REDES Revista Hispana para el Análisis de Redes Sociales

Vol.32,#1, (2021), 82-95

http://revistes.uab.es/redes https://doi.org/10.5565/rev/redes.910

Análisis de redes egocéntricas con R (III). Egoredes múltiples

Raffaele Vacca[1]

University of Florida

RESUMEN

Este texto es el tercero de una serie de cuatro que conjuntamente constituyen un taller sobre análisis de ego-redes (y/o redes personales) con R. El texto está acompañado por ficheros de datos y los scripts de lenguaje R necesarios para realizar las actividades propuestas.

Palabras clave: *Ego-redes - Redes personales – R.*

ABSTRACT

This text is the third of a series of four documents that together constitute a workshop on analysis of ego-networks (and / or personal networks) using R. The text is accompanied by data files and the R scripts necessary to carry out the suggested activities.

Key words: *Ego-networks – Personal networks – R.*

[1] *Contacto el autor: Raffaele Vacca (r.vacca@ufl.edu). Traducción de José Luis Molina*

INTRODUCCIÓN

Ahora que hemos aprendido cómo almacenar y analizar datos de una red egocéntrica, estamos en condiciones de aplicar estas operaciones a una *colección* de estas redes.

Este documento presenta los siguientes temas:

• Listas de R y cómo se pueden utilizar para almacenar múltiples egoredes.

• Almacenamiento de múltiples egoredes como dataframes o listas.

• Dividir-aplicar-combinar: Ejecutar la misma operación en muchas egoredes y volver a combinar los resultados.

• Análisis de la composición de la red de ego: dividir-aplicar-combinar en un dataframe los atributos de los alteri (paquete `dplyr´).

• Análisis de la estructura de la red de ego: dividir-aplicar-combinar utilizando una lista de objetos `igraph´ (paquete `purrr´).

Listas

• Las listas son simples colecciones de objetos.

• Una lista puede contener cualquier clase de objeto, sin restricciones. Una lista puede contener otras listas.

• Las listas se identifican con la etiqueta `list´ en el argumento `type´y `class´.

• Los dataframes no son un tipo de lista. Otros objetos complejos en R se pueden almacenar como listas (usando la etiqueta `list´ en `type`), a pesar de que su `class´ no sea una `list´: e.g. resultados de estimaciones estadísticas o procedimentos de redes para la detección de comunidades.

• Usar `str(list)´para mostrar los tipos y longitudes de los elementos presentes en una lista.

• Tres notaciones diferentes para indexar listas:

1. Notación `[ ]´: e.g.`my.list[3]´.

2. Notación `[[ ]]´: e.g. `my.list[[3]]´ o `my.list[["element.name"]]´.
3. Notación`$`. Esta notación solamente funciona para las listas nombradas con un literal. E.g., `list$element.name´. El mismo resultado se obtiene con la notación`[[ ]]´: `list$element.name` equivale a `list[["element.name"]]´ o a `list[[i]]´ (donde `i´ se refiere a la posición del elemento llamado *element.name* en la lista).

• Estos tres métodos de indexación funcionan exactamente de la misma manera que para los dataframes (consultar el apartado "Indexación y segmentación de data frames" en la Parte I).

• Hay que tener en cuenta que la indexación también se puede utilizar para reordenar los elementos de una lista.

- De hecho, la indexación se puede utilizar para reordenar elementos en cualquier objeto, incluidas filas o columnas en dataframes, elementos vectoriales, filas o columnas de matrices, etc.

¿Qué hace el siguiente código?

- Crea, muestra e indexa una lista.

```
# 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" ...
```

DIVIDIR, APLICAR, COMBINAR EGOREDES

• La estrategia denominada "Dividir-Aplicar-Combinar" (Wickham, 2011) es común en muchos tipos de análisis de datos en las ciencias sociales. Dividir-aplicar-combinar es lo que hacemos siempre que nuestros datos están en un solo archivo u objeto, y:

1. Dividimos el objeto en partes según una o varias (combinaciones de) variables categóricas o factores (dividir).
2. Aplicamos exactamente el mismo tipo de cálculo en cada segmento, de forma idéntica e independiente (aplicar).
3. Seleccionamos los resultados y los volvemos a combinar, e.g. en un nuevo conjunto de datos (combinar).

• La estrategia dividir-aplicar-combinar es esencial en el análisis de egoredes. Con estas redes estamos de forma reiterada (1) dividiendo los datos en partes, cada parte correspondiendo normalmente a un ego; (2) realizando análisis idénticos e independientes de cada segmento (cada egored); (3) combinando los resultados, generalmente en un solo conjunto de datos, para luego asociarlos con otras variables a nivel de ego.

• En el R básico y tradicional, las herramientas comunes para realizar operaciones para dividir-aplicar-combinar incluyen bucles de la colección de funciones de aplicación y agregación.

• Los paquetes *tidyverse* proporcionan nuevas formas de realizar operaciones dividir-aplicar-combinar con un código más eficiente y legible:

- Agrupar y resumir mediante el paquete *dplyr*.
- Mapear mediante la colección de funciones disponibles en el paquete *purrr*.

Composición: agrupación y resumen con *dplyr*

• Siempre que tengamos un conjunto de datos en el que las filas (nivel 1) estén agrupadas o agrupadas por valores de un factor dado (nivel 2), el paquete *dplyr* hace que los resúmenes de nivel 2 sean extremadamente fáciles.

- En el análisis egocéntrico, normalmente tenemos un dataframe de atributos alter cuyas filas (alteri, nivel 1) están agrupados por egos (nivel 2).

• *dplyr* presenta nuevas funciones para calcular estadísticas de resumen sobre múltiples variables en una línea de código: resume () y resume_all (). Un marco de datos se puede agrupar (group_by ()) por una variable, para luego calcular resúmenes estadísticos para cada valor único de esa variable. Si la variable de agrupación es ego_ID, podemos obtener inmediatamente resúmenes estadísticos para cada uno de los cientos o miles de egos con una sola línea de código.

¿Qué hace el siguiente código?

- Utiliza resume() y resume_all() para calcular las variables de resumen sobre la composición de la red para el conjunto de los 102 egos a la vez.
- Fusiona los resultados con otros datos a nivel del ego (combinación de nivel 2).

```
# Load packages.
library(tidyverse)
## Registered S3 methods overwritten by 'ggplot2':
## method           from
## [.quosures       rlang
## c.quosures       rlang
## print.quosures   rlang

## -- Attaching packages --- tidyverse 1.2.1 --

## v ggplot2 3.1.1     v purrr 0.3.2
## v tibble 2.1.3      v dplyr 0.8.1
## v tidyr 0.8.3       v stringr 1.4.0
## v readr 1.3.1       v forcats 0.4.0
```

```
## -- Conflicts ------ tidyverse_conflicts() --

## x dplyr::filter()      masks stats::filter()
## x dplyr::lag()         masks stats::lag()
library(igraph)
##
## Attaching package: 'igraph'
## The following objects are masked from 'package:dplyr':
##
##      as_data_frame, groups, union
## The following objects are masked from 'package:purrr':
##
##      compose, simplify
## The following object is masked from 'package:tidyr':
##
##      crossing
## The following object is masked from 'package:tibble':
##
##      as_data_frame
## The following objects are masked from 'package:stats':
##
##      decompose, spectrum
## The following object is masked from 'package:base':
##
##      union
# Load the data
load("./Data/data.rda")

# The count() function counts the number of rows for each unique value of a
# variable. For example, let's count the number of alters for each ego: this
# is simply the number of rows for each ego_ID in alter.attr.all
(N.alters <- dplyr::count(alter.attr.all, 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
## # ... with 92 more rows
```

```
# The summarise() and summarise_all() functions offer a concise syntax to
# calculate summary statistics on a data frame's variables.

# Let's see what happens if we apply these functions to the alter attribute data
# frame without grouping it by ego ID.

# * Mean alter closeness:
summarise(alter.attr.all, 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:
summarise(alter.attr.all, N.nat = n_distinct(alter.nat))
## # A tibble: 1 x 1
## N.nat
## <int>
## 1 3
# * N of distinct values in *each* of the alter variables:
summarise_all(alter.attr.all, list(~ n_distinct(.)))

## # A tibble: 1 x 11
##  alter_ID   ego_ID     alter_num      alter.sex      alter.age.cat      alter.rel      alter.nat
##  <int>        <int>        <int>         <int>          <int>          <int>          <int>
## 1 4590         102           45             2              8              4              3
## # ... with 4 more variables: alter.res <int>, alter.clo <int>,
## # alter.loan <int>, alter.fam <int>
# What happens is that the function is calculated on all alters from all egos
# (all rows of the alter attribute data frame), not on the set of alters of each
# ego.

# But if we "group" a data frame by a given variable, these summary statistics
# are calculated for each unique value of the grouping variable. So if we group
# alter.attr.all by ego_ID, in one line of code we can calculate a host of
# summary statistics for each ego.
alter.attr.all <- group_by(alter.attr.all, ego_ID)

# Let's re-run the same code as above:
# * Mean alter closeness:
summarise(alter.attr.all, 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
## # ... with 92 more rows
# * N of distinct values in the alter nationality variable:
summarise(alter.attr.all, 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
## # ... with 92 more rows
# * N of distinct values in *each* of the alter variables:
summarise_all(alter.attr.all, list(~ n_distinct(.)))

## # A tibble: 102 x 11
## ego_ID    alter_ID   alter_num   alter.sex   alter.age.cat   alter.rel   alter.nat
##      <dbl>    <int>      <int>       <int>         <int>          <int>       <int>
## 1     28       45       45          2             7              4           2
## 2     29       45       45          2             7              4           2
## 3     33       45       45          2             6              4           3
## 4     35       45       45          2             7              4           3
## 5     39       45       45          2             7              4           3
## 6     40       45       45          2             6              4           3
## 7     45       45       45          2             8              4           3
## 8     46       45       45          2             7              4           3
## 9     47       45       45          2             6              4           1
## 10    48       45       45          2             7              4           3
## # ... with 92 more rows, and 4 more variables: alter.res <int>,
## # alter.clo <int>, alter.loan <int>, alter.fam <int>
# Now each summary statistic is calculated for each ego.

# 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 33 as an example.
alter.attr.28
## # A tibble: 45 x 11
##      alter_ID. ego_ID alter_num alter.sex alter.age.cat alter.rel    alter.nat
##      <dbl>     <dbl>     <dbl>      <fct>   <dbl>  <fct>   <fct>
## 1    2801       28          1        Female    6     Close fa~       Sri Lanka
## 2    2802       28          2        Male      6     Other fa~       Sri Lanka
## 3    2803       28          3        Male      6     Close fa~       Sri Lanka
## 4    2804       28          4        Male      7     Close fa~       Sri Lanka
## 5    2805      28           5        Female    5     Close fa~       Sri Lanka
## 6    2806      28           6        Female    7     Close fa~       Sri Lanka
## 7    2807      28           7        Male      5     Other fa~       Sri Lanka
## 8    2808      28           8        Female    4     Other fa~       Sri Lanka
## 9    2809      28           9        Female    6     Other fa~       Sri Lanka
## 10  2810      28          10        Male      7     Other fa~       Sri Lanka
## # ... with 35 more rows, and 4 more variables: alter.res <fct>,
## # alter.clo <dbl>, alter.loan <fct>, alter.fam <fct>
# 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
## [12] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [23] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [34] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [45] 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 <- dplyr::summarise(alter.attr.all,
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
## # ... with 92 more rows
# After getting compositional summary variables for each ego, we might want to
# merge them with other ego-level data (level-2 join).

# Merge with summary variables.
(ego.df <- left_join(ego.df, 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
##      <dbl>    <fct>        <dbl>       <dbl>       <fct>       <dbl>    <dbl>       <fct>
## 1    28       Male           61        2008          Second~    350        3        Yes
## 2    29       Male           38        2000          Primary    900        4        Yes
## 3    33       Male           30        2010          Primary    200        3        Yes
## 4    35       Male           25        2009          Second~   1000        3        Yes
## 5    39       Male           29        2007          Primary      0         1        No
## 6    40       Male           56        2008          Second~    950        4        Yes
## 7    45       Male           52        1975          Primary   1600        3        Yes
## 8    46       Male           35        2002          Second~   1200        4        Yes
## 9    47       Male           22        2010          Second~    700        4        Yes
## 10   48       Male           51        2007          Primary    950        4        Yes
## # ... with 92 more rows, and 4 more variables: ego.age.cat <dbl>,
## # N.clo.fam <int>, N.oth.fam <int>, N.fri <int>
# We can then ungroup alter.attr.all by ego ID.
alter.attr.all <- ungroup(alter.attr.all)
```

```
# An example of pipes with dplyr.
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
## # ... with 92 more rows
# ***** EXERCISE (1):
# Extract the values of alter.attr.all$alter.sex corresponding to ego_ID 28.
# 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").
# *****

# ***** EXERCISE (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.
# *****)
```

Estructura: aplicar una función a cada elemento de una lista con *purrr*

• En base R, las funciones de la familia *Apply*, como *lapply* y *sapply*, son la forma habitual de seleccionar una función y aplicarla a cada elemento de una lista.

• El paquete *purrr* hace lo mismo, pero de una manera más eficiente y con un código más legible.

• *purrr* proporciona funciones de tipo estable, es decir, que siempre devuelven el mismo tipo de resultado:

- *map* siempre devuelve una lista (esto es el equivalente de *lapply* en base R).
- *map_dbl* devuelve un vector de números de doble precisión. *map_int* devuelve un vector de números enteros.
- *map_chr* devuelve un vector de caracteres. *map_lgl* devuelve un vector lógico.
- *map_dfr* devuelve un dataframe. Asume que se está aplicando a cada egored una operación que devuelve una fila de dataframe para esa red. Luego enlaza las filas obtenidas del dataframe para cada egored en un solo dataframe a nivel de ego.

• Además de resultados estables, las funciones *purrr* también ofrecen una sintaxis muy cómoda: ~f (.x), donde:

- .x representa cada elemento de la lista de entrada.
- f es la función que se ejecutará en ese elemento.

• Por ejemplo, *map(L, ~ .x * 2*) toma cada elemento de la lista L (representado por .x) y lo multiplica dos veces.

• Las funciones *map()* conservan los nombres

de las listas en su resultado. En nuestro caso, esto significa que los IDs de los egos estarán disponibles en la lista de salida, así como los vectores (asumiendo que los IDs de los ego se almacenan como nombres en la lista de entrada).

• map() es útil cuando desea ejecutar funciones que toman un elemento de lista como argumento, e.g. una egored *igraph* y devolver otro elemento de la lista como salida, e.g. otro objeto igraph. Éste es el caso siempre que se quiera manipular las egoredes (por ejemplo, mantener solo un cierto tipo de vínculos o vértices), y almacenar los resultados en una nueva lista.

• *map_dbl()* es útil cuando desea ejecutar funciones que toman un elemento de lista como argumento, e.g. una egored *igraph* y devuelve un número como resultado. Éste es el caso siempre que desee ejecutar una medida estructural como la densidad de enlaces o la centralización en cada red.

¿Qué hace el siguiente código?

- Usa otras funciones de mapeo de *purrr* para calcular una medida estructural en cada egored.

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

```
##         28         29         33         35         39         40
## 0.26161616 0.20404040 0.20909091 0.22323232 0.09292929 0.25757576
##         45         46         47         48         49         51
## 0.18484848 0.26969697 0.53737374 0.20303030 0.42828283 0.16969697
##         52         53         55         56         57         58
## 0.12929293 0.16565657 0.23838384 0.22828283 0.60000000 0.20505051
##         59         60         61         62         64         65
## 0.50505051 0.28888889 0.37676768 0.30202020 0.28282828 0.36060606
##         66         68         69         71         73         74
## 0.30909091 0.23333333 0.23636364 0.47777778 0.40707071 0.42525253
##         78         79         80         81         82         83
## 0.17575758 0.29292929 0.33434343 0.29797980 0.35353535 0.38888889
##         84         85         86         87         88         90
## 0.27575758 0.12525253 0.18989899 0.24444444 0.40101010 0.45353535
##         91         92         93         94         95         97
## 0.47171717 0.43535354 0.14141414 0.26767677 0.36767677 0.35050505
##         99        102        104        105        107        108
## 0.24040404 0.27777778 0.22929293 0.26666667 0.16262626 0.10606061
##        109        110        112        113        114        115
## 0.33535354 0.48686869 0.13939394 0.23636364 0.28787879 0.33232323
##        116        118        119        120        121        122
## 0.22929293 0.37676768 0.33131313 0.20101010 0.23838384 0.35959596
##        123        124        125        126        127        128
## 0.39090909 0.61111111 0.26464646 0.72828283 0.28080808 0.26262626
##        129        130        131        132        133        135
```

```
## 0.22727273 0.22020202 0.27979798 0.29090909 0.27575758 0.22525253
##        136        138        139        140        141        142
## 0.28686869 0.19797980 0.31515152 0.29292929 0.26868687 0.21010101
##        144        146        147        149        151        152
## 0.24949495 0.14747475 0.23030303 0.20707071 0.27272727 0.25151515
##        153        154        155        156        157        158
## 0.34242424 0.31818182 0.39393939 0.24646465 0.37777778 0.23737374
##        159        160        161        162        163        164
## 0.37979798 0.36767677 0.41010101 0.41111111 0.35454545 0.32222222
# Note that the vector names (taken from gr.list names) are ego IDs.

# If you want the same result as a nice data frame with ego IDs, use enframe().
purrr::map_dbl(gr.list, ~ edge_density(.x)) %>%
        enframe()

## # A tibble: 102 x 2
##      name   value
##      <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
## # ... with 92 more rows

# Same thing, with number of components in each ego network.
purrr::map_dbl(gr.list, ~ 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
## # ... with 92 more rows
```

```
# With map_dfr() we can calculate multiple structural measures at once on each
# ego-network, and return the results as data frame.
# This assumes that we are applying an operation that returns one data frame row
# for one 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)),
       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.
purrr::map_dfr(gr.list, ~ tibble(dens= edge_density(.x),
                                 mean.deg= mean(igraph::degree(.x)),
                                 mean.bet= mean(igraph::betweenness(.x)),
                                 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      24.5        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     22.2        0.481
## 5 39        0.0929    4.09      7.18       0.316
## 6 40        0.258    11.3      23.0        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      11.7        0.372
## 10 48       0.203     8.93     33.8        0.365
## # ... with 92 more rows

# ***** EXERCISE (1):
# Get the graph for ego ID 28 from gr.list. Calculate the max betweenness of
# Italian alters on this graph. Based on this code, use ldply() to calculate
# that for all egos, and put the results into a data frame.
# *****
```

```
# ***** EXERCISE (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).
# *****
```

REFERENCIAS

Wickham, Hadley (2011). "The Split-Apply-Combine Strategy for Data Analysis." *Journal of Statistical Software* 40(1):1–29.

Remitido: 07-01-2021

Aceptado: XX-XX-2021