability to select and change various settings and features of its interface: see the `Preferences...` menu option. These are some of the settings you should pay attention to:

- `Preferences... > Code > Editing > Soft-wrap R source file`. Here you can decide whether or not to wrap long code lines in the editor. When code lines in a script are *not* wrapped, be aware that some code will be hidden if script lines are longer than your editor window's width (you'll have to scroll right to see the rest of the code). With a script (`.R`) file open in the editor, try both options (checked and unchecked) to see what you're more comfortable with.
- `Preferences... > Code > Display > Highlight R function calls`. This allows you to highlight all pieces of code that call an R function ("command"). I find function highlights very helpful to navigate a script and suggest that you check this option.

1.4 Data

This workshop uses real-world data collected in 2012 with a personal network survey among 107 Sri Lankan immigrants in Milan, Italy. Out of the 107 respondents, 102 reported their personal network. All data are in the `Data` subfolder.

The data files include ego-level data (gender, age, educational level, etc. for each Sri Lankan respondent), alter attributes (alter's nationality, country of residence, closeness to ego etc.), and information on alter-alter ties. Each personal network has a fixed size of 45 alters. Information about data variables and categories is available in `./Data/codebook.xlsx`.

All data objects are saved as R objects in the R data file `data.rda`. Data objects are the following:

- `ego.df`: A data frame with ego-level attributes for all respondents (egos).
- `alter.attr.all`: A data frame with alter-level attributes for all alters from all respondents.
- `gr.list`: A list. Each list element is one ego-network stored as an `igraph` object.
- `alter.attr.28`: A data frame with alter-level attributes only for alters nominated by ego ID 28.
- `gr.28`: The ego-network of ego ID 28 stored as an `igraph` object.
- `gr.ego.28`: The same as `gr.28`, but with the node of the ego included in the `igraph` object.

The R data objects above were imported from raw csv data files. All csv files are in the `./Data/raw_data/` subfolder:

- `ego_data.csv`: A csv file with ego-level data for all the egos.
- `alter_attributes.csv`: A single csv file including attributes of all alters from all egos.
- `alter_ties_028.csv`: The edge list for ego ID 28's egocentric network.
- `alter_attributes_028.csv`: The alter attributes in ego ID 28's egocentric network.
- `adj_028.csv`: The adjacency matrix for ego ID 28's egocentric network.
- `alter_ties.csv`: A single csv file with the edge list for all alters from all egos.

For most of the workshop we will directly use R data objects in `data.rda`. We will *not* focus on importing ego-network data from outside sources (for example, csv files). However, Section 7.1 shows you how the data objects in `data.rda` were created by importing csv files with the `egor` package. Section 8.2 covers importing ego-network data using just `tidyverse` and `igraph`.

1.5 Author and contacts

I am an assistant professor of sociology at the University of Milan in the Deparment of Social and Political Sciences. My main research and teaching interests are social networks, migration, health inequalities, and studies of science. I also teach and do research on data science, statistics, and computational methods for the social sciences. More information about me, my work and my contact details is here.

Chapter 2

R basics

2.1 Overview

This chapter covers some basic tools and characteristics of the R language that we'll use in the workshop. While this obviously can't be a comprehensive introduction to R, we'll demonstrate some essential R notions and features that are commonly used in social science data analysis, including (egocentric) social network analysis.

The script covers the following topics:

- Starting R, getting help with R.
- Creating and saving R objects.
- Vectors and matrices, data frames and tibbles.
- Arithmetic and relational operations.
- Subsetting vectors, matrices, and data frames.

2.2 Starting R and loading packages

- Before starting any work in R, you normally want to do two things:
  - Make sure your R session is pointing to the correct *working directory*.
  - Install and/or load the *packages* you are going to use.
- **Working directory**. By default, R will look for files and save new files in this directory.
  - Type `getwd()` in the console to view your current working directory.
  - If you opened RStudio by double-clicking on a project (`.Rproj`) file, then the working directory is the folder where that file is located.
  - You can always use `setwd()` to manually change your working directory to any path, but it's usually more convenient to work with R projects and their default working directory instead.

  - In RStudio, you can also check the current working directory by clicking on the `Files` panel.
- **R packages**. There are two steps to using a package in R:
  1. **Install** the package. *You do this just once.* Use `install.packages("package_name")` or the appropriate RStudio menu (`Tools > Install Packages...`). Once you install a package, the package files are in your system R folder and R will be able to always find the package there.
  2. **Load** the package in your current session. Use `library(package_name)` (*no* quotation marks around the package name). *You do this in each R session in which you need the package*, that is, every time you start R and you need the package.
- An R package is just a collection of **functions**. You can only use an R function if that function is included in a package you loaded in the current session.
- Sometimes two different functions from two different packages have the same **name**. For example, both the `igraph` package and the `sna` package have a function called `degree`. If both packages are loaded, typing just `degree` might give you unexpected results, because R will pick one of the two functions (the one in the package that was loaded most recently), which might not be the function you meant.
  - To avoid this problem, you can use the `package::function()` notation: `igraph::degree()` will always call the `degree` function from the `igraph` package, while `sna::degree()` will call the `degree` function from the `sna` package.
- Tip: To check the package that a function comes from, just go to that function's manual page. The package will be indicated in the first line of the page. E.g., type `?degree` to see where the `degree` function comes from.
  - If no currently loaded package has a function called `degree`, then typing `?degree` will cause a warning (`No documentation for 'degree'`).
  - If multiple, currently loaded packages have a function called `degree`, then typing `?degree` will bring up a page with the list of all those packages.
- This workshop will use a number of packages, listed here.

2.2.1 Console vs scripts

- When you open RStudio, you typically see two separate windows: the script editor and the console. You can write R code in either of them.
- **Console**. Here you write R code line by line. Once you type a line, you press `ENTER` to execute it. By pressing `ARROW UP` you go back to the last line you ran. By continuing to press `ARROW UP`, you can navigate through all the lines of code you previously executed. This is called the "commands history" (all the lines of code executed in the current session). You will lose all this code (all the history) when you quit R, unless you

explicitly save the history to a file (which is not what you typically do, you should just write the code in a script).

- **Script editor**. Here you write a script. This is the most common way of working with R. A script is simply a plain text file where all your R code is saved. If your work is in a script, it is **reproducible**.
- Both the R standard GUI and RStudio have a script editor with several helpful tools. Among other things, these allow you to run a script while you write it. By pressing `CTRL+ENTER` (Windows) or `CMD+ENTER` (Mac), you run the script line your cursor is on (or the selected script region).
  - Note that with RStudio you can run the single script line where your cursor is; a whole highlighted region of code; the region of code from the beginning of the script up to the line where your cursor is; the region of code from the line where your cursor is up to the end of the script. See the *Code* menu and its keyboard shortcuts.
- The script editor also allows you to save your script. In RStudio, see `File > Save` and its keyboard shortcut. R script files commonly have a `.R` extension (e.g. "`myscript.R`"). But note that a script file is just a text file (like any `.txt` file), which you can open and edit in any text editor, or in Microsoft Word and the likes.
- You can also run a whole script altogether — this is called **sourcing** a script. By running `source("myscript.R")`, you source the script file `myscript.R` (assuming the file is in your working directory, otherwise you'll have to enter the whole file path). In RStudio: see *Code* > *Source* and its keyboard shortcut.
- In both the console and the script editor, any line that starts by `#` is called a **comment**. R disregards comments — it just prints them as they are in the console (does not parse and execute them as programming code). Remember to always use comments to document what your code is doing (this is good for yourself and for others).
- In RStudio you can navigate the script headings in your script with a drop-down menu in the bottom-left of the script editor. Any line that starts by `#` and ends by `####`, `----`, or `====` is read as a heading by RStudio.

2.2.2 Getting help

- Getting help is one of the most common things you do when using R. As a beginner, you'll constantly need to get help (for example, read manual pages) about R functions. Also as an experienced user, you'll often need to go back to the manual pages of particular functions or other R help resources. At any experience level, using R involves constantly using its documentation and help resources.
- The following are a few help tools in R:
  - `help(...)` or `?...` are the most common ways of getting help: they send you to the R manual page for a specific function. E.g. `help(sum)` or `?sum` (they are equivalent).

  - `help.start()` (or RStudio: `Help > R Help`) gives you general help pages in html (introduction to R, references to all functions in all installed packages, etc.).
  - `demo()` gives you demos on specific topics. Run `demo()` to see all available topics.
  - `example()` gives you example code on specific functions, e.g. `example(sum)` for the function `sum`.

  - `help.search(...)` or `??...` search for a specific string in the manual pages, e.g. `??histogram`.
- In addition to built-in help facilities within R, there are plenty of ways to get **R help online**. Certain popular R packages have their own website, e.g. ggplot2, igraph, and statnet. Other websites for general R help include rdocumentation.org and stackoverflow.com. See the workshop slides or talk to me for more information.

```
# What's the current working directory?
# getwd()
# Un-comment to check your actual working directory.

# Change the working directory.
# setwd("/my/working/directory")
# (Delete the leading "#" and type in your actual working directory's path
# instead of "/my/working/directory")

# You should use R projects (.Rproj) to point to a working directory instead of
# manually changing it.

# Suppose that we want to use the package "igraph" in the following code.
library(igraph)

# Note that we can only load a package if we have it installed. In this case, I
# have igraph already installed. Had this not been the case, I would have have
# needed to install it:
# install.packages("igraph").
# (Packages can also be installed through an RStudio menu item).

# Let's load another suite of packages we'll use in the rest of this script.
library(tidyverse)

# Note that whatever is typed after the "#" sign has no effect but to be printed
# as is in the console: it is a comment.
```

2.3 Objects in R

> In R, everything that **exists** is an object. Everything that **happens** is a function call. - John Chambers

- R is an object-oriented programming language. Everything is contained in an **object**, including data, analysis tools and analysis results. Things such as datasets, commands (called "functions" in R), regression results, descriptive statistics, etc., are all objects.
- An object has a *name* and a *value*. You create an object by **assigning** a value to a name.
  - You assign with `<-` or with `=`.
  - R is **case-sensitive**: the object named `mydata` is different from the object named `Mydata`.
- Whenever you run an operation or execute a function in R, you need to assign the result to an object if you want to save it and re-use it later. **Assign it or lose it**: anything that is not assigned to an object is just printed to the console and lost.
- Objects have a size (bytes, megabytes, etc.) and a **type** (technically, a *class*, a *type* and a *mode* — more on this later).
- A **function** is a particular type of object. Functions take other objects as arguments (input) and return more objects as a result (output). R functions are what other data analysis programs call "commands". See Section 8.1.2 for more about functions.

2.3.1 The workspace

- During your R session, objects (data, results) are located in the computer's main memory. They make up your workspace: the set of **all the objects** currently in memory. They will disappear when you quit R, unless you save them to files on disk.
- What's in your current workspace?
  - The function `ls()` shows you a full list of the objects currently in the workspace.
  - Alternatively, in RStudio open the Environment panel to get a clickable list of objects currently in the workspace (if you don't see your Environment panel, check `Preferences... > Pane Layout`).

2.3.2 Saving and removing objects

- Two main functions to **save** R objects to files: `save()` (saves specific objects, its arguments); `save.image()` (saves all the current workspace).
- Unless you specify a different path, all files you save from R are put in your current working directory.
- The most common file extensions for files that store R objects are `.rda` and `.RData`.

- If you have a file with R objects, say `objects.rda`, you can **load** it in you current R session using the `load()` function: `load(file="objects.rda")`. This assumes `objects.rda` is in your current working directory (otherwise you'll have to specify the whole file path).
- The function `rm()` **removes** specific objects from the workspace. You can use it to clear the workspace from all existing objects by typing `rm(list=ls())` (remember that `ls()` returns a character vector with the names of all the objects in the current workspace).

```
# Create the object a: assign the value 50 to the name "a"
a <- 50

# Display ("print") the object a.
a
```

```
## [1] 50
```

```
# Let's create another object.
b <- "Mark"

# Display it.
b
```

```
## [1] "Mark"
```

```
# Create and display object at the same time.
(obj <- 10)
```

```
## [1] 10
```

```
# Let's reuse the object we created for a simple operation.
a + 3
```

```
## [1] 53
```

```
# What if we want to save this result?
result <- a + 3

# All objects in the workspace
ls()
```

```
## [1] "a"      "b"      "obj"    "result"
```

```
# Now we can view that result whenever we need it.
result
```

```
## [1] 53
```

```
# ...and further re-use it
result*2
```

```
## [1] 106
```

```
# Note that R is case-sensitive, "result" is different from "reSult".
reSult
```

```
## Error in eval(expr, envir, enclos): object 'reSult' not found
```

```
# Let's clear the workspace before proceeding.
rm(list=ls())

# The workspace is now empty.
ls()
```

```
## character(0)
```

2.3.3 Vector and matrix objects

- **Vectors** are the most basic objects you use in R. Vectors can be numeric (numerical data), logical (TRUE/FALSE data), or character (string data).
- The basic function to create a vector is `c` (**concatenate**).
- Other useful functions to create vectors: `rep` and `seq`.
  - Another function we'll use to create vectors later in the workshop is `seq_along`. `seq_along(x)` creates a vector consisting of a sequence of integers from 1 to `length(x)` in steps of 1.
  - Also keep in mind the `:` shortcut: `c(1, 2, 3, 4)` is the same as `1:4`.
- The **length** (number of elements) is a basic property of vectors: `length(x)` returns the length of vector `x`.
- When we `print` vectors, the numbers in square brackets indicate the positions of vector elements.
- To create a matrix: `matrix`. Its main arguments are the cell values (within `c()`), number of rows (`nrow`) and number of columns (`ncol`). Values are arranged in a `nrow` x `ncol` matrix *by column*. See `?matrix`.
- When we `print` matrices, the numbers in square brackets indicate the row and column numbers.

2.3.4 Data frames

- "Data frame" is R's name for dataset. A dataset is a collection of cases (rows), and variables (columns) which are measured on those cases.
- When `print`ed in R, data frames look like matrices. However, unlike matrix columns, data frame columns can be of different types, e.g. a numeric variable and a character variable.
- On the other hand, just like matrix columns, data frame columns (variables) must all have the same `length` (number of cases). You can't put together variables (vectors) of different length in the same data frame.
- Although data frames look like matrices, in R's mind they are a specific kind of *list* (more about lists in Section 5.3). In fact, the `class` of a data

frame is `data.frame`, but the `type` of a data frame is `list`. The list elements for a data frame are its variables (columns).

- **Tibbles.** The tidyverse packages, which we use in this workshop, rely on a more efficient form of data frame, called *tibble*.
  - A tibble has class `tbl_df` *and* `data.frame`. This means that, to R, a tibble is *also* a data frame, and any function that works on data frames normally also works on tibbles.
  - A tibble has a number of advantages over a traditional data frame, some of which we'll see in this workshop.
  - One of the advantages is the clearer and more informative way in which tibbles are printed. When we print a tibble data frame we can immediately see its number of rows, number of columns, names of variables, and type of each variable (numeric, integer, character, etc.).
  - To convert an existing data frame to tibble: `as_tibble`. To create a tibble from scratch (similar to the `data.frame` function in base R): `tibble`.
- While data frames can be created manually in R (with the functions `data.frame` in base R and `tibble` in `tidyverse`), data are most commonly imported into R from external sources, like a csv or txt file.
- We'll import data from csv files using the `read_csv()` function from `tidyverse`.
  - `read_csv()` reads csv files (values separated by "," or ";"). `read_delim()` reads files in which values are separated by any delimiter.
  - These functions have many arguments that make them very flexible and allow users to import basically any kind of table stored in a text file. Check out `?read_delim`.
  - In base R, the corresponding functions are `read.csv()` and `read.table()`.
- Data can also be imported into R from most external file formats (SAS, SPSS, Stata, Excel, etc.) using the tidyverse packages `readxl` and `haven`, or the `foreign` package in traditional R.
- Note that you can click on a data frame's name in RStudio's Environment pane. That will open the data frame in a window, similar to SPSS's data view.

```
# Let's create a simple vector.
(x <- c(1, 2, 3, 4))
```

```
## [1] 1 2 3 4
```

```
# Shortcut for the same thing.
(y <- 1:4)
```

```
## [1] 1 2 3 4
```

```
# What's the length of x?
length(x)
```

```
## [1] 4
```

```
# Note that when we print vectors, numbers in square brackets indicate positions
# of the vector elements.

# Create a simple matrix.
adj <- matrix(c(0,1,0, 1,0,0, 1,1,0), nrow= 3, ncol=3)

# This is what our matrix looks like:
adj
```

```
##      [,1] [,2] [,3]
## [1,]    0    1    1
## [2,]    1    0    1
## [3,]    0    0    0
```

```
# Notice the row and column numbers in square brackets.

# Normally we create data frames by importing data from external files, for
# example csv files.
ego.df <- read_csv("./Data/raw_data/ego_data.csv")
```

```
## Rows: 102 Columns: 9
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (4): ego.sex, ego.edu, ego.empl.bin, ego.age.cat
## dbl (5): ego_ID, ego.age, ego.arr, ego.inc, empl
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
# View the result.
ego.df
```

```
## # A tibble: 102 x 9
##    ego_ID ego.sex ego.age ego.arr ego.edu ego.inc  empl ego.empl.bin ego.age.cat
##     <dbl> <chr>     <dbl>   <dbl> <chr>     <dbl> <dbl> <chr>        <chr>
##  1     28 Male         61    2008 Second~     350     3 Yes          60+
##  2     29 Male         38    2000 Primary     900     4 Yes          36-40
##  3     33 Male         30    2010 Primary     200     3 Yes          26-30
##  4     35 Male         25    2009 Second~    1000     3 Yes          18-25
##  5     39 Male         29    2007 Primary       0     1 No           26-30
##  6     40 Male         56    2008 Second~     950     4 Yes          51-60
##  7     45 Male         52    1975 Primary    1600     3 Yes          51-60
##  8     46 Male         35    2002 Second~    1200     4 Yes          31-35
```

```
##  9     47 Male         22    2010 Second~     700     4 Yes          18-25
## 10     48 Male         51    2007 Primary     950     4 Yes          51-60
## # i 92 more rows
# Note the different pieces of information that are displayed when printing a tibble
# data frame.
```

2.4 Arithmetic, statistical, and relational operations

2.4.1 Arithmetic operations and recycling

- R can work as a normal calculator.
  - Addition/subtraction: `7+3`
  - Multiplication: `7*3`
  - Negative: `-7`
  - Division: `7/3`
  - Integer division: `7%/%3`
  - Integer remainder: `7%%3`
  - Exponentiation: `7^3`
- Many operations involving vectors in R are performed **element-wise**, i.e., separately on each element of the vector (see examples below).
- Most operations on vectors use the **recycling** rule: if a vector is too short, its values are re-used the number of times needed to match the desired length (see examples below).
- Examples of vector operations and recycling:
  - `[1 2 3 4] + [1 2 3 4] = [1+1 2+2 3+3 4+4]` (element-wise addition.)
  - `[1 2 3 4] + 1 = [1+1 2+1 3+1 4+1]` (`1` is recycled 3 times to match the `length` of the first vector.)
  - `[1 2 3 4] + [1 2] = [1+1 2+2 3+1 4+2]` (`[1 2]` is recycled once.)
  - `[1 2 3 4] + [1 2 3] = [1+1 2+2 3+3 4+1]` (`[1 2 3]` is recycled one third of a time: R will warn that the length of longer vector is not a multiple of the length of shorter vector.)

2.4.2 Relational operations and logical vectors

- **Relational** operators: `>`, `<`, `<=`, `>=`. Equal is `==` (NOT `=`). *Not* equal is `!=`.
  - Note: equal is `==`, whereas `=` has a different meaning. `=` is used to assign function arguments (e.g. `matrix(x, nrow = 3, ncol = 4)`), or to assign objects (`x <- 2` is the same as `x = 2`).
- Relational operations result in **logical** vectors: vectors of `TRUE`/`FALSE` values.

- Like arithmetic operations, relational ones are performed element-wise on vectors, and recycling applies.
- Logical operators: `&` for AND, `|` for OR.
- Negation (i.e. opposite) of a logical vector: `!`.
- Is value x in vector y? `x %in% y`.
- R can convert logical vectors to numeric (`as.numeric()`, `as.integer()`). In this conversion, `TRUE` becomes `1` and `FALSE` becomes `0`. Conversely, if converted to logical (`as.logical()`), `1/0` are `TRUE/FALSE`.
  - Therefore, if `x` is a logical vector, `sum(x)` gives the *count* of `TRUE`s in `x` (sum of `1`s in the vector).
  - `mean(x)` gives the *proportion* of `1`s in `x` (mean of a binary vector: sum of `1`s in the vector divided by number of elements in the vector).

2.4.3 Examples of arithmetic and statistical functions

- Arithmetic *vector* functions are performed element-wise, and return a vector. Examples:
  - Exponential: `exp(x)`.
  - Logarithm: `log(x)` (base e) or `log10(x)` (base 10).
- Statistical *scalar* functions are executed on the set of all vector elements taken together, and return a scalar. Examples:
  - `mean(x)` and `median(x)`.
  - Standard deviation and variance: `sd(x)`, `var(x)`.
  - Minimum and maximum: `min(x)`, `max(x)`.
  - `sum(x)`: sum of all elements in `x`.
- `table(x)` is another basic statistical function (but it's not scalar):
  - `table(x)` returns a `table` object with the absolute frequencies of values in `x`.
- Functions such as `sum()`, `mean()` and `table()` are very useful when programming in R (for example, when writing your own functions). However, if you just need descriptive statistics for the data, there are more convenient tools you can use. Some of these are the following functions, which work well with `tidyverse` (we'll see them in Ch. 4):
  - The `skim` function (from the `skimr` package) for descriptive statistics of continuous/quantitative variables.
  - The `tabyl` function (from the `janitor` package) for frequencies of categorical variables.

2.4.4 Missing and infinite values

- Missing values in R are represented by `NA` (Not Available).
  - If your data has a different code for missing values (e.g., -99), you'll have to recode that to NA for R to properly handle missing values in your data.
- Infinity may result from arithmetic operations: `Inf` and `-Inf` (e.g. `3/0`). `NaN` also may result, meaning Not a Number (e.g. `0/0`).

  - While `NA`s can appear in any type of object, `Inf`, `-Inf` and `NaN` can only appear in numeric objects.
- `is.na(x)` checks if each element of `x` is NA and returns TRUE if that's the case, FALSE otherwise. It's a vector function (its value has the same `length` as `x`).

```
# Just a few arithmetic operations between vectors to demonstrate element-wise
# calculations and the recycling rule.

(v1 <- 1:4)
```

```
## [1] 1 2 3 4
```

```
(v2 <- 1:4)
```

```
## [1] 1 2 3 4
```

```
# [1 2 3 4] + [1 2 3 4]
v1 + v2
```

```
## [1] 2 4 6 8
```

```
# [1 2 3 4] + 1
1:4 + 1
```

```
## [1] 2 3 4 5
```

```
# [1 2 3 4] + [1 2]
(v1 <- 1:4)
```

```
## [1] 1 2 3 4
```

```
(v2 <- 1:2)
```

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

```
v1 + v2
```

```
## [1] 2 4 4 6
```

```
# [1 2 3 4] + [1 2 3]
1:4 + 1:3
```

```
## Warning in 1:4 + 1:3: longer object length is not a multiple of shorter object
## length
```

```
## [1] 2 4 6 5
```

```
# Relational operations.

# Let's take a single variable from the ego attribute data: ego's age (in years)
# for the first 10 respondents. Let's put the result in a separate vector. This
# code involves indexing, we'll explain it better below.
```

```
age <- ego.df$ego.age[1:10]
age
```

```
##  [1] 61 38 30 25 29 56 52 35 22 51
```

```
# Note how the following comparisons are performed element-wise, and the value
# to which age is compared (30) is recycled.

# Is age equal to 30?
age==30
```

```
##  [1] FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

```
# The resulting logical vector is TRUE for those elements (i.e., respondents)
# who meet the condition.

# Which respondent's age is greater than 40?
age > 40
```

```
##  [1]  TRUE FALSE FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE  TRUE
```

```
# Which respondent age values are lower than 40 OR greater than 60?
age < 40 | age > 60
```

```
##  [1]  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE  TRUE  TRUE FALSE
```

```
# Which elements of "age" are lower than 40 AND greater than 30?
age < 40 & age > 30
```

```
##  [1] FALSE  TRUE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE
```

```
# Notice the difference between OR (|) and AND (&).

# Is 30 in "age"? I.e., is one of the respondents of 30 years of age?
30 %in% age
```

```
## [1] TRUE
```

```
# Is "age" in c(30, 35)? That is, which values of "age" are either 30 or 35?
age %in% c(30, 35)
```

```
##  [1] FALSE FALSE  TRUE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE
```

```
# A logical vector can be converted to numeric: TRUE becomes 1 and FALSE becomes
# 0.
age > 45
```

```
##  [1]  TRUE FALSE FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE  TRUE
```

```
as.numeric(age > 45)
```

```
##  [1] 1 0 0 0 0 1 1 0 0 1
```

```
# This allows us to use the sum() and the mean() functions to get the count and
# proportion of TRUE's in a logical vector.

# Count of TRUE's: Number of respondents (elements of "age") that are older than
# 40.
sum(age > 45)
```

```
## [1] 4
```

```
# How many respondents in the vector are older than 30?
sum(age > 30)
```

```
## [1] 6
```

```
# Proportion of TRUE's: What's the proportion of "age" elements (respondents)
# that are greater than 50?
mean(age > 50)
```

```
## [1] 0.4
```

```
# ***** EXERCISES
#
# (1) Obtain a logical vector indicating which elements of "age" are smaller than 30
# OR greater than 50. Then obtain a logical vector indicating which elements of
# "age" are smaller than 30 AND greater than 50. Why all elements are FALSE in
# the latter vector?
#
# (2) Using the : shortcut, create a vector that goes from 1 to 100 in steps of 1.
# Obtain a logical vector that is TRUE for the first 10 elements and the last 10
# elements of the vector.
#
# (3) Use the age vector with relational operators and sum/mean to answer these
# questions: How many respondents are younger than 50? What percentage of
# respondents is between 30 and 40 years of age, including 30 and 40? Is there
# any respondent who is younger than 20 OR older than 70 (use any())?
#
# (4) How many elements of the vector 1:100 are greater than the length of that
# vector divided by 2? Use sum().
#
# *****
```

2.5 Subsetting

- **Subsetting** is crucial in R. It means appending an index (or subscript) to an object to subset or extract one (or more) of its elements or components. This is also called “indexing” or “subscripting”.

- Subsetting can be used to **extract** (view, query) the component of an object, or to **replace** it (assign a different value to that element).
- The basic notation for subsetting in R is `[ ]`: `x[i]` gives you the *i*-th element of object `x`.
- **Numeric subsetting** uses integers in square brackets `[ ]`: e.g. `x[3]`. Note that you can use negative integers to index (select) everything *but* that element: e.g. `x[-3]`, `x[c(-2,-4)]`.
- **Logical subsetting** uses logical vectors in square brackets `[ ]`. It's used to subset objects based on a condition, e.g., to index all values in `x` that are greater than 3 (see example code below).
- **Name subsetting** uses element names. Elements in a vector, and rows or columns in a matrix can have names.
  - Names can be displayed and assigned using the `names` function in base R, or `set_names` (just to assign names) in tidyverse.
- When subsetting you must take into account the **number of dimensions** of an object. For example, vectors have one dimension, matrices have two. Arrays can be defined with three dimensions or more (e.g. three-way tables).
  - Square brackets typically contain a slot for each dimension of the object, separated by a comma:
    - `x[i]` indexes the *i*-th element of the one-dimensional object `x`;
    - `x[i,j]` indexes the *i,j*-th element of the two-dimensional object `x` (e.g. `x` is a matrix, *i* refers to a row and *j* refers to a column);
    - `x[i,j,k]` indexes the *i,j,k*-th element of the three-dimensional object `x`, etc.
  - Notice that a dimension's slot may be empty, meaning that we index all elements in that dimension. So, if `x` is a matrix, `x[3,]` will index the whole 3rd row of the matrix – i.e. [row 3, all columns].
  - If `x` has more than one dimension (e.g. it's a matrix), then `x[3]` (no comma, just one slot) is still valid, but it might give you unexpected results.
- Matrices have special functions that can be used for subsetting, e.g. `diagonal()`, `upper.tri()`, `lower.tri()`. These can be useful for manipulating adjacency matrices.
- Particular subsetting rules may apply to particular `class`es of objects, for example lists and data frames (see next Section 2.5.1).

2.5.1 Subsetting data frames

- **List notations.** Data frames are a special class of lists (see Section 5.3 for more about lists). Just like any list, data frames can be subset in the following three ways:
  1. `[ ]` notation, e.g. `df[3]` or `df["variable.name"]`. This returns another data frame that only includes the indexed element(s), e.g. only the 3rd element. Note:
     - This notation *preserves the* `data.frame` *class*: the result is still

a data frame.
  - This notation can be used to index *multiple* elements of a data frame into a new data frame, e.g. `df[c(1,3,5)]` or `df[c("sex", "age")]`
2. `[[ ]]` notation, e.g. `df[[3]]` or `df[["variable.name"]]`. This returns the specific element (column) selected, not as a data frame but as a vector with its own type and class, e.g. the numeric vector *within* the 3rd element of `df`. Note two differences from the `[ ]` notation:
  - `[[ ]]` *does not* preserve the `data.frame` class. The result is *not* a data frame.
  - Consistently, `[[ ]]` can only be used to index a *single* element (column) of the data frame, not multiple elements.
3. The `$` notation. If *variable.name* is the name of a specific variable (column) in `df`, then `df$variable.name` indexes that variable. This is the same as the `[[ ]]` notation: `df$variable.name` is the same as `df[["variable.name"]]`, and it's also the same as `df[[i]]` (where `i` is the position of the variable called *variable.name* in the data frame).

- **Matrix notation.** Data frames can also be subset like a matrix, with the `[ , ]` notation:
  - `df[2,3]`, `df[2, ]`, `df[ ,3]`.
  - `df[,"age"]`, `df[,c("sex", "age")]`, `df[5,"age"]`
- **Keep in mind the difference** between the following:
  - Extracting a data frame's variable (column) in itself, as a vector (numeric, character, etc.) — The single pepper packet by itself in the figure below (panel C). This is given by `df[[i]]`, `df[["variable.name"]]`, `df$variable.name`.
  - Extracting another data frame of just one variable (column) – The single pepper packet *within* the pepper shaker in the figure below (panel B). This is given by `df[i]`, `df["variable.name"]`.
- **Subsetting verbs** in tidyverse. In addition to subsetting data frames via the base indexing syntax described above, we can also use the subsetting functions introduced by the `dplyr` package in tidyverse (see below).

```
# Numeric subsetting ----
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

# Let's use our vector of ego age values again.
age
```

```
##  [1] 61 38 30 25 29 56 52 35 22 51
```

```
# Index its 2nd element: The age of the 2nd respondent.
age[2]
```

```
## [1] 38
```

```
# Its 2nd, 4th and 5th elements.
age[c(2,4,5)]
```

```
## [1] 38 25 29
```

```
# Fifth to seventh elements
age[5:7]
```

```
## [1] 29 56 52
```

```
# Use indexing to assign (replace) an element.
age[2] <- 45

# The content of x has now changed.
age
```

```
##  [1] 61 45 30 25 29 56 52 35 22 51
```

```
# Let's subset the adjacency matrix we created before.
adj
```

```
##      [,1] [,2] [,3]
## [1,]    0    1    1
## [2,]    1    0    1
## [3,]    0    0    0
```

```
# Its 2,3 cell: Edge from node 2 to node 3.
adj[2,3]
```

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

```
# Its 2nd column: All edges to node 2.
adj[,2]
```

```
## [1] 1 0 0
```

```
# Its 2nd and 3rd row: All edges from nodes 2 and 3.
adj[2:3,]
```

```
##      [,1] [,2] [,3]
## [1,]    1    0    1
## [2,]    0    0    0
```

```
# Logical subsetting                                                    ----
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

# Which values of "age" are between 40 and 60?

# Let's create a logical index that flags these values.
(ind <- age > 40 & age < 60)
```

```
##  [1] FALSE  TRUE FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE  TRUE
```

```
# Use this index to extract these values from vector "age" via logical subsetting.
age[ind]
```

```
## [1] 45 56 52 51
```

```
# We could also have typed directly:
age[age > 40 & age < 60]
```

```
## [1] 45 56 52 51
```

```
# Subsetting data frames                                                ----
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

# We'll use our ego-level data frame.
ego.df
```

```
## # A tibble: 102 x 9
##    ego_ID ego.sex ego.age ego.arr ego.edu ego.inc  empl ego.empl.bin ego.age.cat
##     <dbl> <chr>     <dbl>   <dbl> <chr>     <dbl> <dbl> <chr>        <chr>
##  1     28 Male         61    2008 Second~     350     3 Yes          60+
##  2     29 Male         38    2000 Primary     900     4 Yes          36-40
##  3     33 Male         30    2010 Primary     200     3 Yes          26-30
##  4     35 Male         25    2009 Second~    1000     3 Yes          18-25
##  5     39 Male         29    2007 Primary       0     1 No           26-30
##  6     40 Male         56    2008 Second~     950     4 Yes          51-60
##  7     45 Male         52    1975 Primary    1600     3 Yes          51-60
##  8     46 Male         35    2002 Second~    1200     4 Yes          31-35
##  9     47 Male         22    2010 Second~     700     4 Yes          18-25
## 10     48 Male         51    2007 Primary     950     4 Yes          51-60
## # i 92 more rows
```

```
# Numeric subsetting works on data frames too: it allows you to index variables.

# The 3rd variable.
ego.df[3]
```

```
## # A tibble: 102 x 1
##    ego.age
##      <dbl>
##  1      61
##  2      38
##  3      30
##  4      25
##  5      29
##  6      56
##  7      52
##  8      35
##  9      22
## 10      51
## # i 92 more rows
```

```
# Note the difference with the double square bracket.
ego.df[[3]]
```

```
##   [1] 61 38 30 25 29 56 52 35 22 51 50 45 51 32 57 42 32 55 44 25 28 60 54 30 37
##  [26] 41 47 33 37 29 39 31 52 30 57 44 47 42 44 31 31 56 36 47 47 35 43 22 55 32
##  [51] NA 60 51 53 25 33 38 37 27 34 32 26 36 53 42 45 36 54 49 35 27 32 58 51 55
##  [76] 59 27 35 24 52 27 42 47 43 53 54 35 50 51 40 49 33 33 36 32 35 55 61 34 44
## [101] 50 28
```

```
# What do you think is the difference?
class(ego.df[3])
```

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

```
class(ego.df[[3]])
```

```
## [1] "numeric"
```

```
# The [[ ]] notation extracts the actual column as a vector, while [ ] keeps
# the data frame class.

# We can also subset data frames as matrices.
# The second and third columns.
ego.df[,2:3]
```

```
## # A tibble: 102 x 2
##    ego.sex ego.age
##    <chr>     <dbl>
##  1 Male         61
##  2 Male         38
##  3 Male         30
##  4 Male         25
##  5 Male         29
##  6 Male         56
```

```
##  7 Male         52
##  8 Male         35
##  9 Male         22
## 10 Male         51
## # i 92 more rows
```

```
# Lines 1 to 3
ego.df[1:3,]
```

```
## # A tibble: 3 x 9
##   ego_ID ego.sex ego.age ego.arr ego.edu  ego.inc  empl ego.empl.bin ego.age.cat
##    <dbl> <chr>     <dbl>   <dbl> <chr>      <dbl> <dbl> <chr>         <chr>
## 1     28 Male         61    2008 Seconda~     350     3 Yes           60+
## 2     29 Male         38    2000 Primary      900     4 Yes           36-40
## 3     33 Male         30    2010 Primary      200     3 Yes           26-30
```

```
# We can use name indexing with data frames, selecting variables by name
ego.df["ego.age"]
```

```
## # A tibble: 102 x 1
##    ego.age
##      <dbl>
##  1      61
##  2      38
##  3      30
##  4      25
##  5      29
##  6      56
##  7      52
##  8      35
##  9      22
## 10      51
## # i 92 more rows
```

```
ego.df[["ego.age"]]
```

```
##   [1] 61 38 30 25 29 56 52 35 22 51 50 45 51 32 57 42 32 55 44 25 28 60 54 30 37
##  [26] 41 47 33 37 29 39 31 52 30 57 44 47 42 44 31 31 56 36 47 47 35 43 22 55 32
##  [51] NA 60 51 53 25 33 38 37 27 34 32 26 36 53 42 45 36 54 49 35 27 32 58 51 55
##  [76] 59 27 35 24 52 27 42 47 43 53 54 35 50 51 40 49 33 33 36 32 35 55 61 34 44
## [101] 50 28
```

```
# The $ notation is very common and concise. It's equivalent to the [[ notation.
ego.df$ego.age
```

```
##   [1] 61 38 30 25 29 56 52 35 22 51 50 45 51 32 57 42 32 55 44 25 28 60 54 30 37
##  [26] 41 47 33 37 29 39 31 52 30 57 44 47 42 44 31 31 56 36 47 47 35 43 22 55 32
##  [51] NA 60 51 53 25 33 38 37 27 34 32 26 36 53 42 45 36 54 49 35 27 32 58 51 55
##  [76] 59 27 35 24 52 27 42 47 43 53 54 35 50 51 40 49 33 33 36 32 35 55 61 34 44
```

```
## [101] 50 28
```

```
# This is the same as ego.df[[3]] or ego.df[["ego.age"]]
identical(ego.df[[3]], ego.df$ego.age)
```

```
## [1] TRUE
```

```
# With tidyverse, this type of subsetting syntax is replaced by new "verbs"
# (see section below):
# * Index data frame rows: filter() instead of []
# * Index data frame columns: select() instead of []
# * Extract data frame variable as a vector: pull() instead of [[]] or $


# ***** EXERCISES
#
# (1) What's the mean age of respondents whose education is NOT "Primary"? Use
# the ego.age and ego.edu variables. Remember that the "different from"
# comparison operator is !=.
#
# (2) What is the modal education level (i.e., the most common education level)
# of respondents who are older than 40? Use table() or tabyl().
#
# (3) Create the fictitious variable var <- c(1:30, rep(NA, 3), 34:50). Use is.na()
# to index all the NA values in the variable. Then use is.na() to index all
# values that are *not* NA. Hint: Remember the operator used to negate a logical
# vector. Finally, use this indexing to remove all NA values from var.
#
# (4) Use the $ notation to extract the "ego.arr" variable in ego.df. Recode all
# values equal to 2008 as 99. Hint: Index all values equal to 2008, then replace
# them with 99 via the assignment operator.
#
# *****
```

2.6 The `tidyverse` syntax

2.6.1 Pipes and the `|>` operator

- The original pipe operator, `%>%`, was introduced by the `magrittr` package in 2014. It quickly gained popularity in the R community and was adopted by `igraph` and `tidyverse` (among other packages), which we use in this workshop. In 2021, R incorporated the pipe idea with a new, similar (but not identical) operator: `|>`. See this page for an overview of the differences between `|>` and `%>%`.
- The idea behind pipes is in essence very simple:
  - `f(g(x))` becomes `x |> g() |> f()`.

  - For example: `mean(table(x))` becomes `x |> table() |> mean()`.
- So `|>` pipes the output of the previous function (e.g., `table()`) into the input of the following function (e.g., `mean()`). This turns inside-to-outside code into left-to-right code. Because left to right is the direction most of us are used to read in (at least in English and other Western languages), pipes make R code easier to read and follow.
- You may also see pipes concatenating multiple lines of code. That's possible and a common coding style. Instead of

```
x |> table() |> mean()
```

you can write

```
x |>
  table() |>
  mean()
```

2.6.2 Subsetting data frames in `tidyverse`

- Tidyverse includes the `dplyr` package for data frame manipulation. This is a very powerful package for all kinds of data wrangling. To learn more, see the package cheatsheet and vignettes.
- Subsetting data frames with `dplyr`:
  - `dplyr::filter()` is used to subset rows (cases) of a data frame based on one or multiple conditions (for example, respondents with certain values on one or more variables). This preserves the data frame class, similar to `[ ]` indexing.
  - `dplyr::select()` is used to subset columns (variables) of a data frame. You can use full variable names or select variables in many other ways (see examples in the `select` manual page). This preserves the data frame class, similar to `[ ]` indexing.
  - `dplyr::pull()` is used to extract a column as a vector. This does *not* preserve the data frame class, similar to `[[ ]]` or `$`.

```
# The pipe operator                                                    ----
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

# Ego education variable
(edu <- ego.df$ego.edu)
```

```
##   [1] "Secondary"  "Primary"    "Primary"    "Secondary"  "Primary"
##   [6] "Secondary"  "Primary"    "Secondary"  "Secondary"  "Primary"
##  [11] "Secondary"  "Secondary"  "University" "Primary"    "Secondary"
##  [16] "Primary"    "Primary"    "Secondary"  "Secondary"  "Primary"
##  [21] "University" "University" "Primary"    "Secondary"  "Primary"
##  [26] "Primary"    "Secondary"  "Primary"    "Primary"    "Primary"
##  [31] "Secondary"  "Secondary"  "Secondary"  "Primary"    "Primary"
##  [36] "Secondary"  "Secondary"  "Primary"    "Secondary"  "Primary"
```

```
##  [41] "Primary"    "Primary"    "Primary"    "Secondary"  "Primary"
##  [46] "Primary"    "Primary"    "University" "Primary"    "Secondary"
##  [51] "University" "Primary"    "University" "Secondary"  "Secondary"
##  [56] "University" "Primary"    "Primary"    "Secondary"  "Secondary"
##  [61] "Secondary"  "University" "University" "University" "Secondary"
##  [66] "Secondary"  "Secondary"  "University" "Primary"    "Secondary"
##  [71] "Secondary"  "Secondary"  "University" "Primary"    "Secondary"
##  [76] "Secondary"  "Secondary"  "Secondary"  "Secondary"  "Secondary"
##  [81] "Secondary"  "Primary"    "Primary"    "University" "Primary"
##  [86] "Primary"    "Secondary"  "Secondary"  "Secondary"  "University"
##  [91] "Secondary"  "Secondary"  "Primary"    "Primary"    "University"
##  [96] "Primary"    "Primary"    "Primary"    "Primary"    "Secondary"
## [101] "Primary"    "Secondary"
```

```
# Frequency of educational categories in the data
table(edu)
```

```
## edu
##    Primary  Secondary University
##         42         45         15
```

```
# Average frequency
mean(table(edu))
```

```
## [1] 34
```

```
# Let's re-write this with the pipe operator
edu |>
  table() |>
  mean()
```

```
## [1] 34
```

```
# Subsetting data frames with dplyr: filter and select                    ----
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

# Filter to egos who are older than 40
ego.df |>
  filter(ego.age > 40)
```

```
## # A tibble: 51 x 9
##    ego_ID ego.sex ego.age ego.arr ego.edu ego.inc  empl ego.empl.bin ego.age.cat
##     <dbl> <chr>     <dbl>   <dbl> <chr>     <dbl> <dbl> <chr>        <chr>
##  1     28 Male         61    2008 Second~     350     3 Yes          60+
##  2     40 Male         56    2008 Second~     950     4 Yes          51-60
##  3     45 Male         52    1975 Primary    1600     3 Yes          51-60
##  4     48 Male         51    2007 Primary     950     4 Yes          51-60
##  5     49 Male         50    2001 Second~    1300     4 Yes          41-50
##  6     51 Male         45    2011 Second~     480     3 Yes          41-50
```

```
##  7     52 Male         51    2002 Univer~    1200     4 Yes          51-60
##  8     55 Male         57    1979 Second~     470     7 No           51-60
##  9     56 Male         42    1992 Primary    1100     4 Yes          41-50
## 10     58 Male         55    1990 Second~    1450     4 Yes          51-60
## # i 41 more rows
```

```
# Filter to egos with Primary education
ego.df |>
  filter(ego.edu == "Primary")
```

```
## # A tibble: 42 x 9
##    ego_ID ego.sex ego.age ego.arr ego.edu ego.inc  empl ego.empl.bin ego.age.cat
##     <dbl> <chr>     <dbl>   <dbl> <chr>     <dbl> <dbl> <chr>        <chr>
##  1     29 Male         38    2000 Primary     900     4 Yes          36-40
##  2     33 Male         30    2010 Primary     200     3 Yes          26-30
##  3     39 Male         29    2007 Primary       0     1 No           26-30
##  4     45 Male         52    1975 Primary    1600     3 Yes          51-60
##  5     48 Male         51    2007 Primary     950     4 Yes          51-60
##  6     53 Male         32    2003 Primary    1600     1 No           31-35
##  7     56 Male         42    1992 Primary    1100     4 Yes          41-50
##  8     57 Male         32    2000 Primary    1200     4 Yes          31-35
##  9     60 Male         25    2011 Primary       0     1 No           18-25
## 10     64 Male         54    1981 Primary     300     3 Yes          51-60
## # i 32 more rows
```

```
# Combine the two conditions: Intersection.
ego.df |>
  filter(ego.age > 40 & ego.edu == "Primary")
```

```
## # A tibble: 21 x 9
##    ego_ID ego.sex ego.age ego.arr ego.edu ego.inc  empl ego.empl.bin ego.age.cat
##     <dbl> <chr>     <dbl>   <dbl> <chr>     <dbl> <dbl> <chr>        <chr>
##  1     45 Male         52    1975 Primary    1600     3 Yes          51-60
##  2     48 Male         51    2007 Primary     950     4 Yes          51-60
##  3     56 Male         42    1992 Primary    1100     4 Yes          41-50
##  4     64 Male         54    1981 Primary     300     3 Yes          51-60
##  5     68 Male         41    2008 Primary     600     4 Yes          41-50
##  6     82 Male         57    1982 Primary    1190     4 Yes          51-60
##  7     85 Male         42    2008 Primary     400     2 No           41-50
##  8     90 Male         56    2004 Primary     450     4 Yes          51-60
##  9     93 Male         47    2010 Primary     880     4 Yes          41-50
## 10     95 Male         43    1997 Primary     800     4 Yes          41-50
## # i 11 more rows
```

```
# Combine the two conditions: Union.
ego.df |>
  filter(ego.age > 40 | ego.edu == "Primary")
```

```
## # A tibble: 72 x 9
##    ego_ID ego.sex ego.age ego.arr ego.edu ego.inc  empl ego.empl.bin ego.age.cat
##     <dbl> <chr>     <dbl>   <dbl> <chr>     <dbl> <dbl> <chr>        <chr>
##  1     28 Male         61    2008 Second~     350     3 Yes          60+
##  2     29 Male         38    2000 Primary     900     4 Yes          36-40
##  3     33 Male         30    2010 Primary     200     3 Yes          26-30
##  4     39 Male         29    2007 Primary       0     1 No           26-30
##  5     40 Male         56    2008 Second~     950     4 Yes          51-60
##  6     45 Male         52    1975 Primary    1600     3 Yes          51-60
##  7     48 Male         51    2007 Primary     950     4 Yes          51-60
##  8     49 Male         50    2001 Second~    1300     4 Yes          41-50
##  9     51 Male         45    2011 Second~     480     3 Yes          41-50
## 10     52 Male         51    2002 Univer~    1200     4 Yes          51-60
## # i 62 more rows
```

```
# Note that the object ego.df hasn't changed. To re-use any of the filtered data
# frames above, we have to assign them to an object.
ego.df.40 <- ego.df |>
  filter(ego.age > 40)

ego.df.40
```

```
## # A tibble: 51 x 9
##    ego_ID ego.sex ego.age ego.arr ego.edu ego.inc  empl ego.empl.bin ego.age.cat
##     <dbl> <chr>     <dbl>   <dbl> <chr>     <dbl> <dbl> <chr>        <chr>
##  1     28 Male         61    2008 Second~     350     3 Yes          60+
##  2     40 Male         56    2008 Second~     950     4 Yes          51-60
##  3     45 Male         52    1975 Primary    1600     3 Yes          51-60
##  4     48 Male         51    2007 Primary     950     4 Yes          51-60
##  5     49 Male         50    2001 Second~    1300     4 Yes          41-50
##  6     51 Male         45    2011 Second~     480     3 Yes          41-50
##  7     52 Male         51    2002 Univer~    1200     4 Yes          51-60
##  8     55 Male         57    1979 Second~     470     7 No           51-60
##  9     56 Male         42    1992 Primary    1100     4 Yes          41-50
## 10     58 Male         55    1990 Second~    1450     4 Yes          51-60
## # i 41 more rows
```

```
# Select specific variables
ego.df.40 |>
  dplyr::select(ego.sex, ego.age)
```

```
## # A tibble: 51 x 2
##    ego.sex ego.age
##    <chr>     <dbl>
##  1 Male         61
##  2 Male         56
##  3 Male         52
```

```
##  4 Male         51
##  5 Male         50
##  6 Male         45
##  7 Male         51
##  8 Male         57
##  9 Male         42
## 10 Male         55
## # i 41 more rows
```

```
# As usual, we can re-assign the result to the same data object
ego.df.40 <- ego.df.40 |>
  dplyr::select(ego.sex, ego.age)

ego.df.40
```

```
## # A tibble: 51 x 2
##    ego.sex ego.age
##    <chr>     <dbl>
##  1 Male         61
##  2 Male         56
##  3 Male         52
##  4 Male         51
##  5 Male         50
##  6 Male         45
##  7 Male         51
##  8 Male         57
##  9 Male         42
## 10 Male         55
## # i 41 more rows
```

```
# Pull a variable out of a data frame, as a vector
ego.df |>
  pull(ego.age)
```

```
##   [1] 61 38 30 25 29 56 52 35 22 51 50 45 51 32 57 42 32 55 44 25 28 60 54 30 37
##  [26] 41 47 33 37 29 39 31 52 30 57 44 47 42 44 31 31 56 36 47 47 35 43 22 55 32
##  [51] NA 60 51 53 25 33 38 37 27 34 32 26 36 53 42 45 36 54 49 35 27 32 58 51 55
##  [76] 59 27 35 24 52 27 42 47 43 53 54 35 50 51 40 49 33 33 36 32 35 55 61 34 44
## [101] 50 28
```

```
# This is the same as
ego.df$ego.age
```

```
##   [1] 61 38 30 25 29 56 52 35 22 51 50 45 51 32 57 42 32 55 44 25 28 60 54 30 37
##  [26] 41 47 33 37 29 39 31 52 30 57 44 47 42 44 31 31 56 36 47 47 35 43 22 55 32
##  [51] NA 60 51 53 25 33 38 37 27 34 32 26 36 53 42 45 36 54 49 35 27 32 58 51 55
##  [76] 59 27 35 24 52 27 42 47 43 53 54 35 50 51 40 49 33 33 36 32 35 55 61 34 44
## [101] 50 28
```

```
# ***** EXERCISES
#
# (1) Use the [ , ] notation to extract the education level and income of egos who
# are younger than 30. Then do the same thing with the dplyr::filter() function.
#
# *****
```

Chapter 3

Representing and visualizing ego-networks

3.1 Overview

After learning some basics of the R language, we now focus on R tools for egocentric network data. To make things simpler, we'll start by considering just one egocentric network. The following chapters will show you how to replicate the same type of analysis on many ego-networks at once.

This chapter covers the following topics:

- Representing ego-level attribute data and alter-level attribute data in R.
- Joining (merging) ego-level and alter-level data frames.
- Representing alter-alter tie data.
- Representing and manipulating an ego-network as an `igraph` object.
- Visualizing an ego-network.

3.2 Ego-level and alter-level data

- In this section we use the data objects described earlier in Section 1.4. All these objects are stored in the `data.rda` file. In sections 7.1 and 8.2 we'll show how to create these objects from raw (csv) data using `egor` and `tidyverse`.
- Egocentric network data typically include at least three types of data:
  - **Ego-level attribute data**. This is a dataset with attributes of egos. Each row is an ego (typically, a survey respondent) and each column is a characteristic of the ego. Following multilevel modeling

terminology, we call this "level 2" because, in the multilevel structure of ego-network data, egos are the higher-level "groups" in which alters are clustered. In personal network surveys, these data are obtained from standard survey questions asking individual information about the respondent.

  - **Alter-level attribute data**. This is a dataset with attributes of alters and of ego-alter ties. It is also the dataset that, for each alter, indicates who is the ego who nominated that alter: that is, it lists all the ego-alter ties. In this dataset, each row is an alter and each column is a characteristic of the alter, or of the relationship between the alter and the ego who nominated him/her. Following multilevel modeling terminology, we call this "level 1" because alters are the most granular units in the data, clustered within egos. In personal network surveys, these data are obtained from the so-called *name generator* questions, which elicit lists of alters from each ego; and from the *name interpreter* questions, which elicit characteristics of each alter.
  - **Alter-alter tie data**. These are the data about alter-alter ties as reported by the ego. In personal network surveys, these data are obtained from so-called *edge interpreter* questions.

- **File formats:**
  - Ego-level attribute data. This is normally a single dataset (in a single file), with each row representing one ego, and different columns representing different ego-level attributes. This is similar to any respondent-level dataset you have in standard survey data.
  - Alter-level attribute data. Depending on the data collection software, you might have a single dataset (in a single file) including alters nominated by all egos; or separate datasets (in separate files), one for each ego, with the alters nominated by that ego (and with the same variables in all datasets).
  - Alter-alter tie data. These are typically in edge list format or in adjacency matrix format, depending on the data collection software. If the tie data are in adjacency matrices, you normally have a separate adjacency matrix for each ego, in a separate file. If they are in edge lists, you might have a single edge list including alters from all egos (with an additional column indicating the ego that nominated the two alters), or a separate edge list (in a separate file) for each ego.
- The data format described above works well because, in standard egocentric data, the different ego-networks are **separate** (non-overlapping). This means that an alter can only "belong" to one and one only ego, and alter-alter ties only exist between alters nominated by the same egos. In other words, there are no links or overlaps between the networks of different egos.
  - In particular, this means that there is a one-to-one correspondence between alters and ego-alter ties: for each ego-alter tie there is only one alter, and vice versa (so the level of alters is the same as the level

of ego-alter ties).

- In egocentric analysis you are frequently switching between level 1 and 2, and **joining** information from the two levels.
- **Level-1 join.** Bring information from level 2 into the level-1 dataset.
  - For certain types of analysis you need to join ego attributes (level 2) into an alter-level data set (level 1).
  - For example, you need to do this when estimating multilevel models in which the dependent variable is a characteristic of ego-alter ties (e.g., if the tie provides support), and some of the predictors are ego-level characteristics (e.g., gender of the ego).
  - Because there is one ego for multiple alters, this is a one-to-many join, with the same value of an ego attribute (one ego row in the level-2 dataset) being joined to multiple alters (multiple alter rows in the level-1 dataset).
  - The data frames are joined by ego ID.
- **Level-2 join.** Bring information from level 1 into the level-2 dataset.
  - In other cases you want to join alter attributes (level 1) into an ego-level dataset (level 2).
  - For example, you need to do this when you want to analyze summary measures of ego-network characteristics (e.g. average age of alters) among the egos.
  - Because there are multiple alters for one ego, this requires that you first summarize or aggregate the alter attributes for each ego.
  - If you have *continuous* alter attributes, you typically summarize them by taking averages and dispersion measures (variance, standard deviation, etc.) of the alter attribute for each ego: for example, average alter age for each ego, or standard deviation of contact frequency between alters and each ego.
  - If you have *categorical* alter attributes, you normally summarize them by taking counts or proportions of certain categories, or qualitative diversity measures (e.g., generalized variance, entropy): for example, proportion of women, or ethnic diversity in each ego's network.
  - Once you have these summary variables on ego-network *composition*, you can join them with an ego-level dataset by ego ID.
  - The level-2 join can also involve summary variables on ego-network *structure*: for example, average alter degree, ego-network density, or number of components. These variables can be joined with an ego-level dataset by ego ID.
  - We'll consider this type of ego-level summarization and the level-2 join in Chapters 4 and 5 about measures of ego-network composition and structure.
- In base R, data frames can be joined using the `merge` function. However, we use the `dplyr` join functions, whose code is more efficient and readable:
  - `left_join()` retains all rows in the *left* data frame, and discards any row in the right data frame that does not have a match with a row in the left data frame.

  - `right_join()` retains all rows in the *right* data frame, and discards any row in the left data frame that does not have a match with a row in the right data frame.
  - `full_join()` retains all rows from both data frames.
- McCarty and colleagues (2019) provide a more detailed discussion of all types of egocentric network data, levels in egocentric data, and switching and joining operations between levels.
- **What we do in the following code**.
  - View ego-level data (level 2) and alter-level data (level 1) as stored in R data frames.
  - Do the level-1 join: Bring ego attributes into an alter-level data frame.

```
# Load packages.
library(tidyverse)

# Load data.
load("./Data/data.rda")

# The file we just loaded includes the following objects.

# * Ego-level attribute data
ego.df
```

```
## # A tibble: 102 x 9
##    ego_ID ego.sex ego.age ego.arr ego.edu ego.inc  empl ego.empl.bin ego.age.cat
##     <dbl> <fct>     <dbl>   <dbl> <fct>     <dbl> <dbl> <fct>        <fct>
##  1     28 Male         61    2008 Second~     350     3 Yes          60+
##  2     29 Male         38    2000 Primary     900     4 Yes          36-40
##  3     33 Male         30    2010 Primary     200     3 Yes          26-30
##  4     35 Male         25    2009 Second~    1000     3 Yes          18-25
##  5     39 Male         29    2007 Primary       0     1 No           26-30
##  6     40 Male         56    2008 Second~     950     4 Yes          51-60
##  7     45 Male         52    1975 Primary    1600     3 Yes          51-60
##  8     46 Male         35    2002 Second~    1200     4 Yes          31-35
##  9     47 Male         22    2010 Second~     700     4 Yes          18-25
## 10     48 Male         51    2007 Primary     950     4 Yes          51-60
## # i 92 more rows
```

```
# * Attribute data for alters and ego-alter ties (all alters and ego-alter ties
# in one data frame).
alter.attr.all
```

```
## # A tibble: 4,590 x 12
##    alter_ID ego_ID alter_num alter.sex alter.age.cat alter.rel    alter.nat
##       <dbl>  <dbl>     <dbl> <fct>     <fct>         <fct>        <fct>
##  1     2801     28         1 Female    51-60         Close family Sri Lanka
```

```
##  2     2802     28         2 Male      51-60         Other family Sri Lanka
##  3     2803     28         3 Male      51-60         Close family Sri Lanka
##  4     2804     28         4 Male      60+           Close family Sri Lanka
##  5     2805     28         5 Female    41-50         Close family Sri Lanka
##  6     2806     28         6 Female    60+           Close family Sri Lanka
##  7     2807     28         7 Male      41-50         Other family Sri Lanka
##  8     2808     28         8 Female    36-40         Other family Sri Lanka
##  9     2809     28         9 Female    51-60         Other family Sri Lanka
## 10     2810     28        10 Male      60+           Other family Sri Lanka
## # i 4,580 more rows
## # i 5 more variables: alter.res <fct>, alter.clo <dbl>, alter.loan <fct>,
## #   alter.fam <fct>, alter.age <dbl>
```

```
# Level-1 join: Ego attributes into alter-level data (one to many).
(data <- left_join(alter.attr.all, ego.df, by= "ego_ID"))
```

```
## # A tibble: 4,590 x 20
##    alter_ID ego_ID alter_num alter.sex alter.age.cat alter.rel    alter.nat
##       <dbl>  <dbl>     <dbl> <fct>     <fct>         <fct>        <fct>
##  1     2801     28         1 Female    51-60         Close family Sri Lanka
##  2     2802     28         2 Male      51-60         Other family Sri Lanka
##  3     2803     28         3 Male      51-60         Close family Sri Lanka
##  4     2804     28         4 Male      60+           Close family Sri Lanka
##  5     2805     28         5 Female    41-50         Close family Sri Lanka
##  6     2806     28         6 Female    60+           Close family Sri Lanka
##  7     2807     28         7 Male      41-50         Other family Sri Lanka
##  8     2808     28         8 Female    36-40         Other family Sri Lanka
##  9     2809     28         9 Female    51-60         Other family Sri Lanka
## 10     2810     28        10 Male      60+           Other family Sri Lanka
## # i 4,580 more rows
## # i 13 more variables: alter.res <fct>, alter.clo <dbl>, alter.loan <fct>,
## #   alter.fam <fct>, alter.age <dbl>, ego.sex <fct>, ego.age <dbl>,
## #   ego.arr <dbl>, ego.edu <fct>, ego.inc <dbl>, empl <dbl>,
## #   ego.empl.bin <fct>, ego.age.cat <fct>
```

```
# Note that the new data frame has one row for each alter, and the same ego ID
# and ego attribute value is repeated for all alters belonging to the same ego.
data |>
  dplyr::select(alter_ID, ego_ID, ego.age, ego.edu)
```

```
## # A tibble: 4,590 x 4
##    alter_ID ego_ID ego.age ego.edu
##       <dbl>  <dbl>   <dbl> <fct>
##  1     2801     28      61 Secondary
##  2     2802     28      61 Secondary
##  3     2803     28      61 Secondary
##  4     2804     28      61 Secondary
```

```
##  5     2805     28      61 Secondary
##  6     2806     28      61 Secondary
##  7     2807     28      61 Secondary
##  8     2808     28      61 Secondary
##  9     2809     28      61 Secondary
## 10     2810     28      61 Secondary
## # i 4,580 more rows
```

3.3 Networks in R

- There are several packages for network analysis in R. The two main (collections of) R network packages are `igraph` and `statnet`. `igraph` is a single package. It represents networks as objects of class `igraph`. `statnet` is not a package, it's a collection of packages. It represents networks as objects of class `network`.
- `igraph` and `statnet` are in part overlapping, in part complementary. Historically, `igraph` has focused more on network methods developed in computer science and physics, while `statnet` has emphasized network methods developed in statistics and the social sciences. Many things can be done in *both* `igraph` and `statnet`: basic network creation, importing network data, network manipulation, basic network metrics such as centrality, and network visualization. On the other hand, a few things can only be done in `igraph` (e.g., "community detection" algorithms and modularity analysis), and others can only be done in `statnet` (e.g., ERGMs).
- Most of this workshop uses the package `igraph`. This is because I personally find `igraph`'s syntax more intuitive for beginners, especially for simple operations on networks. In Chapter 8.4 we'll also briefly demonstrate how networks are represented in `statnet`'s `network` objects. You should keep in mind that everything we'll do here with `igraph`, can also be done with `statnet`. In general, you'll need to learn about `statnet` too if you want to master a complete toolkit for social network analysis with R.
- We'll also briefly use the `ggraph` package for network visualization and the `tidygraph` package for easier display and manipulation of networks. We don't have time to go into much detail about these two packages, but you can find more tutorials and learning resources about them online.
  - `ggraph` flexible and easy tools for network visualization by applying the `ggplot2` grammar of graphics to network data (learn more here). In addition to `ggraph`, you can also visualize networks with `igraph`, which uses base R plotting (see Section 8.3).
  - `tidygraph` applies the `tidyverse` principles of tidy data to networks. This allows you to easily view and manipulate the basic components of network data in tabular format, i.e., edge data as an edge list and node attribute data as a case-by-variable dataset. More information about `tidygraph` is here.

3.4 Ego-networks as igraph objects

- In igraph, networks are called **graphs** and are represented by objects of class igraph. Nodes are called **vertices** and ties are called **edges**.
- Given an igraph object gr:
  - V(gr) shows you the graph vertices (identified by name, if they have a name attribute, or by integers otherwise). This is an object of class *vertex sequence* (igraph.vs).
  - E(gr) shows you the graph edges. This is an object of class *edge sequence* (igraph.es).
- Vertices, edges and graphs have **attributes**. You can import them from external data files, or you can set them manually in R. If you import data into an igraph object with a vertex attribute called "age", an edge attribute called "strength", and a graph attribute called "size", then
  - V(gr)$age returns the vertex attribute age as a vector;
  - E(gr)$strength returns the edge attribute strength as a vector;
  - gr$size returns the graph attribute size.
- printing an igraph object returns some **summary information** about the network. This includes the counts of vertices and edges, and whether the network is directed, named (i.e. if vertices have a name attribute), weighted (i.e. if edges have a weight attribute), or bipartite (also called two-mode).
- **Querying vertices and edges**
  - Based on attributes. You can query vertices and edges with specific characteristics (attribute values), and save them for re-use. E.g. V(gr)[age=30] returns all vertices whose age attribute equals 30; E(gr)[strength=1] returns all edges whose strength is 1.
  - Based on network structure. The V(gr)[...] and E(gr)[...] syntax can also be used with specific functions that extract information on tie distribution: for example, to query all vertices that are adjacent to a given vertex i, or all edges between two particular subsets of vertices. The main functions here are nei(), inc() and %--% (see code below).
  - More information on useful igraph syntax for vertex and edge indexing is here (vertex indexing) and here (edge indexing).
- **What we do in the following code**.
  - View an ego-network as an igraph object.
  - View vertex and edge attributes in an ego-network.
  - Obtain descriptive statistics for attributes of alters (vertex attributes) and alter-alter ties (edge attributes) in the ego-network.
  - Visualize an ego-network with ggraph, setting aesthetic parameters based on alter attributes.
  - Query vertices and edges based on attributes.

```
# Load packages.
library(igraph)
```

```
library(ggraph)
library(tidygraph)
library(skimr)
library(janitor)

# The data.rda file, which we loaded earlier, also includes the ego-network of
# ego 28 (as igraph object)
gr.28
```

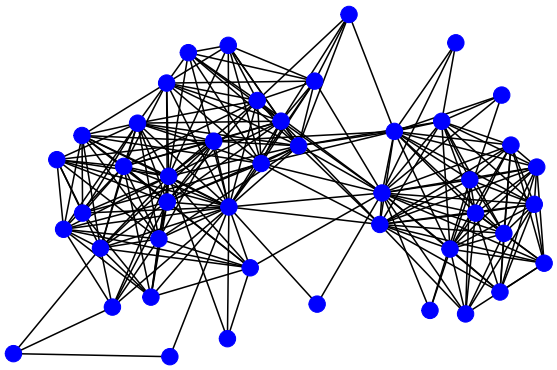
```
## IGRAPH ff0d051 UNW- 45 259 --
## + attr: .egoID (g/n), ego_ID (g/c), name (v/c), alter_num (v/n),
## | alter.sex (v/c), alter.age.cat (v/c), alter.rel (v/c), alter.nat
## | (v/c), alter.res (v/c), alter.clo (v/n), alter.loan (v/c), alter.fam
## | (v/c), alter.age (v/n), weight (e/n)
## + edges from ff0d051 (vertex names):
##  [1] 2801--2802 2801--2803 2801--2804 2801--2805 2801--2806 2801--2807
##  [7] 2801--2808 2801--2809 2801--2810 2801--2811 2801--2812 2801--2813
## [13] 2801--2814 2801--2815 2801--2818 2801--2820 2801--2823 2801--2825
## [19] 2801--2827 2801--2828 2801--2829 2801--2831 2801--2840 2801--2841
## [25] 2802--2803 2802--2804 2802--2805 2802--2806 2802--2807 2802--2808
## + ... omitted several edges
```

```
# The graph is Undirected, Named, Weighted. It has 45 vertices and 259 edges. It
# has a vertex attribute called "name", and an edge attribute called "weight".
# We see several vertex attributes (see codebook.xlsx for their meaning).

# Let's reassign it to a new, generic object name. This makes the code more
# generic and more easily re-usable on any ego-network igraph object
# (of any ego ID).
gr <- gr.28

# We also have the same network, with ego included: note the different number
# of vertices and edges compared to the previous graph.
gr.ego.28
```

```
## IGRAPH 83a192b UNW- 46 304 --
## + attr: .egoID (g/n), ego_ID (g/c), name (v/c), alter_num (v/n),
## | alter.sex (v/c), alter.age.cat (v/c), alter.rel (v/c), alter.nat
## | (v/c), alter.res (v/c), alter.clo (v/n), alter.loan (v/c), alter.fam
## | (v/c), alter.age (v/n), weight (e/n)
## + edges from 83a192b (vertex names):
##  [1] 2801--2802 2801--2803 2801--2804 2801--2805 2801--2806 2801--2807
##  [7] 2801--2808 2801--2809 2801--2810 2801--2811 2801--2812 2801--2813
## [13] 2801--2814 2801--2815 2801--2818 2801--2820 2801--2823 2801--2825
## [19] 2801--2827 2801--2828 2801--2829 2801--2831 2801--2840 2801--2841
## [25] 2802--2803 2802--2804 2802--2805 2802--2806 2802--2807 2802--2808
```

```
## + ... omitted several edges
```

```
gr.ego <- gr.ego.28

# Show the graph. Note that ego is not included.
set.seed(607)
ggraph(gr) +
  geom_edge_link() + # Draw edges
  geom_node_point(size=5, color="blue") + # Draw nodes
  theme_graph(base_family = 'Helvetica') # Set graph theme and font
```



```
# An igraph object can be indexed as an adjacency matrix.

# Adjacency row of alter #3.
gr[3,]
```

```
## 2801 2802 2803 2804 2805 2806 2807 2808 2809 2810 2811 2812 2813 2814 2815 2816
##    1    1    0    1    1    1    1    1    1    1    1    1    0    0    0    0
## 2817 2818 2819 2820 2821 2822 2823 2824 2825 2826 2827 2828 2829 2830 2831 2832
##    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0
## 2833 2834 2835 2836 2837 2838 2839 2840 2841 2842 2843 2844 2845
##    0    0    0    0    0    0    0    1    1    0    0    0    0
```

```
# Adjacency row of alter with vertex name (alter ID) "2805".
gr["2805",]
```

```
## 2801 2802 2803 2804 2805 2806 2807 2808 2809 2810 2811 2812 2813 2814 2815 2816
##    1    1    1    1    0    1    1    1    1    1    1    0    0    0    0    0
## 2817 2818 2819 2820 2821 2822 2823 2824 2825 2826 2827 2828 2829 2830 2831 2832
##    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0
## 2833 2834 2835 2836 2837 2838 2839 2840 2841 2842 2843 2844 2845
##    0    0    0    0    0    0    0    1    1    0    0    0    0
```

```
# Adjacency columns of alters 3-5
gr[,3:5]
```

```
## 45 x 3 sparse Matrix of class "dgCMatrix"
##      2803 2804 2805
## 2801    1    1    1
## 2802    1    1    1
## 2803    .    1    1
## 2804    1    .    1
## 2805    1    1    .
## 2806    1    1    1
## 2807    1    1    1
## 2808    1    1    1
## 2809    1    1    1
## 2810    1    1    1
## 2811    1    2    1
## 2812    1    .    .
## 2813    .    .    .
## 2814    .    .    .
## 2815    .    .    .
## 2816    .    .    .
## 2817    .    .    .
## 2818    .    .    .
## 2819    .    .    .
## 2820    .    .    .
## 2821    .    .    .
## 2822    .    .    .
## 2823    .    .    .
## 2824    .    .    .
## 2825    .    .    .
## 2826    .    .    .
## 2827    .    .    .
## 2828    .    .    .
## 2829    .    .    .
## 2830    .    .    .
## 2831    .    .    .
## 2832    .    .    .
## 2833    .    .    .
## 2834    .    .    .
## 2835    .    .    .
## 2836    .    .    .
## 2837    .    .    .
## 2838    .    .    .
## 2839    .    .    .
## 2840    1    2    1
## 2841    1    1    1
## 2842    .    .    .
## 2843    .    .    .
## 2844    .    .    .
```

```
## 2845    .    .    .
```

```
# Convert and view the ego-network as a tidygraph object: you can now see the
# attribute data and edge list as tabular datasets.
(gr.tbl <- as_tbl_graph(gr))
```

```
## # A tbl_graph: 45 nodes and 259 edges
## #
## # An undirected simple graph with 1 component
## #
## # A tibble: 45 x 11
##   name  alter_num alter.sex alter.age.cat alter.rel    alter.nat alter.res
##   <chr>     <dbl> <chr>     <chr>         <chr>        <chr>     <chr>
## 1 2801          1 Female    51-60         Close family Sri Lanka Sri Lanka
## 2 2802          2 Male      51-60         Other family Sri Lanka Sri Lanka
## 3 2803          3 Male      51-60         Close family Sri Lanka Sri Lanka
## 4 2804          4 Male      60+           Close family Sri Lanka Sri Lanka
## 5 2805          5 Female    41-50         Close family Sri Lanka Sri Lanka
## 6 2806          6 Female    60+           Close family Sri Lanka Sri Lanka
## # i 39 more rows
## # i 4 more variables: alter.clo <dbl>, alter.loan <chr>, alter.fam <chr>,
## #   alter.age <dbl>
## #
## # A tibble: 259 x 3
##    from    to weight
##   <int> <int>  <dbl>
## 1     1     2      1
## 2     1     3      1
## 3     1     4      1
## # i 256 more rows
```

```
# Vertex and edge sequences and attributes                              ----
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

# Vertex sequences and edge sequences can be extracted from a graph.

# Vertex sequence of the whole graph. Notice that vertices are displayed by
# "names" (alter IDs in this case).
V(gr)
```

```
## + 45/45 vertices, named, from ff0d051:
##  [1] 2801 2802 2803 2804 2805 2806 2807 2808 2809 2810 2811 2812 2813 2814 2815
## [16] 2816 2817 2818 2819 2820 2821 2822 2823 2824 2825 2826 2827 2828 2829 2830
## [31] 2831 2832 2833 2834 2835 2836 2837 2838 2839 2840 2841 2842 2843 2844 2845
```

```
# Edge sequence of the whole graph. Notice that vertex names (alter IDs) are
# used here too.
E(gr)
```

```
## + 259/259 edges from ff0d051 (vertex names):
##  [1] 2801--2802 2801--2803 2801--2804 2801--2805 2801--2806 2801--2807
##  [7] 2801--2808 2801--2809 2801--2810 2801--2811 2801--2812 2801--2813
## [13] 2801--2814 2801--2815 2801--2818 2801--2820 2801--2823 2801--2825
## [19] 2801--2827 2801--2828 2801--2829 2801--2831 2801--2840 2801--2841
## [25] 2802--2803 2802--2804 2802--2805 2802--2806 2802--2807 2802--2808
## [31] 2802--2809 2802--2810 2802--2811 2802--2812 2802--2813 2802--2815
## [37] 2802--2823 2802--2831 2802--2840 2802--2841 2803--2804 2803--2805
## [43] 2803--2806 2803--2807 2803--2808 2803--2809 2803--2810 2803--2811
## [49] 2803--2812 2803--2840 2803--2841 2804--2805 2804--2806 2804--2807
## [55] 2804--2808 2804--2809 2804--2810 2804--2811 2804--2840 2804--2841
## + ... omitted several edges
```

```
# View vertex attributes and calculate statistics on them.

# Closeness of alters to ego.
V(gr)$alter.clo
```

```
##  [1] NA  3 NA NA NA NA  5  4  5  5  5  4  3  5  3  3  1  5  4  5  5  5  5  5  2
## [26]  3  4  5  5  5  5  3  5  3  4  5  5  5  5  3  3  3  3  3  5
```

```
# Mean closeness of alters to ego.
mean(V(gr)$alter.clo, na.rm=TRUE)
```

```
## [1] 4.1
```

```
# More descriptive statistics on alter closeness.
skimr::skim_tee(V(gr)$alter.clo)
```

```
## -- Data Summary ------------------------
##                            Values
## Name                       data
## Number of rows             45
## Number of columns          1
## _________________________
## Column type frequency:
##   numeric                  1
## _________________________
## Group variables            None
##
## -- Variable type: numeric -------------------------------------------------------
##   skim_variable n_missing complete_rate mean   sd p0 p25 p50 p75 p100 hist
## 1 data                  5         0.889  4.1 1.08  1   3   5   5    5
```

```
# Alter's country of residence.
V(gr)$alter.res
```

```
##  [1] "Sri Lanka" "Sri Lanka" "Sri Lanka" "Sri Lanka" "Sri Lanka" "Sri Lanka"
##  [7] "Sri Lanka" "Sri Lanka" "Sri Lanka" "Sri Lanka" "Sri Lanka" "Sri Lanka"
## [13] "Italy"     "Italy"     "Italy"     "Italy"     "Italy"     "Italy"
## [19] "Italy"     "Other"     "Italy"     "Italy"     "Italy"     "Italy"
## [25] "Italy"     "Italy"     "Sri Lanka" "Sri Lanka" "Sri Lanka" "Italy"
## [31] "Italy"     "Italy"     "Italy"     "Italy"     "Italy"     "Italy"
## [37] "Italy"     "Italy"     "Italy"     "Sri Lanka" "Sri Lanka" "Italy"
## [43] "Italy"     "Italy"     "Italy"
```

```
# Frequencies.
janitor::tabyl(V(gr)$alter.res)
```

```
##  V(gr)$alter.res  n    percent
##            Italy 27 0.60000000
##            Other  1 0.02222222
##        Sri Lanka 17 0.37777778
```

```
# This is more readable with the pipe operator.
V(gr)$alter.res |>
  tabyl()
```

```
##  V(gr)$alter.res  n    percent
##            Italy 27 0.60000000
##            Other  1 0.02222222
##        Sri Lanka 17 0.37777778
```

```
# Alter IDs are stored in the "name" vertex attribute
V(gr)$name
```

```
##  [1] "2801" "2802" "2803" "2804" "2805" "2806" "2807" "2808" "2809" "2810"
## [11] "2811" "2812" "2813" "2814" "2815" "2816" "2817" "2818" "2819" "2820"
## [21] "2821" "2822" "2823" "2824" "2825" "2826" "2827" "2828" "2829" "2830"
## [31] "2831" "2832" "2833" "2834" "2835" "2836" "2837" "2838" "2839" "2840"
## [41] "2841" "2842" "2843" "2844" "2845"
```

```
# We can also create new vertex attributes.
V(gr)$new.attribute <- 1:45

# Now a new attribute is listed for the graph.
gr
```

```
## IGRAPH ff0d051 UNW- 45 259 --
## + attr: .egoID (g/n), ego_ID (g/c), name (v/c), alter_num (v/n),
## | alter.sex (v/c), alter.age.cat (v/c), alter.rel (v/c), alter.nat
## | (v/c), alter.res (v/c), alter.clo (v/n), alter.loan (v/c), alter.fam
## | (v/c), alter.age (v/n), new.attribute (v/n), weight (e/n)
## + edges from ff0d051 (vertex names):
##  [1] 2801--2802 2801--2803 2801--2804 2801--2805 2801--2806 2801--2807
##  [7] 2801--2808 2801--2809 2801--2810 2801--2811 2801--2812 2801--2813
```

```
## [13] 2801--2814 2801--2815 2801--2818 2801--2820 2801--2823 2801--2825
## [19] 2801--2827 2801--2828 2801--2829 2801--2831 2801--2840 2801--2841
## [25] 2802--2803 2802--2804 2802--2805 2802--2806 2802--2807 2802--2808
## + ... omitted several edges
```

```
# Also edges have attributes, in this case "weight".
E(gr)$weight
```

```
##   [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 2 1 1 1 1 1 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [38] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 1 1 1 1 1 1 1 1 1 1 1 1
##  [75] 1 1 1 1 1 2 1 1 1 1 1 1 1 1 1 1 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 2 1 2 1
## [112] 1 1 1 1 1 1 1 1 2 1 1 1 1 1 1 1 1 1 1 2 1 1 1 1 1 1 1 1 2 1 1 1 1 2 1 1 1
## [149] 1 1 1 1 1 1 1 1 2 2 1 1 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 1 1 1 1 1 1
## [186] 1 1 1 1 1 1 1 1 1 1 1 1 2 1 1 1 1 1 1 1 1 1 1 2 1 1 1 1 2 1 1 2 1 1 1 1 1 1
## [223] 1 1 1 1 1 1 1 1 2 1 1 1 1 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 1 1
```

```
# Frequencies of edge weights.
tabyl(E(gr)$weight)
```

```
##  E(gr)$weight   n    percent
##             1 235 0.90733591
##             2  24 0.09266409
```

```
# Also graphs have attributes, and we can query or assign them. For example,
# assign the "ego_ID" attribute for this graph: this is the personal network of
# ego ID 28.
gr$ego_ID <- 28

# Check that the new attribute was created
gr
```

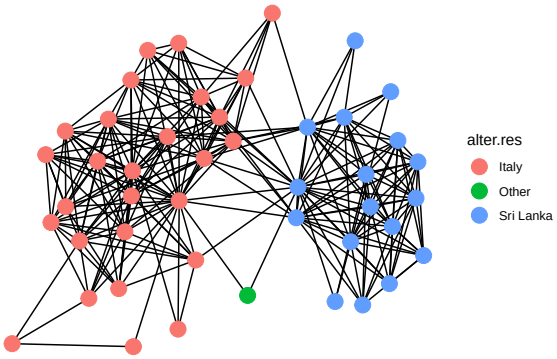
```
## IGRAPH ff0d051 UNW- 45 259 --
## + attr: .egoID (g/n), ego_ID (g/n), name (v/c), alter_num (v/n),
## | alter.sex (v/c), alter.age.cat (v/c), alter.rel (v/c), alter.nat
## | (v/c), alter.res (v/c), alter.clo (v/n), alter.loan (v/c), alter.fam
## | (v/c), alter.age (v/n), new.attribute (v/n), weight (e/n)
## + edges from ff0d051 (vertex names):
##  [1] 2801--2802 2801--2803 2801--2804 2801--2805 2801--2806 2801--2807
##  [7] 2801--2808 2801--2809 2801--2810 2801--2811 2801--2812 2801--2813
## [13] 2801--2814 2801--2815 2801--2818 2801--2820 2801--2823 2801--2825
## [19] 2801--2827 2801--2828 2801--2829 2801--2831 2801--2840 2801--2841
## [25] 2802--2803 2802--2804 2802--2805 2802--2806 2802--2807 2802--2808
## + ... omitted several edges
```

```
# Displaying vertex attributes in network visualization                ----
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

# Let's visualize our ego-network, using different node colors for different
```

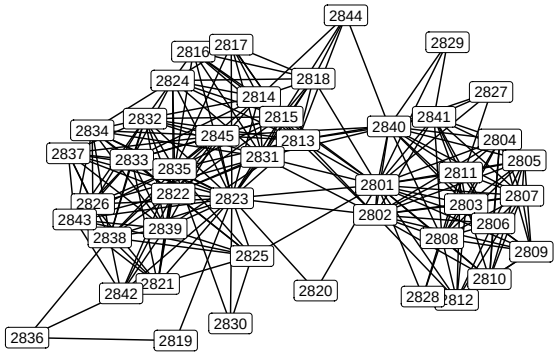
```
# countries of residence of alters.

# Vertex attribute with alter's country of residence.
V(gr)$alter.res
```

```
##  [1] "Sri Lanka" "Sri Lanka" "Sri Lanka" "Sri Lanka" "Sri Lanka" "Sri Lanka"
##  [7] "Sri Lanka" "Sri Lanka" "Sri Lanka" "Sri Lanka" "Sri Lanka" "Sri Lanka"
## [13] "Italy"     "Italy"     "Italy"     "Italy"     "Italy"     "Italy"
## [19] "Italy"     "Other"     "Italy"     "Italy"     "Italy"     "Italy"
## [25] "Italy"     "Italy"     "Sri Lanka" "Sri Lanka" "Sri Lanka" "Italy"
## [31] "Italy"     "Italy"     "Italy"     "Italy"     "Italy"     "Italy"
## [37] "Italy"     "Italy"     "Italy"     "Sri Lanka" "Sri Lanka" "Italy"
## [43] "Italy"     "Italy"     "Italy"
```

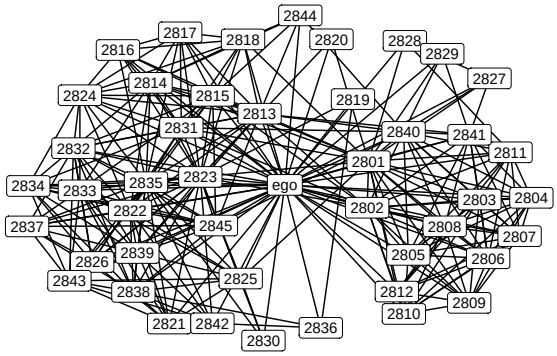
```
# Plot with alter.res as node color
set.seed(607)
ggraph(gr) +
  geom_edge_link() + # Draw edges
  geom_node_point(aes(color= alter.res), size=5) + # Draw nodes setting alter.res
  # as node color and fixed node size
  theme_graph(base_family = 'Helvetica')
```




```
# Plot with alter ID labels instead of circles
set.seed(607)
ggraph(gr) +
  geom_edge_link() +
  geom_node_label(aes(label= name)) +
  theme_graph(base_family = 'Helvetica')
```




```
# Let's look at the same network, but with ego included
set.seed(607)
ggraph(gr.ego) +
  geom_edge_link() +
  geom_node_label(aes(label= name)) +
  theme_graph(base_family = 'Helvetica')
```




```
# Indexing vertices and edges based on attributes or network structure      ----
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

# View only female alters.
V(gr)[alter.sex=="Female"]
```

```
## + 8/45 vertices, named, from ff0d051:
## [1] 2801 2805 2806 2808 2809 2828 2832 2837
```

```
# View residence country just for female alters.
V(gr)[alter.sex=="Female"]$alter.res
```

```
## [1] "Sri Lanka" "Sri Lanka" "Sri Lanka" "Sri Lanka" "Sri Lanka" "Sri Lanka"
## [7] "Italy"     "Italy"
```

```
# Frequencies of countries of residence of female alters.
V(gr)[alter.sex=="Female"]$alter.res |>
```

```
  tabyl()
```

```
##  V(gr)[alter.sex == "Female"]$alter.res n percent
##                                   Italy 2    0.25
##                               Sri Lanka 6    0.75
```

```
# View nationality for alter 2833.
V(gr)[name=="2833"]$alter.nat
```

```
## [1] "Sri Lanka"
```

```
# Edit nationality for alter 2833
V(gr)[name=="2833"]$alter.nat <- "Italy"

# View all alters who know alter 2833 (vertices that are adjacent to vertex
# "2833").
V(gr)[.nei("2833")]
```

```
## + 16/45 vertices, named, from ff0d051:
##  [1] 2814 2815 2821 2822 2823 2824 2825 2826 2832 2834 2835 2837 2838 2839 2843
## [16] 2845
```

```
# View the gender of all alters who know alter 2833.
V(gr)[.nei("2833")]$alter.sex
```

```
##  [1] "Male"   "Male"   "Male"   "Male"   "Male"   "Male"   "Male"   "Male"
##  [9] "Female" "Male"   "Male"   "Female" "Male"   "Male"   "Male"   "Male"
```

```
# View all edges that are incident on alter 2833
E(gr)[.inc("2833")]
```

```
## + 16/259 edges from ff0d051 (vertex names):
##  [1] 2814--2833 2815--2833 2821--2833 2822--2833 2823--2833 2824--2833
##  [7] 2825--2833 2826--2833 2832--2833 2833--2834 2833--2835 2833--2837
## [13] 2833--2838 2833--2839 2833--2843 2833--2845
```

```
# View the weight of these edges.
E(gr)[.inc("2833")]$weight
```

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

```
# Max weight of any edge that is incident on alter 2833.
E(gr)[.inc("2833")]$weight |>
  max()
```

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

```
# View all edges with weight==2 (alters who "maybe" know each other).
E(gr)[weight==2]
```

```
## + 24/259 edges from ff0d051 (vertex names):
```

```
##  [1] 2801--2818 2801--2829 2804--2811 2804--2840 2807--2811 2811--2812
##  [7] 2811--2828 2813--2841 2813--2845 2814--2833 2815--2833 2816--2835
## [13] 2817--2835 2821--2826 2821--2833 2821--2839 2823--2830 2825--2833
## [19] 2827--2828 2830--2835 2831--2840 2833--2843 2834--2843 2840--2844
```

```
# Conditions in R indexing can always be combined: view all "maybe" edges
# that are incident on alter 2833.
E(gr)[weight==2 & .inc("2833")]
```

```
## + 5/259 edges from ff0d051 (vertex names):
## [1] 2814--2833 2815--2833 2821--2833 2825--2833 2833--2843
```

```
# ***** EXERCISES
# (1) Get the average alter closeness (vertex attribute $alter.clo) of Sri Lankan
# alters in the personal network gr. Use mean().
#
# (2) How many edges in the personal network gr involve at least one close
# family member and have $weight==2 (the two alters "maybe" know each other)?
# Remember that close family members have $relation=="Close family". Use .inc().
#
# *****
```

Chapter 4

Ego-network composition

4.1 Overview

Ego-network composition refers to the distribution of attributes of alters (for example, age and race/ethnicity) or attributes of ties between alters and the ego (for example, closeness or frequency of contact). For brevity, these are often called simply *alter attributes* in the following text. Recall that, in typical egocentric network data, the level of alters is the same as the level of ego-alter ties: for each alter there is one and only one ego-alter tie, and vice versa. Like in other chapters, we consider composition by first analyzing just one ego-network, then replicating the same type of analysis on many ego-networks at once.

This chapter covers the following topics:

- Calculating measures of composition for one ego-network.
- Representing data from multiple ego-networks as data frames.
- Running the same operation on many ego-networks and combining results back together (split-apply-combine).
- Split-apply-combine on data frames with `dplyr` to analyze the composition of many ego-networks at once.

4.2 Measures of ego-network composition

- Many different measures can be calculated in R to describe ego-network composition.
  - The result is typically an ego-level summary variable: one that assigns a number to each ego, with that number describing a characteristic of the ego-network composition for each ego.

- To calculate compositional measures we don't need any network or relational data (data about alter-alter ties), we only need the alter-level attribute dataset (for example, with alter age, gender, ethnicity, frequency of contact, etc.). In other words, we only need, for each ego, a list of alters with their characteristics, and no information about ties between alters.
  - So in this section, ego-networks are represented simply by alter-level data frames. We don't need to work with the `igraph` network objects until we want to analyze ego-network structure (alter-alter ties).
- There are at least three types of compositional measures that we may want to calculate on ego networks. All these measures are calculated for each ego.
  - Measures based on *one attribute* of alters. For example, average alter age in the network.
  - Measures based on *multiple attributes* of alters. For example, average frequency of contact (attribute 1) between ego and alters who are family members (attribute 2).
  - Measures of *ego-alter homophily*. These are summary measures of the extent to which alters are similar to the ego who nominated them, with respect to one or more attributes. For example, the proportion of alters who are of the same gender (ethnicity, age bracket) as the ego who nominated them.
- **What we do in the following code**.
  - Look at the alter attribute data frame for one ego.
  - Using this data frame, calculate compositional measures based on one alter attribute.
  - Calculate compositional measures based on two alter attributes.
  - Do the level-1 join: join ego attributes into alter-level data.
  - Using the joined data, calculate compositional measures of homophily between ego and alters.
  - Calculate multiple compositional measures and put them together into one ego-level data frame.

```
# Load packages.
library(tidyverse)
library(skimr)
library(janitor)

# Load data.
load("./Data/data.rda")

# For compositional measures all we need is the alter attribute data frame.
# The data.rda file loaded above includes the alter attribute data frame for
# ego ID 28.
alter.attr.28
```

```
## # A tibble: 45 x 12
```

```
##    alter_ID ego_ID alter_num alter.sex alter.age.cat alter.rel    alter.nat
##       <dbl>  <dbl>     <dbl> <fct>     <fct>         <fct>        <fct>
##  1     2801     28         1 Female    51-60         Close family Sri Lanka
##  2     2802     28         2 Male      51-60         Other family Sri Lanka
##  3     2803     28         3 Male      51-60         Close family Sri Lanka
##  4     2804     28         4 Male      60+           Close family Sri Lanka
##  5     2805     28         5 Female    41-50         Close family Sri Lanka
##  6     2806     28         6 Female    60+           Close family Sri Lanka
##  7     2807     28         7 Male      41-50         Other family Sri Lanka
##  8     2808     28         8 Female    36-40         Other family Sri Lanka
##  9     2809     28         9 Female    51-60         Other family Sri Lanka
## 10     2810     28        10 Male      60+           Other family Sri Lanka
## # i 35 more rows
## # i 5 more variables: alter.res <fct>, alter.clo <dbl>, alter.loan <fct>,
## #   alter.fam <fct>, alter.age <dbl>
```

```
# Compositional measures based on a single alter attribute                ----
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

# Summary of continuous variable: Average alter closeness.

# Check out the relevant variable
alter.attr.28$alter.clo
```

```
##  [1] NA  3 NA NA NA NA  5  4  5  5  5  4  3  5  3  3  1  5  4  5  5  5  5  5  2
## [26]  3  4  5  5  5  5  3  5  3  4  5  5  5  5  3  3  3  3  3  5
```

```
# Or, in tidyverse syntax
alter.attr.28 |>
  pull(alter.clo)
```

```
##  [1] NA  3 NA NA NA NA  5  4  5  5  5  4  3  5  3  3  1  5  4  5  5  5  5  5  2
## [26]  3  4  5  5  5  5  3  5  3  4  5  5  5  5  3  3  3  3  3  5
```

```
# Battery of descriptive stats.
alter.attr.28 |>
  skim_tee(alter.clo)
```

```
## -- Data Summary ------------------------
##                            Values
## Name                       data
## Number of rows             45
## Number of columns          12
## _______________________
## Column type frequency:
##   numeric                  1
## _______________________
## Group variables            None
```

```
##
## -- Variable type: numeric -------------------------------------------------------
##   skim_variable n_missing complete_rate mean   sd p0 p25 p50 p75 p100 hist
## 1 alter.clo             5         0.889  4.1 1.08  1   3   5   5    5
```

```
# Get a single summary measure (useful when writing functions)
mean(alter.attr.28$alter.clo, na.rm = TRUE)
```

```
## [1] 4.1
```

```
# Summary of categorical variable: Proportion female alters.

# Check out the relevant vector
alter.attr.28$alter.sex
```

```
##  [1] Female Male   Male   Male   Female Female Male   Female Female Male
## [11] Male   Male   Male   Male   Male   Male   Male   Male   Male   Male
## [21] Male   Male   Male   Male   Male   Male   Male   Female Male   Male
## [31] Male   Female Male   Male   Male   Male   Female Male   Male   Male
## [41] Male   Male   Male   Male   Male
## Levels: Female Male
```

```
# Get frequencies
alter.attr.28 |>
  tabyl(alter.sex)
```

```
##  alter.sex  n   percent
##     Female  8 0.1777778
##       Male 37 0.8222222
```

```
# Same for nationalities
alter.attr.28 |>
  tabyl(alter.nat)
```

```
##  alter.nat  n    percent
##      Italy  0 0.00000000
##      Other  2 0.04444444
##  Sri Lanka 43 0.95555556
```

```
# Another way to get the proportion of a specific category (this is useful when
# writing functions).
mean(alter.attr.28$alter.sex == "Female")
```

```
## [1] 0.1777778
```

```
mean(alter.attr.28$alter.nat == "Sri Lanka")
```

```
## [1] 0.9555556
```

```
# The function dplyr::summarise() allows us to calculate multiple measures, name
# them, and put them together in a data frame.
```

```
alter.attr.28 |>
  summarise(
    mean.clo = mean(alter.clo, na.rm=TRUE),
    prop.fem = mean(alter.sex=="Female"),
    count.nat.slk = sum(alter.nat=="Sri Lanka"),
    count.nat.ita = sum(alter.nat=="Italy"),
    count.nat.oth = sum(alter.nat=="Other")
  )
```

```
## # A tibble: 1 x 5
##   mean.clo prop.fem count.nat.slk count.nat.ita count.nat.oth
##      <dbl>    <dbl>         <int>         <int>         <int>
## 1      4.1    0.178            43             0             2
```

```
# What if we want to calculate the same measures for all ego-networks in the data?
# We'll have to use the data frame with all alter attributes from all egos.
alter.attr.all
```

```
## # A tibble: 4,590 x 12
##    alter_ID ego_ID alter_num alter.sex alter.age.cat alter.rel    alter.nat
##       <dbl>  <dbl>     <dbl> <fct>     <fct>         <fct>        <fct>
##  1     2801     28         1 Female    51-60         Close family Sri Lanka
##  2     2802     28         2 Male      51-60         Other family Sri Lanka
##  3     2803     28         3 Male      51-60         Close family Sri Lanka
##  4     2804     28         4 Male      60+           Close family Sri Lanka
##  5     2805     28         5 Female    41-50         Close family Sri Lanka
##  6     2806     28         6 Female    60+           Close family Sri Lanka
##  7     2807     28         7 Male      41-50         Other family Sri Lanka
##  8     2808     28         8 Female    36-40         Other family Sri Lanka
##  9     2809     28         9 Female    51-60         Other family Sri Lanka
## 10     2810     28        10 Male      60+           Other family Sri Lanka
## # i 4,580 more rows
## # i 5 more variables: alter.res <fct>, alter.clo <dbl>, alter.loan <fct>,
## #   alter.fam <fct>, alter.age <dbl>
```

```
# dplyr allows us to "group" a data frame by a factor (here, ego IDs) so all
# measures we calculate on that data frame via summarise (means, proportions,
# etc.) are calculated by the groups given by that factor (here, for each ego
# ID).
alter.attr.all |>
  group_by(ego_ID) |>
  summarise(
    mean.clo = mean(alter.clo, na.rm=TRUE),
    prop.fem = mean(alter.sex=="Female"),
    count.nat.slk = sum(alter.nat=="Sri Lanka"),
    count.nat.ita = sum(alter.nat=="Italy"),
    count.nat.oth = sum(alter.nat=="Other")
```

```
  )
```

```
## # A tibble: 102 x 6
##    ego_ID mean.clo prop.fem count.nat.slk count.nat.ita count.nat.oth
##     <dbl>    <dbl>    <dbl>         <int>         <int>         <int>
##  1     28     4.1    0.178             43             0             2
##  2     29     4.03   0.0889            44             1             0
##  3     33     3.62   0.378             32             2            11
##  4     35     3.78   0.289             33             4             8
##  5     39     3.73   0.244             39             5             1
##  6     40     3.32   0.356             34             1            10
##  7     45     4.02   0.244             14            19            12
##  8     46     3.48   0.4               33             7             5
##  9     47     4.05   0.267             45             0             0
## 10     48     4.07   0.311             39             4             2
## # i 92 more rows
```

```
# We'll talk more about this and show more examples in the next sections.


# Compositional measures based on multiple alter attributes                ----
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

# Using indexing we can combine multiple alter attribute variables.

# Mean closeness of alters who are "Friends".

# Check out the relevant vector.
alter.attr.28 |>
  filter(alter.rel=="Friends") |>
  pull(alter.clo)
```

```
##  [1] 5 4 3 5 3 3 5 5 5 5 5 5 4 5 5 5 5 5 4 5 5 5 5 3 3 3 5
```

```
# Get its mean.
alter.attr.28 |>
  filter(alter.rel=="Friends") |>
  pull(alter.clo) |>
  mean()
```

```
## [1] 4.444444
```

```
# Mean closeness of alters who are "Acquaintances".
alter.attr.28 |>
  filter(alter.rel=="Acquaintances") |>
  pull(alter.clo) |>
  mean()
```

```
## [1] 2.75
```

```
# Equivalently (useful for writing functions)
mean(alter.attr.28$alter.clo[alter.attr.28$alter.rel=="Acquaintances"])
```

```
## [1] 2.75
```

```
# Count of close family members who live in Sri Lanka vs those who live in Italy.

# In Sri Lanka.
alter.attr.28 |>
  filter(alter.rel == "Close family", alter.res == "Sri Lanka") |>
  count()
```

```
## # A tibble: 1 x 1
##       n
##   <int>
## 1     5
```

```
# Equivalently (useful for writing functions)
sum(alter.attr.28$alter.rel == "Close family" & alter.attr.28$alter.res == "Sri Lanka")
```

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

```
# In Italy.
sum(alter.attr.28$alter.rel == "Close family" & alter.attr.28$alter.res == "Italy")
```

```
## [1] 0
```

```
# Again, we can put these measures together into a data frame row with dplyr.
alter.attr.28 |>
  summarise(
    mean.clo.fr = mean(alter.clo[alter.rel=="Friends"]),
    mean.clo.acq = mean(alter.clo[alter.rel=="Acquaintances"]),
    count.fam.slk = sum(alter.rel=="Close family" & alter.res=="Sri Lanka"),
    count.fam.ita = sum(alter.rel=="Close family" & alter.res=="Italy")
  )
```

```
## # A tibble: 1 x 4
##   mean.clo.fr mean.clo.acq count.fam.slk count.fam.ita
##         <dbl>        <dbl>         <int>         <int>
## 1        4.44         2.75             5             0
```

```
# Compositional measures of homophily between ego and alters                ----
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

# Level-1 join: Bring ego-level data into alter-level data frame for ego 28.
(data.28 <- left_join(alter.attr.28, ego.df, by= "ego_ID"))
```

```
## # A tibble: 45 x 20
```

```
##    alter_ID ego_ID alter_num alter.sex alter.age.cat alter.rel    alter.nat
##       <dbl>  <dbl>     <dbl> <fct>     <fct>         <fct>        <fct>
##  1     2801     28         1 Female    51-60         Close family Sri Lanka
##  2     2802     28         2 Male      51-60         Other family Sri Lanka
##  3     2803     28         3 Male      51-60         Close family Sri Lanka
##  4     2804     28         4 Male      60+           Close family Sri Lanka
##  5     2805     28         5 Female    41-50         Close family Sri Lanka
##  6     2806     28         6 Female    60+           Close family Sri Lanka
##  7     2807     28         7 Male      41-50         Other family Sri Lanka
##  8     2808     28         8 Female    36-40         Other family Sri Lanka
##  9     2809     28         9 Female    51-60         Other family Sri Lanka
## 10     2810     28        10 Male      60+           Other family Sri Lanka
## # i 35 more rows
## # i 13 more variables: alter.res <fct>, alter.clo <dbl>, alter.loan <fct>,
## #   alter.fam <fct>, alter.age <dbl>, ego.sex <fct>, ego.age <dbl>,
## #   ego.arr <dbl>, ego.edu <fct>, ego.inc <dbl>, empl <dbl>,
## #   ego.empl.bin <fct>, ego.age.cat <fct>
```

```
# Note the left join: We only retain rows in the left data frame (i.e., alters
# of ego 28), and discard all egos in the right data frame that do not
# correspond to those rows.

# Example: Proportion of alters of the same gender as ego.

# View the relevant data
data.28 |>
  dplyr::select(alter_ID, ego_ID, alter.sex, ego.sex)
```

```
## # A tibble: 45 x 4
##    alter_ID ego_ID alter.sex ego.sex
##       <dbl>  <dbl> <fct>     <fct>
##  1     2801     28 Female    Male
##  2     2802     28 Male      Male
##  3     2803     28 Male      Male
##  4     2804     28 Male      Male
##  5     2805     28 Female    Male
##  6     2806     28 Female    Male
##  7     2807     28 Male      Male
##  8     2808     28 Female    Male
##  9     2809     28 Female    Male
## 10     2810     28 Male      Male
## # i 35 more rows
```

```
# First create a vector that is TRUE whenever alter has the same sex as ego in
# data.28 (the joined data frame for ego 28).
data.28$alter.sex == data.28$ego.sex
```

```
##  [1] FALSE  TRUE  TRUE  TRUE FALSE FALSE  TRUE FALSE FALSE  TRUE  TRUE  TRUE
## [13]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [25]  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE
## [37] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
```

```
# The proportion we're looking for is simply the proportion of TRUE's in this
# vector.
mean(data.28$alter.sex == data.28$ego.sex)
```

```
## [1] 0.8222222
```

```
# Similarly: count of alters who are in the same age bracket as the ego.
sum(data.28$alter.age.cat == data.28$ego.age.cat)
```

```
## [1] 9
```

```
# Again, we can put these measures together into a data frame row with dplyr.
data.28 |>
  summarise(
    prop.same.gender = mean(alter.sex == ego.sex),
    count.same.age = sum(alter.age.cat == ego.age.cat)
  )
```

```
## # A tibble: 1 x 2
##   prop.same.gender count.same.age
##              <dbl>          <int>
## 1            0.822              9
```

4.3 Analyzing the composition of many ego-networks

4.3.1 Split-apply-combine in egocentric network analysis

- Now that we've learned how to represent and analyze data on one ego-network, we're ready to scale these operations up to a *collection* of many ego-networks.
- Often in social science data analaysis (or any data analysis), our data are in a single file, dataset or object, and we need to:
  1. *Split* the object into pieces based on one or multiple (combinations of) categorical variables or factors.
  2. *Apply* exactly the same type of calculation on each piece, identically and independently.
  3. *Combine* all results back together, for example into a new dataset.
- This has been called the **split-apply-combine** strategy [Wickham, 2011] and is essential in egocentric network analysis. With ego-networks, we are constantly (1) splitting the data into pieces, each piece typically corresponding to one ego; (2) performing identical and independent analyses

on each piece (each ego-network); (3) combining the results back together, typically into a single ego-level dataset, to then associate them with other ego-level variables.
- In base and traditional R, common tools to perform split-apply-combine operations include `for` loops, the `apply` family of functions, and `aggregate`.
- The `tidyverse` packages provide new ways of conducting split-apply-combine operations with more efficient and readable code:
  - Grouping and summarizing data frames with the `dplyr` package. This is particularly relevant to ego-network composition (next section).
  - Applying the same function to all elements in a list with the `map` family of functions in the `purrr` package. This is more relevant to ego-network structure (see Section 5.4)

4.3.2 Grouping and summarizing with `dplyr`

- Whenever we have a dataset in which rows (level 1) are clustered or grouped by values of a given factor (level 2), the package `dplyr` makes level-2 summarizations very easy.
  - In egocentric analysis, we typically have an alter attribute data frame whose rows (alters, level 1) are clustered by egos (level 2).
- In general, the `dplyr::summarise` function allows us to calculate summary statistics on a single variable or on multiple variables in a data frame.
  - To calculate the same summary statistic on multiple variables, we select them with `across()`.
- If we run `summarise` after *grouping* the data frame by a factor variable with `group_by`, then the data frame will be "split" by levels (categories) of that factor, and the summary statistics will be calculated on each piece: that is, for each unique level of the grouping factor.
  - So if the grouping factor is the ego ID, we can immediately obtain summary statistics for each of hundreds or thousands of egos in one line of code. The code below provides examples.
- **What we do in the following code**.
  - Use `summarise` to calculate summary variables on network composition for all of the 102 egos at once.
  - Join the results with other ego-level data (level-2 join).

```
# The summarise function offers a concise syntax to calculate summary
# statistics on a data frame's variables.

# Let's see what happens if we apply this function to the alter attribute data
# frame including all alters, without grouping it by ego ID.

# * Mean alter closeness:
```

```
alter.attr.all |>
  summarise(mean.clo = mean(alter.clo, na.rm = TRUE))
```

```
## # A tibble: 1 x 1
##   mean.clo
##      <dbl>
## 1     3.87
```

```
# * N of distinct values in the alter nationality variable (i.e., number of
# distinct nationalities of alters):
alter.attr.all |>
  summarise(N.nat = n_distinct(alter.nat))
```

```
## # A tibble: 1 x 1
##   N.nat
##   <int>
## 1     3
```

```
# * N of distinct values in the alter nationality, country of residence, and
# age bracket variables. In this case, we apply the same summarizing function
# to multiple variables (not just one), to be selected via across().
alter.attr.all |>
  summarise(
    across(c(alter.nat, alter.res, alter.age.cat),
           n_distinct)
    )
```

```
## # A tibble: 1 x 3
##   alter.nat alter.res alter.age.cat
##       <int>     <int>         <int>
## 1         3         3             8
```

```
# Because we ran this without previously grouping the data frame by ego ID, each
# function is calculated on all alters from all egos pooled (all rows of the
# alter attribute data frame), not on the set of alters of each ego.

# If we group the data frame by ego_ID, each of those summary statistics is
# calculated for each ego:

# * Mean alter closeness:
alter.attr.all |>
  # Group by ego ID
  group_by(ego_ID) |>
  # Calculate summary measure
  summarise(mean.clo = mean(alter.clo, na.rm = TRUE))
```

```
## # A tibble: 102 x 2
##    ego_ID mean.clo
```

```
##     <dbl>    <dbl>
##  1     28     4.1
##  2     29     4.03
##  3     33     3.62
##  4     35     3.78
##  5     39     3.73
##  6     40     3.32
##  7     45     4.02
##  8     46     3.48
##  9     47     4.05
## 10     48     4.07
## # i 92 more rows
```

```
# * N of distinct values in the alter nationality variable (i.e., number of
# distinct nationalities of alters):
alter.attr.all |>
  group_by(ego_ID) |>
  summarise(N.nat = n_distinct(alter.nat))
```

```
## # A tibble: 102 x 2
##    ego_ID N.nat
##     <dbl> <int>
##  1     28     2
##  2     29     2
##  3     33     3
##  4     35     3
##  5     39     3
##  6     40     3
##  7     45     3
##  8     46     3
##  9     47     1
## 10     48     3
## # i 92 more rows
```

```
# We can also "permanently" group the data frame by ego_ID and then calculate all
# our summary measures by ego ID.
alter.attr.all <- alter.attr.all |>
  group_by(ego_ID)

# * N of distinct values in the alter nationality, country of residence, and
# age bracket variables:
alter.attr.all |>
  summarise(
    across(c(alter.nat, alter.res, alter.age.cat),
           n_distinct)
    )
```

```
## # A tibble: 102 x 4
##    ego_ID alter.nat alter.res alter.age.cat
##     <dbl>     <int>     <int>         <int>
##  1     28         2         3             7
##  2     29         2         3             7
##  3     33         3         2             6
##  4     35         3         3             7
##  5     39         3         3             7
##  6     40         3         3             6
##  7     45         3         3             8
##  8     46         3         3             7
##  9     47         1         3             6
## 10     48         3         3             7
## # i 92 more rows
```

```
# We can also use summarise to run more complex functions on alter attributes
# by ego.

# Imagine we want to count the number of alters who are "Close family", "Other
# family", and "Friends" in an ego-network.

# Let's consider the ego-network of ego ID 28 as an example.
alter.attr.28
```

```
## # A tibble: 45 x 12
##    alter_ID ego_ID alter_num alter.sex alter.age.cat alter.rel    alter.nat
##       <dbl>  <dbl>     <dbl> <fct>     <fct>         <fct>        <fct>
##  1     2801     28         1 Female    51-60         Close family Sri Lanka
##  2     2802     28         2 Male      51-60         Other family Sri Lanka
##  3     2803     28         3 Male      51-60         Close family Sri Lanka
##  4     2804     28         4 Male      60+           Close family Sri Lanka
##  5     2805     28         5 Female    41-50         Close family Sri Lanka
##  6     2806     28         6 Female    60+           Close family Sri Lanka
##  7     2807     28         7 Male      41-50         Other family Sri Lanka
##  8     2808     28         8 Female    36-40         Other family Sri Lanka
##  9     2809     28         9 Female    51-60         Other family Sri Lanka
## 10     2810     28        10 Male      60+           Other family Sri Lanka
## # i 35 more rows
## # i 5 more variables: alter.res <fct>, alter.clo <dbl>, alter.loan <fct>,
## #   alter.fam <fct>, alter.age <dbl>
```

```
# Calculate the number of alters in each relationship type in this ego-network.

# Vector of alter relationship attribute.
alter.attr.28$alter.rel
```

```
##  [1] Close family  Other family  Close family  Close family  Close family
```

```
##  [6] Close family  Other family  Other family  Other family  Other family
## [11] Friends       Friends       Friends       Friends       Friends
## [16] Friends       Acquaintances Friends       Acquaintances Friends
## [21] Friends       Friends       Friends       Friends       Acquaintances
## [26] Acquaintances Friends       Friends       Friends       Friends
## [31] Friends       Acquaintances Friends       Acquaintances Friends
## [36] Friends       Friends       Friends       Friends       Friends
## [41] Friends       Acquaintances Acquaintances Friends       Friends
## Levels: Acquaintances Close family Friends Other family
```

```
# Flag with TRUE whenever alter is "Close family"
alter.attr.28$alter.rel=="Close family"
```

```
##  [1]  TRUE FALSE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
## [13] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [25] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [37] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

```
# Count the number of TRUE's
sum(alter.attr.28$alter.rel=="Close family")
```

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

```
# The same can be done for "Other family" and "Friends"
sum(alter.attr.28$alter.rel=="Other family")
```

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

```
sum(alter.attr.28$alter.rel=="Friends")
```

```
## [1] 27
```

```
# With dplyr we can run the same operations for every ego
N.rel <- alter.attr.all |>
  summarise(N.clo.fam = sum(alter.rel=="Close family"),
            N.oth.fam = sum(alter.rel=="Other family"),
            N.fri = sum(alter.rel=="Friends"))
N.rel
```

```
## # A tibble: 102 x 4
##    ego_ID N.clo.fam N.oth.fam N.fri
##     <dbl>     <int>     <int> <int>
##  1     28         5         5    27
##  2     29         5         6    30
##  3     33         3        16    13
##  4     35         4         1    38
##  5     39         4         9    27
##  6     40         5         7    17
##  7     45         3         6    30
##  8     46         3        14    12
```

```
##  9     47         1        20    18
## 10     48         4         6    26
## # i 92 more rows
```

```
# After getting compositional summary variables for each ego, we might want to
# join them with other ego-level data (level-2 join).

# Merge with summary variables.
ego.df |>
  left_join(N.rel, by= "ego_ID")
```

```
## # A tibble: 102 x 12
##    ego_ID ego.sex ego.age ego.arr ego.edu ego.inc  empl ego.empl.bin ego.age.cat
##     <dbl> <fct>     <dbl>   <dbl> <fct>     <dbl> <dbl> <fct>        <fct>
##  1     28 Male         61    2008 Second~     350     3 Yes          60+
##  2     29 Male         38    2000 Primary     900     4 Yes          36-40
##  3     33 Male         30    2010 Primary     200     3 Yes          26-30
##  4     35 Male         25    2009 Second~    1000     3 Yes          18-25
##  5     39 Male         29    2007 Primary       0     1 No           26-30
##  6     40 Male         56    2008 Second~     950     4 Yes          51-60
##  7     45 Male         52    1975 Primary    1600     3 Yes          51-60
##  8     46 Male         35    2002 Second~    1200     4 Yes          31-35
##  9     47 Male         22    2010 Second~     700     4 Yes          18-25
## 10     48 Male         51    2007 Primary     950     4 Yes          51-60
## # i 92 more rows
## # i 3 more variables: N.clo.fam <int>, N.oth.fam <int>, N.fri <int>
```

```
# We can then ungroup alter.attr.all by ego ID to remove the grouping information.
alter.attr.all <- ungroup(alter.attr.all)

# To get the size of each personal network, we can simpy use the count() function:
# it counts the number of rows for each unique value of a variable. The
# number of rows for each unique value of ego_ID in alter.attr.all is the number
# of alters for each ego (personal network size).
alter.attr.all |>
  dplyr::count(ego_ID)
```

```
## # A tibble: 102 x 2
##    ego_ID     n
##     <dbl> <int>
##  1     28    45
##  2     29    45
##  3     33    45
##  4     35    45
##  5     39    45
##  6     40    45
##  7     45    45
```

```
##  8     46    45
##  9     47    45
## 10     48    45
## # i 92 more rows
```

```
# ***** EXERCISES
#
# (1) Extract the values of alter.attr.all$alter.sex corresponding to ego_ID 28
# (dplyr::filter). Calculate the proportion of "Female" values. Based on this
# code, use summarise() to calculate the proportion of women in every ego's
# personal network. Hint: mean(alter.sex=="Female").
#
# (2) Subset alter.attr.all to the rows corresponding to ego_ID 53 (dplyr::filter).
# Using the resulting data frame, calculate the average closeness ($alter.clo)
# of Italian alters (i.e. $alter.nat=="Italy"). Based on this code, run
# summarise() to calculate the average closeness of Italian alters for all egos.
#
# *****
```

Chapter 5

Ego-network structure

5.1 Overview

Ego-network structure refers to the distribution of ties among alters. As usual, we first illustrate analyses on one ego-network, then replicate them on many ego-networks at once.

This chapter covers the following topics:

- Calculating measures of ego-network structure.
- R lists and how they can be used to represent many ego-networks.
- Split-apply-combine on lists with `purrr` to analyze the structure of many ego-networks at once.

5.2 Measures of ego-network structure

- Ego-network structure can be described using different measures, either at the alter level (e.g., alter centrality measures) or at the ego level (e.g., ego-network density).
- Some of these structural measures are calculated on the alter-alter network *excluding* the ego. Other measures (e.g., ego betweenness, constraint) are calculated on the ego-network *including* the ego. We will see examples of both.
- Certain measures combine information about both structure and composition: they are based on data about both the distribution of alter-alter ties and the distribution of alter attributes. An example is the average degree centrality (network structure) of alters who are family members (network composition).

- Note that whenever ego-network structure is considered (whether by itself or in combination with composition), we need data on alter-alter ties. Unlike in Chapter 4, just the alter attribute data frame will not be sufficient.
  - So in this section we need to work with the `igraph` objects containing the alter-alter tie information for our ego-networks (and possibly also incorporate alter attribute data frames if our measures are a combination of structure and composition).
- **What we do in the following code**.
  - Consider ego-network structural measures based on the distribution of alter-alter ties, with the ego excluded: density, number of components, average alter degree, maximum alter betweenness, number of isolates. Calculate these using `igraph` functions.
  - Consider structural measures that require the ego to be included in the `igraph` object: ego betweenness, constraint. Calculate these with `igraph`.
  - Calculate measures combining ego-network structure and composition: type of relationship between ego and the most between-central alter; average degree centrality of alters who are family members; density of ties among alters in certain categories (e.g., alters who live in Sri Lanka).

```
# Load packages.
library(tidyverse)
library(igraph)

# Load data.
load("./Data/data.rda")

# For structural measures we need the ego-network as an igraph.

# The data.rda file, which we loaded earlier, includes the igraph object of
# ego ID 28's network.
gr.28
```

```
## IGRAPH ff0d051 UNW- 45 259 --
## + attr: .egoID (g/n), ego_ID (g/c), name (v/c), alter_num (v/n),
## | alter.sex (v/c), alter.age.cat (v/c), alter.rel (v/c), alter.nat
## | (v/c), alter.res (v/c), alter.clo (v/n), alter.loan (v/c), alter.fam
## | (v/c), alter.age (v/n), weight (e/n)
## + edges from ff0d051 (vertex names):
##  [1] 2801--2802 2801--2803 2801--2804 2801--2805 2801--2806 2801--2807
##  [7] 2801--2808 2801--2809 2801--2810 2801--2811 2801--2812 2801--2813
## [13] 2801--2814 2801--2815 2801--2818 2801--2820 2801--2823 2801--2825
## [19] 2801--2827 2801--2828 2801--2829 2801--2831 2801--2840 2801--2841
## [25] 2802--2803 2802--2804 2802--2805 2802--2806 2802--2807 2802--2808
## + ... omitted several edges
```

```
# Let's reassign it to a new, generic object name. This makes the code more
# generic and more easily re-usable on any ego-network igraph object
# (of any ego ID).
gr <- gr.28

# Measures based only on the structure of alter-alter ties                    ----
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

# Structural characteristics of the network.

# Network density.
edge_density(gr)
```

```
## [1] 0.2616162
```

```
# Number of components.
components(gr)
```

```
## $membership
## 2801 2802 2803 2804 2805 2806 2807 2808 2809 2810 2811 2812 2813 2814 2815 2816
##    1    1    1    1    1    1    1    1    1    1    1    1    1    1    1    1
## 2817 2818 2819 2820 2821 2822 2823 2824 2825 2826 2827 2828 2829 2830 2831 2832
##    1    1    1    1    1    1    1    1    1    1    1    1    1    1    1    1
## 2833 2834 2835 2836 2837 2838 2839 2840 2841 2842 2843 2844 2845
##    1    1    1    1    1    1    1    1    1    1    1    1    1
##
## $csize
## [1] 45
##
## $no
## [1] 1
```

```
# This is a list, we only need its 3rd element (number of components).
components(gr)$no
```

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

```
# Summarization of structural characteristics of the alters.

# Average alter degree.
degree(gr) |> mean()
```

```
## [1] 11.51111
```

```
# Max alter betweenness.
betweenness(gr, weights = NA) |> max()
```

```
## [1] 257.0168
```

```
# Number of "isolate" alters.

# Check out the alter degree vector.
degree(gr)
```

```
## 2801 2802 2803 2804 2805 2806 2807 2808 2809 2810 2811 2812 2813 2814 2815 2816
##   24   17   13   12   12   13   13   12   10    9   11    8   18   14   16    9
## 2817 2818 2819 2820 2821 2822 2823 2824 2825 2826 2827 2828 2829 2830 2831 2832
##    9   10    2    2    9   18   27   13   10    8    4    3    3    3   15   16
## 2833 2834 2835 2836 2837 2838 2839 2840 2841 2842 2843 2844 2845
##   16   12   21    3   10   14   14   16   13    7   12    5   12
```

```
# Count the number of isolates, i.e. alters with degree==0
sum(degree(gr)==0)
```

```
## [1] 0
```

```
# All sorts of more complicated structural analyses can be run on an ego-network
# using igraph or statnet functions. For example, here are the results of the
# Girvan-Newman community-detection algorithm on the ego-network.
cluster_edge_betweenness(gr, weights= NA)
```

```
## IGRAPH clustering edge betweenness, groups: 10, mod: 0.41
## + groups:
##   $`1`
##    [1] "2801" "2802" "2803" "2804" "2805" "2806" "2807" "2808" "2809" "2810"
##   [11] "2811" "2812" "2840" "2841"
##
##   $`2`
##   [1] "2813" "2814" "2815" "2816" "2817" "2818" "2824" "2831"
##
##   $`3`
##   [1] "2819" "2836"
##
##   + ... omitted several groups/vertices
```

```
# What if we want to calculate the same structural measure (e.g. density) on all
# ego-networks? We can take the list of all ego-networks and run the same
# function on every list element with the purrr package in tidyverse.

# List that contains all our ego-networks.
class(gr.list)
```

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

```
length(gr.list)
```

```
## [1] 102
```

```
head(gr.list)
```

```
## $`28`
## IGRAPH ff0d051 UNW- 45 259 --
## + attr: .egoID (g/n), ego_ID (g/c), name (v/c), alter_num (v/n),
## | alter.sex (v/c), alter.age.cat (v/c), alter.rel (v/c), alter.nat
## | (v/c), alter.res (v/c), alter.clo (v/n), alter.loan (v/c), alter.fam
## | (v/c), alter.age (v/n), weight (e/n)
## + edges from ff0d051 (vertex names):
##  [1] 2801--2802 2801--2803 2801--2804 2801--2805 2801--2806 2801--2807
##  [7] 2801--2808 2801--2809 2801--2810 2801--2811 2801--2812 2801--2813
## [13] 2801--2814 2801--2815 2801--2818 2801--2820 2801--2823 2801--2825
## [19] 2801--2827 2801--2828 2801--2829 2801--2831 2801--2840 2801--2841
## [25] 2802--2803 2802--2804 2802--2805 2802--2806 2802--2807 2802--2808
## + ... omitted several edges
##
## $`29`
## IGRAPH 85d9dbc UNW- 45 202 --
## + attr: .egoID (g/n), ego_ID (g/c), name (v/c), alter_num (v/n),
## | alter.sex (v/c), alter.age.cat (v/c), alter.rel (v/c), alter.nat
## | (v/c), alter.res (v/c), alter.clo (v/n), alter.loan (v/c), alter.fam
## | (v/c), alter.age (v/n), weight (e/n)
## + edges from 85d9dbc (vertex names):
##  [1] 2901--2902 2901--2904 2901--2906 2901--2907 2901--2908 2901--2909
##  [7] 2901--2910 2901--2911 2901--2915 2901--2916 2901--2917 2901--2918
## [13] 2901--2919 2901--2927 2901--2928 2901--2929 2901--2930 2901--2931
## [19] 2901--2936 2901--2942 2901--2945 2902--2903 2902--2904 2902--2905
## [25] 2902--2906 2902--2907 2902--2908 2902--2909 2902--2910 2902--2911
## + ... omitted several edges
##
## $`33`
## IGRAPH 0674232 UNW- 45 207 --
## + attr: .egoID (g/n), ego_ID (g/c), name (v/c), alter_num (v/n),
## | alter.sex (v/c), alter.age.cat (v/c), alter.rel (v/c), alter.nat
## | (v/c), alter.res (v/c), alter.clo (v/n), alter.loan (v/c), alter.fam
## | (v/c), alter.age (v/n), weight (e/n)
## + edges from 0674232 (vertex names):
##  [1] 3301--3302 3301--3303 3301--3304 3301--3305 3301--3306 3301--3307
##  [7] 3301--3308 3301--3309 3301--3310 3301--3311 3301--3312 3301--3313
## [13] 3301--3314 3301--3315 3301--3316 3301--3317 3301--3318 3301--3319
## [19] 3301--3320 3301--3321 3301--3322 3301--3323 3302--3303 3302--3304
## [25] 3302--3305 3302--3306 3302--3307 3302--3308 3302--3309 3302--3310
## + ... omitted several edges
##
## $`35`
```

```
## IGRAPH 88571bb UNW- 45 221 --
## + attr: .egoID (g/n), ego_ID (g/c), name (v/c), alter_num (v/n),
## | alter.sex (v/c), alter.age.cat (v/c), alter.rel (v/c), alter.nat
## | (v/c), alter.res (v/c), alter.clo (v/n), alter.loan (v/c), alter.fam
## | (v/c), alter.age (v/n), weight (e/n)
## + edges from 88571bb (vertex names):
##  [1] 3501--3502 3501--3503 3501--3504 3501--3505 3501--3506 3501--3507
##  [7] 3501--3508 3501--3509 3501--3510 3501--3512 3501--3513 3501--3514
## [13] 3501--3516 3501--3517 3501--3522 3501--3523 3501--3524 3501--3525
## [19] 3501--3526 3501--3527 3501--3528 3501--3529 3501--3530 3501--3532
## [25] 3501--3533 3501--3534 3501--3535 3501--3536 3501--3540 3501--3543
## + ... omitted several edges
##
## $`39`
## IGRAPH 2de49b3 UNW- 45 92 --
## + attr: .egoID (g/n), ego_ID (g/c), name (v/c), alter_num (v/n),
## | alter.sex (v/c), alter.age.cat (v/c), alter.rel (v/c), alter.nat
## | (v/c), alter.res (v/c), alter.clo (v/n), alter.loan (v/c), alter.fam
## | (v/c), alter.age (v/n), weight (e/n)
## + edges from 2de49b3 (vertex names):
##  [1] 3901--3902 3901--3903 3901--3904 3901--3905 3901--3906 3901--3907
##  [7] 3901--3911 3901--3913 3901--3920 3901--3924 3901--3925 3901--3931
## [13] 3901--3932 3901--3933 3901--3937 3901--3938 3901--3940 3901--3941
## [19] 3902--3903 3902--3904 3902--3905 3902--3906 3902--3907 3902--3908
## [25] 3902--3909 3902--3910 3902--3913 3902--3924 3902--3925 3902--3931
## + ... omitted several edges
##
## $`40`
## IGRAPH 0c45639 UNW- 45 255 --
## + attr: .egoID (g/n), ego_ID (g/c), name (v/c), alter_num (v/n),
## | alter.sex (v/c), alter.age.cat (v/c), alter.rel (v/c), alter.nat
## | (v/c), alter.res (v/c), alter.clo (v/n), alter.loan (v/c), alter.fam
## | (v/c), alter.age (v/n), weight (e/n)
## + edges from 0c45639 (vertex names):
##  [1] 4001--4003 4001--4007 4001--4008 4001--4009 4001--4036 4001--4040
##  [7] 4001--4045 4002--4003 4002--4004 4002--4006 4002--4008 4002--4009
## [13] 4002--4010 4002--4011 4002--4012 4002--4013 4002--4014 4002--4015
## [19] 4002--4016 4002--4017 4002--4021 4002--4029 4002--4036 4002--4037
## [25] 4002--4038 4002--4040 4002--4041 4002--4042 4002--4043 4002--4044
## + ... omitted several edges
```

```
# We can use purrr::map() to run the same function on every element of the list.
purrr::map_dbl(gr.list, edge_density)
```

```
##         28         29         33         35         39         40         45
## 0.26161616 0.20404040 0.20909091 0.22323232 0.09292929 0.25757576 0.18484848
```

```
##         46         47         48         49         51         52         53
## 0.26969697 0.53737374 0.20303030 0.42828283 0.16969697 0.12929293 0.16565657
##         55         56         57         58         59         60         61
## 0.23838384 0.22828283 0.60000000 0.20505051 0.50505051 0.28888889 0.37676768
##         62         64         65         66         68         69         71
## 0.30202020 0.28282828 0.36060606 0.30909091 0.23333333 0.23636364 0.47777778
##         73         74         78         79         80         81         82
## 0.40707071 0.42525253 0.17575758 0.29292929 0.33434343 0.29797980 0.35353535
##         83         84         85         86         87         88         90
## 0.38888889 0.27575758 0.12525253 0.18989899 0.24444444 0.40101010 0.45353535
##         91         92         93         94         95         97         99
## 0.47171717 0.43535354 0.14141414 0.26767677 0.36767677 0.35050505 0.24040404
##        102        104        105        107        108        109        110
## 0.27777778 0.22929293 0.26666667 0.16262626 0.10606061 0.33535354 0.48686869
##        112        113        114        115        116        118        119
## 0.13939394 0.23636364 0.28787879 0.33232323 0.22929293 0.37676768 0.33131313
##        120        121        122        123        124        125        126
## 0.20101010 0.23838384 0.35959596 0.39090909 0.61111111 0.26464646 0.72828283
##        127        128        129        130        131        132        133
## 0.28080808 0.26262626 0.22727273 0.22020202 0.27979798 0.29090909 0.27575758
##        135        136        138        139        140        141        142
## 0.22525253 0.28686869 0.19797980 0.31515152 0.29292929 0.26868687 0.21010101
##        144        146        147        149        151        152        153
## 0.24949495 0.14747475 0.23030303 0.20707071 0.27272727 0.25151515 0.34242424
##        154        155        156        157        158        159        160
## 0.31818182 0.39393939 0.24646465 0.37777778 0.23737374 0.37979798 0.36767677
##        161        162        163        164
## 0.41010101 0.41111111 0.35454545 0.32222222
```

```
# We'll talk more about this and show more examples in the section about
# multiple ego-networks.

# Structural measures requiring ego in the network                    ----
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

# We now need the ego-network with ego included. The file data.rda, loaded earlier,
# includes this ego-network for ego ID 28.
gr.ego.28
```

```
## IGRAPH 83a192b UNW- 46 304 --
## + attr: .egoID (g/n), ego_ID (g/c), name (v/c), alter_num (v/n),
## | alter.sex (v/c), alter.age.cat (v/c), alter.rel (v/c), alter.nat
## | (v/c), alter.res (v/c), alter.clo (v/n), alter.loan (v/c), alter.fam
## | (v/c), alter.age (v/n), weight (e/n)
## + edges from 83a192b (vertex names):
##  [1] 2801--2802 2801--2803 2801--2804 2801--2805 2801--2806 2801--2807
```

```
##  [7] 2801--2808 2801--2809 2801--2810 2801--2811 2801--2812 2801--2813
## [13] 2801--2814 2801--2815 2801--2818 2801--2820 2801--2823 2801--2825
## [19] 2801--2827 2801--2828 2801--2829 2801--2831 2801--2840 2801--2841
## [25] 2802--2803 2802--2804 2802--2805 2802--2806 2802--2807 2802--2808
## + ... omitted several edges
```

```
# Let's reassign it to another, generic object name.
gr.ego <- gr.ego.28

# Ego's betweenness centrality.
# weights = NA so the function doesn't use the $weight edge attribute to
# calculate weighted betweenness.
betweenness(gr.ego, v = "ego", weights = NA)
```

```
##      ego
## 434.0271
```

```
# Ego's constraint.
constraint(gr.ego, nodes= "ego")
```

```
## ego
##  NA
```

```
# Measures combining composition and structure: From structure to composition ----
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

# Non-network attribute of alters selected based on structural characteristics.

# Type of relationship with alter with max betweenness.

# Logical index for alter with max betweenness.
ind <- betweenness(gr, weights = NA) == max(betweenness(gr, weights = NA))

# Get that alter.
V(gr)[ind]
```

```
## + 1/45 vertex, named, from ff0d051:
## [1] 2801
```

```
# Get type of relation of that alter.
V(gr)[ind]$alter.rel
```

```
## [1] "Close family"
```

```
# Note that there might be multiple alters with the same (maximum) value of
# betweenness. For that case, we'll need more complicated code (see exercise).

# Measures combining composition and structure: From composition to structure ----
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

```
# Structural characteristics of alters selected based on composition.

# Average degree of Close family.

# Vertex sequence of Close family members.
(clo.fam.vs <- V(gr)[alter.rel=="Close family"])
```

```
## + 5/45 vertices, named, from ff0d051:
## [1] 2801 2803 2804 2805 2806
```

```
# Get their average degree.
gr |>
  degree(v = clo.fam.vs) |>
  mean()
```

```
## [1] 14.8
```

```
# Count of ties between alters who live in Sri Lanka.

# First get the vertex sequence of alters who live in Sri Lanka.
(alters.sl <- V(gr)[alter.res=="Sri Lanka"])
```

```
## + 17/45 vertices, named, from ff0d051:
##  [1] 2801 2802 2803 2804 2805 2806 2807 2808 2809 2810 2811 2812 2827 2828 2829
## [16] 2840 2841
```

```
# Then get the edges among them.
E(gr)[alters.sl %--% alters.sl]
```

```
## + 88/259 edges from ff0d051 (vertex names):
##  [1] 2801--2802 2801--2803 2801--2804 2801--2805 2801--2806 2801--2807
##  [7] 2801--2808 2801--2809 2801--2810 2801--2811 2801--2812 2801--2827
## [13] 2801--2828 2801--2829 2801--2840 2801--2841 2802--2803 2802--2804
## [19] 2802--2805 2802--2806 2802--2807 2802--2808 2802--2809 2802--2810
## [25] 2802--2811 2802--2812 2802--2840 2802--2841 2803--2804 2803--2805
## [31] 2803--2806 2803--2807 2803--2808 2803--2809 2803--2810 2803--2811
## [37] 2803--2812 2803--2840 2803--2841 2804--2805 2804--2806 2804--2807
## [43] 2804--2808 2804--2809 2804--2810 2804--2811 2804--2840 2804--2841
## [49] 2805--2806 2805--2807 2805--2808 2805--2809 2805--2810 2805--2811
## [55] 2805--2840 2805--2841 2806--2807 2806--2808 2806--2809 2806--2810
## + ... omitted several edges
```

```
# How many edges are there between alters who live in Sri Lanka?
E(gr)[alters.sl %--% alters.sl] |>
  length()
```

```
## [1] 88
```

```
# Density between alters who live in Sri Lanka.

# Get subgraph of relevant alters
induced_subgraph(gr, vids = alters.sl)
```

```
## IGRAPH 780f29a UNW- 17 88 --
## + attr: .egoID (g/n), ego_ID (g/c), name (v/c), alter_num (v/n),
## | alter.sex (v/c), alter.age.cat (v/c), alter.rel (v/c), alter.nat
## | (v/c), alter.res (v/c), alter.clo (v/n), alter.loan (v/c), alter.fam
## | (v/c), alter.age (v/n), weight (e/n)
## + edges from 780f29a (vertex names):
##  [1] 2801--2802 2801--2803 2802--2803 2801--2804 2802--2804 2803--2804
##  [7] 2801--2805 2802--2805 2803--2805 2804--2805 2801--2806 2802--2806
## [13] 2803--2806 2804--2806 2805--2806 2801--2807 2802--2807 2803--2807
## [19] 2804--2807 2805--2807 2806--2807 2801--2808 2802--2808 2803--2808
## [25] 2804--2808 2805--2808 2806--2808 2807--2808 2801--2809 2802--2809
## + ... omitted several edges
```

```
# Get the density of this subgraph.
induced_subgraph(gr, vids = alters.sl) |>
  edge_density()
```

```
## [1] 0.6470588
```

5.3 R lists

- A lists is simply a **collection** of objects. It can contain any kind of object, with no restriction. A list can contain other lists.
- Lists have `list` as `type` *and* `class`.
- Data frames are a type of list. Other complex objects in R are also *stored* as lists (have `list` as `type`), although their `class` is not `list`: for example, results from statistical estimations or network community detection procedures.
- Use `str(list)` to display the types and lengths of elements in a list.
- List may be *named*, that is, have element names. You can view or assign names with the `names` function (base R) or the `set_names` function (tidyverse).
- **Three different notations to index lists**:
  1. `[ ]` notation, e.g. `my.list[3]`.
  2. `[[ ]]` notation, e.g. `my.list[[3]]` or `my.list[["element.name"]]`.
  3. The `$` notation. This only works for named lists. E.g., `list$element.name`. This is the same as the `[[ ]]` notation: `list$element.name` is the same as `list[["element.name"]]` or `list[[i]]` (where `i` is the position of the element called *element.name* in the list).

- These three indexing methods work in exactly the same way as for data frames (see Section 2.5.1).
- **What we do in the following code**.
  - Create, display, and index a list.

```
# Let's get some objects to put in a list.

# A simple numeric vector.
(num <- 1:10)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
# A matrix.
(mat <- matrix(1:4, nrow=2, ncol=2))
```

```
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

```
# A character vector.
(char <- colors()[1:5])
```

```
## [1] "white"         "aliceblue"     "antiquewhite"  "antiquewhite1"
## [5] "antiquewhite2"
```

```
# Create a list that contains all these objects.
L <- list(num, mat, char)

# Display it
L
```

```
## [[1]]
##  [1]  1  2  3  4  5  6  7  8  9 10
##
## [[2]]
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
##
## [[3]]
## [1] "white"         "aliceblue"     "antiquewhite"  "antiquewhite1"
## [5] "antiquewhite2"
```

```
# Create a named list
L <- list(numbers= num, matrix= mat, colors= char)

# Display it
L
```

```
## $numbers
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
##
## $matrix
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
##
## $colors
## [1] "white"         "aliceblue"     "antiquewhite"  "antiquewhite1"
## [5] "antiquewhite2"
```

```
# Type and class
typeof(L)
```

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

```
class(L)
```

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

```
# Extract the first element of L
# [ ] notation (result is a list containing the element).
L[1]
```

```
## $numbers
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
# [[ ]] notation (result is the element itself, no longer in a list).
L[[1]]
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
# $ notation (result is the element itself, no longer in a list).
L$numbers
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
# Name indexing.
L[["numbers"]]
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
# Types of elements in L.
str(L)
```

```
## List of 3
##  $ numbers: int [1:10] 1 2 3 4 5 6 7 8 9 10
##  $ matrix : int [1:2, 1:2] 1 2 3 4
##  $ colors : chr [1:5] "white" "aliceblue" "antiquewhite" "antiquewhite1" ...
```

5.4 Analyzing the structure of many ego-networks

- In base R, functions of the `apply` family, such as `lapply` and `sapply`, are the traditional way to take a function and apply it to every element of a list. In tidyverse, the `purrr` package does the same thing but in more efficient ways and with more readable code.
- We use `purrr` to apply the same function to each element of a list of ego-networks, that is, to each ego-network. Depending on the function, the result for each ego-network may be a single number or a more complex object (for example, a data frame).
- `purrr` provides type-stable functions, which always return the same type of output:
  - `map` always returns a list. (This is the equivalent of `lapply` in base R).
  - `map_dbl` always returns a vector of double-precision numbers. `map_int` returns a vector of integer numbers. `map_chr` returns a character vector. `map_lgl` returns a logical vector.
  - `map_dfr` returns a data frame. It assumes that you are applying an operation to each list element, which returns one data frame row for that element. It then binds together the data frame rows obtained for each element, combining them into a single data frame.
- In addition to type-stable output, the `map` functions also offer a convenient formula syntax: `~ f(.x)`, where: (1) `.x` represents each element of the input list; (2) `f` is the function to be executed on that element.
  - For example, `map(L, ~ .x * 2)` takes each element of the list `L` (represented by `.x`) and multiplies it times 2.
- The `map` functions preserve list names in their output. If we have a list of ego-networks whose names are the ego IDs, this means that ego IDs will be preserved in result lists and vectors.
- `map` is also useful when you want to run functions that take a list element as argument (e.g. an `igraph` ego-network), and return another list element as output (e.g. another `igraph` object). This is the case whenever you want to manipulate the ego-networks (for example, only keep a certain type of ties or vertices in the networks) and store the results in a new list.
- `map_dbl` is useful when you want to run functions that take a list element as argument (e.g. an `igraph` ego-network), and return a number as output. This is the case whenever you want to run a structural measure such as tie density or centralization on each network.
- **What we do in the following code**.
  - Use `purrr` functions to calculate the same structural measures on every ego-network in the data.

```
# A simple structural measure such as network density can be applied to every
# ego-network (i.e., every element of our list), and returns a scalar for each
# network. All the scalars can be put in a vector.
```

```
gr.list |>
  purrr::map_dbl(edge_density)
```

```
##         28         29         33         35         39         40         45
## 0.26161616 0.20404040 0.20909091 0.22323232 0.09292929 0.25757576 0.18484848
##         46         47         48         49         51         52         53
## 0.26969697 0.53737374 0.20303030 0.42828283 0.16969697 0.12929293 0.16565657
##         55         56         57         58         59         60         61
## 0.23838384 0.22828283 0.60000000 0.20505051 0.50505051 0.28888889 0.37676768
##         62         64         65         66         68         69         71
## 0.30202020 0.28282828 0.36060606 0.30909091 0.23333333 0.23636364 0.47777778
##         73         74         78         79         80         81         82
## 0.40707071 0.42525253 0.17575758 0.29292929 0.33434343 0.29797980 0.35353535
##         83         84         85         86         87         88         90
## 0.38888889 0.27575758 0.12525253 0.18989899 0.24444444 0.40101010 0.45353535
##         91         92         93         94         95         97         99
## 0.47171717 0.43535354 0.14141414 0.26767677 0.36767677 0.35050505 0.24040404
##        102        104        105        107        108        109        110
## 0.27777778 0.22929293 0.26666667 0.16262626 0.10606061 0.33535354 0.48686869
##        112        113        114        115        116        118        119
## 0.13939394 0.23636364 0.28787879 0.33232323 0.22929293 0.37676768 0.33131313
##        120        121        122        123        124        125        126
## 0.20101010 0.23838384 0.35959596 0.39090909 0.61111111 0.26464646 0.72828283
##        127        128        129        130        131        132        133
## 0.28080808 0.26262626 0.22727273 0.22020202 0.27979798 0.29090909 0.27575758
##        135        136        138        139        140        141        142
## 0.22525253 0.28686869 0.19797980 0.31515152 0.29292929 0.26868687 0.21010101
##        144        146        147        149        151        152        153
## 0.24949495 0.14747475 0.23030303 0.20707071 0.27272727 0.25151515 0.34242424
##        154        155        156        157        158        159        160
## 0.31818182 0.39393939 0.24646465 0.37777778 0.23737374 0.37979798 0.36767677
##        161        162        163        164
## 0.41010101 0.41111111 0.35454545 0.32222222
```

```
# Note that the vector names (taken from gr.list names) are the ego IDs.

# If you want the same result as a nice data frame with ego IDs, use enframe().
gr.list |>
  map_dbl(edge_density) |>
  enframe(name = "ego_ID", value = "density")
```

```
## # A tibble: 102 x 2
##    ego_ID density
##    <chr>    <dbl>
##  1 28       0.262
##  2 29       0.204
```

```
##  3 33      0.209
##  4 35      0.223
##  5 39      0.0929
##  6 40      0.258
##  7 45      0.185
##  8 46      0.270
##  9 47      0.537
## 10 48      0.203
## # i 92 more rows
```

```
# Same thing, with number of components in each ego network. Note the ~ .x
# syntax
gr.list |>
  map_dbl(~ components(.x)$no) |>
  enframe()
```

```
## # A tibble: 102 x 2
##    name  value
##    <chr> <dbl>
##  1 28        1
##  2 29        4
##  3 33        7
##  4 35        2
##  5 39       20
##  6 40        2
##  7 45        1
##  8 46        2
##  9 47        1
## 10 48        1
## # i 92 more rows
```

```
# With map_dfr() we can calculate multiple structural measures at once on every
# ego-network, and return the results as a single ego-level data frame.

# This assumes that we are applying an operation that returns one data frame row
# for each ego-network. For example, take one ego-network from our list.
gr <- gr.list[[10]]

# Apply an operation to that ego-network, which returns one data frame row.
tibble(dens = edge_density(gr),
       mean.deg = mean(igraph::degree(gr)),
       mean.bet = mean(igraph::betweenness(gr, weights = NA)),
       deg.centr = centr_degree(gr)$centralization)
```

```
## # A tibble: 1 x 4
##    dens mean.deg mean.bet deg.centr
##   <dbl>    <dbl>    <dbl>     <dbl>
```

```
## 1 0.203     8.93     33.8     0.365
```

```
# Now we can do the same thing but for all ego-networks at once. The result is a
# single ego-level data frame, with one row for each ego.
# Note the .id argument in map_dfr().
gr.list |>
  map_dfr(~ tibble(dens= edge_density(.x),
                   mean.deg= mean(igraph::degree(.x)),
                   mean.bet= mean(igraph::betweenness(.x, weights = NA)),
                   deg.centr= centr_degree(.x)$centralization),
          .id = "ego_ID")
```

```
## # A tibble: 102 x 5
##    ego_ID   dens mean.deg mean.bet deg.centr
##    <chr>   <dbl>    <dbl>    <dbl>     <dbl>
##  1 28     0.262     11.5     22.9      0.352
##  2 29     0.204      8.98    23.9      0.273
##  3 33     0.209      9.2      4.56     0.291
##  4 35     0.223      9.82    22.0      0.481
##  5 39     0.0929     4.09     7.18     0.316
##  6 40     0.258     11.3     21.3      0.492
##  7 45     0.185      8.13    20.6      0.679
##  8 46     0.270     11.9     19.6      0.503
##  9 47     0.537     23.6     10.5      0.372
## 10 48     0.203      8.93    33.8      0.365
## # i 92 more rows
```

```
# ***** EXERCISES
#
# (1) Get the graph for ego ID 28 from gr.list. Get the max betweenness of
# Italian alters on this graph. Based on this code, use map() to get the same
# measure that for all egos.
#
# (2) Given a personal network gr, the subgraph of close family in the personal
# network is induced_subgraph(gr, V(gr)[alter.rel=="Close family"]). Use
# purrr::map() to get the family subgraph for every personal network in gr.list,
# and put the results in a new list called gr.list.family. Use the formula
# notation (~ .x).
#
# *****
```

Chapter 6

Multilevel modeling of ego-network data

In progress.

6.1 Resources

6.1.1 Multilevel models for egocentric network data

Specific resources on multilevel modeling of egocentric/personal network data:

- Vacca [2018]: overview of multilevel modeling for egocentric/personal network data, with R demo (the code below is drawn in part from this article).
- van Duijn [2013]: overview of multilevel modeling for egocentric and sociocentric data.
- Vacca et al. [2019]: multilevel models for overlapping egocentric networks.
- McCarty et al. [2019]: Ch. 13.
- Perry et al. [2018]: Ch. 8.
- Crossley et al. [2015]: Ch. 6.

6.1.2 Multilevel modeling in general

General resources on multilevel models:

- Rasbash et al. [2008]: Extensive online course on multilevel modeling for the social sciences, including Stata and R implementation.
- Snijders and Bosker [2012]: mode in-depth statistical treatment, social science focus.
- Goldstein [2010]
- Fox and Weisberg [2018]: Ch. 7, R implementation with `lme4`.

- GLMM FAQ by Ben Bolker and others: online list of R packages and resources on linear and generalized linear mixed models.

6.2 Prepare the data

```
# Load packages.
library(tidyverse)
library(lme4)
library(car)
library(skimr)
library(janitor)
library(broom.mixed)

# Clear the workspace from all previous objects
rm(list=ls())

# Load the data
load("./Data/data.rda")

# Create data frame object for models (level-1 join)
(model.data <- left_join(alter.attr.all, ego.df, by= "ego_ID"))
```

```
## # A tibble: 4,590 x 20
##    alter_ID ego_ID alter_num alter.sex alter.age.cat alter.rel    alter.nat
##       <dbl>  <dbl>     <dbl> <fct>     <fct>         <fct>        <fct>
##  1     2801     28         1 Female    51-60         Close family Sri Lanka
##  2     2802     28         2 Male      51-60         Other family Sri Lanka
##  3     2803     28         3 Male      51-60         Close family Sri Lanka
##  4     2804     28         4 Male      60+           Close family Sri Lanka
##  5     2805     28         5 Female    41-50         Close family Sri Lanka
##  6     2806     28         6 Female    60+           Close family Sri Lanka
##  7     2807     28         7 Male      41-50         Other family Sri Lanka
##  8     2808     28         8 Female    36-40         Other family Sri Lanka
##  9     2809     28         9 Female    51-60         Other family Sri Lanka
## 10     2810     28        10 Male      60+           Other family Sri Lanka
## # i 4,580 more rows
## # i 13 more variables: alter.res <fct>, alter.clo <dbl>, alter.loan <fct>,
## #   alter.fam <fct>, alter.age <dbl>, ego.sex <fct>, ego.age <dbl>,
## #   ego.arr <dbl>, ego.edu <fct>, ego.inc <dbl>, empl <dbl>,
## #   ego.empl.bin <fct>, ego.age.cat <fct>
```

```
## Create variables to be used in multilevel models                    ====
# ============================================================================ =

# Ego-alter age homophily variable
```

```
# - - - - - - - - - - - - - - - - - -

# (TRUE if alter and ego are in the same age bracket)
model.data <- model.data |>
  mutate(alter.same.age = (alter.age.cat==ego.age.cat))

# See result
tabyl(model.data$alter.same.age)
```

```
##  model.data$alter.same.age    n    percent valid_percent
##                      FALSE 3228 0.70326797     0.7113266
##                       TRUE 1310 0.28540305     0.2886734
##                         NA   52 0.01132898            NA
```

```
# Recode: TRUE = Yes, FALSE = No
model.data <- model.data |>
  mutate(alter.same.age = as.character(alter.same.age),
         alter.same.age = fct_recode(alter.same.age,
                                     Yes = "TRUE", No = "FALSE"))

# See result
tabyl(model.data$alter.same.age)
```

```
##  model.data$alter.same.age    n    percent valid_percent
##                         No 3228 0.70326797     0.7113266
##                        Yes 1310 0.28540305     0.2886734
##                       <NA>   52 0.01132898            NA
```

```
# Centered/rescaled versions of ego and alter age
# - - - - - - - - - - - - - - - - - - - - - - - -

# This is done for easier interpretation of model coefficients
model.data <- model.data |>
  # Ego age centered around its mean and scaled by 5 (1 unit = 5 years)
  mutate(ego.age.cen = scale(ego.age, scale= 5),
         # Alter age category centered around its mean
         alter.age.cat.cen = scale(as.numeric(alter.age.cat), scale= FALSE))

# Count of family members in ego-network
# - - - - - - - - - - - - - - - - - - - - - - - -

model.data <- model.data |>
  group_by(ego_ID) |>
  mutate(net.count.fam = sum(alter.fam=="Yes", na.rm=TRUE)) |>
  ungroup()

# Center and rescale by 5 (+1 unit = 5 more family members in ego-network)
```

```
model.data <- model.data |>
  mutate(net.count.fam.cen = scale(net.count.fam, scale=5))
```

6.3 Random intercept models

```
## m1: Variance components models                                          ====
# ============================================================================ =

# Variance components model: level 1 is ties, level 2 is egos, random intercept,
# no predictor
m1 <- glmer(alter.loan ~ # Dependent variable
              (1 | ego_ID), # Intercept (1) varies in level-2 units (ego_ID)
            family = binomial("logit"), # Model class (logistic)
            data = model.data) # Data object

# View results
car::S(m1)
```

```
## Generalized linear mixed model fit by ML
## Call: glmer(formula = alter.loan ~ (1 | ego_ID), data = model.data, family =
##             binomial("logit"))
##
## Estimates of Fixed Effects:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept)  0.05376    0.09609   0.559    0.576
##
## Exponentiated Fixed Effects and Confidence Bounds:
##             Estimate     2.5 %   97.5 %
## (Intercept) 1.055234 0.8740859 1.273925
##
## Estimates of Random Effects (Covariance Components):
##  Groups Name        Std.Dev.
##  ego_ID (Intercept) 0.9001
##
## Number of obs: 4021, groups:  ego_ID, 101
##
##   logLik       df      AIC      BIC
## -2577.23        2  5158.46  5171.06
```

```
## m2: Add tie characteristics as predictors (level 1)                     ====
# ============================================================================ =

# Add alter.fam and alter.same.age as predictors

# See descriptives for the new predictors
```

```
tabyl(model.data$alter.fam)
```

```
##  model.data$alter.fam    n   percent
##                   No 3202 0.6976035
##                  Yes 1388 0.3023965
```

```
tabyl(model.data$alter.same.age)
```

```
##  model.data$alter.same.age    n    percent valid_percent
##                         No 3228 0.70326797     0.7113266
##                        Yes 1310 0.28540305     0.2886734
##                       <NA>   52 0.01132898            NA
```

```
# Estimate the model and view results
m2 <- glmer(alter.loan ~ # Dependent variable
              alter.fam + alter.same.age + # Tie characteristics
              (1 | ego_ID), # Intercept (1) varies in level-2 units (ego_ID)
            family = binomial("logit"), # Model class (logistic)
            data = model.data) # Data object
car::S(m2)
```

```
## Generalized linear mixed model fit by ML
## Call: glmer(formula = alter.loan ~ alter.fam + alter.same.age + (1 | ego_ID),
##             data = model.data, family = binomial("logit"))
##
## Estimates of Fixed Effects:
##                   Estimate Std. Error z value Pr(>|z|)
## (Intercept)       -0.21556    0.10511  -2.051   0.0403 *
## alter.famYes       1.01248    0.09284  10.906   <2e-16 ***
## alter.same.ageYes  0.20385    0.07951   2.564   0.0104 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Exponentiated Fixed Effects and Confidence Bounds:
##                   Estimate     2.5 %    97.5 %
## (Intercept)       0.806089 0.6560166 0.9904925
## alter.famYes      2.752416 2.2945237 3.3016850
## alter.same.ageYes 1.226113 1.0491868 1.4328735
##
## Estimates of Random Effects (Covariance Components):
##  Groups Name        Std.Dev.
##  ego_ID (Intercept) 0.9372
##
## Number of obs: 3972, groups:  ego_ID, 100
##
##   logLik       df      AIC      BIC
## -2478.16        4  4964.31  4989.46
```

```
## m3: Add ego characteristics as predictors (level 2)                    ====
# ============================================================================ =

# Add ego age (ego.age.cen), employment status (ego.empl.bin),
# educational level (ego.edu)

# See descriptives for the new predictors
skim_tee(model.data$ego.age.cen)
```

```
## -- Data Summary ------------------------
##                            Values
## Name                       data
## Number of rows             4590
## Number of columns          1
## _________________________
## Column type frequency:
##   numeric                  1
## _________________________
## Group variables            None
##
## -- Variable type: numeric --------------------------------------------------------
##   skim_variable n_missing complete_rate     mean   sd    p0   p25     p50  p75
## 1 V1                   45         0.990 1.09e-15 2.14 -3.85 -1.85 -0.0515 1.95
##   p100 hist
## 1 3.95
```

```
tabyl(model.data$ego.empl.bin)
```

```
##  model.data$ego.empl.bin    n   percent
##                       No  900 0.1960784
##                      Yes 3690 0.8039216
```

```
tabyl(model.data$ego.edu)
```

```
##  model.data$ego.edu    n   percent
##             Primary 1890 0.4117647
##           Secondary 2025 0.4411765
##          University  675 0.1470588
```

```
# Estimate the model and view results
m3 <- glmer(alter.loan ~ alter.fam + alter.same.age + # Tie characteristics
              ego.age.cen + ego.empl.bin + ego.edu + # Ego characteristics
              (1 | ego_ID), # Intercept (1) varies in level-2 units
            family = binomial("logit"), # Model class (logistic)
            data = model.data) # Data
car::S(m3)
```

```
## Generalized linear mixed model fit by ML
```

```
## Call: glmer(formula = alter.loan ~ alter.fam + alter.same.age + ego.age.cen +
##             ego.empl.bin + ego.edu + (1 | ego_ID), data = model.data, family =
##             binomial("logit"))
##
## Estimates of Fixed Effects:
##                   Estimate Std. Error z value Pr(>|z|)
## (Intercept)       -0.61436    0.22536  -2.726  0.00641 **
## alter.famYes       1.00698    0.09285  10.845  < 2e-16 ***
## alter.same.ageYes  0.19790    0.07950   2.489  0.01281 *
## ego.age.cen       -0.01306    0.04487  -0.291  0.77099
## ego.empl.binYes    0.07830    0.24563   0.319  0.74991
## ego.eduSecondary   0.49402    0.20962   2.357  0.01844 *
## ego.eduUniversity  0.86482    0.29947   2.888  0.00388 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Exponentiated Fixed Effects and Confidence Bounds:
##                     Estimate     2.5 %    97.5 %
## (Intercept)       0.5409892 0.3478234 0.8414306
## alter.famYes      2.7373325 2.2818749 3.2836986
## alter.same.ageYes 1.2188387 1.0429695 1.4243637
## ego.age.cen       0.9870230 0.9039204 1.0777658
## ego.empl.binYes   1.0814450 0.6682212 1.7502037
## ego.eduSecondary  1.6388905 1.0867238 2.4716141
## ego.eduUniversity 2.3745730 1.3202994 4.2706956
##
## Estimates of Random Effects (Covariance Components):
##  Groups Name        Std.Dev.
##  ego_ID (Intercept) 0.879
##
## Number of obs: 3972, groups:  ego_ID, 100
##
##   logLik       df      AIC      BIC
## -2472.68        8  4961.36  5011.66
```

```
## m4: Add alter characteristics as predictors (level 1)                    ====
# ============================================================================ =

# Add alter sex and alter age (centered)

# See descriptives for the new predictors
tabyl(model.data$alter.sex)
```

```
##  model.data$alter.sex    n   percent
##                Female 1297 0.2825708
##                  Male 3293 0.7174292
```

```
skim_tee(model.data$alter.age.cat.cen)
```

```
## -- Data Summary ------------------------
##                            Values
## Name                       data
## Number of rows             4590
## Number of columns          1
## _________________________
## Column type frequency:
##   numeric                  1
## _________________________
## Group variables            None
##
## -- Variable type: numeric -------------------------------------------------------
##   skim_variable n_missing complete_rate     mean   sd    p0   p25    p50   p75
## 1 V1                    8         0.998 1.95e-15 1.64 -3.13 -1.13 -0.126 0.874
##   p100 hist
## 1 2.87
```

```
# Estimate the model and view results
m4 <- glmer(alter.loan ~ alter.fam + alter.same.age + # Tie characteristics
              ego.age.cen + ego.empl.bin + ego.edu + # Ego characteristics
              alter.sex + alter.age.cat.cen + # Alter characteristics
              (1 | ego_ID), # Intercept (1) varies in level-2 units
            family = binomial("logit"), # Model class (logistic)
            data= model.data) # Data
car::S(m4)
```

```
## Generalized linear mixed model fit by ML
## Call: glmer(formula = alter.loan ~ alter.fam + alter.same.age + ego.age.cen +
##              ego.empl.bin + ego.edu + alter.sex + alter.age.cat.cen + (1 | ego_ID), d
##              model.data, family = binomial("logit"))
##
## Estimates of Fixed Effects:
##                   Estimate Std. Error z value Pr(>|z|)
## (Intercept)       -0.79887    0.23618  -3.383 0.000718 ***
## alter.famYes       1.04455    0.09450  11.053  < 2e-16 ***
## alter.same.ageYes  0.18047    0.07984   2.261 0.023786 *
## ego.age.cen       -0.02345    0.04544  -0.516 0.605772
## ego.empl.binYes    0.06895    0.24628   0.280 0.779486
## ego.eduSecondary   0.49039    0.21008   2.334 0.019578 *
## ego.eduUniversity  0.87968    0.30013   2.931 0.003379 **
## alter.sexMale      0.25266    0.08596   2.939 0.003289 **
## alter.age.cat.cen  0.03834    0.02475   1.549 0.121300
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
##
## Exponentiated Fixed Effects and Confidence Bounds:
##                     Estimate     2.5 %    97.5 %
## (Intercept)       0.4498365 0.2831526 0.7146425
## alter.famYes      2.8421300 2.3615731 3.4204755
## alter.same.ageYes 1.1977844 1.0242887 1.4006672
## ego.age.cen       0.9768214 0.8935895 1.0678058
## ego.empl.binYes   1.0713874 0.6611755 1.7361061
## ego.eduSecondary  1.6329480 1.0818205 2.4648443
## ego.eduUniversity 2.4101365 1.3383500 4.3402381
## alter.sexMale     1.2874404 1.0878274 1.5236817
## alter.age.cat.cen 1.0390877 0.9898894 1.0907312
##
## Estimates of Random Effects (Covariance Components):
##  Groups Name        Std.Dev.
##  ego_ID (Intercept) 0.881
##
## Number of obs: 3972, groups:  ego_ID, 100
##
##   logLik       df      AIC      BIC
## -2467.02       10  4954.04  5016.91
```

```
## m5: Add network characteristics as predictors (level 2)                ====
# ============================================================================ =

# Add count of family members in network (centered)

# See descriptives for the new predictor
skim_tee(model.data$net.count.fam.cen)
```

```
## -- Data Summary ------------------------
##                            Values
## Name                       data
## Number of rows             4590
## Number of columns          1
## _________________________
## Column type frequency:
##   numeric                  1
## _________________________
## Group variables            None
##
## -- Variable type: numeric -------------------------------------------------------
##   skim_variable n_missing complete_rate     mean   sd    p0    p25    p50   p75
## 1 V1                    0             1 2.30e-16 1.02 -1.72 -0.722 -0.122 0.478
##   p100 hist
## 1 2.88
```

```r
# Estimate model and see results
m5 <- glmer(alter.loan ~ alter.fam + alter.same.age + # Tie characteristics
              ego.age.cen + ego.empl.bin + ego.edu + # Ego characteristics
              alter.sex + alter.age.cat.cen +  # Alter characteristics
              net.count.fam.cen + # Ego-network characteristics
              (1 | ego_ID), # Intercept (1) varies in level-2 units
            family = binomial("logit"), # Model class (logistic)
            data= model.data) # Data
car::S(m5)
```

```
## Generalized linear mixed model fit by ML
## Call: glmer(formula = alter.loan ~ alter.fam + alter.same.age + ego.age.cen +
##             ego.empl.bin + ego.edu + alter.sex + alter.age.cat.cen + net.count.fam.
##             | ego_ID), data = model.data, family = binomial("logit"))
##
## Estimates of Fixed Effects:
##                    Estimate Std. Error z value Pr(>|z|)
## (Intercept)       -0.85010    0.23451  -3.625 0.000289 ***
## alter.famYes       1.06161    0.09498  11.178  < 2e-16 ***
## alter.same.ageYes  0.18265    0.07984   2.288 0.022143 *
## ego.age.cen       -0.02294    0.04473  -0.513 0.608080
## ego.empl.binYes    0.11925    0.24383   0.489 0.624801
## ego.eduSecondary   0.49636    0.20680   2.400 0.016385 *
## ego.eduUniversity  0.94460    0.29757   3.174 0.001502 **
## alter.sexMale      0.25065    0.08595   2.916 0.003544 **
## alter.age.cat.cen  0.03879    0.02474   1.568 0.116935
## net.count.fam.cen -0.17863    0.09517  -1.877 0.060523 .
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Exponentiated Fixed Effects and Confidence Bounds:
##                     Estimate     2.5 %    97.5 %
## (Intercept)       0.4273712 0.2698904 0.6767418
## alter.famYes      2.8910105 2.3999717 3.4825169
## alter.same.ageYes 1.2003998 1.0265266 1.4037236
## ego.age.cen       0.9773249 0.8952995 1.0668653
## ego.empl.binYes   1.1266465 0.6986206 1.8169121
## ego.eduSecondary  1.6427235 1.0953169 2.4637077
## ego.eduUniversity 2.5717720 1.4352830 4.6081582
## alter.sexMale     1.2848651 1.0856573 1.5206256
## alter.age.cat.cen 1.0395482 0.9903442 1.0911968
## net.count.fam.cen 0.8364157 0.6940866 1.0079307
##
## Estimates of Random Effects (Covariance Components):
##  Groups Name        Std.Dev.
```

```
##  ego_ID (Intercept) 0.8647
##
## Number of obs: 3972, groups:  ego_ID, 100
##
##   logLik       df      AIC      BIC
## -2465.28       11  4952.56  5021.72
```

```
# Results as tidy data frame
tidy(m5)
```

```
## # A tibble: 11 x 7
##    effect   group  term              estimate std.error statistic   p.value
##    <chr>    <chr>  <chr>                <dbl>     <dbl>     <dbl>     <dbl>
##  1 fixed    <NA>   (Intercept)        -0.850     0.235     -3.62   2.89e- 4
##  2 fixed    <NA>   alter.famYes         1.06      0.0950    11.2    5.25e-29
##  3 fixed    <NA>   alter.same.ageYes    0.183     0.0798     2.29   2.21e- 2
##  4 fixed    <NA>   ego.age.cen         -0.0229    0.0447    -0.513  6.08e- 1
##  5 fixed    <NA>   ego.empl.binYes      0.119     0.244      0.489  6.25e- 1
##  6 fixed    <NA>   ego.eduSecondary     0.496     0.207      2.40   1.64e- 2
##  7 fixed    <NA>   ego.eduUniversity    0.945     0.298      3.17   1.50e- 3
##  8 fixed    <NA>   alter.sexMale        0.251     0.0860     2.92   3.54e- 3
##  9 fixed    <NA>   alter.age.cat.cen    0.0388    0.0247     1.57   1.17e- 1
## 10 fixed    <NA>   net.count.fam.cen   -0.179     0.0952    -1.88   6.05e- 2
## 11 ran_pars ego_ID sd__(Intercept)      0.865    NA         NA      NA
```

```
## Plot predictor effects                                                ====
# ============================================================================ =

library(ggeffects)

# Probability of financial support as a function of alter.fam
ggpredict(m5, "alter.fam")
```

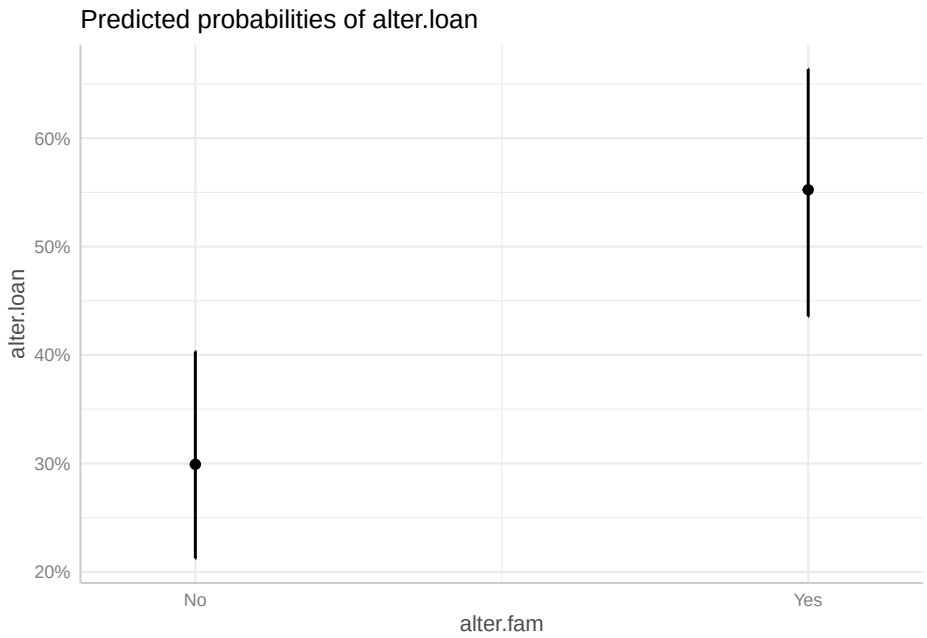
```
## # Predicted probabilities of alter.loan
##
## alter.fam | Predicted |       95% CI
## ------------------------------------
## No        |      0.30 | [0.21, 0.40]
## Yes       |      0.55 | [0.44, 0.66]
##
## Adjusted for:
## *    alter.same.age =      No
## *       ego.age.cen =   -0.03
## *      ego.empl.bin =      No
## *           ego.edu = Primary
## *         alter.sex =  Female
## * alter.age.cat.cen =   -0.05
```

```
## * net.count.fam.cen =    0.00
## *            ego_ID = 0 (population-level)
```

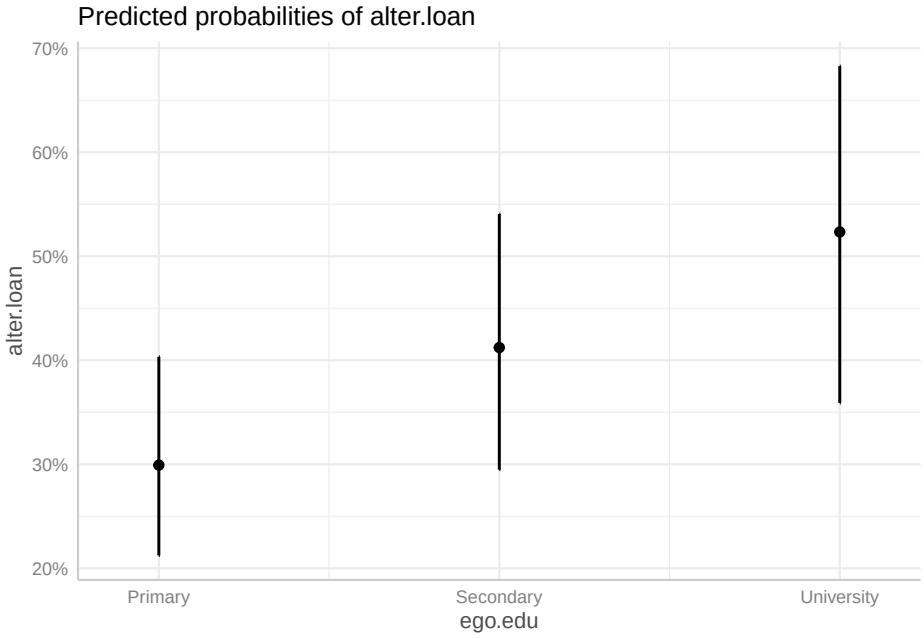
```
# You can convert it to a tidy data frame
ggpredict(m5, "alter.fam") |>
  as_tibble()
```

```
## # A tibble: 2 x 6
##   x     predicted std.error conf.low conf.high group
##   <fct>     <dbl>     <dbl>    <dbl>     <dbl> <fct>
## 1 No        0.299     0.234    0.212     0.403 1
## 2 Yes       0.552     0.239    0.436     0.664 1
```

```
# You can also plot the result
ggpredict(m5, "alter.fam") |>
  plot()
```




```
# Probability of financial support as a function of ego.edu
ggpredict(m5, "ego.edu") |>
  plot()
```



6.4 Random slope models

```
## m6: Random slope for alter.fam                                          ====
# ============================================================================ =

# Fit model
set.seed(2707)
m6 <- glmer(alter.loan ~ alter.fam + alter.same.age + # Tie characteristics
              ego.age.cen + ego.empl.bin + ego.edu + # Ego characteristics
              alter.sex + alter.age.cat.cen +  # Alter characteristics
              net.count.fam.cen + # Ego-network characteristics
              (1 + alter.fam | ego_ID), # Both intercept (1) and alter.fam
            # slppe vary in level-2 units (ego_ID)
            family = binomial("logit"), # Model class (logistic)
            data = model.data) # Data

# Re-fit with starting values from previous fit to address convergence warnings

# Get estimate values from previous fit
ss <- getME(m6, c("theta", "fixef"))

# Refit by setting ss as starting values
m6 <- glmer(alter.loan ~ alter.fam + alter.same.age + # Tie characteristics
              ego.age.cen + ego.empl.bin + ego.edu + # Ego characteristics
```

```
              alter.sex + alter.age.cat.cen +  # Alter characteristics
              net.count.fam.cen + # Ego-network characteristics
              (1 + alter.fam | ego_ID), # Both intercept (1) and alter.fam
            start= ss, #
            # slppe vary in level-2 units (ego_ID)
            family = binomial("logit"), # Model class (logistic)
            data= model.data) # Data

# View results
car::S(m6)
```

```
## Generalized linear mixed model fit by ML
## Call: glmer(formula = alter.loan ~ alter.fam + alter.same.age + ego.age.cen +
##             ego.empl.bin + ego.edu + alter.sex + alter.age.cat.cen + net.count.fam.
##             + alter.fam | ego_ID), data = model.data, family = binomial("logit"), s
##             ss)
##
## Estimates of Fixed Effects:
##                   Estimate Std. Error z value Pr(>|z|)
## (Intercept)       -0.91968    0.24605  -3.738 0.000186 ***
## alter.famYes       1.10130    0.12286   8.964  < 2e-16 ***
## alter.same.ageYes  0.18966    0.08067   2.351 0.018713 *
## ego.age.cen       -0.02932    0.04559  -0.643 0.520114
## ego.empl.binYes    0.16612    0.25192   0.659 0.509615
## ego.eduSecondary   0.52521    0.21101   2.489 0.012809 *
## ego.eduUniversity  1.00011    0.30744   3.253 0.001142 **
## alter.sexMale      0.25913    0.08691   2.982 0.002868 **
## alter.age.cat.cen  0.04229    0.02515   1.682 0.092584 .
## net.count.fam.cen -0.18543    0.09693  -1.913 0.055745 .
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Exponentiated Fixed Effects and Confidence Bounds:
##                    Estimate     2.5 %    97.5 %
## (Intercept)       0.3986475 0.2461204 0.6456995
## alter.famYes      3.0080801 2.3643604 3.8270587
## alter.same.ageYes 1.2088406 1.0320627 1.4158981
## ego.age.cen       0.9711061 0.8881039 1.0618657
## ego.empl.binYes   1.1807181 0.7206342 1.9345392
## ego.eduSecondary  1.6908172 1.1181093 2.5568725
## ego.eduUniversity 2.7185746 1.4881625 4.9662908
## alter.sexMale     1.2958056 1.0928514 1.5364505
## alter.age.cat.cen 1.0432016 0.9930326 1.0959051
## net.count.fam.cen 0.8307445 0.6870036 1.0045601
##
```

```
## Estimates of Random Effects (Covariance Components):
##  Groups Name         Std.Dev. Corr
##  ego_ID (Intercept)  0.9054
##         alter.famYes 0.6697   -0.27
##
## Number of obs: 3972, groups:  ego_ID, 100
##
##   logLik       df      AIC      BIC
## -2460.22       13  4946.44  5028.17
```

```
# Results as tidy data frame
tidy(m6)
```

```
## # A tibble: 13 x 7
##    effect   group  term                    estimate std.error statistic   p.value
##    <chr>    <chr>  <chr>                      <dbl>     <dbl>     <dbl>     <dbl>
##  1 fixed    <NA>   (Intercept)              -0.920     0.246     -3.74   1.86e- 4
##  2 fixed    <NA>   alter.famYes              1.10      0.123      8.96   3.13e-19
##  3 fixed    <NA>   alter.same.ageYes         0.190     0.0807     2.35   1.87e- 2
##  4 fixed    <NA>   ego.age.cen              -0.0293    0.0456    -0.643  5.20e- 1
##  5 fixed    <NA>   ego.empl.binYes           0.166     0.252      0.659  5.10e- 1
##  6 fixed    <NA>   ego.eduSecondary          0.525     0.211      2.49   1.28e- 2
##  7 fixed    <NA>   ego.eduUniversity         1.00      0.307      3.25   1.14e- 3
##  8 fixed    <NA>   alter.sexMale             0.259     0.0869     2.98   2.87e- 3
##  9 fixed    <NA>   alter.age.cat.cen         0.0423    0.0251     1.68   9.26e- 2
## 10 fixed    <NA>   net.count.fam.cen        -0.185     0.0969    -1.91   5.57e- 2
## 11 ran_pars ego_ID sd__(Intercept)           0.905    NA         NA      NA
## 12 ran_pars ego_ID cor__(Intercept).alte~   -0.275    NA         NA      NA
## 13 ran_pars ego_ID sd__alter.famYes          0.670    NA         NA      NA
```

6.5 Tests of significance

```
## Test significance of ego-level random intercept                    ====
# ============================================================================ =

# Test that there is significant clustering by egos, i.e. ego-level variance
# of random intercepts is significantly higher than 0. This means comparing
# the random-intercept null model (i.e. "variance components" model) to the
# single-level null model.

# First estimate the simpler, single-level null model: m0, which is nested in m1
m0 <- glm(alter.loan ~ 1, family = binomial("logit"), data= model.data)

# Then conduct a LRT comparing deviance of m0 to deviance of m1.
```

```
# Difference between deviances.
(val <- -2*logLik(m0)) - (-2*logLik(m1))
```

```
## 'log Lik.' 416.3027 (df=1)
```

```
# Compare this difference to chi-squared distribution with 1 degree of freedom.
pchisq(val, df= 1, lower.tail = FALSE)
```

```
## 'log Lik.' 0 (df=1)
```

```
# The same result is obtained using the anova() function
anova(m1, m0, refit=FALSE)
```

```
## Data: model.data
## Models:
## m0: alter.loan ~ 1
## m1: alter.loan ~ (1 | ego_ID)
##    npar    AIC    BIC  logLik deviance Chisq Df Pr(>Chisq)
## m0    1 5572.8 5579.1 -2785.4   5570.8
## m1    2 5158.5 5171.1 -2577.2   5154.5 416.3  1  < 2.2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
## Test significance of ego-level random slope for alter.fam                ====
# ============================================================================ =

# This is done with a LRT comparing the same model with random slope for alter.fam
# (m6) and without it (m5). Note that m5 is nested in m6, that's why we can
# use LRT.

# Difference between deviances.
(val <- (-2*logLik(m5)) - (-2*logLik(m6)))
```

```
## 'log Lik.' 10.12578 (df=11)
```

```
# Compare this difference to chi-squared distribution with 2 degrees of freedom.
pchisq(val, df= 2, lower.tail = FALSE)
```

```
## 'log Lik.' 0.006327254 (df=11)
```

```
# Same results with anova() function
anova(m6, m5, refit=FALSE)
```

```
## Data: model.data
## Models:
## m5: alter.loan ~ alter.fam + alter.same.age + ego.age.cen + ego.empl.bin + ego.edu +
## m6: alter.loan ~ alter.fam + alter.same.age + ego.age.cen + ego.empl.bin + ego.edu +
##    npar    AIC    BIC  logLik deviance  Chisq Df Pr(>Chisq)
## m5   11 4952.6 5021.7 -2465.3   4930.6
```

```
## m6   13 4946.4 5028.2 -2460.2   4920.4 10.126  2   0.006327 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Chapter 7

The egor package

The egor package provides tools for data import, data manipulation, network measures and visualization for egocentric network analysis. This chapter currently focuses on egocentric data import, with Section 7.1 showing how we can use egor to read raw csv files into the R data objects used throughout this workshop (those saved in the data.rda file).

More information about this package can be found in its github website and main vignette. A list of all egor vignettes is here.

7.1 Importing ego-network data

- Here we show how to use egor to import egocentric data stored in csv files. This operation requires that all needed csv files are first imported into R data frames, using standard data import functions such as read_csv(). The egor functions then take these data frames as input and return an egor object as output.
- Once we have the data in an egor object, this can be manipulated and analyzed in various and powerful ways. It can also further be converted into the other types of objects we have used throughout this workshop, such as a data frame of alter attributes or a list of igraph objects.
- In this example, we import the data with the function threefiles_to_egor. This assumes that the data are stored in three datasets: (1) a datasets with ego attributes for all egos; (2) a datasets with attributes of the alters and ego-alter ties, for all alters from all egos; (3) a datasets of alter-alter ties with a single edge list for all alters from all egos. These correspond to the three main components of egocentric data discussed in Section 3.2: (1) ego attributes; (2) alter attributes; (3) alter-alter ties.
- In some cases, these three components are combined into two or even one single dataset. egor offers functions to import data in these formats as

well: `onefile_to_egor` and `twofiles_to_egor`. See their manual page for the contents and features that the functions assume in these data.

- Finally, `egor` has functions to import egocentric data obtained from popular data collection software, such as `read_openeddi`, `read_egoweb`, and `read_egonet`.
- Regardless of the format of the original data, an `egor` object always stores the data in the three components indicated above (ego attributes, alter attributes, alter-alter ties). Each of the three components can be "activated" using the `activate` function, similar to the `tidygraph` package. After we select which of the three components we want to work on (via `activate`), an `egor` object can be easily manipulated with `dplyr` verbs.
- **What we do in the following code**.
  - Read three csv files (ego attributes, alter attributes, alter-alter ties) into R data frames.
  - Create an `egor` object from these three data frames.
  - Convert the `egor` object to a list of `igraph` ego-networks.
- The resulting data objects are those that we used in the previous chapters (saved in the `data.rda` file).

```
# Load packages
library(tidyverse)
library(igraph)
library(egor)
library(janitor)

# Import the raw csv files into R as data frames.

# Ego attributes
(ego.df <- read_csv("./Data/raw_data/ego_data.csv"))
```

```
## # A tibble: 102 x 9
##    ego_ID ego.sex ego.age ego.arr ego.edu ego.inc  empl ego.empl.bin ego.age.cat
##     <dbl> <chr>     <dbl>   <dbl> <chr>     <dbl> <dbl> <chr>        <chr>
##  1     28 Male         61    2008 Second~     350     3 Yes          60+
##  2     29 Male         38    2000 Primary     900     4 Yes          36-40
##  3     33 Male         30    2010 Primary     200     3 Yes          26-30
##  4     35 Male         25    2009 Second~    1000     3 Yes          18-25
##  5     39 Male         29    2007 Primary       0     1 No           26-30
##  6     40 Male         56    2008 Second~     950     4 Yes          51-60
##  7     45 Male         52    1975 Primary    1600     3 Yes          51-60
##  8     46 Male         35    2002 Second~    1200     4 Yes          31-35
##  9     47 Male         22    2010 Second~     700     4 Yes          18-25
## 10     48 Male         51    2007 Primary     950     4 Yes          51-60
## # i 92 more rows
```

```
# Convert all character variables to factor
# Using dplyr's across(), this code takes all ego.df variables that are character
```

```
# (where(is.character)) and applies the as.factor() function to each, converting
# it to a factor variable. The resulting data frame is re-assigned to ego.df.
ego.df <- ego.df |>
  mutate(across(where(is.character), as.factor))

# Alter attributes (all alters from all egos in same data frame).
(alter.attr.all <- read_csv("./Data/raw_data/alter_attributes.csv"))
```

```
## # A tibble: 4,590 x 12
##    alter_ID ego_ID alter_num alter.sex alter.age.cat alter.rel    alter.nat
##       <dbl>  <dbl>     <dbl> <chr>     <chr>         <chr>        <chr>
##  1     2801     28         1 Female    51-60         Close family Sri Lanka
##  2     2802     28         2 Male      51-60         Other family Sri Lanka
##  3     2803     28         3 Male      51-60         Close family Sri Lanka
##  4     2804     28         4 Male      60+           Close family Sri Lanka
##  5     2805     28         5 Female    41-50         Close family Sri Lanka
##  6     2806     28         6 Female    60+           Close family Sri Lanka
##  7     2807     28         7 Male      41-50         Other family Sri Lanka
##  8     2808     28         8 Female    36-40         Other family Sri Lanka
##  9     2809     28         9 Female    51-60         Other family Sri Lanka
## 10     2810     28        10 Male      60+           Other family Sri Lanka
## # i 4,580 more rows
## # i 5 more variables: alter.res <chr>, alter.clo <dbl>, alter.loan <chr>,
## #   alter.fam <chr>, alter.age <dbl>
```

```
# Convert all character variables to factor.
alter.attr.all <- alter.attr.all |>
  mutate(across(where(is.character), as.factor))

# To make things easier in the rest of the workshop materials, make sure the
# factors of alter gender and ego gender have the same levels (categories).

# Levels of alter.sex variable.
alter.attr.all |>
  pull(alter.sex) |>
  levels()
```

```
## [1] "Female" "Male"
```

```
# Levels of ego.sex variable: Female doesn't appear because no ego is female in
# these data.
ego.df |>
  pull(ego.sex) |>
  levels()
```

```
## [1] "Male"
```

```
ego.df |>
  tabyl(ego.sex)
```

```
##  ego.sex   n percent
##     Male 102       1
```

```
# Add the Female level to the ego.sex factor.
ego.df <- ego.df |>
  mutate(ego.sex = fct_expand(ego.sex, "Female"))

# See the result
ego.df |>
  pull(ego.sex) |>
  levels()
```

```
## [1] "Male"   "Female"
```

```
# Note that this doesn't alter the actual data in ego.df, just adds "Female" to
# the list of possible levels (categories) in ego.sex.
ego.df |>
  tabyl(ego.sex)
```

```
##  ego.sex   n percent
##     Male 102       1
##   Female   0       0
```

```
# Alter-alter ties (single edge list with all alters from all egos).
(alter_ties <- read_csv("./Data/raw_data/alter_ties.csv"))
```

```
## # A tibble: 30,064 x 4
##     from    to ego_ID weight
##    <dbl> <dbl>  <dbl>  <dbl>
##  1  2801  2802     28      1
##  2  2801  2803     28      1
##  3  2801  2804     28      1
##  4  2801  2805     28      1
##  5  2801  2806     28      1
##  6  2801  2807     28      1
##  7  2801  2808     28      1
##  8  2801  2809     28      1
##  9  2801  2810     28      1
## 10  2801  2811     28      1
## # i 30,054 more rows
```

```
# Using these 3 data frames, create an egor object.
egor.obj <- egor::threefiles_to_egor(egos= ego.df,
                                     alters.df= alter.attr.all,
                                     edges= alter_ties,
```

```
                                ID.vars = list(ego = "ego_ID",
                                               alter = "alter_ID",
                                               source = "from",
                                               target = "to"))

# The egor object can immediately be converted to a list of igraph networks.
gr.list <- egor::as_igraph(egor.obj)

# See the result
head(gr.list)
```

```
## $`28`
## IGRAPH a283c5c UNW- 45 259 --
## + attr: .egoID (g/n), name (v/c), alter_num (v/n), alter.sex (v/c),
## | alter.age.cat (v/c), alter.rel (v/c), alter.nat (v/c), alter.res
## | (v/c), alter.clo (v/n), alter.loan (v/c), alter.fam (v/c), alter.age
## | (v/n), weight (e/n)
## + edges from a283c5c (vertex names):
##  [1] 2801--2802 2801--2803 2801--2804 2801--2805 2801--2806 2801--2807
##  [7] 2801--2808 2801--2809 2801--2810 2801--2811 2801--2812 2801--2813
## [13] 2801--2814 2801--2815 2801--2818 2801--2820 2801--2823 2801--2825
## [19] 2801--2827 2801--2828 2801--2829 2801--2831 2801--2840 2801--2841
## [25] 2802--2803 2802--2804 2802--2805 2802--2806 2802--2807 2802--2808
## + ... omitted several edges
##
## $`29`
## IGRAPH 9405cb6 UNW- 45 202 --
## + attr: .egoID (g/n), name (v/c), alter_num (v/n), alter.sex (v/c),
## | alter.age.cat (v/c), alter.rel (v/c), alter.nat (v/c), alter.res
## | (v/c), alter.clo (v/n), alter.loan (v/c), alter.fam (v/c), alter.age
## | (v/n), weight (e/n)
## + edges from 9405cb6 (vertex names):
##  [1] 2901--2902 2901--2904 2901--2906 2901--2907 2901--2908 2901--2909
##  [7] 2901--2910 2901--2911 2901--2915 2901--2916 2901--2917 2901--2918
## [13] 2901--2919 2901--2927 2901--2928 2901--2929 2901--2930 2901--2931
## [19] 2901--2936 2901--2942 2901--2945 2902--2903 2902--2904 2902--2905
## [25] 2902--2906 2902--2907 2902--2908 2902--2909 2902--2910 2902--2911
## + ... omitted several edges
##
## $`33`
## IGRAPH 17a159b UNW- 45 207 --
## + attr: .egoID (g/n), name (v/c), alter_num (v/n), alter.sex (v/c),
## | alter.age.cat (v/c), alter.rel (v/c), alter.nat (v/c), alter.res
## | (v/c), alter.clo (v/n), alter.loan (v/c), alter.fam (v/c), alter.age
## | (v/n), weight (e/n)
```

```
## + edges from 17a159b (vertex names):
##  [1] 3301--3302 3301--3303 3301--3304 3301--3305 3301--3306 3301--3307
##  [7] 3301--3308 3301--3309 3301--3310 3301--3311 3301--3312 3301--3313
## [13] 3301--3314 3301--3315 3301--3316 3301--3317 3301--3318 3301--3319
## [19] 3301--3320 3301--3321 3301--3322 3301--3323 3302--3303 3302--3304
## [25] 3302--3305 3302--3306 3302--3307 3302--3308 3302--3309 3302--3310
## + ... omitted several edges
##
## $`35`
## IGRAPH 495fe16 UNW- 45 221 --
## + attr: .egoID (g/n), name (v/c), alter_num (v/n), alter.sex (v/c),
## | alter.age.cat (v/c), alter.rel (v/c), alter.nat (v/c), alter.res
## | (v/c), alter.clo (v/n), alter.loan (v/c), alter.fam (v/c), alter.age
## | (v/n), weight (e/n)
## + edges from 495fe16 (vertex names):
##  [1] 3501--3502 3501--3503 3501--3504 3501--3505 3501--3506 3501--3507
##  [7] 3501--3508 3501--3509 3501--3510 3501--3512 3501--3513 3501--3514
## [13] 3501--3516 3501--3517 3501--3522 3501--3523 3501--3524 3501--3525
## [19] 3501--3526 3501--3527 3501--3528 3501--3529 3501--3530 3501--3532
## [25] 3501--3533 3501--3534 3501--3535 3501--3536 3501--3540 3501--3543
## + ... omitted several edges
##
## $`39`
## IGRAPH 616ebb8 UNW- 45 92 --
## + attr: .egoID (g/n), name (v/c), alter_num (v/n), alter.sex (v/c),
## | alter.age.cat (v/c), alter.rel (v/c), alter.nat (v/c), alter.res
## | (v/c), alter.clo (v/n), alter.loan (v/c), alter.fam (v/c), alter.age
## | (v/n), weight (e/n)
## + edges from 616ebb8 (vertex names):
##  [1] 3901--3902 3901--3903 3901--3904 3901--3905 3901--3906 3901--3907
##  [7] 3901--3911 3901--3913 3901--3920 3901--3924 3901--3925 3901--3931
## [13] 3901--3932 3901--3933 3901--3937 3901--3938 3901--3940 3901--3941
## [19] 3902--3903 3902--3904 3902--3905 3902--3906 3902--3907 3902--3908
## [25] 3902--3909 3902--3910 3902--3913 3902--3924 3902--3925 3902--3931
## + ... omitted several edges
##
## $`40`
## IGRAPH 439c39c UNW- 45 255 --
## + attr: .egoID (g/n), name (v/c), alter_num (v/n), alter.sex (v/c),
## | alter.age.cat (v/c), alter.rel (v/c), alter.nat (v/c), alter.res
## | (v/c), alter.clo (v/n), alter.loan (v/c), alter.fam (v/c), alter.age
## | (v/n), weight (e/n)
## + edges from 439c39c (vertex names):
##  [1] 4001--4003 4001--4007 4001--4008 4001--4009 4001--4036 4001--4040
##  [7] 4001--4045 4002--4003 4002--4004 4002--4006 4002--4008 4002--4009
## [13] 4002--4010 4002--4011 4002--4012 4002--4013 4002--4014 4002--4015
```

```
## [19] 4002--4016 4002--4017 4002--4021 4002--4029 4002--4036 4002--4037
## [25] 4002--4038 4002--4040 4002--4041 4002--4042 4002--4043 4002--4044
## + ... omitted several edges
```

```
# Note that gr.list is a named list, with each element's name being the
# corresponding ego ID.
names(gr.list)
```

```
##   [1] "28"  "29"  "33"  "35"  "39"  "40"  "45"  "46"  "47"  "48"  "49"  "51"
##  [13] "52"  "53"  "55"  "56"  "57"  "58"  "59"  "60"  "61"  "62"  "64"  "65"
##  [25] "66"  "68"  "69"  "71"  "73"  "74"  "78"  "79"  "80"  "81"  "82"  "83"
##  [37] "84"  "85"  "86"  "87"  "88"  "90"  "91"  "92"  "93"  "94"  "95"  "97"
##  [49] "99"  "102" "104" "105" "107" "108" "109" "110" "112" "113" "114" "115"
##  [61] "116" "118" "119" "120" "121" "122" "123" "124" "125" "126" "127" "128"
##  [73] "129" "130" "131" "132" "133" "135" "136" "138" "139" "140" "141" "142"
##  [85] "144" "146" "147" "149" "151" "152" "153" "154" "155" "156" "157" "158"
##  [97] "159" "160" "161" "162" "163" "164"
```

```
# In gr.list, the igraph ego-networks do not include the ego node. We can create
# the same list, but now include a node for ego in each igraph ego-network.
gr.list.ego <- egor::as_igraph(egor.obj, include.ego = TRUE)

# Let's look at the same network with and without the ego.

# Without ego (gr.list).
(gr <- gr.list[["28"]])
```

```
## IGRAPH a283c5c UNW- 45 259 --
## + attr: .egoID (g/n), name (v/c), alter_num (v/n), alter.sex (v/c),
## | alter.age.cat (v/c), alter.rel (v/c), alter.nat (v/c), alter.res
## | (v/c), alter.clo (v/n), alter.loan (v/c), alter.fam (v/c), alter.age
## | (v/n), weight (e/n)
## + edges from a283c5c (vertex names):
##  [1] 2801--2802 2801--2803 2801--2804 2801--2805 2801--2806 2801--2807
##  [7] 2801--2808 2801--2809 2801--2810 2801--2811 2801--2812 2801--2813
## [13] 2801--2814 2801--2815 2801--2818 2801--2820 2801--2823 2801--2825
## [19] 2801--2827 2801--2828 2801--2829 2801--2831 2801--2840 2801--2841
## [25] 2802--2803 2802--2804 2802--2805 2802--2806 2802--2807 2802--2808
## + ... omitted several edges
```

```
# Just the vertex sequence.
V(gr)
```

```
## + 45/45 vertices, named, from a283c5c:
##  [1] 2801 2802 2803 2804 2805 2806 2807 2808 2809 2810 2811 2812 2813 2814 2815
## [16] 2816 2817 2818 2819 2820 2821 2822 2823 2824 2825 2826 2827 2828 2829 2830
## [31] 2831 2832 2833 2834 2835 2836 2837 2838 2839 2840 2841 2842 2843 2844 2845
```

```
# With ego (gr.list.ego).
(gr.ego <- gr.list.ego[["28"]])
```

```
## IGRAPH 43711af UNW- 46 304 --
## + attr: .egoID (g/n), name (v/c), alter_num (v/n), alter.sex (v/c),
## | alter.age.cat (v/c), alter.rel (v/c), alter.nat (v/c), alter.res
## | (v/c), alter.clo (v/n), alter.loan (v/c), alter.fam (v/c), alter.age
## | (v/n), weight (e/n)
## + edges from 43711af (vertex names):
##  [1] 2801--2802 2801--2803 2801--2804 2801--2805 2801--2806 2801--2807
##  [7] 2801--2808 2801--2809 2801--2810 2801--2811 2801--2812 2801--2813
## [13] 2801--2814 2801--2815 2801--2818 2801--2820 2801--2823 2801--2825
## [19] 2801--2827 2801--2828 2801--2829 2801--2831 2801--2840 2801--2841
## [25] 2802--2803 2802--2804 2802--2805 2802--2806 2802--2807 2802--2808
## + ... omitted several edges
```

```
# Just the vertex sequence. Note the name of the last node.
V(gr.ego)
```

```
## + 46/46 vertices, named, from 43711af:
##  [1] 2801 2802 2803 2804 2805 2806 2807 2808 2809 2810 2811 2812 2813 2814 2815
## [16] 2816 2817 2818 2819 2820 2821 2822 2823 2824 2825 2826 2827 2828 2829 2830
## [31] 2831 2832 2833 2834 2835 2836 2837 2838 2839 2840 2841 2842 2843 2844 2845
## [46] ego
```

```
# In our ego-network data, alter-alter ties have weights (1 or 2, see the data
# codebook).
E(gr)$weight
```

```
##   [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 2 1 1 1 1 1 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [38] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [75] 1 1 1 1 1 2 1 1 1 1 1 1 1 1 1 1 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 1 2 1
## [112] 1 1 1 1 1 1 1 1 2 1 1 1 1 1 1 1 1 1 1 2 1 1 1 1 1 1 1 1 2 1 1 1 1 2 1 1 1
## [149] 1 1 1 1 1 1 1 1 2 2 1 1 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 1 1 1 1 1 1
## [186] 1 1 1 1 1 1 1 1 1 1 1 1 2 1 1 1 1 1 1 1 1 1 1 2 1 1 1 1 2 1 1 2 1 1 1 1 1 1
## [223] 1 1 1 1 1 1 1 2 1 1 1 1 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 1 1
```

```
# These weights are obviously not defined for ego-alter ties. So in the
# ego-network with the ego included, the tie weights are NA for ego-alter ties.
E(gr.ego)$weight
```

```
##   [1]  1  1  1  1  1  1  1  1  1  1  1  1  1  1  2  1  1  1  1  1  2  1  1  1  1
##  [26]  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
##  [51]  1  1  1  1  1  1  1  2  2  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
##  [76]  1  1  1  1  2  1  1  1  1  1  1  1  1  1  1  2  2  1  1  1  1  1  1  1  1
## [101]  1  1  1  1  1  1  1  2  1  2  1  1  1  1  1  1  1  1  1  2  1  1  1  1  1
## [126]  1  1  1  1  1  2  1  1  1  1  1  1  1  1  2  1  1  1  1  2  1  1  1  1  1
## [151]  1  1  1  1  1  2  2  1  1  2  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
```

```
## [176]  1  1  1  2  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  2  1  1  1
## [201]  1  1  1  1  1  1  1  2  1  1  1  1  2  1  1  2  1  1  1  1  1  1  1  1
## [226]  1  1  1  1  2  1  1  1  1  2  1  1  1  1  1  1  1  1  1  1  1  1  1  1
## [251]  1  1  1  1  1  1  2  1  1 NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
## [276] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
## [301] NA NA NA NA
```

```
E(gr.ego)[inc("ego")]$weight
```

```
## Warning: 'inc' is deprecated.
## Use '.inc' instead.
## See help("Deprecated")
```

```
##  [1] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
## [26] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
```

```
# This may create problems in certain functions, for example the plot.igraph()
# function. So let's replace those NA's with a distinctive weight value for
# ego-alter ties: 3.
E(gr.ego)$weight <- E(gr.ego)$weight |>
  replace_na(3)

# See the result
E(gr.ego)$weight
```

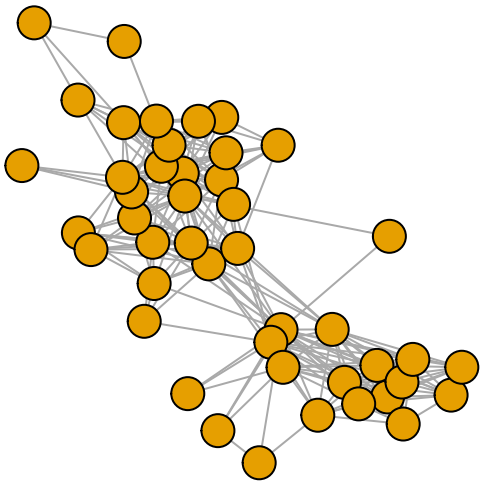
```
##   [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 2 1 1 1 1 1 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [38] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [75] 1 1 1 1 1 2 1 1 1 1 1 1 1 1 1 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 2 1 2 1
## [112] 1 1 1 1 1 1 1 1 2 1 1 1 1 1 1 1 1 1 2 1 1 1 1 1 1 1 1 2 1 1 1 2 1 1 1
## [149] 1 1 1 1 1 1 1 2 2 1 1 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 1 1 1 1 1 1
## [186] 1 1 1 1 1 1 1 1 1 1 1 2 1 1 1 1 1 1 1 1 1 1 2 1 1 1 1 2 1 1 2 1 1 1 1 1 1
## [223] 1 1 1 1 1 1 1 2 1 1 1 1 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 1 1
## [260] 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
## [297] 3 3 3 3 3 3 3 3
```

```
E(gr.ego)[inc("ego")]$weight
```

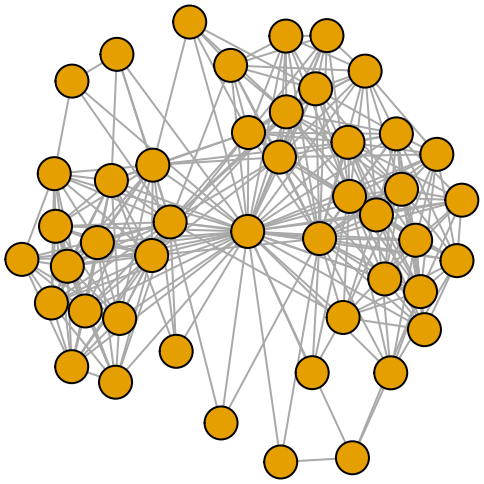
```
## Warning: 'inc' is deprecated.
## Use '.inc' instead.
## See help("Deprecated")
```

```
##  [1] 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
## [39] 3 3 3 3 3 3 3
```

```
# Let's now plot the two ego-networks.

# Without ego.
plot(gr, vertex.label = NA)
```



```
# With ego.
plot(gr.ego, vertex.label = NA)
```



```
# Let's do the weight replacement operation above (NA replaced by 3 for ego-alter
# tie weights) for all the ego-networks in gr.list.ego.
for (i in seq_along(gr.list.ego)) {
  E(gr.ego)$weight <- E(gr.ego)$weight |>
    replace_na(3)
}

# The operation above can be done with tidyverse map() functions as well. First
# define the function operating on a single graph.
weight_replace <- function(gr) {
  E(gr)$weight <- E(gr)$weight |>
    replace_na(3)
```

```
  # Return value
  gr
}

# Then apply the function to all graphs in the list.
new.list <- gr.list.ego |>
  map(weight_replace)

# Sometimes it's useful to have ego ID as a graph attribute of the ego-network
# in addition to having it as name of the ego-network's element in the list.
# So let's add ego ID as the $ego_ID graph attribute in each list element.

# List of graphs without the ego.
for (i in seq_along(gr.list)) {
  gr.list[[i]]$ego_ID <- names(gr.list)[[i]]
}

# List of graphs with the ego.
for (i in seq_along(gr.list.ego)) {
  gr.list.ego[[i]]$ego_ID <- names(gr.list.ego)[[i]]
}

# See the result
gr.list[["28"]]
```

```
## IGRAPH a283c5c UNW- 45 259 --
## + attr: .egoID (g/n), ego_ID (g/c), name (v/c), alter_num (v/n),
## | alter.sex (v/c), alter.age.cat (v/c), alter.rel (v/c), alter.nat
## | (v/c), alter.res (v/c), alter.clo (v/n), alter.loan (v/c), alter.fam
## | (v/c), alter.age (v/n), weight (e/n)
## + edges from a283c5c (vertex names):
##  [1] 2801--2802 2801--2803 2801--2804 2801--2805 2801--2806 2801--2807
##  [7] 2801--2808 2801--2809 2801--2810 2801--2811 2801--2812 2801--2813
## [13] 2801--2814 2801--2815 2801--2818 2801--2820 2801--2823 2801--2825
## [19] 2801--2827 2801--2828 2801--2829 2801--2831 2801--2840 2801--2841
## [25] 2802--2803 2802--2804 2802--2805 2802--2806 2802--2807 2802--2808
## + ... omitted several edges
```

```
gr.list[["28"]]$ego_ID
```

```
## [1] "28"
```

```
# Extract alter attributes and graph for one ego (ego ID 28)

# Alter attribute data frame.
alter.attr.28 <- alter.attr.all |>
```

```
  filter(ego_ID==28)

# Ego-network graph.
gr.28 <- gr.list[["28"]]

# Ego-network graph, with ego included.
gr.ego.28 <- gr.list.ego[["28"]]

# Save all data to file.
save(ego.df, alter.attr.all, gr.list, alter.attr.28, gr.28, gr.ego.28, file="./Data/dat
```

7.2 Analyzing and visualizing ego-network data

In progress.

`egor` also simplifies some of the analysis operations that we have seen in the previous chapters, such as the calculation of certain compositional and structural measures.

Chapter 8

Supplementary topics

8.1 More R programming topics

8.1.1 Types and classes of objects

This is a quick summary of the basics about **types** and **classes** of objects in R.

- Three functions are used to know what kind of object you are dealing with in R: `class()`, `mode()`, and `typeof()`.
- For most purposes, you only need to know what the **class** of an object is. This is returned by `class()`. The class of an object determines what R functions you can or cannot run on that object, and how functions will behave when you run them on the object. In particular, if the function has a `method` for a specific class A of objects, it will use that method whenever an object of class A is given as its argument.
- `typeof()` and `mode()` return the **type** and **mode** of an object, respectively. Although they refer to slightly different classifications of objects, type and mode give essentially the same kind of information — the type of data structure in which the object is stored, also called the R "internal type" or "storage mode". For example, an object can be internally stored in R as double-precision numbers, integer numbers, or character strings.
  - You should prefer `typeof()` over `mode()`. `mode()` refers to the old S classification of types and is mostly used for S compatibility.
- While most times all you need to know is the `class` of an object, there are a few cases in which knowing the `type` is useful too. For example, you may want to know the `type` of a matrix object (whose `class` is always `matrix`) to check if the values in the matrix are being stored as numbers or character strings (that will affect the result of some functions).

- Main classes/types of objects
  - `numeric`: Numerical data (integer, real or complex numbers).
  - `logical`: `TRUE/FALSE` data.
  - `character`: String data.
  - `factor`: Categorical data, that is, integer numbers with string labels attached. May be *unordered* factors (nominal data) or *ordered* factors (ordinal data).
- Special and complex classes/types
  - `list`: A collection of elements of any type, including numeric, character, logical (see Section 5.3).
  - `data.frame`: A dataset. In R, a data frame is a special kind of list (its type is `list` but its class is `data.frame`), where each variable (column) is a list element (see Section 2.3.4)
  - `matrix`: Matrix values can be numeric, character, logical etc. So an object can have `matrix` as class and `numeric`, `character` or `logical` as type. While data frames can contain variables of different type (e.g. a character variable and a numeric variable), matrices can only contain values of *one* type.
  - Functions (more on this in Section 8.1.2).
  - Expressions.
  - Formulas.
  - Other objects: Statistical results (e.g. linear model estimates), dendrograms, graphics objects, etc.
- Relevant functions
  - `class()`, `typeof()` and `mode()`, as discussed above.
  - `is.`*type* functions verify that an object is in a specific type or class: e.g. `is.numeric(x)`, `is.character(x)` (they return `TRUE` or `FALSE`).
  - `as.`*type* functions convert objects between types or classes: e.g. `as.numeric()`, `as.character()`. If the conversion is impossible, the result is `NA`: e.g. `as.numeric("abc")` returns `NA`.

```
# A numeric vector of integers.
n <- 1:100

# Let's check the class and type.
class(n)
```

```
## [1] "integer"
```

```
typeof(n)
```

```
## [1] "integer"
```

```
# A character object.
(char <- c("a", "b", "c", "d", "e", "f"))
```

```
## [1] "a" "b" "c" "d" "e" "f"
```

```
# Class and type.
class(char)
```

```
## [1] "character"
```

```
typeof(char)
```

```
## [1] "character"
```

```
# Let's put n in a matrix.
(M <- matrix(n, nrow=10, ncol=10))
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
##  [1,]    1   11   21   31   41   51   61   71   81    91
##  [2,]    2   12   22   32   42   52   62   72   82    92
##  [3,]    3   13   23   33   43   53   63   73   83    93
##  [4,]    4   14   24   34   44   54   64   74   84    94
##  [5,]    5   15   25   35   45   55   65   75   85    95
##  [6,]    6   16   26   36   46   56   66   76   86    96
##  [7,]    7   17   27   37   47   57   67   77   87    97
##  [8,]    8   18   28   38   48   58   68   78   88    98
##  [9,]    9   19   29   39   49   59   69   79   89    99
## [10,]   10   20   30   40   50   60   70   80   90   100
```

```
# Class/type of this object.
class(M)
```

```
## [1] "matrix" "array"
```

```
# Type and mode tell us that this is an *integer* matrix.
typeof(M)
```

```
## [1] "integer"
```

```
# There are character and logical matrices too.
char
```

```
## [1] "a" "b" "c" "d" "e" "f"
```

```
(C <- matrix(char, nrow=3, ncol= 2))
```

```
##      [,1] [,2]
## [1,] "a"  "d"
## [2,] "b"  "e"
## [3,] "c"  "f"
```

```
# Class and type.
class(C)
```

```
## [1] "matrix" "array"
```

```
typeof(C)
```

```
## [1] "character"
```

```
# Notice that a matrix can contain numbers but still be stored as character.
(M <- matrix(c("1", "2", "3", "4"), nrow=2, ncol=2))
```

```
##      [,1] [,2]
## [1,] "1"  "3"
## [2,] "2"  "4"
```

```
class(M)
```

```
## [1] "matrix" "array"
```

```
typeof(M)
```

```
## [1] "character"
```

```
# Let's convert "char" to factor.
char
```

```
## [1] "a" "b" "c" "d" "e" "f"
```

```
(char <- as.factor(char))
```

```
## [1] a b c d e f
## Levels: a b c d e f
```

```
# This means that now char is not just a collection of strings, it is a
# categorical variable in R's mind: it is a collection of numbers with character
# labels attached.

# Compare the different behavior of as.numeric(): char as character...
(char <- c("a", "b", "c", "d", "e", "f"))
```

```
## [1] "a" "b" "c" "d" "e" "f"
```

```
# Convert to numeric
as.numeric(char)
```

```
## Warning: NAs introduced by coercion
```

```
## [1] NA NA NA NA NA NA
```

```
# ...versus char as factor.
char <- c("a", "b", "c", "d", "e", "f")
(char <- as.factor(char))
```

```
## [1] a b c d e f
## Levels: a b c d e f
```

```
as.numeric(char)
```

```
## [1] 1 2 3 4 5 6
```

```
# char is a different object in R's mind when it's character vs when
# it's factor. Characters can't be converted to numbers,
# but factors can.

# ***** EXERCISE:
# Using "as.factor()", convert the "educ" object to factor. Assign the
# result to "educ.fc". Print both "educ" and "educ.fc". Do you notice any
# difference in the way they are printed? Convert "educ" and "educ.fc" to
# numeric. Why is the result different?
# *****
```

8.1.2 Writing your own R functions

- One of the most powerful tools in R is the ability to **write your own functions**.
- A function is a **piece of code** that operates on one or multiple **arguments** (the *input*), and returns an *output* (the function **value** in R terminology). Everything that happens in R is done by a function.
- Many R functions have **default values** for their arguments: if you don't specify the argument's value, the function will use the default.
- Once you write a function and define its arguments, you can run that function on any argument values you want — provided that the function code actually works on those argument values. For example, if a function takes an `igraph` object as an argument, you'll be able to run that function on any network you like, provided that the network is an `igraph` object. If your network is a `network` object (created by a `statnet` function), the function will likely return an error.
- R functions, combined with functional and summarization methods such as those seen in Sections 4.3 and 5.4, are the best way to run exactly the same code on many different objects (for example, many different ego-networks). Functions are **crucial for code reproducibility** in R. If you write functions, you won't need to re-write (copy and paste) the same code over and over again — you just write it once in the function, then run the function any time and on any arguments you need. This yields clearer, shorter, more readable code with less errors.
- New functions are also commonly used to **redefine existing functions** by pre-setting the value of specific arguments. For example, if you want all your plots to have `red` as color, you can take R's existing plotting function `plot`, and wrap it in a new function that always executes `plot` with the argument `col="red"`. Your function would be something like `my.plot <- function(...) {plot(..., col="red")}` (examples below).
- **Tips and tricks** with functions:

  - `stopifnot()` is useful to check that function arguments are of the type that was intended by the function author. It stops the function if a certain condition is not met by a function argument (e.g. argument is *not* an `igraph` object, if the function was written for `igraph` objects).
  - `return()` allows you to explicitly set the output that the function will return (clearer code). It is also used to stop function execution earlier under certain conditions. Note: If you don't use `return()`, the function value (output) is the last object that is printed at the end of the function code.
  - `if` is a flow control tool that is frequently used within functions: it specifies what the function should do `if` a certain condition is met at one point.
  - First think particular, then generalize. When you want to write a function, it's a good idea to first try the code on a "real", specific existing object in your workspace. If the code does what you want on that object, you can then wrap it into a general function to be run on any similar object (see examples in the code below).
- In ego-network analysis, we often want to write functions that calculate measures of ego-network composition and structure that we are interested in. These functions will have different arguments depending on whether they look at network composition or structure:
  - Functions that calculate compositional measures typically require just the alter attribute data frame as argument.
  - Functions that calculate structural measures require the ego-network as an `igraph` object (or `network` object in `statnet`), which store the alter-alter tie information.
  - Functions that calculate measures combining composition and structure require the `igraph` object, with alter attributes incorporated in the object as vertex attributes.
- **What we do in the following code**.
  - Demonstrate the basics of R functions using simple examples.
  - Consider some of the compositional and structural measures we calculated in previous chapters, and convert them to general functions that can be applied to any ego-network: average closeness of alters; proportion of female alters; proportion of alters of the same gender as ego; function that returns multiple compositional measures into a data frame; maximum alter betweenness; number of components and isolates in the ego-network; tie density between alters who live in Sri Lanka.

```
# Basics of R functions
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

# Any piece of code you can write and run in R, you can also put in a function.
```

```r
# Let's write a trivial function that takes its argument and multiplies it by 2.
times2 <- function(x) {
  x*2
}

# Now we can run the function on any argument.
times2(x= 3)
```

```
## [1] 6
```

```r
times2(x= 10)
```

```
## [1] 20
```

```r
times2(50)
```

```
## [1] 100
```

```r
# A function that takes its argument and prints a sentence with it:
myoutput <- function(word) {
  print(paste("My output is", word))
}

# Let's run the function.
myoutput("cat")
```

```
## [1] "My output is cat"
```

```r
myoutput(word= "table")
```

```
## [1] "My output is table"
```

```r
myoutput("any word here")
```

```
## [1] "My output is any word here"
```

```r
# Not necessarily a useful function...

# Note that the function output is the last object that is printed at the end
# of the function code.
times2 <- function(x) {
  y <- x*2
  y
}
times2(x=4)
```

```
## [1] 8
```

```r
# If nothing is printed, then the function returns nothing.
times2 <- function(x) {
  y <- x*2
```

```
}
times2(x=4)

# A function will return an error if it's executed on arguments that are not
# suitable for the code inside the function. E.g., R can't multiply "a" by 2...
times2 <- function(x) {
  x*2
}
times2(x= "a")
```

```
## Error in x * 2: non-numeric argument to binary operator
```

```
# Let's then specify that the function's argument must be numeric.
times2 <- function(x) {
  stopifnot(is.numeric(x))
  x*2
}

# Let's try it now.
times2(x= "a")
```

```
## Error in times2(x = "a"): is.numeric(x) is not TRUE
```

```
# This still throws and error, but it makes the error clearer to the user and
# it immidiately indicates where the problem is.

# Using if, we can also re-write the function so that it returns NA with a
# warning if its argument is not numeric -- instead of just stopping with an
# error.
times2 <- function(x) {
  # If x is not numeric
  if(!is.numeric(x)) {
    # Give the warning
    warning("Your argument is not numeric!", call. = FALSE)
    # Return missing value
    return(NA)
    # Otherwise, return x*2
  } else {
    return(x*2)
  }
}

# Try the function
times2(2)
```

```
## [1] 4
```

```
times2("a")
```

```
## Warning: Your argument is not numeric!
```

```
## [1] NA
```

```
# Writing functions that calculate ego-network measures
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

# Now that we know how to write functions, we can convert any ego-network
# measure we calculated above to a general function.

# This is a function that takes the alter-level data frame for an ego, and
# calculates average closeness of alters.
avg_alt_clo <- function(df) {
  mean(df$alter.clo, na.rm = TRUE)
}

# Let's run this function on our alter data frame for ego 28
avg_alt_clo(df= alter.attr.28)
```

```
## [1] 4.1
```

```
# Function that takes the alter-level data frame for an ego, and calculates
# the proportion of female alters.
prop_fem <- function(df) {
  mean(df$alter.sex == "Female")
}

# Apply the function to our data frame
prop_fem(df= alter.attr.28)
```

```
## [1] 0.1777778
```

```
# Function that takes two arguments: the alter-level data frame for an ego, and
# the ego-level data frame; and calculates the proportion of alters of the same
# sex as ego.
same_sex <- function(df, ego.df) {

  # Join alter and ego attribute data.
  df <- left_join(df, ego.df, by= "ego_ID")

  # Return the measure
  mean(df$alter.sex == df$ego.sex)
}

# Apply the function
same_sex(df= alter.attr.28, ego.df= ego.df)
```

```
## [1] 0.8222222
```

```
# Function that takes the alter-level data frame for an ego, and calculates
# multiple compositional measures.
comp_meas <- function(df) {
  df |>
    summarise(
      mean.clo = mean(alter.clo, na.rm=TRUE),
      prop.fem = mean(alter.sex=="Female"),
      count.nat.slk = sum(alter.nat=="Sri Lanka"),
      count.nat.ita = sum(alter.nat=="Italy"),
      count.nat.oth = sum(alter.nat=="Other")
    )
}

# Let's apply the function.
comp_meas(alter.attr.28)
```

```
## # A tibble: 1 x 5
##   mean.clo prop.fem count.nat.slk count.nat.ita count.nat.oth
##      <dbl>    <dbl>         <int>         <int>         <int>
## 1      4.1    0.178            43             0             2
```

```
# Function that takes an ego-network as igraph object, and calculates max alter
# betweenness.
max_betw <- function(x) x |> igraph::betweenness(weights = NA) |> max()

# Apply to our graph.
max_betw(x = gr)
```

```
## [1] 257.0168
```

```
# Function that takes an ego-network as igraph object, and calculates the number
# of components and the number of isolates.
comp_iso <- function(x) {

  # Get N components
  N.comp <- igraph::components(x)$no

  # Get N isolates
  N.iso <- sum(igraph::degree(x)==0)

  # Return output
  tibble::tibble(N.comp = N.comp, N.iso= N.iso)
}

# Apply the function
comp_iso(gr)
```

```
## # A tibble: 1 x 2
##   N.comp N.iso
##    <int> <int>
## 1      1     0
```

```
# Function that takes an ego-network as igraph object, and calculates the
# density of ties among alters who live in Sri Lanka.
sl_dens <- function(x) {

  # Get the vertex sequence of alters who live in Sri Lanka.
  alters.sl <- V(x)[alter.res=="Sri Lanka"]

  # Get the subgraph of those alters
  sl.subg <- induced_subgraph(x, vids = alters.sl)

  # Return the density of this subgraph.
  edge_density(sl.subg)
}

# Run the function on our ego-network of ego 28.
sl_dens(x= gr)
```

```
## [1] 0.6470588
```

```
# ***** EXERCISE:
# Write a function that takes an ego's edge list as argument. The function
# returns the number of "maybe" edges in the corresponding personal network (use
# data frame indexing and sum()). Run this function on the edge lists of ego ID
# 47, 53 and 162. HINT: You should first try the code on one edge list from the
# list elist.all.list An edge is "uncertain" if the corresponding value in
# the edge list is 2. Remember that the ego IDs are in names(elist.all.list).
# *****
```

8.2 Importing ego-network data with `igraph` and `tidyverse`

This section demonstrates how to import ego-network data into R "manually", without using `egor`. The `egor` packages has simplified many of these operations (see Section 7).

8.2.1 Importing data for one ego-network

- Different functions exist to create `igraph` objects from data manually typed in R or (more typically) imported from external data sources – such

as adjacency matrices or edge lists in an external csv file. Some of these functions are `graph_from_edgelist`, `graph_from_adjacency_matrix`, `graph_from_data_frame`.

- The following code assumes a common format for egocentric data (which, however, may not be the one you have): (1) a dataset with ego attributes for all egos; (2) a dataset with alter attributes for all alters from all egos; (3) a dataset of alter-alter ties with a single edge list for all alters from all egos.
- **What we do in the following code**.
  - Import an ego-network from 2 csv files: the csv file with the edge list (alter-alter ties) and the csv file with alter attributes for one ego. Convert it to igraph object with alter attributes: `graph_from_data_frame()`.

```
# Alter attribute and tie data for one ego-network
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

# Create graph with ego-network for ego 28

# Get the ego-network of ego ID 28 from an external edge list and a data set
# with alter attributes.

# Read in the edge list for ego 28. This is a personal network edge list.
(elist.28 <- read_csv("./Data/raw_data/alter_ties_028.csv"))
```

```
## Rows: 259 Columns: 3
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## dbl (3): from, to, weight
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
## # A tibble: 259 x 3
##     from    to weight
##    <dbl> <dbl>  <dbl>
##  1  2801  2802      1
##  2  2801  2803      1
##  3  2801  2804      1
##  4  2801  2805      1
##  5  2801  2806      1
##  6  2801  2807      1
##  7  2801  2808      1
##  8  2801  2809      1
##  9  2801  2810      1
## 10  2801  2811      1
## # i 249 more rows
```

```
# Read in the alter attribute data for ego 28.
(alter.attr.28 <- read_csv("./Data/raw_data/alter_attributes_028.csv"))
```

```
## Rows: 45 Columns: 12
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (7): alter.sex, alter.age.cat, alter.rel, alter.nat, alter.res, alter.lo...
## dbl (5): alter_ID, ego_ID, alter_num, alter.clo, alter.age
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
## # A tibble: 45 x 12
##    alter_ID ego_ID alter_num alter.sex alter.age.cat alter.rel    alter.nat
##       <dbl>  <dbl>     <dbl> <chr>     <chr>         <chr>        <chr>
##  1     2801     28         1 Female    51-60         Close family Sri Lanka
##  2     2802     28         2 Male      51-60         Other family Sri Lanka
##  3     2803     28         3 Male      51-60         Close family Sri Lanka
##  4     2804     28         4 Male      60+           Close family Sri Lanka
##  5     2805     28         5 Female    41-50         Close family Sri Lanka
##  6     2806     28         6 Female    60+           Close family Sri Lanka
##  7     2807     28         7 Male      41-50         Other family Sri Lanka
##  8     2808     28         8 Female    36-40         Other family Sri Lanka
##  9     2809     28         9 Female    51-60         Other family Sri Lanka
## 10     2810     28        10 Male      60+           Other family Sri Lanka
## # i 35 more rows
## # i 5 more variables: alter.res <chr>, alter.clo <dbl>, alter.loan <chr>,
## #   alter.fam <chr>, alter.age <dbl>
```

```
# Convert to graph with vertex attributes.
gr.28 <- graph_from_data_frame(d= elist.28, vertices= alter.attr.28, directed= FALSE)

# Note directed= FALSE

# A network object can be created from an adjacency matrix too: import adjacency
# matrix for ego-network of ego 28.

# Read the adjacency matrix.
adj.28 <- read.csv("./Data/raw_data/adj_028.csv", row.names=1) |>
  as.matrix()
# Set column names the same as rownames
colnames(adj.28) <- rownames(adj.28)

# Convert to igraph
graph_from_adjacency_matrix(adj.28, mode="upper")
```

```
## IGRAPH 05d3a61 UN-- 45 283 --
```

```
## + attr: name (v/c)
## + edges from 05d3a61 (vertex names):
##  [1] 2801--2802 2801--2803 2801--2804 2801--2805 2801--2806 2801--2807
##  [7] 2801--2808 2801--2809 2801--2810 2801--2811 2801--2812 2801--2813
## [13] 2801--2814 2801--2815 2801--2818 2801--2818 2801--2820 2801--2823
## [19] 2801--2825 2801--2827 2801--2828 2801--2829 2801--2829 2801--2831
## [25] 2801--2840 2801--2841 2802--2803 2802--2804 2802--2805 2802--2806
## [31] 2802--2807 2802--2808 2802--2809 2802--2810 2802--2811 2802--2812
## [37] 2802--2813 2802--2815 2802--2823 2802--2831 2802--2840 2802--2841
## [43] 2803--2804 2803--2805 2803--2806 2803--2807 2803--2808 2803--2809
## + ... omitted several edges
```

8.2.2 Importing data on many ego-networks as data frames and lists

- **Lists** are very convenient objects to store multiple pieces of data, such as multiple adjacency matrices or multiple alter attribute data frames, one for each ego. Once we have all our data pieces (e.g. matrices or data frames) into a single list, we can very easily do two things:
  - Run a function on *every* piece in batch (see Sections 4.3 and 5.4).
  - Run a function on *all* the pieces together, using `do.call()`. `do.call(function, list)` executes `function` using all the elements of `list` as its arguments.
- In many cases, attributes of all alters from all the egos are stored in a single tabular dataset, e.g., a single csv file (such as `alter_attributes.csv` in our data). Alter tie data can also be stored in a single csv file, for example as an edge list with an additional column indicating the ego ID (such as `alter_ties.csv` in our data). Data of this type can be easily imported into R using the `read_csv` function in tidyverse.
- In other cases, alter attributes or alter-alter edge lists are stored in different `csv` files, one for each ego. These can also be imported using `read_csv()` within a `for` loop.
- Regardless of the external csv data source, once the data are in R we might want to store them as a single *data frame* with all alters from all egos, or as a *list* of separate data frames, one for each ego.
  - While a single data frame is more compact, there are scenarios in which coding is simpler if data for different egos are located in different data frames, and all these data frames are gathered in a list.
- In R it is easy to switch between a single data frame (pooling alters from all egos) and a list (with alters from each ego in a separate data frame)
  - Separate data frames, say `df1` and `df2`, can be appended into a single data frame using `bind_rows`, which stacks their rows together: e.g. `bind_rows(df1, df2)`. If we have 100 egos, we will want to bind 100 data frames. If the data frames are part of a list, say `df.list`, this can be done simply as `bind_rows(df.list)`. The result is a single data frame in which all data from all egos are stacked together

by rows. Note that this requires that all data frames in `df.list` have the same variables with the same names.
  - A single data frame, say `df` can be split into a list of data frames (one for each ego) by running `split(df, f= df$egoID)` – where `df$egoID` is the variable with the ego IDs. This splits the single data frame into a list of separate data frames, one for each value of `egoID` (i.e., one for each ego).
- When you store ego data frames into a list, ego IDs can be conveniently saved as **names of list elements** (set via `names` or `set_names`). If you're using *numeric* ego IDs, you should be careful to the order in which list elements are stored in the list. Depending on the sequence of your numeric ego IDs, the 53rd data frame in the list is not necessarily the data frame of ego ID=53. Thus, you need to keep in mind the difference between `list[[53]]` (numeric indexing: get the 53rd list element) and `list[["53"]]` (name indexing: get the list element named "53").
- **What we do in the following code**.
  - Import the alter attribute data frame and edge list for all alters from `alter_attributes.csv` and `alter_ties.csv`.
  - Demonstrate how alter attribute data frames and edge lists can be easily split by ego (using `split`) or combined into a single data frame with all alters (using `bind_rows`).

```
# All ego and alter attributes
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

# Import all ego and alter attributes

# Import ego-level data.
(ego.df <- read_csv("./Data/raw_data/ego_data.csv"))
```

```
## Rows: 102 Columns: 9
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (4): ego.sex, ego.edu, ego.empl.bin, ego.age.cat
## dbl (5): ego_ID, ego.age, ego.arr, ego.inc, empl
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
## # A tibble: 102 x 9
##    ego_ID ego.sex ego.age ego.arr ego.edu ego.inc  empl ego.empl.bin ego.age.cat
##     <dbl> <chr>     <dbl>   <dbl> <chr>     <dbl> <dbl> <chr>        <chr>
##  1     28 Male         61    2008 Second~     350     3 Yes          60+
##  2     29 Male         38    2000 Primary     900     4 Yes          36-40
##  3     33 Male         30    2010 Primary     200     3 Yes          26-30
##  4     35 Male         25    2009 Second~    1000     3 Yes          18-25
##  5     39 Male         29    2007 Primary       0     1 No           26-30
```

```
##  6     40 Male         56    2008 Second~     950     4 Yes          51-60
##  7     45 Male         52    1975 Primary    1600     3 Yes          51-60
##  8     46 Male         35    2002 Second~    1200     4 Yes          31-35
##  9     47 Male         22    2010 Second~     700     4 Yes          18-25
## 10     48 Male         51    2007 Primary     950     4 Yes          51-60
## # i 92 more rows
```

```
# Import the csv file with attributes of all alters from all egos.
(alter.attr.all <- read_csv("./Data/raw_data/alter_attributes.csv"))
```

```
## Rows: 4590 Columns: 12
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (7): alter.sex, alter.age.cat, alter.rel, alter.nat, alter.res, alter.lo...
## dbl (5): alter_ID, ego_ID, alter_num, alter.clo, alter.age
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
## # A tibble: 4,590 x 12
##    alter_ID ego_ID alter_num alter.sex alter.age.cat alter.rel    alter.nat
##       <dbl>  <dbl>     <dbl> <chr>     <chr>         <chr>        <chr>
##  1     2801     28         1 Female    51-60         Close family Sri Lanka
##  2     2802     28         2 Male      51-60         Other family Sri Lanka
##  3     2803     28         3 Male      51-60         Close family Sri Lanka
##  4     2804     28         4 Male      60+           Close family Sri Lanka
##  5     2805     28         5 Female    41-50         Close family Sri Lanka
##  6     2806     28         6 Female    60+           Close family Sri Lanka
##  7     2807     28         7 Male      41-50         Other family Sri Lanka
##  8     2808     28         8 Female    36-40         Other family Sri Lanka
##  9     2809     28         9 Female    51-60         Other family Sri Lanka
## 10     2810     28        10 Male      60+           Other family Sri Lanka
## # i 4,580 more rows
## # i 5 more variables: alter.res <chr>, alter.clo <dbl>, alter.loan <chr>,
## #   alter.fam <chr>, alter.age <dbl>
```

```
# Note the number of rows (number of all alters).

# We can easily split a single data frame into a list of data frames, with one
# separate element for each ego.
alter.attr.list <- split(alter.attr.all, f= alter.attr.all$ego_ID)

# The result is a list of N alter attribute data frames, one for each ego.
length(alter.attr.list)
```

```
## [1] 102
```

```
# The values of "f" in split() (the split factor) are preserved as list names.
# In our case these are the ego IDs.
names(alter.attr.list)
```

```
##   [1] "28"  "29"  "33"  "35"  "39"  "40"  "45"  "46"  "47"  "48"  "49"  "51"
##  [13] "52"  "53"  "55"  "56"  "57"  "58"  "59"  "60"  "61"  "62"  "64"  "65"
##  [25] "66"  "68"  "69"  "71"  "73"  "74"  "78"  "79"  "80"  "81"  "82"  "83"
##  [37] "84"  "85"  "86"  "87"  "88"  "90"  "91"  "92"  "93"  "94"  "95"  "97"
##  [49] "99"  "102" "104" "105" "107" "108" "109" "110" "112" "113" "114" "115"
##  [61] "116" "118" "119" "120" "121" "122" "123" "124" "125" "126" "127" "128"
##  [73] "129" "130" "131" "132" "133" "135" "136" "138" "139" "140" "141" "142"
##  [85] "144" "146" "147" "149" "151" "152" "153" "154" "155" "156" "157" "158"
##  [97] "159" "160" "161" "162" "163" "164"
```

```
# The list element is the alter attribute data for a single ego. Get the
# attribute data frame for ego ID 94.
alter.attr.list[["94"]]
```

```
## # A tibble: 45 x 12
##    alter_ID ego_ID alter_num alter.sex alter.age.cat alter.rel    alter.nat
##       <dbl>  <dbl>     <dbl> <chr>     <chr>         <chr>        <chr>
##  1     9401     94         1 Male      60+           Close family Sri Lanka
##  2     9402     94         2 Female    60+           Close family Sri Lanka
##  3     9403     94         3 Female    26-30         Close family Sri Lanka
##  4     9404     94         4 Male      26-30         Close family Sri Lanka
##  5     9405     94         5 Male      31-35         Close family Sri Lanka
##  6     9406     94         6 Female    60+           Other family Sri Lanka
##  7     9407     94         7 Male      51-60         Other family Sri Lanka
##  8     9408     94         8 Female    36-40         Other family Sri Lanka
##  9     9409     94         9 Female    51-60         Other family Sri Lanka
## 10     9410     94        10 Male      18-25         Other family Sri Lanka
## # i 35 more rows
## # i 5 more variables: alter.res <chr>, alter.clo <dbl>, alter.loan <chr>,
## #   alter.fam <chr>, alter.age <dbl>
```

```
# The reverse of splitting is also easy to do: if we have N data frames in a
# list, we can easily bind them into a single data frame by stacking their rows
# together.
alter.attr.all.2 <- bind_rows(alter.attr.list)

# The result is a single alter-level data frame where each row is an alter, and
# alters from all egos are in the same data frame.
alter.attr.all.2
```

```
## # A tibble: 4,590 x 12
##    alter_ID ego_ID alter_num alter.sex alter.age.cat alter.rel    alter.nat
##       <dbl>  <dbl>     <dbl> <chr>     <chr>         <chr>        <chr>
```

```
##  1     2801     28         1 Female    51-60         Close family Sri Lanka
##  2     2802     28         2 Male      51-60         Other family Sri Lanka
##  3     2803     28         3 Male      51-60         Close family Sri Lanka
##  4     2804     28         4 Male      60+           Close family Sri Lanka
##  5     2805     28         5 Female    41-50         Close family Sri Lanka
##  6     2806     28         6 Female    60+           Close family Sri Lanka
##  7     2807     28         7 Male      41-50         Other family Sri Lanka
##  8     2808     28         8 Female    36-40         Other family Sri Lanka
##  9     2809     28         9 Female    51-60         Other family Sri Lanka
## 10     2810     28        10 Male      60+           Other family Sri Lanka
## # i 4,580 more rows
## # i 5 more variables: alter.res <chr>, alter.clo <dbl>, alter.loan <chr>,
## #   alter.fam <chr>, alter.age <dbl>
```

```
# 45 alters x 102 egos = 4590 alters.

# Edge list with alter-alter ties
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

# Let's import the csv containing a single edge list with alters from all
# egos.
(elist.all <- read_csv("./Data/raw_data/alter_ties.csv"))
```

```
## Rows: 30064 Columns: 4
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## dbl (4): from, to, ego_ID, weight
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
## # A tibble: 30,064 x 4
##     from    to ego_ID weight
##    <dbl> <dbl>  <dbl>  <dbl>
##  1  2801  2802     28      1
##  2  2801  2803     28      1
##  3  2801  2804     28      1
##  4  2801  2805     28      1
##  5  2801  2806     28      1
##  6  2801  2807     28      1
##  7  2801  2808     28      1
##  8  2801  2809     28      1
##  9  2801  2810     28      1
## 10  2801  2811     28      1
## # i 30,054 more rows
```

```
# Exactly the same split by ego ID can be done for the edge list with
# alter-alter ties.
elist.all.list <- split(elist.all, f= elist.all$ego_ID)

# See the result.
length(elist.all.list)
```

```
## [1] 102
```

```
names(elist.all.list)
```

```
##   [1] "28"  "29"  "33"  "35"  "39"  "40"  "45"  "46"  "47"  "48"  "49"  "51"
##  [13] "52"  "53"  "55"  "56"  "57"  "58"  "59"  "60"  "61"  "62"  "64"  "65"
##  [25] "66"  "68"  "69"  "71"  "73"  "74"  "78"  "79"  "80"  "81"  "82"  "83"
##  [37] "84"  "85"  "86"  "87"  "88"  "90"  "91"  "92"  "93"  "94"  "95"  "97"
##  [49] "99"  "102" "104" "105" "107" "108" "109" "110" "112" "113" "114" "115"
##  [61] "116" "118" "119" "120" "121" "122" "123" "124" "125" "126" "127" "128"
##  [73] "129" "130" "131" "132" "133" "135" "136" "138" "139" "140" "141" "142"
##  [85] "144" "146" "147" "149" "151" "152" "153" "154" "155" "156" "157" "158"
##  [97] "159" "160" "161" "162" "163" "164"
```

```
elist.all.list[["94"]]
```

```
## # A tibble: 265 x 4
##     from    to ego_ID weight
##    <dbl> <dbl>  <dbl>  <dbl>
##  1  9401  9402     94      1
##  2  9401  9403     94      1
##  3  9401  9404     94      1
##  4  9401  9405     94      1
##  5  9401  9406     94      1
##  6  9401  9407     94      1
##  7  9401  9408     94      1
##  8  9401  9409     94      1
##  9  9401  9410     94      1
## 10  9401  9411     94      1
## # i 255 more rows
```

8.2.3 Creating lists of ego-networks as `igraph` objects with `purrr`

- `purrr::map()` is useful when you want to run functions that take a list element as argument, e.g. an `igraph` ego network; and return another list element as output, e.g. another `igraph` object. This is the case whenever you want to manipulate the ego networks (e.g. only keep a certain type of ties or vertices), and store the results in a new list.
- In certain cases, you want to run a function that takes *two* arguments from

two different lists, say `L1` and `L2`, for each ego. For example, for each ego you may want to take its alter-alter edge list (argument 1, from `L1`), take its alter attributes (argument 2, from `L2`), and put them together into an `igraph` object.
  - The function `purrr::map2()` does this with very little code. For `i`= 1, ..., I, `map2()` takes the `i`-th element of `L1` and the `i`-th element of `L2`, and runs your function I times, one for each pair of arguments `L1[[i]]` and `L2[[i]]`.
  - Note that `L1` and `L2` need to be two "parallel" lists: the first element of `L1` corresponds to (will be combined with) the first element of `L2`; the second element of `L1` with the second element of `L2`, ..., the I-the element of `L1` with the I-th element of `L2`.
  - `map2()` has a formula notation too: `map2(L1, L2, ~ f(.x, .y)`, where `.x` refers to each element of `L1` and `.y` refers to each element of `L2`.

- **What we do in the following code**.
  - Use `map()` to convert each ego's edge list to an `igraph` object.
  - Use `map2()` to convert each ego's edge list *and* alter attribute data frame to an `igraph` object.

```
# Creating a list of ego-networks as igraph objects
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

# We have a list of alter edge lists, with each list element being the edge list
# of one ego.
length(elist.all.list)
```

```
## [1] 102
```

```
names(elist.all.list)
```

```
##   [1] "28"  "29"  "33"  "35"  "39"  "40"  "45"  "46"  "47"  "48"  "49"  "51"
##  [13] "52"  "53"  "55"  "56"  "57"  "58"  "59"  "60"  "61"  "62"  "64"  "65"
##  [25] "66"  "68"  "69"  "71"  "73"  "74"  "78"  "79"  "80"  "81"  "82"  "83"
##  [37] "84"  "85"  "86"  "87"  "88"  "90"  "91"  "92"  "93"  "94"  "95"  "97"
##  [49] "99"  "102" "104" "105" "107" "108" "109" "110" "112" "113" "114" "115"
##  [61] "116" "118" "119" "120" "121" "122" "123" "124" "125" "126" "127" "128"
##  [73] "129" "130" "131" "132" "133" "135" "136" "138" "139" "140" "141" "142"
##  [85] "144" "146" "147" "149" "151" "152" "153" "154" "155" "156" "157" "158"
##  [97] "159" "160" "161" "162" "163" "164"
```

```
elist.all.list[[1]]
```

```
## # A tibble: 259 x 4
##     from    to ego_ID weight
##    <dbl> <dbl>  <dbl>  <dbl>
##  1  2801  2802     28      1
##  2  2801  2803     28      1
```

```
##  3  2801  2804     28      1
##  4  2801  2805     28      1
##  5  2801  2806     28      1
##  6  2801  2807     28      1
##  7  2801  2808     28      1
##  8  2801  2809     28      1
##  9  2801  2810     28      1
## 10  2801  2811     28      1
## # i 249 more rows
```

```
# We saw that we can use igraph to convert one ego-network edge list
# to an igraph object. Let's do it for the first element (i.e., the first ego)
# of elist.all.list.
elist <- elist.all.list[[1]]
(gr <- graph_from_data_frame(d= elist, directed= FALSE))
```

```
## IGRAPH 71d9c02 UNW- 45 259 --
## + attr: name (v/c), ego_ID (e/n), weight (e/n)
## + edges from 71d9c02 (vertex names):
##  [1] 2801--2802 2801--2803 2801--2804 2801--2805 2801--2806 2801--2807
##  [7] 2801--2808 2801--2809 2801--2810 2801--2811 2801--2812 2801--2813
## [13] 2801--2814 2801--2815 2801--2818 2801--2820 2801--2823 2801--2825
## [19] 2801--2827 2801--2828 2801--2829 2801--2831 2801--2840 2801--2841
## [25] 2802--2803 2802--2804 2802--2805 2802--2806 2802--2807 2802--2808
## [31] 2802--2809 2802--2810 2802--2811 2802--2812 2802--2813 2802--2815
## [37] 2802--2823 2802--2831 2802--2840 2802--2841 2803--2804 2803--2805
## [43] 2803--2806 2803--2807 2803--2808 2803--2809 2803--2810 2803--2811
## + ... omitted several edges
```

```
# With map(), we can do the same thing for all egos at once.
gr.list <- purrr::map(.x= elist.all.list,
                      ~ graph_from_data_frame(d= .x, directed= FALSE))

# The result is a list of graphs, with each element corresponding to an ego.

# The list has as many elements as egos.
length(gr.list)
```

```
## [1] 102
```

```
# The element names are the ego IDs (taken from the names of elist.all.list).
names(gr.list)
```

```
##   [1] "28"  "29"  "33"  "35"  "39"  "40"  "45"  "46"  "47"  "48"  "49"  "51"
##  [13] "52"  "53"  "55"  "56"  "57"  "58"  "59"  "60"  "61"  "62"  "64"  "65"
##  [25] "66"  "68"  "69"  "71"  "73"  "74"  "78"  "79"  "80"  "81"  "82"  "83"
##  [37] "84"  "85"  "86"  "87"  "88"  "90"  "91"  "92"  "93"  "94"  "95"  "97"
##  [49] "99"  "102" "104" "105" "107" "108" "109" "110" "112" "113" "114" "115"
```

```
##  [61] "116" "118" "119" "120" "121" "122" "123" "124" "125" "126" "127" "128"
##  [73] "129" "130" "131" "132" "133" "135" "136" "138" "139" "140" "141" "142"
##  [85] "144" "146" "147" "149" "151" "152" "153" "154" "155" "156" "157" "158"
##  [97] "159" "160" "161" "162" "163" "164"
```

```
# We can extract each ego's graph.
# By position.
gr.list[[1]]
```

```
## IGRAPH 76794df UNW- 45 259 --
## + attr: name (v/c), ego_ID (e/n), weight (e/n)
## + edges from 76794df (vertex names):
##  [1] 2801--2802 2801--2803 2801--2804 2801--2805 2801--2806 2801--2807
##  [7] 2801--2808 2801--2809 2801--2810 2801--2811 2801--2812 2801--2813
## [13] 2801--2814 2801--2815 2801--2818 2801--2820 2801--2823 2801--2825
## [19] 2801--2827 2801--2828 2801--2829 2801--2831 2801--2840 2801--2841
## [25] 2802--2803 2802--2804 2802--2805 2802--2806 2802--2807 2802--2808
## [31] 2802--2809 2802--2810 2802--2811 2802--2812 2802--2813 2802--2815
## [37] 2802--2823 2802--2831 2802--2840 2802--2841 2803--2804 2803--2805
## [43] 2803--2806 2803--2807 2803--2808 2803--2809 2803--2810 2803--2811
## + ... omitted several edges
```

```
# By name (ego ID).
gr.list[["94"]]
```

```
## IGRAPH 3fee39f UNW- 42 265 --
## + attr: name (v/c), ego_ID (e/n), weight (e/n)
## + edges from 3fee39f (vertex names):
##  [1] 9401--9402 9401--9403 9401--9404 9401--9405 9401--9406 9401--9407
##  [7] 9401--9408 9401--9409 9401--9410 9401--9411 9401--9412 9401--9413
## [13] 9401--9414 9401--9415 9401--9438 9401--9443 9401--9444 9402--9403
## [19] 9402--9404 9402--9405 9402--9406 9402--9407 9402--9408 9402--9409
## [25] 9402--9410 9402--9411 9402--9412 9402--9413 9402--9414 9402--9415
## [31] 9402--9418 9402--9419 9402--9435 9402--9438 9402--9443 9402--9444
## [37] 9403--9404 9403--9405 9403--9406 9403--9407 9403--9408 9403--9409
## [43] 9403--9410 9403--9411 9403--9412 9403--9413 9403--9414 9403--9415
## + ... omitted several edges
```

```
# If we have a function that takes two arguments, we can use map2(). For
# example, graph_from_data_frame() can take two arguments: the ego's edge list
# and the ego's alter attribute data frame.

# We now have a list of edge lists, one for each ego.
length(elist.all.list)
```

```
## [1] 102
```

```
# And a lsist of alter attribute data frames, one for each ego.
length(alter.attr.list)
```

```
## [1] 102
```

```
# Let's run graph_from_data_frame() for the first ego, i.e., using the first
# element of each of the two lists.
(gr <- graph_from_data_frame(d= elist.all.list[[1]],
                             vertices= alter.attr.list[[1]], directed= FALSE))
```

```
## IGRAPH 27e212c UNW- 45 259 --
## + attr: name (v/c), ego_ID (v/n), alter_num (v/n), alter.sex (v/c),
## | alter.age.cat (v/c), alter.rel (v/c), alter.nat (v/c), alter.res
## | (v/c), alter.clo (v/n), alter.loan (v/c), alter.fam (v/c), alter.age
## | (v/n), ego_ID (e/n), weight (e/n)
## + edges from 27e212c (vertex names):
##  [1] 2801--2802 2801--2803 2801--2804 2801--2805 2801--2806 2801--2807
##  [7] 2801--2808 2801--2809 2801--2810 2801--2811 2801--2812 2801--2813
## [13] 2801--2814 2801--2815 2801--2818 2801--2820 2801--2823 2801--2825
## [19] 2801--2827 2801--2828 2801--2829 2801--2831 2801--2840 2801--2841
## [25] 2802--2803 2802--2804 2802--2805 2802--2806 2802--2807 2802--2808
## + ... omitted several edges
```

```
# We can now run the same function for all egos at once.
gr.list <- purrr::map2(.x= elist.all.list, .y= alter.attr.list,
                       ~ graph_from_data_frame(d= .x, vertices= .y, directed= FALSE))

# The result is again a list, with each element corresponding to one ego.
```

8.3 More operations with igraph

- More examples of network visualization with `igraph`.
- Add vertices to `igraph` objects, for example to add the ego node.

```
# Example of network visualization with igraph
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

# Get graph of one ego-network.
gr <- gr.28

# Vertex attribute with alter's country of residence.
V(gr)$alter.res
```

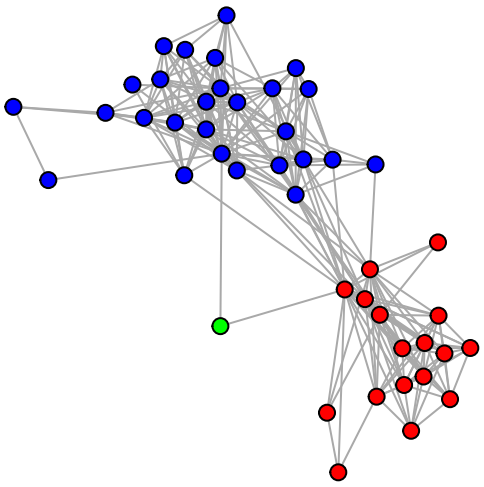
```
##  [1] "Sri Lanka" "Sri Lanka" "Sri Lanka" "Sri Lanka" "Sri Lanka" "Sri Lanka"
##  [7] "Sri Lanka" "Sri Lanka" "Sri Lanka" "Sri Lanka" "Sri Lanka" "Sri Lanka"
## [13] "Italy"     "Italy"     "Italy"     "Italy"     "Italy"     "Italy"
## [19] "Italy"     "Other"     "Italy"     "Italy"     "Italy"     "Italy"
```

```
## [25] "Italy"     "Italy"     "Sri Lanka" "Sri Lanka" "Sri Lanka" "Italy"
## [31] "Italy"     "Italy"     "Italy"     "Italy"     "Italy"     "Italy"
## [37] "Italy"     "Italy"     "Italy"     "Sri Lanka" "Sri Lanka" "Italy"
## [43] "Italy"     "Italy"     "Italy"
```

```
# Set red as default vertex color.
color <- rep("red", vcount(gr))
color
```

```
##  [1] "red" "red" "red" "red" "red" "red" "red" "red" "red" "red" "red" "red"
## [13] "red" "red" "red" "red" "red" "red" "red" "red" "red" "red" "red" "red"
## [25] "red" "red" "red" "red" "red" "red" "red" "red" "red" "red" "red" "red"
## [37] "red" "red" "red" "red" "red" "red" "red" "red" "red"
```

```
# Set color=="blue" when vertex (alter) lives in Italy.
color[V(gr)$alter.res=="Italy"] <- "blue"

# Set color=="green" when vertex (alter) lives in Other country.
color[V(gr)$alter.res=="Other"] <- "green"

# Plot using the "color" vector we just created.
set.seed(607)
plot(gr, vertex.label= NA, vertex.size=7, vertex.color= color)
```



```
# Add ego node to igraph object with igraph operations
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

# Use the + operator to add a named vertex, e.g. ego, to a graph.
gr.ego <- gr + "ego"

# The graph has now one more node. It has the same number of edges because we
# haven't added edges between ego and alters.
```

```
vcount(gr)
```

```
## [1] 45
```

```
vcount(gr.ego)
```

```
## [1] 46
```

```
ecount(gr)
```

```
## [1] 259
```

```
ecount(gr.ego)
```

```
## [1] 259
```

```
# Ego is the 46th vertex.
V(gr.ego)
```

```
## + 46/46 vertices, named, from 7292e66:
##  [1] 2801 2802 2803 2804 2805 2806 2807 2808 2809 2810 2811 2812 2813 2814 2815
## [16] 2816 2817 2818 2819 2820 2821 2822 2823 2824 2825 2826 2827 2828 2829 2830
## [31] 2831 2832 2833 2834 2835 2836 2837 2838 2839 2840 2841 2842 2843 2844 2845
## [46] ego
```

```
# Let's add all edges between ego and the alters.

# Ego's adjacency row. Note that the last (46th) cell is the diagonal cell
# between ego and itself.
gr.ego["ego",]
```

```
## 2801 2802 2803 2804 2805 2806 2807 2808 2809 2810 2811 2812 2813 2814 2815 2816
##    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0
## 2817 2818 2819 2820 2821 2822 2823 2824 2825 2826 2827 2828 2829 2830 2831 2832
##    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0
## 2833 2834 2835 2836 2837 2838 2839 2840 2841 2842 2843 2844 2845  ego
##    0    0    0    0    0    0    0    0    0    0    0    0    0    0
```

```
# Let's select ego's row without the 46th cell: i.e., only cells (ties) between
# ego and the 45 alters.
gr.ego["ego", -46]
```

```
## 2801 2802 2803 2804 2805 2806 2807 2808 2809 2810 2811 2812 2813 2814 2815 2816
##    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0
## 2817 2818 2819 2820 2821 2822 2823 2824 2825 2826 2827 2828 2829 2830 2831 2832
##    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0
## 2833 2834 2835 2836 2837 2838 2839 2840 2841 2842 2843 2844 2845
##    0    0    0    0    0    0    0    0    0    0    0    0    0
```

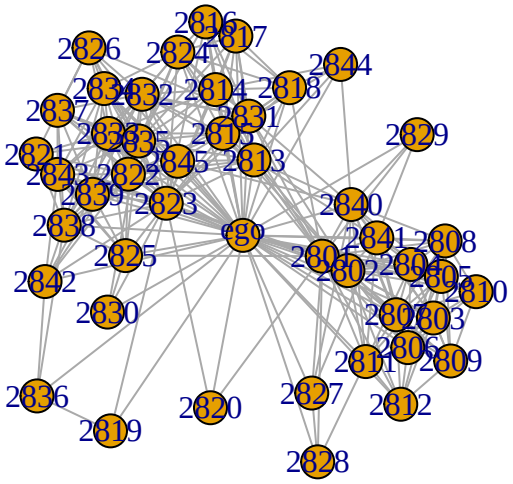
```
# Note that we can write this in a more general way, which works for any graph
# regardless of the number of vertices. Creating general code that applies to
# different circumstances is essential when writing R functions.
gr.ego["ego",-vcount(gr.ego)]
```

```
## 2801 2802 2803 2804 2805 2806 2807 2808 2809 2810 2811 2812 2813 2814 2815 2816
##    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0
## 2817 2818 2819 2820 2821 2822 2823 2824 2825 2826 2827 2828 2829 2830 2831 2832
##    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0
## 2833 2834 2835 2836 2837 2838 2839 2840 2841 2842 2843 2844 2845
##    0    0    0    0    0    0    0    0    0    0    0    0    0
```

```
# Let's set ego's entire adjacency row with alters to 1's. This adds edges
# between ego and all alters.
gr.ego["ego",-vcount(gr.ego)] <- 1

# We now have 45 more edges.
ecount(gr.ego)
```

```
## [1] 304
```

```
# See what the ego-network looks like now.
set.seed(123)
plot(gr.ego)
```



8.4 The statnet suite of packages

- statnet is a collection of different packages. The statnet packages we'll use in the following code are network and sna.
- While igraph represents networks as objects of class igraph, statnet

represents networks as objects of class `network`.

- If your network is already stored in an `igraph` object but you want to analyze it with `statnet`, you can easily convert the `igraph` object into a `network` object using the `intergraph` package (function `asNetwork()`). The conversion preserves edge (adjacency) data, vertex attributes, edge attributes, and network attributes.
- Of course, you can also create a `network` object from external data, such as an adjacency matrix stored in a csv file.
- Similar to `igraph`, `statnet` allows you to import, set and view edge, vertex, and network attributes in addition to edge (adjacency) data.
- `igraph` and `statnet` have overlapping function names. That is, there are `igraph` functions and `statnet` functions that have the same name, although they are obviously different functions. For example, both `igraph` and `statnet` have a function called `degree` to calculate vertex degree centrality. When both `igraph` and `statnet` are loaded, you should use the `package::function()` notation to specify which package you want to take the function from (see Section 2.2).

```
# Load the statnet package "network" (we use suppressMessages() to avoid
# printing verbose loading messages) and intergraph.
library(network)
library(sna)
library(intergraph)

# Let's convert our ego-network to a statnet "network" object.
net.gr <- asNetwork(gr)

# Check out the object's class.
class(net.gr)
```

```
## [1] "network"
```

```
# Print the object for basic information. Note that vertex and edge attributes
# are correctly imported.
net.gr
```

```
##  Network attributes:
##   vertices = 45
##   directed = FALSE
##   hyper = FALSE
##   loops = FALSE
##   multiple = FALSE
##   bipartite = FALSE
##   total edges= 259
##     missing edges= 0
##     non-missing edges= 259
##
```

```
##  Vertex attribute names:
##     alter_num alter.age alter.age.cat alter.clo alter.fam alter.loan alter.nat alter
##
##  Edge attribute names:
##     weight
```

```
# Alternatively, we can create a network object from an adjacency matrix.

# Adjacency matrix for ego 20
head(adj.28)
```

```
##      2801 2802 2803 2804 2805 2806 2807 2808 2809 2810 2811 2812 2813 2814 2815
## 2801   NA    1    1    1    1    1    1    1    1    1    1    1    1    1    1
## 2802    1   NA    1    1    1    1    1    1    1    1    1    1    1    0    1
## 2803    1    1   NA    1    1    1    1    1    1    1    1    1    0    0    0
## 2804    1    1    1   NA    1    1    1    1    1    1    2    0    0    0    0
## 2805    1    1    1    1   NA    1    1    1    1    1    1    0    0    0    0
## 2806    1    1    1    1    1   NA    1    1    1    1    1    1    0    0    0
##      2816 2817 2818 2819 2820 2821 2822 2823 2824 2825 2826 2827 2828 2829 2830
## 2801    0    0    2    0    1    0    0    1    0    1    0    1    1    2    0
## 2802    0    0    0    0    0    0    0    1    0    0    0    0    0    0    0
## 2803    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0
## 2804    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0
## 2805    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0
## 2806    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0
##      2831 2832 2833 2834 2835 2836 2837 2838 2839 2840 2841 2842 2843 2844 2845
## 2801    1    0    0    0    0    0    0    0    0    1    1    0    0    0    0
## 2802    1    0    0    0    0    0    0    0    0    1    1    0    0    0    0
## 2803    0    0    0    0    0    0    0    0    0    1    1    0    0    0    0
## 2804    0    0    0    0    0    0    0    0    0    2    1    0    0    0    0
## 2805    0    0    0    0    0    0    0    0    0    1    1    0    0    0    0
## 2806    0    0    0    0    0    0    0    0    0    1    1    0    0    0    0
```

```
# Convert to network object.
net <- as.network(adj.28, matrix.type = "adjacency", directed=FALSE,
                  ignore.eval=FALSE, names.eval = "weight")

# NOTE:
# -- ignore.eval=FALSE imports adjacency cell values as edge attribute.
# -- names.eval = "weight" indicates that the resulting edge attribute should be
# called "weight".

# Print the result. Note that there is only "vertex.names" as vertex attribute.
net
```

```
##  Network attributes:
##   vertices = 45
```

```
##   directed = FALSE
##   hyper = FALSE
##   loops = FALSE
##   multiple = FALSE
##   bipartite = FALSE
##   total edges= 259
##     missing edges= 0
##     non-missing edges= 259
##
##  Vertex attribute names:
##     vertex.names
##
##  Edge attribute names:
##     weight
```

```
# The matrix column names are imported as vertex attribute "vertex name".
net %v% "vertex.names"
```

```
##  [1] "2801" "2802" "2803" "2804" "2805" "2806" "2807" "2808" "2809" "2810"
## [11] "2811" "2812" "2813" "2814" "2815" "2816" "2817" "2818" "2819" "2820"
## [21] "2821" "2822" "2823" "2824" "2825" "2826" "2827" "2828" "2829" "2830"
## [31] "2831" "2832" "2833" "2834" "2835" "2836" "2837" "2838" "2839" "2840"
## [41] "2841" "2842" "2843" "2844" "2845"
```

```
# We want to import more vertex attributes. Let's get alter attributes for ego 28.
vert.attr <- alter.attr.28

# We can now take columns from "vert.attr" and set them as attributes. However,
# we first need to make sure that nodes are in the same order in the attribute
# data frame (vert.attr) and in the network object (net).
identical(net %v% "vertex.names", as.character(vert.attr$alter_ID))
```

```
## [1] TRUE
```

```
# Let's import, for example, the sex alter attribute.
# There's no sex attribute right now in the "net" object.
net %v% "alter.sex"
```

```
##  [1] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
## [26] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
```

```
# Let's set it.
net %v% "alter.sex" <- vert.attr$alter.sex


# Basic statistics: Number of nodes, dyads, edges in the network.
network.size(net)
```

```
## [1] 45
```

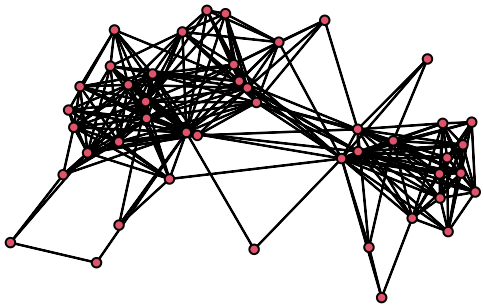
```
network.dyadcount(net)
```

```
## [1] 990
```

```
network.edgecount(net)
```

```
## [1] 259
```

```
# Simple plot
set.seed(2106)
plot(net)
```



```
# A network object can be indexed as an adjacency matrix.

# Adjacency matrix row for alter #3
net[3,]
```

```
##  [1] 1 1 0 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [39] 0 1 1 0 0 0 0
```

```
# ...for first 3 alters
net[1:3,]
```

```
##      2801 2802 2803 2804 2805 2806 2807 2808 2809 2810 2811 2812 2813 2814 2815
## 2801    0    1    1    1    1    1    1    1    1    1    1    1    1    1    1
## 2802    1    0    1    1    1    1    1    1    1    1    1    1    1    0    1
## 2803    1    1    0    1    1    1    1    1    1    1    1    1    1    0    0
##      2816 2817 2818 2819 2820 2821 2822 2823 2824 2825 2826 2827 2828 2829 2830
## 2801    0    0    1    0    1    0    0    1    0    1    0    1    1    1    0
## 2802    0    0    0    0    0    0    0    1    0    0    0    0    0    0    0
## 2803    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0
##      2831 2832 2833 2834 2835 2836 2837 2838 2839 2840 2841 2842 2843 2844 2845
## 2801    1    0    0    0    0    0    0    0    0    1    1    0    0    0    0
## 2802    1    0    0    0    0    0    0    0    0    1    1    0    0    0    0
## 2803    0    0    0    0    0    0    0    0    0    1    1    0    0    0    0
```

```
# Get a specific vertex attribute.
net %v% "alter.sex"
```

```
##  [1] "Female" "Male"   "Male"   "Male"   "Female" "Female" "Male"   "Female"
##  [9] "Female" "Male"   "Male"   "Male"   "Male"   "Male"   "Male"   "Male"
## [17] "Male"   "Male"   "Male"   "Male"   "Male"   "Male"   "Male"   "Male"
## [25] "Male"   "Male"   "Male"   "Female" "Male"   "Male"   "Male"   "Female"
## [33] "Male"   "Male"   "Male"   "Male"   "Female" "Male"   "Male"   "Male"
## [41] "Male"   "Male"   "Male"   "Male"   "Male"
```

```
# Get a specific edge attribute.
net %e% "weight"
```

```
##   [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 2 1 1 1 1 1 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [38] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [75] 1 1 1 1 1 2 1 1 1 1 1 1 1 1 1 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 1 2 1
## [112] 1 1 1 1 1 1 1 1 2 1 1 1 1 1 1 1 1 1 2 1 1 1 1 1 1 1 1 2 1 1 1 1 2 1 1 1
## [149] 1 1 1 1 1 1 1 2 2 1 1 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 1 1 1 1 1 1
## [186] 1 1 1 1 1 1 1 1 1 1 1 2 1 1 1 1 1 1 1 1 1 1 2 1 1 1 1 2 1 1 2 1 1 1 1 1 1
## [223] 1 1 1 1 1 1 1 1 2 1 1 1 1 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 1 1
```

```
# When both igraph and statnet are loaded, you should call their functions using
# the "::" notation.

# E.g. there is a "degree()" function both in igraph and statnet (sna package).
# Check out the help for degree: ?degree

# If we just call "degree()", R will take it from the most recently loaded
# package, in this case statnet (sna). This causes an error if degree() is
# applied to an igraph object.
degree(gr)
```

```
## Error in FUN(X[[i]], ...): as.edgelist.sna input must be an adjacency matrix/array, edgelist m
```

```
# Let's specify we want igraph's degree().
igraph::degree(gr)
```

```
## 2801 2802 2803 2804 2805 2806 2807 2808 2809 2810 2811 2812 2813 2814 2815 2816
##   24   17   13   12   12   13   13   12   10    9   11    8   18   14   16    9
## 2817 2818 2819 2820 2821 2822 2823 2824 2825 2826 2827 2828 2829 2830 2831 2832
##    9   10    2    2    9   18   27   13   10    8    4    3    3    3   15   16
## 2833 2834 2835 2836 2837 2838 2839 2840 2841 2842 2843 2844 2845
##   16   12   21    3   10   14   14   16   13    7   12    5   12
```

```
# Before proceeding, let's un-load the sna package to avoid function name
# overlaps with igraph.
detach("package:sna", unload=TRUE)
```

8.5 Illustrative example: personal networks of Sri Lankan immigrants in Italy

In this section, we'll apply some of the tools presented earlier to a specific case study about the personal networks of Sri Lankan immigrants in Italy. We will demonstrate the use of R in four major tasks of personal network analysis:

1. **Visualization**. We'll write an R function that plots a personal network with specific graphical parameters. We'll then run this function on many personal networks and we'll export the output to an external pdf file.
2. **Compositional analysis**. We'll calculate multiple measures on network composition for all our personal networks at once.
3. **Structural analysis**. We'll write an R function that calculates multiple structural measures on a personal network and we'll run it on all the personal networks in our data at once.
4. **Association with ego-level variables**. We'll merge the results of personal network compositional and structural analysis with non-network ego-level data, and we'll analyze the association between ego-level attributes and personal network characteristics.

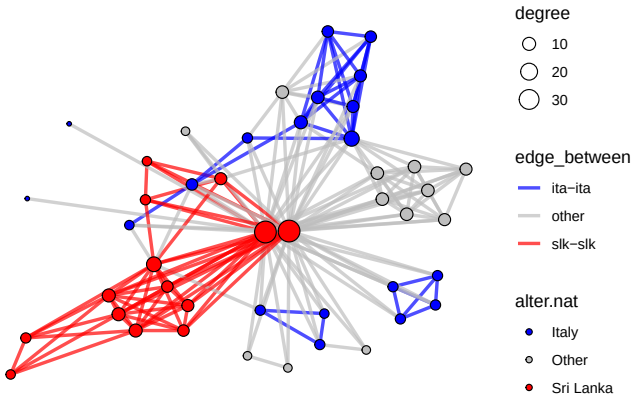
```
# Load package
library(igraph)
library(ggraph)

# Load the data
load("./Data/data.rda")

# Visualization
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

# We want to produce the same kind of plot for all our personal networks, with
# the following characteristics:
# -- Vertices sized by degree centrality.
# -- Sri Lankan alters in red, Italian alters in blue, Other alters in
# grey.
# -- Edges among Sri Lankans in light red, edges among Italians in light blue,
# all other edges in grey.

# Let's first try this on a single ego-network.

# Get an ego-network's igaph object.
gr <- gr.list[["45"]]

# Vertex sequence of Sri Lankans.
slk <- V(gr)[alter.nat=="Sri Lanka"]
```

```
# Vertex sequence of Italians.
ita <- V(gr)[alter.nat=="Italy"]
# Vertex sequence of Other.
oth <- V(gr)[alter.nat=="Other"]

# Edge attribute indicating which group the edge is between.
# Residual value.
E(gr)$edge_between <- "other"
# Overwrite value for edges among Sri Lankans.
E(gr)[slk %--% slk]$edge_between <- "slk-slk"
# Overwrite value for edges among Italians.
E(gr)[ita %--% ita]$edge_between <- "ita-ita"

# Calculate degree centrality and set it as vertex attribute in gr.
igraph::degree(gr)
```

```
## 4501 4502 4503 4504 4505 4506 4507 4508 4509 4510 4511 4512 4513 4514 4515 4516
##   38   14   38    8    7    7   10    9    9    9    9    8    9    8    5    4
## 4517 4518 4519 4520 4521 4522 4523 4524 4525 4526 4527 4528 4529 4530 4531 4532
##   10   10    7   15    5    5    3    4    1    5    5    4    8    3    3    3
## 4533 4534 4535 4536 4537 4538 4539 4540 4541 4542 4543 4544 4545
##    5    5    5    4    1    5    8   10    9    7    7    8    9
```

```
V(gr)$degree <- igraph::degree(gr)

# Plot
set.seed(613)
ggraph(gr, layout= "fr") +
  geom_edge_link(aes(color=edge_between), width=1, alpha= 0.7) + # Edges
  geom_node_point(aes(fill=alter.nat, size= degree), shape=21) + # Nodes
  scale_fill_manual(values=c("Italy"="blue", "Sri Lanka"="red", "Other"="grey")) + # Colors for n
  scale_edge_color_manual(values=c("ita-ita"="blue", "slk-slk"="red", "other"="grey")) +
  # Colors for edges
  ggtitle(gr$ego_ID) + # Ego ID as plot title
  theme_graph(base_family = 'Helvetica') # Empty plot theme for network visualizations
```

45




```
# Let's now plot the first 10 personal networks. We won't print the plots in the
# GUI, but we'll export them to an external pdf file.

# Put all the code above in a single function
my.plot <- function(gr) {

  # Vertex sequence of Sri Lankans.
  slk <- V(gr)[alter.nat=="Sri Lanka"]
  # Vertex sequence of Italians.
  ita <- V(gr)[alter.nat=="Italy"]
  # Vertex sequence of Other.
  oth <- V(gr)[alter.nat=="Other"]

  # Edge attribute indicating which group the edge is between.
  # Residual value.
  E(gr)$edge_between <- "other"
  # Overwrite value for edges among Sri Lankans.
  E(gr)[slk %--% slk]$edge_between <- "slk-slk"
  # Overwrite value for edges among Italians.
  E(gr)[ita %--% ita]$edge_between <- "ita-ita"

  # Calculate degree centrality and set it as vertex attribute in gr
  degree(gr)
  V(gr)$degree <- degree(gr)

  # Plot
  set.seed(613)
  ggraph(gr, layout= "fr") +
    geom_edge_link(aes(color=edge_between), width=1, alpha= 0.7) +
    geom_node_point(aes(fill=alter.nat, size= degree), shape=21) +
    scale_fill_manual(values=c("Italy"="blue", "Sri Lanka"="red", "Other"="grey")) +
```

```
    scale_edge_color_manual(values=c("ita-ita"="blue", "slk-slk"="red", "other"="grey")) +
    ggtitle(gr$ego_ID) +
    theme_graph(base_family = 'Helvetica')
}

# Run the function we just created on each of the first 10 personal networks
# with a for loop.

# Open pdf device
pdf(file = "./Figures/ego_nets_graphs.pdf")

# For loop to make plot for each i in first 10 ego-networks.
for (i in 1:10) {

  # Get the graph from list
  gr <- gr.list[[i]]

  # Run the plot function and print (needed for ggplot within a for loop).
  my.plot(gr) |> print()
}

# Close pdf device
dev.off()
```

```
## pdf
##   2
```

```
# Same result, but with tidyverse code:

# Open pdf device
pdf(file = "./Figures/ego_nets_graphs.pdf")

# Get list of ego-networks (first 10 elements)
gr.list[1:10] |>
  # walk: similar to map(), but calls a function for its side effects (here,
  # producing a plot)
  walk(~ my.plot(.x) |> print())

# Close pdf device
dev.off()
```

```
## pdf
##   2
```

```
# Compositional analysis
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

```
# Frequency of different nationalities in the personal networks.

# Group the pooled alter attribute data frame by ego_ID.
alter.attr.all <- alter.attr.all |>
  group_by(ego_ID)

# Run the compositional measures.
comp.measures <- dplyr::summarise(alter.attr.all,
                                  # N Italians
                                  N.ita= sum(alter.nat=="Italy"),
                                  # N Sri Lankans
                                  N.slk= sum(alter.nat=="Sri Lanka"),
                                  # N Others
                                  N.oth= sum(alter.nat=="Other"))

# Add the relative frequency of Italians.
comp.measures <- comp.measures |>
  mutate(prop.ita = N.ita/45)

# Show result.
comp.measures
```

```
## # A tibble: 102 x 5
##    ego_ID N.ita N.slk N.oth prop.ita
##     <dbl> <int> <int> <int>    <dbl>
##  1     28     0    43     2   0
##  2     29     1    44     0   0.0222
##  3     33     2    32    11   0.0444
##  4     35     4    33     8   0.0889
##  5     39     5    39     1   0.111
##  6     40     1    34    10   0.0222
##  7     45    19    14    12   0.422
##  8     46     7    33     5   0.156
##  9     47     0    45     0   0
## 10     48     4    39     2   0.0889
## # i 92 more rows
```

```
# Structural analysis
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

# We are going to write a function that takes an igraph object as argument and
# returns 3 values:
# -- The density of ties among Italian alters
# -- The average degree of Italian alters.
# -- The average degree of Italian alters, as a proportion of the overall
# average degree.
```

```
# Write the function.
str.ita <- function(gr) {

  # Vertex sequence of Italians in the graph.
  ita <- V(gr)[alter.nat=="Italy"]

  # Number of Italians.
  n <- length(ita)

  # Number of all possible edges among Italians.
  possible <- n*(n-1)/2

  # Actual number of edges among Italians.
  actual <- E(gr)[ita %--% ita] |> length()
  # Notice that this is 0 if there are no Italian contacts, i.e. length(ita)==0.

  # Density of connections among Italians.
  dens.ita <- actual/possible
  # Notice that this is NaN if there are no Italian contacts.

  # Average degree of Italian contacts.
  avg.deg.ita <- igraph::degree(gr, V(gr)[alter.nat=="Italy"]) |> mean()
  # This is also NaN if there are no Italians.

  # Average degree of Italians, as a proportion of overall average degree.
  avg.deg.ita.prop <- avg.deg.ita/mean(igraph::degree(gr))
  # This is also NaN if there are no Italians.

  # Return the 3 results as a tibble
  tibble(dens.ita = dens.ita,
         avg.deg.ita = avg.deg.ita,
         avg.deg.ita.prop = avg.deg.ita.prop)
}

# Run the function on all personal networks.
(str.ita.df <- map_dfr(gr.list, str.ita, .id= "ego_ID"))
```

```
## # A tibble: 102 x 4
##    ego_ID dens.ita avg.deg.ita avg.deg.ita.prop
##    <chr>     <dbl>       <dbl>            <dbl>
##  1 28      NaN          NaN             NaN
##  2 29      NaN            0               0
##  3 33        1            1               0.109
##  4 35        0            3.25            0.331
##  5 39        0            0               0
```

```
##  6 40      NaN            3               0.265
##  7 45        0.193        6.11            0.751
##  8 46        0.571       10.7             0.903
##  9 47      NaN          NaN             NaN
## 10 48        1            5               0.560
## # i 92 more rows
```

```
# Convert ego_ID from character to numeric for consistency with ego attribute
# data.
str.ita.df <- str.ita.df |>
  mutate(ego_ID = as.numeric(ego_ID))

# Association with ego-level variables
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

# Ego attribute data
ego.df
```

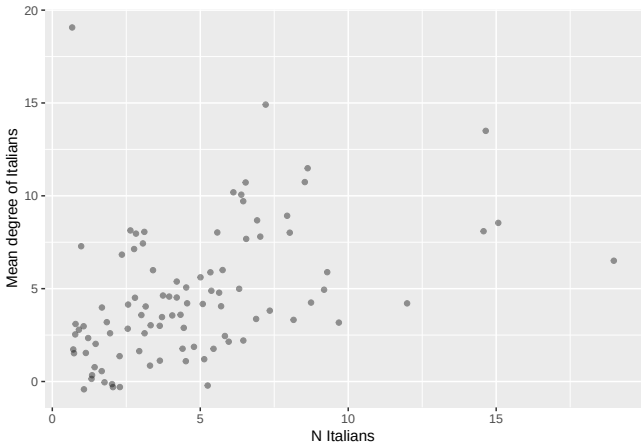
```
## # A tibble: 102 x 9
##    ego_ID ego.sex ego.age ego.arr ego.edu ego.inc  empl ego.empl.bin ego.age.cat
##     <dbl> <fct>     <dbl>   <dbl> <fct>     <dbl> <dbl> <fct>        <fct>
##  1     28 Male         61    2008 Second~     350     3 Yes          60+
##  2     29 Male         38    2000 Primary     900     4 Yes          36-40
##  3     33 Male         30    2010 Primary     200     3 Yes          26-30
##  4     35 Male         25    2009 Second~    1000     3 Yes          18-25
##  5     39 Male         29    2007 Primary       0     1 No           26-30
##  6     40 Male         56    2008 Second~     950     4 Yes          51-60
##  7     45 Male         52    1975 Primary    1600     3 Yes          51-60
##  8     46 Male         35    2002 Second~    1200     4 Yes          31-35
##  9     47 Male         22    2010 Second~     700     4 Yes          18-25
## 10     48 Male         51    2007 Primary     950     4 Yes          51-60
## # i 92 more rows
```

```
# Join ego-level data with compositional variables obtained above.
ego.df <- ego.df |>
  left_join(comp.measures, by= "ego_ID")

# Merge ego-level data with structural variables obtained above.
ego.df <- ego.df |>
  left_join(str.ita.df, by= "ego_ID")

# See the result (just a subset of the variables)
ego.df |>
  dplyr::select(ego_ID, ego.sex, ego.age, N.slk, prop.ita, dens.ita)
```

```
## # A tibble: 102 x 6
##    ego_ID ego.sex ego.age N.slk prop.ita dens.ita
```

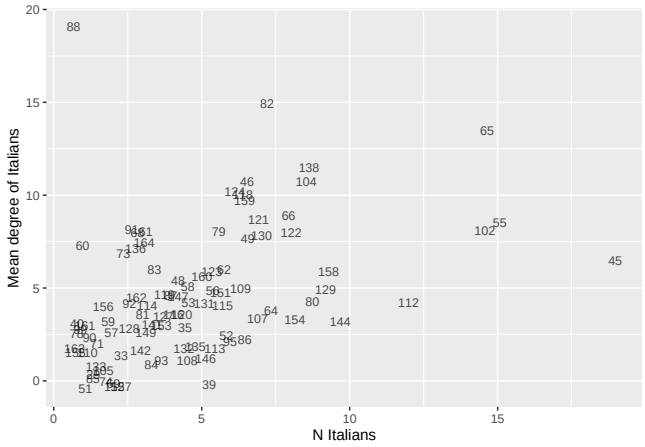
```
##      <dbl> <fct>     <dbl> <int>    <dbl>    <dbl>
##  1     28 Male         61    43   0       NaN
##  2     29 Male         38    44   0.0222  NaN
##  3     33 Male         30    32   0.0444    1
##  4     35 Male         25    33   0.0889    0
##  5     39 Male         29    39   0.111     0
##  6     40 Male         56    34   0.0222  NaN
##  7     45 Male         52    14   0.422     0.193
##  8     46 Male         35    33   0.156     0.571
##  9     47 Male         22    45   0       NaN
## 10     48 Male         51    39   0.0889    1
## # i 92 more rows
```

```
# Plot number by mean degree of Italians.
# To avoid warnings from ggplot(), let's remove cases with NA values on the
# relevant variables.
data <- ego.df |>
  filter(complete.cases(N.ita, avg.deg.ita))

# Set seed for reproducibility (jittering is random).
set.seed(613)
# Get and save plot
p <- ggplot(data= data, aes(x= N.ita, y= avg.deg.ita)) +
  geom_point(size=1.5, alpha=0.4, position=position_jitter(height=0.5, width=0.5)) +
  labs(x= "N Italians", y= "Mean degree of Italians")
# Note that we slightly jitter the points to avoid overplotting.
# Print plot.
print(p)
```

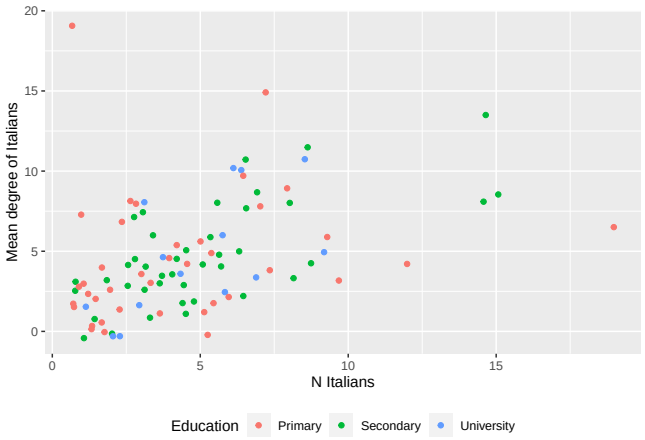



```
# Same as above, but with ego_IDs as labels to identify the egos.
# Set seed for reproducibility (jittering is random).
set.seed(613)
```

```
# Get and save plot
p <- ggplot(data= data, aes(x= N.ita, y= avg.deg.ita)) +
  geom_text(aes(label= ego_ID), size=3, alpha=0.7,
            position=position_jitter(height=0.5, width=0.5)) +
  labs(x= "N Italians", y= "Mean degree of Italians")
# Note that we slightly jitter the labels with position_jitter() so that egoIDs
# don't overlap and are more readable.
# Print plot.
print(p)
```

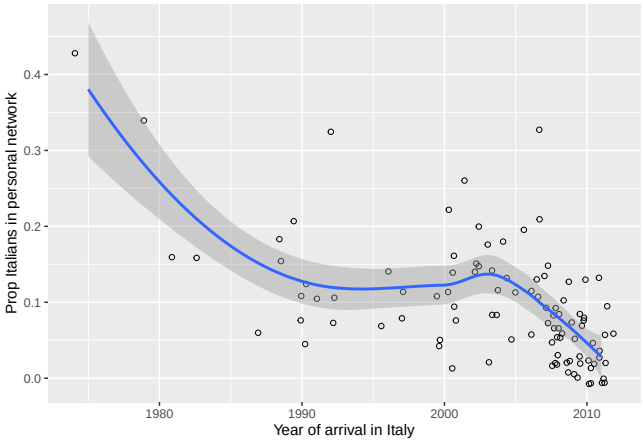



```
# Same as above, with color representing ego's educational level.
# Set seed for reproducibility (jittering is random).
set.seed(613)
# Get and save plot
p <- ggplot(data= data, aes(x= N.ita, y= avg.deg.ita)) +
  geom_point(size=1.5, aes(color= as.factor(ego.edu)),
             position=position_jitter(height=0.5, width=0.5)) +
  theme(legend.position="bottom") +
  labs(x= "N Italians", y= "Mean degree of Italians", color= "Education")
# Note that we slightly jitter the points to avoid overplotting.
# Print plot.
print(p)
```



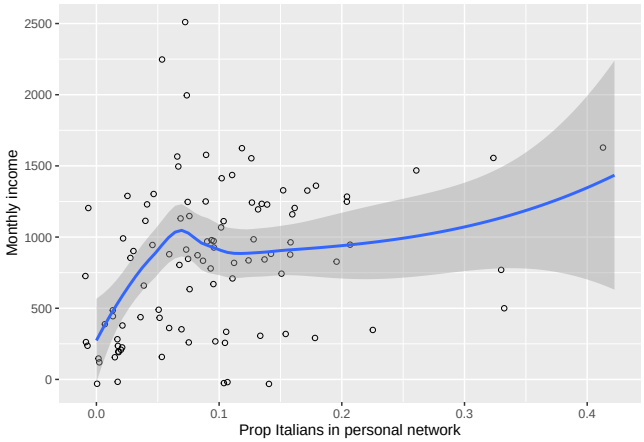

```
# Do Sri Lankans who arrived to Italy more recently have fewer Italian contacts
# in their personal network?

# To avoid warnings from ggplot(), let's remove cases with NA values on the
# relevant variables.
data <- ego.df |>
  filter(complete.cases(ego.arr, prop.ita))

# Set seed for reproducibility (jittering is random).
set.seed(613)
# Get and save plot
p <- ggplot(data= data, aes(x= ego.arr, y= prop.ita)) +
  geom_point(shape= 1, size= 1.5, position= position_jitter(width= 1, height= 0.01)) +
  geom_smooth(method="loess") +
  labs(x= "Year of arrival in Italy", y= "Prop Italians in personal network")
# Print plot.
print(p)
```

```
## `geom_smooth()` using formula = 'y ~ x'
```




```
# Do Sri Lankans with more Italian contacts in their personal network have higher
# income?

# To avoid warnings from ggplot(), let's remove cases with NA values on the
# relevant variables.
data <- ego.df |>
  filter(complete.cases(prop.ita, ego.inc))

# Set seed for reproducibility (jittering is random).
set.seed(613)
# Get and save plot
p <- ggplot(data= data, aes(x= prop.ita, y= ego.inc)) +
  geom_jitter(width= 0.01, height= 50, shape= 1, size= 1.5) +
  geom_smooth(method="loess") +
  labs(x= "Prop Italians in personal network", y= "Monthly income")
## Print plot.
print(p)
```

```
## `geom_smooth()` using formula = 'y ~ x'
```


2500
2000
1500
1000
500
0
Monthly income
0.0
0.1
0.2
0.3
0.4
Prop Italians in personal network

Bibliography

Nick Crossley, Elisa Bellotti, Gemma Edwards, Martin G. Everett, Johan Koskinen, and Mark Tranmer. *Social network analysis for ego-nets*. SAGE Publications, London, August 2015. ISBN 978-1-4462-6776-9.

John Fox and Sanford Weisberg. *An R Companion to Applied Regression*. SAGE Publications, Inc, Los Angeles, 3rd edition edition, October 2018. ISBN 978-1-5443-3647-3.

Harvey Goldstein. *Multilevel Statistical Models*. Wiley, Chichester, West Sussex, 4 edition edition, November 2010. ISBN 978-0-470-74865-7.

Christopher McCarty, Miranda J. Lubbers, Raffaele Vacca, and José Luis Molina. *Conducting personal network research: a practical guide*. Methodology in the social sciences. The Guilford Press, New York City, 2019. ISBN 978-1-4625-3838-6.

Brea L. Perry, Steve Borgatti, and Bernice A. Pescosolido. *Egocentric network analysis: Foundation, methods, and models*. Cambridge University Press, February 2018. ISBN 978-1-107-57931-6.

J. Rasbash, F. Steele, and George Reckie. *LEMMA: Learning environment for multilevel methodology and applications*. University of Bristol: Centre for Multilevel Modelling, 2008. URL https://www.cmm.bris.ac.uk/lemma.

Tom Snijders and R. J. Bosker. *Multilevel analysis: An introduction to basic and advanced multilevel modeling*. SAGE Publications, London; Thousand Oaks, Calif, 2nd edition edition, 2012. ISBN 0-7619-5889-4.

Raffaele Vacca. Multilevel models for personal networks: methods and applications. *Statistica Applicata - Italian Journal of Applied Statistics*, 30 (1):59–97, 2018. ISSN 2038-5587. doi: 10.26398/IJAS.0030-003. URL https://www.sa-ijas.org/ojs/index.php/sa-ijas/article/view/30-3.

Raffaele Vacca, Jeanne-Marie R. Stacciarini, and Mark Tranmer. Cross-classified multilevel models for personal networks: Detecting and accounting for overlapping actors. *Sociological Methods & Research*, page 004912411988245, November 2019. ISSN 0049-1241, 1552-8294. doi: 10.1

177/0049124119882450. URL http://journals.sagepub.com/doi/10.1177/0049124119882450.

Marijtje A. J van Duijn. Multilevel modeling of social network and relational data. In Jeffrey S. Simonoff, Marc A. Scott, and Brian D. Marx, editors, *The SAGE handbook of multilevel modeling*, pages 599–618. SAGE, Los Angeles, 2013. ISBN 978-0-85702-564-7.

Hadley Wickham. The Split-Apply-Combine Strategy for Data Analysis. *Journal of Statistical Software*, 40(1):1–29, 2011. ISSN 1548-7660. URL http://www.jstatsoft.org/v40/i01.